Report Assignment 2
Software Evolution

Quinten Heijn - 10374442
Dylan Bartels - 10607072

22 November 2017

Introduction

As stated in "Software measurement: A necessary scientific basis"[4] measurement activities must have clear objectives. This is why we will work within a scenario that will provide a frame for all the decision making in this assignment.

The case is as follows: a small team wants to use old code for a database written in Java. However, some of the functionality is outdated and has to revised. Since the team is not familiar the code base they want to do an automated assessment of duplication in the code. Finding the pieces of duplicate code will ensure that they only have to do any changes once. This will not only save them time, but will also prevent inconsistencies.

Definitions

We give a short list of the definition we will be using in this report, so that we will not have explain them later on.

- **Duplication classes**:
- **Code duplication**:
  - **Type 1**: Exact copy, ignoring whitespace and comments.
  - **Type 2**: syntactical copy, changes allowed in variable,type,function identifiers.
  - **Type 3**: Copy with changed, added, and deleted statements

  - **Type 4**: Functionality is the same, code may be completely different

- **AST**: Abstract Syntax Tree

Goal-Question-Metric

The Goal-Question-Metric (GQM) approach is a fairly trivial method for making sure your using the right metrics[7]. We will use this approach to provide a structure for our scenario.

Goal

- **Object of examination**: The total code base of the old database.
- **Purpose**: Making revision to the code faster and saver.
- **Focus**: Code duplication.
- **Viewpoint**: A small team that is unfamiliar with the code.
- **Environment**: Application development in Java.

Question

- **Question 1**: Which duplication classes are present in the code base?
- **Question 2**: Which duplication classes can be refactored?
- **Question 2**: Which units in the duplication classes have to revised?

Metric

- **Metric 1**: Duplication classes in code base.
- **Metric 2**: The subsets in the duplication classes that can be easily refactored to a new single unit.
- **Metric 3**: Units that need to revised.

Strategy

Based on the CQM we can begin to formulate a precise strategy to answer the posed questions. An important detail about this case is that not every method has to revised. Even though it might be good practice to remove all duplication, a small team will not always have the resources to fix everything. Automating the refactoring comes with some risk.[3] This means that the team probably wants to pin-point duplicates that are relevant to begin with. Since the code has to be revised any way, they do not care that much about exact close, but more about similar piece of code. By mapping all of the type 2 clone classes the team can update the code while refactoring the clones.

Clone Detector Algorithm

To find the clones we will use an algorithm that finds type 2 clones, but can easily be extend to also search for type 1 clones. The algorithm does this by analyzing an abstract syntax tree of the code by creating an suffix tree and is based on an artice by Falke[5]. Using Ukkonen's algorithm this can all be done in $O(n)$ time. It consists of the following steps:

1. Parse the program and generate an AST.
2. Serialize the AST.
3. Apply suffix tree detection to the AST.
4. Analyzing the suffix tree.

Parsing the AST is taken care of by the M3 library in Rascal. The other steps we will explain in more detail.

Serialize AST

Because Ukkonen's algorithm needs a string/list as input, we need to serialize the AST.The AST is serialized by visit all the nodes. For each node we add an id that represents the node type to a list. In a second visit trough the AST we only look at the groups of nodes types (declarations, expression, statement), so we can at the location of the node to the same list. The result is a list relation of ids and locations that can be used as input for the suffix tree detection. The benefit of this approach is that we can easily trace back the location of a clone. We can also expend clone detection application to have options for ignoring input statement, disregarding types and other AST nodes. The downside is that this list can become large, resulting in memory issues once we create the suffix tree. We'll come back to this problem later on. A solution might be to group AST nodes together and give an unique id to each unique group.

Apply Suffix Tree Detection

[2] The next step creates a suffix tree from the list of serialized nodes. We've done this using Ukkonen's algorithm[6]. This algorithm scans the input string from left to right in $O(n)$, making this a fast and intuitive solution. We won't explain the algorithm in depth, but we will give the basics using sudo code.

```
def createSuffixTree(code):
    suffixTree = [root]
    for (node [in] code) {
        if (node is on root node)
            followEdge()
        else
           suffixTree += createNewNode()
    }

    return M

def followEdge(suffixTree):
    if (node is on egde) {
        followEdge()
    } else {
        suffixTree += createNewNode()
    }

    return suffixTree
```

There are three rules needed for edge extension that basically ensure that that the algorithm knows where to add the next node in the suffix tree. Since Rascal does not have pointers, we represented the suffix tree in a list of tree nodes. Matching the id of every node in suffix tree with the location in list ensured us that we did not have go trough the list so search for a stored node.

However, as we said before, there is an issue with the amount of memory that is needed for scanning big projects. Once the suffix tree becomes bigger than can be stored in memory, every list look up requires a call to the hard disk. This slows the algorithm down a lot. More memory or, as said before, more efficient grouping of nodes, could solve this issue. Sadly we did not have the time to fully explore these options.

Analyzing

Next we need to analyze the suffix tree, to prepare it for visualization. The first step is to calculate clone length. This is done by following the edges and summing the length of the previous edges and storing it on the node. This way the length at every node resembles the length of a clone class and the length of the leaves resembles the length of the actual clones.

The next step is filtering the suffix tree. Since every combination of nodes as it occurs in the AST is stored in the suffix tree, we some clones that only occur once and very small clones that only consist of a few abstract syntax nodes and are not relevant. By recursively removing leave nodes with a length that is smaller than a set threshold, tree will only contain the clones and clone classes that we are interested in. However cleaning up the tree is a difficult task, because we want to maintain the connection between node id and the location of that node in the list that represents the suffix tree. This also has to be done in $O(n)$ to not lose the efficiency of the algorithm.

The last step is adding the location of the clone at every leaf node. This is simply done by it up in the list relation we made earlier, since the suffix tree is basically nothing more than a collection of pointers to this list. Since we have the location of the clone, we can easily compare clones to see if they are also type one clones. Since every type 1 clone is also a type 2 clone.

Visualization

The visualization is influenced by [1] which states the relation of sets and ways of visualizing these relations. Bilal Alsallakh et al. give a run-through on the most commonly used visualizations and in which cases these excel. Euler and Venn diagrams are among the oldest and most commonly used visualizations when working with sets. In these visualizations the sets are illustrated mostly as round colours and overlapping where the subset is also in other sets.

In the code duplication assignment we are working with sets representing the duplicates, these duplicates can occur multiple amount of time within a project. A Ven diagram like representation offers a great way to capture groupings of duplicates. In such a ven diagram the same duplicates would be grouped in the same circle giving immediate insight on the amount of times a duplicate occurs. The circle itself offers the possibility to be sized depending upon the size of the duplicate. This way an observant of the visualization can intuitively see the duplicates and the size of the duplicates.

Results/evaluation

In the end we ran into a problem with filtering the suffix tree. So we ended up with results that we either did not trust or that were difficult to make sense of. With small test cases our implementation worked fine, but with large projects the results became unrealistic, however, due to size it was hard to analyze what went wrong.

Implementing Ukkonen's algorithm was way more difficult than expected and it took several tries to get to the implementation we ended up using. In hindsight we were probably too ambitious and should have stuck with just analyzing a hashed AST. If we had another week or so, we could have probably made it work. But due to time constraints we could not try a different approach or fix the one we had.

- Percentage of duplicate lines: ?
- Number of clones: ?
- Number of clone classes: ?
- Biggest clone in lines: ?
- Biggest clone class: ?
- Example clones: ?

References

[1] Bilal Alsallakh, Luana Micallef, Wolfgang Aigner, Helwig Hauser, Silvia Miksch, and Peter Rodgers. Visualizing sets and set-typed data: State-of-the-art and future challenges. *Eurographics Conference on Visualization*, 2014.

[2] Brenda S Baker. Parameterized pattern matching: Algorithms and applications. *Journal of computer and system sciences*, 52(1):28–42, 1996.

[3] James R Cordy. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 196–205. IEEE, 2003.

[4] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on software engineering*, 20(3):199–206, 1994.

[5] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.

[6] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

[7] Joost Visser, Sylvan Rigal, Gijs Wijnholds, and Zeeger Lubsen. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.", 2016.