Rationale assignment series 2

Abel Kaandorp (12012149)
Sjoerd van der Heijden (10336001)

December 19, 2017

1 Introduction

In this document we will describe how we approached the different tasks of assignment series 2 and why. We do so by discussing the modules we built to complete the assignment, and the rationale for building them that way. The modules are geared towards clone-detection and clone visualization, and will be discussed in that order. Additionally, a dummy java-project was written, with the goal of being a test for the clone detection code, which will be discussed afterwards.

2 Clone Detection

Even though clone detection was part of the previous assignment series as well, we decided not to reuse the code. This is because previously we converted the code to strings, which we cross-matched, but methods that include parsing the code at one point are more rigorous and more efficient (and the previous code did not work as intended, yet). The one such method we picked uses abstract syntax trees (ASTs), because of the powerful and convenient library Rascal has for working with these constructs. Furthermore, ASTs allow us to easily detect type 2 clones as well, once the detection of type 1 clones is implemented. However, initially the focus is on detecting type 1 clones, especially due to time constraints.

Our approach is based on a paper by Baxter *et al.*[1]. For this approach, first the ASTs of a project are generated and a map is initialized for storing one of each node in the AST. The AST is visited, and for each node we check whether it is present in the map. If not, the node is added as a key, with its location and mass as value. If the node is already present in the map, the location and mass are added to the value. In this way, clones that occur many times get more and more corresponding locations stored in the map. One subtlety is that all nodes have a unique location, so before being able to check their equality, their location must be erased. To work around losing information, a dummy instance of the node gets its location erased, and is then compared to the map keys.

[1] `http://leodemoura.github.io/files/ICSM98.pdf`

From this elaborate map, a new map is instanced, which only holds the keys that have multiple values, i.e. the clone nodes. During the building of this new map, a set is also constructed, which holds the children of clone nodes that occur as often as their parents. This is because these children are redundant; they only occur within bigger clones. In a separate loop these clones are removed from the clone map again. This cannot be done within the first loop, due to the unsorted nature of maps. Note: clones that occur multiple times within a single bigger clone are still counted, as this may still be desirable knowledge.

We use maps for finding duplicate nodes, because the inherent hashing of maps makes this very fast. The elaborateness of the maps is to make sure no knowledge is lost. The process of getting the final map with clones contains three loops (which some may think is a lot) because editing something that is being looped over is bad practice, and because maps are unordered. The latter is an issue, because if maps would be ordered, the last two loops could be integrated; children could be ignored when building the map, rather than having to be removed later.

3 Visualization

Due to the state of visual libraries within Rascal, we decided to use javascript for the visualization of our clone detection. We chose the d3js library because of our experience with this library in past software projects, and options for interactivity are native to this library. Our implementation takes in a json file in which the specifics of the clones are logged, and turns it into a chord diagram, with on the edge of the circle the relevant filenames within the project, and the connections represent clones that occur in both connected files. This json file can be written as from the rascal project, for a given source project. The d3 library fetches the relevant objects from the file and creates constants, from which the chord diagram is initialized. Our visualization has been successfully tested on both the smallsql and the hsqldb project. Fig. 1 shows our visualization with regards to the smallsql project. What we still wanted to implement was that, interactively with the user, different elements in the diagram could be highlighted for clarity: when hovering the cursor over a node, the links connected to this node change colour. When clicking a node, a text window would display the clone code. Unfortunately these features could not be implemented due to time constraints.

The chord diagram helps in maintenance efforts by showing where clones

occur, and how often each clone occurs. The interactive elements we still wished to implement would further aid in such efforts by increasing the readability of the diagram, and by allowing the maintainer to see how big the clones are and help them in deciding how they may wish to handle the clones.

We chose the chord diagram of the d3js library over other visualization techniques, because we find most techniques lacking. The arc diagram for example is poorly scalable, as are diagrams that show node structures. Dot plots appear to lack what is required of figures in the first place, namely being easy to interpret. Libraries other than the d3js were not a good option, as they may not as readily allow for interaction as the d3js, and our experience with other libraries is simply lacking.

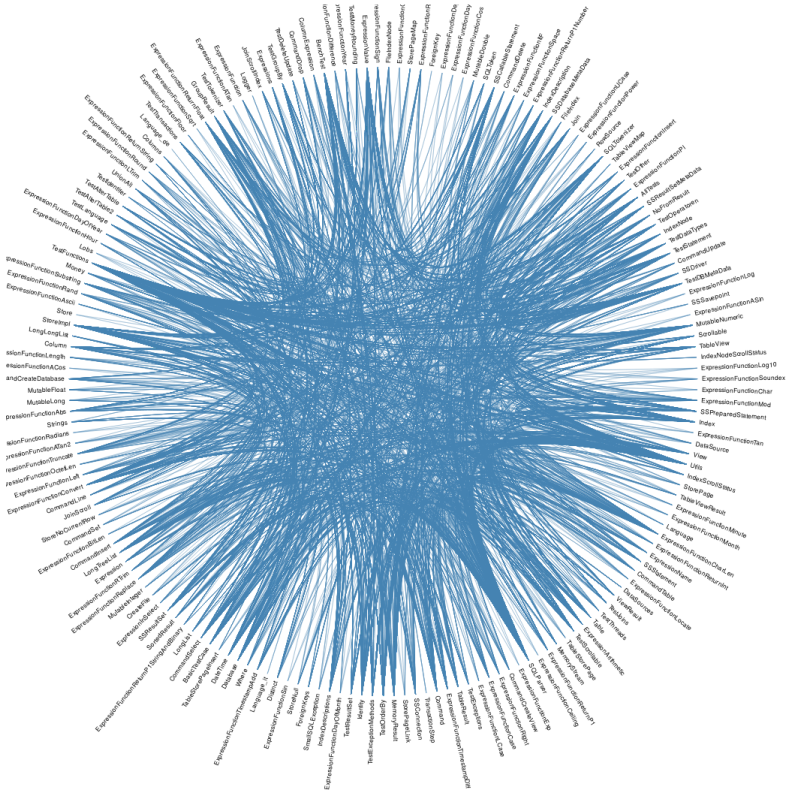

Figure 1: The smallsql project, as visualized by our code.

4 Testing

As a test for the code, we built a simple java project with several different clones which it should (or should not) detect. By visual inspection of the code and comparison to the result of the clone detection on it, we verified that the code works as intended. We included simple clones, clones containing smaller clones, clones containing multiple smaller clones (subtly different from previous statement), code that is not a clone (to verify that we do not get false positives) and clones but with comments in different places. Our test shows that simple clones are detected, small clones that are contained in bigger clones are not detected, unless it occurs multiple times in the same big clone, or if the small clone occurs outside of the big clone, too. Code that occurs only once in a project is ignored and comments do not affect clone detection. All of these results are as expected and as intended.

5 Results

The following output has been generated by our tool regarding the smallsql java project:

```
Starting clone detection at : 23:19:33
Start detection of clone type 1
Number of unique nodes: 29921
Ended clone detection at 23:20:02.749+0000
Number of nodes within clones: 21082
This gives a percentage of clone nodes of 70.45887504 % (including originals)
Number of different clones: 3179
```

This output was generated for a minimum node mass of 4. We see here that the code runs in only 30 seconds for the smallsql project, which I think is nice (especially compared to the run time required for our assignment series 1). It also shows that the greater part of the nodes in the project appear within clones, which seems problematic. What this means is subtle however, as the most “important” nodes are close to the root (and the root itself) and less likely to be part of the clones, while less interesting nodes, such as simple constants, occur far more often in clones; clones are detected top-to-bottom, which means that they always contain some leaves of the AST, even though these leaves are less interesting from a maintenance point of view. An example of this: refactoring code to replace ”1+1” everywhere is less interesting than being able to remove entire files of code due to redundancy. However, within the clone percentage we found, essentially the former is more often present than the latter. Note: I’m not sure how how

much worse it is to look at the number of nodes compared to the number of lines. LoC does not seem to tell you as much about a given code as you might hope it does.

The following output has been generated by our tool regarding the hsqldb java project, again with a node mass threshold of 4:

```
Starting clone detection at : 23:22:29
Start detection of clone type 1
Number of unique nodes: 185241
Ended clone detection at 23:25:57.273+0000
Number of nodes within clones: 143107
This gives a percentage of clone nodes of 77.25449550 % (including originals)
Number of different clones: 20892
```

Again this code runs in a most reasonable amount of time, especially considering that ASTs have to be built as well, within the logged period of time. We see that the clone percentage exceeds even that of the smallsql project, suggesting a great potential for refactoring to improve the code, though the same pitfall applies as was discussed before.