University Institute of Lisbon

Departamento de Ciências e Tecnologias da Informação

An Eclipse Plug-in for Metamodel Driven Measurement

Pedro Janeiro Coimbra

A Dissertation presented in partial fulfilment of the Requirements for the Degree of

Master in Computer Science and Business Management

Supervisor:
Fernando Brito e Abreu, PhD, Associate Professor, DCTI/ISCTE-IUL

September, 2013

[This page was intentionally left blank]

Abstract

In this dissertation, we present a new plug-in for the Eclipse integrated development environment that calculates software quality metrics using a metamodel driven approach to software measurement.

Metamodel driven measurement is a technique that surged with the popularization of object-oriented systems and model-driven approaches to software design. It involves of instantiating software designs according to a language metamodel and calculating metrics with formalized queries over the obtained data.

Our objectives consisted of creating a new Eclipse plug-in to analyse software developed in Java that, thanks to the metamodel driven approach, would allow users to easily define new software metrics without having to change a single line of Java code. To achieve our goals, we devised the Eclipse Java Metamodel, a new Java metamodel based on data provided by Eclipse's Java Development Tools and implemented it on a prototype Eclipse plug-in. We have also formalized certain software metrics and an existing library for metrics extraction called FLAME, as sets of queries over our developed metamodel using the Object Constraint Language, which can be used directly on our prototype.

Keywords: *Software Engineering; Software Measurement; Software Metrics; Metamodel Driven Measurement; Java; Java Model; Eclipse IDE; Abstract Syntax Tree*

Resumo

Nesta dissertação apresentamos uma nova extensão para o ambiente de desenvolvimento integrado *Eclipse* para o cálculo de métricas de qualidade de *software* através da medição por metamodelos.

Medição por metamodelos é uma abordagem à medição de *software* que surgiu com a popularização de sistemas orientados por objectos e *design* de *software* através de modelos. Esta técnica consiste em medir *software* através de definições formalizadas de métricas como *queries* sobre um metamodelo representativo da linguagem sobre a qual o *software* foi conceptualizado ou desenvolvido.

Os nossos objectivos consistem em criar uma nova extensão para *Eclipse* para analisar *software* desenvolvido em Java que, graças ao uso de metamodelos, permite a utilizadores calcular novas métricas de *software* facilmente sem ter que programar uma única linha de código em Java. Para concretizar estes objectivos, desenvolvemos o *Eclipse Java Metamodel*, um metamodelo da linguagem Java baseado nas *Java Development Tools* oferecidas pelo *Eclipse* e implementámos uma extensão protótipo. Também formalizámos certas métricas de *software* e uma biblioteca existente para o auxílio de cálculo de métricas chamada FLAME, como conjuntos de *queries* sobre o metamodelo feitas com a *Object Constraint Language*, que podem ser usadas directamente no nosso protótipo.

Palavras-chave: *Engenharia de Software; Medição de Software; Métricas de Software; Medição por Metamodelos; Java; Java Model; Eclipse IDE; Abstract Syntax Tree*

Acknowledgements

Coming as a surprise to no one, work on this dissertation has been toiling, stressful and exhaustive over my youthful and inexperienced self. And like many others, I feel like my success was due to the help of those around me.

Firstly, I thank my supervisor. Throughout this year, often have I found myself lost or uncertain of the quality of my work and just as often have I relied on his direction and assurance. I relied, perhaps too many times, on his knowledge of Software Engineering and abilities as a researcher to aid my work.

Secondly, I must thank my family and closest friends, for their constant support and understanding, during times I could give nothing in return.

Thirdly, I thank my MIG classmates, invaluable friends who egged me on and offered me precious advice to help my work.

Finally, I must thank the marvels of modern digital entertainment. As the anxieties of work creeped my mind ever closer to the fringes of sanity, occasional oases of relaxation allowed to keep it focused and content.

[This page was intentionally left blank]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Software quality has long been a subject of much research in the area of software engineering. From the need of applying quantitative research over software, a new field of study, of software metrics and measurement, arose.

Over the last four decades, this research on software measurement resulted in not only software metrics such as McCabe's cyclomatic complexity [1], the Chidamber and Kemerer metrics [2] and Abreu's MOOD2 set [3] for object-oriented systems, that have been applied to asses software quality characteristics such as a system's overall reliability [4], but also several measurement techniques.

With the popularization of object-oriented software and model-driven engineering (or MDE, for short) for software development, several approaches to measurement equally took a model-driven perspective. Prevalent especially in the last decade and a half, several studies detailing model-driven approaches to software measurement and respective tools applying such techniques have been presented. Within these studies, the term "meta-model", or "metamodel", was often found [3, 5, 6, 7, 8], referring to the model defining the structure of the language in which a system is constructed, as opposed to the structure of the system itself - which in turn can be viewed as the "instantiation" of the metamodel. The term "metamodel-driven measurement", commonly and hereinafter referred to as M2DM, was coined by Abreu [3] for the specific act of using metamodels to describe both the systems to be measured and the metrics themselves. Several other studies, using techniques similar or equivalent to M2DM were also proposed, but often used other naming. Authors often claimed several advantages for these metamodel-driven approaches, such as achieving easier and more exact metrics formalizations [3, 6], providing easier opportunities to measure systems constructed with different but similar languages or add new metrics to measurement tools [5], or simply to take away the complex code or compiler perspective into a more readable model-oriented logic [7]. These advantages were achieved not only because of the construction of formal metamodels, but also because of the use of specific languages that can query the said metamodels, such as the Structured Query Language, SQL [5], XQuery [9] or, more commonly, the Object Constraint Language, OCL [3, 6, 10, 7, 8]. The full findings of M2DM and such

similar studies can be found in section 2.2 of the following chapter.

Unfortunately, despite several studies detailing accompanying tools for software measurement, few have become or stayed available online at the time of this writing. In fact, we recognized a certain vacuum regarding existing MD2M tools. From this vacuum, we identified the need for a new research effort in this area.

According to the "Transparent Language Popularity Index" available online and permanently updated [11], Java is the most widely used object-oriented programming language with circa 20% of the market share of all programming languages. Meanwhile, Eclipse [12] is the second most widely used IDE globally and by far the first used by open source communities. With the popularity of Eclipse, several plug-ins capable of metrics extraction have been developed and made available in the Eclipse Marketplace [13], making use of the IDE's own project and code interpreting tools - the Eclipse Java Development Tools [14] (or JDT, for short). These plug-ins offer extensive measuring and reporting capabilities, but none have yet implemented the M2DM approach to metrics definition, and the advantages that come with it. Yet, the tools the JDT offer can be easily be utilized to aid our own research - and two of them in particular: the Java Model (from here on referred to as EJM) and the Abstract Syntax Tree (commonly referred to as AST).

We started the development of a M2DM plug-in for Eclipse, borrowing from the JDT to create a new Java metamodel and project-crawling mechanisms to instantiate it with data representative of a Java project to analyse. In the future, we aim to implement our developed M2DM tools for Java into existing plug-ins, making use of their established interface and reporting capabilities - namely, the popular open-source Eclipse metrics plug-in by Frank Sauer [15].

The contributions of this dissertation consist of the work completed to this date of this M2DM project: the resulting Java meta-model created from JDT components, the method of transforming a Java project in Eclipse into a meta-model instance and the first functioning metrics plug-in prototype capable of M2DM functionalities.

This document has been organized as follows: Chapter 2 explores the current situation regarding M2DM and MDE-based metrics research, as well as provides an overview of existing Eclipse plug-ins capable of metrics extraction and an introduction to the tools used to aid the development of our goal plug-in, including the JDT itself. In chapter 3, we present the Eclipse Java Metamodel (hereinafter referred to as EJMM) and detail several aspects of its structure, components and origins. Then, in chapter 4, we provide examples of the use of the developed EJMM with a series of OCL functions that traverse the metamodel to extract metrics or aid metrics extraction. Next, in chapter 5, we provide an overview of the developed

prototype. We describe its architecture, the chosen rules pertaining to the EJMM instantiation and the validation methods for the tool. Finally, we draw some conclusions on chapter 6 and forecast future work.

1.1 Regarding terminology

Since this dissertation aims to create a representation of the Java language as described by the existing JDT, there will be several cases of having different concepts with the same term in the following text. For instance, a Java statement in code is represented by the *Statement* class of the JDT which in turn is reflected by a *Statement* metaclass in the EJMM. The difference is that a statement, non-capitalized and non-italicized, refers to a generic occurrence of a statement in code (and much like statements, the same can be said of packages, types, methods, etc.) as opposed to its abstract representation as a Java class in the JDT or as a metaclass in the EJMM, in which case they are capitalized in the same way they are declared in code or in the metamodel and italicized. Still, JDT component names and EJMM component names may conflict, in which case the concepts of ”class” - or in some cases, ”interface” - and ”metaclass” will help to differentiate what the text refers to. For instance, a ”metaclass” refers to a class of the metamodel, a class to represent all classes. Thus, components of the JDT are specific Java classes or interfaces and are called as such when mentioned in text to aid comprehension. In some cases, JDT components are accompanied by the package in which they can be found.

[This page was intentionally left blank]

2. RELATED WORK

2.1 Introduction

Development of model-driven metrics tools has been the subject of much research for over a decade. In this chapter we will describe related work on tools that have some similarities to our proposed metrics plug-in, while highlighting the key differences between them. Then, we will review the state of the art regarding existing Eclipse metrics plug-ins that can be found online. Finally, we will present a brief overview of the tools chosen to help create a M2DM plug-in, such as the Eclipse Java Development Tools, and the UML-based Specification Environment.

2.2 Studies in M2DM and MDE-based measurement

M2DM studies originated as a natural progression from previous studies on software metrics formalization. Abreu first employed M2DM techniques to formalize the MOOD2 metrics over the GOODLY design language metamodel [3] using OCL. Since then, the use of OCL for M2DM has been popularized within the QUASAR research group. First efforts consisted of formalizing MOOD2 [6, 16] and MOOSE [10] metrics extraction for the UML metamodel, but over the following years, M2DM practices were used for more specific goals. This includes studies such as a means to formalize quality metrics for object-relational database schemas using OCL over SQL ontologies [17], to introduce quantitative approaches to component-based design by formalizing metrics for component-based systems over extended versions of the CORBA Components Metamodel and UML metamodel [18], to define OCL metrics over the System Definition Model for IT infrastructure evaluation [19], to compare modularity between aspect-oriented and object-oriented designs [20] over the QUASAR-developed PIMETA metamodel and to define complexity metrics for process models over the BPMN metamodel [21].

Regarding studies on for metrics calculation tools, Java Metrics Reporter (JMR) [23] was a pioneer software in its use of a Java Model to calculate metrics. In their paper, the

authors present a newly-developed Java Model much akin to the EJM supplied by the Eclipse JDT, sharing even a similar hierarchical structure and the existence of a single node class from which all Java elements inherit (in this case, called *JElement*). Extensions to the Java Model and implementation of further metrics calculation capabilities would be done through typical OO subclassing of the JMR Java Model. Though this approach is stated to be much simpler to handling directly with parser logic, it would still entail more programming (and building) work than by using the OCL-based metrics paradigm [3, 16, 10, 7]. This technique of handling a Java Model for data extraction is similar to what is commonly found in existing Eclipse metrics plug-ins, such as Frank Sauer's metrics tool [15], using EJM-supplied data. The JMR would be a considerable alternative to the Eclipse Java Development Tools for the construction of a M2DM tool, but unfortunately, at the time of writing, we have not been able to find it available online.

The Extensible Metrics toolBench for Empirical Research (EMBER) was introduced in [5]. Unlike the JMR, it used a metrics approach much more similar to the OCL-based paradigm. It used a database schema to represent a metamodel aimed at OO languages such as C++ and Java, parsed software to load the database and SQL queries were used to calculate metrics. This method is analogous to our own attempt at metamodel representation through UML, loading through the USE tool and then using OCL statements to calculate metrics. However, since the database schema aims to be fitting for several OO languages, it is much simpler than the EJMM proposed in this article. This simplicity certainly allows the tool to have a larger breadth in terms of languages supported, but does not offer the detail that our proposed EJMM provides for Java. One example is considering annotations used and type parameters. Another, more metrics-focused example is recording the statements of a method, allowing, for instance, to calculate the weighted methods per class [2] using McCabe's cyclomatic complexity metric [1] to determine the complexity of each method.

The Jade Bird Object Oriented Metrics Tool (JBOOMT) [24] also takes a model-based approach to calculate metrics of C++ software. In this case, however, the MDE approach involves creating metrics models rather than using metamodels to represent the software's model language and extracting data from their instantiation. JBOOMT thus requires a different model for each metric, storing software data and metrics calculations. This approach allowed their authors to create hierarchies among metrics, such as aggregated metrics or association between related measures. For instance, having a method's cyclomatic complexity [1] being a factor to determine the software's overall complexity. To further illustrate such relations between metrics, authors introduce the concepts of "internal attributes" like the method's complexity and "external attributes" as software-wide measures such as complex-

ity or flexibility.

Antoniol et al. [7] proposed a tool that navigates Java AST objects using OCL expressions. The authors use a metamodel based on the JavaCC compiling rules and exemplify its functionalities by formalizing a small set of software metrics. All in all, the purpose and method of their tool is essentially the same as ours, though using a different metamodel with different capabilities. Their presented AST metamodel retains a node tree structure, whereas our current EJMM aims to capture more directly a Java project's structure (despite internally, the EJM and Eclipse AST also having similar node tree structures). Furthermore, since the metamodel presented by Antoniol et al. comprises only the AST, it can only represent the code of a software system; from the class declaration to its contents, leaving out the software's overall structure and composition, such as folders and packages. With the lack of visibility of interactions between modules, structural analysis might become difficult.

Still, the work of Antoniol et al. is clearly a precursor to our own, and the use of OCL reveals another advantage of the inherent portability, as the metrics defined in these authors' paper can easily be applied to our EJMM with little effort.

The Design-Metrics Crawler [9] was a proposed tool to measure software designs specified in XMI files. Aiming to support UML diagrams and MOF-based software design languages, the DM Crawler used the XQuery language to formalize and calculate metrics. Unlike a language meant to define constraints such as OCL, the authors claim that XQuery's queries would be more appropriate for execution and would allow the definition of more complex metrics more easily since it is possible to define variables and control flow. While the first version of OCL had some limitations regarding those aspects, OCL2 has removed those limitations. We found no evidence that this tool could be used with Java.

The Metrino tool [25] is a metrics tool that aimed to measure any software devised in any Domain Specific Language with metamodels based on the Meta Object Facility, using OCL for the definition of domain rules. OCL is also used for the definition and calculation of metrics, under OMG's Software Metrics Meta-model. Currently, it is available online and affiliated with the ModelBus tool [26]. Their authors claim that Metrino can be used for UML models, as well as for any Domain Specific Modeling Language (DSL) based on MOF, but not for Java.

Finally, McQuillan [8], also aiming to achieve a MDE approach to software measurement, created a MOF-compliant metamodel for metrics extraction for Java and UML models. Metrics formalization took the form of OCL expressions over the metrics metamodel. To test and implement this technique, the author claims to have developed the Defining Metrics at the Meta-Level (dMML) tool, capable of measuring Java programs. Unfortunately this seems to

have been a research prototype only, for academic purposes, since it is not available online for evaluation.

2.3 Existing Eclipse metrics plug-ins

Currently, there are several plug-ins for the Eclipse IDE that support software metrics extraction. In this section, we will present a few existing plug-ins that can be found on the Eclipse Marketplace [13] and the Yoxos marketplace [27], as well as Frank Sauer's [15] plug-in. Using the sites' respective search engines, we used the keyword "metrics" and selected plug-ins that offered software metrics extraction capabilities. Our main interests when testing these plug-ins were which metrics they supported and their extension capabilities. Although the open-source solutions provided a way to extend their metrics calculation capabilities by changing their source code, we found that no tool provided a front-end solution to incorporate new software metrics.

2.3.1 Open-source plug-ins

Name:	Eclipse Metrics plugin 1.3.6
By:	Frank Sauer et al.
Project site:	`http://metrics.sourceforge.net/`
License:	CPL
Last update:	2005-07-08 (plug-in); 2013-04-24 (site)
Downloads:	99.533 total from last plug-in update until 2013-09-05; 11.728 total from 2012-09-05 to 2013-09-05; 774 total from 2013-08-05 to 2013-09-05
Description:	Frank Sauer's plug-in, though having nearly a decade of age, is still one of the more proven and popular options regarding open-source metrics plug-ins for Eclipse. Though there is a separate continued version [28], Sauer's still has a fairly active community, going so far as to having community-created patches as recent as November 2012. The plug-in itself offers a simple interface and reporting capabilities with which users can define optimal ranges and issue warnings for certain metrics, as well as being able to export calculated metrics to XML files. Being one of the most popular, this plug-in was our first choice for a target in which to implement M2DM features.
Metrics supported:	Abstractness; Afferent Coupling; Cyclomatic Complexity; Depth of Inheritance Tree; Efferent Coupling; Instability; Lack of Cohesion of Methods; Method Lines of Code; Nested Block Depth; Normalized Distance; Number of Attributes; Number of Children; Number of Classes; Number of Interfaces; Number of Methods; Number of Overridden Methods; Number of Packages; Number of Parameters; Number of Static Attributes; Number of Static Methods; Specialization Index; Total Lines of Code; Weighted Methods per Class;
Extension capabilities:	Requires the editing of the plugin.xml file to declare new metrics (name, abbreviation, level and propagation settings and calculator) and extension of a *Calculator* class to define how the metrics are calculated.

Name:	Eclipse Metrics plugin (Continued) 1.3.8
By:	Guillaume Boissier et al.
Marketplace site:	`http://marketplace.eclipse.org/content/eclipse-metrics-plugin-continued`
Project site:	`http://metrics2.sourceforge.net/`
License:	CPL
Last update:	2010-07-30 (plug-in); 2013-04-24 (site)
Downloads:	6.185 total from last plug-in update until 2013-09-05; 2.598 total from 2012-09-05 to 2013-09-05; 216 total from 2013-08-05 to 2013-09-05
Marketplace installs:	0 from 2012-10 to 2013-09
Description:	Continuation from Frank Sauer's plug-in led by Guillaume Boissier. Though from a user perspective, the plug-in functions very much the same, its source code has been fairly revamped and improved, including in the way metrics are calculated. Though it is an update to its widespread predecessor, it does not enjoy as much popularity, and in result, visibility. As far as targets for M2DM implementation, Boissier's version has become as much of an alternative as Sauer's and though our current choice has been unaltered, it might easily change if any issues surface during the time of implementation.
Metrics supported:	Abstractness; Afferent Coupling; Cyclomatic Complexity; Depth of Inheritance Tree; Efferent Coupling; Instability; Lack of Cohesion of Methods; Method Lines of Code; Nested Block Depth; Normalized Distance; Number of Attributes; Number of Children; Number of Classes; Number of Interfaces; Number of Methods; Number of Overridden Methods; Number of Packages; Number of Parameters; Number of Static Attributes; Number of Static Methods; Specialization Index; Total Lines of Code; Weighted Methods per Class;
Extension capabilities:	Extension to calculate new metrics is done the same way as in version 1.3.6.

Name:	Eclipse Metrics 3.12.0
By:	Lance Walton/State of Flow
Marketplace site:	`http://marketplace.eclipse.org/content/eclipse-metrics`
Project site:	`http://eclipse-metrics.sourceforge.net/`
License:	CPL
Last update:	2011-04-10 (plug-in); 2013-05-02 (site)
Downloads:	8.313 total from last plug-in update until 2013-09-05; 3.891 total from 2012-09-05 to 2013-09-05; 345 total from 2013-08-05 to 2013-09-05
Marketplace installs:	5.264 from 2012-10 to 2013-09
Description:	This plug-in offers more extensive exporting capabilities compared to Frank Sauer's, including XML, HTML and CSV exporting, as well as integration with Eclipse's Problems view to issue warnings regarding certain metrics. However, though open-source, extending it to support more metrics is a more difficult task, due to its lack of documentation on the matter.
Metrics supported:	Cyclomatic Complexity; Efferent Couplings; Feature Envy; Lack of Cohesion in Methods (Chidamber & Kemerer, Henderson-Sellers, Pairwise Field Irrelation and Total Correlation); Lines of Code in Method; Number of Classes; Number of Fields; Number of Levels; Number of Locals in Scope; Number of Packages; Number of Parameters; Number of Statements; Total Lines of Code; Weighted Methods Per Class;
Extension capabilities:	Requires the creation of new calculator classes. Current plug-in contains several abstract calculator classes for specific data extraction tasks (for instance, inspecting an AST).

Name:	Project Usus 0.7.2
By:	Stefan Schuerle et al.
Marketplace site:	`http://marketplace.eclipse.org/content/project-usus`
Project site:	`http://www.projectusus.org`; `http://github.com/usus/usus-plugins` (source code only)
License:	EPL
Last update:	2013-01-27
Downloads:	35 total from plug-in release until 2013-09-06
Marketplace installs:	222 from 2012-10 to 2013-09
Description:	Project Usus is a set of Eclipse plug-ins to facilitate project monitoring, calculating metrics, identifying "hotspots" that deserve more attention and creating graphs for module dependencies, among other functionalities.
Metrics supported:	Average Component Dependency; Class Size; Cyclomatic Complexity; Lack of Cohesion of Classes; Method Length; Mudholes; Number of Fields (non-static, non-final and public) ; Package Size; Packages with Cyclic Dependencies; Unreferenced Classes;
Extension capabilities:	Features an extension collector to join user-created extension points to the Project Usus environment. Includes abstract classes and interfaces to structure how new calculators present their metrics data.

Name:	Java Metrics 0.9.4
By:	Devon Carew
Marketplace site:	`http://yoxos.eclipsesource.com/yoxos/node/org.dcarew.javancss.feature.feature.group`
Project site:	`http://code.google.com/p/eclipse-plugin-potpourri/`
License:	Apache License 2.0
Last update:	2010-06-22
Downloads:	N/A
Description:	As of now, this is a simple plug-in that calculates a small selection of metrics on a project level.
Metrics supported:	Cyclomatic Complexity; Non-Commenting Source Statements; Number of Files; Number of Packages; Total File Size
Extension capabilities:	N/A

2.3.2 Commercial plug-ins

Name:	inCode Helium 2.0.1
By:	Intooitus
Marketplace site:	`http://marketplace.eclipse.org/content/incode-helium`
Project site:	`http://www.intooitus.com/products/incode`
License:	Commercial (Free trial available)
Description:	Plug-in version of inFusion (see subsection 2.3.3). Although reduced in functionality compared to its stand-alone version, it still retains several of its features, including metrics calculation. Overall, inCode Helium is a tool with extensive graphical capabilities to aid detection of design flaws in code.
Metrics supported:	Average Function Weight; Access to Local Data; Average Method Weight; Access to Foreign Data; Base-class Overriding Ratio; Base-class Usage Ratio; Capsules Providing Foreign Data; Class Weight; Cyclomatic Number; Depth of Inheritance Tree; Dispersion Ratio; Fan-In; Fan-Out; Foreign Data Providers; Height of Inheritance Tree; Incoming Coupling Dispersion for an Operation; Incoming Coupling Dispersion for an Operation; Locality of Data Accesses; Loose Capsule Cohesion; Lines of Code; Lines of Comments; Maximum Nesting Level; Number of Added Services; Number of Attributes; Number of Abstract Classes; Number of Accessor Methods; Number of Abstract Methods; Number of Accessed Variables; Number of Classes; Number of Children; Number of Global Functions; Number of Global Variables; Number of Incoming Calls; Number of Local Variables; Number of Methods; Number of Modules; Number of Outgoing Calls; Number of Packages; Number of Parameters; Number of Protected Attributes; Number of Protected Attributes; Number of Public Attributes; Number of Public Methods; Number of Overriding Methods; Outgoing Coupling Dispersion for an Operation; Outgoing Coupling Intensity for an Operation; Outgoing Dependency on Delegators; Percentage of Newly Added Services; Tight Capsule Cohesion; Weighted Operation Count;

Name:	STAN - Structure Analysis for Java 2.1.1
By:	Odysseus Software GmbH
Marketplace site:	`http://marketplace.eclipse.org/content/stan-structure-analysis-java`
Project site:	`http://stan4j.com/`
License:	Free for non-commercial use
Last update:	2013-06-26
Downloads:	N/A
Marketplace installs:	597 from 2012-10 to 2013-09
Description:	STAN is a general tool for analysing a project's structure, showing module composition and couplings, as well as calculating distances and pollution. Furthermore, it has a dedicated metrics view showing several different software metrics. STAN is available as both an Eclipse plug-in or as a stand-alone application.
Metrics supported:	Abstractness; Afferent Coupling; Average Absolute Distance; Average Component Dependency between Libraries, between Packages and between Units; Coupling between Objects; Cyclomatic Complexity; Depth of Inheritance Tree; Distance; Efferent Coupling; Estimated Lines of Code; Fat for Library Dependencies, Package Dependencies and Class Dependencies; Fat; Instability; Lack of Cohesion in Methods; Number of Children; Number of Fields; Number of Instructions; Number of Libraries; Number of Member Classes; Number of Methods; Number of Packages; Number of Top Level Classes (Units); Response for a Class; Tangled for Library Dependencies and Package Dependencies; Tangled; Weighted Methods per Class;

2.3.3 Other

The following tools have not been thoroughly tested due to assorted difficulties.

Name:	inFusion Hydrogen 1.7
By:	Intooitus
Marketplace site:	`http://marketplace.eclipse.org/content/infusion-hydrogen`
Project site:	`http://www.intooitus.com/products/infusion`
License:	Commercial (Free trial available)
Description:	inFusion Hydrogen is actually not a plug-in for Eclipse, but instead a stand-alone application using the Eclipse framework and with its own page in the Eclipse Marketplace.
Notes:	Due to compatibility issues, we have not been able to test this application in time for publication.

Name:	Scertify™Professional 1.10
By:	Tocea
Marketplace site:	`http://marketplace.eclipse.org/node/626557`
Project site:	`http://tocea.com/shop/professional-edition/scertify-professional-edition`
License:	Commercial (Free trial available)
Notes:	Due to incompatibility issues, we have unfortunately not been able to test this tool in time for publication.

Name:	IntoJ Suite
By:	intoJ Team, University of Hagen
Marketplace site:	`http://yoxos.eclipsesource.com/yoxos/node/org.intoJ.suite.feature.feature.group`
Project site:	`http://wiki.fernuni-hagen.de/intoj/index.php/Hauptseite`
License:	CPL
Notes:	Since the IntoJ Suite wasn't available for installation in time for publication, we have unfortunately not been able to test its metrics calculation capabilities.

2.4 The Eclipse JDT

The Eclipse Java Development Tools, or JDT [14], are a series of plug-ins for Eclipse that provide developers access to several functions commonly associated with an IDE, such as code interpretation and Java project building. Several of the previously mentioned metrics

plug-ins make use of the JDT to extract the data required to calculate software metrics. In our M2DM plug-in, the usefulness of the JDT is two-fold: first, it provides a series of components from which to base the construction of a Java metamodel, and second, it provides ways to turn Java projects into abstract data as instances of the metamodel.

From the JDT, two components were of special interest to the construction of the EJMM: the Java Model (EJM) and the Java Abstract Syntax Tree (AST).

The EJM, defined by a series of interfaces located in the `org.eclipse.jdt.core`, offers a representation of a Java project and all its components, such as its folders, packages, compilation units, etc. Though it also provides a vision of a compilation unit's or class file's contents, such as its enclosed types and the type's members, it does not offer a full view of the Java code contained.

The AST, on the other hand, provides tools to analyse a block of source code and construct an abstract representation of it. It is contained in the `org.eclipse.jdt.core.dom` package as a series of classes representing different types of statements supported by the Java language (e.g. type declaration statement, field declaration statement, method invocation or comments in code).

Both the AST and the EJM have tree structures where nodes are connected to each other by hierarchical, parent-child, links. For the AST, all classes inherit from the *ASTNode* class. As for the EJM, all the interfaces inherit from the base *IJavaElement* interface. This means that traversing their trees can be done in a top-down approach, beginning from a single node that is parent to all other nodes.

In sum, the great advantage of using both the AST and the EJM in conjunction is the fact that they provide complementary views of a system. Whereas the EJM can present the structure and hierarchy of files of a Java project, the AST provides the structure and contents of the actual code. Thus, they both were our source of inspiration while designing the EJMM. However, it should be noted that the EJMM, although including representative metaclasses for *ASTNode* and *IJavaElement*, is not meant to inherit the tree structure of the AST and EJM functionally. Therefore, there are no generic parent-child connections in the EJMM, just unique meta-associations between specific metaclasses.

Specific information regarding how the EJM and AST were used to form the EJMM can be found in chapter 3 and further information on how the JDT was used by the M2DM plug-in to instantiate the EJMM can be found in chapter 5, section 5.4.

2.5 USE: The UML-based Specification Environment

Developed in the University of Bremen, the UML-based Specification Environment (USE, for short), is a Java, open-source tool that allows the creation of UML class models and instantiate them with objects to create a simulated system state [29] [30]. Adding to these functionalities, USE allows the use of the Object Constraint Language (OCL) [31] - a language specified by the Object Management Group (OMG) to complement UML diagrams with textual descriptions of a system's constraints - to declare both constraints of a model as well as use the language for querying model instances. Furthermore, USE also allows the definition of model operations using OCL as well.

This powerful solution was our choice to provide an environment where the EJMM could be declared, instantiated with concrete data and inspected. Being open-source and developed in Java, this means that our M2DM plug-in can also easily handle USE objects with little intermediation.

Despite the available graphical capabilities, USE class models must be declared textually. Embedded in the model components description, it is possible to define operation bodies and to declare OCL constraints (class invariants, as well as pre and post-conditions on operations). Furthermore, there is a possibility of incremental loading of model files, each being extended by the next. What this means is that although the EJMM could be defined within a single file, extensions could be made to add operations to respective metaclasses and expand their functionality. This would allow for users to be able to define their own metrics-calculating operations for specific metaclasses in separate files, load them in the M2DM tool and then query for their returned values. This simple method is the one that would substantiate the aforementioned advantages of M2DM regarding to the ease of adding new metrics.

To facilitate our integration of USE capabilities, we made use of a façade component interface named J-USE, produced within the QUASAR group [32], that provides a Java API for USE services.

[This page was intentionally left blank]

3. THE ECLIPSE JAVA METAMODEL

3.1 Introduction

The EJMM was obtained by reverse engineering and composing two Eclipse JDT components: the Eclipse Java Model (from now on referred to as EJM) and the Eclipse Abstract Syntax Tree (or AST for short). The EJM contains several interfaces that provide a vision over a Java project's structure under a tree architecture. The AST, on the other hand, deals with parsed source code. It allows the analysis of a source code file represented also as a tree, down to each statement and expression that compose the methods of a class [14, 33]. Although the EJM already provides a fairly complete vision of the software's structure (including, for instance, which classes are declared, as well as their fields and methods), the AST provides the minutia of a software application that can only be found within the code itself. The two components complement each other to create a highly detailed Java metamodel. However, the EJMM does not employ the EJM's and the AST's tree structures. Instead, we have opted for a simpler and more direct representation of a Java project, with meta-associations connecting specific metaclasses that would be otherwise represented as connected nodes of a tree. The node metaclasses remain, but purely as a means to generalize different metaclasses and to distribute the information they contain.

In this chapter, we will present only the most notable aspects regarding the EJMM specification. For the full EJMM specification, as a USE specification file, please refer to Appendix A.1.

3.2 Metamodel presentation

In this section, we will present the EJMM in its current form, as a set of class diagrams. Figure 3.1, represents the basic structure of a Java project and its hierarchical structure, including type inheritance and interface implementations. All the metaclasses present in Fig. 3.1 inherit, directly or indirectly, from the base *JavaElement* metaclass. The latter represents any node of a Java project tree, confers it a name (equal to the name declared in the source

code) and a unique identifier. Also included in this diagram are the three enumerations used by the metamodel: visibility types, Java types and package fragment root types. The first determines the visibility of an element, such as whether a method is public, private, protected or has default visibility (the default is the containing package and is applicable when the visibility keyword is omitted). The Java types enumeration serves to identify different kinds of a *Type* meta-object (Java class, interface, annotation type or enumeration). Package fragment roots can be folders or archives and the latter can be either .jar files or .zip files.

Figure 3.2 shows the contents of the *Type* metaclass, such as fields, methods, static initializers and type parameters. These components are referred to as members in the EJM and such is reflected by the abstract metaclass *Member* from which they inherit. The *LocalVariable* metaclass is a special case - since a local variable can only be located in one place, the meta-associations between it and either *Method* or *Initializer* are mutually exclusive. In terms of instantiation, this means that only one of these meta-associations would be defined. The attributes *arrayDimensions* and *returnTypeArrayDimensions* determine the number of array dimensions a *LocalVariable*, *Field* or a *Method*'s *returnType* has.

Figure 3.3 focuses on presenting another component of the EJM - annotations. These are attached to *Annotatable* meta-objects, such as local variable declarations, Java members or package declarations. Since we do not include package declarations in the EJMM, to represent package declaration annotations, we have instead made *CompilationUnit* an *Annotatable* metaclass, meta-associations between it and *Annotations* represent the package declaration annotations. Besides the occurrences of annotations in code, the metamodel also registers which class declares the annotation in question and the parameter values of each occurrence (currently under the form of strings as a generalized measure to deal with different field types) and the annotation class field that each value refers to.

In the final diagram (Fig. 3.4), the AST metaclasses are depicted. Inheriting from a base *ASTNode* metaclass, only comments and statements have been chosen to be included in the EJMM. Comments are directly linked to compilation units through a single meta-association. Statements can take several forms, but only *Block* statements are meta-associated to initializers or methods. Much like local variables, the *Block*'s meta-associations are mutually exclusive - it can only be associated with either an *Initializer* or a *Method*, or none at all.

All other statement types can only be found as aggregate parts of blocks, directly, or as part of other statements within the aggregation. Furthermore, there are extra meta-associations to represent dependencies between a *Statement* and a *Type* (*typeDependencies*), *Field* (*fieldsAccessed*) or *Method* (*methodsCalled*). The latter represents the types, fields and methods that are used or invoked and used by the expressions that compose the statement in question.

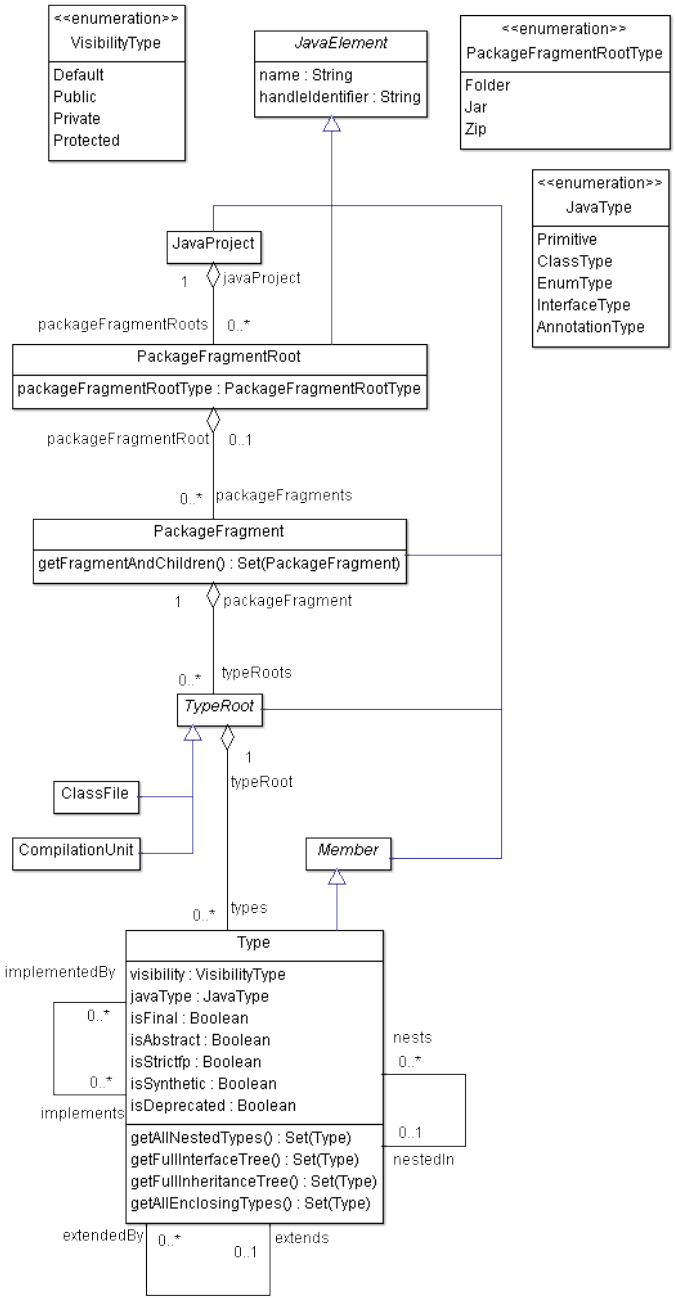


Fig. 3.1: Eclipse Java Metamodel - Java Project Structure

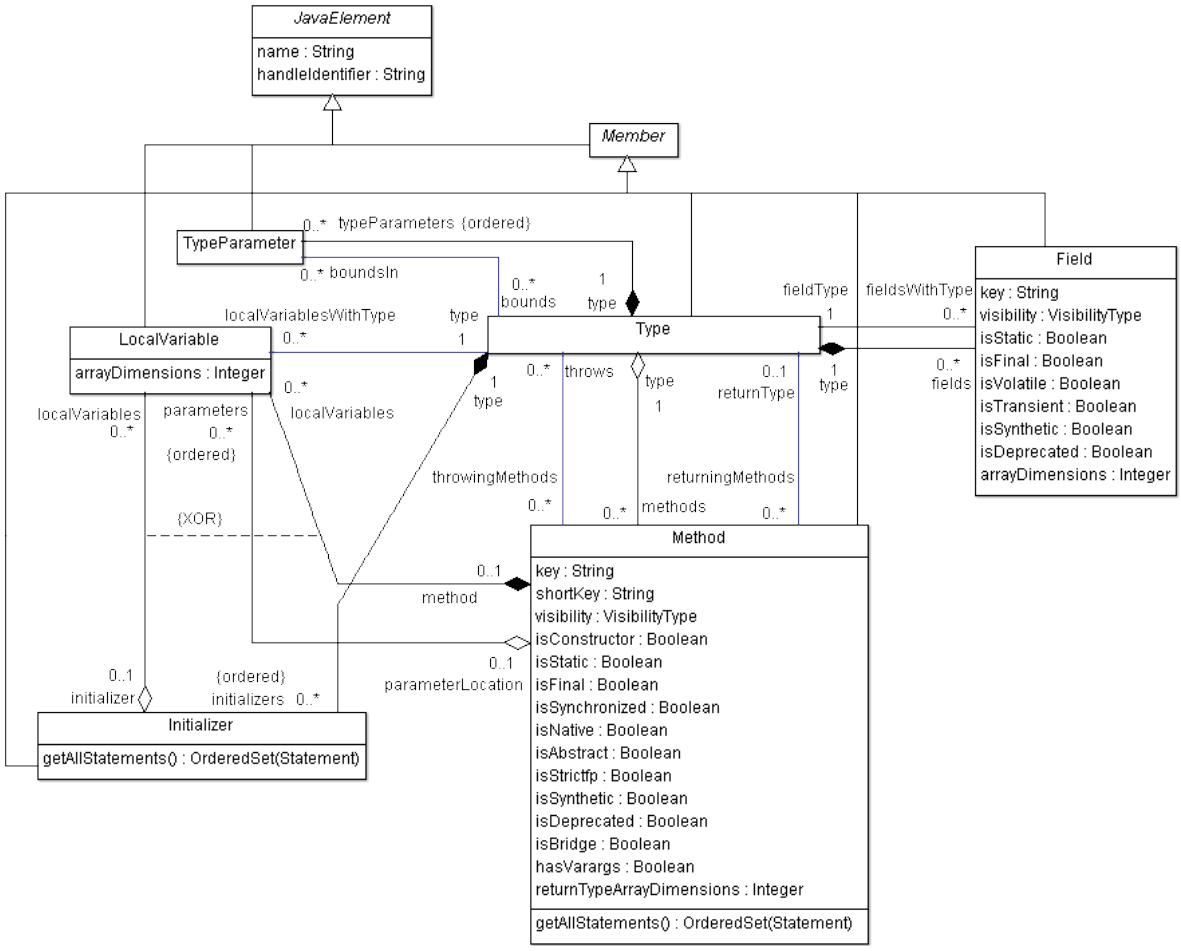


Fig. 3.2: Eclipse Java Metamodel - Type Components

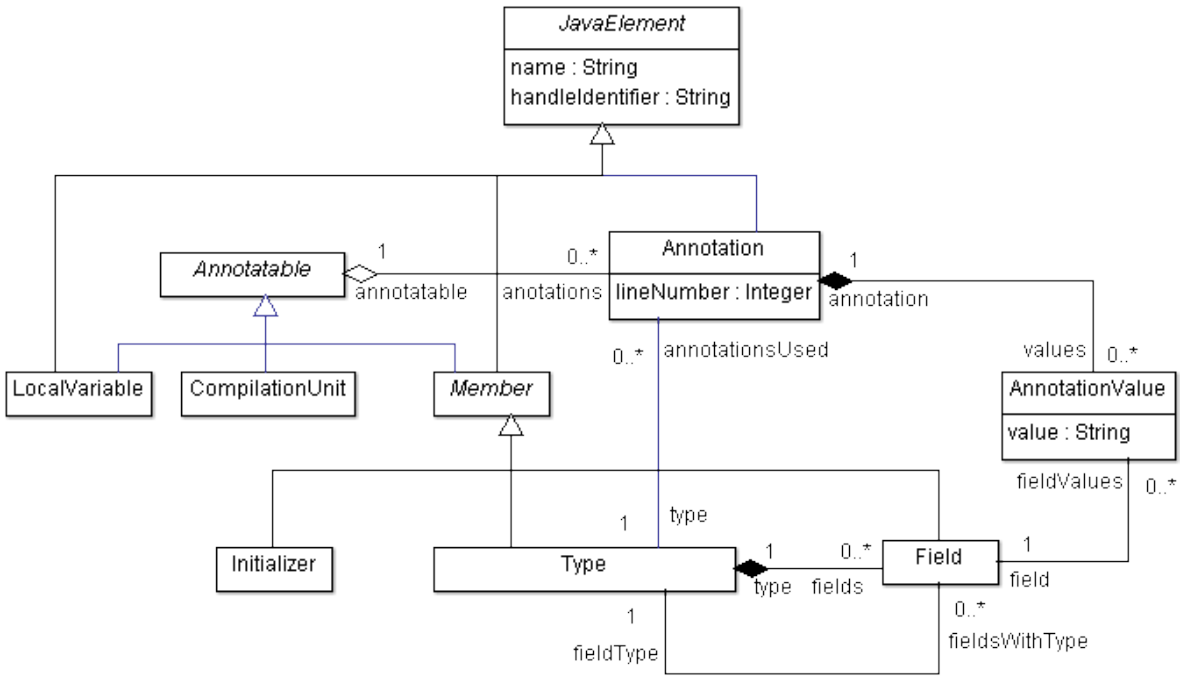


Fig. 3.3: Eclipse Java Metamodel - Annotations

The field *startPosition* of the *ASTNode* metaclass indicates the first character byte of the piece of code that a node represents within the entire block of code from which the AST was generated (thus, when analyzing the code of a compilation unit, the *startPosition* indicates the location of the node in the source file). The *length* attribute indicates the size of the piece of code in bytes. The *Statement* metaclass also contains a *conditionalOperatorCount* attribute. This attribute was specifically created for metrics extraction - it represents the number of conditional expressions contained in the statement plus the number of conditional operators used by the expressions contained in the statement (specifically, ”and” operators, ”or” operators and conditional expressions), to aid calculating a method’s cyclomatic complexity [1]. It was included in the EJMM and not as an extension due to the necessity of capturing new information during the moment the metamodel is instantiated.

Operations declared in the metamodel metaclasses are purely auxiliary, aiming to aid the definition of OCL statements. They are recursive operations that return all the hierarchical children or parents of a given meta-object (for instance all the statements contained in a block, or all the components of a type’s superclass hierarchy).

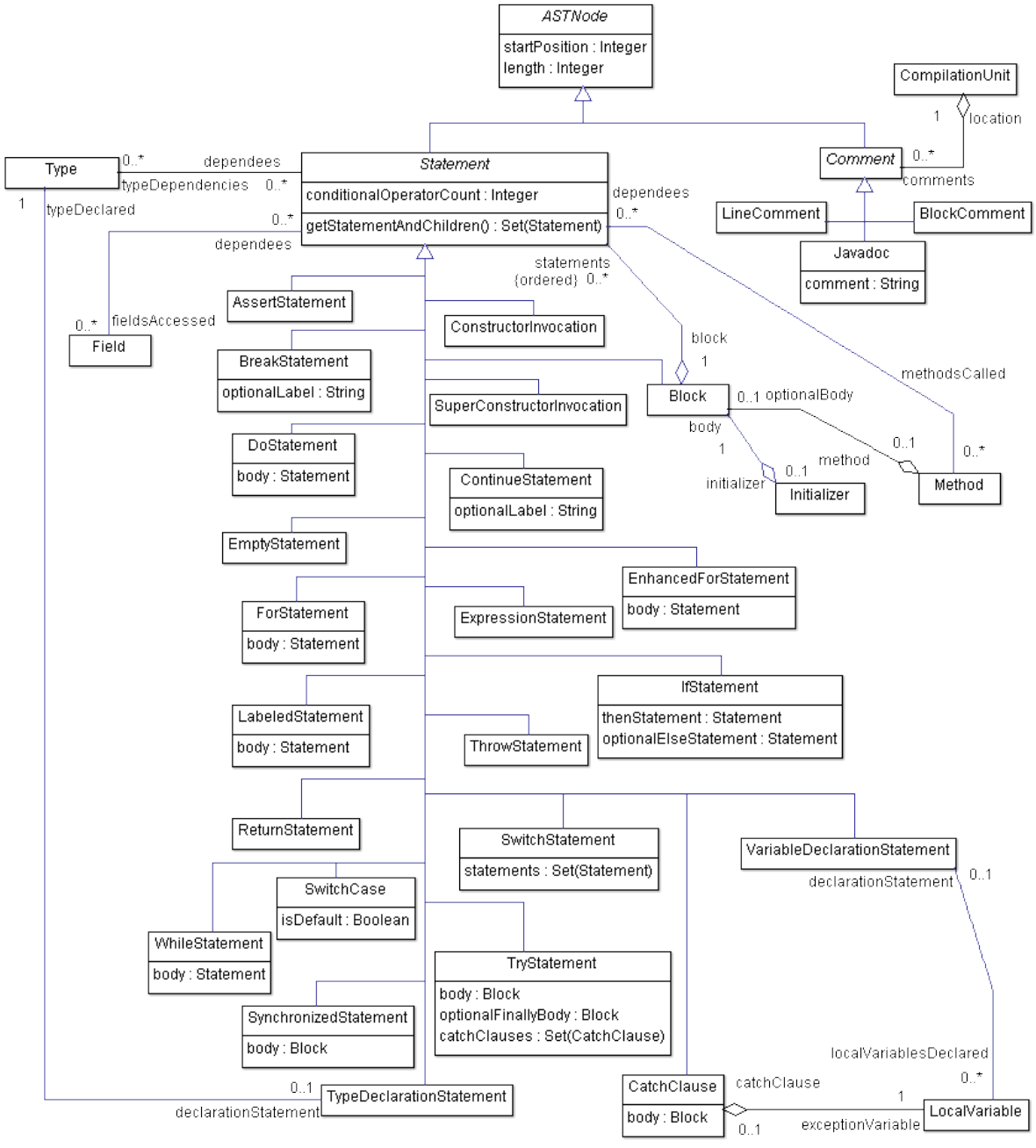


Fig. 3.4: Eclipse Java Metamodel - Abstract Syntax Tree Components

3.3 Reverse engineering information

As previously stated, the EJMM is the result of reverse engineering the EJM and AST provided by the Eclipse JDT. Metaclasses that inherit from the *JavaElement* metaclass have their origins in homonymous interfaces of the EJM package (`org.eclipse.jdt.core`). Metaclasses that inherit from the *ASTNode* metaclass have their origins in homonymous classes of the AST package (`org.eclipse.jdt.core.dom`). Thus, it is safe to say that metaclasses are representations of the EJM and AST interfaces and classes. The exception is the *AnnotationValue* metaclass, which is built from the values obtained through the *IAnnotation* interface. A complete overview of the metaclass origins can be found in the following section.

Meta-associations are representations of getters found in the metaclasses' origin. A special case was found in the relation between *PackageFragmentRoot* and *PackageFragment*, as the original *IPackageFragmentRoot* provides *IPackageFragments* through the generic Java model child getter, rather than a specific one for package fragments. For most statement types, fields have translated to attributes in their respective metaclass. The biggest exception is the *Block* metaclass, as its contents have been translated into a single aggregation of *Statement*.

Metaclass attributes are also derived from original attributes or getters. However, several attributes of the *Type*, *Method* and *Field* are derived from a special integer flag their original interfaces provide. The flags, as defined by the class *org.eclipse.jdt.core.Flags*, are used to differentiate a Java member's visibility (reflected on the enumeration *VisibilityType*) or different Java types (enumeration *JavaType*). Flags were also used to create the *isFinal*, *isSynthetic*, *isDeprecated*, *isStatic*, *isSynchronized*, *isNative*, *isBridge*, *hasVarargs*, *isVolatile*, *isTransient* and *isStrictfp* attributes. Furthermore, as mentioned previously, the *conditionalOperatorCount* is calculated at the moment of instantiation by counting the number of conditional "and" and "or" operators and conditional expressions contained inside a statement.

3.4 Metaclass presentation

Metaclass name:	*JavaElement*
Origin:	*org.eclipse.jdt.core.IJavaElement*
Description	The original interface is implemented by all EJM tree nodes and translates to an abstract class on the metamodel that is inherited, directly or indirectly, by all metaclasses that originate from the EJM. Unlike *IJavaElement*, the *JavaElement* metaclass is not used for building a tree structure, but instead to generalize data into meta-objects such as element name or handle. The *name* attribute is the one used to store the element name (generally from the *getElementName()* method). *handleIdentifier* is a special identifier string that provides the complete location of the element. No two different *JavaElements* have the same *handleIdentifier*.

Metaclass name:	*JavaProject*
Origin:	*org.eclipse.jdt.core.IJavaProject*
Description:	Represents the folder that holds the Java project.

Metaclass name:	*PackageFragmentRoot*
Origin:	*org.eclipse.jdt.core.IPackageFragmentRoot*
Description	Represents a folder within the Java project that may contain packages. Since packages can be contained in either folders or archive files, the metaclass includes a specific attribute that dictates the type of package root in question (see the description of the *PackageFragmentRootType* enumeration for further information). A *PackageFragmentRoot*'s *name* is actually the name of its respective resource (obtained from the *getResource() method*), such as the folder or archive name, since the *getElementName()* method for *IPackageFragment* instances returns empty strings.

Metaclass name:	*PackageFragment*
Origin:	*org.eclipse.jdt.core.IPackageFragment*
Description:	Represents a single package.

Metaclass name:	*Annotatable*
Origin:	*org.eclipse.jdt.core.IAnnotatable*
Description:	Abstract metaclass class that represents any object that can have annotations attached. Concrete metaclasses that may have annotations extend this metaclass.

Metaclass name:	*TypeRoot*
Origin:	*org.eclipse.jdt.core.ITypeRoot*
Description:	Represents a file in which a type can be declared.

Metaclass name:	*ClassFile*
Origin:	*org.eclipse.jdt.core.IPackageFragment*
Description:	Represents a binary class file containing a single type declaration in bytecode.

Metaclass name:	*CompilationUnit*
Origin:	*org.eclipse.jdt.core.ICompilationUnit*
Description:	Represents a source file that may contain several type declarations.

Metaclass name:	*Member*
Origin:	*org.eclipse.jdt.core.IMember*
Description:	*Members* represent any Java elements that are components of a Java type, such as methods, fields, the type itself, as well as inner types. Concrete members inherit from this metaclass.

Metaclass name:	*Type*
Origin:	*org.eclipse.jdt.core.IType*
Description:	Represents a single Java type. This may be a class, an interface, an annotation definition, an enumeration or a primitive type (such as integers and booleans). See the enumeration *JavaType* for more information.

Metaclass name:	*TypeParameter*
Origin:	*org.eclipse.jdt.core.ITypeParameter*
Description:	Represents a parameter for a single type. Not to be confused with type arguments, that are concrete instantiations of parameters. Parameter bounds are defined by a meta-association between parameter (*boundsIn*) and the bound type (*bounds*).

Metaclass name:	*Field*
Origin:	*org.eclipse.jdt.core.IField*
Description:	Represents a field declared within a type. The attribute *arrayDimensions* indicates the number of array dimensions this *Field* has (zero, if not an array).

Metaclass name:	*Initializer*
Origin:	*org.eclipse.jdt.core.IInitializer*
Description:	Represents a static initializer of a class. Initializers have no name, but within the same type, they can be identified by the order in which they are declared.

Metaclass name:	*Method*
Origin:	*org.eclipse.jdt.core.IMethod*
Description:	Represents a method declared in a type. It is linked to its body through a meta-aggregation with the *Block* metaclass unless it is an abstract method. The *shortKey* attribute contains only the method and parameter section of the method's full key, representing the method signature - essentially, a substring of *key*. The *shortKey* exists mainly to aid the comparison between two methods belonging to different types, since the *key* contains also the location of the method. The *hasVarargs* attribute determines whether or not the *Method* has a variable number arguments (notice that only the last parameter in a method's parameter list can be used multiply). The attribute *arrayDimensions* indicates the number of array dimensions this *Method*'s *returnType* has (zero, if the method doesn't return an array).

Metaclass name:	*LocalVariable*
Origin:	*org.eclipse.jdt.core.ILocalVariable*
Description:	Represents a single local variable contained in either a method body, an initializer body or a method parameter. Its meta-associations with *Method* or *Initializer* mutually exclusive - only one can be defined, denoting the location where the local variable is declared. The attribute *arrayDimensions* indicates the number of array dimensions this *LocalVariable* has (zero, if not an array).

Metaclass name:	*Annotation*
Origin:	*org.eclipse.jdt.core.IAnnotation*
Description:	Represents a single occurrence of an annotation within source code. If the aforesaid occurrence has annotation arguments, they are saved as *AnnotationValue* instances.

Metaclass name:	*AnnotationValue*
Origin:	*org.eclipse.jdt.core.IMemberValuePair*
Description:	Unlike other metaclasses derived from EJM components, *AnnotationValue* is not a direct conversion of an EJM interface. Instead, it is a metaclass created from data obtained from *IAnnotation*. Values are saved as strings and are meta-associated with the corresponding field of the annotation type.

Metaclass name:	*ASTNode*
Origin:	*org.eclipse.jdt.core.dom.ASTNode*
Description:	Base abstract node metaclass for AST nodes. Much like the *JavaElement* metaclass, the main purpose of *ASTNode* is to generalize data such as the starting byte of the node inside the source text (useful for ordering) and its length in bytes.

Metaclass name:	*Comment*
Origin:	*org.eclipse.jdt.core.dom.Comment*
Description:	Abstract metaclass representing a comment in the source code of a compilation unit.

Metaclass name:	*LineComment*
Origin:	*org.eclipse.jdt.core.dom.LineComment*
Description:	Concrete metaclass for single line comments.

Metaclass name:	*BlockComment*
Origin:	*org.eclipse.jdt.core.dom.BlockComment*
Description:	Concrete metaclass for single block comments.

Metaclass name:	*Javadoc*
Origin:	*org.eclipse.jdt.core.dom.Javadoc*
Description:	Concrete metaclass for Javadoc comment blocks.

Metaclass name:	*Statement*
Origin:	*org.eclipse.jdt.core.dom.Statement*
Description:	Abstract metaclass for Java statements. To generalize type references in concrete statements, we have included a *typeDependency* meta-association. To generalize method calls, we have the *methodsCalled* meta-association and to generalize access to a field, we have *fieldsAccessed*. In the cases of statement types that cannot depend on any type, method or field (for instance, *EmptyStatement*), the meta-associations are not instantiated. The auxiliary method *getStatementAndChildren()* is a recursive method that returns the statement and all statements included in it, directly or indirectly. The *conditionalOperatorCount* attribute represents the number of conditional expressions and conditional operators contained in the statement.

Metaclass name:	*AssertStatement*
Origin:	*org.eclipse.jdt.core.dom.AssertStatement*
Description:	Concrete *Statement* metaclass to represent ”assert” statements.

Metaclass name:	*Block*
Origin:	*org.eclipse.jdt.core.dom.Block*
Description:	Concrete statement type that represents a block of code which may contain more statements, defined by a meta-composition.

Metaclass name:	*BreakStatement*
Origin:	*org.eclipse.jdt.core.dom.BreakStatement*
Description:	Concrete *Statement* metaclass to represent ”break” statements.

Metaclass name:	*ConstructorInvocation*
Origin:	*org.eclipse.jdt.core.dom.ConstructorInvocation*
Description:	For *ConstructorInvocation* statements, the depended types include constructor arguments, the type that the constructor refers to and the type arguments used. The *methodsCalled* meta-association is also used to link the *ConstructorInvocation* with its respective constructor *Method*.

Metaclass name:	*ContinueStatement*
Origin:	*org.eclipse.jdt.core.dom.ContinueStatement*
Description:	Concrete *Statement* metaclass to represent ”continue” statements.

Metaclass name:	*DoStatement*
Origin:	*org.eclipse.jdt.core.dom.DoStatement*
Description:	Concrete *Statement* metaclass to represent ”do” statements.

Metaclass name:	*EmptyStatement*
Origin:	*org.eclipse.jdt.core.dom.EmptyStatement*
Description:	Concrete *Statement* metaclass to represent empty statements.

Metaclass name:	*EnhancedForStatement*
Origin:	*org.eclipse.jdt.core.dom.EnhancedForStatement*
Description:	Concrete *Statement* metaclass to represent enhanced ”for” statements.

Metaclass name:	*ExpressionStatement*
Origin:	*org.eclipse.jdt.core.dom.ExpressionStatement*
Description:	Concrete *Statement* metaclass representing regular statements containing an expression.

Metaclass name:	*ForStatement*
Origin:	*org.eclipse.jdt.core.dom.ForStatement*
Description:	Concrete *Statement* metaclass to represent ”for” statements.

Metaclass name:	*IfStatement*
Origin:	*org.eclipse.jdt.core.dom.IfStatement*
Description:	Concrete *Statement* metaclass to represent ”if” statements.

Metaclass name:	*LabeledStatement*
Origin:	*org.eclipse.jdt.core.dom.LabeledStatement*
Description:	Concrete *Statement* metaclass to represent labelled statements.

Metaclass name:	*ReturnStatement*
Origin:	*org.eclipse.jdt.core.dom.ReturnStatement*
Description:	Concrete *Statement* metaclass to represent ”return” statements.

Metaclass name:	*SuperConstructorInvocation*
Origin:	*org.eclipse.jdt.core.dom.SuperConstructorInvocation*
Description:	Much like a *ConstructorInvocation*, the *typeDependencies* of a *SuperConstructorInvocation* consist of the declaring type, the type arguments used and dependencies found in the constructor arguments. The *methodsCalled* meta-association is also used to link the *SuperConstructorInvocation* with its respective constructor *Method*

Metaclass name:	*SwitchCase*
Origin:	*org.eclipse.jdt.core.dom.SwitchCase*
Description:	*Statement* metaclass to represent a statement beginning with the keyword ”case”. Not to be confused with *SwitchStatement*. In Java, *SwitchCases* are part of the contents of a *SwitchStatement*.

Metaclass name:	*SwitchStatement*
Origin:	*org.eclipse.jdt.core.dom.SwitchStatement*
Description:	Concrete *Statement* metaclass to represent ”switch” statements.

Metaclass name:	*SynchronizedStatement*
Origin:	*org.eclipse.jdt.core.dom.SynchronizedStatement*
Description:	Concrete *Statement* metaclass to represent ”synchronized” statements.

Metaclass name:	*ThrowStatement*
Origin:	*org.eclipse.jdt.core.dom.ThrowStatement*
Description:	Concrete *Statement* metaclass to represent ”throw” statements.

Metaclass name:	*TryStatement*
Origin:	*org.eclipse.jdt.core.dom.TryStatement*
Description:	Concrete *Statement* metaclass to represent ”try” statements.

Metaclass name:	*TypeDeclarationStatement*
Origin:	*org.eclipse.jdt.core.dom.TypeDeclarationStatement*
Description:	The type declared in this statement also contributes for its dependencies.

Metaclass name:	*VariableDeclarationStatement*
Origin:	*org.eclipse.jdt.core.dom.VariableDeclarationStatement*
Description:	*VariableDeclarationStatements* may contain several fragments for each variable declared within the same statement, thus leading to the instantiation of the corresponding meta-association with the *LocalVariable* metaclass. The initializer expression contained within a *VariableDeclarationStatement* is scanned to instantiate dependency meta-associations and to obtain the value of the *conditionalOperatorCound* attribute.

Metaclass name:	*WhileStatement*
Origin:	*org.eclipse.jdt.core.dom.WhileStatement*
Description:	Concrete *Statement* metaclass to represent ”while” statements.

Metaclass name:	*CatchClause*
Origin:	*org.eclipse.jdt.core.dom.CatchClause*
Description:	The base AST class *CatchClause* does actually not inherit from the *Statement* class. However, for simplification purposes, it does so in the EJMM.

Enumeration name:	*JavaType*
Description:	Used to differentiate types of the *Type* metaclass. A type may be a class, an enumeration, an interface, an annotation definition or a primitive type. Primitive types include: *int*, *boolean*, *long*, *double*, *float*, *byte*, *short*, *char* and, for simplification purposes, *void* (not to be confused with null values).

Enumeration name:	*VisibilityType*
Description:	Used to define the visibility of a Java member. Default visibility refers to the visibility given to members in which the visibility keyword is omitted (it is visible only inside the package where it is declared).

Enumeration name:	*PackageFragmentRootType*
Description:	Package fragment roots may be either folders or archive files. In the latter case, .zip files and .jar files are differentiated.

3.5 Methods and constraints

As shown in the metamodel presentation section, the EJMM includes certain restrictions and already declares some operations for ease of use. In this section, we shall present the OCL expressions that define both invariants used and operations declared in the EJMM.

3.5.1 Operations

Type operations

Operation:	*getAllNestedTypes()*
Description:	Recursive operation that returns all types nested inside *self*.

Definition:	`getAllNestedTypes() : Set(Type) =`
	`  self.nests`
	`  ->union(self.nests.getAllNestedTypes())`
	`  ->asSet`

Operation:	*getAllEnclosingTypes()*
Description:	Recursive operation that returns all types in which *self* is nested.

Definition:	`getAllEnclosingTypes() : Set(Type) =`
	`  if(self.nestedIn->isUndefined)`
	`  then Set{}`
	`  else`
	`  Set{self.nestedIn}->`
	`  union(self.nestedIn.getAllEnclosingTypes())`
	`  ->asSet`
	`  endif`

Operation:	*getFullInterfaceTree()*
Description:	Recursive operation that returns all interfaces implemented by the type, directly or indirectly.

Definition:	`getFullInterfaceTree() : Set(Type) =`
	`  if(self.javaType = #InterfaceType)`
	`  then Set{self}`
	`  ->union(self.implements.getFullInterfaceTree())`
	`  ->asSet`
	`  else self.implements.getFullInterfaceTree()`
	`  ->asSet`
	`  endif`

Operation:	*getFullInheritanceTree()*
Description:	Recursive operation that returns the full inheritance chain of all superclasses of the *Type*.

Definition:	`getFullInheritanceTree() : Set(Type) =`
	`  if(self.extends.oclIsUndefined)`
	`  then Set{}`
	`  else`
	`  Set{self.extends}`
	`  ->union(self.extends.getFullInheritanceTree())`
	`  ->asSet`
	`  endif`

Initializer operations

Operation:	*getAllStatements()*
Description:	Returns all statements contained in the *Initializer*'s body, ordered by occurrence in the source.

Definition:	`getAllStatements() : OrderedSet(Statement) =`
	`  body.getStatementAndChildren()`
	`  ->sortedBy(startPosition)->asOrderedSet`

Method operations

Operation:	*getAllStatements()*
Description:	Returns all statements contained in the *Method*'s body, if any, ordered by occurrence in the source.

Definition:	`getAllStatements() : OrderedSet(Statement) =`
	`  if(optionalBody.isDefined)`
	`  then optionalBody.getStatementAndChildren()`
	`  ->sortedBy(startPosition)->asOrderedSet`
	`  else OrderedSet{}`
	`  endif`

The *getStatementAndChildren()* operation

The *Statement* metaclass declares the *getStatementAndChildren()* operation, a recursive method to return itself and all other *Statements* included in it, directly and indirectly. The concrete definition varies among the several *Statement* sub-metaclasses. This does not include the statements contained inside types declared inside a *Statement*.

Context:	*Statement*
Definition:	`getStatementAndChildren() : Set(Statement) =` `  Set{}`

Context:	*AssertStatement*, *BreakStatement*, *ConstructorInvocation*, *ContinueStatement*, *EmptyStatement*, *ExpressionStatement*, *ReturnStatement*, *SuperConstructorInvocation*, *SwitchCase*, *ThrowStatement*, *TypeDeclarationStatement*, *VariableDeclarationStatement*
Definition:	`getStatementAndChildren() : Set(Statement) = Set{self}`

Context:	*Block*, *SwitchStatement*
Definition:	`getStatementAndChildren() : Set(Statement) = Set{self}` `  ->union(statements.getStatementAndChildren()->asSet)`

Context:	*DoStatement*, *EnhancedForStatement*, *ForStatement*, *LabeledStatement*, *SynchronizedStatement*, *WhileStatement*, *CatchClause*

Definition:	`getStatementAndChildren() : Set(Statement) =`
	`  if(body.isDefined)`
	`  then Set{self}->union(body.getStatementAndChildren())`
	`  else Set{self}`
	`  endif`

Context:	*IfStatement*

Definition:	`getStatementAndChildren() : Set(Statement) =`
	`  Set{self}`
	`  ->union(thenStatement.getStatementAndChildren())`
	`  ->union(if(optionalElseStatement.isDefined)`
	`  then`
	`  optionalElseStatement.getStatementAndChildren()`
	`  else Set{}`
	`  endif)`

Context:	*TryStatement*

Definition:	`getStatementAndChildren() : Set(Statement) =`
	`  Set{self}`
	`  ->union(if(body.isDefined)`
	`  then body.getStatementAndChildren()->asSet`
	`  else Set{}`
	`  endif)`
	`  ->union(catchClauses.getStatementAndChildren()`
	`  ->asSet)`
	`  ->union(if(optionalFinallyBody.isDefined)`
	`  then`
	`  optionalFinallyBody.getStatementAndChildren()`
	`  ->asSet`
	`  else Set{}`
	`  endif)`

3.5.2 Constraints

We mentioned previously two restrictions to the EJMM - specifically, two invariants for the *LocalVariable* and *Block* metaclasses. These invariants have been translated as OCL con-

straints as follows:

LocalVariable - A *LocalVariable* may only be inside an *Initializer*, a *Method* or be a *Method*'s parameter. It can be in one and only one of these locations.

```
context LocalVariable
    inv localVariableExclusiveLocation:
        self.parameterLocation.isDefined() xor
        self.method.isDefined() xor
        self.initializer.isDefined()
```

Though this condition can be already observed in the class diagrams, declaration of an OCL constraint is needed for the USE specification of the EJMM.

Block - A *Block* may only serve as a body to either a *Initializer* or a *Method*, but not both at the same time. Note that a *Block* may not be necessarily linked directly to a *Method* or *Initializer*. It may, instead, be a part of another *Statement* (for instance, a *Block* may be the *thenStatement* of an *IfStatement*), in which case both meta-associations will be undefined.

```
inv blockExclusiveLocation:
    not(self.method.isDefined() and
        self.initializer.isDefined())
```

We have chosen to only create constraints that would help maintain the metamodel integrity and cannot be specified by the metamodel itself. This means we have not chosen to verify for Java syntactical correctness, as we can already determine whether or not a Java project has compile errors through Eclipse. Thus, the guarantee that an EJMM instance is representative of a correct Java project comes from the plug-in itself, as it will only allow instantiations of projects in which compiling errors cannot be found.

4. OCL APPLICATIONS OF THE EJMM

4.1 Introduction

While developing the M2DM plug-in, we have identified the need to create certain artefacts containing OCL expressions to navigate the EJMM and extract data. The main concern was the implementation of the Formal Library for Metrics Extraction (FLAME) [6] over the EJMM. With the FLAME translated, current OCL formalizations for metrics could be applied with little to no modification. In this chapter, we will include a few OCL expressions to demonstrate basic EJMM navigation. In a second section, we will present the OCL methods that comprise the newly-developed FLAME for the EJMM. The FLAME is an extension of the EJMM and is defined in a seperate file. Finally, in the last section, we will present a few more operations required for calculating specific metrics found in literature.

4.2 EJMM navigation

Querying the EJMM is a fairly easy matter thanks to the simplicity of OCL. Traversing the metamodel can be done by accessing to metaclasses' meta-associations (and the meta-associations' meta-associations until finding the desired metaclasses or meta-objects). For instance, the following expression returns all the *Statements* that depend on the *Type int*:

```
int.dependedStatements
```

To retrieve all the *length* values of the *Statements*, we simply need to append the *length* attribute, like so:

```
int.dependedStatements.length
```

And to calculate the total length:

```
int.dependedStatements.length->sum
```

Please note that meta-association navigations that take a single step and with multiplicity higher than 1 (such as from *Type* to *dependendStatements*), return a *Set* of the target metaclass. However, when navigating through two or more metaclasses, a *Bag* is returned instead, as there might be repeated instances. Operations return their respective return type, but when being called after a meta-association, the results are also collected in a *Bag* of the return type. The same occurs for attributes used after a meta-association.

There might be times when the user does not have access to a meta-object's specific name, but has knowledge of its declared name in Java. For these occasions, the *name* attribute can be used to retrieve a specific meta-object from all instances of its metaclass. For example, to access all the *Types* included in the "Foobar" *JavaProject*, we can use the following expression:

```
JavaProject.allInstances->any(name = 'Foobar').packageFragmentRoots
.packageFragments.typeRoots.types
```

Note that while our M2DM tool only instantiates one project at a time, the previous expression always returns results different from the expression Type.allInstances, since external and basic *Types* are contained in *TypeRoots* and *PackageFragments* that are not part of an analysed *JavaProject*.

These last few OCL expressions can serve as queries for the OCL evaluator in USE or our M2DM prototype plug-in. Defining OCL operations in a USE specification is a similar process. Here is an example of an operation defined for the *Method* metaclass that returns true if a *Method* is either abstract or part of an interface:

```
isAbstactOrInterface() : Boolean =
   self.isAbstract or self.type.javaType = #InterfaceType
```

Note that it is possible to give use to the "self" keyword to refer to the current instance of a metaclass, much like the "this" keyword in Java, and then navigate the EJMM from there. Several more examples of operations can be found in Subsection 3.5.1 in chapter 3, and more are to follow in the following two sections of this chapter.

4.3 FLAME for EJMM

The Formal Library for Aiding Metrics Extraction (FLAME) [6, 16] is an OCL library devised to help implement existing M2DM metrics (such as the MOOD [34] and Chidamber

& Kemerer metrics set [2]) over the UML metamodel. For the sake of portability of those metrics sets we decided to implement the FLAME upon the EJMM. The original FLAME specification defined upon the UML metamodel and, as such, we had to rewrite it for compliance with existing EJMM metaclasses.

The process of conversion is mostly a matter of translating UML concepts into Java ones, with several UML generalized elements translating to specific EJMM metaclasses. Rules of visibility, inheritance and overriding have also been revised to what is dictated in the Java tutorials [35]. This conversion process also incurred to the formalization of ”client” and ”supplier” concepts for Java constructs.

The following aspects regarding the FLAME for EJMM are of note:

- Operations that deal specifically with ”classes” in the UML FLAME (especially those defined at the *PackageFragment* level) deal only with *Types* that are classes - their *javaType* equals *ClassType*.

- We use the *Member* metaclass as a representative of a ”feature”. However, we did not consider all *Member* types to be features - *Initializers* in particularly. Nested *Types* are still considered features of their enclosing *Type*.

- Regarding overriding, we decided that only *Methods* override and *Fields* do not. This is due to the distinction found in the Java tutorials [35], where methods can override, but fields can only become hidden.

- The clients of a *Field* are all the *Types* which access the *Field* in the bodies of its declared *Initializers* or *Methods*. Likewise for *Methods*.

- Clients of a *Type*, however, are a more complex matter. To aid the definition of a *Type*'s clients, we first determined what *Type* suppliers a given *Type* needs. We figured out that the suppliers of a type are the types it inherits from, the types of its fields, return types of its methods, the types of the annotations used for it and its members, the types of all local variables found in its methods or initializers and the types invoked in the bodies of all of its methods and initializers.

The first version of the FLAME for EJMM can be found in Appendix section A.1. All names of pre-existing FLAME operations have been preserved except for the *client* operation, changed to the clearer *clients* name.

4.4 Other Operations and Metrics

Despite the translation of FLAME to a EJMM-compliant version already allowing the use of existing software metrics, in this section, we shall cover a few more operations for metrics calculation. Firstly, we present an example of the use of the *conditionalOperatorCount* attribute of the *Statement* metaclass. The attribute was created with the intention of helping calculate McCabe's [1] cyclomatic complexity of a method. One possible formalization for such a metric - specifically made to match the way it is calculated in Frank Sauer's metrics plug-in [15] - can be defined as follows:

```
complexity() : Integer = 1 +
   self.getAllStatements()->select(s |
   not(s.oclIsKindOf(ReturnStatement))).conditionalOperatorCount->
   excluding(oclUndefined(Integer))->sum + self.getAllStatements()->
   select(s | s.oclIsKindOf(CatchClause) or
   s.oclIsKindOf(DoStatement) or s.oclIsKindOf(ForStatement) or
   s.oclIsKindOf(IfStatement) or (s.oclIsKindOf(SwitchCase) and
   not(s.oclAsType(SwitchCase).isDefault)) or
   s.oclIsKindOf(WhileStatement))->size
```

And the same metric including corrections found in the continued version of the plug-in, by Guillaume Boissier [28], which also counts the number of enhanced for statements:

```
complexity() : Integer = 1 +
   self.getAllStatements()->select(s |
   not(s.oclIsKindOf(ReturnStatement))).conditionalOperatorCount->
   excluding(oclUndefined(Integer))->sum + self.getAllStatements()->
   select(s | s.oclIsKindOf(CatchClause) or
   s.oclIsKindOf(DoStatement) or s.oclIsKindOf(ForStatement) or
   s.oclIsKindOf(EnhancedForStatement) or -- Newly added
   s.oclIsKindOf(IfStatement) or (s.oclIsKindOf(SwitchCase) and
   not(s.oclAsType(SwitchCase).isDefault)) or
   s.oclIsKindOf(WhileStatement))->size
```

Cahill et al. [23] propose, with their Source Lines of Code (SLOC), that counting the number of statements is a more accurate measurement of size of source code. With the EJMM, we can calculate the number of *Statements* in the body of a *Method* or an *Initializer*, as follows:

```
SLOC() : Integer =
   self.getAllStatements()->select(s | not(s.oclIsKindOf(Block)))->size
```

Note that *Block* statements are not included as they are just containers of statements without any information of its own - in Java, blocks are delimited by a pair of braces ({}).

Furthermore, we can also make a full statement count for a *CompilationUnit* if we were to consider *Type*, *Field*, *Initializer* and *Method* declarations as statements.

```
SLOC() : Integer =
   self.types->size + self.types.initializers->size +
   self.types.methods->size + self.types.fields->size +
   self.types.methods.SLOC() + self.types.initializers.SLOC()
```

Another basic metric defined by Cahill et al. is the Comment Lines of Code (CLOC). Though the EJMM does not count the number of text lines on a *ASTNode*, much like SLOC, we can count the total number of *Comments* in a *CompilationUnit* and even the total size of all *Comments* in bytes:

```
totalComments() : Integer =
   self.comments->size

totalCommentSize() : Integer =
   self.comments.length->sum
```

[This page was intentionally left blank]

5. ECLIPSE PLUG-IN FOR M2DM

5.1 Introduction

Before creating the final product of the M2DM project, a modified version of Frank Sauer's [15] metrics plug-in for Eclipse, we first sought to create a simple prototype plug-in, serving as a proof-of-concept for M2DM capabilities. This prototype, though lacking in advanced reporting, automation and exporting features, served as a testing ground in which to implement EJMM instantiation and navigation capabilities. In this chapter, we shall present the current prototype, its structure and architecture, explain the process in which it converts a Java project in Eclipse into an EJMM instance consisting of USE [29, 30] objects, and finally, our validation methods for the tool.

5.2 Tool architecture

To create the prototype M2DM tool, we gathered three main components:

i An OCL compiler embedded in the UML-based Specification Environment (USE) [29, 30]

ii A façade interface named J-USE produced within the QUASAR group [32] that provides a Java API for USE services

iii A transformation component that goes through the EJM and the Java AST and generates EJMM instances by requesting USE services through J-USE

J-USE allows loading UML models or metamodels, instantiate them, test defined OCL contracts and execute other OCL expressions as queries over the model or metamodel instances. Thus, we will be able to perform traversal operations defined in OCL upon the EJMM and check the properties of concrete JVM instantiations, that is, of actual Java programs.

The EJMM is defined as a USE specification file (UML class diagram in textual format) distributed with the plug-in. Metrics definitions are expressed using OCL as operation definitions upon the FLAME library or upon the EJMM directly, without needing to change any Java code.

The plug-in source code includes four packages and two Jar archives. The latter are archive versions of USE and J-USE. The four packages are:

- `ejm2metrics` - the base package containing the *Activator* class required by Eclipse to run the plug-in.
- `ejm2.views` - contains the declaration of the M2DM view, including its buttons, menus, text labels and text editors.
- `ejm2.ui` - contains classes that create an instance of the M2DM view and can capture event triggers fired by the view's buttons and OCL evaluator text area to initiate the tool's functions.
- `ejm2.tools` - package containing the classes responsible for loading the EJMM and crawl through a supplied Java project to instantiate the metamodel through USE objects.

Note that, for the source code and namespaces, we refer to the EJMM as "EJM2" to help visualize name meanings and remove naming ambiguity when several words are attached.

5.3 Tool presentation

The plug-in interface consists of an Eclipse view from which users can initiate the loading process and input OCL queries for evaluations. Like most externally-installed views in Eclipse, our plug-in is accessed by the "Other" option of the "Show View" sub-menu in the "Window" menu. We have included the plug-in view in the "M2DM" category and dubbed it the "Interactive view".

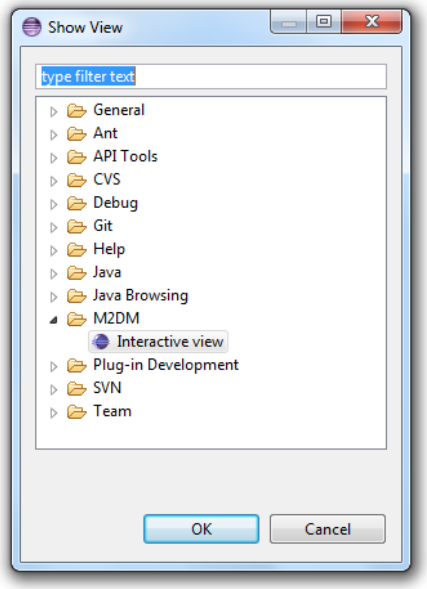


Fig. 5.1: Selecting the MD2M prototype view

From the M2DM view, the user must first locate their USE installation folder using the ”Select USE directory” button and the location of the .use file containing the EJMM using the ”Select EJMM path” button. Then, the user must select the project to analyse from the drop-down menu. Only Java projects without errors can be selected. After these steps have been taken, the user can press the ”Instantiate EJMM” button to initiate the metamodel instantiation process. Once that process is finished, the upper text box becomes active, in which the user may input OCL queries in the same fashion as the OCL evaluator of the USE tools. Query results appear in the larger text area below. The user may also choose to load extra .use files extending the metamodel, such as the one containing the FLAME formalization in chapter 4. Several files may be selected and the user must do so before initializing the instantiation process.

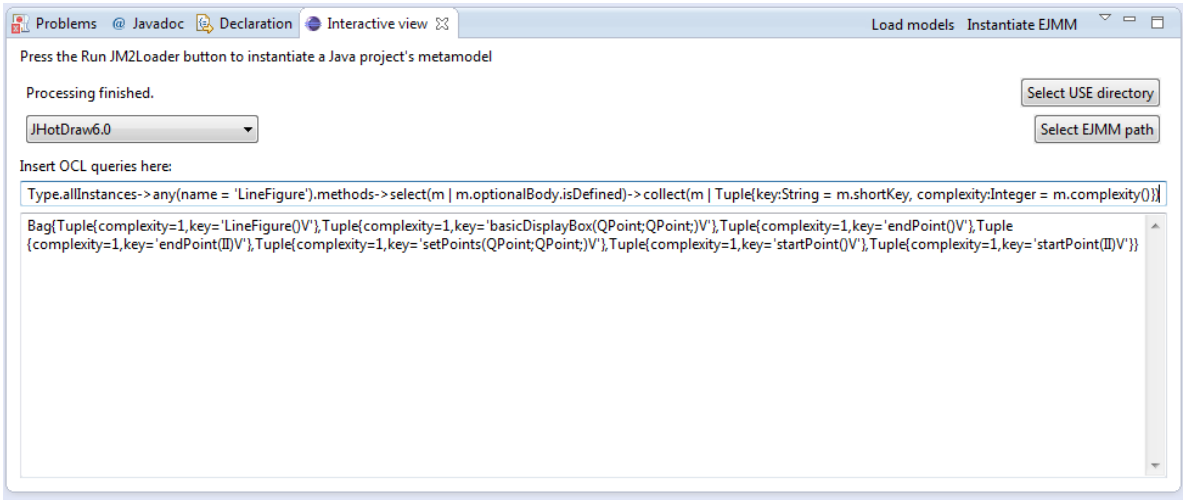


Fig. 5.2: The M2DM prototype view

The current prototype and its source code can be found in `http://code.google.com/p/m2dm/`.

5.4 Instantiation process

Despite the EJMM defining the structure of the data, the instantiation process, where EJM and AST components are translated into USE objects, determines the exact data that is included. Decisions made during the EJMM instantiation tools for the metrics plug-in greatly affected what is available for analysis. First of all, the current instantiation process can be divided in two major phases: a first one that traverses a Java project's respective Java Model tree, creating USE objects of the whole structure and a second phase that inspects the contents of the EJM and AST components to create links between them.

5.4.1 First instantiation phase

The first instantiation takes a top-down approach, from Java project to type components. During this phase, the following processes take place:

1. Firstly, common meta-objects are created, such as primitive types and place-holder locations for said primitive types and external types found during the second processing phase. The entire list of these common meta-objects can be found in section C.1 of Appendix C.

2. The top *JavaProject* is created.

3. Then, its *PackageFragmentRoots* meta-objects, obtained through the *getAllPackageFragmentRoots()* method from *IJavaProject* are created, as well as the meta-links with the corresponding *JavaProject*.

4. *PackageFragment* meta-objects are then created from the *getChildren()* method from *IPackageFragmentRoot*, as well as the corresponding meta-links. Note that this method is not specific to *IPackageFragmentRoot* and is meant to return the direct children of a given *IJavaElement*, which in this case, is a *IPackageFragment*. Packages that do not contain Java resources (either compilation units or class files) do not get respective instances in the EJMM.

5. Each *IPackageFragment* is scanned for either compilation units or class files (through the *getCompilationUnits()* and *getClassFiles()* methods from *IPackageFragment*, respectively), creating meta-objects from each, as well as the corresponding meta-links.

6. Then, types contained in class files and compilation units (obtained through the *getType()* and *getAllTypes()* methods from *IClassFile* and *ICompilationUnit*, respectively) have their *Type* meta-objects created, as well as the meta-links between *Types* and their *TypeRoot*.

7. Each *IType* obtained previously is scanned for methods, initializers and fields (through the *getMethods()*, *getInitializers()* and *getFields()* methods, respectively), creating *Method*, *Initializer* and *Field* meta-objects.

8. *Methods*' *type*, *Initializers*' *type* and *Fields*' *type* are defined at this moment.

9. The instantiation process moves on to the second phase.

5.4.2 Second instantiation phase

Once all the major components of the Java project are known, the second processing phase begins. This second process is responsible for creating every meta-object and link between meta-objects that were not included in the first processing phase, including type inheritance, field types or return types. Firstly, previously-scanned compilation units are retrieved. The annotations attached to its package declaration are analysed, then its declared types. Types are now checked for type parameters, bounds, inheritance and nested types. Then, its methods are scanned for its parameters (creating *LocalVariable* instances), return type, exception thrown and attached annotations. Fields are then inspected for their field type and

annotations attached. Once the types are fully scanned, an AST created from the compilation unit code is analysed. Using a visitor pattern (made possible by extending the *org.eclipse.jdt.core.dom.ASTVisitor* class), ASTs are scanned for comments, method declarations and initializers. The *MethodDeclaration* and *Initializer* classes of the AST (part of the `org.eclipse.jdt.core.dom` package) are the ones used to bridge the EJM and AST components of the EJMM. *Block* instances are created, linked to their respective *Method* or *Initializer* and then their contents are analysed using a recursive function specific for each type of *Statement*. Each *Statement* that contains an expression has its expression inspected for dependencies and conditional operators and statements contained within are also inspected using the same recursive function. For a full list of *Statements* that may contain *Expressions*, see Table C.1. A special case is the *VariableDeclarationStatement* class, which contains *VariableDeclarationFragments* (of the `org.eclipse.jdt.core.dom` package). These are scanned to find the variable initializer *Expression*, which adds directly to the result of the *VariableDeclarationStatement*. The methods that retrieve dependencies and conditional operators function much alike the statement analysing process in that expressions are scanned for other expressions contained within, adding to the final result.

Type dependencies are determined by the type bindings (*ITypeBinding* of the *org.eclipse.jdt.core.dom* package) that can be obtained from some expressions. The types that these bindings return are later used to create the *typeDependencies* meta-links. Specifically, the following *Expression* (found in the *org.eclipse.jdt.core.dom* package) subclasses produce type bindings:

- *CastExpression*
- *ClassInstanceCreation*
- *FieldAccess* (produces two type bindings, one for the field type and another for the field's declaring type)
- *InstanceofExpression* (from its right operand)
- *MethodInvocation* (produces two type bindings, one for the method's return type and another for for the method's declaring type; its arguments are also scanned)
- *SimpleName*
- *QualifiedName*
- *TypeLiteral*

Field accesses are determined by the *IVariableBindings* obtained from the *FieldAccess* class. Method calls are determined by the *IMethodBindings* obtained from the *ClassInstanceCreation* and *MethodInvocation* classes. *ConstructorInvocation* and *SuperConstructorInvocation* statements also contain *IMethodBindings* regarding their declaring constructor, which are also used for generating *methodsCalled* meta-links.

Regarding the *conditionalOperatorCount* attribute of the *Statement* metaclass, statements that have expressions are scanned to find two specific types of expressions that may add to its value: *ConditionalExpression* and *InfixExpression*. *ConditionalExpressions* found always add 1 to the count, but *InfixExpressions* only add if its operator (from the *getOperator()* method) is a conditional ”and” or conditional ”or” operator - in which case it adds 1 to the count plus the total number of extended operands. Extended operands of an *InfixExpression* are obtained through the *extendedOperands()* method and define the remaining *Expressions* linked to the *InfixExpression* with the same operator. Note that, being a recursive process, any *Expression* contained inside another *Expression* also adds its value. These current rules were chosen to aid the calculation of a method's cyclomatic complexity [1] in the same way Frank Sauer's metrics plug-in [15] does.

During the second processing phase, if a reference is made to a type that was not instantiated during the first phase, a new meta-object is created to represent the newly found external type and linked to the *TypeRoot* reserved for external types. This meta-association allows for easy distinction between internal and external types to a Java project. If, when creating an *AnnotationValue* meta-object, an annotation member-value pair (*IMemberValuePair*) has a member name that does not correspond with any existing *Field* - a common occurrence when inspecting the values of an annotation with an external declaring type - a new *Field* meta-object is created, with a *name* equal the member name and the rest of its attributes left undefined. The new *Field* is then linked to the *AnnotationValue*. Furthermore, if a field that was not instantiated during the first phase is accessed by a *Statement*, a new *Field* meta-object is created, linked to its respective *Type* and declaring *Type*. If these *Types* cannot be found either, new external *Types* are created, since a *IVariableBinding* can also produce the *ITypeBinding* for both its type and declaring type. Subsequent references to the same type, such as from analysing different *Statements*, are directed to this *Type* instance. The same logic is applied for *methodCalls*. If a method that was not instantiated during the first processing phase is called, a new *Method* instance is created and linked to its declaring *Type* and returning *Type*.

5.4.3 Other rules

As can be observed from the rules defined in this section, only types defined in compilation units are subject to the second processing phase in the current version of the tool. Binary types are and their structure are included in the instantiation and thus have respective meta-objects, but their contents are not inspected.

All meta-objects originating from EJM components have unique USE object names equal to their *handleIdentifier* with all non-alphanumeric and non-underscore characters removed. Since this might result in name conflicts between fields and methods with the same signature when excluded of invalid characters, all *Method* and *Field* meta-objects have names starting with "METHOD_" and "FIELD_" prefixes, respectively. Initializers are an exception, as their name is equal to the processed *handleIdentifier* of the *Type* in which they are declared, concatenated with the string "_Initializer" and its *occurrenceCount*. AST meta-objects have the name of the metaclass followed by the current count of total meta-objects for the metaclass for their USE object name. External *Types* have names equal to their type signature. External *Fields* and *Methods* found when scanning for *Statement* dependencies follow the same naming convention as regular *Fields* or *Methods*, however, the EJM may provide incorrect handle identifiers for JRE library components - the functionality of the prototype does not suffer, but the respective *handleIdentifier* attribute and USE object identifier may produce undesirable values. For instance, the EJM might provide a handle identifier of a JRE library component with a project name different from the one instantiated. Basic types have names equal to their *name* attribute (for instance, *int*'s meta-object name is also *int*).

These rules often result in long names that are hard to read, so it is more advisable to search for specific meta-objects from all its instances with the "name" attribute, if applicable. For instance, the following OCL expression returns the first *Type* with the name "Foobar":

```
Type.allInstances->any(name = 'Foobar')
```

Since the instantiation rules heavily influence the data that the EJMM offers, they are subject to change depending on future requirements.

5.4.4 Instantiation classes

The following classes are responsible for the instantiation tasks:

- *JM2Loader*
- *JM2ASTLoader*

- *StatementInspector*
- *ExpressionAnalyzer*

The *JM2Loader* class is responsible for initiating the full instantiation process through the static*loadEJMMfromProject* method, which receives a single *IJavaProject* as a parameter. The *JM2Loader* class is also responsible for all EJM-related operations. During the second instantiation phase, it calls upon services provided by the *JM2ASTLoader* class, that uses the aforementioned visitor pattern to find methods (*MethodDeclaration*), initializers (*Initializer* class of the AST) and comments (*Javadoc*, *LineComment* and *BlockComment* classes of the AST). To inspect the contents of methods and initializers, it uses the *StatementInspector* class. *Statement* types that contain expressions call upon the *ExpressionAnalyzer*'s two methods to return statement type dependencies and the value to use on the *conditionalOperatorCount* attribute.

5.5 Tool validation

For validation sake of our M2DM tool, we first defined its quality model. The quality characteristics that we identify in a metrics collection tool are then the following:

- **Transparency** - the tool should allow a clear identification of the calculation algorithms used in metrics collection;
- **Extensibility** - the tool must allow adding new metrics.
- **Scalability** - the tool should allow metrics collection in a reasonable time frame, at the expense of resources typically available in current personal computers;
- **Accuracy** - the collected metrics values should be accurate;

For the remainder of this section, we will provide our evaluation of the M2DM prototype we have developed.

5.5.1 Transparency

Transparency is one of the biggest strengths of our tool. As mentioned previously, metrics definitions are perfectly identified in separate OCL files loaded by the user.

5.5.2 Extensibility

Tool extensibility (i.e. adding new metrics sets) is fairly straightforward since the concepts used in metrics definition are those defined in the EJMM. The user only needs to create a new file with OCL definitions of the desired metrics. To validate this capability, we have developed the FLAME [6] for EJMM, presented in the previous chapter. With FLAME, it is possible to reuse metrics sets specifications that were already formalized using the FLAME, such as the MOOD [34] metrics set. Validation of the metrics sets thus obtained is subject of future work, using benchmark data, obtained from a commercial tool that implements the MOOD set.

5.5.3 Scalability

To test the scalability, we ran the M2DM tool over Java projects of increasing size, comparing the duration of the instantiation process. We chose five different projects as test cases, including three key components for the M2DM project: J-USE [32], Frank Sauer's metrics plug-in for Eclipse [15] and the USE tool [29] in its 3.0.6 version. We also tested the instantiation time over a software application that is widely-used for research purposes, JHotDraw [36] (version 6.0), and over a popular open-source application, SweetHome3D [37] (version 4.1), which enjoys over one hundred thousand weekly downloads at the time of writing. Project size, for these tests, was determined by the total number of USE objects and USE links generated by the instantiation process (meta-info). The sizes of projects can be found in Table 5.1, showing the order of size of the Java projects tested, J-USE being the smallest and USE the largest. To further illustrate differences in sizes, we have also included the total number of *Types* instantiated for each project (including external and basic types).

Project / **Version**	**J-USE** **1.0**	**Eclipse Metrics** **1.3.6**	**JHotDraw** **6.0**	**SweetHome3D** **4.1**	**USE** **3.0.6**
Types	195	820	3,403	5,731	9,261
Meta-objects	3,491	17,240	64,147	136,716	211,452
Meta-links	10,356	43,303	112,806	311,791	395,460
Total meta-info	13,847	60,543	176,953	448,507	606,912

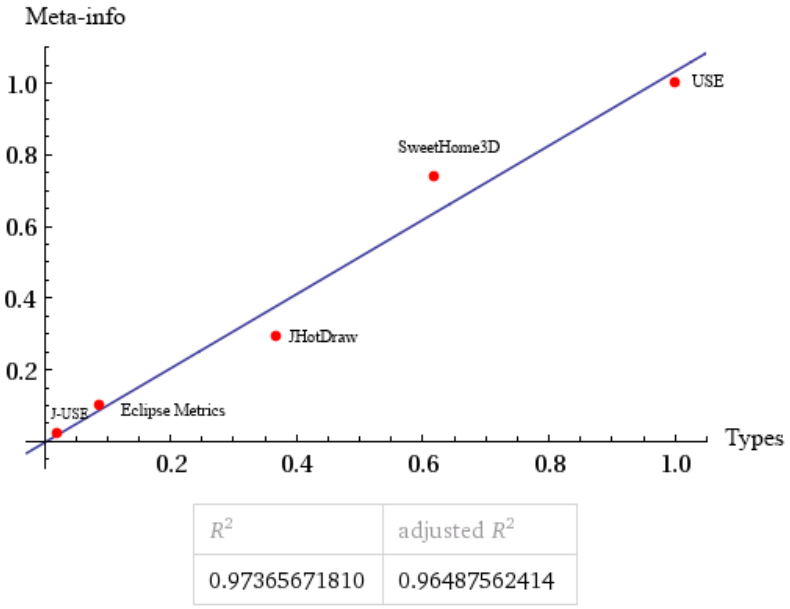
Tab. 5.1: Scalability tests: project sizes

To give a better view of the contribution of quantity of types in a project to its size in meta-info, we mapped normalized meta-info values to normalized *Type* quantity values and

found the following linear approximation:

$$y = 1.036296166469x - 0.003765204055325$$

Where y represents meta-info and x represents the number of *Types*. This results in the following plot:



R^2	adjusted R^2
0.97365671810	0.96487562414

Fig. 5.3: Type quantity to project size plot

This suggests that the relation between amount of types in a project and resulting meta-info is close to a linear one.

The scalability tests consisted of running the instantiation process fifty times over each project, in order of largest in size to smallest, and registering the times for the first instantiation phase, the second instantiation phase and the total duration for the whole process. All tests were made under the same Eclipse session and due to the JVM being under heavier load during the very first instantiation (belonging to USE 3.0.6), its value was discarded to avoid skewed results. Tests were made on a machine with a dual-core processor running at 3GHz each and 4GB of physical memory. JVM maximum heap space was set to 1024MB, as opposed to the default 512MB, to allow the instantiation of the largest project, USE. The full results can be found in tables C.2 trough C.6 with values in seconds. A summary of the tests can be found in Table 5.2.

Project	1st instantiation phase	2nd instantiation phase	Total duration		Average proportion	
	Mean [s]	Mean [s]	Mean [s]	SD [s]	1st phase	2nd phase
J-USE	0.049	0.648	0.697	0.033	7.08%	92.92%
Eclipse Metrics	0.377	3.952	4.329	0.279	8.71%	91.29%
JHotDraw	3.604	10.189	13.793	0.591	26.13%	73.87%
SweetHome3D	8.270	36.034	44.304	1.617	18.67%	81.33%
USE	22.509	70.973	93.482	0.585	24.08%	75.92%

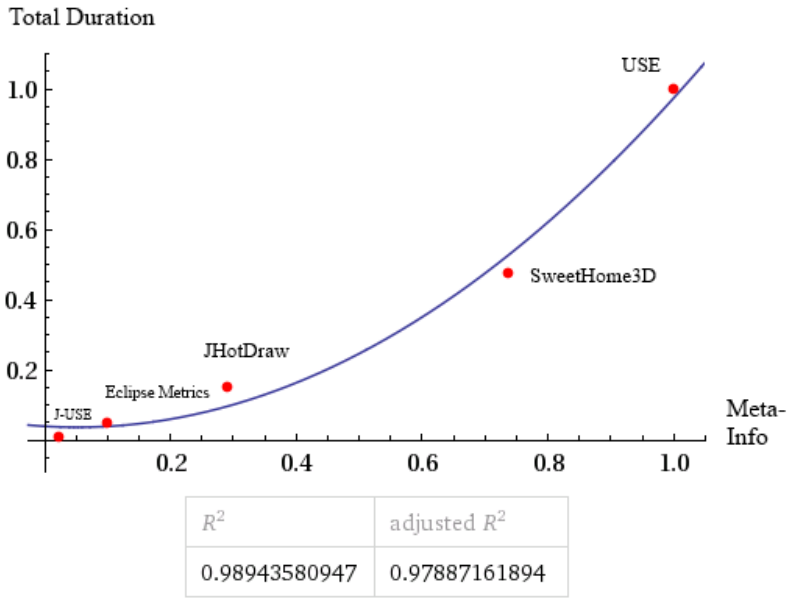
Tab. 5.2: Scalability tests summary

As expected, the larger the Java project, the longer it takes for the instantiation process to finish. The results are fairly consistent for each project, enjoying low standard deviations. We can also observe that the second instantiation phase is consistently the most responsible for the duration times, due to the tool having to iterate over several ASTs. We can confirm that all values are acceptable for everyday use, with even a large scale project such as USE having instantiation times that take around a minute and a half.

The next concern was determining how much the durations increased compared to increases in project sizes. To do this, we used a polynomial regression of degree 2 where the instantiation duration is given as a function of project size in meta-info. By mapping normalized size to normalized duration values, we obtained the following function:

$$y = 1.042918105510x^2 - 0.1075328190896x + 0.03894345306869$$

Where y refers to instantiation duration and x the project size. This function results in the following plot:



R^2	adjusted R^2
0.98943580947	0.97887161894

Fig. 5.4: Instantiation duration to project size plot

We can use this quadratic approximation to suggest that changes in size incur more-than-proportional changes in process duration. This shows it will be exceedingly difficult to analyse increasingly larger Java projects.

5.5.4 Accuracy

To determine the prototype's accuracy, our validation approach is based on a comparison between the values of the same metrics collected with the M2DM tool with those of the original Frank Sauer plug-in. The latter is in use for several years and has a large number of downloads and several updates, so we have a good confidence that its calculated values can be used as a benchmark.

To this date, we have tested the prototype's capability for calculating a method's cyclomatic complexity [1]. To do this, we employed the OCL operation defined in section 4.4 of the previous chapter that was defined to create an approximation to Frank Sauer's calculation method. The JHotDraw application [36] was chosen as a test subject. From it, we used the M2DM prototype to calculate the complexity of all non-abstract methods of all non-interface and non-nested types that are not included in JHotDraw's test packages (the full list of packages we have excluded from our analysis can be found in Table C.7), for a total of 245 types. We then selected a sample of 1005 methods chosen by alphabetical order of the compilation unit in which they are contained. The full results can be found in Tables C.8 through C.26.

The columns named ”Source” determine the compilation unit where the method, named in columns ”Method name”, can be found. The columns named ”EM value” determine the complexity value of the method as calculated by Frank Sauer’s plug-in. Columns named ”M2DM value” determine the complexity as initially calculated by our prototype. The final columns, named ”Recalculated value”, determine the complexity as calculated by our prototype after further corrections - more information on that later.

Regarding the first calculation by our M2DM prototype, 97.21% of all values coincided between the two tools, as such:

Total values:	1005
Matched values	977
Different values	28
Percentage of coincidence:	97.21%
Average difference:	0.046766
Average difference from different values:	1.678571
Largest difference:	7

Tab. 5.3: M2DM to Eclipse Metrics comparison: initial results

Facing these results, we inspected the cases where values were different and proceeded to perform corrections over our M2DM tool to provide more accurate results. The corrections made included:

- Including extended operands of *InfixExpressions* in the analysis of expressions to obtain the *conditionalOperatorCount*
- Including *MethodInvocation* argument expressions (as obtained by the *arguments()* method)
- Analysing the expressions contained in *VariableDeclarationFragments* of a *VariableDeclarationStatement*
- Excluding *ReturnStatements* from the calculation (the expression found in section 4.4 is the final version)

Due to time constraints for the publishing of this dissertation, we did not manage to repeat the full metrics extraction for all 1005 methods. Instead, we calculated the new values for the 28 methods in which values differed, found in columns named ”Recalculated values” in Tables C.8 through C.26. Out of the 28 different values, 22 were corrected to coincide

between the two plug-ins and 6 remained different. The six remaining methods with different complexity values differed for two reasons:

i Frank Sauer's plug-in counts all occurrences of "&&" and "||" in the source code, as opposed to identifying the *Operator* object of *InfixExpresions* in the AST. This results in commented code adding to a method's complexity. This is the case in the *findConnectionTarget* method in *ChangeConnectionHandle.java*.

ii Frank Sauer's plug-in also adds to a method's complexity those of all methods of all anonymous classes contained inside it. This happens in the *CTXWindowMenu* constructor in *CTXWindowMenu.java*, *createComponentListener* in *DesktopEventService.java*, *createDragGestureListener* in *DragNDropTool.java*, and *open* and *createDesktopListener* in *DrawApplication.java*.

As the correctness of these points is a subject of discussion, we have decided to not change the prototype's calculation process further.

However, corrections made to the plug-in might have come with possibilities of side-effects changing values that were accurate from the start. To ensure that this is not the case, we recalculated the complexity for a smaller sample of 48 methods in which the previous test provided 100% accurate results: all methods of the *CompositeFigure* type, from the *CompositeFigure.java* compilation unit. The results can be found in Table C.27. The values remained 100% accurate. Though it is not proof that there were no unwanted side-effects, we took it as evidence with acceptable confidence, that the values that were previously accurate remained so. Thus, if we consider that all previous values that coincided still coincide, the new accuracy validation results can be summarized thus:

Total values:	1005
Matched values	999
Different values	6
Percentage of coincidence:	99.4%
Average difference:	0.017191
Average difference from different values:	3
Largest difference:	7

Tab. 5.4: M2DM to Eclipse Metrics comparison: final results

These final results were considered adequate for the M2DM tool in its prototypical stages.

[This page was intentionally left blank]

6. CONCLUSIONS AND FUTURE WORK

6.1 Discussion

In this dissertation we tackled the first challenges in creating a M2DM plug-in for Eclipse. Firstly, we have created a new Java metamodel, the EJMM, based on data provided by the Eclipse Java Model and Abstract Syntax Tree. Secondly, we have successfully created a prototype plug-in that can analyse a Java project in Eclipse and translate it to an instance of the EJMM that is navigable using OCL as a query language. Furthermore, we have presented an EJMM-functional version of FLAME [6], to ease future work on further implementations of software metrics over the EJMM.

From the development of the mentioned artefacts, points of discussion surfaced that made us take certain decisions.

One of our first questions was one of complexity regarding the metamodel itself. The EJMM is not a direct translation of the EJM and AST, but a simplification (and unification) of both. As one of the advantages of M2DM cited by its authors being the distancing of complex parser logic and programming languages to a static metamodel and the simpler OCL for expressions [3, 7], it would be counter-productive to create a metamodel that is hard to navigate. On the other hand, with greater detail, such as more metaclasses and attributes, there would be more points of measurement. Ultimately, the stance we took was somewhat of a midpoint. The result of this was the inclusion of Java constructs that are often not measured, such as *Annotations* and the entire breadth of *Statement* types, as well as the exclusion of certain details, such as package declarations (IPackageDeclaration of the EJM) or the abstract *Expression* class of the AST and its subclasses. The information these would provide that we have deemed relevant for measurement has been abbreviated in other *Metaclasses* - package declaration annotations translated into the *CompilationUnit* metaclass inheritance of the *Annotatable* metaclass, and the dependencies meta-associations and *conditionalOperatorCount* attribute for the *Statement* metaclass. We wish that the EJMM, in its current version, will be useful for the unforeseeable future of M2DM and hope our choices of simplification do not go against the requirements of future users. Despite this wish, we are not any less open to

make changes to the version presented in this dissertation.

The development of the prototype also raised several questions relevant to our research. Despite the structure provided by the EJMM, the contents are still at the mercy of the rules defined by the instantiation process. Despite the EJMM metaclasses being representations of JDT equivalent interfaces and classes, the complex nature of Java projects resulted in specific choices to determine what gets instantiated and how. These choices may have a direct impact on metrics values, even with the same OCL expressions. While OCL formalizations for metrics definitions can be evaluated and validated through its syntactical correctness, instantiation correctness is more difficult to ascertain. However, some choices made are fairly unique and may contribute to more complete analysis of Java projects. Specifically, instantiating binary and external types is something that cannot be found in Frank Sauer's existing plug-in [15]. This can be useful to determine, for instance, outward dependencies from developed resources to third-party ones.

Choices in EJMM structure and instantiation policies are why validation of the tool's accuracy was a considerable concern. The tests presented in the previous chapter are promising for a prototype, as we achieved almost 100% accuracy in relation to Frank Sauer's plug-in, but fairly inconclusive, as only one metric was tested.

Scalability tests showed promising results as well, as despite a more-than-proportional rise in instantiation times in relation to rises in project sizes, even large-scale projects such as USE showed low absolute instantiation duration values, with average times only slightly above a minute and a half. However, it is still of note that we needed to increase the JVM maximum heap size for such large-scale projects. Memory issues may come with the implication that changes to the EJMM involving the addition of more meta-associations or meta-objects could make the scalability problematic.

6.2 Future work

As accuracy tests are lacking, we hope that in the future, we will be able to perform more and more complete tests with other metrics supported by Frank Sauer's plug-in. The extensibility tests, as mentioned in the previous chapter, will also be the subject of future work, as the FLAME has not been subject of thorough validation. And of course, if new inaccuracies are found, the EJMM and the instantiation process will be subject to change accordingly. In regards to scalability, it would be of interest, in the subject of future work, to measure the time it takes for the prototype to execute metrics calculations over large sets of meta-objects. Typical OCL queries result in very short execution times under a second, but unmeasured.

The prototype plug-in's functionality shall also be subject of future work, as there are a few technical flaws yet to be addressed. Firstly, the tool is still fairly unstable and prone to crashing Eclipse when encountering USE specification files with syntactical errors. Secondly, the user must still specify the locations of an USE installation on their machine and the EJMM USE specification file to initiate the loading and instantiation process. Since these two components are always required to run the tool, we plan to find a way to bundle these with the plug-in distribution and take away this requirement for the user. Thirdly, in terms of scalability, we believe that the current prototype still has much room for improvement regarding optimization of the instantiation process, which can be subject of future work. Fourthly, the calculation of metrics depends on using the built-in OCL evaluator, which requires the input of textual OCL expressions whenever we want to view the value of a certain metric. This is a fairly time-consuming and impractical solution and automation of this process would greatly improve the tool's usefulness. Fortunately, this last matter is something that will be resolved naturally with the next stage of the M2DM plug-in project, since Frank Sauer's metrics plug-in already provides extensive automation capabilities, as well as metrics propagation and an orderly display of results.

Automatic metrics propagation in itself is also a subject of future study. As Frank Sauer's plug-in offers several propagation options (such as calculating total, average or maximum values), the question remains on how does the user define, on the tool front-end, how their user-defined metrics should propagate.

Ultimately, the work presented in this dissertation serves as a proof of concept of M2DM research. By creating the EJMM and measuring software through OCL-defined metrics, we have successfully segregated the complex compiler and parser logic from the simpler and more explicit model-driven side. It is indeed possible at this point for users to create new software metrics without writing a single line of Java code. Furthermore, since adding EJMM extensions with metrics definitions can be done purely from the tool's front-end, users can add new metrics without having to recompile an existing tool for individual use. This feature is something that is not common in current software metrics tools.

We hope that the contributions of this dissertation will facilitate further studies on M2DM in the context of Java development, and with them, further aid the Software Engineering community in its continued research on the subject of software quality.

[This page was intentionally left blank]

BIBLIOGRAPHY

[1] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[3] F. Brito e Abreu, "Using OCL to formalize object oriented metrics definitions," Technical Report ES007/2001, INESC, May 2001.

[4] N. E. Fenton, *Software metrics : a rigorous approach*. London: Chapman & Hall, 1991.

[5] F. G. Wilkie and T. J. Harmer, "Tool support for measuring complexity in heterogeneous object-oriented software," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 152–161, 2002.

[6] A. L. Baroni, *Formal Definition of Object-Oriented Design Metrics*. M.S. Thesis, Vrije Universiteit Brussel (VUB), Brussels, Belgium, August 2002.

[7] G. Antoniol, M. Di Penta, and E. Merlo, "YAAB (Yet Another AST Browser): using OCL to navigate ASTs," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC)*, pp. 13–22, 2003.

[8] J. A. McQuillan, *Using Model Driven Engineering to Reliably Automate the Measurement of Object-Oriented Software*. Ph.D. Dissertation, National University of Ireland, Maynooth, 2011.

[9] M. El Wakil, A. El Bastawissi, M. Boshra, and A. Fahmy, "A novel approach to formalize and collect object-oriented design metrics," in *Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering (ICEASE)*, 2005.

[10] A. L. Baroni and F. Brito e Abreu, "An OCL-based formalization of the MOOSE metrics suite," in *Proceedings of the International Workshop on Quantitative Ap-*

proaches in Object-Oriented Software Engineering (QAOOSE'2003) at ECOOP'2003, vol. 3013/2004, pp. 92–106, Springer, July 2003.

[11] G. de Montmollin, "The transparent language popularity index," 2013. Available: `http://lang-index.sourceforge.net/` Accessed: 2013-05-30.

[12] The Eclipse Foundation, "Eclipse - The Eclipse Foundation open source community website." 2013. Available: `http://www.eclipse.org` Accessed: 2013-05-30.

[13] The Eclipse Foundation, "Eclipse marketplace," 2013. Available: `http://marketplace.eclipse.org` Accessed: 2013-05-30.

[14] The Eclipse Foundation, "Eclipse Java Development Tools (JDT) overview," 2013. Available: `http://www.eclipse.org/jdt/overview.php` Accessed: 2013-05-30.

[15] F. Sauer, "Metrics 1.3.6.," 2013. Available: `http://metrics.sourceforge.net/` Accessed: 2013-05-30.

[16] A. L. Baroni and F. Brito e Abreu, "Formalizing object-oriented design metrics upon the UML meta-model," in *Actas do XVI Simpósio Brasileiro de Engenharia de Software (SBES)*, pp. 130–145, Biblioteca Digital Brasileira de Computação (BDBComp), Sociedade Brasileira de Computação, October 2002.

[17] A. L. Baroni, C. Calero, F. Brito e Abreu, and M. Piattini, "Object-relational database metrics formalization," in *6th International Conference on Quality Software (QSIC'2006)*, pp. 30–37, IEEE Computer Society Press, Oct 26-28 2006.

[18] M. Goulão, *Component-Based Software Engineering: a Quantitative Approach*. Ph.D. Dissertation, FCT/UNL, Caparica, Portugal, December 2008.

[19] L. Ferreira da Silva, *Assessment of IT Infrastructures: A Model Driven Approach*. M.S. Thesis, FCT/UNL, Caparica, Portugal, December 2008.

[20] S. Bryton, *Modularity Improvements with Aspect-Oriented Programming*. M.S. Thesis, FCT/UNL, Caparica, Portugal, July 2008.

[21] F. Brito e Abreu, R. de Bragança V da Porciúncula, J. Freitas, and J. Costa, "Definition and validation of metrics for itsm process models," in *2010 Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 79–88, 2010.

[22] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

[23] J. Cahill, J. M. Hogan, and R. Thomas, "The Java Metrics Reporter - an extensible tool for OO software analysis," in *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, pp. 507–516, 2002.

[24] H. Mei, T. Xie, and F. Yang, "A model-based approach to object-oriented software metrics," *Journal of Computer Science and Technology*, vol. 17, no. 6, pp. 757–769, 2002.

[25] M. Engelhardt, C. Hein, T. Ritter, and M. Wagner, "Generation of formal model metrics for MOF based domain specific languages," *Electronic Communications of the EASST*, vol. 24, 2009.

[26] C. Hein, M. Engelhardt, T. Ritter, and M. Wagner, "Metrino," 2013. Available: `http://www.modelbus.org/modelbus/index.php/metrino` Accessed: 2013-05-30.

[27] EclipseSource, "Yoxos," 2013. Available: `http://yoxos.eclipsesource.com/` Accessed: 2013-09-06.

[28] G. Boissier, "Metrics 1.3.8.," 2013. Available: `http://metrics2.sourceforge.net/` Accessed: 2013-09-06.

[29] Database Systems Group, University of Bremen, "Sourceforge.net: The UML-based specification environment," 2013. Available: `http://sourceforge.net/apps/mediawiki/useocl/` Accessed: 2013-05-30.

[30] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 27–34, 2007.

[31] The Object Management Group, "OMG object constraint language (OCL), Version 2.3.1." OMG Document Number: formal/2012-01-01 Available: `http://www.omg.org/spec/OCL/2.3.1` Accessed: 2013-09-29

[32] QUASAR, "Java facade and code generator for USE (UML-based specification environment)," 2013. Available: `http://code.google.com/p/j-use/` Accessed: 2013-05-30.

[33] T. Kuhn and O. Thomann, "Eclipse corner article: Abstract Syntax Tree," 2006. Available: `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html` Accessed: 2013-05-30.

[34] F. B. Abreu, L. M. Ochoa, and M. Goulão, "The GOODLY design language for MOOD2 metrics collection," in *3rd ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'1999)* (F. B. e. Abreu, H. Sarahoui, and H. Zuse, eds.), 1999.

[35] Oracle, "The Java™tutorials," 2013. Available: `http://docs.oracle.com/javase/tutorial/java/index.html` Accessed: 22-09-2013.

[36] E. Gamma and T. Eggenschwiler, "JHotDraw start page," 2013. Available: `http://www.jhotdraw.org/` Accessed: 13-08-2013.

[37] E. Puybaret, "Sweet Home 3D - draw floor plans and arrange furniture freely," 2013. Available: `http://www.sweethome3d.com/` Accessed: 15-09-2013.

APPENDIX

A. THE ECLIPSE JAVA METAMODEL APPENDIX

A.1 EJMM USE Specification

```
model EclipseJavaModel

--- Enumerations

enum JavaType {Primitive, ClassType, EnumType,
               InterfaceType, AnnotationType}

enum VisibilityType {Default, Public, Private, Protected}

enum PackageFragmentRootType {Folder, Jar, Zip}

-- Metaclasses

abstract class JavaElement
   attributes
     name : String
     handleIdentifier : String
end --JavaElement

class JavaProject < JavaElement
end --JavaProject

class PackageFragmentRoot < JavaElement
   attributes
      packageFragmentRootType : PackageFragmentRootType
end --PackageFragmentRoot
```

```
class PackageFragment <  JavaElement
end --PackageFragment

abstract class Annotatable
end

abstract class TypeRoot < JavaElement
end --TypeRoot

class ClassFile < TypeRoot
end --ClassFile

class CompilationUnit < TypeRoot, Annotatable
-- Annotations associated with a TypeRoot refer to their package
-- declaration
end --CompilationUnit

abstract class Member < JavaElement, Annotatable
end --Member

class Type < Member
   attributes
      visibility : VisibilityType
      javaType : JavaType
      isFinal : Boolean
      isAbstract : Boolean
      isStrictfp : Boolean
      isSynthetic : Boolean
      isDeprecated : Boolean
   operations
      getAllNestedTypes() : Set(Type) =
         self.nests
         ->union(self.nests.getAllNestedTypes())
         ->asSet
      getAllEnclosingTypes() : Set(Type) =
         if(self.nestedIn->isUndefined)
         then Set{}
```

```
         else Set{self.nestedIn}->
            union(self.nestedIn.getAllEnclosingTypes())->asSet
         endif
      getFullInterfaceTree() : Set(Type) =
         if(self.javaType = #InterfaceType)
         then Set{self}
            ->union(self.implements.getFullInterfaceTree())
            ->asSet
         else self.implements.getFullInterfaceTree()
            ->asSet
         endif
      getFullInheritanceTree() : Set(Type) =
         if(self.extends.oclIsUndefined)
         then Set{}
         else Set{self.extends}->
            union(self.extends.getFullInheritanceTree())->asSet
         endif
end --Type

class TypeParameter < JavaElement
end --TypeParameter

class Field < Member
   attributes
      key : String
      visibility : VisibilityType
      isStatic : Boolean
      isFinal : Boolean
      isVolatile : Boolean
      isTransient : Boolean
      isSynthetic : Boolean
      isDeprecated : Boolean
      arrayDimensions : Integer
end --Field

class Initializer < Member
   operations --
```

```
      getAllStatements() : OrderedSet(Statement) =
         body.getStatementAndChildren()
            ->sortedBy(startPosition)->asOrderedSet
end --Initializer

class Method  < Member
   attributes
      key : String
      shortKey : String
      visibility : VisibilityType
      isConstructor : Boolean
      isStatic : Boolean
      isFinal : Boolean
      isSynchronized : Boolean
      isNative : Boolean
      isAbstract : Boolean
      isStrictfp : Boolean
      isSynthetic : Boolean
      isDeprecated : Boolean
      isBridge : Boolean
      hasVarargs : Boolean
      returnTypeArrayDimensions: Integer
   operations
      getAllStatements() : OrderedSet(Statement) =
         if(optionalBody.isDefined)
         then optionalBody.getStatementAndChildren()
              ->sortedBy(startPosition)->asOrderedSet
         else OrderedSet{}
         endif
end --Method

class LocalVariable < JavaElement, Annotatable
   attributes
      arrayDimensions : Integer
end --LocalVariable

class Annotation < JavaElement
```

```
   attributes
      lineNumber : Integer
end --Annotation

class AnnotationValue
   attributes
      value : String
end --AnnotationValue

abstract class ASTNode
   attributes
      startPosition : Integer
      length : Integer
end --ASTNode

abstract class Comment < ASTNode
end --Comment

class LineComment < Comment
end --LineComment

class BlockComment < Comment
end --BlockComment

class Javadoc < Comment
end --Javadoc

abstract class Statement < ASTNode
   attributes
      conditionalOperatorCount : Integer
   operations
      getStatementAndChildren() : Set(Statement) = Set{}
end --Statement

class AssertStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
```

```
end --AssertStatement

class Block < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
         ->union(statements.getStatementAndChildren()->asSet)
end --Block

class BreakStatement < Statement
   attributes
      optionalLabel : String
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --BreakStatement

class ConstructorInvocation < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --ConstructorInvocation

class ContinueStatement < Statement
   attributes
      optionalLabel : String
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --ContinueStatement

class DoStatement < Statement
   attributes
      body : Statement
   operations
      getStatementAndChildren() : Set(Statement) =
         if(body.isDefined)
         then Set{self}->union(body.getStatementAndChildren())
         else Set{self}
         endif
end --DoStatement
```

```
class EmptyStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --EmptyStatement

class EnhancedForStatement < Statement
   attributes
      body : Statement
   operations
      getStatementAndChildren() : Set(Statement) =
         if(body.isDefined)
         then Set{self}->union(body.getStatementAndChildren())
         else Set{self}
         endif
end --EnhancedForStatement

class ExpressionStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --ExpressionStatement

class ForStatement < Statement
   attributes
      body : Statement
   operations
      getStatementAndChildren() : Set(Statement) =
         if(body.isDefined)
         then Set{self}->union(body.getStatementAndChildren())
         else Set{self}
         endif
end --ForStatement

class IfStatement < Statement
   attributes
      thenStatement : Statement
      optionalElseStatement : Statement
```

```
   operations
      getStatementAndChildren() : Set(Statement) =
         Set{self}->union(thenStatement.getStatementAndChildren())
         ->union(if(optionalElseStatement.isDefined)
                 then optionalElseStatement.getStatementAndChildren()
                 else Set{}
                 endif)
end --IfStatement

class LabeledStatement < Statement
   attributes
      body : Statement
   operations
      getStatementAndChildren() : Set(Statement) =
       if(body.isDefined)
       then Set{self}->union(body.getStatementAndChildren())
       else Set{self}
       endif
end --LabeledStatement

class ReturnStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --ReturnStatement

class SuperConstructorInvocation < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --SuperConstructorInvocation

class SwitchCase < Statement
   attributes
      isDefault : Boolean
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --SwitchCase
```

```
class SwitchStatement < Statement
   attributes
      statements : Set(Statement)
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
         ->union(statements.getStatementAndChildren()->asSet)
end --SwitchStatement

class SynchronizedStatement < Statement
   attributes
      body : Block
   operations
      getStatementAndChildren() : Set(Statement) =
        if(body.isDefined)
        then Set{self}->union(body.getStatementAndChildren())
        else Set{self}
        endif
end --SynchronizedStatement

class ThrowStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --ThrowStatement

class TryStatement < Statement
   attributes
      body : Block
      catchClauses : Set(CatchClause)
      optionalFinallyBody : Block
   operations
      getStatementAndChildren() : Set(Statement) =
         Set{self}
         ->union(if(body.isDefined)
                 then body.getStatementAndChildren()->asSet
                 else Set{}
                 endif)
         ->union(catchClauses.getStatementAndChildren()
```

```
                    ->asSet)
         ->union(if(optionalFinallyBody.isDefined)
                 then
                    optionalFinallyBody.getStatementAndChildren()
                    ->asSet
                 else Set{}
                 endif)
end --TryStatement

class TypeDeclarationStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
end --TypeDeclarationStatement

class VariableDeclarationStatement < Statement
   operations
      getStatementAndChildren() : Set(Statement) = Set{self}
 end --VariableDeclarationStatement

class WhileStatement < Statement
   attributes
      body : Statement
   operations
      getStatementAndChildren() : Set(Statement) =
         if(body.isDefined)
         then Set{self}->union(body.getStatementAndChildren())
         else Set{self}
         endif
end --WhileStatement

class CatchClause < Statement
-- CatchClause is now a Statement subclass to aid recursive functions
   attributes
      body : Block
   operations
      getStatementAndChildren() : Set(Statement) =
         if(body.isDefined)
```

```
         then Set{self}->union(body.getStatementAndChildren())
         else Set{self}
         endif
end --CatchClause

--- Meta-Associations ---------------------------------

aggregation A_JavaProject_PackageFragmentRoot between
   JavaProject [1]
   PackageFragmentRoot [*] role packageFragmentRoots
end

aggregation A_PackageFragmentRoot_PackageFragment between
   PackageFragmentRoot [1]
   PackageFragment[*] role packageFragments
end

aggregation A_PackageFragment_TypeRoot between
   PackageFragment [1]
   TypeRoot [*] role typeRoots
end

aggregation A_TypeRoot_Type between
   TypeRoot [1]
   Type [*] role types
end

composition A_Type_TypeParameter between
   Type [1]
   TypeParameter [*] role typeParameters ordered
end

association B_Type_TypeParameter between
   Type [*] role bounds
   TypeParameter [*] role boundsIn
end
```

```
aggregation A_Type_Method between
   Type [1]
   Method [*] role methods
end

association A_Method_Type between
   Method [*] role returningMethods
   Type [0..1] role returnType
end

association B_Method_Type between
   Method [*] role throwingMethods
   Type [*] role throws
end

association A_LocalVariable_Type between
   LocalVariable [*] role localVariablesWithType
   Type [1]
end

aggregation A_Annotatable_Annotation between
   Annotatable [1]
   Annotation [*] role annotations
end

association A_Type_Annotation between
   Type [1]
   Annotation [*] role annotationsUsed
end

composition A_Annotation_AnnotationValue between
   Annotation [1]
   AnnotationValue [*] role values
end

association A_Field_AnnotationValue between
   Field [1]
```

```
   AnnotationValue [*] role fieldValues
end

aggregation A_CompilationUnit_Comment between
   CompilationUnit [1] role location
   Comment [*] role comments
end

composition A_Type_Initializer between
   Type [1]
   Initializer [0..*] role initializers ordered
end

composition A_Type_Field between
   Type [1]
   Field [*] role fields
end

association A_Field_Type between
   Type [1] role fieldType
   Field [*] role fieldsWithType
end

composition A_Method_LocalVariable between
   Method [0..1]
   LocalVariable [*] role localVariables
end

aggregation B_Method_LocalVariable between
   Method [0..1] role parameterLocation
   LocalVariable [*] role parameters ordered
end

aggregation A_Initializer_LocalVariable between
   Initializer [0..1]
   LocalVariable [*] role localVariables
end
```

```
aggregation A_Initializer_Block between
   Initializer [0..1]
   Block [1] role body
end

aggregation A_Method_Block between
   Method [0..1]
   Block [0..1] role optionalBody
end

aggregation A_Block_Statement between
  Block [0..1]
   Statement [*] role statements ordered
end

association A_Type_Type between
   Type [*] role nests
   Type [0..1] role nestedIn
end --A_Type_Type

association B_Type_Type between
   Type [0..1] role extends
   Type [*] role extendedBy
end --B_Type_Type

association C_Type_Type between
   Type [0..*] role implements
   Type [*] role implementedBy
end --C_Type_Type

association A_Statement_Type between
   Statement [*] role dependee
   Type [*] role typeDependencies
end -- A_Statement_Type

association A_Statement_Field between
```

```
   Statement [*] role dependee
   Field [*] role fieldsAccessed
end -- A_Statement_Field

association A_Statement_Method between
   Statement [*] role dependee
   Method [*] role methodsCalled
end -- A_Statement_Method

aggregation A_CatchClause_LocalVariable between
   CatchClause [0..1]
   LocalVariable [1] role exceptionVariable
end -- A_CatchClause_LocalVariable

association A_VariableDeclarationStatement_LocalVariable between
   VariableDeclarationStatement [0..1] role declarationStatement
   LocalVariable [*] role localVariablesDeclared
end -- A_VariableDeclarationStatement_LocalVariable

association A_TypeDeclarationStatement_Type between
   TypeDeclarationStatement [0..1] role declarationStatement
   Type [1] role typeDeclared
end -- A_TypeDeclarationStatement_Type

 -------------------------- CONSTRAINTS -----------------------------
constraints

context LocalVariable
   inv localVariableExclusiveLocation:
   -- A LocalVariable instance can only be either a method parameter,
   -- a method local variable or an initializer variable.
      self.parameterLocation.isDefined() xor self.method.isDefined()
      xor self.initializer.isDefined()

context Block
   inv blockExclusiveLocation:
   -- If a block belongs to a method, it can not belong to an initializer.
```

```
-- If it belongs to a initializer, it can not belong to a method.
   not(self.method.isDefined() and self.initializer.isDefined())
```

B. OCL APPLICATIONS APPENDIX

B.1 FLAME for EJMM specification

***JavaElement* operations**

Operation:	*clients()*
Informal definition:	Set containing all direct clients of the *JavaElement*.
Definition:	`clients() : Set(JavaElement) =` `  oclUndefined(Set(JavaElement))`
Notes:	Taken from the *client* operation for the *ModelElement* metaclass. This operation is left undefined in the *JavaElement* abstract class. Concrete implementations can be found in metaclasses *Type*, *Field* and *Method*.

Operation:	*allClients()*
Informal definition:	Set containing all the *JavaElements* that are clients of this *JavaElement*, including the clients of these *JavaElements*. This is the transitive closure.

Definition:	`allClients() : Set(JavaElement) =`
	`  self.clients()->union(self.clients()->`
	`  collect(m : JavaElement |`
	`  m.allClients())->flatten)->asSet`

***PackageFragment* operations**

Operation:	*allClasses()*
Informal definition:	Set of all *Types* that are classes belonging to the current *PackageFragment*.

Definition:	`allClasses(): Set(Type) =`
	`  self.typeRoots.types`
	`  ->select(javaType = JavaType::ClassType)`
	`  ->asSet`

Operation:	*isInternal(c : Type)*
Informal definition:	True if the *Type* received as parameter belongs to the current *PackageFragment*.
Definition:	`isInternal(c : Type) : Boolean =` `  self.typeRoots.types->includes(c)`

Operation:	*internalBaseClasses()*
Informal definition:	Set of base classes in the current *PackageFragment*.

Definition:	`internalBaseClasses() : Set(Type) =`
	`  self.allClasses().parents()->`
	`  select(javaType = JavaType::ClassType)->asSet`

Operation:	*baseClassesInPackages(p : PackageFragment)*
Informal definition:	Set of base classes in both the current *PackageFragment* and the one bound to the parameter.

Definition:	`baseClassesInPackages(p : PackageFragment) :`
	`Set(Type) =`
	`  self.internalBaseClasses()->`
	`  union(p.internalBaseClasses())`

Operation:	*baseClasses(p : PackageFragment)*
Informal definition:	Set of base classes in the current *PackageFragment* that belong to the p *PackageFragment*.

Definition:	`baseClasses(p : PackageFragment) : Set (Type) =`
	`  self.internalBaseClasses()`
	`  ->select(c: Type | p.isInternal(c))`

Operation:	*internalSupplierClasses()*
Informal definition:	Set of supplier classes in the current *PackageFragment*.
Definition:	`internalSupplierClasses() : Set (Type) =` `  self.supplierClasses(self)`

Operation:	*supplierClassesInPackages(p : PackageFragment)*
Informal definition:	Set of supplier classes in both the current *PackageFragment* and the one bound to the parameter.

Definition:	`supplierClassesInPackages(p : PackageFragment) :`
	`Set(Type) =`
	`  self.internalSupplierClasses()->`
	`  union(p.internalSupplierClasses())`

Operation:	*supplierClasses(p : PackageFragment)*
Informal definition:	Set of supplier classes in the current *PackageFragment* that belong to the p *PackageFragment* (excludes inheritance).

Definition:	`supplierClasses(p : PackageFragment) : Set(Type) =`
	`  self.allClasses().coupledClasses()->`
	`  select(c: Type | p.isInternal(c))->asSet`

Operation:	*relatedClasses(p : PackageFragment)*
Informal definition:	Set of classes from the "p" *PackageFragment* that are either base or supplier classes.
Definition:	`relatedClasses(p : PackageFragment) : Set(Type) =` `  baseClasses(p)->union(supplierClasses(p))`

Operation:	*TC()* (Total Classes)
Informal definition:	Number of classes in the *PackageFragment*.
Definition:	`TC() : Integer = allClasses()->size()`

Operation:	*CN()* (Classes Number)
Informal definition:	Number of classes in the *PackageFragment* (replaces TC that was defined in the MOODLib).
Definition:	`CN() : Integer = TC()`

Operation:	*TON()* (Total Operations New)
Informal definition:	Total number of new *Methods* in the *PackageFragment*.
Definition:	`TON() : Integer = allClasses().NON()->sum`

Operation:	*TOO()* (Total Operations Overridden)
Informal definition:	Total number of overridden *Methods* in the *PackageFragment*.
Definition:	`TOO() : Integer = allClasses().OON()->sum`

Operation:	*TOD()* (Total Operations Defined)
Informal definition:	Total number of defined *Methods* in the *PackageFragment*.
Definition:	`TOD() : Integer = allClasses().DON()->sum`

Operation:	*TOI()* (Total Operations Inherited)
Informal definition:	Total number of inherited *Methods* in the *PackageFragment*.
Definition:	`TOI() : Integer = allClasses().ION()->sum`

Operation:	*TON()* (Total Operations Available)
Informal definition:	Total number of available *Methods* in the *PackageFragment*.
Definition:	`TOA() : Integer = allClasses().AON()->sum`

Operation:	*TAN()* (Total Attributes New)
Informal definition:	Total number of new *Fields* in the *PackageFragment*.
Definition:	`TAN() : Integer = allClasses().NAN()->sum`

Operation:	*TAO()* (Total Attributes Overridden)
Informal definition:	Total number of overridden *Fields* in the *PackageFragment*.
Definition:	`TAO() : Integer = allClasses().OAN()->sum`

Operation:	*TAD()* (Total Attributes Defined)
Informal definition:	Total number of defined *Fields* in the *PackageFragment*.
Definition:	`TAD() : Integer = allClasses().DAN()->sum`

Operation:	*TAI()* (Total Attributes Inherited)
Informal definition:	Total number of inherited *Fields* in the *PackageFragment*.
Definition:	`TAI() : Integer = allClasses().IAN()->sum`

Operation:	*TON()* (Total Attributes Available)
Informal definition:	Total number of available *Fields* in the *PackageFragment*.
Definition:	`TAA() : Integer = allClasses().AAN()->sum`

Operation:	*IL(p : PackageFragment)* (Inheritance Links)
Informal definition:	Total number of inheritance relations where the derived classes belongs to the current *PackageFragment* and the base one belongs to the given "p" *PackageFragment* (from MOODLib).

Definition:	`IL(p : PackageFragment): Integer =`
	`  allClasses().parents()`
	`  ->select(t: Type | p.isInternal(t))->size`

Operation:	*TIL()* (Total Inheritance Links)
Informal definition:	Total number of inheritance relations where the derived classes belongs to the current *PackageFragment*.
Definition:	`TIL() : Integer = allClasses().PARN()->sum`

Operation:	*CL(p : PackageFragment)* (Coupling Links)
Informal definition:	Total number of coupling relations where the client class belongs to the current *PackageFragment* and the supplier class belongs to the given "p" *PackageFragment* (excludes inheritance).
Definition:	`CL(p : PackageFragment) : Integer =` `  self.supplierClasses(p)->size()`

Operation:	*TCL()* (Total Coupling Links)
Informal definition:	Total number of distinct coupling relations where the client class belongs to the current *PackageFragment* (excludes inheritance).

Definition:	`TCL() : Integer =`
	`  allClasses()->collect(coupledClasses()->`
	`  select(c: Type | self.isInternal(c))->size())`
	`  ->sum`

Operation:	*AVN(a : Field)* (Attribute Visibility Number)
Informal definition:	Number of classes in the considered *PackageFragment* where the *Field* can be accessed.
Definition:	`AVN(a : Field) : Integer = FVN(a)`

Operation:	*APV(a : Field)* (Attribute to Package Visibility)
Informal definition:	Percentage of classes in the considered *PackageFragment* where the *Field* can be accessed (excludes the *Type* where the *Field* is declared).
Definition:	`APV(a : Field) : Real =` `  (self.AVN(a) -1) / (self.TC() -1)`

Operation:	*OVN(o : Method)* (Operation Visibility Number)
Informal definition:	Number of classes in the considered *PackageFragment* where the *Method* can be accessed.
Definition:	`OVN(o : Method) : Integer = FVN(o)`

Operation:	*OPV(o : Method)* (Operation to Package Visibility)
Informal definition:	Percentage of classes in the considered *PackageFragment* where the *Method* can be accessed (excludes the *Type* where the *Method* is declared).
Definition:	`OPV(o : Method) : Real =` `  (self.OVN(o) -1) / (self.TC() -1)`

Operation:	*FVN(f : Member)* (Feature Visibility Number)
Informal definition:	Number of classes in the considered *PackageFragment* where the *Member* can be accessed (*Member* is expected to be either a *Type*, *Field* or *Method*).
Definition:	`FVN(f : Member) : Integer =` `  self.allClasses()->select(FCV(f))->size`

Operation:	*FPV(f : Member)* (Feature to Package Visibility)
Informal definition:	Percentage of classes in the considered *PackageFragment* where the *Member* can be accessed (excludes the *Type* where the *Member* is declared).
Definition:	`FPV(f : Member) : Real =` `(self.FVN(f) -1) / (self.TC() -1)`

Operation:	*FUN(f : Member)* (Feature Use Number)
Informal definition:	Number of classes that use the *Member* (excludes the *Type* where the *Member* is declared).

Definition:	`FUN(f : Member) : Integer =`
	`self.allClasses()->`
	`select(allFeatures()->includes(f))->size() - 1`

Operation:	*PNAN()* (Package New Attributes Number)
Informal definition:	Number of new *Fields* in the *PackageFragment*.
Definition:	`PNAN() : Integer = TAN()`

Operation:	*PDAN()* (Package Defined Attributes Number)
Informal definition:	Number of defined *Fields* in the *PackageFragment*.
Definition:	`PDAN() : Integer = TAD()`

Operation:	*PIAN()* (Package Inherited Attributes Number)
Informal definition:	Number of inherited *Fields* in the *PackageFragment*.
Definition:	`PIAN() : Integer = TAI()`

Operation:	*POAN()* (Package Overridden Attributes Number)
Informal definition:	Number of overridden *Fields* in the *PackageFragment*.
Definition:	`POAN() : Integer = TAO()`

Operation:	*PAAN()* (Package Available Attributes Number)
Informal definition:	Number of available *Fields* in the *PackageFragment*.
Definition:	`PAAN() : Integer = TAA()`

Operation:	*PNON()* (Package New Operations Number)
Informal definition:	Number of new *Methods* in the *PackageFragment*.
Definition:	`PNON() : Integer = TON()`

Operation:	*PDON()* (Package Defined Operations Number)
Informal definition:	Number of defined *Methods* in the *PackageFragment*.
Definition:	`PDON() : Integer = TOD()`

Operation:	*PION()* (Package Inherited Methods Number)
Informal definition:	Number of inherited *Fields* in the *PackageFragment*.
Definition:	`PION() : Integer = TOI()`

Operation:	*POON()* (Package Overridden Operations Number)
Informal definition:	Number of overridden *Methods* in the *PackageFragment*.
Definition:	`POON() : Integer = TOO()`

Operation:	*PAON()* (Package Available Operations Number)
Informal definition:	Number of available *Methods* in the *PackageFragment*.
Definition:	`PAON() : Integer = TOA()`

Operation:	*EILN(p: PackageFragment)* (External Inheritance Links Number)
Informal definition:	Number of inheritance relations where the derived classes belong to the current *PackageFragment* and the base ones belong to the *PackageFragment* "p" given as parameter.
Definition:	`EILN(p: PackageFragment) : Integer = IL(p)`

Operation:	*IILN()* (Internal Inheritance Links Number)
Informal definition:	Number of inheritance relations where the base and derived classes belong to the current *PackageFragment*.
Definition:	`IILN() : Integer = TIL()`

Operation:	*ECLN(p: PackageFragment)* (External Coupling Links Number)
Informal definition:	Number of coupling relations where the client class belongs to the current *PackageFragment* and the supplier class belongs to the *PackageFragment* "p" (excludes inheritance).
Definition:	`ECLN(p: PackageFragment) : Integer = CL(p)`

Operation:	*ICLN()* (Internal Coupling Links Number)
Informal definition:	Number of distinct coupling relations where both the client and the supplier classes belong to the current *PackageFragment* (excludes inheritance).
Definition:	`ICLN() : Integer = TCL()`

Member operations

Operation:	*FUN()* (Feature Use Number)
Informal definition:	Number of classes that use the *Member*.
Definition:	`FUN() : Integer =` `  self.allClients()->size`

Type operations

Operation:	*suppliers()*
Informal definition:	Set of all other *Types* that the current *Type* depends on. Includes inheritance.

Definition:	`suppliers(): Set(Type) =`
	`  self.initializers.localVariables.type->asSet->`
	`  union((self.initializers.getAllStatements()`
	`  .typeDependencies)->asSet)->`
	`  union(self.typeParameters.bounds->asSet)->`
	`  union(self.ascendants())->`
	`  union(self.fields.fieldType)->`
	`  union(self.methods.returnType)->`
	`  union(self.methods.parameters.type)->`
	`  union(self.methods.localVariables.type)->`
	`  union(self.methods.getAllStatements()`
	`  .typeDependencies->asSet)->`
	`  union(self.annotations.type)->`
	`  union(self.initializers.annotations.type->asSet)->`
	`  union(self.fields.annotations.type)->`
	`  union(self.methods.annotations.type)`
	`  ->asSet->excluding(self)`

Notes:	This operation is not part of the original FLAME specification, but was defined here to help discern which are the supplier Types and, in turn, help find out the clients of a Type.

Operation:	*clients()*
Informal definition:	Set of all other *Types* that the depend on the current *Type*. Concrete implementation of the *JavaElement* operation.

Definition:	`clients(): Set(Type) =`
	`  Type.allInstances->select(suppliers()`
	`  ->includes(self))`
	`  ->excluding(self)`

Operation:	*isRoot()*
Informal definition:	True if the *Type* has no ascendants.
Definition:	`isRoot() : Boolean =` `  self.parents()->isEmpty()`

Operation:	*isLeaf()*
Informal definition:	True if the *Type* has no children.
Definition:	`isLeaf() : Boolean =` `  self.children()->isEmpty()`

Operation:	*children()*
Informal definition:	Set of directly derived *Types*.
Definition:	`children(): Set(Type) =` `  self.extendedBy->union(self.implementedBy)`

Operation:	*descendants()*
Informal definition:	Set of all derived *Types* (either directly or indirectly).

Definition:	`descendants() : Set (Type) =`
	`  children()->iterate(elem: Type;`
	`  acc: Set(Type)= children() |`
	`  acc->union(elem.descendants()))`

Operation:	*parents()*
Informal definition:	Set of *Types* from which the current *Type* derives directly.
Definition:	`parents(): Set(Type) =` `  self.extends->union(self.implements)`

Operation:	*parents()*
Informal definition:	Set of *Types* from which the current *Type* derives directly or indirectly.

Definition:	`ascendants() : Set (Type) =`
	`  parents()->iterate(elem: Type;`
	`  acc: Set(Type) = parents() |`
	`  acc->union(elem.ascendants()))`

Operation:	*CHIN() (Children Number)*
Informal definition:	Number of children of the current *Type*.
Definition:	`CHIN(): Integer = children()->size`

Operation:	*DESN() (Descendants Number)*
Informal definition:	Number of descendants of the current *Type*.
Definition:	`DESN() : Integer = descendants()->size()`

Operation:	*PARN() (Parents Number)*
Informal definition:	Number of parents of the current *Type*.
Definition:	`PARN() : Integer = parents()->size()`

Operation:	*ASCN() (Ascendants Number)*
Informal definition:	Number of ascendants of the current *Type*.
Definition:	`ASCN() : Integer = ascendants()->size()`

Operation:	*coupledClasses()*
Informal definition:	Set of *Types* to which the current *Type* is coupled (excluding inheritance).
Definition:	`coupledClasses(): Set(Type) =` `  self.suppliers() - self.ascendants()`

Operation:	*definedFeatures()*
Informal definition:	Set of features that belong only to this *Type*. Excludes inheritance.

Definition:	`definedFeatures() : Set(Member) =`
	`  self.fields->union(self.methods)->`
	`  union(self.nestedFeatures())`

Operation:	*nestedFeatures()*
Informal definition:	Set of features that belong to all *Types* nested in the current *Type*. Excludes inheritance.

Definition:	`nestedFeatures() : Set(Member) =`
	`  self.getAllNestedTypes()->`
	`  union(self.getAllNestedTypes().methods)->`
	`  union(self.getAllNestedTypes().fields)->asSet`

Notes:	Auxiliary operation not featured in the original FLAME specification.

Operation:	*FCV(m : Member)* (Feature to Classifier Visibility)
Informal definition:	True if the *Member* "m" is visible to the current *Type*

Definition:	`FCV (m : Member) : Boolean =`
	`  if (m.oclIsTypeOf(Method)) then`
	`  self.methods->includes(m) or`
	`  self.getAllNestedTypes().methods->includes(m) or`
	`  self.getAllEnclosingTypes().methods->includes(m) or`
	`  self.nestedIn.nests.methods->includes(m) or`
	`  m.oclAsType(Method).visibility = #Public or`
	`  m.oclAsType(Method).visibility = #Protected and`
	`  self.parents().methods->includes(m) or`
	`  m.oclAsType(Method).visibility = #Default and`
	`  m.oclAsType(Method).type.typeRoot.packageFragment =`
	`  self.typeRoot.packageFragment`
	`  else`
	`  if (m.oclIsTypeOf(Field)) then`
	`  self.fields->includes(m) or`
	`  self.getAllNestedTypes().fields->includes(m) or`
	`  self.getAllEnclosingTypes().fields->includes(m) or`
	`  self.nestedIn.nests.fields->includes(m) or`
	`  m.oclAsType(Field).visibility = #Public or`
	`  m.oclAsType(Field).visibility = #Protected and`
	`  self.parents().methods->includes(m) or`
	`  m.oclAsType(Field).visibility = #Default and`
	`  m.oclAsType(Field).type.typeRoot.packageFragment =`
	`  self.typeRoot.packageFragment`
	`  else`
	`  if (m.oclIsTypeOf(Type)) then`
	`  self.getAllNestedTypes()->includes(m) or`
	`  self.getAllNestedTypes()->includes(m) or`
	`  self.nestedIn.nests->includes(m) or`
	`  m.oclAsType(Type).visibility = #Public or`
	`  m.oclAsType(Type).visibility = #Protected and`
	`  self.parents().methods->includes(m) or`
	`  m.oclAsType(Type).visibility = #Default and`
	`  m.oclAsType(Type).typeRoot.packageFragment =`
	`  self.typeRoot.packageFragment`
	`  else`
	`  false`
	`  endif`
	`  endif`
	`  endif`

Operation:	*ACV(a : Field)* (Attribute to Classifier Visibility)
Informal definition:	True if the *Field* "a" is visible to the current *Type*
Definition:	`ACV(a : Field) : Boolean = FCV(a)`

Operation:	*OCV(o : Method)* (Operation to Classifier Visibility)
Informal definition:	True if the *Method* "o" is visible to the current *Type*
Definition:	`OCV(o : Method) : Boolean = FCV(o)`

Operation:	*directlyInheritedFeatures()*
Informal definition:	Set of directly inherited *Members*.

Definition:	`directlyInheritedFeatures() : Set(Member) =`
	`  self.parents().definedFeatures()->`
	`  select(m: Member | self.FCV(m))->asSet`
	`  ->reject(oclIsTypeOf(Method) and`
	`  oclAsType(Method).isConstructor)`

Operation:	*allInheritedFeatures()*
Informal definition:	Set containing all inherited *Members* (both directly and indirectly).

Definition:	`allInheritedFeatures() : Set(Member) =`
	`  self.ascendants().definedFeatures()`
	`  ->select(m: Member | self.FCV(m))->asSet`
	`  ->reject(oclIsTypeOf(Method) and`
	`  oclAsType(Method).isConstructor)`

Operation:	*allFeatures()*
Informal definition:	Set containing all *Members* of the *Type* itself and all its inherited *Members*.

Definition:	`allFeatures() : Set(Member) =`
	`  self.allInheritedFeatures()->`
	`  union(self.definedFeatures())->asSet`

Operation:	*newFeatures()*
Informal definition:	Set of *Members* declared in the current *Type*. This definition excludes inherited *Members* (and consequently, it excludes overridden *Methods*).

Definition:	`newFeatures() : Set (Member) =`
	`  definedAttributes()->`
	`  union(definedFeatures()->`
	`  select(oclIsTypeOf(Type)))->`
	`  union(definedOperations()->`
	`  reject(m1 : Method |`
	`  self.allInheritedOperations()`
	`  ->exists(m2: Method |`
	`  m1.shortKey = m2.shortKey and`
	`  m1.returnType = m2.returnType and`
	`  m1.returnTypeArrayDimensions =`
	`  m2.returnTypeArrayDimensions)))`

Operation:	*overriddenFeatures()*
Informal definition:	Set of redefined *Members* in the *Type*.
Definition:	`overriddenFeatures() : Set (Member) =` `  definedFeatures()-newFeatures()`
Notes:	In Java, only methods can be overridden, so this operation should only return *Method* instances.

Operation:	*definedOperations()*
Informal definition:	Set of *Methods* declared in the current *Type*.

Definition:	`definedOperations() : Set(Method) =`
	`  self.definedFeatures()->select(f |`
	`  f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*directlyInheritedOperations()*
Informal definition:	Set of directly inherited *Methods*.

Definition:	`directlyInheritedOperations() : Set(Method) =`
	`  self.directlyInheritedFeatures()->`
	`  select(f | f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*allInheritedOperations()*
Informal definition:	Set of all inherited *Methods* (directly or indirectly).

Definition:	`allInheritedOperations() : Set(Method) =`
	`  self.allInheritedFeatures()->`
	`  select(f | f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*newOperations()*
Informal definition:	Set of *Methods* declared in the current *Type*.

Definition:	`newOperations() : Set (Method) =`
	`  self.newFeatures()->`
	`  select(f | f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*overriddenOperations()*
Informal definition:	Set of redefined *Methods* in the *Type*.

Definition:	`overriddenOperations() : Set (Method) =`
	`  self.overriddenFeatures()`
	`  ->select(f | f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*allOperations()*
Informal definition:	Set containing all *Methods* of the *Type* itself and all its inherited *Methods*.

Definition:	`allOperations() : Set (Method) =`
	`  self.allFeatures()->select(f |`
	`  f.oclIsKindOf(Method))->`
	`  collect(f | f.oclAsType(Method))->asSet`

Operation:	*definedAttributes()*
Informal definition:	Set of *Fields* declared in the *Type*.

Definition:	`definedAttributes() : Set(Field) =`
	`  self.definedFeatures()->`
	`  select(f | f.oclIsKindOf(Field))->`
	`  collect(f | f.oclAsType(Field))->asSet`

Operation:	*directlyInheritedAttributes()*
Informal definition:	Set of directly inherited *Fields*.

Definition:	`directlyInheritedAttributes() : Set(Field) =`
	`  self.directlyInheritedFeatures()->`
	`  select(f | f.oclIsKindOf(Field))->`
	`  collect(f | f.oclAsType(Field))->asSet`

Operation:	*allInheritedAttributes()*
Informal definition:	Set of all inherited *Fields* (directly or indirectly).

Definition:	`allInheritedAttributes() : Set(Field) =`
	`  self.directlyInheritedFeatures()->`
	`  select(f | f.oclIsKindOf(Field))->`
	`  collect(f | f.oclAsType(Field))->asSet`

Operation:	*newAttributes()*
Informal definition:	Set of *Fields* declared in the current *Type*.
Definition:	`newAttributes() : Set (Field) =` `  self.definedAttributes()`

Operation:	*overriddenAttributes()*
Informal definition:	Set of redefined *Fields* in the *Type*.
Definition:	`overriddenAttributes() : Set (Field) = Set{}`
Notes:	There is no field overriding in Java, only field hiding (when two fields share the same name). Thus, this operation always returns an empty Set.

Operation:	*NON()* (New Operations Number)
Informal definition:	Number of new *Methods* in the *Type*.
Definition:	`NON() : Integer = newOperations()->size()`

Operation:	*ION()* (Inherited Operations Number)
Informal definition:	Number of inherited *Methods* in the *Type*.
Definition:	`ION() : Integer =` `  allInheritedOperations()->size()`

Operation:	*OON()* (Overridden Operations Number)
Informal definition:	Number of overridden *Methods* in the *Type*.
Definition:	`OON() : Integer =` `  overriddenOperations()->size()`

Operation:	*DON()* (Defined Operations Number)
Informal definition:	Number of defined *Methods* in the *Type*.
Definition:	`DON() : Integer =` `  definedOperations()->size()`

Operation:	*AON()* (Available Operations Number)
Informal definition:	Number of available *Methods* in the *Type*.
Definition:	`AON() : Integer = allOperations()->size()`

Operation:	*NAN()* (New Attribute Number)
Informal definition:	Number of new *Fields* belonging to the *Type*.
Definition:	`NAN() : Integer = newAttributes()->size()`

Operation:	*IAN()* (Inherited Attributes Number)
Informal definition:	Number of inherited *Fields* in the *Type*.
Definition:	`IAN() : Integer =` `  allInheritedAttributes()->size()`

Operation:	*OAN()* (Overridden Attributes Number)
Informal definition:	Number of overridden *Fields* in the *Type*.
Definition:	`OAN() : Integer =` `  overriddenAttributes()->size()`

Operation:	*DAN()* (Defined Attributes Number)
Informal definition:	Number of defined *Fields* in the *Type*.
Definition:	`DAN() : Integer =` `  definedAttributes()->size()`

Operation:	*AAN()* (Available Operations Number)
Informal definition:	Number of all *Fields* in the *Type*.
Definition:	`AAN() : Integer = allAttributes()->size()`

Field operations

Operation:	*AUN()* (Feature Use Number)
Informal definition:	Number of *Types* that use the *Field* (excludes the *Type* where the *Field* is declared).
Definition:	`AUN() : Integer = self.FUN()`

Operation:	*clients()*
Informal definition:	Set of all *Types* that the depend on the current *Field* (excludes the *Type* where the *Field* is declared). Concrete implementation of the *JavaElement* operation.

Definition:	`clients(): Set(Type) =`
	`  Type.allInstances->`
	`  select(methods.getAllStatements()->`
	`  union(initializers.getAllStatements()`
	`  ->asSet).fieldsAccessed->includes(self))`
	`  ->excluding(self.type)`

Method operations

Operation:	*OUN()* (Feature Use Number)
Informal definition:	Number of *Types* that use the *Method* (excludes the *Type* where the *Method* is declared).
Definition:	`OUN() : Integer = self.FUN()`

Operation:	*clients()*
Informal definition:	Set of all *Types* that the depend on the current *Method* (excludes the *Type* where the *Method* is declared). Concrete implementation of the *JavaElement* operation.

Definition:	`clients(): Set(Type) =`
	`  Type.allInstances->`
	`  select(methods.getAllStatements()->`
	`  union(initializers.getAllStatements()->`
	`  asSet).methodsCalled->includes(self))`
	`  ->excluding(self.type)`

C. TOOL ARCHITECTURE APPENDIX

C.1 Common Meta-objects

Name:	BasicTypes
Type:	*PackageFragment*
Description:	Place-holder *PackageFragment* to house all primitive and other basic types.
Attributes:	*name* = ”BasicTypesPackage”
Meta-associations:	*TypeRoots* = *BasicTypesClassFile*

Name:	ExternalTypes
Type:	*PackageFragment*
Description:	Place-holder *PackageFragment* to house all external types found during the second processing moment.
Attributes:	*name* = ”ExternalTypesPackage”
Meta-associations:	*TypeRoots* = *ExternalTypesClassFile*

Name:	BasicTypesClassFile
Type:	*ClassFile*
Description:	Place-holder *TypeRoot* to house all primitive and other basic types.
Attributes:	*name* = ”BasicTypes”
Meta-associations:	*packageFrament* = *BasicTypes*

Name:	ExternalTypesClassFile
Type:	*ClassFile*
Description:	Place-holder *TypeRoot* to house all external types found during the second processing moment.
Attributes:	*name* = ”ExternalTypes”
Meta-associations:	*packageFrament* = *ExternalTypes*

Name:	int
Type:	*Type*
Description:	Meta-object representing the *int* primitive type in Java.
Attributes:	*name* = ”int” *handleIdentifier* = ”int” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	boolean
Type:	*Type*
Description:	Meta-object representing the *boolean* primitive type in Java.
Attributes:	*name* = ”boolean” *handleIdentifier* = ”boolean” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	long
Type:	*Type*
Description:	Meta-object representing the *long* primitive type in Java.
Attributes:	*name* = ”long” *handleIdentifier* = ”long” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	double
Type:	*Type*
Description:	Meta-object representing the *double* primitive type in Java.
Attributes:	*name* = ”double” *handleIdentifier* = ”double” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	float
Type:	*Type*
Description:	Meta-object representing the *float* primitive type in Java.
Attributes:	*name* = ”float” *handleIdentifier* = ”float” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	byte
Type:	*Type*
Description:	Meta-object representing the *byte* primitive type in Java.
Attributes:	*name* = ”byte” *handleIdentifier* = ”byte” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	short
Type:	*Type*
Description:	Meta-object representing the *short* primitive type in Java.
Attributes:	*name* = ”short” *handleIdentifier* = ”short” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	char
Type:	*char*
Description:	Meta-object representing the *char* primitive type in Java.
Attributes:	*name* = ”char” *handleIdentifier* = ”char” *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	void
Type:	*Type*
Description:	Meta-object to represent the *void* keyword for method return types. Though not a typical primitive type, it is used to simplify the instantiation process and is an alternative to leaving a *Method*'s *returnType* undefined.
Attributes:	*name* = "void" *handleIdentifier* = "void" *javaType* = *JavaType.Primitive*
Meta-associations:	*typeRoot* = *BasicTypes*

Name:	String
Type:	*Type*
Description:	Meta-object representing the *String* class. Considered a basic type.
Attributes:	*name* = "String" *javaType* = *JavaType.ClassType*
Meta-associations:	*typeRoot* = *BasicTypes*

***Statement* type**	**Expressions obtained with**
AssertStatement	*getExpression()*
ConstructorInvocation	*arguments()*
EnhancedForStatement	*getExpression()*
ForStatement	*getExpression()* *initializers()* *updaters()*
IfStatement	*getExpression()*
ReturnStatement	*getExpression()*
SuperConstructorInvocation	*getExpression()* *arguments()*
SwitchCase	*getExpression()*
SwitchStatement	*getExpression()*
SynchronizedStatement	*getExpression()*
ThrowStatement	*getExpression()*
VariableDeclarationStatement	*getInitializer()*∗
WhileStatement	*getExpression()*

∗*getInitializer()* is a method belonging to the *VariableDeclarationFragment* class obtained from the *VariableDeclarationStatement* method *fragments()*

Tab. C.1: *Statements* that include *Expressions*

C.2 Tool Scalability Validation

J-USE		
1st moment duration (s)	**2nd moment duration (s)**	**Total duration (s)**
0.138567358	0.747325717	0.885893075
0.044774942	0.676637136	0.721412078
0.045630071	0.626382202	0.672012273
0.082243229	0.624962108	0.707205337
0.044419577	0.640152674	0.684572251
0.044239676	0.618268551	0.662508227
0.050511644	0.626959797	0.677471441
0.045018338	0.662793953	0.707812291
0.046376303	0.649754693	0.696130996
0.045171612	0.652569278	0.69774089
0.044212025	0.655792139	0.700004164
0.045867663	0.63377932	0.679646983
0.045503423	0.65344523	0.698948653
0.044734661	0.712570629	0.75730529
0.050262445	0.667886832	0.718149277
0.044477611	0.65629122	0.700768831
0.044201102	0.623381912	0.667583014
0.044318191	0.633928498	0.678246689
0.044880767	0.657366873	0.70224764
0.044344135	0.624628591	0.668972726
0.043887383	0.641893996	0.685781379
0.043860416	0.650841952	0.694702368
0.044928559	0.623711333	0.668639892
0.044299758	0.641901847	0.686201605
0.044128049	0.621285226	0.665413275
0.049457497	0.658578049	0.708035546
0.045541315	0.68301322	0.728554535
0.043660032	0.615574469	0.659234501
0.043623164	0.638185707	0.681808871
0.043559328	0.641914819	0.685474147
0.045773104	0.64944985	0.695222954
0.043640233	0.64434673	0.687986963
0.044211001	0.614166323	0.658377324
0.098327157	0.62506998	0.723397137
0.044265621	0.654502789	0.69876841
0.067628552	0.636491495	0.704120047
0.049494365	0.676276992	0.725771357
0.043762785	0.655919811	0.699682596
0.043719089	0.643672868	0.687391957
0.046204253	0.651571798	0.697776051
0.044904662	0.662871444	0.707776106
0.043355532	0.637072162	0.680427694
0.045533463	0.646083955	0.691617418
0.044893056	0.651763647	0.696656703
0.053882318	0.642580488	0.696462806
0.043948148	0.660547064	0.704495212
0.044378955	0.631737936	0.676116891
0.044103471	0.64862681	0.692730281
0.045034041	0.661765409	0.70679945
0.043761078	0.647218664	0.690979742

Tab. C.2: Scalability tests: J-USE

Eclipse Metrics 1.3.6		
1st moment duration (s)	**2nd moment duration (s)**	**Total duration (s)**
0.729034842	4.660256218	5.38929106
0.386690853	3.835194362	4.221885215
0.361328866	3.81258244	4.173911306
0.384242557	4.312641839	4.696884396
0.371381834	4.163232952	4.534614786
0.38931871	3.745352607	4.134671317
0.358372955	3.784383679	4.142756634
0.361059868	3.775033932	4.1360938
0.363045268	3.841220543	4.204265811
0.380042357	4.244929992	4.624972349
0.341714845	3.848697883	4.190412728
0.36204813	3.828271064	4.190319194
0.361110732	3.836746565	4.197857297
0.374429914	3.773210682	4.147640596
0.372920724	3.785919155	4.158839879
0.360204398	4.323948311	4.684152709
0.341420244	3.828319196	4.16973944
0.363839291	3.810124245	4.173963536
0.389480177	3.800325256	4.189805433
0.465910922	4.008803216	4.474714138
0.36434998	3.904086661	4.268436641
0.366787011	4.434043471	4.800830482
0.372080274	3.819730359	4.191810633
0.356429201	3.85831185	4.214741051
0.366725564	3.822981214	4.189706778
0.364029093	3.895570195	4.259599288
0.394780951	4.058720878	4.453501829
0.368714037	4.293776439	4.662490476
0.352417778	3.771928501	4.124346279
0.368341945	3.913246607	4.281588552
0.367487499	3.773473878	4.140961377
0.357664274	3.855779235	4.213443509
0.373382254	3.863464469	4.236846723
0.377946695	4.696512987	5.074459682
0.367882122	3.737474499	4.105356621
0.357796383	3.898219216	4.256015599
0.378381939	3.818801155	4.197183094
0.36066388	3.789388145	4.150052025
0.371884669	3.822767175	4.194651844
0.375795389	4.232449211	4.6082446
0.332035336	3.755353344	4.08738868
0.360139879	3.831950676	4.192090555
0.365662201	3.782180485	4.147842686
0.371237434	3.784739044	4.155976478
0.358887398	3.928731774	4.287619172
0.365423242	4.640639807	5.006063049
0.38696429	3.925762891	4.312727181
0.397991523	4.023475925	4.421467448
0.363535474	3.815715522	4.179250996
0.357204108	3.819455558	4.176659666

Tab. C.3: Scalability tests: Eclipse Metrics

JHotDraw 6.0		
1st moment duration (s)	**2nd moment duration (s)**	**Total duration (s)**
5.885808416	11.02789119	16.9136996
3.751500998	9.817519315	13.56902031
3.785597245	9.847677532	13.63327478
3.512636786	11.73312512	15.24576191
3.961988596	9.977678583	13.93966718
3.659689887	9.980969036	13.64065892
4.052728492	9.979217474	14.03194597
3.526514107	9.998311249	13.52482536
3.533637789	10.02182234	13.55546012
4.20936859	9.628041937	13.83741053
3.454652923	9.594333143	13.04898607
3.473705052	9.617337638	13.09104269
4.185685796	9.703233512	13.88891931
3.482400395	9.575655838	13.05805623
3.456480271	9.885988999	13.34246927
3.897244267	9.829931139	13.72717541
3.427165233	10.36703587	13.7942011
3.520988713	10.51192947	14.03291818
3.455980506	10.35171695	13.80769745
3.300358369	9.500361168	12.80071954
3.722493535	9.558759111	13.28125265
3.539852064	9.846574912	13.38642698
3.829809954	9.861241134	13.69105109
3.402032646	9.793565473	13.19559812
3.858543639	9.485217029	13.34376067
3.444650479	10.2068015	13.65145198
3.518259129	10.26122113	13.77948026
3.4913603	10.34467007	13.83603037
3.4101323	10.17569018	13.58582248
3.400442551	10.16922944	13.56967199
3.405124422	10.18165048	13.5867749
3.385572871	10.34631103	13.7318839
3.384354185	10.79985997	14.18421416
3.496614646	10.4241134	13.92072804
3.398760627	10.2662314	13.66499203
3.428950592	10.49721511	13.92616571
3.446044287	10.59339779	14.03944208
3.538236707	10.24275104	13.78098774
3.504884303	10.2740443	13.77892861
3.496744367	10.30361264	13.800357
3.453369036	10.29146606	13.74483509
3.452069445	10.38268523	13.83475468
3.457159936	10.3778139	13.83497384
3.454208462	10.47050233	13.92471079
3.464064458	10.66587651	14.12994097
3.368010135	10.49639242	13.86440255
3.429178967	10.39494686	13.82412582
3.44457333	10.24276435	13.68733768
3.604434926	10.86804769	14.47248261
3.430005079	10.67380617	14.10381125

Tab. C.4: Scalability tests: JHotDraw

SweetHome3D 4.1		
1st moment duration (s)	**2nd moment duration (s)**	**Total duration (s)**
15.24985799	36.9863217	52.23617969
7.988581899	35.73407822	43.72266012
7.758661548	35.73088165	43.48954319
8.330309717	37.04797323	45.37828294
8.565748636	35.99900491	44.56475355
7.966747616	36.42630942	44.39305704
7.811845425	35.8156848	43.62753022
7.708633624	35.85355112	43.56218475
7.852815472	36.00495975	43.85777522
8.112502834	35.43353893	43.54604176
7.918747422	36.4442958	44.36304322
8.121456252	36.05953607	44.18099232
7.99357749	36.54834429	44.54192178
8.045332398	35.96242418	44.00775658
8.041637764	35.73357027	43.77520803
8.108896615	36.28056684	44.38946345
8.149192458	35.86342419	44.01261665
8.147118984	35.7896551	43.93677408
8.094457402	35.93821909	44.0326765
8.25824542	36.1159469	44.37419232
8.084708594	36.00179048	44.08649907
8.176627578	36.05131386	44.22794144
8.328279256	36.18350035	44.51177961
8.710531634	36.46780859	45.17834023
8.84477624	36.56703082	45.41180706
9.04061673	36.99142857	46.0320453
9.034634244	37.00179765	46.03643189
9.646755086	37.67746496	47.32422004
9.756266164	37.55518191	47.31144807
10.26926548	37.57252754	47.84179302
7.777358653	35.43811873	43.21547738
7.857042641	35.35602161	43.21306425
7.684487592	35.49264043	43.17712802
7.575759955	35.5039213	43.07968126
7.680430084	35.31191677	42.99234686
7.861408405	35.51470241	43.37611081
7.847577511	35.52369099	43.3712685
7.755314086	36.52926349	44.28457758
7.698288446	35.39947818	43.09776663
8.586519896	36.00521065	44.59173055
7.747012	35.18864205	42.93565405
7.57583335	35.65385692	43.22969027
7.60464555	36.15068561	43.75533116
8.190426383	35.61180166	43.80222805
7.73609537	35.85609807	43.59219344
7.600362055	35.50194034	43.1023024
7.619077253	35.10624589	42.72532314
7.641315717	35.72861564	43.36993136
7.73646678	35.64593	43.38239678
7.590292021	35.36619986	42.95649188

Tab. C.5: Scalability tests: SweetHome3D

USE 3.0.6		
1st moment duration (s)	**2nd moment duration (s)**	**Total duration (s)**
22.52295534	71.13184205	93.65479739
22.57525234	70.88761869	93.46287103
22.63305698	71.07171064	93.70476762
22.62291868	70.76623515	93.38915383
22.476252	70.84026335	93.31651535
22.38949929	70.79991698	93.18941627
22.70859505	70.9646936	93.67328865
22.32469796	70.84448062	93.16917857
22.40361523	70.81773574	93.22135097
22.52322365	70.85377847	93.37700213
22.56906981	70.72920894	93.29827875
22.57133991	70.67656716	93.24790707
22.22832241	70.78960389	93.0179263
22.68491294	70.69541549	93.38032842
22.69659014	70.60033816	93.2969283
22.83829433	71.49382054	94.33211487
22.44942179	70.65547125	93.10489304
22.54991493	70.82724832	93.37716325
22.33390262	70.8475601	93.18146272
22.50948561	70.71844285	93.22792847
22.39726236	70.94582581	93.34308817
22.52500082	70.80430698	93.3293078
22.42894581	70.68437016	93.11331597
22.56192633	70.58295259	93.14487892
22.30305655	70.73348936	93.03654591
22.65078665	71.38398653	94.03477318
22.48590523	71.8665856	94.35249083
22.38586201	73.15895321	95.54481522
24.0711607	72.17015625	96.24131695
22.68207958	71.3058882	93.98796778
22.47448269	71.2801746	93.75465729
22.63282042	71.11198088	93.7448013
22.32585656	70.72913725	93.05499382
22.29854229	71.18929441	93.48783669
22.27113892	71.05448893	93.32562785
22.64930682	70.77174177	93.42104859
22.37720866	70.77063267	93.14784132
22.55761109	70.63896266	93.19657375
22.45928496	70.83882755	93.2981125
22.47959059	70.74091311	93.2205037
22.21703505	70.75321433	92.97024938
22.31336213	70.8846068	93.19796893
22.27584468	70.92372765	93.19957233
22.59162438	70.78329574	93.37492012
22.29911169	71.14699439	93.44610608
22.54624249	70.90636188	93.45260437
22.31895306	70.7785176	93.09747066
22.47358796	70.82401044	93.2975984
22.34896142	70.90954583	93.25850725
22.44972322	70.97058801	93.42031123

Tab. C.6: Scalability tests: USE

C.3 Tool Accuracy Validation

Package name
org.jhotdraw.test
org.jhotdraw.test.contrib
org.jhotdraw.test.figures
org.jhotdraw.test.framework
org.jhotdraw.test.samples.javadraw
org.jhotdraw.test.samples.minimap
org.jhotdraw.test.samples.net
org.jhotdraw.test.samples.nothing
org.jhotdraw.test.samples.pert
org.jhotdraw.test.standard
org.jhotdraw.test.util
org.jhotdraw.test.util.collections.jdk11
org.jhotdraw.test.util.collections.jdk12

Tab. C.7: Excluded packages

Source	Method name	EM value	M2DM value	Recalculated value
AbstractCommand.java	viewSelectionChanged	8	8	
AbstractCommand.java	isExecutable	4	4	
AbstractCommand.java	addCommandListener	2	2	
AbstractCommand.java	fireCommandExecutableEvent	2	2	
AbstractCommand.java	fireCommandExecutedEvent	2	2	
AbstractCommand.java	fireCommandNotExecutableEvent	2	2	
AbstractCommand.java	removeCommandListener	2	2	
AbstractCommand.java	dispose	2	2	
AbstractCommand.java	execute	2	2	
AbstractCommand.java	EventDispatcher	1	1	
AbstractCommand.java	AbstractCommand	1	1	
AbstractCommand.java	AbstractCommand	1	1	
AbstractCommand.java	addCommandListener	1	1	
AbstractCommand.java	createEventDispatcher	1	1	
AbstractCommand.java	createViewChangeListener	1	1	
AbstractCommand.java	figureSelectionChanged	1	1	
AbstractCommand.java	getDrawingEditor	1	1	
AbstractCommand.java	getEventDispatcher	1	1	
AbstractCommand.java	getUndoActivity	1	1	
AbstractCommand.java	isExecutableWithView	1	1	
AbstractCommand.java	isViewRequired	1	1	
AbstractCommand.java	name	1	1	
AbstractCommand.java	removeCommandListener	1	1	
AbstractCommand.java	setDrawingEditor	1	1	
AbstractCommand.java	setEventDispatcher	1	1	
AbstractCommand.java	setName	1	1	
AbstractCommand.java	setUndoActivity	1	1	
AbstractCommand.java	view	1	1	
AbstractCommand.java	viewCreated	1	1	
AbstractCommand.java	viewDestroying	1	1	
AbstractConnector.java	AbstractConnector	1	1	
AbstractConnector.java	AbstractConnector	1	1	
AbstractConnector.java	connectorVisibility	1	1	
AbstractConnector.java	containsPoint	1	1	
AbstractConnector.java	displayBox	1	1	
AbstractConnector.java	draw	1	1	
AbstractConnector.java	findEnd	1	1	
AbstractConnector.java	findPoint	1	1	
AbstractConnector.java	findStart	1	1	
AbstractConnector.java	owner	1	1	
AbstractConnector.java	read	1	1	
AbstractConnector.java	write	1	1	
AbstractContentProducer.java	AbstractContentProducer	1	1	
AbstractContentProducer.java	read	1	1	
AbstractContentProducer.java	write	1	1	
AbstractFigure.java	clone	4	4	
AbstractFigure.java	visit	4	4	
AbstractFigure.java	changed	2	2	
AbstractFigure.java	findFigureInside	2	2	
AbstractFigure.java	invalidate	2	2	
AbstractFigure.java	release	2	2	
AbstractFigure.java	AbstractFigure	1	1	
AbstractFigure.java	addDependendFigure	1	1	
AbstractFigure.java	addFigureChangeListener	1	1	
AbstractFigure.java	addToContainer	1	1	
AbstractFigure.java	basicMoveBy	1	1	
AbstractFigure.java	canConnect	1	1	
AbstractFigure.java	center	1	1	
AbstractFigure.java	connectedTextLocator	1	1	
AbstractFigure.java	connectionInsets	1	1	
AbstractFigure.java	connectorAt	1	1	
AbstractFigure.java	connectorVisibility	1	1	
AbstractFigure.java	containsPoint	1	1	
AbstractFigure.java	decompose	1	1	
AbstractFigure.java	displayBox	1	1	
AbstractFigure.java	displayBox	1	1	
AbstractFigure.java	figures	1	1	
AbstractFigure.java	getAttribute	1	1	
AbstractFigure.java	getAttribute	1	1	
AbstractFigure.java	getDecoratedFigure	1	1	
AbstractFigure.java	getDependendFigures	1	1	
AbstractFigure.java	getTextHolder	1	1	
AbstractFigure.java	getZValue	1	1	
AbstractFigure.java	includes	1	1	
AbstractFigure.java	invalidateRectangle	1	1	
AbstractFigure.java	isEmpty	1	2	1
AbstractFigure.java	listener	1	1	
AbstractFigure.java	moveBy	1	1	
AbstractFigure.java	read	1	1	
AbstractFigure.java	removeDependendFigure	1	1	
AbstractFigure.java	removeFigureChangeListener	1	1	
AbstractFigure.java	removeFromContainer	1	1	
AbstractFigure.java	setAttribute	1	1	
AbstractFigure.java	setAttribute	1	1	
AbstractFigure.java	setZValue	1	1	
AbstractFigure.java	size	1	1	
AbstractFigure.java	willChange	1	1	
AbstractFigure.java	write	1	1	

Tab. C.8: M2DM to Eclipse Metrics comparison: *AbstractCommand.java* to *AbstractFigure.java*

Source	Method name	EM value	M2DM value	Recalculated value
AbstractHandle.java	AbstractHandle	1	1	
AbstractHandle.java	containsPoint	1	1	
AbstractHandle.java	displayBox	1	1	
AbstractHandle.java	draw	1	1	
AbstractHandle.java	getCursor	1	1	
AbstractHandle.java	getUndoActivity	1	1	
AbstractHandle.java	invokeEnd	1	1	
AbstractHandle.java	invokeEnd	1	1	
AbstractHandle.java	invokeStart	1	1	
AbstractHandle.java	invokeStart	1	1	
AbstractHandle.java	invokeStep	1	1	
AbstractHandle.java	invokeStep	1	1	
AbstractHandle.java	owner	1	1	
AbstractHandle.java	setUndoActivity	1	1	
AbstractLineDecoration.java	read	4	4	
AbstractLineDecoration.java	draw	3	3	
AbstractLineDecoration.java	write	3	3	
AbstractLineDecoration.java	displayBox	2	2	
AbstractLineDecoration.java	AbstractLineDecoration	1	1	
AbstractLineDecoration.java	getBorderColor	1	1	
AbstractLineDecoration.java	getFillColor	1	1	
AbstractLineDecoration.java	setBorderColor	1	1	
AbstractLineDecoration.java	setFillColor	1	1	
AbstractLocator.java	clone	2	2	
AbstractLocator.java	AbstractLocator	1	1	
AbstractLocator.java	read	1	1	
AbstractLocator.java	write	1	1	
AbstractTool.java	checkUsable	3	2	3
AbstractTool.java	deactivate	3	3	
AbstractTool.java	setEnabled	3	3	
AbstractTool.java	setUsable	3	3	
AbstractTool.java	activate	2	2	
AbstractTool.java	viewSelectionChanged	2	2	
AbstractTool.java	AbstractTool	1	1	
AbstractTool.java	addToolListener	1	1	
AbstractTool.java	createEventDispatcher	1	1	
AbstractTool.java	createViewChangeListener	1	1	
AbstractTool.java	drawing	1	1	
AbstractTool.java	editor	1	1	
AbstractTool.java	getActiveDrawing	1	1	
AbstractTool.java	getActiveView	1	1	
AbstractTool.java	getAnchorX	1	1	
AbstractTool.java	getAnchorY	1	1	
AbstractTool.java	getEventDispatcher	1	1	
AbstractTool.java	getUndoActivity	1	1	
AbstractTool.java	isActive	1	2	1
AbstractTool.java	isEnabled	1	1	
AbstractTool.java	isUsable	1	2	1
AbstractTool.java	keyDown	1	1	
AbstractTool.java	mouseDown	1	1	
AbstractTool.java	mouseDrag	1	1	
AbstractTool.java	mouseMove	1	1	
AbstractTool.java	mouseUp	1	1	
AbstractTool.java	removeToolListener	1	1	
AbstractTool.java	setAnchorX	1	1	
AbstractTool.java	setAnchorY	1	1	
AbstractTool.java	setEditor	1	1	
AbstractTool.java	setEventDispatcher	1	1	
AbstractTool.java	setUndoActivity	1	1	
AbstractTool.java	setView	1	1	
AbstractTool.java	view	1	1	
AbstractTool.java	viewCreated	1	1	
AbstractTool.java	viewDestroying	1	1	
ActionTool.java	mouseDown	2	2	
ActionTool.java	ActionTool	1	1	
ActionTool.java	mouseUp	1	1	

Tab. C.9: M2DM to Eclipse Metrics comparison: *AbstractHandle.java* to *ActionTool.java*

Source	Method name	EM value	M2DM value	Recalculated value
AlignCommand.java	AlignCommand	1	1	
AlignCommand.java	createUndoActivity	1	1	
AlignCommand.java	execute	1	1	
AlignCommand.java	getAlignment	1	1	
AlignCommand.java	isExecutableWithView	1	1	
AlignCommand.java	setAlignment	1	1	
AnimationDecorator.java	animationStep	9	9	
AnimationDecorator.java	AnimationDecorator	1	1	
AnimationDecorator.java	AnimationDecorator	1	1	
AnimationDecorator.java	basicDisplayBox	1	1	
AnimationDecorator.java	basicMoveBy	1	1	
AnimationDecorator.java	displayBox	1	1	
AnimationDecorator.java	read	1	1	
AnimationDecorator.java	velocity	1	1	
AnimationDecorator.java	velocity	1	1	
AnimationDecorator.java	write	1	1	
Animator.java	run	3	3	
Animator.java	Animator	1	1	
Animator.java	end	1	1	
Animator.java	start	1	1	
AreaTracker.java	AreaTracker	1	1	
AreaTracker.java	drawXORRect	1	1	
AreaTracker.java	eraseRubberBand	1	1	
AreaTracker.java	getArea	1	1	
AreaTracker.java	mouseDown	1	1	
AreaTracker.java	mouseDrag	1	1	
AreaTracker.java	mouseUp	1	1	
AreaTracker.java	rubberBand	1	1	
ArrowTip.java	ArrowTip	1	1	
ArrowTip.java	ArrowTip	1	1	
ArrowTip.java	addPointRelative	1	1	
ArrowTip.java	getAngle	1	1	
ArrowTip.java	getInnerRadius	1	1	
ArrowTip.java	getOuterRadius	1	1	
ArrowTip.java	outline	1	1	
ArrowTip.java	outline	1	1	
ArrowTip.java	read	1	1	
ArrowTip.java	setAngle	1	1	
ArrowTip.java	setInnerRadius	1	1	
ArrowTip.java	setOuterRadius	1	1	
ArrowTip.java	write	1	1	
AttributeFigure.java	draw	3	3	
AttributeFigure.java	getAttribute	3	3	
AttributeFigure.java	writeObject	3	3	
AttributeFigure.java	getDefaultAttribute	2	2	
AttributeFigure.java	getDefaultAttribute	2	2	
AttributeFigure.java	initDefaultAttribute	2	2	
AttributeFigure.java	read	2	2	
AttributeFigure.java	setAttribute	2	2	
AttributeFigure.java	write	2	2	
AttributeFigure.java	AttributeFigure	1	1	
AttributeFigure.java	drawBackground	1	1	
AttributeFigure.java	drawFrame	1	1	
AttributeFigure.java	getAttribute	1	1	
AttributeFigure.java	getFillColor	1	1	
AttributeFigure.java	getFrameColor	1	1	
AttributeFigure.java	initializeAttributes	1	1	
AttributeFigure.java	setAttribute	1	1	
AttributeFigure.java	setDefaultAttribute	1	1	
AttributeFigureContentProducer.java	getContent	2	2	
AttributeFigureContentProducer.java	AttributeFigureContentProducer	1	1	
AttributeFigureContentProducer.java	read	1	1	
AttributeFigureContentProducer.java	write	1	1	
AutoscrollHelper.java	autoscroll	9	9	
AutoscrollHelper.java	AutoscrollHelper	1	1	
AutoscrollHelper.java	getAutoscrollInsets	1	1	
AutoscrollHelper.java	getAutoscrollMargin	1	1	
AutoscrollHelper.java	getSize	1	1	
AutoscrollHelper.java	getVisibleRect	1	1	
AutoscrollHelper.java	scrollRectToVisible	1	1	
AutoscrollHelper.java	setAutoscrollMargin	1	1	
AWTCursor.java	AWTCursor	1	1	
AWTCursor.java	AWTCursor	1	1	

Tab. C.10: M2DM to Eclipse Metrics comparison: *AlignCommand.java* to *AWTCursor.java*

Source	Method name	EM value	M2DM value	Recalculated value
BorderDecorator.java	getBorderOffset	2	2	
BorderDecorator.java	BorderDecorator	1	1	
BorderDecorator.java	BorderDecorator	1	1	
BorderDecorator.java	connectionInsets	1	1	
BorderDecorator.java	displayBox	1	1	
BorderDecorator.java	draw	1	1	
BorderDecorator.java	figureInvalidated	1	1	
BorderDecorator.java	initialize	1	1	
BorderDecorator.java	setBorderOffset	1	1	
BorderTool.java	mouseDown	4	4	
BorderTool.java	BorderTool	1	1	
BorderTool.java	action	1	1	
BorderTool.java	createUndoActivity	1	1	
BorderTool.java	reverseAction	1	1	
BouncingDrawing.java	add	3	3	
BouncingDrawing.java	animationStep	3	3	
BouncingDrawing.java	replace	3	3	
BouncingDrawing.java	remove	2	2	
Bounds.java	cropLine	38	38	
Bounds.java	intersectsLine	19	20	19
Bounds.java	equals	7	5	7
Bounds.java	intersectsBounds	4	8	4
Bounds.java	expandToRatio	3	3	
Bounds.java	hashCode	3	3	
Bounds.java	intersect	3	3	
Bounds.java	setCenter	2	2	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	Bounds	1	1	
Bounds.java	asRectangle2D	1	1	
Bounds.java	completelyContainsLine	1	2	1
Bounds.java	getCenter	1	1	
Bounds.java	getEast	1	1	
Bounds.java	getGreaterX	1	1	
Bounds.java	getGreaterY	1	1	
Bounds.java	getHeight	1	1	
Bounds.java	getLesserX	1	1	
Bounds.java	getLesserY	1	1	
Bounds.java	getNorth	1	1	
Bounds.java	getSouth	1	1	
Bounds.java	getWest	1	1	
Bounds.java	getWidth	1	1	
Bounds.java	includeBounds	1	1	
Bounds.java	includeLine	1	1	
Bounds.java	includeLine	1	1	
Bounds.java	includePoint	1	1	
Bounds.java	includePoint	1	1	
Bounds.java	includeRectangle2D	1	1	
Bounds.java	includeXCoordinate	1	1	
Bounds.java	includeYCoordinate	1	1	
Bounds.java	intersectsLine	1	1	
Bounds.java	intersectsPoint	1	2	1
Bounds.java	intersectsPoint	1	1	
Bounds.java	isCompletelyInside	1	2	1
Bounds.java	max	1	1	
Bounds.java	min	1	1	
Bounds.java	offset	1	1	
Bounds.java	shiftBy	1	1	
Bounds.java	toString	1	1	
Bounds.java	zoomBy	1	1	
BoxHandleKit.java	addCornerHandles	1	1	
BoxHandleKit.java	addHandles	1	1	
BoxHandleKit.java	east	1	1	
BoxHandleKit.java	northEast	1	1	
BoxHandleKit.java	northWest	1	1	
BoxHandleKit.java	north	1	1	
BoxHandleKit.java	southEast	1	1	
BoxHandleKit.java	southWest	1	1	
BoxHandleKit.java	south	1	1	
BoxHandleKit.java	west	1	1	
BringToFrontCommand.java	execute	2	2	
BringToFrontCommand.java	BringToFrontCommand	1	1	
BringToFrontCommand.java	createUndoActivity	1	1	
BringToFrontCommand.java	isExecutableWithView	1	1	
BufferedUpdateStrategy.java	draw	4	3	4

Tab. C.11: M2DM to Eclipse Metrics comparison: *BorderDecorator.java* to *BufferedUpdateStrategy.java*

Source	Method name	EM value	M2DM value	Recalculated value
ChangeAttributeCommand.java	execute	2	2	
ChangeAttributeCommand.java	ChangeAttributeCommand	1	1	
ChangeAttributeCommand.java	createUndoActivity	1	1	
ChangeAttributeCommand.java	isExecutableWithView	1	1	
ChangeConnectionEndHandle.java	ChangeConnectionEndHandle	1	1	
ChangeConnectionEndHandle.java	canConnectTo	1	1	
ChangeConnectionEndHandle.java	connect	1	1	
ChangeConnectionEndHandle.java	createUndoActivity	1	1	
ChangeConnectionEndHandle.java	disconnect	1	1	
ChangeConnectionEndHandle.java	locate	1	1	
ChangeConnectionEndHandle.java	setPoint	1	1	
ChangeConnectionEndHandle.java	target	1	1	
ChangeConnectionHandle.java	findConnectionTarget	7	5	6
ChangeConnectionHandle.java	invokeEnd	6	5	6
ChangeConnectionHandle.java	findConnectableFigure	5	5	
ChangeConnectionHandle.java	invokeStep	5	5	
ChangeConnectionHandle.java	source	2	2	
ChangeConnectionHandle.java	ChangeConnectionHandle	1	1	
ChangeConnectionHandle.java	draw	1	1	
ChangeConnectionHandle.java	findConnector	1	1	
ChangeConnectionHandle.java	getConnection	1	1	
ChangeConnectionHandle.java	getTargetFigure	1	1	
ChangeConnectionHandle.java	invokeStart	1	1	
ChangeConnectionHandle.java	setConnection	1	1	
ChangeConnectionHandle.java	setTargetFigure	1	1	
ChangeConnectionStartHandle.java	ChangeConnectionStartHandle	1	1	
ChangeConnectionStartHandle.java	canConnectTo	1	1	
ChangeConnectionStartHandle.java	connect	1	1	
ChangeConnectionStartHandle.java	createUndoActivity	1	1	
ChangeConnectionStartHandle.java	disconnect	1	1	
ChangeConnectionStartHandle.java	locate	1	1	
ChangeConnectionStartHandle.java	setPoint	1	1	
ChangeConnectionStartHandle.java	target	1	1	
ChopBoxConnector.java	findEnd	2	2	
ChopBoxConnector.java	findStart	2	2	
ChopBoxConnector.java	ChopBoxConnector	1	1	
ChopBoxConnector.java	ChopBoxConnector	1	1	
ChopBoxConnector.java	chop	1	1	
ChopDiamondConnector.java	chop	13	13	
ChopDiamondConnector.java	ChopDiamondConnector	1	1	
ChopDiamondConnector.java	ChopDiamondConnector	1	1	
ChopEllipseConnector.java	ChopEllipseConnector	1	1	
ChopEllipseConnector.java	ChopEllipseConnector	1	1	
ChopEllipseConnector.java	chop	1	1	
ChopPolygonConnector.java	ChopPolygonConnector	1	1	
ChopPolygonConnector.java	ChopPolygonConnector	1	1	
ChopPolygonConnector.java	chop	1	1	
Clipboard.java	Clipboard	1	1	
Clipboard.java	getClipboard	1	1	
Clipboard.java	getContents	1	1	
Clipboard.java	setContents	1	1	

Tab. C.12: M2DM to Eclipse Metrics comparison: *ChangeAttributeCommand.java* to *Clipboard.java*

Source	Method name	EM value	M2DM value	Recalculated value
ClippingUpdateStrategy.java	draw	6	6	
ClippingUpdateStrategy.java	ClippingUpdateStrategy	1	1	
CollectionsFactory.java	createCollectionsFactory	4	4	
CollectionsFactory.java	determineCollectionsFactory	2	2	
CollectionsFactory.java	isJDK12	2	2	
CollectionsFactory.java	current	1	1	
CollectionsFactoryJDK11.java	CollectionsFactoryJDK11	1	1	
CollectionsFactoryJDK11.java	createList	1	1	
CollectionsFactoryJDK11.java	createList	1	1	
CollectionsFactoryJDK11.java	createList	1	1	
CollectionsFactoryJDK11.java	createMap	1	1	
CollectionsFactoryJDK11.java	createMap	1	1	
CollectionsFactoryJDK11.java	createSet	1	1	
CollectionsFactoryJDK11.java	createSet	1	1	
CollectionsFactoryJDK12.java	CollectionsFactoryJDK12	1	1	
CollectionsFactoryJDK12.java	createList	1	1	
CollectionsFactoryJDK12.java	createList	1	1	
CollectionsFactoryJDK12.java	createList	1	1	
CollectionsFactoryJDK12.java	createMap	1	1	
CollectionsFactoryJDK12.java	createMap	1	1	
CollectionsFactoryJDK12.java	createSet	1	1	
CollectionsFactoryJDK12.java	createSet	1	1	
ColorContentProducer.java	getContent	2	1	2
ColorContentProducer.java	read	2	2	
ColorContentProducer.java	write	2	2	
ColorContentProducer.java	ColorContentProducer	1	1	
ColorContentProducer.java	ColorContentProducer	1	1	
ColorContentProducer.java	getColor	1	1	
ColorContentProducer.java	setColor	1	1	
ColorMap.java	colorIndex	3	3	
ColorMap.java	color	3	3	
ColorMap.java	color	3	3	
ColorMap.java	name	3	3	
ColorMap.java	ColorEntry	1	1	
ColorMap.java	isTransparent	1	1	
ColorMap.java	size	1	1	
CommandButton.java	actionPerformed	2	2	
CommandButton.java	CommandButton	1	1	
CommandCheckBoxMenuItem.java	CommandCheckBoxMenuItem	1	1	
CommandCheckBoxMenuItem.java	CommandCheckBoxMenuItem	1	1	
CommandCheckBoxMenuItem.java	CommandCheckBoxMenuItem	1	1	
CommandCheckBoxMenuItem.java	CommandCheckBoxMenuItem	1	1	
CommandCheckBoxMenuItem.java	getCommand	1	1	
CommandCheckBoxMenuItem.java	setCommand	1	1	
CommandChoice.java	itemStateChanged	4	4	
CommandChoice.java	CommandChoice	1	1	
CommandChoice.java	addItem	1	1	
CommandMenu.java	actionPerformed	4	4	
CommandMenu.java	checkEnabled	3	3	
CommandMenu.java	enable	3	3	
CommandMenu.java	CommandMenu	1	1	
CommandMenu.java	addCheckItem	1	1	
CommandMenu.java	addMenuItem	1	1	
CommandMenu.java	add	1	1	
CommandMenu.java	add	1	1	
CommandMenu.java	commandExecutable	1	1	
CommandMenu.java	commandExecuted	1	1	
CommandMenu.java	commandNotExecutable	1	1	
CommandMenu.java	remove	1	1	
CommandMenu.java	remove	1	1	
CommandMenuItem.java	CommandMenuItem	1	1	
CommandMenuItem.java	CommandMenuItem	1	1	
CommandMenuItem.java	CommandMenuItem	1	1	
CommandMenuItem.java	actionPerformed	1	1	
CommandMenuItem.java	getCommand	1	1	
CommandMenuItem.java	setCommand	1	1	

Tab. C.13: M2DM to Eclipse Metrics comparison: *ClippingUpdateStrategy.java* to *CommandMenuItem.java*

Source	Method name	EM value	M2DM value	Recalculated value
ComponentFigure.java	ComponentFigure	1	1	
ComponentFigure.java	ComponentFigure	1	1	
ComponentFigure.java	basicDisplayBox	1	1	
ComponentFigure.java	basicMoveBy	1	1	
ComponentFigure.java	displayBox	1	1	
ComponentFigure.java	draw	1	1	
ComponentFigure.java	getComponent	1	1	
ComponentFigure.java	handles	1	1	
ComponentFigure.java	setComponent	1	1	
CompositeFigure.java	findFigureInsideWithout	7	7	
CompositeFigure.java	findFigureWithout	5	5	
CompositeFigure.java	findFigure	5	5	
CompositeFigure.java	sendToLayer	5	5	
CompositeFigure.java	_addToQuadTree	4	4	
CompositeFigure.java	figures	4	4	
CompositeFigure.java	findFigureInside	4	4	
CompositeFigure.java	includes	4	4	
CompositeFigure.java	assignFiguresToPredecessorZValue	3	3	
CompositeFigure.java	assignFiguresToSuccessorZValue	3	3	
CompositeFigure.java	findFigure	3	3	
CompositeFigure.java	findFigure	3	3	
CompositeFigure.java	getFigureFromLayer	3	3	
CompositeFigure.java	_clearQuadTree	2	2	
CompositeFigure.java	_removeFromQuadTree	2	2	
CompositeFigure.java	addAll	2	2	
CompositeFigure.java	add	2	2	
CompositeFigure.java	basicMoveBy	2	2	
CompositeFigure.java	bringToFront	2	2	
CompositeFigure.java	draw	2	2	
CompositeFigure.java	figureInvalidated	2	2	
CompositeFigure.java	figureRemoved	2	2	
CompositeFigure.java	figureRequestRemove	2	2	
CompositeFigure.java	figureRequestUpdate	2	2	
CompositeFigure.java	getLayer	2	2	
CompositeFigure.java	init	2	2	
CompositeFigure.java	orphanAll	2	2	
CompositeFigure.java	readObject	2	2	
CompositeFigure.java	read	2	2	
CompositeFigure.java	release	2	2	
CompositeFigure.java	removeAll	2	2	
CompositeFigure.java	removeAll	2	2	
CompositeFigure.java	remove	2	2	
CompositeFigure.java	replace	2	2	
CompositeFigure.java	sendToBack	2	2	
CompositeFigure.java	write	2	2	
CompositeFigure.java	CompositeFigure	1	1	
CompositeFigure.java	addAll	1	1	
CompositeFigure.java	containsFigure	1	1	
CompositeFigure.java	draw	1	1	
CompositeFigure.java	figureAt	1	1	
CompositeFigure.java	figureChanged	1	1	
CompositeFigure.java	figureCount	1	1	
CompositeFigure.java	figures	1	1	
CompositeFigure.java	figuresReverse	1	1	
CompositeFigure.java	orphanAll	1	1	
CompositeFigure.java	orphan	1	1	
CompositeFigure.java	removeAll	1	1	

Tab. C.14: M2DM to Eclipse Metrics comparison: *ComponentFigure.java* to *CompositeFigure.java*

Source	Method name	EM value	M2DM value	Recalculated value
CompositeFigureCreationTool.java	mouseDown	3	3	
CompositeFigureCreationTool.java	mouseMove	3	3	
CompositeFigureCreationTool.java	CompositeFigureCreationTool	1	1	
CompositeFigureCreationTool.java	getContainerFigure	1	1	
CompositeFigureCreationTool.java	setContainerFigure	1	1	
CompositeFigureCreationTool.java	toolDone	1	1	
ConnectedTextTool.java	endEdit	5	5	
ConnectedTextTool.java	mouseDown	5	4	5
ConnectedTextTool.java	ConnectedTextTool	1	1	
ConnectedTextTool.java	activate	1	1	
ConnectedTextTool.java	createDeleteUndoActivity	1	1	
ConnectedTextTool.java	createUndoActivity	1	1	
ConnectedTextTool.java	getConnectedFigure	1	1	
ConnectedTextTool.java	setConnectedFigure	1	1	
ConnectionHandle.java	findConnectableFigure	5	4	5
ConnectionHandle.java	findConnectionTarget	5	3	5
ConnectionHandle.java	invokeStep	5	5	
ConnectionHandle.java	invokeEnd	3	3	
ConnectionHandle.java	ConnectionHandle	1	1	
ConnectionHandle.java	createConnection	1	1	
ConnectionHandle.java	createUndoActivity	1	1	
ConnectionHandle.java	draw	1	1	
ConnectionHandle.java	findConnector	1	1	
ConnectionHandle.java	getConnection	1	1	
ConnectionHandle.java	getCursor	1	1	
ConnectionHandle.java	getTargetFigure	1	1	
ConnectionHandle.java	invokeStart	1	1	
ConnectionHandle.java	setConnection	1	1	
ConnectionHandle.java	setTargetFigure	1	1	
ConnectionHandle.java	startConnector	1	1	

Tab. C.15: M2DM to Eclipse Metrics comparison: *CompositeFigureCreationTool.java* to *ConnectionHandle.java*

Source	Method name	EM value	M2DM value	Recalculated value
ConnectionTool.java	trackConnectors	7	7	
ConnectionTool.java	findTarget	6	6	
ConnectionTool.java	findConnectableFigure	5	4	5
ConnectionTool.java	mouseDown	5	5	
ConnectionTool.java	mouseUp	5	5	
ConnectionTool.java	findConnection	4	4	
ConnectionTool.java	mouseDrag	4	4	
ConnectionTool.java	findConnectionStart	3	3	
ConnectionTool.java	deactivate	2	2	
ConnectionTool.java	ConnectionTool	1	1	
ConnectionTool.java	createConnection	1	1	
ConnectionTool.java	createUndoActivity	1	1	
ConnectionTool.java	findConnector	1	1	
ConnectionTool.java	findSource	1	1	
ConnectionTool.java	getAddedFigure	1	1	
ConnectionTool.java	getConnection	1	1	
ConnectionTool.java	getEndConnector	1	1	
ConnectionTool.java	getStartConnector	1	1	
ConnectionTool.java	getTargetConnector	1	1	
ConnectionTool.java	getTargetFigure	1	1	
ConnectionTool.java	mouseMove	1	1	
ConnectionTool.java	setAddedFigure	1	1	
ConnectionTool.java	setConnection	1	1	
ConnectionTool.java	setEndConnector	1	1	
ConnectionTool.java	setStartConnector	1	1	
ConnectionTool.java	setTargetConnector	1	1	
ConnectionTool.java	setTargetFigure	1	1	
ContentProducerRegistry.java	getSuperClassContentProducer	7	7	
ContentProducerRegistry.java	getExactContentProducer	3	3	
ContentProducerRegistry.java	read	3	3	
ContentProducerRegistry.java	getContentProducer	2	2	
ContentProducerRegistry.java	unregisterContentProducer	2	2	
ContentProducerRegistry.java	write	2	2	
ContentProducerRegistry.java	ContentProducerRegistry	1	1	
ContentProducerRegistry.java	ContentProducerRegistry	1	1	
ContentProducerRegistry.java	getDefaultContentProducer	1	1	
ContentProducerRegistry.java	getExactDefaultContentProducer	1	1	
ContentProducerRegistry.java	getParent	1	1	
ContentProducerRegistry.java	isAutonomous	1	1	
ContentProducerRegistry.java	registerContentProducer	1	1	
ContentProducerRegistry.java	registerDefaultContentProducer	1	1	
ContentProducerRegistry.java	setAutonomous	1	1	
ContentProducerRegistry.java	setParent	1	1	
ContentProducerRegistry.java	unregisterDefaultContentProducer	1	1	

Tab. C.16: M2DM to Eclipse Metrics comparison: *ConnectionTool.java* to *ContentProducerRegistry.java*

Source	Method name	EM value	M2DM value	Recalculated value
CopyCommand.java	CopyCommand	1	1	
CopyCommand.java	execute	1	1	
CopyCommand.java	isExecutableWithView	1	1	
CreationTool.java	mouseUp	4	4	
CreationTool.java	activate	2	2	
CreationTool.java	createFigure	2	2	
CreationTool.java	mouseDrag	2	2	
CreationTool.java	CreationTool	1	1	
CreationTool.java	CreationTool	1	1	
CreationTool.java	createUndoActivity	1	1	
CreationTool.java	deactivate	1	1	
CreationTool.java	getAddedFigure	1	1	
CreationTool.java	getAddedFigures	1	1	
CreationTool.java	getCreatedFigure	1	1	
CreationTool.java	getPrototypeFigure	1	1	
CreationTool.java	mouseDown	1	1	
CreationTool.java	setAddedFigures	1	1	
CreationTool.java	setAddedFigure	1	1	
CreationTool.java	setCreatedFigure	1	1	
CreationTool.java	setPrototypeFigure	1	1	
CTXCommandMenu.java	checkEnabled	6	6	
CTXCommandMenu.java	actionPerformed	4	4	
CTXCommandMenu.java	enable	3	3	
CTXCommandMenu.java	CTXCommandMenu	1	1	
CTXCommandMenu.java	addCheckItem	1	1	
CTXCommandMenu.java	addMenuItem	1	1	
CTXCommandMenu.java	add	1	1	
CTXCommandMenu.java	add	1	1	
CTXCommandMenu.java	add	1	1	
CTXCommandMenu.java	add	1	1	
CTXCommandMenu.java	commandExecutable	1	1	
CTXCommandMenu.java	commandExecuted	1	1	
CTXCommandMenu.java	commandNotExecutable	1	1	
CTXCommandMenu.java	remove	1	1	
CTXCommandMenu.java	remove	1	1	
CTXWindowMenu.java	CTXWindowMenu	6	1	1
CTXWindowMenu.java	buildChildMenus	3	3	
CTXWindowMenu.java	removeWindowsList	2	2	
CustomSelectionTool.java	showPopupMenu	6	6	
CustomSelectionTool.java	handlePopupMenu	4	4	
CustomSelectionTool.java	mouseUp	3	3	
CustomSelectionTool.java	mouseDown	2	2	
CustomSelectionTool.java	mouseDrag	2	2	
CustomSelectionTool.java	CustomSelectionTool	1	1	
CustomSelectionTool.java	handleMouseClick	1	1	
CustomSelectionTool.java	handleMouseDoubleClick	1	1	
CustomSelectionTool.java	handleMouseDown	1	1	
CustomSelectionTool.java	handleMouseUp	1	1	
CustomToolBar.java	activateTools	3	3	
CustomToolBar.java	add	2	2	
CustomToolBar.java	switchToEditTools	2	2	
CustomToolBar.java	switchToStandardTools	2	2	
CustomToolBar.java	switchToolBar	2	2	
CustomToolBar.java	CustomToolBar	1	1	
CutCommand.java	execute	4	4	
CutCommand.java	CutCommand	1	1	
CutCommand.java	createUndoActivity	1	1	
CutCommand.java	isExecutableWithView	1	1	

Tab. C.17: M2DM to Eclipse Metrics comparison: *CopyCommand.java* to *CutCommand.java*

Source	Method name	EM value	M2DM value	Recalculated value
DecoratorFigure.java	findFigureInside	3	3	
DecoratorFigure.java	figureInvalidated	2	2	
DecoratorFigure.java	figureRequestRemove	2	2	
DecoratorFigure.java	figureRequestUpdate	2	2	
DecoratorFigure.java	DecoratorFigure	1	1	
DecoratorFigure.java	DecoratorFigure	1	1	
DecoratorFigure.java	addDependendFigure	1	1	
DecoratorFigure.java	basicDisplayBox	1	1	
DecoratorFigure.java	basicMoveBy	1	1	
DecoratorFigure.java	canConnect	1	1	
DecoratorFigure.java	connectedTextLocator	1	1	
DecoratorFigure.java	connectionInsets	1	1	
DecoratorFigure.java	connectorAt	1	1	
DecoratorFigure.java	connectorVisibility	1	1	
DecoratorFigure.java	containsPoint	1	1	
DecoratorFigure.java	decompose	1	1	
DecoratorFigure.java	decorate	1	1	
DecoratorFigure.java	displayBox	1	1	
DecoratorFigure.java	draw	1	1	
DecoratorFigure.java	figureChanged	1	1	
DecoratorFigure.java	figureRemoved	1	1	
DecoratorFigure.java	figures	1	1	
DecoratorFigure.java	getAttribute	1	1	
DecoratorFigure.java	getAttribute	1	1	
DecoratorFigure.java	getDecoratedFigure	1	1	
DecoratorFigure.java	getDependendFigures	1	1	
DecoratorFigure.java	getTextHolder	1	1	
DecoratorFigure.java	handles	1	1	
DecoratorFigure.java	includes	1	2	1
DecoratorFigure.java	initialize	1	1	
DecoratorFigure.java	moveBy	1	1	
DecoratorFigure.java	peelDecoration	1	1	
DecoratorFigure.java	readObject	1	1	
DecoratorFigure.java	read	1	1	
DecoratorFigure.java	release	1	1	
DecoratorFigure.java	removeDependendFigure	1	1	
DecoratorFigure.java	setAttribute	1	1	
DecoratorFigure.java	setAttribute	1	1	
DecoratorFigure.java	setDecoratedFigure	1	1	
DecoratorFigure.java	write	1	1	

Tab. C.18: M2DM to Eclipse Metrics comparison: *DecoratorFigure.java*

Source	Method name	EM value	M2DM value	Recalculated value
DeleteCommand.java	execute	4	4	
DeleteCommand.java	DeleteCommand	1	1	
DeleteCommand.java	createUndoActivity	1	1	
DeleteCommand.java	isExecutableWithView	1	1	
DeleteFromDrawingVisitor.java	visitFigure	3	3	
DeleteFromDrawingVisitor.java	DeleteFromDrawingVisitor	1	1	
DeleteFromDrawingVisitor.java	getDeletedFigures	1	1	
DeleteFromDrawingVisitor.java	getDrawing	1	1	
DeleteFromDrawingVisitor.java	setDrawing	1	1	
DeleteFromDrawingVisitor.java	visitFigureChangeListener	1	1	
DeleteFromDrawingVisitor.java	visitHandle	1	1	
DesktopEvent.java	DesktopEvent	1	1	
DesktopEvent.java	DesktopEvent	1	1	
DesktopEvent.java	getDrawingView	1	1	
DesktopEvent.java	getPreviousDrawingView	1	1	
DesktopEvent.java	setDrawingView	1	1	
DesktopEvent.java	setPreviousDrawingView	1	1	
DesktopEventService.java	createComponentListener	3	1	1
DesktopEventService.java	getDrawingViews	3	3	
DesktopEventService.java	removeComponent	3	3	
DesktopEventService.java	fireDrawingViewAddedEvent	2	2	
DesktopEventService.java	fireDrawingViewRemovedEvent	2	2	
DesktopEventService.java	fireDrawingViewSelectedEvent	2	2	
DesktopEventService.java	DesktopEventService	1	1	
DesktopEventService.java	addComponent	1	1	
DesktopEventService.java	addDesktopListener	1	1	
DesktopEventService.java	createDesktopEvent	1	1	
DesktopEventService.java	getActiveDrawingView	1	1	
DesktopEventService.java	getContainer	1	1	
DesktopEventService.java	getDesktop	1	1	
DesktopEventService.java	removeAllComponents	1	1	
DesktopEventService.java	removeDesktopListener	1	1	
DesktopEventService.java	setActiveDrawingView	1	1	
DesktopEventService.java	setContainer	1	1	
DesktopEventService.java	setDesktop	1	1	

Tab. C.19: M2DM to Eclipse Metrics comparison: *DeleteCommand.java* to *DesktopEventService.java*

Source	Method name	EM value	M2DM value	Recalculated value
DiamondFigure.java	DiamondFigure	1	1	
DiamondFigure.java	DiamondFigure	1	1	
DiamondFigure.java	chop	1	1	
DiamondFigure.java	connectionInsets	1	1	
DiamondFigure.java	connectorAt	1	1	
DiamondFigure.java	containsPoint	1	1	
DiamondFigure.java	draw	1	1	
DiamondFigure.java	getPolygon	1	1	
DiamondFigureGeometricAdapter.java	DiamondFigureGeometricAdapter	1	1	
DiamondFigureGeometricAdapter.java	DiamondFigureGeometricAdapter	1	1	
DiamondFigureGeometricAdapter.java	getShape	1	1	
DisposableResourceManagerFactory.java	initManager	6	6	
DisposableResourceManagerFactory.java	createStandardHolder	1	1	
DisposableResourceManagerFactory.java	getManager	1	1	
DisposableResourceManagerFactory.java	setStrategy	1	1	
DNDFigures.java	DNDFigures	2	2	
DNDFigures.java	getFigures	1	1	
DNDFigures.java	getOrigin	1	1	
DNDFiguresTransferable.java	getTransferData	2	2	
DNDFiguresTransferable.java	DNDFiguresTransferable	1	1	
DNDFiguresTransferable.java	getTransferDataFlavors	1	1	
DNDFiguresTransferable.java	isDataFlavorSupported	1	1	
DNDHelper.java	processReceivedData	10	10	
DNDHelper.java	createDropTarget	3	3	
DNDHelper.java	deinitialize	3	3	
DNDHelper.java	initialize	3	3	
DNDHelper.java	setDropTarget	3	3	
DNDHelper.java	createDragGestureRecognizer	2	2	
DNDHelper.java	destroyDragGestreRecognizer	2	2	
DNDHelper.java	DNDHelper	1	1	
DNDHelper.java	createDragSourceListener	1	1	
DNDHelper.java	createDropTargetListener	1	1	
DNDHelper.java	getDragGestureListener	1	1	
DNDHelper.java	getDragGestureRecognizer	1	1	
DNDHelper.java	getDragSourceActions	1	1	
DNDHelper.java	getDragSourceListener	1	1	
DNDHelper.java	getDropTargetActions	1	1	
DNDHelper.java	getDropTargetListener	1	1	
DNDHelper.java	setDragGestureListener	1	1	
DNDHelper.java	setDragGestureRecognizer	1	1	
DNDHelper.java	setDragSourceListener	1	1	
DNDHelper.java	setDropTargetListener	1	1	

Tab. C.20: M2DM to Eclipse Metrics comparison: *DiamondFigure.java* to *DNDHelper.java*

Source	Method name	EM value	M2DM value	Recalculated value
DoubleBufferImage.java	DoubleBufferImage	1	1	
DoubleBufferImage.java	flush	1	1	
DoubleBufferImage.java	getGraphics	1	1	
DoubleBufferImage.java	getHeight	1	1	
DoubleBufferImage.java	getProperty	1	1	
DoubleBufferImage.java	getRealImage	1	1	
DoubleBufferImage.java	getScaledInstance	1	1	
DoubleBufferImage.java	getSource	1	1	
DoubleBufferImage.java	getWidth	1	1	
DragNDropTool.java	createDragGestureListener	8	1	1
DragNDropTool.java	mouseDown	8	8	
DragNDropTool.java	setCursor	4	4	
DragNDropTool.java	mouseDrag	2	2	
DragNDropTool.java	mouseMove	2	2	
DragNDropTool.java	mouseUp	2	2	
DragNDropTool.java	viewCreated	2	2	
DragNDropTool.java	viewDestroying	2	2	
DragNDropTool.java	DragNDropTool	1	1	
DragNDropTool.java	activate	1	1	
DragNDropTool.java	createAreaTracker	1	1	
DragNDropTool.java	createDragTracker	1	1	
DragNDropTool.java	createHandleTracker	1	1	
DragNDropTool.java	deactivate	1	1	
DragNDropTool.java	getDragGestureListener	1	1	
DragNDropTool.java	isDragOn	1	1	
DragNDropTool.java	setDragGestureListener	1	1	
DragNDropTool.java	setDragOn	1	1	
DragTracker.java	mouseDrag	4	3	4
DragTracker.java	mouseDown	3	3	
DragTracker.java	deactivate	2	2	
DragTracker.java	DragTracker	1	1	
DragTracker.java	activate	1	1	
DragTracker.java	createUndoActivity	1	1	
DragTracker.java	getAnchorFigure	1	1	
DragTracker.java	getLastMouseX	1	1	
DragTracker.java	getLastMouseY	1	1	
DragTracker.java	hasMoved	1	1	
DragTracker.java	setAnchorFigure	1	1	
DragTracker.java	setHasMoved	1	1	
DragTracker.java	setLastMouseX	1	1	
DragTracker.java	setLastMouseY	1	1	

Tab. C.21: M2DM to Eclipse Metrics comparison: *DoubleBufferImage.java* to *DragTracker.java*

Source	Method name	EM value	M2DM value	Recalculated value
DrawApplet.java	createButtons	4	4	
DrawApplet.java	setupAttributes	4	4	
DrawApplet.java	guessType	3	3	
DrawApplet.java	loadDrawing	3	3	
DrawApplet.java	paletteUserOver	3	3	
DrawApplet.java	readDrawing	3	3	
DrawApplet.java	readFromObjectInput	3	3	
DrawApplet.java	setSelected	3	3	
DrawApplet.java	setTool	3	3	
DrawApplet.java	createColorChoice	2	2	
DrawApplet.java	createFontChoice	2	2	
DrawApplet.java	readFromStorableInput	2	2	
DrawApplet.java	showHelp	2	2	
DrawApplet.java	addViewChangeListener	1	1	
DrawApplet.java	createAttributeChoices	1	1	
DrawApplet.java	createAttributesPanel	1	1	
DrawApplet.java	createButtonPanel	1	1	
DrawApplet.java	createDrawing	1	1	
DrawApplet.java	createDrawingView	1	1	
DrawApplet.java	createIconkit	1	1	
DrawApplet.java	createSelectionTool	1	1	
DrawApplet.java	createToolButton	1	1	
DrawApplet.java	createToolPalette	1	1	
DrawApplet.java	createTools	1	1	
DrawApplet.java	drawing	1	1	
DrawApplet.java	figureSelectionChanged	1	1	
DrawApplet.java	getRequiredVersions	1	1	
DrawApplet.java	getUndoManager	1	1	
DrawApplet.java	getVersionControlStrategy	1	1	
DrawApplet.java	init	1	1	
DrawApplet.java	initDrawing	1	1	
DrawApplet.java	paletteUserSelected	1	1	
DrawApplet.java	removeViewChangeListener	1	1	
DrawApplet.java	setBufferedDisplayUpdate	1	1	
DrawApplet.java	setSimpleDisplayUpdate	1	1	
DrawApplet.java	setUndoManager	1	1	
DrawApplet.java	tool	1	1	
DrawApplet.java	toolDone	1	1	
DrawApplet.java	view	1	1	
DrawApplet.java	viewSelectionChanged	1	1	
DrawApplet.java	views	1	1	

Tab. C.22: M2DM to Eclipse Metrics comparison: *DrawApplet.java*

Source	Method name	EM value	M2DM value	Recalculated value
DrawApplication.java	open	6	5	5
DrawApplication.java	promptSaveAs	5	5	
DrawApplication.java	promptOpen	4	4	
DrawApplication.java	setTool	4	4	
DrawApplication.java	checkCommandMenus	3	3	
DrawApplication.java	checkCommandMenu	3	3	
DrawApplication.java	createDesktopListener	3	1	1
DrawApplication.java	loadDrawing	3	3	
DrawApplication.java	newView	3	3	
DrawApplication.java	paletteUserOver	3	3	
DrawApplication.java	print	3	3	
DrawApplication.java	saveDrawing	3	3	
DrawApplication.java	setSelected	3	3	
DrawApplication.java	addMenuIfPossible	2	2	
DrawApplication.java	createColorMenu	2	2	
DrawApplication.java	createFontMenu	2	2	
DrawApplication.java	createFontSizeMenu	2	2	
DrawApplication.java	createLookAndFeelMenu	2	2	
DrawApplication.java	endApp	2	2	
DrawApplication.java	fireViewCreatedEvent	2	2	
DrawApplication.java	fireViewDestroyingEvent	2	2	
DrawApplication.java	fireViewSelectionChangedEvent	2	2	
DrawApplication.java	getDefaultTool	2	2	
DrawApplication.java	newLookAndFeel	2	2	
DrawApplication.java	newWindow	2	2	
DrawApplication.java	setDefaultTool	2	2	
DrawApplication.java	setDrawingTitle	2	2	
DrawApplication.java	toolDone	2	2	
DrawApplication.java	DrawApplication	1	1	
DrawApplication.java	DrawApplication	1	1	
DrawApplication.java	addListeners	1	1	
DrawApplication.java	addViewChangeListener	1	1	

Tab. C.23: M2DM to Eclipse Metrics comparison: *DrawApplication.java* (1/2)

Source	Method name	EM value	M2DM value	Recalculated value
DrawApplication.java	closeQuery	1	1	
DrawApplication.java	createAlignmentMenu	1	1	
DrawApplication.java	createApplication	1	1	
DrawApplication.java	createArrowMenu	1	1	
DrawApplication.java	createAttributesMenu	1	1	
DrawApplication.java	createDebugMenu	1	1	
DrawApplication.java	createDefaultTool	1	1	
DrawApplication.java	createDesktop	1	1	
DrawApplication.java	createDrawing	1	1	
DrawApplication.java	createDrawingView	1	1	
DrawApplication.java	createDrawingView	1	1	
DrawApplication.java	createEditMenu	1	1	
DrawApplication.java	createFileMenu	1	1	
DrawApplication.java	createFontStyleMenu	1	1	
DrawApplication.java	createIconkit	1	1	
DrawApplication.java	createInitialDrawingView	1	1	
DrawApplication.java	createMenus	1	1	
DrawApplication.java	createOpenFileChooser	1	1	
DrawApplication.java	createSaveFileChooser	1	1	
DrawApplication.java	createSelectionTool	1	1	
DrawApplication.java	createStatusLine	1	1	
DrawApplication.java	createStorageFormatManager	1	1	
DrawApplication.java	createToolButton	1	1	
DrawApplication.java	createToolPalette	1	1	
DrawApplication.java	createTools	1	1	
DrawApplication.java	defaultSize	1	1	
DrawApplication.java	destroy	1	1	
DrawApplication.java	exit	1	1	
DrawApplication.java	figureSelectionChanged	1	1	
DrawApplication.java	getApplicationName	1	1	
DrawApplication.java	getDefaultDrawingTitle	1	1	
DrawApplication.java	getDesktop	1	1	
DrawApplication.java	getDesktopListener	1	1	
DrawApplication.java	getDrawingTitle	1	1	
DrawApplication.java	getDrawingViewSize	1	1	
DrawApplication.java	getIconkit	1	1	
DrawApplication.java	getRequiredVersions	1	1	
DrawApplication.java	getStatusLine	1	1	
DrawApplication.java	getStorageFormatManager	1	1	
DrawApplication.java	getUndoManager	1	1	
DrawApplication.java	getVersionControlStrategy	1	1	
DrawApplication.java	newWindow	1	1	
DrawApplication.java	open	1	1	
DrawApplication.java	paletteUserSelected	1	1	
DrawApplication.java	promptNew	1	1	
DrawApplication.java	removeViewChangeListener	1	1	
DrawApplication.java	setApplicationName	1	1	
DrawApplication.java	setDesktopListener	1	1	
DrawApplication.java	setDesktop	1	1	
DrawApplication.java	setIconkit	1	1	
DrawApplication.java	setStatusLine	1	1	
DrawApplication.java	setStorageFormatManager	1	1	
DrawApplication.java	setUndoManager	1	1	
DrawApplication.java	setView	1	1	
DrawApplication.java	showStatus	1	1	
DrawApplication.java	tool	1	1	
DrawApplication.java	view	1	1	
DrawApplication.java	views	1	1	

Tab. C.24: M2DM to Eclipse Metrics comparison: *DrawApplication.java* (2/2)

Source	Method name	EM value	M2DM value	Recalculated value
DrawingChangeEvent.java	DrawingChangeEvent	1	1	
DrawingChangeEvent.java	getDrawing	1	1	
DrawingChangeEvent.java	getInvalidatedRectangle	1	1	
DuplicateCommand.java	DuplicateCommand	1	1	
DuplicateCommand.java	createUndoActivity	1	1	
DuplicateCommand.java	execute	1	1	
DuplicateCommand.java	isExecutableWithView	1	1	
ElbowConnection.java	updatePoints	5	5	
ElbowConnection.java	handles	3	3	
ElbowConnection.java	ElbowConnection	1	1	
ElbowConnection.java	connectedTextLocator	1	1	
ElbowConnection.java	layoutConnection	1	1	
ElbowConnection.java	updateConnection	1	1	
ElbowConnection.java	locate	1	1	
ElbowHandle.java	constrainX	3	3	
ElbowHandle.java	constrainY	3	3	
ElbowHandle.java	invokeStep	2	2	
ElbowHandle.java	ElbowHandle	1	1	
ElbowHandle.java	draw	1	1	
ElbowHandle.java	invokeStart	1	1	
ElbowHandle.java	isVertical	1	1	
ElbowHandle.java	locate	1	1	
ElbowHandle.java	ownerConnection	1	1	
EllipseFigure.java	EllipseFigure	1	1	
EllipseFigure.java	EllipseFigure	1	1	
EllipseFigure.java	basicDisplayBox	1	1	
EllipseFigure.java	basicMoveBy	1	1	
EllipseFigure.java	connectionInsets	1	1	
EllipseFigure.java	connectorAt	1	1	
EllipseFigure.java	displayBox	1	1	
EllipseFigure.java	drawBackground	1	1	
EllipseFigure.java	drawFrame	1	1	
EllipseFigure.java	handles	1	1	
EllipseFigure.java	read	1	1	
EllipseFigure.java	write	1	1	
EllipseFigureGeometricAdapter.java	EllipseFigureGeometricAdapter	1	1	
EllipseFigureGeometricAdapter.java	EllipseFigureGeometricAdapter	1	1	
EllipseFigureGeometricAdapter.java	getShape	1	1	
ETSLADisposalStrategy.java	dispose	4	4	
ETSLADisposalStrategy.java	startDisposing	3	3	
ETSLADisposalStrategy.java	stopDisposing	3	3	
ETSLADisposalStrategy.java	initDisposalThread	2	2	
ETSLADisposalStrategy.java	setManager	2	2	
ETSLADisposalStrategy.java	setPeriodicity	2	2	
ETSLADisposalStrategy.java	ETSLADisposalStrategy	1	1	
ETSLADisposalStrategy.java	ETSLADisposalStrategy	1	1	
ETSLADisposalStrategy.java	ETSLADisposalStrategy	1	1	
ETSLADisposalStrategy.java	getManager	1	1	
ETSLADisposalStrategy.java	getPeriodicity	1	1	

Tab. C.25: M2DM to Eclipse Metrics comparison: *DrawingChangeEvent.java* to *ETSLADisposal-Strategy*

Source	Method name	EM value	M2DM value	Recalculated value
FastBufferedUpdateStrategy.java	_checkCaches	7	5	7
FastBufferedUpdateStrategy.java	draw	6	6	
FastBufferedUpdateStrategy.java	FastBufferedUpdateStrategy	1	1	
FigureAndEnumerator.java	nextFigure	3	3	
FigureAndEnumerator.java	FigureAndEnumerator	1	1	
FigureAndEnumerator.java	hasNextFigure	1	2	1
FigureAndEnumerator.java	reset	1	1	
FigureAttributeConstant.java	equals	8	8	
FigureAttributeConstant.java	getConstant	5	4	5
FigureAttributeConstant.java	addConstant	4	4	
FigureAttributeConstant.java	FigureAttributeConstant	1	1	
FigureAttributeConstant.java	FigureAttributeConstant	1	1	
FigureAttributeConstant.java	getConstant	1	1	
FigureAttributeConstant.java	getID	1	1	
FigureAttributeConstant.java	getName	1	1	
FigureAttributeConstant.java	hashCode	1	1	
FigureAttributeConstant.java	setID	1	1	
FigureAttributeConstant.java	setName	1	1	
FigureAttributes.java	read	10	10	
FigureAttributes.java	write	9	9	
FigureAttributes.java	clone	2	2	
FigureAttributes.java	set	2	2	
FigureAttributes.java	writeColor	2	2	
FigureAttributes.java	FigureAttributes	1	1	
FigureAttributes.java	get	1	1	
FigureAttributes.java	hasDefined	1	1	
FigureAttributes.java	readColor	1	1	
FigureChangeAdapter.java	figureChanged	1	1	
FigureChangeAdapter.java	figureInvalidated	1	1	
FigureChangeAdapter.java	figureRemoved	1	1	
FigureChangeAdapter.java	figureRequestRemove	1	1	
FigureChangeAdapter.java	figureRequestUpdate	1	1	
FigureChangeEvent.java	FigureChangeEvent	1	1	
FigureChangeEvent.java	FigureChangeEvent	1	1	
FigureChangeEvent.java	FigureChangeEvent	1	1	
FigureChangeEvent.java	getFigure	1	1	
FigureChangeEvent.java	getInvalidatedRectangle	1	1	
FigureChangeEvent.java	getNestedEvent	1	1	
FigureChangeEventMulticaster.java	remove	5	5	
FigureChangeEventMulticaster.java	removeInternal	4	4	
FigureChangeEventMulticaster.java	addInternal	3	3	
FigureChangeEventMulticaster.java	FigureChangeEventMulticaster	1	1	
FigureChangeEventMulticaster.java	add	1	1	
FigureChangeEventMulticaster.java	figureChanged	1	1	
FigureChangeEventMulticaster.java	figureInvalidated	1	1	
FigureChangeEventMulticaster.java	figureRemoved	1	1	
FigureChangeEventMulticaster.java	figureRequestRemove	1	1	
FigureChangeEventMulticaster.java	figureRequestUpdate	1	1	
FigureChangeEventMulticaster.java	remove	1	1	
FigureDataContentProducer.java	getContent	5	5	
FigureDataContentProducer.java	FigureDataContentProducer	1	1	
FigureDataContentProducer.java	read	1	1	
FigureDataContentProducer.java	write	1	1	

Tab. C.26: M2DM to Eclipse Metrics comparison: *FastBufferedUpdateStrategy.java* to *FigureDataContentProducer*

Source	Method name	EM value	Recalculated value
CompositeFigure.java	findFigureInsideWithout	7	7
CompositeFigure.java	findFigureWithout	5	5
CompositeFigure.java	findFigure	5	5
CompositeFigure.java	sendToLayer	5	5
CompositeFigure.java	_addToQuadTree	4	4
CompositeFigure.java	figures	4	4
CompositeFigure.java	findFigureInside	4	4
CompositeFigure.java	includes	4	4
CompositeFigure.java	assignFiguresToPredecessorZValue	3	3
CompositeFigure.java	assignFiguresToSuccessorZValue	3	3
CompositeFigure.java	findFigure	3	3
CompositeFigure.java	findFigure	3	3
CompositeFigure.java	getFigureFromLayer	3	3
CompositeFigure.java	_clearQuadTree	2	2
CompositeFigure.java	_removeFromQuadTree	2	2
CompositeFigure.java	addAll	2	2
CompositeFigure.java	add	2	2
CompositeFigure.java	basicMoveBy	2	2
CompositeFigure.java	bringToFront	2	2
CompositeFigure.java	draw	2	2
CompositeFigure.java	figureInvalidated	2	2
CompositeFigure.java	figureRemoved	2	2
CompositeFigure.java	figureRequestRemove	2	2
CompositeFigure.java	figureRequestUpdate	2	2
CompositeFigure.java	getLayer	2	2
CompositeFigure.java	init	2	2
CompositeFigure.java	orphanAll	2	2
CompositeFigure.java	readObject	2	2
CompositeFigure.java	read	2	2
CompositeFigure.java	release	2	2
CompositeFigure.java	removeAll	2	2
CompositeFigure.java	removeAll	2	2
CompositeFigure.java	remove	2	2
CompositeFigure.java	replace	2	2
CompositeFigure.java	sendToBack	2	2
CompositeFigure.java	write	2	2
CompositeFigure.java	CompositeFigure	1	1
CompositeFigure.java	addAll	1	1
CompositeFigure.java	containsFigure	1	1
CompositeFigure.java	draw	1	1
CompositeFigure.java	figureAt	1	1
CompositeFigure.java	figureChanged	1	1
CompositeFigure.java	figureCount	1	1
CompositeFigure.java	figures	1	1
CompositeFigure.java	figuresReverse	1	1
CompositeFigure.java	orphanAll	1	1
CompositeFigure.java	orphan	1	1
CompositeFigure.java	removeAll	1	1

Tab. C.27: M2DM to Eclipse Metrics comparison: *ComponentFigure.java* recalculated values