University of Amsterdam

Software Evolution Series 2

Report

December 14, 2017

Students:
Cornelius Ries
11884827

Piotr Kosytorz
11876964

Tutor:
Riemer van Rozen

Course:
Software Evolution

1 Introduction

This documents contains our notes and answers to the questions about software metrics (practical lab Series 2).

Rewrite the introduction

2 Definitions

In order to provide a tool to satisfy the assignment's objectives, a good understanding of the key concepts used in clone detection is needed. Based on a biref literature study, we have selected the following definitions as the base for our work and evaluation of the cloning problem:

2.1 Clones and clone classes

- **Clone relation** is an equivalence relation on code portions [2].
- **Clone pair** is a set of two code portions that are in clone relation with each other (compare [2]).
- **Clone class** is an equivalence class of clone relation. In other words, it is a set of multiple (more than one) code portions which hold mutual clone relation to each other (compare [2]).
- **The biggest clone** is is the biggest (has the largest number of lines of code) code portion, for which exists at least one code portion, which holds clone relation with it.
- **The biggest clone class** is the biggest equivalence class of clone relation (has the most elements) in the domain of examined project.

2.2 Clone types

- **Type-1 clones** are per definition from [3] strict textual clones, excluding comments and white spaces.

- **Type-2 clones** are definition from [3] *a syntactically identical copy; only variable, type, or function identifiers have been changed.*
- **Type-3 clones** are per definition from [3] copies with further changes like adding/removing statements.

3 Algorithm design

- Type-3 clean method (pseudocode) / We dont have it yet
- Formalised algorithm (pseudocode with references to type-1, type-2 clean methods)
- Describe how the algorithm is covered in the literature (and provide references).

3.1 General algorithm design

The idea and algorithm of our duplication detection is based on the information from [4] and [1]. The main idea behind this approach is to use the abstract syntax tree for clone detection. After creating the ast from the source files, the nodes are hashed into different buckets. If a bucket has more than one element, it follows, that there are multiple locations in the source code of this node. This counts as one clone class. There are multiple types of clones that our tool can detect. These are type 1 and type 2 clones.

3.2 Type-1 cleaning

If using the AST approach, the bucket key is in general just the node itself. With rascal though, there was a change in the implementation that shifted the `loc` and other informations of a node from the annotations on the node, to information contained in the node. This messed up the matching because every location was unique, which resulted in every node (because of subnodes) being unique. To circumvante the problem we remove the unnessecary information from all the relevant nodes. The actual cleaning `cleanNode` is done by visiting all the subnodes of a relevant node and replacing the node with a cleaned version of the original node using the `insert` statement.

3.3 Type-2 cleaning

For type 2 detection we also use the AST. To streamline all the information that is contained in the node we visit all the subnodes of a node and replace specific variables/identifiers/types with a general one. The replacement is done using the rascal `visit - case =>` notation.

```
case Type _ => lang::java::m3::AST::float()
case Modifier _ => lang::java::m3::AST::\public()
case \method(_, _, a, b) => \method(lang::java::m3::AST::float(), "method", a,
    b)
case \method(_, _, a, b, c) => \method(lang::java::m3::AST::float(), "method",
    a, b, c)
case \parameter(a, _, b) => \parameter(a, "parameter", b)
case \vararg(a, _) => \vararg(a, "vararg")
case \annotationTypeMember(a, _) => \annotationTypeMember(a, "
  annotationTypeMember")
case \annotationTypeMember(a, _, b) => \annotationTypeMember(a, "
  annotationTypeMember", b)
case \typeParameter(_, a) => \typeParameter("typeParameter", a)
case \constructor(_, a, b, c) => \constructor("constructor", a, b, c)
case \interface(_, a, b, c) => \interface("interface", a, b, c)
case \class(_, a, b, c) => \class("class", a, b, c)
case \enumConstant(_, a) => \enumConstant("enumConstant", a)
case \enumConstant(_, a, b) => \enumConstant("enumConstant", a, b)
case \methodCall(a, _, b) => \methodCall(a, "methodCall", b)
case \methodCall(a, b, _, c) => \methodCall(a, b, "methodCall", c)
case \simpleName(_) => \simpleName("simpleName")
case \number(_) => \number("125681651")
case \variable(_, a) => \variable("variable", a)
case \variable(_, a, b) => \variable("variable", a, b)
case \booleanLiteral(_) => \booleanLiteral(true)
case \stringLiteral(_) => \stringLiteral("stringLiteral")
case \characterLiteral(_) => \characterLiteral("a")
```

3.4 The main algorithm

Here we provide a textual representation of our main algorithm. For further information please have a look into the source file `DuplicationsAnalyzer.rsc`.

```
Build the AST of the project.

For all nodes in AST
- If size of node > threshold
        - Clean node for type 1 detection.
        - Clean node for type 2 detection.
        - Collect node in map with cleaned node as key,
          relation of original node and location as value

For all keys in Map build a set of clone classes
- Collect all values from the map for current key
- If size of collected values > 1
        Then
                add values as a new clone class to set

For all previously detected clone classes filter subclone classes
- If another clone class exists for which all locations
  include the locations of the current one
        Then
                -
        Else
                Add to new Set

Collect all filtered clone classes in output format
        - Collect locations
        - Determine line statistics
        - Determine clone type
```

4 Main program validity

We provide an automatic testing suite to verify that the tool works as designed. To maintain a good separation of concerns between implementation and tests, all tests are in separate files that extend their original rascal module. This will result in better maintainability, readability and extensibility for the future.

The tests are split in random generated tests where applicable. For example while operating on files and their locations rascal is not able to provide a proper generator for reading the contents of a file. For this we provide three test files that are used in tests where files are needed.

Especially the Clone Detection is covered with random test generators, testing the properties of the tested functions as well as tests where the implementation is tested against the provided test files to assure the continued working in case of refactoring. For design changes the tests may have to be adjusted.

To run the tests, import all the modules below and execute :test in the rascal console. The `projectLocation` in `Configuration.rsc` has to be set to the projects location in your eclipse!

- DuplicationsAnalyzerTests
- RaterTests
- UtilsTests
- VolumeAnalyzerTests

5 The tool

- General description and purpose, used solutions (webserver, REST api, ReactJS app, d3 graphs, etc)
- The 3 main requirements (your tool satis

  es from the perspective of a maintainer) [ref to storey1999cognitive]:
  1. Get a comprehensible overview (of code quality and duplications)
     - Readable table
     - SIG maintanability index
  2. Get a deep insight into the clones
     - navigate easily through them
     - provide and extended search function (easy to use = we reduce eort)
  3. See how the clones spread over my project (=Improve comprehension)
     - provide insightful visualization = Provide eective presentation styles (graph)
     - show how the

       les containing clones relate to each other (=indicate options for further exploration)
- Implementation choices
- The visualisation (constellation) - how does it help a maintainer or a developer?

6 Tool manual

- How to use
- Where is the Rascal project
- Where is the original visualisation project

To run and use the tool we provide the source code as a eclipse project.

6.1 How to run

In order to run the program, please follow the steps:

1. Please import the project into your eclipse with a working rascal installation.
2. Open `Configuration.rsc` and adjust the location of the `projectLocation` to match the path of the project to your eclise
3. Do the same for the `smallSqlProject` and `hqSqlProject`
4. Start a rascal console and import the `Main` module
5. run `startServe();`
6. open a browser and point it towards `http://localhost:5433` or to the location of `serveAddress` in case you changed it

7 Summary

- Sum things up
- Write how the tool performs on hsqlsb,
- Write how it can be improved

References

[1] Ira D Baxter et al. "Clone detection using abstract syntax trees". In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE. 1998, pp. 368–377.

[2] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[3] Rainer Koschke. *Identifying and Removing Software Clones*. 2008.

[4] Flavius-Mihai Lazar and Ovidiu Banias. "Clone detection algorithm based on the Abstract Syntax Tree approach". In: *Applied Computational Intelligence and Informatics (SACI), 2014 IEEE 9th International Symposium on*. IEEE. 2014, pp. 73–78.