Practical Lab Series 2 - Code Duplication

Jovan Maric - 10443762
Hector Stenger - 10398872

Introduction

The following document is written to complement our achievements of the second assignment for Software Evolution. A clone detection tool which emphasizes what we have learned in the second part of the course.

This document contains two parts: The clone detection part and the visualization part. The clone part contains an explanation of the “backend”, which is built with the programming language called Rascal[1]. The visualization contains the “frontend”, which takes the generated results from the backend and transforms them into readable data. The frontend is written with the currently trending language called javascript[2] and the web development framework React[3].

Clone detection

For detecting clones we reused some of the logic contained in the first assignment. At the core of our implementation still stands the fitting of different sized windows. We have applied type-1 and type-2 code detection, as depicted in “A Survey of Software Clone Detection Techniques[4]” and “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach[5]” .

The goal of the clone detection tool is to provide maintainers of software with a proper tested and hackable tool, with a focus on known data structures and simple code.

We managed to compute the code-duplications of our own fixtures and smallSQL, however we did not manage to do this on the HSQLdb project. The reason is likely that our implementation runs out-of-memory as it always gave this error after a while.

Our clone detection can be split up in the following phases:

1. We load all the files from the project into memory. Before setting the values we do a little of cleaning. This cleaning consists of parsing the files individually and visiting the nodes so we can clean different things like ‘vardecids’, string literals etc. After this we un-parse the file back to a list of string and trim the file of comments while returning each individual file as a tuple with its original index number. This is important because removing white spaces and comments will have an effect on the line numbers. If we would not be storing the original line numbers there would be no exact way of telling what the correct line numbers would be.

[1] http://www.rascal-mpl.org/
[2] https://developer.mozilla.org/bm/docs/Web/JavaScript
[3] https://reactjs.org/
[4] http://www.cs.uccs.edu/~jkalita/papers/2016/SheneamerAbdullahIJCA2016.pdf
[5] https://www.cs.usask.ca/~croy/papers/2009/RCK_SCP_Clones.pdf

Storing all the cleaned files in an in memory data store gives us the advantage that we don't have to call any expensive IO functions in the remainder of the code. This greatly improves our runtime and we will explain that in a second.

2. The most important part of our algorithm is the fact that we are working with windows. These windows represent lines of code and we create every possible configuration of windows for the project. For instance the first five lines of a file are located in two different windows, if of course our window size is four. The first four lines are located in the first window, while the second window consists of the second until the fifth line.
3. The different windows in this case are the actual patterns we are trying to match with each other. Because they are normalised in the first phase we have a bigger chance of actually fitting different windows together. To check whether this is the case we let Rascal hash the window content to an unique identifier so it can store the location of the window in the corresponding value for that key. If we do this for every pattern we have we could eventually filter out all the values that have two or more values. Two or more values in this case mean that two 'different' windows hashed to the same value and are therefore proven to be equal.
4. As we now have all the different locations that are matched with each other we can try to expand the window and check whether the values still fit. By expanding we mean adding an extra line to the pattern. So four line patterns become five line patterns etc. We can do this because we store the original file and line number location for each match and because we use an in memory data store for accessing files we do not have to wait for the completion of an expensive IO operation.

   After getting the increased window we do the same thing as before. We hash that window value and store the line numbers and locations of that pattern in the corresponding value. Whenever we are expanding windows we store all the increased windows in new maps.
5. After increasing the windows we filter the in memory data store, which in our case is basically a map, for all the keys that have two or more values related to it. Just like we did before. Because we are using a new map every time we expand we can easily check after filtering if the size of the map equals zero. If the size of the filtered data store equals zero we know that expanding the window did not yield us any new increased patterns.
6. If by any chance we do get increased windows that are duplicates there is one last thing we need to do before again expanding the window size. There is a chance that the previous window of four or x-1 is contained in the bigger window of x. For instance it could be that all the matches of four could be expanded by an extra line. Therefore we need to see if the previous matched locations are a subset of the newly expanded locations. If the previous smaller windows are contained within the one bigger window we can safely remove the smaller windows as they are subsets. If however they are not, which means they aren't a subset of the bigger window we keep them.
7. After checking all the newly found windows for subsets we increase the window size and in our case call recursive the same function but with a higher window size. This way of gradually expanding windows gives us the opportunity to decrease the haystack on every iteration, lowering the amount of comparisons that are needed and

also the values stored in the data store.

One slight variation in this logic is out of bound checking for reloading increased windows. Any increased window that goes out of bound just doesn’t get stored in the new memory data store. Before retrieving a new window we check whether the size is lower than the index plus window size we are trying to get.

One large downside of doing it with windows is that big windows are probably a lot more expensive to store in the datastore. Hashing and comparing big patterns, say for instance window sizes of 50 lines, might take a while. We are not completely sure but we blame this fact for the high runtime our clone detection has with the big project. Small windows are easily processed and mostly done within five seconds, big windows however might take more than ten minutes. We tried to profile to discover the bottleneck it but could not locate the so to say origin of the rascal.

To counter this we could have changed the implementation a bit. What we could have done is also remove import statements which we currently don’t do. Import statements are in now way related to bad code practices. There is no design pattern that could be introduced in order to clean the amount of import statements once has in its source code. Therefore they should be taken into account when calculating clones.

Another thing is we could add ‘optimistic’ window changes. Rather than adding one each iterations we could add for instance add two or three lines. Whenever we fail to match an increased window we could rollback and rechange the window size to one. In really big clones this would lower the amount of comparisons that are needed. On the other side you’ll need more logic to check subsets and lowering the window size after failing to match increased window might give you headaches.

Pseudo code:

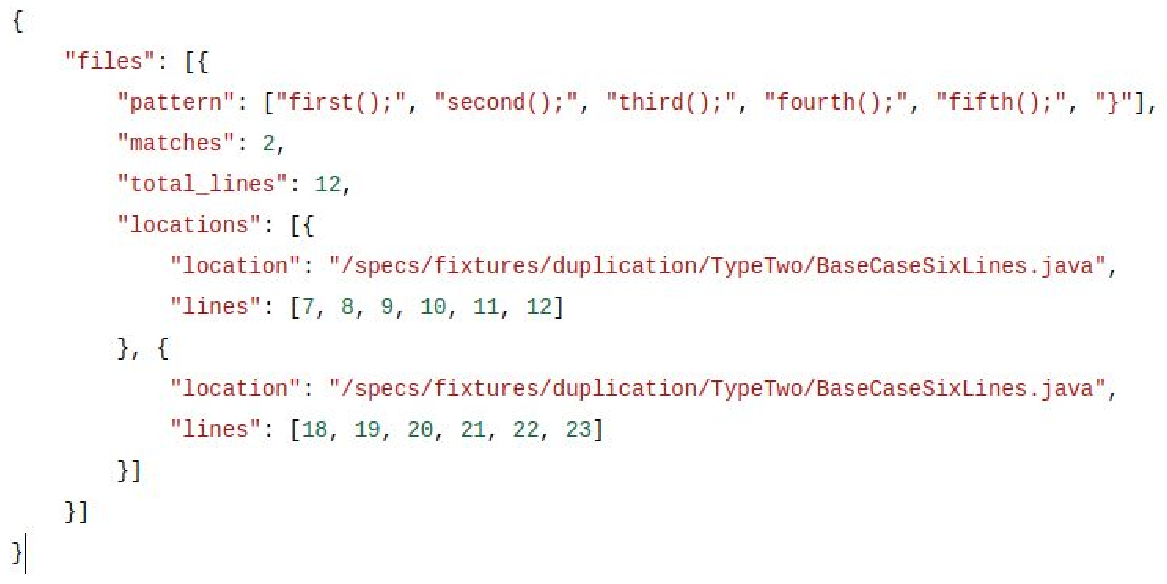
```
1. Retrieve and clean all lines from file into memory.
2. Create windows
3. Hash all different window string representations over Map where location is
   the value.
4. Filter values on map where size is higher than two.
5. Increase window size
6. Use all found duplicates to increase size of windows and map these
   duplicates over a Map using same hashing logic as step two.
7. Remove previous duplicates if they are contained within the newly increased
   window size (subsets);
8. Keep repeating from step 4 until no new duplicates are found.
```

Json representation

We could have chosen multiple formats for the serialization of the, by the clone detection tool provided, data. However we wanted to keep the data as concise and readable as possible. Considering that JSON[6] is the easiest to integrate and natively supported in Javascript, we chose this representation.

An added plus is that most developers have experience with this data representation, which therefore lowers the threshold for understanding the application. This improves the maintainability and makes it easier to understand what the clone detection tool does and how the tool works.

Figure 1. depicts a snippet of the serialized data



```
{
    "files": [{
        "pattern": ["first();", "second();", "third();", "fourth();", "fifth();", "}"],
        "matches": 2,
        "total_lines": 12,
        "locations": [{
            "location": "/specs/fixtures/duplication/TypeTwo/BaseCaseSixLines.java",
            "lines": [7, 8, 9, 10, 11, 12]
        }, {
            "location": "/specs/fixtures/duplication/TypeTwo/BaseCaseSixLines.java",
            "lines": [18, 19, 20, 21, 22, 23]
        }]
    }]
}
```

Figure 1: JSON data representation

The patterns refer to the patterns which are found on other locations. The matches refer to how many other locations are found. The total_lines refer to the total amount of found duplications in the locations.

Each location depicts the location in the repo, and on which specific lines they are found. This data is used in the visualization to correctly display the file and highlight the lines.

Tests

To make our code understandable for ourselves and to be sure that it works, we enforced a test-driven approach. The tests cover almost all aspects of the code duplication tool. We chose not to test functionality which is provided by Rascal, as we assume that the provided api is properly working and tested.

As with the previous assignment, we continued to make use colors for when tests succeed or fail. This gives a clear visual impulse on to where something has failed, which helped a great deal in debugging and therefore saved us time.

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

The visual aspect of the tests, even if only subtle, make the testing and understanding process of our tool much more tolerable and concise. While the high code coverage aids in the understandability of the used algorithms and design decisions of the underlying code. Therefore creating an easier learning process and increasing the understandability for new and current maintainers of the clone detection tool.

For the visualization part we considered to implement testing, however this proved to be a bit too complicated. Our first impression was to test the underlying structures, however this did not provide us with the information we wanted. The visualization is quite simple, and therefore would result in testing too fine-grained bits of code, which in the end wouldn't say enough about the quality. We therefore agreed to go by the End-to-End approach, which would test the application as a whole. However, as we thought about it more, we came to the conclusion that because of the simplicity of the application we wouldn't be testing any differently than from what we were seeing. Therefore we chose to leave this untested and cross compared the results with the serialized data to be sure that it was correct. This gave us more time to work on making the clone detection tool more performant.

Clone detection statistics

The results in figure 2 depict the statistics which were retrieved by running the clone detection tool on the SmallSQL[7] and HyperSQL[8] projects. The results give a good indication of the quality of the code in the projects.

Statistic	**SmallSQL**	**HSQL**
Percentage of duplicated lines	-/24050	-
Number of clones	3759	-
Number of clone classes	183	-
Biggest clone (in lines)	114	-
Biggest clone class	/smallsql-0.21/database/language/Language.java	-

Figure 2: Clone detection statistics SmallSQL and HSQL

Visualization

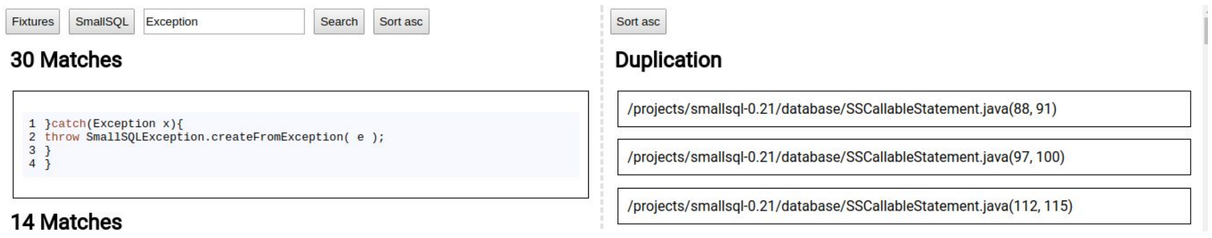
As has been implied earlier, we used a web development framework to write the frontend part of the clone detection tool. The main requirement we tried to adhere to is the proper display of the by the backend provided data. We chose to display the duplicates in a way which mimics how tools like github and other version control systems display changes. Therefore providing an environment which is familiar and explanatory by itself to most maintainers.

[7] http://www.smallsql.de/
[8] http://hsqldb.org/

The main goal which we wanted to accomplish with the visualization was to help a maintainer get quick and insightful information in the provided data. The maintainer can change what is displayed by means of filtering or sorting, therefore forming the data in a way which complies to what is needed.

By using React, the library proved to be a great choice for changing how data is shown, as it is fast and reliable. It didn't only make the development aspect enjoyable, but it also saved us a ton of time on debugging. We both have experience with java user interfaces, and simply didn't want to travel on that path again.

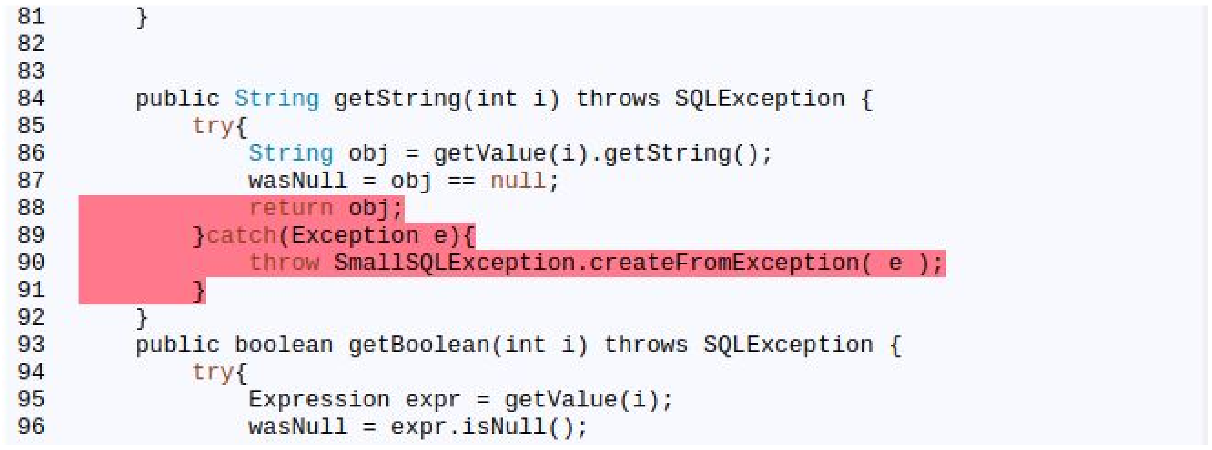
However, as depicted in the figures 3. and 4., we did not focus a great deal on styling. Our primary focus was on making the visualization usable. The visualization is simple and therefore does not need fancy eye candy to bring the message across.



```
1 }catch(Exception x){
2 throw SmallSQLException.createFromException( e );
3 }
4 }
```

/projects/smallsql-0.21/database/SSCallableStatement.java(88, 91)
/projects/smallsql-0.21/database/SSCallableStatement.java(97, 100)
/projects/smallsql-0.21/database/SSCallableStatement.java(112, 115)

Figure 3: Matches and duplications



```
    }


    public String getString(int i) throws SQLException {
        try{
            String obj = getValue(i).getString();
            wasNull = obj == null;
            return obj;
        }catch(Exception e){
            throw SmallSQLException.createFromException( e );
        }
    }
    public boolean getBoolean(int i) throws SQLException {
        try{
            Expression expr = getValue(i);
            wasNull = expr.isNull();
```

Figure 4: Code highlighting

We chose to divide the visualization into a left part and a right part. The left part shows the match and the amount of duplications that have been found. The right part shows the duplication locations. To view the duplications on the right, you first need to select a pattern on the left. The patterns can be sorted and searched for.

When a duplication on the left is clicked, it shows the file location content, and shows the duplication in a highlight.