Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 65–66
DOI: 10.15439/2023F2071
ISSN 2300-5963 ACSIS, Vol. 35

Towards reliable rule mining about code smells: The McPython approach

Maciej Ziobrowski, Mirosław Ochodek, Jerzy Nawrocki, Bartosz Walter
Poznan University of Technology, Poznań, Poland
maciej.ziobrowski@student.put.poznan.pl; {miroslaw.ochodek, jerzy.nawrocki, bartosz.walter}@put.poznan.pl

***Index Terms*—Rule mining, code smell, McPython, Python, domain specific languages**

Problem

Code smell is a risky pattern in code that can lead, in the future, to problems with code maintenance. One of the approaches to identifying smells in the code is metric-based smell detection. A classic example is the *God Class* smell which can be detected by using three metrics (see, e.g., [1], [2], [3]):

- Weighted Method Count (WMC – sum of McCabe's complexity of all methods in the analysed class),
- Tight Class Cohesion (TCC – relative number of directly connected methods within the analysed class), and
- Access to Foreign Data (ATFD – number of classes containing attributes referenced by the analysed class directly or via get/set methods).

To make a decision (smelly / not smelly), computed metrics are compared against predefined thresholds. So, the quality of smell detection depends not only on a set of chosen metrics, but also on their thresholds.

Unfortunately, the quality of the existing smell detectors is still not satisfactory (cf. [4]) and there is a need for more research in the area. One of the issues worth investigation is the impact of a set of code smells on severity of the detriment caused by them. To conduct this research in a clear and reproducible way one needs an appropriate workbench (a critical review of the literature in the area is presented in [5]).

The proposed workbench

In this paper, it is postulated that empirical research on smell detectors should be based on (1) precise definitions of the analysed smells, and (2) smell detection rules (including metric thresholds) should be mined from software repositories using machine learning (ML).

The overall architecture of the proposed workbench is illustrated in Fig. 1. Given a code repository, a code smell detector identifies all smelly classes while the issue detector identifies troublesome classes (e.g., defective classes - here one can use an idea proposed by Śliwerski *et al.* in [6]). The reports generated by both detectors are consolidated to produce a decision table (the decision table of Fig. 1 refers to the *God Class* smell with three thresholds, _WMC, _TCC, and _ATFD, corresponding to the three metrics mentioned earlier). Given a decision table, one can use e.g. C4.5 algorithm to get a decision tree (see [7] or [8]). Another option is to apply rough-set approach (see e.g., [9]).

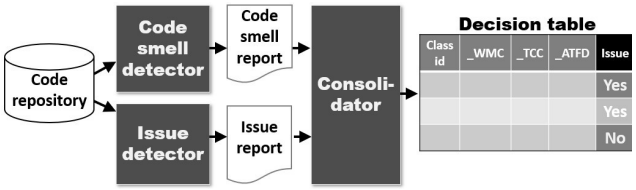

Fig. 1. Architecture of the proposed workbench.

The McPython language

For defining metric-based code smells we propose a domain-specific language, McPython (Meta Code in Python). Its notation is based on Python. Description of a smell detector consists of three parts: code model, smell definitions, and query.

Code model defines all the code attributes needed for detection of a given smell (it corresponds to view model in the 3-layer model of code proposed in [10]). Those attributes are provided by another program, code modeller, and code model just defines what is needed from the code modeller. As McPython is focused on object-oriented languages, there are four categories of entities represented in each model, namely: classes, their attributes, methods, and their parameters. An example of code model is presented in the first part of Listing 1. Each code entity has a number of attributes along with their JSON types (*nat* is an extra type denoting natural numbers and it is a subset of *int*). Each description of entity category starts with the `ent` keyword and ends with a double colon (::).

A smell definition is a Python-like function returning a Boolean value. It is accompanied by a set of auxiliary functions (some of them can be imported). The second part of Listing 1 contains a function named `GodClass` defining the *God Class* smell and an auxiliary function `WMC`. The `GodClass` function uses three special parameters called *thresholds*: `_WMC`, `_TCC`, and `_ATFD`. A threshold represents an upper/lower bound on some metric. It is declared in a separate line, its name begins with an underscore ('_') and is preceded with the `thr` keyword.

Smell definitions can refer to attributes specified in the code model and they can contain mathematical symbols such as summation ($\sum$) or quantifiers($\forall$, $\exists$). On the other hand there are some restrictions imposed on McPython code:

Listing 1. God Class detector in McPython.

```
--- Model:
ent class:
    name:    string,    # class's name
    methods: list,      # method ids
ent method:
    name: string,  # method's name
    McCABE: nat :: # cyclomatic complx.
--- Smells:
import TCC, ATFD
def WMC(c: class):
    return ∑ m ∈ c.methods: m.McCABE::
thr _WMC
thr _TCC
thr _ATFD
def GodClass(c: class):
    return WMC(c) ⩾ _WMC ∧
           TCC(c) < _TCC ∧ ATFD > _ATFD
--- Query:
_WMC  ∈ [45, 47]
_TCC  ∈ [0.4, 0.3, 0.2]
_ATFD ∈ [4, 6]
select c.name for c ∈ class \
              where GodClass(c)
```

- each variable is assigned a value only once;
- there are no compound statements like `while` or `if`.

Parameters of McPython functions can have types assigned to them. Those types are categories of code entities, e.g., `class` or `method`.

The third part of code in McPython is a `query`. It starts with specifying the values of the thresholds one is interested in. Then comes the `select` clause which resembles the one known from SQL. The result of the query is a report showing the requested attributes of all the code entities matching the query for all the possible combinations of the values of thresholds.

Implementation remarks

McPython definition of a smell detector is encoded in Unicode what makes all the mathematical symbols easily available. When McPython code is ready one has:

- to translate it to Python 3, and
- to generate a model of the analysed code (smell detector expects on the input a code model, not the code itself).

The process is illustrated in Fig. 2. An advantage of running smell detector on a code model instead of the code itself is possibility of using the same definition of a code smell on repositories written in different programming language, provided that one has a code modeller for a given language.

Current version of McPython translator is written in Python 3 (Python accepts Unicode as an input). Model of the analysed code is implemented as a list of all its entities (position of an entity on the list serves as its identifier) and it is read with the library function `json.loads`. McPython constructs concerning operations over sets, e.g., a universal quantifier ($\forall$) or summation ($\sum$), are translated as calls to an appropriate function (definitions of those functions are added to the generated code). Those functions have two parameters: a set of code entities (represented by their identifiers) and a condition or expression that is evaluated for each element of a given set (here lambda expressions of Python proved very useful).

Code modeller for the Python language (cf. Fig. 2) is built with the help of Python's `ast` module and the `NodeVisitor` class contained in it. First all class nodes of a given abstract syntax tree are visited and then their method are analysed. The collected data are stored as an array of dictionaries and transformed into JSON with the `dumps` function of the `json` module.

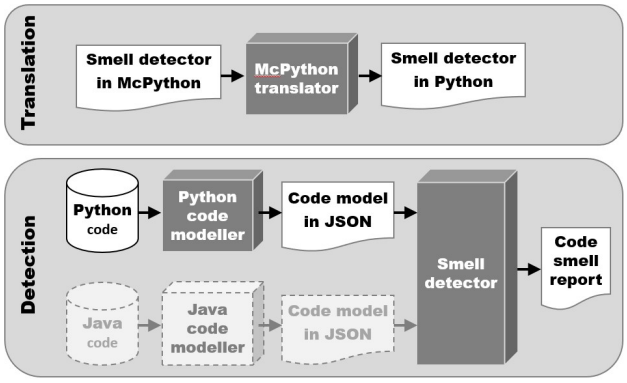

Fig. 2. Translation and detection phase.

References

[1] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[2] Ł. Puławski, "Improvement of design anti-pattern detection with spatio-temporal rules in the software development process," in *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2022, pp. 521–528.

[3] Łukasz Puławski, "Methods of detecting spatio-temporal patterns in software development processes," Ph.D. dissertation, University of Warsaw, 2022. [Online]. Available: https://ornak.icm.edu.pl/bitstream/handle/item/4533/0000-DR-1827-praca.pdf?sequence=1

[4] T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and solutions for refactoring adoption: An industrial perspective," *IEEE Software*, vol. 32, no. 6, pp. 44–51, 2015.

[5] T. Lewowski and L. Madeyski, "How far are we from reproducible research on code smell detection? a systematic literature review," *Information and Software Technology*, vol. 144, p. 106783, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058492100224X

[6] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1–5, may 2005. [Online]. Available: https://doi.org/10.1145/1082983.1083147

[7] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[8] "Weka 3: Machine learning software in java," https://www.cs.waikato.ac.nz/ml/weka/.

[9] Z. Pawlak, J. Grzymala-Busse, R. Slowinski, and W. Ziarko, "Rough sets," *Commun. ACM*, vol. 38, no. 11, p. 88–95, nov 1995. [Online]. Available: https://doi.org/10.1145/219717.219791

[10] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, "An extensible meta-model for program analysis," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 592–607, 2007.