JavaCheck: A Domain Specific Language for the static analysis of Java code

Sara Pérez-Soler, Juan de Lara
Modelling & Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)
e-mail: {sara.perezs, juan.delara}@uam.es

***Abstract*—The increasing complexity of software systems has raised the need for code analysis tools to assess its quality. However, these tools offer predefined metrics or evaluation criteria, which are frequently hard to extend or modify.**

For this purpose, we have developed JavaCheck, a Domain-Specific Language targeted to define expected properties of Java code bases. JavaCheck can be used in a variety of scenarios related to quality assurance: to define expected code styles (e.g., naming conventions), specify programming conventions (e.g., private attributes), detect code smells possibly indicating errors (e.g., equals method with no hashCode), and detect patterns (e.g., uses of Singleton) or requirements demanded in a project (e.g., a class with name synonym to "Professor").

***Index Terms*—Domain-Specific Languages, Source code analysis, Quality**

I. Introduction

Software projects are increasing their complexity and size to address the requirements of today's systems. Software is typically developed by (sometimes large) teams of programmers with dissimilar skills. Hence, it is common practice to use tools to check code quality or help in enforcing company or project code standards [2], [12]. However, sometimes these tools are rigid, or difficult to adapt and extend.

To improve this situation, we have created a Domain Specific Language (DSL) called JavaCheck. The language permits expressing predicates to be evaluated over the source code bases of Java projects. The DSL is flexible and allows the expression of style and programming conventions, can be used to search for occurrences of programming idioms and patterns, and to express code smells [5] possibly indicating some potential problem. JavaCheck is connected with services to detect synonyms in several languages, which permits its use to specify expected domain requirements (e.g., to partially automate the correction of programming exercises). The DSL has been created using Model-based technology (EMF and Xtext) and is integrated within Eclipse. Hence, it shows the potential of Model-driven engineering in the programming domain.

Paper organization. Sec. II overviews our approach, explaining its different parts. Sec. III describes tool support and some initial experiments. Sec. IV compares with related work and Sec. V finishes with the conclusions and future work.

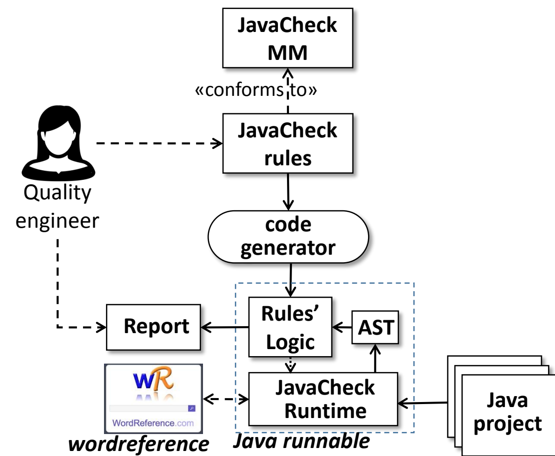

Fig. 1: Overview of our approach

II. Approach

Fig 1 shows the general architecture of our approach. We have created a DSL called JavaCheck, which can be used to define predicates that Java projects should fulfil.

Predicates can be used to express general quality properties (e.g., ensure all attributes of a class are private), accepted Java style guidelines (e.g., class names in upper camel case, constant names in uppercase), project-specific guidelines (e.g., maximum number of classes in a package), application-specific checkings (there should be a class with a name synonym to "Machine"), or smells of possible errors (a class redefining method equals, but not hashcode). JavaCheck has a textual syntax and has been defined through a meta-model.

The predicates expressed with JavaCheck are compiled into Java. This Java code uses a library we have built, which offers services to parse Java code into an Abstract Syntax Tree (AST), or to issue queries on *Wordreference*[1], to obtain lists of synonyms in both English and Spanish.

Fig 2 shows a small part of our meta-model. RuleSet is the root class, and contains a list of project names to be checked and a set of sentences to check on them. There are two types of sentences: the rules that will be evaluated, and intermediate

[1]http://www.wordreference.com/

variables to store collections of elements that have some properties. All sentences have a type element, that can be File, Package, Interface, Class, Enum, Method or Attribute and a clause that needs to be satisfied. The satisfy clause contains all properties that the element must comply with. The rules have a quantifier (all, exist or one) and a filter, with same structure as the satisfy clause. The rules can also reference variables.

Listing 1 shows some simple JAVACHECK sentences. The first sentence collects all methods named *equals* with a parameter of type *Object* and return type *boolean*, in a collection variable named Equals. The second collection named HashCode contains all methods named *hashCode*, without parameters and integer return type. Lines 5-6 show a rule that checks that all classes with one method in the Equals collection also have one method in HashCode collection. Overall, occurrences of this rule may signal potential problems in the Java code.

```
1 Equals: Method satisfy name="equals" and return type=Primitive.boolean
      and
2     parameter size=1 types=["Object"];
3 HashCode: Method satisfy name="hashCode" and return type=Primitive.int
      and parameter size=0;
4
5 all Class which have { one Method in Equals }
6 satisfy have { one Method in HashCode };
```

Listing 1: JAVACHECK program to detect classes with equals but no hashCode

JAVACHECK is evaluated over the AST of Java code. The AST is a tree representation of the syntactic structure of source code. We have programmed a library to create and explore the AST, and evaluate all the sentences. The library defines all the functionality of the static analyser, and the code generator needs only to synthesize code for the specific sentences by calling the library. When all sentences are generated, they can be evaluated scanning all nodes of the AST and checking the properties.

```
1 all Class satisfy name type= upper camel case;
```

Listing 2: JAVACHECK naming convention rule

Listing 2 shows another example to check a naming convention for classes. In particular, it checks that all class names are written in upper camel case. In this example the analyser first obtains all class declaration nodes in the ASTs of the project, and then checks that all names of these nodes are in upper camel case.

III. Tool support and experimentation

We have created an Eclipse plugin for JAVACHECK using Xtext and Xtend. The DSL allows us to express characteristics of Java programs, and then reports the result of the analysis.

Listing 3 shows a example of JAVACHECK file with 4 rules. First, the project or projects to be analysed should be indicated (line 1). All projects to be analysed are required to be in the Eclipse workspace. A "*" in the name makes JAVACHECK take all projects in the workspace.

Following the project name, a JAVACHECK program contains the sentences to be checked. In the listing we show some examples. The first rule (lines 3-5) checks that all attributes that are not static and final (i.e., all attributes that are not constant), must be private or protected. The second rule (lines 7) states that every method must have a JavaDoc comment with @parameter and @return tags. The third rule (lines 9-11) checks that the project has one class named *User* or a synonym (in English), and this class must have an attribute named *address*. For this rule WordReference is used to obtain the synonyms. Finally, the last rule checks that all abstract classes have some children class.

```
1  Projects Name: *;
2
3  all Attribute
4  which is not modified with [static and final]
5  satisfy is modified with [private or protected];
6
7  all Method satisfy JavaDoc @parameter @return;
8
9  one Class satisfy name like "User", English and have {
10   one Attribute satisfy name="address"
11 };
12
13 all Class which is modified with [abstract] satisfy is superclass;
```

Listing 3: Example JAVACHECK program

```
1  all class which modifiers: [ (abstract) ] satisfy is superclass [1..*]
2  Checked.....ERROR
3  PASS:
4   These elements do not satisfy is superclass [1..*]:
5   – In file D:\Workspace\Evaluate\src\abstractClass\Plane.java the class
        Plane (line: 3)
6
7  FAIL:
8   These elements satisfy is superclass [1..*]:
9   – In file D:\Workspace\Evaluate\src\abstractClass\Element.java the class
        Element (line: 3)
10    is super of:
11    In file D:\Workspace\Evaluate\src\restOfClass\Point.java the class Point (
        line: 5)
```

Listing 4: Example of JAVACHECK report

Listing 4 shows the JAVACHECK report produced by the last rule of Listing 3. Currently, the report is a text file showing if the rule is met or not (in this case it is not), and then listing all the elements that pass and that do not pass the rule.

A. Experimentation

We present two preliminary experiments with JAVACHECK. The first one is directed to assess expressivity, and usefulness to detect problems in the code. The second one is directed to check its scalability.

In the first experiment we use JAVACHECK as a way to semi-automatically assess student projects related to the creation of an information system for an antiquarian. We used three types of rules for validation: style, programming and domain-specific. Some style rules included: all files have only a class, an interface or an enumeration; every file has less than 2000 lines of code (LOC); methods' bodies should be less than 30 LOC; every attribute that is not constant must be written in lower camel case, while constant attributes must be in upper case; the enumerations, classes and interfaces names must be in upper camel case; every class, method, interface and enumeration must have a JavaDoc comment; every package must have a Java file (i.e., must not be empty), among others (a total of 15 rules).

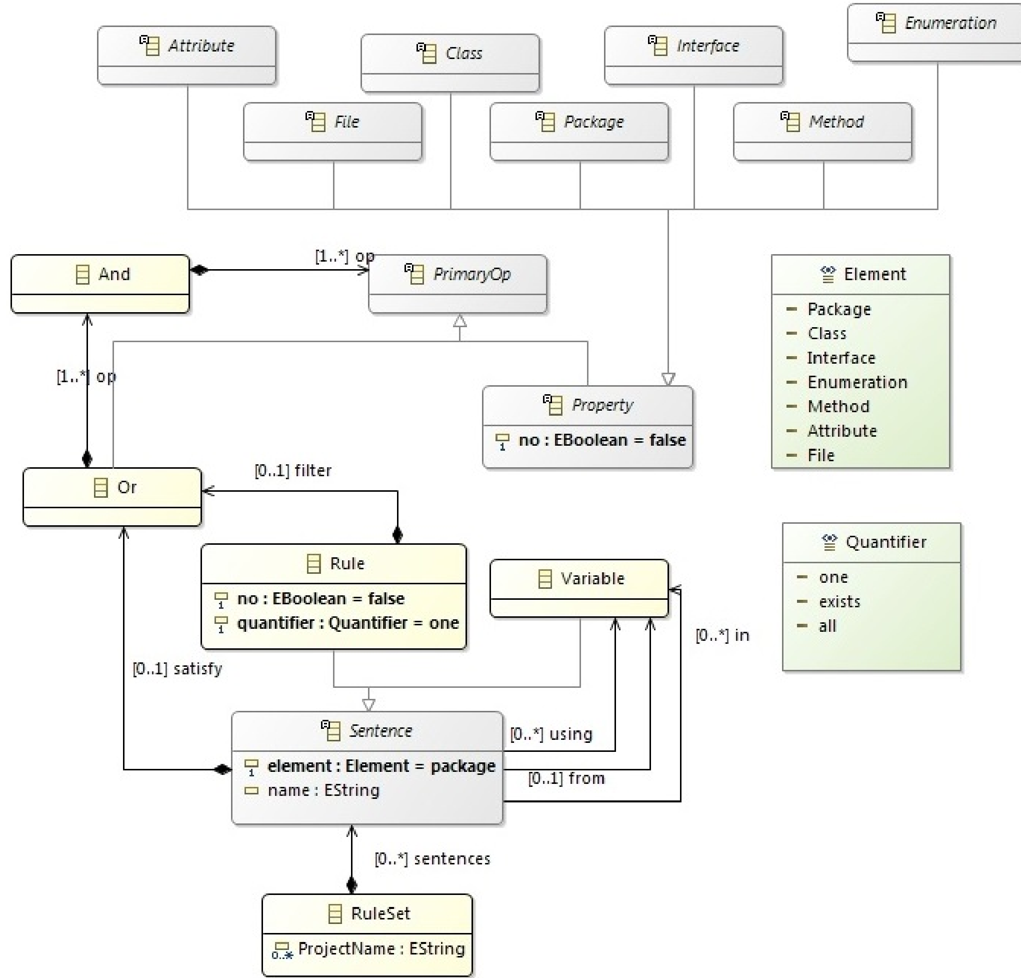

Fig. 2: Meta-model of JAVACHECK (excerpt)

Programming rules included: every abstract class must have some children; all interfaces must be extended by other interface or implemented by some class; every class that implements Comparable must override equals and hashCode methods and there is no method that returns a value of type Object. Regarding domain-specific rules, after reading the requirements, we included these rules: there is one class called *Item* or a synonym, which is abstract and public, is extended 3 or 4 times and has an identifier that is integer or long attribute, a name or description, a date and a price. The project must also have three classes, with names synonym to *Small*, *Bulky* and *WorkOfArts*, all extending *Item*.

We were able to successfully encode these rules in JAVACHECK, hence showing good expressivity. Regarding usefulness, we found several problems in the analysed code. The most prominent ones included, lack of JavaDoc comments, many methods over 30 LOC, abstract classes with no children, seven methods returning Object and a class *Item* with no date. As we were able to found these defects, we concluded that JAVACHECK was useful for our purpose.

To measure performance we run the previous 15 style rules on a larger project, the *org.eclipse.jdt.core* of the library *org.eclipse.jdt.core* of Eclipse. This project has 1,443 files with 1,442 classes, 238 interfaces, 17 enumerations, 24.290 methods and 12.709 attributes. The check took 6 minutes approximately, which we see as reasonable for large projects.

IV. RELATED WORK

Model-Driven Engineering (MDE) has been used to solve different problems in the programming domain, like reverse engineering [1], repository mining [10] or comparing open source software using quality models [9]. Many times, the code needs to be represented as a model, conforming to a meta-model, so that it can be queried and processed using model management tools. However, this has the drawback of requiring too long pre-processing times [10]. To solve this issue, in [6] the Epsilon model connectivity layer was extended so that the Epsilon model management languages can be run

on Java programs. Our approach goes in this direction, as JavaCheck executes on Java ASTs. However, our approach is more direct, as the generated code can directly access the AST with no need for an intermediate layer. Moreover JavaCheck is a DSL specifically designed for querying Java ASTs.

Regarding code analysis tools, there are two main types: static and dynamic. The first ones analyse the code without running the program, and can be used in the earliest phases of programming. The second ones, typically testing tools, can be used at the end of the coding.

There are many tools able to assess code quality in a static way [4]. PMD [8] works over Java and other languages, and can detect potential problems like empty try/catch/finally/switch statements, dead code, overcomplicate expressions or duplicate code. New rules can be added by coding them in Java or XPath [13]. FindBugs [3] focuses on finding coding errors and supports only Java. It cannot be increased with new rules and works over Java bytecode. CheckStyle [2] focuses on analysing style conventions. The tool is highly configurable with a XML file and allows creating new rules, coding them in Java and using ASTs. SonarQube [12] uses various static analysis tools like PMD, CheckStyle or FindBugs to obtain metrics to help improving the quality of the source code. It can show information about the architecture, design, duplicate code, programming rules, possible errors and their possible solutions. Finally, Semmle QL [11], [7] is a query language over code. This language is based on DataLog, so it needs to process the code first, to obtain a relational representation.

In conclusion, with respect to MDE approaches to solve problems in the programming domain, JavaCheck can be executed on ASTs more directly, due to its compilation approach. With respect to existing source code analysis tools, JavaCheck is a DSL that permits a high customization of queries, with no need for low-level coding based on XPath or ASTs.

V. Conclusions and Future Work

We have presented JavaCheck, a DSL for expressing rules to be checked on Java projects. We have built an Eclipse plugin which permits the integration of JavaCheck with the Java IDE, and performed some initial experiments showing promising results.

In the future, we want to improve tooling, e.g., enhancing the reporting facility. We would also like to extend the expressiveness of the language to consider the analysis of method bodies, which currently cannot be analysed. Finally, we are considering adding the possibility to define quick-fixes, to be fired when some rule fails.

Acknowledgements.

Work funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

References

[1] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. Modisco: A model driven reverse engineering framework. *Information & Software Technology*, 56(8):1012–1032, 2014.
[2] CheckStyle. http://checkstyle.sourceforge.net/.
[3] FindBugs. http://findbugs.sourceforge.net//.
[4] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5: 1–38, 2012.
[5] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
[6] A. García-Domínguez and D. S. Kolovos. Models from code, or code as models? In *Proc. OCL@MODELS*, volume 1756 of *CEUR Workshop Proceedings*, pages 137–148, 2016.
[7] E. Hajiyev, M. Verbaere, O. de Moor, and K. D. Volder. Codequest: querying source code with datalog. In *Proc. OOPSLA 2005*, pages 102–103. ACM, 2005.
[8] PMD. https://pmd.github.io/.
[9] D. D. Ruscio, D. S. Kolovos, Y. Korkontzelos, N. Matragkas, and J. J. Vinju. Supporting custom quality models to analyse and compare open-source software. In *Proc. QUATIC 2016*, pages 94–99. IEEE Computer Society, 2016.
[10] M. Scheidgen, M. Schmidt, and J. Fischer. Creating and analyzing source code repository models - A model-based approach to mining software repositories. In *Proc. MODELSWARD*, pages 329–336. SciTePress, 2017.
[11] Semmle. https://semmle.com/products/semmle-ql/.
[12] SonarQube. https://www.sonarqube.org/.
[13] XPath. https://www.w3schools.com/xml/xpath_intro.asp.