Building an Operable Graph Representation of a Java Program as a basis for automatic software maintainability analysis

Sébastien Bertrand, Pierre--alexandre Favier, Jean-Marc André

Building an Operable Graph Representation of a Java Program as a basis for automatic software maintainability analysis

Sébastien Bertrand[1,2], Pierre-Alexandre Favier[1,3], and Jean-Marc André[1,3]

[1]IMS Laboratory, University of Bordeaux, UMR 5218 CNRS, France
[2]onepoint, Sud-Ouest, France
[3]ENSC, Bordeaux INP, France

s.bertrand@groupeonepoint.com, {pierre-alexandre.favier, jean-marc.andre}@ensc.fr

Résumé

Dans le cadre d'un projet de recherche concernant l'évaluation de la maintenabilité logicielle en collaboration avec l'équipe de développement, nous nous sommes intéressés à l'utilisation fréquente de métriques comme prédicteurs. De nombreuses métriques existent, souvent avec des implémentations opaques et discutables. Nous affirmons que les métriques mélangent l'évaluation de la présentation, de la structure et du modèle. Afin de se concentrer sur les vrais défauts de maintenabilité détectables, nous avons calculé des métriques uniquement basées sur la structure du programme. Notre approche a consisté à analyser le code source de programmes Java *comme un graphe, et calculer les métriques dans un langage de requête déclaratif. À cette fin, nous avons développé* Javanalyser *et implémenté 34 métriques en utilisant* Spoon *pour analyser les programmes* Java*, et* Neo4j *comme base de données de graphes. Nous allons montrer que le graphe de programme constitue une base solide pour calculer les métriques et mener de futures études d'apprentissage automatique pour évaluer la maintenabilité.*

Mots-clés

Maintenabilité logicielle, Analyse de programme, Graphe de programme.

Abstract

As a part of a research project concerning software maintainability assessment in collaboration with the development team, we were interested in the frequent use of metrics as predictors. Many metrics exist, often with opaque and arguable implementations. We claim metrics mix the assessment of presentation, structure and model. In order to focus on true detectable maintainability defects, we computed metrics solely based on the structure of the program. Our approach was to parse the source code of Java *programs as a graph, and to compute metrics in a declarative query language. To this end, we developed* Javanalyser *and implemented 34 metrics using* Spoon *to parse* Java *programs and* Neo4j *as graph database. We will show that the program graph constitutes a steady basis to compute metrics and conduct future machine-learning studies to assess maintainability.*

Keywords

Software Maintainability, Program Analysis, Program Graphs.

1 Introduction

Software maintainability is paramount to reducing the cost of systems in time [15, 17]. Our research project concerns software maintainability assessment working in collaboration with the development team [13]. As part of this effort, we began to reproduce a study from Schnappinger *et al.* [33], because their work is based on a recent, high quality software maintainability dataset [32]. This introduction presents what maintainability and metrics are, before continuing on the importance of the program structure and having clear and unambiguous metrics. Finally, we will present *Javanalyser*, the tool we developed to answer our research questions.

Maintainability is defined as the efficiency with which the software can be corrected, improved or adapted to changes in the system or in the specifications, either technical or functional. According to the ISO 25010 [25], maintainability is composed of five subcharacteristics:

- Modularity: degree of decoupling between components;
- Reusability: degree of potential reuse of a component;
- Analysability: degree to which the implementation of a component can be understood and debugged;
- Modifiability: degree to which a component can be modified without introducing defects in other components;
- Testability: degree to which a component can be tested against a set of technical and functional specifications.

Most studies try to predict maintainability by using metrics as predictors [11, 20]. There is a great number of existing

metrics [19], which can be broadly categorized in product metrics relating to the structure of the software, for example the number of lines of code, and process metrics relating to the activity of developing the software, for example the number of hours needed to correct a bug [22]. Product metrics are trying to pinpoint the intrinsic cause of maintainability defects, and process metrics are extrinsic giveaways of these defects. We can metaphorise that product metrics are the disease while process metrics are the symptoms.

The development of a software in general, and maintainability in particular, can also be approached from three angles:

- The presentation of the source code, encompassing all the style rules applied to write the program, for instance how many classes there is per file, where a blank line should be inserted for clarity, or the naming convention of classes and methods;
- The structure of the source code, characterized by the components of the program, their responsibilities, and the relationships that exist between them;
- The model of the problem the program is trying to solve, which can be highlighted or not by the organisation of the code, for instance the Domain Driven Design [21] explicitly puts the model forward.

The presentation of the source code can be a true issue in some context, as shown by obfuscation tools that can modify all named elements to reduce readability. But when developing an application, presentation can be put under control, as there exist many tools that enforce formatting and naming convention, such as Prettier [5] for Javascript or Pylint [6] for Python.
The modelisation of the problem to solve is not only related the implementation of the program, but to the whole development process and the business maturity of the development team. Despite being an important stake in software development, problems related to modelisation go beyond the scope of program analysis. A program that perfectly respects the subcharacteristics of maintainability defined above can implement a model completely out of phase with the business, thus making it very difficult to evolve when new requirements emerge [18]. Typically, in this case, product metrics would remain stable and process metrics would drastically increase. The modelisation angle can be viewed as the goal the team is trying to reach, when the goal is wrong for any reason the risks are high to encounter maintainability problems.
On the other hand, the internal structure of the program can be very different for equivalent modelisation of the business (*i.e.* the same functional scope), thus harbouring maintainability flaws. Analysability is obviously related to the presentation of the code, but bad encapsulation or factorisation can also lead to readability issues, such as methods with too many arguments or very big classes difficult to apprehend. It seems legitimate to relate the modularity, reusability and modifiability to the sole structure of the program, because their main concerns are about the components of the program. Testability is related to both the model and the structure implemented by the source code, because tests are designed to check that the implementation (the program) matches the specifications (the model).
Then, however related to the presentation and the model, it seems that the structural design of the program is the main factor impacting the maintainability.

Moreover, while reproducing the study from Schnappinger *et al.* [33], we encountered many problems collecting metrics. Many tools exist and many metrics have been studied [12, 28]. Some studies are based on unmaintained or deprecated tools, and each tool implements metrics computation differently, often with very little documentation available.

We make the assumption that focusing on the structural analysis of a program will allow us to detect predictable maintainability defects, that depend solely on the program and not an external context such as the chosen modelisation. We developed our tool called *Javanalyser*, that parses the abstract syntax tree of a *Java* program and load its structure as a graph within a graph database (*Neo4j* [4]). We focused on having an operable graph and tested it by implementing a set of 34 product metrics as *Cypher* queries [23], leveraging declarative programming to have concise, mutable and explicit definition of metrics. *Javanalyser* is available under the open-source MIT licence. The goal is to build a steady basis to conduct machine-learning studies to assess maintainability.

2 Method

This section explains in detail how we built *Javanalyser* to process a *Java* program and the design choices we made. Basically, our approach was to parse the source code, compute the corresponding graph, and load it into a graph database. When implementing the metrics, we had to manage external references towards the projects' dependencies and define how to walk along the relationships of the graph. Finally, we were able to compute metrics based solely on the graph.

Parsing Java. We used *Spoon* [31] to parse *Java* programs, it produces an abstract syntax tree designed to be both complete and understandable for *Java* developers. Before, we considered two other parsers. At first, we wanted to use directly the *Eclipse Java Development Tools* [1], which is internally used by *Spoon*, but it was very difficult to make it run outside an Eclipse environment, and its documentation is sparse on this issue. Then, we tried JavaParser [2], which was simple to install within our solution. However, code references to external dependencies (typically specified by the `CLASSPATH`) were not properly parsed by its symbol solver in our tests. On the other hand, *Spoon* is able to parse properly all external references within a *Java* program. This point was paramount for being able to compute a graph from the abstract syntax tree (AST), as we must be able to detect that two leaves from the AST are referencing

the same type (a `Class` or an `Enum` in *Java*). An atomic element of the *Spoon* meta-model is a `CtElement`, which we encapsulated within a `ProgramElement` for easy manipulation. Hereafter, we will call "program element" the nodes from the AST produces by *Spoon*.

The graph. Internally, we implemented a `Scanner` in *Javanalyser* to walk through the AST produced by *Spoon* and compute a graph by collecting additional edges between references and referenced elements. The Scanner from *Spoon* do not automatically walk along these relationships to avoid infinite loop when scanning the AST. In this process, we also implemented some simplifications within the graph, for instance:

- we avoided creating vertices for `TypeReference` by linking the referencing program element to the referenced type directly;
- we chose not to specify obvious relationships such as `ThisAccess` which can be inferred from context;
- we avoided creating extraneous vertices for very simple elements, such as a *VariableAccess* without `Cast`, `Annotation` or `Comment`, that is only a reference to a `Variable`.

We only implemented reversible simplifications, that allow to infer the correct AST from the graph. These simplifications were designed to produce a graph that matches more intuitively the code. However, the trade-off of these simplifications was that the graph produced has a database schema depending on the context. We accepted this matter of fact, as our ultimate goal is to assess maintainability, we wanted to produce a graph that matches more closely the "point of view" of a developer.

Graph database. We used *Neo4j* [4] as the graph database. *Neo4j* allows to use a flexible property graph schema. A property graph is a labeled directed multigraph, sometimes called labeled multidigraph. A multigraph allows self-loop edges and parallel edges between nodes. A labeled multidigraph has labeled vertices and arcs. In *Neo4j*, vertices can have multiple labels although edges can only have one label called type. That is why *Neo4j* matched perfectly our requirements, each program element of the AST from *Spoon* being precisely typed (for instance `CtClass` or `CtConstructor`), and having a defined role in its parent's program element (for instance `FOR_INIT` or `EXPRESSION`). The trickiest part was to load the data as quickly as possible, because parsing an actual *Java* program from scratch involves a huge graph.

External references. There are two types of references, internal and external to the parsed *Java* program. Internal references are declared in another part of the parsed *Java* program. External references reference program elements declared in dependencies (sometimes called libraries) used by the parsed *Java* program, typically passed along the `CLASSPATH` variable for a *Java* program. We paid particular attention to parse and identify each of these dependencies. If there is a missing Jar within the `CLASSPATH`, the corresponding external references are flagged as broken references. But if these external references are known, we implemented a walk along their parents' nodes within the AST to provide potential useful additional information. Finally, we flagged external references as "shadow", which is the term used by *Spoon*. Identifying external references is useful when implementing metrics, according to their definition.

Walking the graph. There are many types of relationships within the graph. Because we lost the presentation information conveyed by the segregation of code in files, we had to define how to walk along relationships within the graph. In other words, we wanted to be able to query program elements belonging to a class, leaving apart program elements belonging to other classes. Actually, we classified relationships' types in six sets:

- organisational: describing the structure of modules, packages and classes;
- inner: describing how nodes belong to one another, for instance a local variable belongs to a method which belongs to a class;
- type: linking typed elements to their type (`Class`, `Enum`, ...);
- outer: describing references to potential outer elements, which are typically members of other classes;
- comment: linking program elements to their comment;
- flow: describing the control and data flow of the code.

These sets of relationships were instrumental for the design of the metrics queries.

Metrics. We implemented the computation of metrics in *Cypher* [23], the declarative query language for property graphs associated with *Neo4j*. To ensure that the implementation of our metrics was correct, we qualitatively compared our results with *SonarQube* [7] and *SourceMeter* [8] to detect potential defects within our queries. This whole process helped us to iteratively design *Javanalyser*. Moreover, every metric we planned was successfully implemented and tuned to our expectation. This final task showed the flexibility and versatility of the computation of metrics based on the graph.

3 Results

Foremost, the goal is to build a steady basis to conduct machine-learning studies to assess maintainability. As we focus on the structure of the program, we want to have a presentation-independent representation. This is why we build *Javanalyser* to represent code as a graph, depending only on the structure of the code, and allowing to easily extract data such as metrics. *Javanalyser* parses a *Java* program, produces a *Neo4j* [4] graph, and outputs metrics in a CSV file. The source code

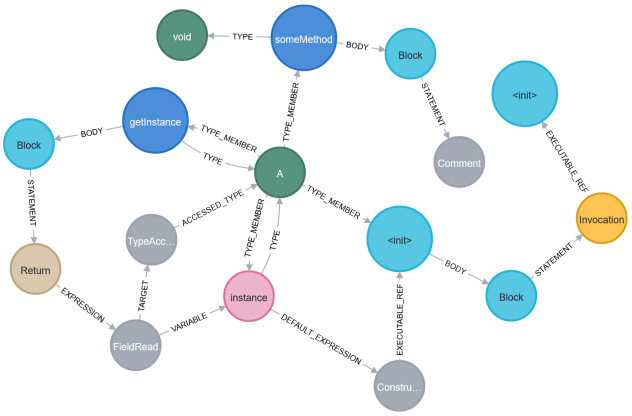

Figure 1: Screenshot of the graph of the singleton pattern

of *Javanalyser* is available under the free (as in freedom) MIT licence on GitLab at `https://gitlab.com/onepoint/research/javanalyser`.

Figure 1 presents an example of the singleton pattern implementation from listing 1. The graph is a labeled directed multigraph. Each node represents an element from the source code. There are 89 existing node labels, for instance there is "`Class`", "`LocalVariable`" or "`While`". Edges represent oriented relationship between nodes, for instance in our example the class "`A`" has four members. There are 93 existing relationship's types, for instance "`TYPE_MEMBER`", "`DEFAULT_EXPRESSION`" or "`EXPRESSION`". All names (node label and relationship type) comes from the *Spoon* [31] meta-model, except "`STATEMENT_ORDER`" which we introduced to obviously keep track of the execution flow within a block.

```
package fr.onepoint.javanalyser.tests.singleton;

public class A {

  private static final A instance = new A();

  private A() {
  }

  public static A getInstance() {
    return instance;
  }

  public void someMethod() {
    // This is a method
  }
}
```

Listing 1: A singleton pattern in *Java*

Thirty-four metrics have been implemented as *Cypher* queries. Listing 2 shows an example with the implementation of the cyclomatic complexity. *Javanalyser* aggregates the results of these metrics for each class of the parsed *Java* program. Then, it produces a CSV file listing classes and associated metrics. Metrics computations are only based on the graph, no pre-computation is done by *Javanalyser*.

```
//Cyclomatic Complexity
CALL {
  MATCH (class:Class)
  OPTIONAL MATCH (class)-[:ANNOTATION|ARGUMENT|ASSIGNED|
     ASSIGNMENT|BODY|CASE|CATCH|CONDITION|
     DEFAULT_EXPRESSION|DIMENSION|ELSE|EXPRESSION|
     FINALIZER|FOREACH_VARIABLE|FOR_INIT|FOR_UPDATE|
     LEFT_OPERAND|NESTED_TYPE|PARAMETER|RIGHT_OPERAND|
     STATEMENT|TARGET|THEN|TYPE_MEMBER|TYPE_PARAMETER|
     VALUE *0..]->(node)
  RETURN class, node
}
WITH class, node
WHERE
  (node:Constructor AND NOT node.implicit)
  OR node:Method
  OR node:AnonymousExecutable
  OR node:If
  OR node:Conditional
  OR node:For
  OR node:ForEach
  OR node:While
  OR node:Do
  OR (node:Case AND EXISTS((node)-[:EXPRESSION]->()))
  OR node:Catch
  OR (node:BinaryOperator AND (node.operator = "AND" OR
     node.operator = "OR"))
RETURN class.id AS id, class.name AS class, count(
     distinct(node)) AS cyc
```

Listing 2: Cyclomatic Complexity Query

Implemented metrics include:

- Number of children [16], which is the number of classes that directly inherit from the class;
- Depth of Inheritance Tree [16], which is the number of parents the class inherits;
- Number of nodes from the class, which is the graph analog of the number of lines of code;
- Maximum of methods' number of nodes;
- Number of nodes within methods;
- Average methods' number of nodes;
- Nesting Level Else-If, which measure complexity as the depth of the maximum embeddedness of its conditional, iteration and exception handling block scopes;
- Cyclomatic complexity [29], which corresponds to the number of linearly independent paths;
- Maximum length of loops;
- Number of loops;
- Number of outgoing invocation, which is actually the number of executable referenced;
- Coupling between objects [16], which counts the number of non-inheritance related classes, *i.e.* the classes that are acted upon by the class or act upon the class (*e.g.* calling methods or holding instance variables);
- Maximum nesting depth;
- Number of methods [27];
- Cognitive complexity [14], which is an evolution of the cyclomatic complexity;
- Number of local attributes and methods declared in the class [27];

- Message-passing coupling [27], which is the number of call statements from the class to other classes;
- Response for a class [16], which is the cardinality of the set of local methods and the methods called by these;
- Lack of cohesion in methods [16, 24], which is the number of sets of methods bound by at least one common instance variable;
- Number of nodes embedded in blocks deeper than 4;
- Data Abstraction Coupling [27], which is the number of variables defined in the class and having an abstract data type.

Moreover, *Javanalyser* outputs two other CSV files. The first lists broken references, which denotes there was some missing dependencies that need to be passed as an argument to the command-line. The second represents the schema of the graph, *i.e.* the effective relationships between nodes' types. These relationships are weighted by counting the number of instances. This weighted schema can be used to compare at a large scale the programming style of different *Java* programs.

Figure 2 presents an extract from the graph produced by parsing *Art of Illusion*[1], which is around 100 kilo lines of code. This extraction presents an overview of the *Java* packages (in brown), and most classes (in green) and interfaces (in pink) from the project. *Javanalyser* parses this project and computes all metrics within approximately 5 minutes on an average computer[2]. A lot of the development effort was put into optimizing the processing speed, ultimately dividing the total execution time by 50. To date, we worked around 130 days and wrote about 1700 lines of code to develop *Javanalyser*, which we shared to the community under the open-source MIT licence. *Javanalyser* is a console application that can be pipelined within a more global process, and more Cypher queries can be easily added. This allows to use this tool for batch processing large datasets such as the *GitHub Java Corpus* [10].

4 Discussion

We designed our work as a unified maintainability-analysis framework, spanning from metrics computation to program representation. This discussion begins with the stakes of the definition of metrics, then we focus on how to count complexity with the case of `Optional` in *Java*. Finally, we will discuss program representation by covering *jQAssistant* [3], ontology-based program analysis, and graph representations.

Definition of Metrics. When we compared our metrics to *SonarQube* [7] and *SourceMeter* [8], it jumped out their implementations of metrics often differ. Even for a metric as plain as the number of lines of code, we counted many

[1] `https://github.com/ArtOfIllusion/ArtOfIllusion`
[2] Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, 16 Go of RAM

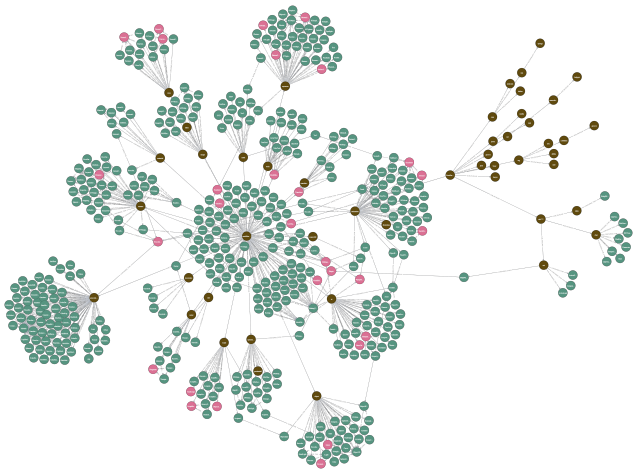

Figure 2: An extract of the graph produced by *Javanalyser* when parsing *Art of Illusion*

variations with none being equal as shown by an example from *Art of Illusion* in table 1. Moreover, reading documentations is nor practical nor always sufficient to grab all the subtleties of metrics computation. For instance the documentation of *SourceMeter* does not state the computation of cyclomatic complexity ignores nested classes, which is most surprising because the declaration on-the-fly of a nested class is clearly a source of complexity, and a backdoor for masking complexity if ignored.
Whether open-source or well documented, imperative implementations of metrics remains opaque, hard to discuss and hard to adapt according to the context. On the other hand, our declarative implementations of metrics within *Cypher* queries [23] embodies formal definitions and are all at once univocal, transparent, easily shareable and mutable.

Tool	Metric name	Value
SonarQube	Lines	480
	Lines of Code	368
	Comment Lines	28
SourceMeter	Lines of Code	314
	Total Lines of Code	453
	Logical Lines of Code[3]	233
	Total Logical Lines of Code[3]	357
	Comment Lines of Code	23
	Documentation Lines of Code	16
	Total Comment Lines of Code	34

Table 1: Example of lines of code metrics on the class `ActorEditorWindow`

How to count complexity. Most product metrics are basically specialized counters of some sort, for instance the cyclomatic complexity [29] counts the number of predicates[4], the depth of inheritance tree [16] counts the number of parent classes, or the message passing coupling [27] counts the number of "foreign" calls. Some metrics use different top-level aggregation operations, like the *average methods' number of nodes* that computes an average of counts. In all cases, these metrics base their computation on simply

[3] A logical line of code correspond to an executable statement.
[4] An operator or function that returns either true or false.

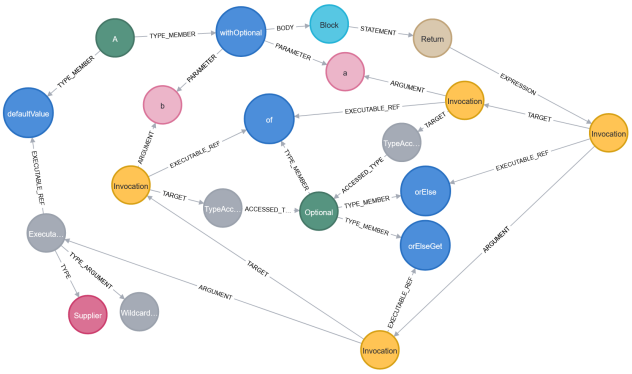

Figure 3: Screenshot of `withOptional` graph

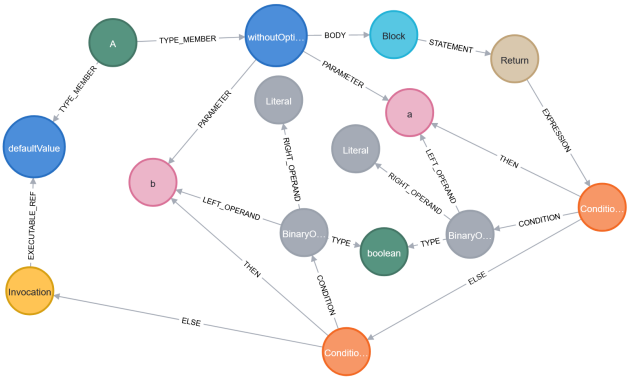

Figure 4: Screenshot of `withoutOptional` graph

counting targeted program elements. Interestingly, the lack of cohesion in methods [24] is more complex, as it is associated with the independent subsets of methods within a class.

However, deciding what to count for a given metrics is not obvious. Do we count indiscriminately each node or a subset of so-called statement nodes? Do we count external references? Do we count nested classes or inner classes? We can go even further by considering the `Optional` in *Java* versus the classic if-then-else structure. Listing 3 presents two methods, `withOptional` and `withoutOptional`. These methods have intrinsically the same behaviour but their computed *cyclomatic complexity* differ. We argue this difference is not legitimate, especially considering that the *cyclomatic complexity* was built to count the number of paths within a piece of code. Figures 3 and 4 show the use of `Optional` does not decrease the number of paths, which is coherent with the fact that these methods implement the same behaviour. The same reasoning can be made with *Java* `Stream` versus classic `for` loop.

Our approach allows to easily take into account new structures contributing to complexity by adding the corresponding selectors to the query (within `WHERE` clauses).

```
package fr.onepoint.javanalyser.tests.optional;

import java.util.Optional;

public class A {

  // Cyclomatic Complexity = 1
  String withOptional(String a, String b) {
    return Optional.of(a).orElse(Optional.of(b).
     orElseGet(this::defaultValue));
  }

  // Cyclomatic Complexity = 3
  String withoutOptional(String a, String b) {
    return a != null ? a : b != null ? b : defaultValue
     ();
  }

  String defaultValue() {
    return "Default Value";
  }
}
```

Listing 3: Code with or without optional

Program Representation. The *cyclomatic complexity* is built by considering the complexity of the associated program graph. Computationally, through graph theory, it can be reduced at counting the number of predicates. But its inception is based on a graph representation of the control flow of the program [29]. Lemberger and Morel use *cyclomatic complexity* to define a scale-invariant measure of complexity [26]. They also emphasize the idea that summation of metrics is not a natural condition to compute higher-level metrics, *i.e.* the complexity of a class is not the summation of the complexity of its methods.

This leads us to consider program representation, which must be first complete and accurate, possibly opinionated, but also freely interpretable to allow discussion and subsequent exploration by machine-learning algorithms. Our approach was to build a complete program graph based on a carefully simplified structure and leaving aside presentation. Müller *et al.* [30] have a similar approach, based on *Neo4j* [4] and *jQAssistant* [3]. However *jQAssistant* does not feed the graph with the content of the methods, making it impossible to compute low-level metrics.

Another approach involves ontology-based program analysis. Graph4Code [9] exploits RDF graphs where each program is represented in a separate graph, which are interconnected by fully-qualified function names (package.classname.function). Documentation resources, as well as StackOverflow and StackExchange posts are also represented in separate graphs and connected to program graphs by fully-qualified function names. Applications such as anti-pattern detection or a debugging tool are implemented via smart SPARQL queries on the RDF graph. Zhao *et al.* [34] propose a solution that uses the SWI-Prolog reasoning engine and custom APIs to exploit and possibly update a knowledge base. The authors also argue that among implementations of program analysis, those based on a declarative approach are easier to implement and maintain than those based on an imperative approach. Actually, our *Cypher* queries were a bit messy, for instance due to the systematic repetition of the set of inner relationships. The use of ontologies could simplify our implementation and be more powerful than mere *Neo4j* queries, as it will permit the introduction of more general concepts.

5 Conclusions

In order to focus on true detectable maintainability defects, we built *Javanalyser* to represent code as a graph, depending only on the structure of the code and allowing to easily extract data such as metrics. Thirty-four metrics were implemented as *Cypher* queries [23], embodying a formal definition of these metrics. Moreover, declarative queries allow taking into account the complexity of up-to-now ignored *Java* structures such as `Optional` or `Stream`. *Javanalyser* foreshadows a unified maintainability-analysis framework to conduct machine-learning studies to assess maintainability.

Future works include testing our graph and metrics against the software maintainability dataset [32]. Leveraging ontology-based program analysis [34] could also simplify our implementation and be more powerful than mere *Cypher* queries, as it will permit the introduction of more general concepts, thus allowing higher level analysis. Finally, studying the graph simplifications we designed would be required to ascertain the graph is faithful to the developer's cognitive representation of the program.

Acknowledgments

We thank our collaborators at *onepoint*[5] for their insightful advices, in particular Damien Bonvillain, Alexandra Delmas, Jérôme Fillioux, and Jérôme Lelong.

References

[1] Eclipse Java Development Tools. `https://www.eclipse.org/jdt/`.

[2] JavaParser. `https://javaparser.org/`.

[3] jQAssistant. `https://jqassistant.org/`.

[4] Neo4j. `https://neo4j.com/`.

[5] Prettier. `https://prettier.io/`.

[6] Pylint. `https://pylint.org/`.

[7] SonarQube. `https://www.sonarqube.org/`.

[8] SourceMeter. `https://www.sourcemeter.com/`.

[9] Ibrahim Abdelaziz, Julian Dolby, James P. McCusker, and Kavitha Srinivas. Graph4Code: A Machine Interpretable Knowledge Graph for Code. *arXiv:2002.09440 [cs]*, May 2020. `https://arxiv.org/abs/2002.09440`.

[10] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216, May 2013.

[5] `https://www.groupeonepoint.com/`

[11] Hadeel Alsolai and Marc Roper. A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, 119:106214, March 2020.

[12] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming*, 2020:1–26, August 2020.

[13] Sébastien Bertrand, Pierre-Alexandre Favier, and Jean-Marc André. Pragmatic Software Maintainability Management Using a Multi-agent System Working in Collaboration with the Development Team. In Sara Rodríguez González, Alfonso González-Briones, Arkadiusz Gola, George Katranas, Michela Ricca, Roussanka Loukanova, and Javier Prieto, editors, *Distributed Computing and Artificial Intelligence, Special Sessions, 17th International Conference*, Advances in Intelligent Systems and Computing, pages 201–204, Cham, 2020. Springer International Publishing.

[14] G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, pages 57–58, New York, NY, USA, May 2018. Association for Computing Machinery. `https://doi.org/10.1145/3194164.3194186`.

[15] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Barry Boehm, and Lin Shi. Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It? In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 377–378, May 2017.

[16] Shyam R. Chidamber and Chris F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *OOPSLA '91: Conference Proceedings on Object-oriented Programming, Systems, Languages, and Applications*, volume 26, pages 197–211, Phoenix, Arizona, USA, November 1991. Association for Computing Machinery.

[17] Don Coleman, Bruce Lowther, and Paul Oman. The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1):3–16, April 1995.

[18] Ward Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, December 1992.

[19] Sara Elmidaoui, Laila Cheikhi, and Ali Idri. Towards a Taxonomy of Software Maintainability Predictors. In Álvaro Rocha, Hojjat Adeli, Luís Paulo Reis, and

Sandra Costanzo, editors, *New Knowledge in Information Systems and Technologies*, volume 930 of *Advances in Intelligent Systems and Computing*, pages 823–832. Springer, 2019.

[20] Sara Elmidaoui, Laila Cheikhi, Ali Idri, and Alain Abran. Empirical Studies on Software Product Maintainability Prediction: A Systematic Mapping and Review. *e-Informatica Software Engineering Journal*, 13(1):141–202, 2019.

[21] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, Boston, 2004.

[22] Norman E Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson, second edition, 1996.

[23] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, May 2018. Association for Computing Machinery.

[24] Martin Hitz and Behzad Montazeri. Measuring Coupling and Cohesion In Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, page 10, Mexico, Monterrey, 1995.

[25] ISO/IEC. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25010:2011, ISO/IEC, 2011. `https://www.iso.org/standard/35733.html`.

[26] Pirmin Lemberger and Médéric Morel. Two Measures of Code Complexity. pages 195–206. January 2013.

[27] Wei Li and Sallie Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111–122, November 1993. `http://www.sciencedirect.com/science/article/pii/016412129390077B`.

[28] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis - ISSTA '08*, page 131, Seattle, WA, USA, 2008. ACM Press.

[29] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[30] Richard Müller, Dirk Mahler, Michael Hunger, Jens Nerche, and Markus Harrer. Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 107–111, Madrid, September 2018. IEEE.

[31] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, September 2016.

[32] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 278–289, Adelaide, Australia, September 2020. IEEE.

[33] Markus Schnappinger, Arnaud Fietzke, and Alexander Pretschner. Human-level Ordinal Maintainability Prediction Based on Static Code Metrics. In *EASE 2021: Evaluation and Assessment in Software Engineering*, pages 160–169, Trondheim, Norway, June 2021. ACM.

[34] Yue Zhao, Guoyang Chen, Chunhua Liao, and Xipeng Shen. Towards Ontology-Based Program Analysis. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming*, volume 56 of *Leibniz International Proceedings in Informatics*, pages 26:1–26:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.