|  |
| --- |
| UVA Master Software engineering |
| Design Document |
| Software Evolution Lab2 |
|  |
| **Jeffrey Klaassen & Jonatan Felipez Queijo** |
| **10/12/2014** |

|  |
| --- |
| This document describes our design and provides a rationale for our most important design decisions for the Lab2 assignment in the master course Software Evolution. |

# Program structure

The tool is structured as follows:

* The **CloneDetection** module handles all of the clone detection
* The **Eval** module handles the testing harness of the project
* The **Export** module handles the exporting of data for use in visualization

Our argumentation for choosing this structure is that it provided us the most intuitive separation of concern that we could think of.

# Clone detection approach

For clone detection multiple approaches were considered. We tried to globally assess their fitness for our assignment, keeping in mind that our time was limited.

Approaches considered

* AST (Baxter)
  + Use AST sub-trees and bucket-sorting to find clones
* Token-based
  + Tokenize code and try to find clones by looking for duplicate token sets
* SIG sequence based as made in the previous practical
  + Filter comments and empty lines and look for duplicate sequences of 6 lines
* AST + Suffix trees
  + Same as Baxter, but build suffix trees to

#### Approach chosen: modified AST Baxter

Since Rascal has an AST available that we had some experience with, we decided to not go for a token approach.

We did not choose for suffix trees as these are added for performance gain and we were primarily interested in correct output. If the performance of the algorithm is not fast enough, a possible performance gain can be achieved by implementing suffix trees can be added instead of hashing using sub-trees.

We decided to use the basic algorithm presented in the paper by Baxter modulo the hashing based on the similarity of clones.

## Clone candidate elicitation

We only consider **sequences of statements** as clone candidate. We do this as sequences of statements are often the result of copying and pasting and might be an opportunity to abstract (e.g. moved to a new method)**.**

## AST statement sequence

Statement sequences are considered as clone candidates (depending on the mass of the sequence). The reason for this is that we want to be able to find partial clones in declarations, for example the first half of the body of a method.

This also means that we do not consider single statements as clone candidates unless their weight (the amount of statements they are containing) is bigger than the given threshold.

## Threshold

Since code fragments can be arbitrarily small, we need to somehow set a threshold that determines whether a code fragment is big enough to be considered a clone so that noise can be reduced.

The methods we considered were:

* **Count the nodes in the code fragments’ AST**, this method was inspired by Baxter
* **Count the lines of code of a fragment**, this method was inspired by the SIG paper

Since we decided to use an AST, we felt inclined to use the Baxter approach. Furthermore, the AST did not filter the comments, making the second approach sensitive to heavily commented fragments.

Eventually we decided that the “**mass**” of a fragment would be **calculated by counting the statements in a code fragment**. Our reason for this is that we found from experience that there is a positive correlation in Java between the amount of statements a clone contains and the chance that they are a candidate for abstraction.

## Clone type detection

Our tool is capable of detecting so-called Type1 clones, which are literal clones. Our approach mimics the simplified method introduced by Baxter et al.

Type 1 clones are literal clones. This means that fragment A is a clone of fragment B if their AST is identical.

Maintainers could be interested in these clones because these are often the result of the copying and pasting of code which perhaps could be abstracted into a method to increase maintainability.

For example clones please consider the file that is exported after running the clone detection

#### Why does this work?

In the AST of the project we collect **each declaration** that has a bigger mass than the threshold as well as **all sequences of statements** that have a bigger mass than the threshold.

Given these, we look for exact AST matches in these collections. These exact matches are clones as shown by Baxter.

Since we only look for literal clones (Type1), it is enough to do an exact match on an AST. The fact that Type1 clones exhibit a transitive relation means that this algorithm gives us all the clone classes.

## Clone Classes (Sub-sumption)

Given all the clone classes we still need to **filter sub-clones to** be left with only maximal clones.

**A clone x is a sub-clone if**

**X** is contained within a clone **Y**   
AND all clones of **X** are contained in clones of **Y**

Using this as filter criterion, we filter all sub-clones from the clone classes, which leaves us only with all maximum clones.

# Testing Harness

* How did we test
* Eval class

# Visualization

We visualize our clone report in a html web application. The application can be run after the clone detection has been run. The first page shows .

We try to provide a top-down overview by default, which can be refined into a finer low level overview to facilitate root-cause analysis (“WHY IS THIS CLONE CLASS SO BIG?!?!”);

## Maintainability

* Why is this tool maintainable?
  + Asserts
  + Testing Harness (automatic testing)
  + Separation of concerns
* Why would it not be maintainable
  + Lots of pattern matching
  + Use of different languages
    - Back-end: Rascal
    - Front-end: Html, css, Javascript

# Future additions

To detect Type 2 clones, which are refactored clones, one would need to alter the AST of the program such that all name, type and function identifiers would be ignored if they were different. This can by done by visiting the tree and replacing these by “”.