The Eclipse Java Metamodel
Scaffolding Software Engineering Research on Java Projects with MDE Techniques

Pedro Janeiro Coimbra[1] and Fernando Brito e Abreu[2,3]
[1]*ADETTI-IUL, Av.ª das Forças Armadas, 1649-026 Lisboa, Portugal*
[2]*DCTI, ISCTE-IUL, Av.ª das Forças Armadas, 1649-026 Lisboa, Portugal*
[3]*CITI, FCT/UNL, Quinta da Torre, 2829-516 Caparica, Portugal*
pedrojcoimbra@gmail.com, fba@{iscte-iul.pt, fct.unl.pt}

Keywords: Model-Driven Engineering, Metamodeling Techniques, Eclipse IDE, Java Projects, Software Metrics.

Abstract: Java on the Eclipse IDE is a frequent choice for software development nowadays. Software Engineering researchers have built program analysis tools in that environment for several purposes. However, that requires a deep understanding of Eclipse internals, such as the Java AST.
This paper discusses the feasibility of a metamodel-driven approach to scaffold the construction of such tools. Its core is the Eclipse Java Metamodel (EJMM), obtained through reverse engineering. The latter is instantiated with meta-objects representing the constructs of a given Java program. We then use OCL to traverse programs very easily. To validate the feasibility of our metamodel-driven approach to program analysis, we developed an Eclipse plug-in based on it, to support the metamodel-driven measurement (M2DM) approach.

1 INTRODUCTION

Over the past decade several Software Engineering researchers have proposed Java code analysis tools, often with a focus on software measurement (Wilkie and Harmer, 2002; Cahill et al., 2002; Antoniol et al., 2003; McQuillan, 2011). As a follow-up, we searched for opportunities within the Eclipse IDE to further aid research over Java software systems using MDE techniques. Our attention turned to the set of Eclipse plug-ins called Eclipse Java Development Tools (JDT) (The Eclipse Foundation, 2013), which offer control and access to typical compiler and IDE functions for code interpretation and project analysis. Several such tools have been developed as Eclipse plug-ins, but producing them requires a thorough knowledge of the Eclipse core internals, such as the Java AST. As distancing software analysis from the complex parsing logic to a more commonly understood OO perspective became a subject of study (Antoniol et al., 2003), we tasked ourselves to create the Eclipse Java Metamodel, hereinafter referred as EJMM, by reverse engineering the relevant parts of the Eclipse JDT.

Our main objective here is to show the feasibility of a higher level of abstraction upon which researchers can perform applied research using Java programs as the object of study. Examples may include program comprehension, metrics collection, code smells detection, program transformations (e.g., refactoring actions) or program evolution. In other words, we seek a metamodel-driven approach to scaffold the research upon Java code repositories.

To reify this proposal, we developed an Eclipse plug-in supporting a model-driven representation of Java programs based on EJMM instantiation. Using this higher-level abstraction we can traverse programs very easily, using model-driven techniques such as executing OCL expressions upon the EJMM. To assess the feasibility of our approach, we built a plug-in for metrics collection on top of it, thus obtaining a MetaModel-Driven Measurement (M2DM) tool for Java, within the Eclipse environment. M2DM was initially proposed in (Brito e Abreu, 2001).

This paper is organized as follows: section 2 describes the EJMM; section 3 describes the generic architecture of our M2DM Eclipse plug-in, based on EJMM, as well as choices regarding its instantiation process and its expected validation; section 4 presents a few examples of the use of OCL to traverse the EJMM; section 5 presents a preliminary validation; section 6 describes the related work; finally, section 7 draws some conclusions and outlines future work.

2 ECLIPSE JAVA METAMODEL

The EJMM was obtained by reverse engineering and composing two Eclipse JDT components: the Eclipse Java Model (hereinafter referred 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 (The Eclipse Foundation, 2013; Kuhn and Thomann, 2006). 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 adopted a simpler and more direct representation of a Java project, with meta-associations connecting specific metaclasses. The node metaclasses remain, but purely as a means to generalize different metaclasses and to disclose the information they contain.

Figure 1 represents the basic structure of a Java project and its hierarchical structure, including type inheritance and interface implementations. All metaclasses in Fig. 1 inherit, directly or indirectly, from the base *JavaElement* metaclass. The latter represents any node of a project tree, confers it a name (the one declared in the source code) and an unique identifier. Also included in this diagram are three enumerations: visibility types, Java types and package fragment root types. The first determines the visibility of an element (public, private, protected or default). The Java types enumeration characterizes the *Type* metaobject (Java class, interface, annotation type or enumeration). Package fragment roots can be folders or archives and the latter can be jar or zip files.

Figure 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. This can be checked by the following OCL invariant defined in the EJMM:

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

The fields *arrayDimensions* and *returnTypeArrayDimensions* determine the number of array dimensions a *LocalVariable*, *Field* or a *Method*'s *returnType* has.

In the final diagram (Fig. 3), 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 shapes, but only *Block* statements are meta-associated to initializers or methods. Much like local variables, the *Block*'s meta-associations are mutually exclusive - a block can only be associated with either an *Initializer* or a *Method*, or none at all. This can be upheld with the following OCL constraint:

```
context Block

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

Note that a *Block* may not be necessarily linked directly to a *Method* or *Initializer*. Instead, it may 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.

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. These dependency meta-associations are based on the AST bindings found in expressions contained inside statements, that hold several information about their contents. 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* field indicates the size of the piece of code in bytes. The *Statement* metaclass contains a *conditionalOperatorCount* field that 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). This field can be used for defining metrics such as the cyclomatic complexity (McCabe, 1976).

The previous metaclasses are based on the homonymous EJM interfaces (within `org.eclipse.jdt.core` package) or AST classes (within `org.eclipse.jdt.core.dom` package). A full list of metaclass origins can be found in (Coimbra, 2013).

While most metaclass fields derived from the original fields or getters, some of *Type*, *Method* and *Field* derived from an integer flag of the corresponding interfaces. The flags, defined in *org.eclipse.jdt.core.Flags*, differentiate members visibility (reflected on the enumeration *VisibilityType*) or Java types (enumeration *JavaType*). Flags were also used to create the *isFinal*, *isSynthetic*, *isDeprecated*, *isStatic*, *isSynchronized*, *isNative*, *isBridge*, *hasVarargs*, *isVolatile*, *isTransient* and *isStrictfp* fields.

Several auxiliary methods are declared in the metaclasses to support the definition of OCL statements. Examples include recursive methods that return the hierarchical children or parents of a given meta-object (e.g., all statements contained in a block, or all components of a type's superclass hierarchy).

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 been translated into fields in their corresponding metaclass. The most notable exception is the *Block* metaclass, as its contents is translated to a single aggregation of *Statement*.

Due to space constraints the full EJMM cannot be described herein. Please refer to (Coimbra, 2013).

3 M2DM TOOL BASED ON EJMM

As a first application of the EJMM, we created an open-source M2DM tool for Java, as an Eclipse plug-in (QUASAR, 2013b). The components of this tool are: i) a widely used open-source metrics plug-in for Eclipse produced by Frank Sauer (Sauer, 2013), ii) an OCL compiler embedded in the UML-based Specification Environment (USE) (Gogolla et al., 2007; Database Systems Group, University of Bremen, 2013), iii) a facade component interface named J-USE, produced within the QUASAR group (QUASAR, 2013a), that provides a Java API for USE services and, finally, iv) a transformation component that goes through 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, evaluate defined OCL constraints (invariants, pre and post-conditions) and execute other OCL expressions as queries over the model or metamodel instances. Those queries are indeed traversal operations upon the EJMM that allow evaluating expressions on concrete instantiations, that is, on actual Java programs. Frank Sauer's plug-in will only provide the functionality related to displaying and exporting metrics.

The EJMM is defined as a USE specification file (UML class diagram in textual format) distributed with the plug-in. A metrics set definition is expressed using OCL upon the EJMM on a separate USE specification file. The user will then be able to define and load new metric sets. In most tools (like the one from Frank Sauer) the metrics calculation algorithms are embedded somewhere in the tool's source code, thus hampering the addition of new metrics, due to the understandability effort required for creating the extension and producing a new executable.

4 USING OCL UPON THE EJMM

We will now present a few examples of using OCL to traverse the EJMM and retrieve specific data or meta-objects. The first is the well-known method's cyclomatic complexity metric (McCabe, 1976), using Frank Sauer's plug-in algorithm:

```
context Method

cyclomaticComplexity(): Integer =
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
```

The next examples are taken from of our implementation of the FLAME (Formal Library for Aiding Metrics Extraction) library (Baroni, 2002). The full implementation can be found in (QUASAR, 2013b).

```
context Type

children(): Set(Type) =
self.extendedBy->union(self.implementedBy)

CHIN(): Integer =
```

```
children()->size

definedFeatures() : Set(Member) =
self.fields->union(self.methods)->asSet

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

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

5 VALIDATION

Validating a metrics collection tool requires the identification of its quality model. Therefore, we start by enumerating its quality characteristics:

- Transparency - the calculation algorithms used in metrics collection should be clearly identified;
- Extensibility - new metrics should be easy to add;
- Scalability - the metrics collection process should not degrade drastically with reasonably large systems, given regular computing resources;
- Accuracy - the collected values for the metrics should be accurate.

Regarding transparency, recall that metrics definitions are contained in separate files. Since OCL has good expressiveness, anyone familiar with the semantics of UML class diagrams will understand the metrics collection algorithms, as the one provided earlier.

Tool extensibility (i.e. adding new metrics sets) is straightforward because the concepts used in metrics formalization are those defined in the EJMM. We just need to create a new file with the OCL definitions of the corresponding metrics. This capability is evidenced by the availability of the MOOD (Brito e Abreu and Carapuca, 1994) and the MOOSE (Chidamber and Kemerer, 1994) metrics sets built on top of FLAME, made available at (QUASAR, 2013b).

To test the scalability, we ran the M2DM tool upon Java projects of increasing size, comparing the duration of the instantiation process. We chose five different projects as test cases, including three key components of the M2DM plug-in itself: J-USE (QUASAR, 2013a), Frank Sauer's metrics plug-in (Sauer, 2013) and the USE tool (Database Systems Group, University of Bremen, 2013). The remaining two are a widely-used case-study for research purposes, JHotDraw (Gamma and Eggenschwiler, 2013), and a popular open-source application, SweetHome3D (Puybaret, 2013), which enjoys over one hundred thousand weekly downloads at the time of writing. We recorded the amount of EJMM objects and links created and the duration of the full instantiation process. Tests were made on a machine with a dual-core processor running at 3GHz each and 4GB of physical memory, and 1024 maximum JVM heap size. Tests consisted of running the instantiation process fifty times upon each project, from largest to smallest, under the same Eclipse session, and registering the total duration in seconds for the whole instantiation process. Its average value, represented in Table 1, allows claiming that the M2DM plug-in scales up nicely.

Table 1: Instantiation process test results.

Project	#Objects	#Links	Time(s)
J-USE (v.1.0)	3491	10356	0.7
Metrics (v.1.3.6)	17240	43303	4.3
JHotDraw (v.6.0)	64147	112806	13.8
SweetHome3D (v.4.0)	136716	311791	44.3
USE (v.3.0.6)	211452	395460	93.5

Regarding accuracy, our validation approach is based on a comparison between the values of the same metrics collected with the M2DM tool, with those obtained with the Frank Sauer plug-in. The latter is in use for several years and has a large number of downloads and several updates, so we had a good confidence that its calculated values could be used as a benchmark. The first metric tested was the method's cyclomatic complexity (McCabe, 1976), taking a random sample of 1005 non-abstract methods from non-interface and non-nested types from JHotDraw's source code. Out of 1005 cases, 6 presented different values between the prototype and Frank Sauer's plug-in, 5 of which were due to the M2DM approach not factoring the complexity of methods of anonymous classes inside an analyzed method and 1 differing due to Frank Sauer's plug-in counting occurrences of "&&" and "||" in commented code. Further accuracy tests are still required, but it is worth mentioning that this preliminary test allowed us to unveil a bug in Frank Sauer's plug-in.

6 RELATED WORK

Development of model-driven metrics tools has long been a research subject. In this section we will briefly survey, chronologically, related proposals.

The Java Metrics Reporter (JMR) (Cahill et al., 2002) pioneered in using a Java Model to calculate metrics. The latter is much akin to Eclipse JDT's EJM, sharing a similar hierarchical structure and a single node class from which all Java elements inherit

(named *JElement*). Extensions to the Java Model and implementation of metrics calculation is done by subclassing JMR's Java Model. Though this approach is stated to be much simpler than straight parser logic, it would still entail some programming effort. This technique is similar to what is commonly found in existing Eclipse metrics plug-ins, such as Frank Sauer's metrics tool (Sauer, 2013), using EJM-supplied data. Unfortunately, at the time of writing, we have not been able to find it available online.

The Extensible Metrics toolBench for Empirical Research (EMBER) uses a database schema to represent a metamodel aimed at OO languages such as C++ and Java (Wilkie and Harmer, 2002). It parsed the target software project to load the database and SQL queries were used to calculate metrics. Since the database schema aims to fit several OO languages, it is more generic than the EJMM proposed herein. This lack of detail shows up in the absence of type parameters, method statements or annotations, and prevents the calculation of well-known metrics such as the weighted methods per class (Chidamber and Kemerer, 1994) using McCabe's cyclomatic complexity metric (McCabe, 1976).

Antoniol et al. proposed a tool that navigates Java AST objects using OCL expressions (Antoniol et al., 2003). The authors use a metamodel based on the JavaCC compiling rules and exemplify its functionalities by formalizing a small set of software metrics. Their approach is similar to ours, though using a different metamodel. Their AST metamodel retains a node tree structure, whereas our EJMM aims to capture more directly a Java project's structure (despite that both the EJM and Eclipse AST have similar node tree structures). Furthermore, since their metamodel comprises only the AST, it excludes the project's overall structure (e.g., folders and packages). This drawback prevents the calculation of coarse-grained metrics, such as those defined at module / subsystem or system levels, like the MOOD metrics set (Brito e Abreu and Carapuca, 1994).

The Design-Metrics (DM) Crawler allows to collect, store, display and analyze object-oriented design metrics from XMI files using XQuery expressions (El Wakil et al., 2005). XQuery queries are used to formalize and calculate metrics. Their authors claimed this language allows the definition of complex metrics more easily than OCL, 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 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 (Engelhardt et al., 2009). OCL is also used for the definition and calculation of metrics, under OMGs Software Metrics Metamodel. Currently, it is available online and affiliated with the ModelBus tool (Hein et al., 2013). 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.

MoDisco is an Eclipse open source framework dedicated to Model Driven Reverse Engineering (MDRE) by supporting the development of tools for legacy migration or modernization, quality assurance or retro-documentation (Bruneliere et al., 2010). It allows Java reverse engineering, including a Java metamodel, corresponding discoverer and transformation to OMG's Knowledge Discovery Metamodel (KDM). MoDisco Java metamodel is also based on the Eclipse AST, extended with metaclasses to represent Java project components such as packages and compilation units. As for metrics definition, they can be expressed with Java queries on a separate project that uses the MoDisco API. This allows a separation of concerns, by detaching metrics definition from its collection internals. Nevertheless, our solution provides a higher abstraction level, granted by the use of the FLAME OCL API upon the EJMM.

McQuillan proposed a MOF-compliant metamodel for metrics extraction for Java and UML models (McQuillan, 2011). Metrics formalization was also in OCL over that 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.

Other non model-driven approaches, like metaprogramming, exist for metrics definition and calculation. One such example is Rascal, a Domain Specific Language (DSL) that has been used mainly for source code analysis and transformation. Its specific constructs include the visit statement (for structure-shy traversal and transformation of arbitrary data such as parse trees), comprehensions (for querying and creating sets, relations, maps or lists) and regular expressions (for string matching), that can be used to define source code metrics. In (Klint et al., 2010), for instance, the authors used Rascal to construct generic (parse-tree-based) metric calculators for comparing six implementations of the same DSL using different languages (Java, JavaScript, C#) and DSL tools

(ANTLR, OMeta, Microsoft M).

7 CONCLUSIONS/FUTURE WORK

The proposed EJMM metamodel was instantiated with data provided by the Java Model and the Eclipse JDT's AST. It allowed the formalization in OCL (and collection) of several metrics sets on top of it easily. It matches very closely the concepts of the Java language and, in our view, is a better alternative when compared with related works that sacrifice functionality for the sake of portability.

Although we have mainly used our MDE technique for metrics collection, we realized that the current plug-in architecture (except for the Frank Sauer's metrics visualization component) is generic and it may be applied to develop other tools that require Java source code analysis, in areas such as program comprehension, code smells detection, refactoring or software evolution, to name a few. Demonstrating the feasibility of our approach in those research areas will be a subject of our future work.

REFERENCES

Antoniol, G., Di Penta, M., and Merlo, E. (2003). Yaab (yet another ast browser): using ocl to navigate asts. In *Intern. Workshop on Program Comprehension*, pages 13–22. IEEE.

Baroni, A. L. (2002). *Formal Definition of Object-Oriented Design Metrics*. Msc thesis, Vrije Universiteit Brussel (VUB), Brussels, Belgium.

Brito e Abreu, F. (2001). Using ocl to formalize object oriented metrics definitions. Technical Report ES007/2001, INESC.

Brito e Abreu, F. and Carapuca, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *4th International Conference on Software Quality*, McLean, Virginia, USA. American Society for Quality.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). Modisco: a generic and extensible framework for model driven reverse engineering. In *ASE'10*, ASE'10, pages 173–174, New York, NY, USA. ACM.

Cahill, J., Hogan, J. M., and Thomas, R. (2002). The java metrics reporter - an extensible tool for oo software analysis. In *9th Asia-Pacific Software Engineering Conference*, pages 507–516.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Coimbra, P. J. (2013). *An Eclipse Plug-in for Metamodel Driven Measurement*. Msc thesis, ISCTE-IUL, Lisbon, Portugal.

Database Systems Group, University of Bremen (2013). Sourceforge.net: The uml-based specification environment. http:// sourceforge.net/apps/mediawiki/useocl/. Accessed: 2013-05-30.

El Wakil, M., El Bastawissi, A., Boshra, M., and Fahmy, A. (2005). A novel approach to formalize and collect object-oriented design-metrics. In *9th Intern. Conf. on Empirical Assessment in Software Engineering*.

Engelhardt, M., Hein, C., Ritter, T., and Wagner, M. (2009). Generation of formal model metrics for mof based domain specific languages. *Electronic Communications of the EASST*, 24.

Gamma, E. and Eggenschwiler, T. (2013). Jhotdraw start page. Accessed: 13-08-2013.

Gogolla, M., Bttner, F., and Richters, M. (2007). Use: A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(13):27–34.

Hein, C., Engelhardt, M., Ritter, T., and Wagner, M. (2013). Metrino. http://www. modelbus.org/modelbus/index.php/metrino. Accessed: 2013-05-30.

Klint, P., van der Storm, T., and Vinju, J. (2010). On the impact of dsl tools on the maintainability of language implementations. In *10th Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 10:1–10:9, New York, NY, USA. ACM.

Kuhn, T. and Thomann, O. (2006). Eclipse corner article: Abstract syntax tree. http:// www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. Accessed: 2013-05-30.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

McQuillan, J. A. (2011). *Using Model Driven Engineering to Reliably Automate the Measurement of Object-Oriented Software*. Phd thesis, National University of Ireland, Maynooth.

Puybaret, E. (2013). Sweet home 3d - draw floor plans and arrange furniture freely:. Accessed: 15-09-2013.

QUASAR (2013a). Java facade and code generator for use (uml-based specification environment). http:// code.google.com/p/j-use/. Accessed: 2013-05-30.

QUASAR (2013b). Metamodel driven measurement (m2dm) tool. https://code.google.com/p/m2dm/. Accessed: 2013-09-06.

Sauer, F. (2013). Metrics 1.3.6. http:// metrics.sourceforge.net/. Accessed: 2013-05-30.

The Eclipse Foundation (2013). Eclipse java development tools (jdt) overview. http://www.eclipse.org/jdt/overview.php. Accessed: 2013-05-30.

Wilkie, F. G. and Harmer, T. J. (2002). Tool support for measuring complexity in heterogeneous object-oriented software. In *ICSM'2002*, pages 152–161.

APPENDIX

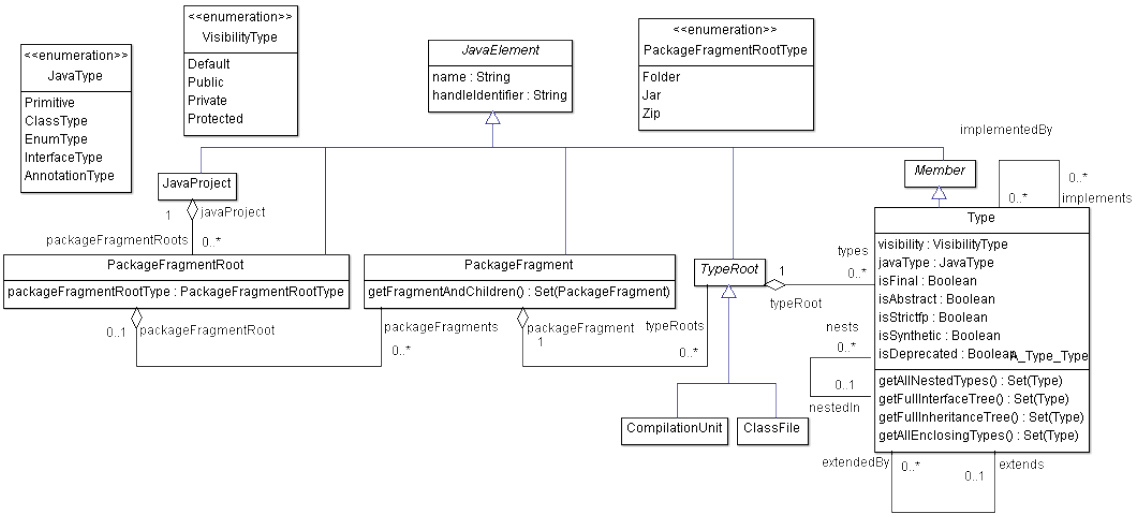

Figure 1: Eclipse Java Metamodel - Java Project Structure.

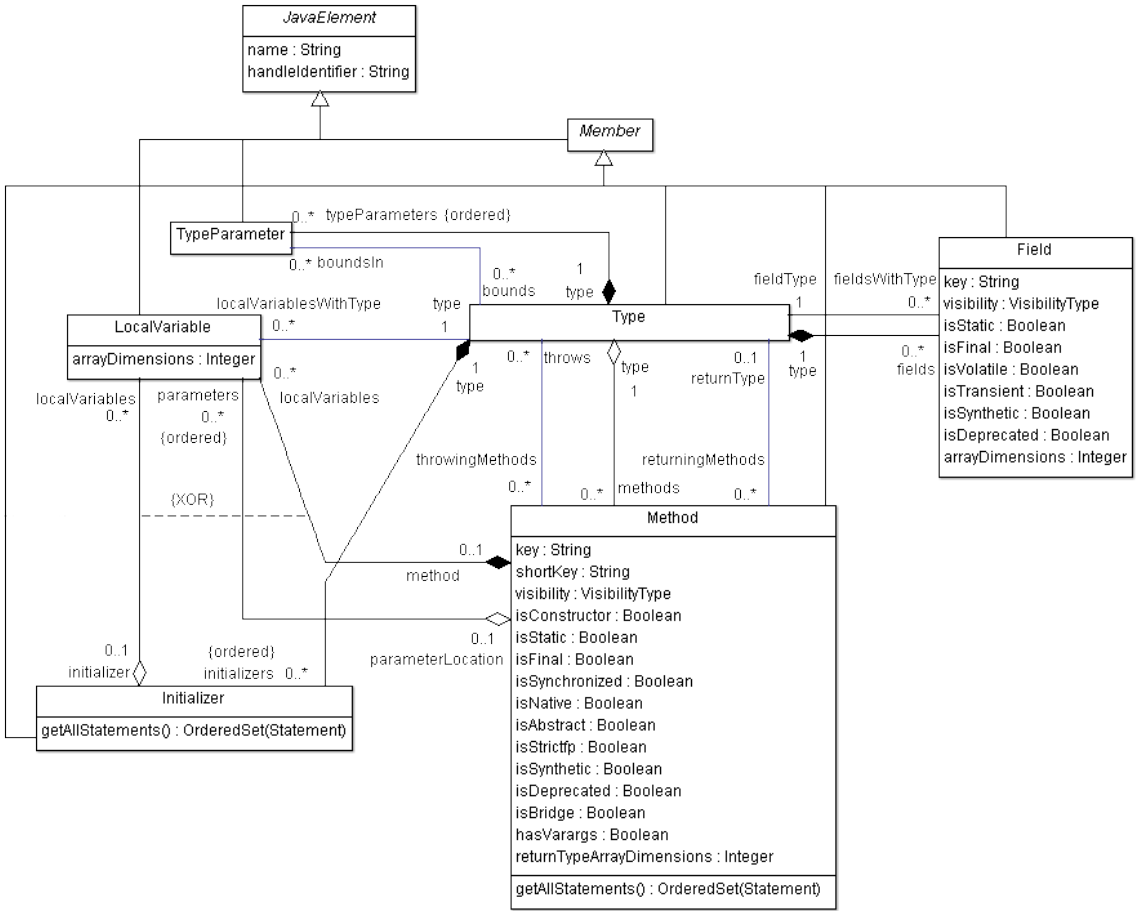

Figure 2: Eclipse Java Metamodel - Type Components.

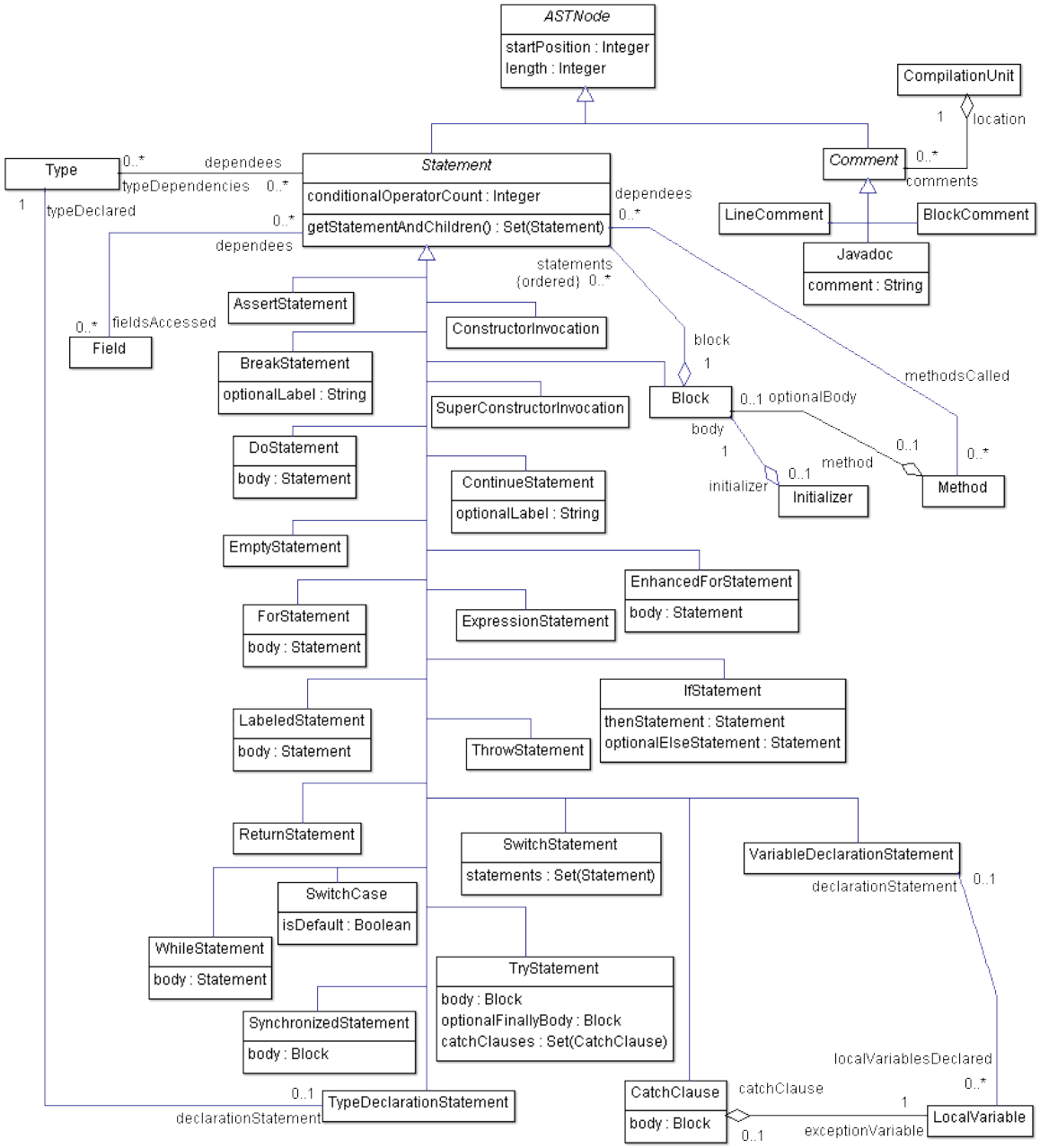

Figure 3: Eclipse Java Metamodel - Abstract Syntax Tree Components.