ViMC – Interactive Tool for Measuring Software Applications

Dan C. Cosma, Oana Gugea, Tamara Avramovic
Department of Computer and Information Technology
Politehnica University Timişoara
Romania
dan.cosma@cs.upt.ro

***Abstract*—Modern software applications are complex, and their maintenance implies elaborate analysis tools that reveal their structure and functionality, for purposes like software understanding or quality assessment. This paper introduces *ViMC*, a tool we have developed to support software engineers in analyzing the structure of object-oriented software applications, by focusing on measuring the various aspects that define the properties of the entities within a software project. The tool provides a set of predefined software metrics that can be applied to sets of classes and packages within a given software application, while allowing the user to dynamically create new metrics, as needed. The definition of custom metrics is done either through programming, or by interactively combining existing metrics via a user interface.**

***Index Terms*—software engineering, software metrics, reverse engineering, software understanding**

I. Introduction

Software applications are nowadays very complex systems, developed using different types of technologies, based on various frameworks, and consisting of large amounts of code and data artifacts. They are usually built by large teams, over significant periods of time, and their maintenance requirements are vital for their success.

Among the main challenges in the software engineering field there is the task of managing this complexity, by enabling the engineers to easily *understand* the systems as they evolve, even when they were built by other parties or when new teams need to take over the development. Another essential concern is assessing the structural properties of software systems in order to evaluate their design and implementation quality, to capture design flaws or to plan changes. For these purposes, software engineers employ specialized methodologies supported by specific tools that help them analyze the systems, extract the relevant information, and provide the knowledge items needed to understand them.

One of the most widely used approach is *reverse engineering* the software system [1] which basically consists of identifying the system components and their relation to each other, then representing the system at a higher level of abstraction fit for capturing the aspects required by the engineer's purpose. Many methodologies in this area are supported by tools that automatically read the application's source code, create a model of the software, and employ specific techniques that work on that model to assess system properties. Software metrics are often used as essential components of these techniques, custom built to measure those properties that are deemed important by the purpose of the methodology.

ViMC is a tool we developed to support software engineers in their work by providing an easy-to-use and flexible framework for assessing metrics-based properties of an application, focused on the feature of enabling the engineers to easily build a library of metrics fit for their purpose.

The tool uses static analysis, works on Java applications, and is developed as a plugin for the popular Eclipse IDE, in order to help the analysis process within a familiar environment. *ViMC* reads the source code of the application, creates a model to represent it internally, and provides flexible ways to create metrics that can then be easily used on system components to assess their properties.

The tool comes with an initial library of main metrics, while the user is enabled to extend this library by either defining completely new metrics, or combining existing ones. To ensure flexibility, new metrics can be added to the tool both by writing small sections of Java code, and by using a dedicated user interface.

The rest of the paper will present the related work in the field, then discuss the design-related aspects regarding the tool, and provide insight into the ways the tool can be used to extend the metrics library.

II. Related Work

There are various approaches and tools that allow software engineers to analyze systems by reverse engineering, in order to achieve software understanding or support system evolution and maintenance.

Amalfitano et al. [2] introduce a reverse engineering process focused specifically on Web applications, based on dynamic analysis that focuses on modeling an application's presentation layer. Their process is tailored for the particular kind of applications they target, and the approach differs from ours as it is based on dynamic, rather than static analysis.

Ricca and Tonella [3] identify static Web pages in a site suited for transformation in dynamic pages, by applying a clustering technique based on a similarity software metric. The metric is important in the approach, as it provides means of

978-1-5386-4444-7/18/$31.00 ©2018 IEEE

understanding which parts of the code can be refactored into a same component that generates dynamic content. The software understanding is achieved in this case by focusing on a custom metric, but the tools they use do not allow adding new metrics or using new measured properties.

Another analysis process aimed on system understanding is that employed by Pinzger et al. [4], who extract information from distributed applications to find clues about their architecture. They analyze the code to identify components and measure their inter-dependency, while building a model to describe the system.

An interesting approach was developed by Cox et al [5], who define a "Dependency Freshness" attribute that assesses the way a system depends on third-party components. The attribute is calculated using metrics, which are different from ours, in that they are based on generic system-wide properties, such as version numbers, rather than measuring structural properties within the code. In fact, we believe our tool could be valuable when creating a methodology that complements theirs by using the code-related information to better assess the strength of the dependencies they measure.

We were ourselves involved in developing tools for software understanding in the past, and we have also previously worked on related technologies. In an approach for understanding distributed software systems, we have developed the tool called niSiDe [6], which is used for reverse engineering and assessing the properties of complex software. We have also developed a tool for analyzing Web applications [7], which uses visualization to help engineers understand the interaction patterns between system components.

Rather than being built for measuring the quality of software products like commercial tools such as SonarQube [8], our tool focuses on helping researchers develop novel analysis techniques through step-by step experimentation with custom-built sets of metrics, at the desired level of detail within the code. The researcher may later decide to include the discovered techniques in more complex tools or as parts of higher-level software engineering methodologies.

Tools are built using tools, and we have based the development of *ViMC* on the XCORE framework built by Stefanica and Mihancea [9], which provides the means for modeling software systems in a uniform way, regardless of the actual underlying model. The advantage is that future developments of our tool will be able to integrate with other analysis tools we plan to write based on XCORE.

III. Modeling the analyzed application

ViMC is written in Java, and uses the XCORE framework mentioned above. In order to provide support for the software engineers directly in an environment specific to application development, we chose to implement our tool as a plugin for the widely-used Eclipse IDE. This way, users will not have to switch context when applying the various stages of their analysis methodology, therefore being able to use the same familiar environment they normally use.

A. Main entities

The target of the tool is the static analysis of existing Java applications, starting from their source code. For this purpose, the first step of any such approach is, necessarily, loading the source code, and creating a representation of the various code fragments so that they can be analyzed. In other words, this usually means that the tool involved has to build a certain *model* of the application. The types of entities described by a model are various in nature, and the concepts they refer to depend on the aspects the engineer needs to assess. Any object-oriented system can be represented with a model using basic concepts such as classes, methods, fields, etc.. However, if we need to assess specific properties of a system, for example those that describe their role in a Web application, we may use additional or different types of entities to represent the system, such as views, controllers, data objects. Using the established software engineering terminology, we say we can choose to define different *meta-models* for representing application according to the purpose of our analysis.

One of the advantages of the XCORE framework we use in *ViMC* is that the definition of meta-models is done in a way that allows for easy integration with other similar tools that use related, but not entirely similar, concepts. The meta-model can be uniformly described, regardless of the actual backend the tool uses to extract the information from the analyzed application. In our case of using Eclipse as environment, the backend is represented by the specific Abstract Syntax Tree (AST) of the analyzed application, created at runtime by the Eclipse JDT framework [10]. We parse this tree, and use the detailed information provided by it when necessary, but the higher-level concepts are defined uniformly in our meta-model.

The metamodel we have defined and used for representing the target system consists of the following basic types of entities:

- *MPackage* - models a Java package
- *MClass* - models a Java class
- *MInterface* - models a Java interface
- *MMethod* - models a Java method

For the purposes of our tool, we decided that these entities were sufficient in order to represent the application, in a manner that allows for flexibility in defining metrics to apply on the system. The underlying objects that are used in the implementation of these entities, are naturally those belonging to the Eclipse JDT Core Model, such as IType, IField, IMethod, etc.

B. Metrics

An additional main concept defined by our tool models the metrics themselves. Because our main purpose is to create a library of metrics that the user can easily and dynamically extend, we needed to create a first-class entity that models this concern. While this entity does not describe the analyzed system per-se, it definitely counts as the representative of a main concept in our approach, therefore it can be treated at the same level as other meta-model entities.

A key requirement in this respect was to provide a way for describing metrics that can subsequently be re-used to define other metrics, for as may times as needed. The natural way to do it was to create an abstract class that will serve as the hierarchic base for all the metrics in the system. We have called this entity *MetricInterface*, and all metrics, either belonging to the tool, or created by the user must be extend it. The class defines a method that must be implemented by all derived classes, called calculateMetric(), which contains the actual implementation of the respective measurement.

The signature of the calculateMetric() method is important, in that it defines the way metrics can interact with system entities in order to compute their properties. We define two separate sets of entities that this method applies to:

- the source set,
- the destination set

Both are basically lists of entities (classes, interfaces, etc.) that are given as arguments to the calculateMetric method, in order to provide a flexible way of describing the objects the metric looks at when computing the property. A particular metric may use both these arguments, or it can only use one of them, depending on its purpose. Similarly, the length of the two lists and the meaning of the objects that belong to them depend entirely on the metric's semantics, and can vary from a single one element list, in the case of metrics that apply to a single entity, to variable length lists, where necessary.

Moreover, although we have used the names "source" and "destination" to designate these lists, this doesn't imply any semantic predetermination for the metrics the user can add. The two names are useful when metrics calculate, for example, direct dependencies (e.g. method calls, references, etc.) between a "source" set of classes and a "destination" one, but in a more general case the meaning of the two sets is up to the engineer.

The calculateMetric() method returns a value of type *double*, which represents the value of the property calculated by the metric on the entities it was applied to. Because all metrics conform to this rule, they can be easily composed with each other in arithmetic expressions when necessary.

ViMC defines several pre-defined metrics, which are implemented by using specific property calculators in XCORE-supported entities (XMetrics), as depicted by Figure 1. Additional metrics can be created, and metrics can be combined with each other as discussed in the following sections.

IV. Assessing system properties

As mentioned above, *ViMC* provides means for applying various measurements on the analyzed software system, as well as defining new metrics at user's discretion. In order to ease the system assessment, our tool implements the following features:

- Provides a set of predefined metrics that are readily available for the user,
- Allows creating new metrics directly from the user interface, by building expressions based on existing metrics,
- Provides complete flexibility by defining a mechanism that can be used for adding new metrics through programming

We discuss each of these features as follows.

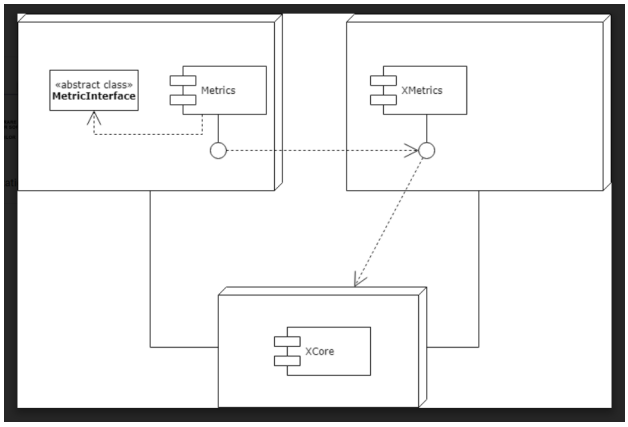

Fig. 1: Metrics and their relation to XCORE

A. Predefined metrics

In order to provide the user with as many options as possible, *ViMC* defines a comprehensive set of about 30 software metrics that are available from scratch. The selection of metrics covers various common and widely-used properties of system entities, and are written in such a way that they can be re-used to participate in defining more complex metrics.

The predefined metrics are:

- number of public methods
- number of private methods
- number of protected methods
- number of abstract methods
- number of accessor methods
- number of calls to a method in workspace
- number of implementations of interface in package
- number of implementations of interface in project
- number of extending interfaces
- number of interfaces implemented
- number of implementations of class in package
- number of implementations of class in project
- number of references to class in workspace
- number of references to class in project
- number of references to class in package
- base class overriding ratio
- number of constructors
- number of inheriting classes
- number of public attributes
- number of private attributes
- number of protected attributes
- number of public members
- number of private members
- number of protected members
- number of abstract classes in package
- number of classes in package

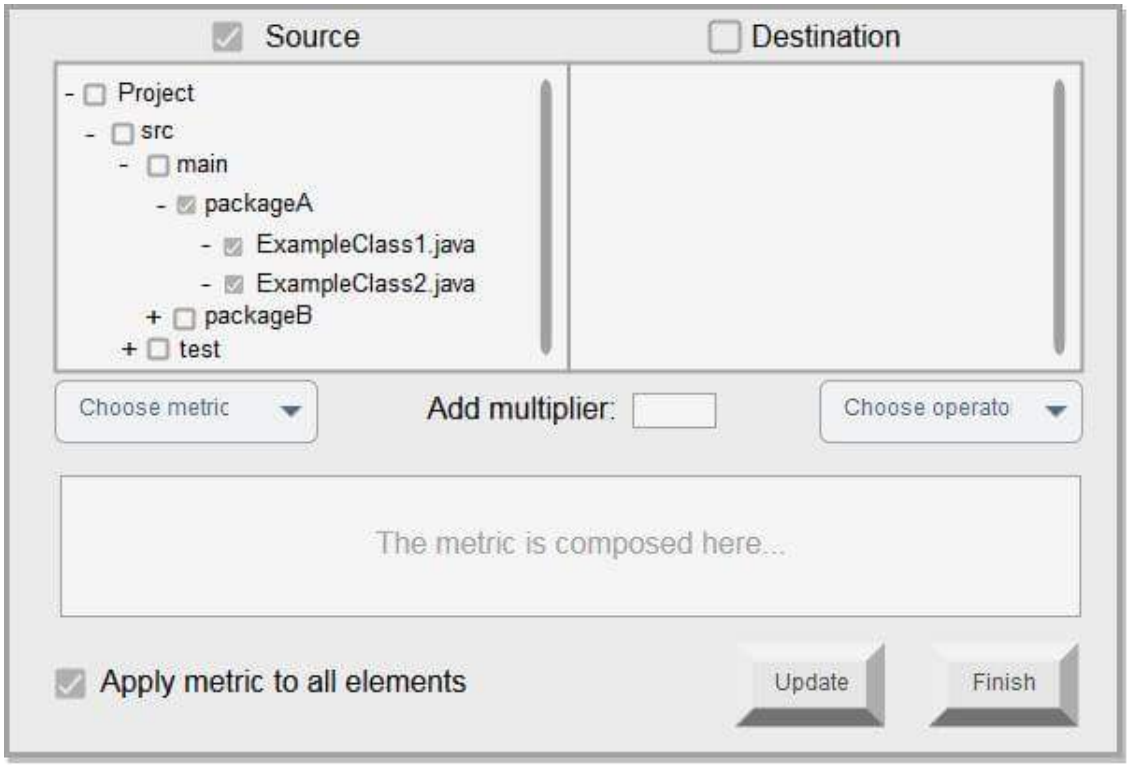

Fig. 2: Adding new metrics using the UI

- number of inheritances in package
- number of interfaces in package
- average number of arguments in method
- number of arguments to a method
- number of methods

Depending on their goal, they can be applied at various levels: to basic system entities such as classes, to groups of classes such as packages, or can even assess properties at the system level. Each metric returns, when applied, a value that represents the specific measurement made on its selected target. The metrics implementation is uniform as they adhere to the same conceptual interface, and makes use of the features provided by XCORE and the underlying model specific to JDT.

The actual application of both predefined and custom metrics to show the calculated measurements is done using the Insider visual component provided by XCORE.

B. Adding metrics through the user interface

Existing metrics can be joined together to form more elaborate metrics. For this purpose, the user interface provides an interactive window the engineer can use for composing metrics in expressions (Figure 2). In order to support metrics that can only be applied to specific entities, the user is presented with two optional lists of system entities, where instantiations of the two lists of objects that will be passed as arguments to the metric can be selected if needed.

For any new metric, the user can select, step by step, any of the existing metrics as parts of an expression that defines the new metric (Figure 2). For example, if the user wants to create a new metric that is based on two existing metrics *M1* and *M2*, and the needed formula is *2*M1+7*M2*, the user will have first to select metric *M1*, then use a multiplier of *2*, select a ”+” operator, then press ”Update”. The current expression will show up in the lower window area as *”2*M1+”*. At this point, the user can select the second metric, multiplier *7*, and press ”Update” again to see the expression updated to *”2*M1+7*M2”*.

C. Adding complex metrics through programming

Not all measurements needed by an engineer can be described as simple arithmetic expressions between existing measurements. More often than not, static analysis techniques need very particular types of assessments related to the system entities, that can only be defined in terms of algorithms. Imagine, for instance, that we need to calculate the number of methods in a package called from within a class external to that package. There is no formula to capture this need, and the only way of achieving this is to instruct the tool what are the steps involved in the actual calculation: take each method in the caller class, verify whether it belong to a class in the target package, increment the value if true, repeat.

Having this fact in mind, we had to find a way to provide the software engineer with the means of achieving this level of flexibility when defining metrics, while keeping things simple

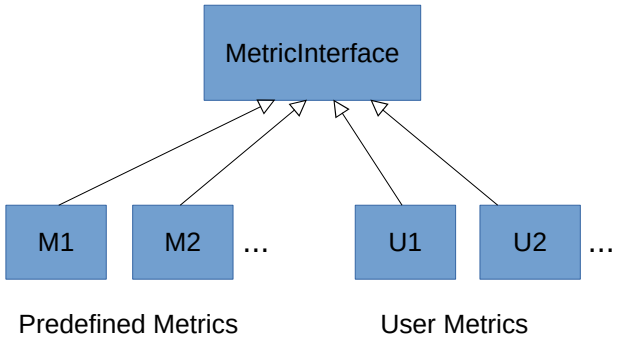

Fig. 3: The MetricInterface

enough to let her focus on the important task of defining the algorithm itself, rather than being concerned with various details regarding the minutia of static analysis. We decided that the best way to do it was to shape the layout of our reusable metrics framework in such a way that they can be easily understood and used by a third party to directly write new metrics. The solution we came up with is based on the following principles:

- *Hide the details regarding the system representation, and the analysis framework we use, as much as possible*. This is why we decided to hide all details related to XCORE and to implement our own layer of metrics based on the MetricInterface entity, as discussed above.
- *Design the metric interface in a flexible manner, while allowing for metrics to combine with each other*. For this purpose, we decided that all metrics should return a floating point number, and designated two optional entities lists as the arguments to each and all metrics (the ”source” and ”destination” sets, described in the previous section).
- *Allow as much flexibility as possible for measurements that need a higher level of detail.* This is the reason why we did *not* deny the engineer access to the lower-level artifacts in the system representation, such as the underlying JDT objects that directly represent code fragments as loaded by Eclipse.

The solution is based on defining the MetricInterface abstract class, and allowing users to create their own classes derived from it (Figure 3). The only method users have to implement is *calculateMetric()*, which is the actual body of the metric itself, and nothing more. All the other details are handled by the tool. The engineer needs to understand only a small part of our framework, particularly the MetricInterface, and the behaviour of the various predefined metrics we provide. Nevertheless, if the user wants and has the knowledge, it can delve into lower level details, such as querying JDT-specific objects.

We can consequently define two types of metrics that can be implemented by the users through programming:

- *Composed metrics*, which only use existing metrics and compose them using simple algorithms;
- *Detailed metrics*, which can freely use existing metrics, more complex algorithms, and even lower-level insight into the system representation.

V. A CASE STUDY: DEFINING METRICS BY PROGRAMMING

This section presents an example of how custom metrics can be written by the user of our tool, with various degrees of complexity in defining the actual measurement. We will discuss both types of approaches mentioned in the previous section, Composed metrics and Detailed metrics.

A. Example of a composed metric

```
package metrics.utils;

import java.util.ArrayList;
import metrics.classes.NoOfAccessorMethods;
import metrics.classes.NoOfConstructors;

public class ExposedMethods extends MetricInterface
    {
@Override
public double calculateMetric(ArrayList<Object>
    source, ArrayList<Object> destination)
{
  NoOfConstructors metric1 = new NoOfConstructors();
  metric1.calculate(source, destination);
  double noOfConstructors = 0;

  NoOfAccessorMethods metric2 = new
      NoOfAccessorMethods();
  metric2.calculate(source, destination);
  double noOfAccessorMethods = 0;
  try {
    noOfConstructors = metric1.getMetricValue();
    noOfAccessorMethods = metric2.getMetricValue();
  } catch (MetricNotInitialised e) {
    System.out.println("Metric␣Not␣Initialised");
    e.printStackTrace();
  }

  return (noOfConstructors + noOfAccessorMethods);
}
}
```

To show how simple metrics can be easily implemented, the above listing contains the complete code for implementing a custom metric that calculates the number of methods exposed by a class. The code only uses one of the two argument lists received by *calculateMetric()*, and indeed only the first element of it, representing the analyzed class. The metric computes the result by using the output of two predefined metrics, which calculate the number of constructors and the number of accessor methods defined in the class, respectively. The two helper metrics also use only the first element of the source list argument for getting the class to analyze, which is documented in our tool's documentation.

Note: when implementing a new metric, one has to override the abstract method calculateMetric(), while for applying a metric, the call should be made to the calculate() method. This is because the calculate() method is implemented using the Template Method pattern, and it calls calculateMetric().

B. Detailed metrics

The code below shows how a more advanced user can implement a metric that uses JDT-specific underlying objects for computing a complex metric that needs specific processing of the analyzed system entities. The metric calculates the number of classes within several packages (given in the "destination" list) that extend specific classes enumerated in the "source" list provided as the first argument.

The metric parses the lists received as arguments to the *calculateMetric()* method, and uses methods specific to the JDT-provided underlying objects to find specific attributes, such as their names. The level of detail available this way allows for maximal flexibility when designing metrics, while the user does not need to deal with the more involved issues related to the static analysis and system representation.

```
package metrics.utils;

import java.util.ArrayList;
import java.util.List;

import org.eclipse.jdt.core.IPackageFragment;
import org.eclipse.jdt.core.JavaModelException;

import vimc.metamodel.entity.MClass;
import vimc.metamodel.entity.MPackage;
import vimc.metamodel.factory.Factory;

public class NoOfDerivedClasses extends
    MetricInterface {

@Override
public double calculateMetric(ArrayList<Object>
    source, ArrayList<Object> destination) {
  // source: list with classes for which we search
      for sub classes
  // destination: list of packages in which we
      search the subclasses
  MClass mClass = null;
  MPackage mPackage = null;
  int noOfClassExtensions = 0;

  for (int i = 0; i < source.size(); i++) {
    mClass = (MClass) source.get(i);
    // current class name
    String superClassName =
        mClass.getUnderlyingObject()
      .getElementName();
    for (int j = 0; j < destination.size(); j++) {
      // for all packages in the destination list
      mPackage = (MPackage) destination.get(j);
      List<MClass> mClassesInPackage =
          mPackage.classesGroup().getElements();
      // for all classes in the package
      for (MClass eachMClass : mClassesInPackage) {
        String nameOfClass;
        try {
          nameOfClass =
              eachMClass.getUnderlyingObject()
            .getSuperclassName();
          // count the number of classes which have
              as super class the class given as
          // argument
          if (nameOfClass != null) {
          if
          (nameOfClass.equals(superClassName))
          {
              noOfClassExtensions++;
          }
          }
        } catch (JavaModelException e) {
          e.printStackTrace();
        }
      }
    }
  }

  return noOfClassExtensions;
}
}
```

VI. Conclusions

This paper presented *ViMC*, our approach for building a tool that enables software engineers to analyze object-oriented Java applications by building a library of metrics that assess system properties. The metrics can either be added using a specific user interface, or can be described by implementing minimal amounts of Java code that reuse existing metrics and allows for different levels of access to the modeled system's entities.

Future development of this tool aims at further refining the user interface for interactively adding metrics, providing a larger set of predefined metrics, as well as tailoring the tool to specific software engineering analysis scenarios, such as assessing dependencies between different applications, or finding structural traits related to code reuse.

References

[1] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, pp. 13–17, Jan. 1990.

[2] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "An iterative approach for the reverse engineering of rich internet application user interfaces," in *ICIW 2010*.

[3] F. Ricca and P. Tonella, "Using clustering to support the migration from static to dynamic web pages," in *IWPC '03*. IEEE CS Press, 2003.

[4] M. Pinzger, J. Oberleitner, and H. Gall, "Analyzing and understanding architectural characteristics of COM+ components," in *IWPC '03*. IEEE CS Press, 2003.

[5] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 109–118. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819027

[6] D. C. Cosma, "niSiDe: Interactive tool for understanding distributed software," in *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008) Timisoara, Romania*, 2008.

[7] D. C. Cosma and P. F. Mihancea, "Understanding web applications using component based visual patterns," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 281–284. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820324

[8] SonarSource. SonarQube – Continuous Code Quality. [Online]. Available: https://www.sonarqube.org/

[9] A. Stefanica and P. F. Mihancea, "Xcore: Support for developing program analysis tools," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 462–466.

[10] Eclipse Foundation. Eclipse Java development tools. [Online]. Available: https://www.eclipse.org/jdt/