Software Evolution: Series 2 Design Document

Matt Chapman, Dennis van der Werf

December 23, 2016

1 Introduction

This document serves as a basic design document accompanying the creation of the *series 2* assignment deliverable: a working clone detection tool with visualisation of clones.

The goal of this document is to explain the basic workings of the tool, along with some rationale for the design decisions that were taken during development.

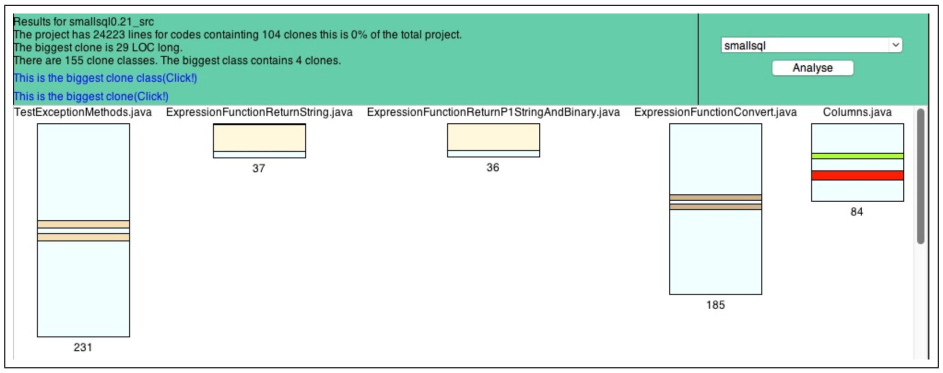
2 Using the tool

To begin using the tool, open the project in Eclipse, and begin a Rascal console session. In this console run the commands `import CloneVisualisation; startTool();` to start the tool.

In the top right corner of the screen there is a combobox for selecting one of two different projects: `hsqldb` or `smallsql`[1]. Select one of these projects and click *Analyse* to begin the analysis.

Figure 1 shows an example of the view presented when a project is analysed. The top bar (green) displays some basic clone metrics for the project in question, along with links that can be clicked to display the biggest clone and clone class.

[1]In it's default state, the clone highlighting in this project will not work correctly with the tool, due to it using `CR LF` for new lines. These characters are ignored when files are read by Rascal, but not when creating `loc` objects. It is necessary to write a script to handle replacing these characters - which will be made available in the project repository.



Results for smallsql0.21_src
The project has 24223 lines for codes containting 104 clones this is 0% of the total project.
The biggest clone is 29 LOC long.
There are 155 clone classes. The biggest class contains 4 clones.
This is the biggest clone class(Click!)
This is the biggest clone(Click!)

smallsql
Analyse
TestExceptionMethods.java
ExpressionFunctionReturnString.java
ExpressionFunctionReturnP1StringAndBinary.java
ExpressionFunctionConvert.java
Columns.java
231
37
36
185
84

Figure 1: Duplication Visualisation

The main view shows each file in the analysed project, with the file name listed above a box that is scaled according to the true size of the file. Under each box is the total file size in LOC (Lines of Code), albeit after the file has been filtered for comments and white space.

Clones are rendered as overlapping, colour coded boxes. Each clone class is rendered in a unique colour, and boxes of the same colour can be assumed to be members of the same clone class. Clicking on a clone class will render only the locations in which the clone class exists. The top bar will display metrics pertaining to the clone class in question.

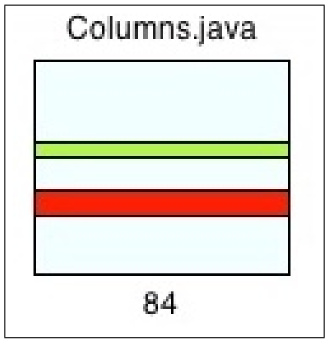


Figure 2: An example file with 2 clone classes

The files are sorted such that the files containing the largest clones are rendered first on the left. The files containing the smallest detected clones are on the far right of the visualisation.

Clicking on an instance of a clone class within this view will then open the file, with the clone itself highlighted in the editor. Clicking multiple clone instances will open multiple editor instances for easy comparison and verification of detected clones.

Clicking *return to overview* will return you to the start screen, where the entire project is displayed.

3 Design Rationale

This section discusses the major design decisions, as well as providing a basic explanation of the algorithm used for clone detection.

3.1 Satisfied Maintainer Requirements

As stated in the paper *Cognitive design elements to support the construction of a mental model during software exploration* by Storey et al., there are a number of possible requirements that a maintainer may have of a software tool. We believe that the tool we have created satisfies the following requirements[4] from this paper, in the following ways:

- **Improve Program Comprehension**: We fulfil this requirement by providing top-down comprehension facilities through the use of generalised statistics and metrics, and also through the visualisation itself. As the visualisation is very much a "top down" view of the analysed project, and shows an easy to read summary of detected clones through colour coded notation, we believe that this requirement is fulfilled.
- **Facilitate Navigation**: We have made the visualisation (and indeed some of the basic metrics) interactive. By clicking the rendered clones, the user can navigate backwards and forwards through varying *depths* of views, from an overall project view, to clone classes, to the clones themselves.
- **Reduce Disorientation**: By implementing our visualisation natively in Rascal using the `Vis` library, we have successfully minimised maintainer disorientation. The visualisation is rendered directly in a tab contained within Eclipse, and also opens all files (when the user instructs it to do so) in the Eclipse native editing environment - with the clone conveniently highlighted. We also ensure that views are re-used where possible, to prevent an overload of tabs and other windows building up.

3.2 Clone Detection Algorithm

Listing 1: Clone Detection Algorithm

```
Function detectClones(project)
{
    for(every file in project)
    {
        filter comments & white space;
        get first 6 lines of filtered file;
        hash first 6 lines;

        if(first 6 lines already in clone set)
            mark as duplicate;
        else
            add to clone set;
    }

    for(every file in project)
    }
        while(first duplicate is still a duplicate)
        {
            grow duplicate bounds by 1 line;
        }

        flag subsumed clones;

        remove subsumed clone classes;

        for(every other duplicate in file)
        {
            while(duplicate is still a duplicate)
            {
                grow duplicate bounds by 1 line;
            }
        }

        re-filter for subsumed clones;
        write results to file;
        render the results;
    }
}
```

The listing above (Listing 1) shows the basic pseudo-code behind the clone detection algorithm that we are using.

This algorithm works by detecting *Type 1* clones, that is "Identical code fragments except for variations in white space (may be also variations in layout) and comments"[3]. While it would be possible to write an implementation for *Type 2* clones (via the use of Abstract Syntax Trees and sub-tree comparison therein) we opted instead to extend, optimise and improve the clone detection algorithm we had in place from our *Series 1* deliverable.

While the pseudo-code in Listing 1 gives a succinct explanation of it's workings, some further explanation follows. The algorithm begins by obtaining all of the source files contained within a given project, and then filtering them to remove white space and comments. These elements are surplus to requirements and should not be considered in any clone detection algorithm. For each obtained and filtered file, we take the first 6 lines of that file and insert it into a map as the key - which makes use of hashing for the key, is much faster than direct string comparison.

If the key (the hashed 6 lines) already exists in the map then we flag it as a clone. Otherwise, it is added to the map structure. Once all files have been examined, the algorithm starts *growing* the first clone in each file, incrementally extending the clone 'window' by one line, and checking if this clone now subsumes any others. If it does, then the clone is flagged as a subsumed clone. If all clones in a class are flagged as subsumed then the clone class is removed from the data structure.

This 'grow' operation is then carried out on all of the clones to ensure that the clones are as big as possible. The clones are extended by one line at a time, and checked to see if they are still equal. Once one clone or the other in a given clone class is no longer equal, we store the upper bound of the windows size wherein the clones are still equal and move on to the next one.

Once all of the clones have been grown as far as possible, we re-filter for any further subsumed clones (to avoid duplicates). Once this final filtering step is completed, the clones and clone classes are then passed to the reporting and rendering libraries to create the final output of the algorithm.

3.3 Visualisation

The text *Identifying and Removing Software Clones* by Koschke provides several examples of possible clone visualisations.

One visualisation in particular was created by Tairas et al. in their tool for *Visualization of Clone Detection Results*[5].

This visualisation makes use of three component parts, namely *boxes, lines and kinds.* Boxes are the containers representing the files, the lines represent the bounds of the various detected clones, and the kinds are represented by varying colours of the lines.

This type of visualisation (boxes representing files, arranged vertically and overlaid with coloured bars to represent areas of the file) also appears several times in the lecture slides *Towards Visual Software Analytics* by Paul Klint[1].

Being that our aim was to create a tool for a maintainer, as opposed to just blindly print out metrics, we opted to reproduce this visualisation using the Rascal `Vis` library. There were certainly other options available to us (namely various JavaScript libraries with "pre-baked" visualisation options that only require the "plugging in" of existing data, though there were two main reasons that we did not take this avenue:

- **Maintainer requirements:** As stated in a previous section, our intention was to meet the requirement of reducing maintainer disorientation. If we utilised an external library (as opposed to the Rascal library - which renders its results directly in Eclipse) then it was likely that the maintainer would have to leave their IDE in order to view results, thus rendering our attempt to minimise maintainer disorientation somewhat pointless.

- **Platform Familiarity:** While using an established Javascript library could be viewed by some as the "easy option", the fact that we were already familiar with Rascal as a platform meant that the "educational overhead" of learning to use the Rascal `Vis` library was far lower than attempting to implement visualisation in Javascript.

The built-in Rascal libraries do offer the possibility of using the `Outline` object to create visualisations such as this. Unfortunately the existence of this object came far too late in the development process to be of any use, and as such the process below become a valuable *learning experience* regarding reading documentation and planning before beginning an implementation.

Our visualisation starts by generating a box for a given file. This box is of fixed width and it's length is bounded by the length of the file itself. Once this container is created, we create a list of boxes, one for each clone contained within the file. These boxes are again of fixed width, and bounded by the length of the clone they are going to represent. As we know the start line of any given clone, they are aligned vertically using `FProperty valign(n)` where `n` is set according to how far through the container file the start line of the clone is situated. These boxes are then added to a list and overlaid on the container box, which is in turn added to a `vcat` object along with the file name and file length. These figures are then added to a `hcat` object which contains all of the completed figures.

4 Testing

The tool has built in unit tests that make use of the Rascal testing framework. The testing framework allows the creation of methods with the modifier `test`, which informs the Rascal interpreter that they are unit test methods.

The tool has close to 100% coverage of the `CloneDetection.rsc` Rascal module. These tests cover generalised validity tests such as running the tool against `hsqldb` and checking for thrown exceptions, and processing empty projects - but also has numerous tests for checking the validity of the output from helper methods that are used to manipulate data and carry out various other operations when the clone detector is run.

To execute the tests, run `Series2UnitTests;` followed by `:test`. The Rascal test framework will execute all of the methods in the test suite and return a test report in the console.

5 Caveats

5.1 Known Issues

1. There is some evidence to suggest that the *filtering* algorithm used to strip comments and whitespace before running the clone detection algorithm, does not produce perfect results. Indeed we have noticed that there are some styles of multi-line comment that the regular expression we are using may not catch in certain situations.

2. The clone detection algorithm only detects *Type 1* clones at this time. Future iterations would make use of Abstract Syntax Trees and subtree comparison to detect *Type 2* clones, but time constraints lead us to extend our existing *Type 1* detection algorithm rather than beginning a new implementation.

References

[1] Paul Klint. Towards visual software analytics. Lecture Slides.

[2] Rainer Koschke. *Identifying and Removing Software Clones*, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-76440-3. doi: 10.1007/978-3-540-76440-3_2. URL `http://dx.doi.org/10.1007/978-3-540-76440-3_2`.

[3] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University at Kingston, Ontario, Canada, September 2007.

[4] M.-A.D Storey, F.D Fracchia, and H.A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171 – 185, 1999. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/S0164-1212(98)10055-9. URL `http://www.sciencedirect.com/science/article/pii/S0164121298100559`.

[5] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 50–54, New York, NY, USA, 2006. ACM. ISBN 1-59593-621-1. doi: 10.1145/1188835.1188846. URL `http://doi.acm.org.proxy.uba.uva.nl:2048/10.1145/1188835.1188846`.