Software evolution report series 2

- Simon Schneider
- Laurance Saess

This report documents the problems and solutions that were used for this code duplication assignment. Firstly, we will introduce the requirements of a maintainer in order to understand the goals of the report generation and visualization.

The following chapters explain different parts of the clone detection system and their inner workings. The final chapters discuss our decisions, that were made on the basis of the initial requirements, and possibilities to extend the tool.

Related files / projects

- Manual: manual.md
- Result tables: src/series2/docs/results/results.tsv
- Result JSON exports: /final_output/results
- Eclipse respoistory with test files: https://github.com/lauwdude/use-test-project
- Duplication visualisation:
    - Repository: https://github.com/schneider-simon/duplication-visualization
    - Online demo: https://software-evolution.schneider.click

Maintainer requirements

1. Give new maintainers a quick understanding of the program

Maintainers need to have an understanding of a program. There are multiple ways of getting an understanding at a global level of the program. In the paper [Storey, Fracchia, Müller, 1999] these are Macro-strategies. One strategy is called the "As-needed macro-strategies" strategy [Storey, Fracchia, Müller, 1999]. You only take a look at a part of the code when you needed it. With this approach, code clones give a very negative impact. When you want to make a change, you first have to find where the duplicates are, otherwise you can introduce bugs into the system. Our goal is to visualization of clone classes. We are going to resolve this by adding support for clicking so that you can see where else the same code is used.

2. Show clone hotspots

One issue in maintenance can be that duplicate is required to extent current functionality. When a maintainer wants to abstract these parts, he has to know on what locations these abstractions can be introduced. When you can find clones, and view how to clones look like, you can faster spot places where you can introduce abstraction. This is why we are making it possible to show the code per duplicate. Clicking in the clone will result in a overview of the code.

3. Help maintainer to reduce duplicated code quickly

When a maintainer wants to improve the code quality, it is useful to know in what part of the project the most duplicates are. This way, you know on what part of the project the most progress can be booked. It can be use as a guide for reducing complexity. To solve this, we added an overview of the largest code classes and a code preview panel.

Clone detection

We use an approach that uses the AST to detect code clones, as described by [Koschke, 2008].

What code types do we detect

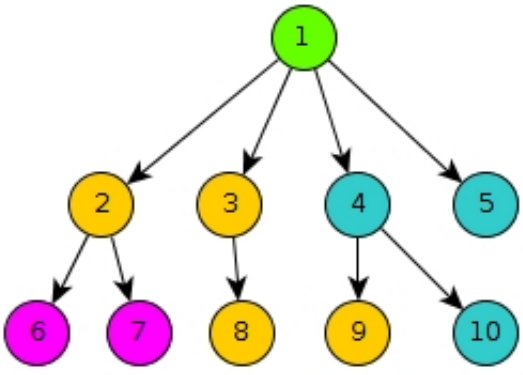
In order to detect software clones, we have to know what clones are and what type of clones exist. There multiple types of clones, some can be detected with a code analysis tool and others require code execution.

We are using Rascal for the detection tool and only want to apply static code analysis. According to [Roy, Cordy, 2007], the clones types for static code analysis are:

- **Type 1 clones**: The code fragments are identical, except for variations in white space and comments.
- **Type 2 clones**: The code fragments are structurally and syntactically identical, except variations are allowed in identifiers, literals, types, layout and comments.
- **Type 3 clones**: The code fragments are copied with further modifications. Statements can be altered in addition to variations in identifiers, literals, types, layout and comments.

Our tool focuses on all these types of clones with an AST clone detection approach. So, what do these types of clones mean to us. An AST is a tree structure of the code, so we are going to compare nodes instead of code fragments. Source-code can be displayed in an AST structure as follows:

Figure: Example abstract syntax tree with duplicate nodes



Than we can define clones like this (where the symbol == means comparing nodes without looking at the sub-nodes):

- **Type 1 clone**: `node 6 == node 9`, `node 7 == node 10` and `node 2 == node 4` then

node 2 is a type 1 clone of node 4, node 6 is a type 1 clone of node 9, node 7 is a type 1 clone of node 10.

- **Type 2 clone**: Node `node 9 == node 10` where identifiers, literals, types, and layout are removed. Then node 9 is a type 2 clone of node 10.
- **Type 3 clone**: Node `node 2 == node 3` and `node 6 or node 7 == node 8` where identifiers, literals, types, and layout are removed. Then node 2 is a type 3 clone of node 3. With type 3 clones you have to set a similarity threshold. It could be true that node 2 is a type 3 clone of node 3, but node 3 is not a type 3 clone of node 2.

The algorithm

The algorithm is the basic algorithm described by [Khatoon, Singh, Shukla, 2012]. The only difference is that we do not use an hash function ans that clone pairs are managed outside the detection code.

The hashes are used so that clones can be stored into buckets. We use relations instead so that we can use relation operations. There is a relation between node a and node b when they are registered as a clone.

In pseudo code:

```
* x is the clone type
* y is the project location
* z is the min number of sub notes
* z' is the min number of lines of code per node
doCloneDetection(x,y,z,z')
  type = x
  a = (for type = 1:100 2:100 3:30)

  ast = loadAstForProject(y);
  ast <- normalize @ type 2 / type 3
  astList = getAllNodes(ast)
  astList <- remove when subitems less than z
  astList <- remove node when less than z' lines of code
  astList <- remove with invalid location

  @no duplicate compares
  @Do not compare when node is a subnode of that node
  @similarity of nodeA and NodeB > a
  compare astList astList to nodeA nodeB:
      return add connection:<nodeA.1,nodeB.l>

  return set nodes:astList
  return
```

Parameters

The real project has the following parameters:

- set[Declaration] ast
- bool normalizeAST
- int minimalNodeGroupSize
- int minimalCodeSize
- real minimalSimularity

What is a little bit different than the pseudo code. In the section we are going to describe what every parameter is and how it translates to the real project.

X is the clone type

You can define the clone type in the pseudo code. In the real project you have to translate it like this:

Type 1:

- normalizeAST = `false`
- minimalSimularity = `100`

Type 2:

- normalizeAST = `true`
- minimalSimularity = `100`

Type 3:

- normalizeAST = `true`
- minimalSimularity = `50` <- or any other similarity factor you prefer.

For these settings, normalizeAST will remove all information that are type or name related. With this setting `int test` is the same as `float test2`.

For these setting, minimalSimularity will defined a percentage of how much of the node has to be the same to be considered equivalent.

Y is the project location

The pseudo code will genarate an AST based on the location of the project. The clone detection function requires the AST already what is done in the main.

- ast = `createAstsFromEclipseProject(createM3FromEclipseProject(y), true);`

Z is the min number of sub notes and Z' are the minimum lines of code per node

You can display the AST as an tree. When you compare the nodes, there will be a lot of useless small clones. This parameter can be used to define a minimum size. Nodes are only considered that contain z sub nodes or has minimum z' lines of code.

- int minimalNodeGroupSize = z
- int minimalCodeSize = z'

The node normalize function

The nodes are normalized for type 2 and type 3 clones. We replace all identifiers, literals, types and layout in the AST with a static value.

```
//Will remove all items that are inrelevant for type 2 and 3
public node normalizeNode(node nodeItem) {

    return visit(nodeItem) {
        case \enumConstant(_, args, cls) => \enumConstant("enumConstant", args, cls)
        case \enumConstant(_, args) => \enumConstant("enumConstant", args)
        case \class(_, ext, imp, bod) => \class("class", ext, imp, bod)
        case \interface(_, ext, imp, bod) => \interface("interface", ext, imp, bod)
        case \method(_, _, a, b, c) => \method(defaultType, "method", a, b, c)
        case \method(Type a,str b,list[Declaration] c,list[Expression] d) => \method(a,b,c,d)
        case \constructor(_, pars, expr, impl) => \constructor("constructor", pars, expr, impl)
        case \variable(_,ext) => \variable("variableName",ext)
        case \variable(_,ext, ini) => \variable("variable",ext,ini)
        case \typeParameter(_, list[Type] ext) => \typeParameter("typeParameter",ext)
        case \annotationType(_, bod) => \annotationType("annotationType", bod)
        case \annotationTypeMember(_, _) => \annotationTypeMember(defaultType, "annotationTypeMember")
        case \annotationTypeMember(_, _, def) => \annotationTypeMember(defaultType, "annotationTypeMember", def)
        case \parameter(_, _, ext) => \parameter(defaultType, "parameter", ext)
        case \vararg(_, _) => \vararg(defaultType, "vararg")
        case \characterLiteral(_) => \characterLiteral("a")
        case \fieldAccess(is, _) => \fieldAccess(is, "fa")
        case \methodCall(is, _, arg) => \methodCall(is, "methodCall", arg)
        case \methodCall(is, expr, _, arg) => \methodCall(is, expr, "methodCall", arg)
        case \number(_) => \number("1")
        case \booleanLiteral(_) => \booleanLiteral(true)
        case \stringLiteral(_) => \stringLiteral("str")
        case \type(_) => \type(defaultType)
        case \simpleName(_) => \simpleName("simpleName")
        case \markerAnnotation(_) => \markerAnnotation("markerAnnotation")
        case \normalAnnotation(_, memb) => \normalAnnotation("normalAnnotation", memb)
```

```
31          case \memberValuePair(_, vl) =>
    \memberValuePair("memberValuePair", vl)
32          case \singleMemberAnnotation(_, vl) =>
    \singleMemberAnnotation("singleMemberAnnotation", vl)
33          case \break(_) => \break("break")
34          case \continue(_) => \continue("continue")
35          case \label(_, bdy) => \label("label", bdy)
36          case Type _ => defaultType
37          case Modifier _ => lang::java::jdt::m3::AST::\public()
38      }
39  }
40
```

The similarity function

We looked at multiple similarity function. One is described by [Baxter, Yahin, Moura, Sant'Anna, Bier, 1998]. Where:

```
Similarity = 2 x S / (2 x S + L + R)
        where:
        S = number of shared nodes
        L = number of different nodes in sub-tree 1
        R = number of different nodes in sub-tree 2
```

Lets assume that two nodes are going to be compared:

```
Node1.sub = [1..50]
Node2.sub = [41..90]
```

They share 10 nodes that are the same. Sub tree L and R have 40 nodes that are different.

This is going to result in:

```
2 * 10 / (2 * 10 + 40 + 40) = 0.20
```

In another case:

```
Node1.sub = [1..50]
Node2.sub = [21..70]
```

They share 30 nodes that are the same. Sub tree L and R have 20 nodes that are different.

This is going to result in:

```
2 * 30 / (2 * 30 + 20 + 20) = 0.60
```

In another case:

```
Node1.sub = [1..50]
Node2.sub = [11..60]
```

They share 40 nodes that are the same. Sub tree L and R have 10 nodes that are different.

This is going to result in:

```
2 * 40 / (2 * 40 + 10 + 10) = 0.80
```

What translates in rascal to this:

```
public num nodeSimilarity(node nodeA, node nodeB) {
    list[node] nodeList1 = nodeToNodeList(nodeA); // 50 nodes
    list[node] nodeList2 = nodeToNodeList(nodeB); // 50 nodes

    list[node] sameItems = nodeList1 & nodeList2; // 40 nodes
    int sharedNodes = size(sameItems); //40

    num nodeADiff = size([nodei | nodei <- nodeList1, nodei notin sameItems]); //10
    num nodeBDiff = size([nodei | nodei <- nodeList2, nodei notin sameItems]); // 10

    //(2.0 * 40 / (2 * 40 + 10 + 10)) * 100.0
    num sim = (2.0 * sharedNodes / (2.0 * sharedNodes + nodeADiff + nodeBDiff)) * 100.0;

    return sim;
}
```

Finding what lines are duplication

We use an custom algorithm for detecting the amount of duplicate lines. The algorithm works like this:

1. Get all locations that contain an duplicate
2. Request from the M3 model, all the locations with comments
3. Go through every duplication, and get the lines for every location and look per line:
    1. If it is a one line comment
    2. If it is an empty line
    3. If the line is in a multi-line comment
4. When when of above is true, the line is counted as a non-duplicate

5. When above is false, the line is counted as a duplicate

We wrote the following unit test:

```
import javax.annotation.Generated;

class dupTest {
    public void testM() {           //Comment
        int i1 = 1 + 1;             //Comment
        int i2 = 1 + 1 * 2;         //Comment
/*test*/int i3 = 1 + 1 / 4;/*test*/
}

    public void testM2() {          //Some test comment
/*test*/int i1 = 1 + 1;
/*
 *
 * Test
 *
 */
 /*est*/int i2 = 1 + 1 * 2;/*test*/
        /*
         * test
         */
        int i3 = 1 + 1 / 4;
    }

    public void testM3() {
        int i2 = 1 /*test*/+ 1 * 2;
        //test
        /*


        */
        int i3 = 1 + 1 / 4;
        /*
        int i1 = 1 + 1;
        int i2 = 1 + 1 * 2;
        */
    }

}


```

The file has the following duplicate lines:

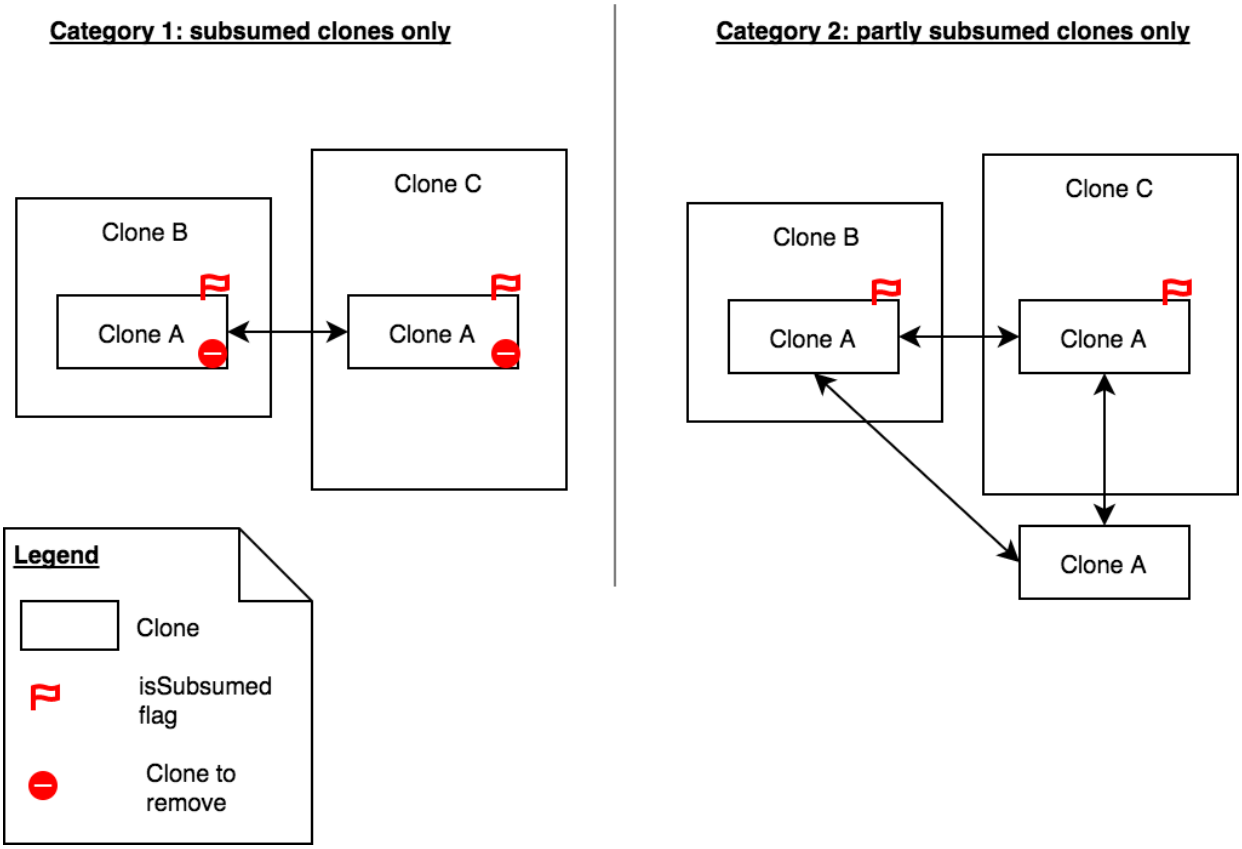
```
[4,5,6,8,10,11,21,22,25,31]
```

What is correct.

Clone classes

The diagram below shows our approach to delete unnecessary clones, which are subsumed by other clones.

Figure: Detecting clone classes



At first, we flagged every clone that was included in another clone. For this, we can use the underlying location of a cloned node.

```
1  public bool containsLargerLocation(NodeIdLocation nlA,
   list[NodeIdLocation] nls){
2      return any(NodeIdLocation nlB <- nls, nlA.id != nlB.id && nlB.l >=
   nlA.l);
3  }
```

This code above can tell us if a node (`nlA`) is located inside a clone of the node list (`nls`). In this first approach, we simply flagged and deleted the clones from the clone graph.

We soon recognized another type of clones (category 2), in which included clones are not deleted because they appear outside of other clones. In order to build the right clone classes, we use the following algorithm:

1. **Flag** every clone that is fully included in another clone.
2. **Build a graph** of clones

1. Use the **transitive closure** for clone type 1 and 2. Type3 clones must not be transitive.

3. **Remove a class of clones** from the output if all nodes are flagged as deleted.
4. **Output** the clones as a graph that builds a cluster for every clone class (please see visualization below)

Visualization

Enclosure diagram

In his book, Adam Tornhill describes the methods and visualizations that criminologists use to find murderers and describes how software engineers can learn from this, by using similar methods to detect code smells or "dirty" developers. [Tornhill, 2015]

One example of visualizations that criminologists supposedly use maps in which they highlight hotspots in order to predict future crimes.

Figure: Clone enclose diagram to show clone overview (Step 1, before zoom)

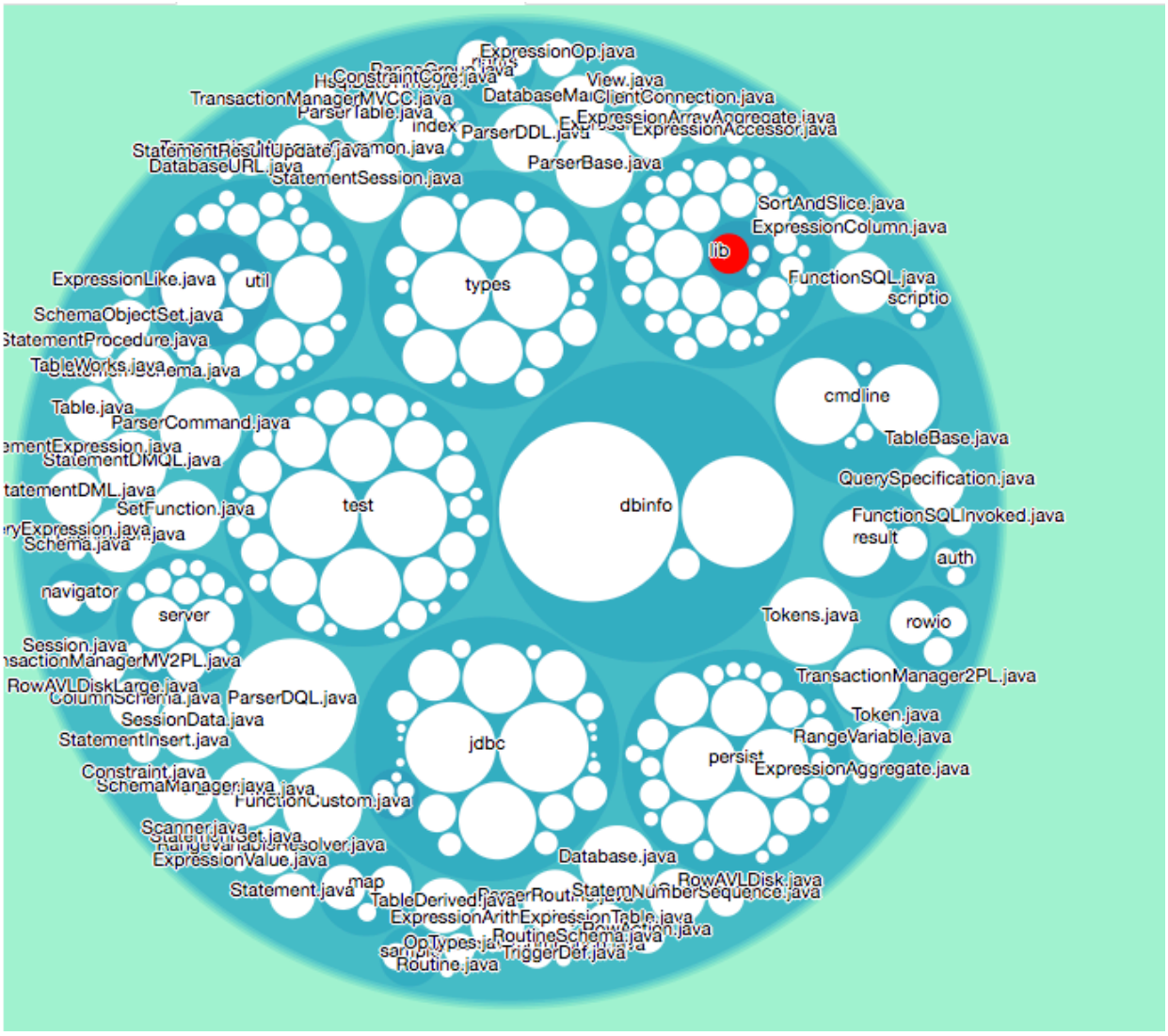
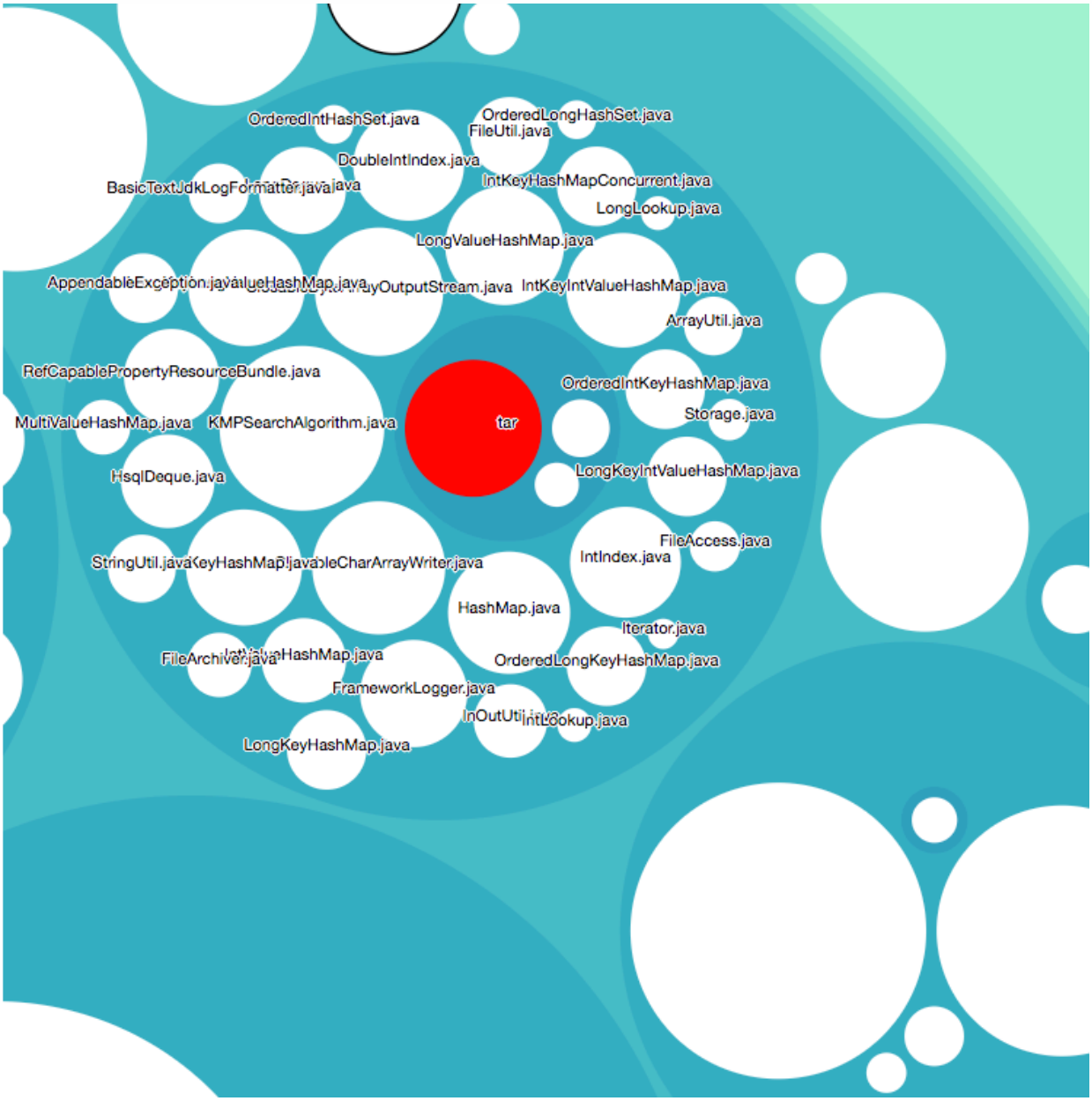


Figure: Clone enclose diagram to show clone overview (Step 2, after zoom to lib folder)



The visualization above tries to imitate this approach by creating a zoomable map of code clone criminals.

We used an enclosure diagram in order to handle large software systems. This diagram is based on a geometric layout algorithm called circle packing. Each circle represents a module or code class of the system. The more duplicate lines a module has the larger the circle. We can immediately see the files with the largest clones and in which folders they are located.

Properties of the enclose diagram

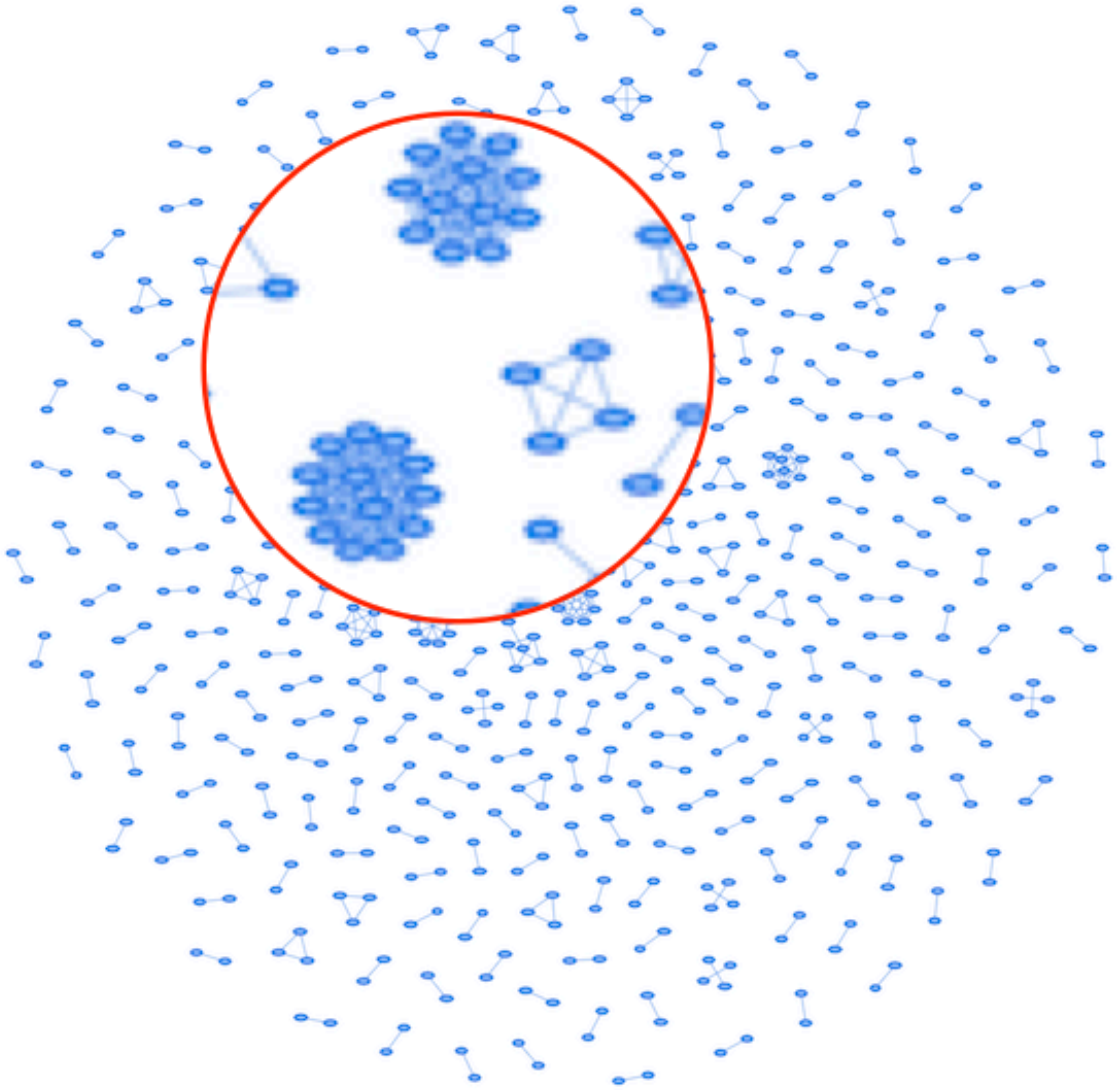
- *Pro*: Even very large projects can be visualized. Very small circles can be hidden before a zoom step to increase performance.
- *Pro* : A maintainer can focus on one specific part of the project (see step 2) or get a quick overview (step 1)
- *Pro* : For future work one could highlight the bubbles or use diffrent shapes to show relations or other properties like lines of code.

- *Contra*: It is relativly hard to implement the packed circles chart and other visualizations like the clone graph (see below) or the "friendship wheel" express the relations between duplicates better.

Clone graphs

The enclosure diagram is good to get a quick overview over the project, but it does not tell anything about how clones are connected inside the system.

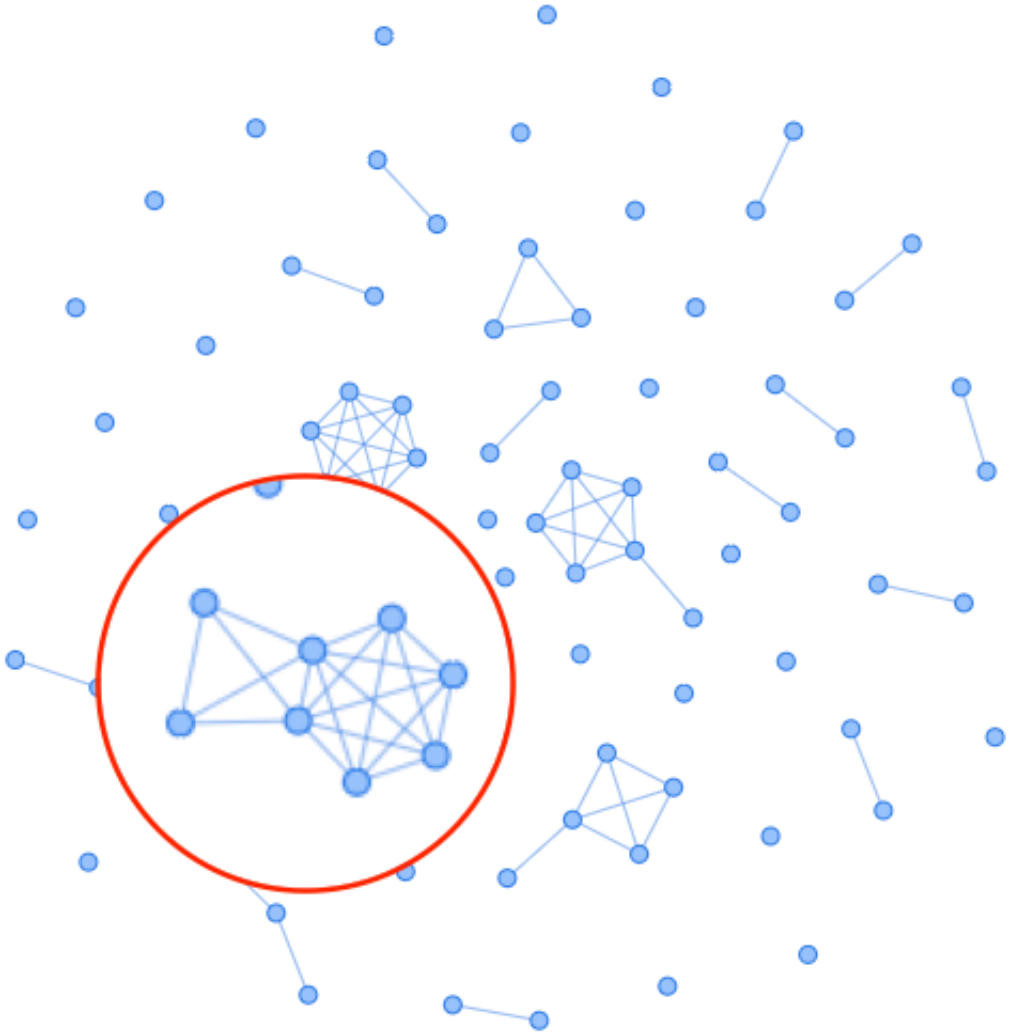
Figure: Graph of clone classes (approach 1)



To show the maintainer how the clone classes in his project are structured we created the interactive diagram above.

In the first approach, we used the clones as nodes and their connections as edges. If there are two clones of type A in one file, they will be displayed as two connected nodes. This view nicely shows that clone classes can be seen as clusters that are not interconnected. A behavior that we used to detect them (see the pseudo algorithm in "clone classes" section).

Figure: Graph of clone files (approach 2)



The second approach is more useful for a maintainer. In this view, the files are represented as nodes and they are connected by an edge as soon as one duplicate is found in both files. A highlighted area shows that these clusters are not fully interconnected, unlike in the first approach where the clones were fully transitive. If `class A` and `class B` share a clone, and `class B` and `class C` share a clone, `class A` and `class C` are not connected (not transitive).

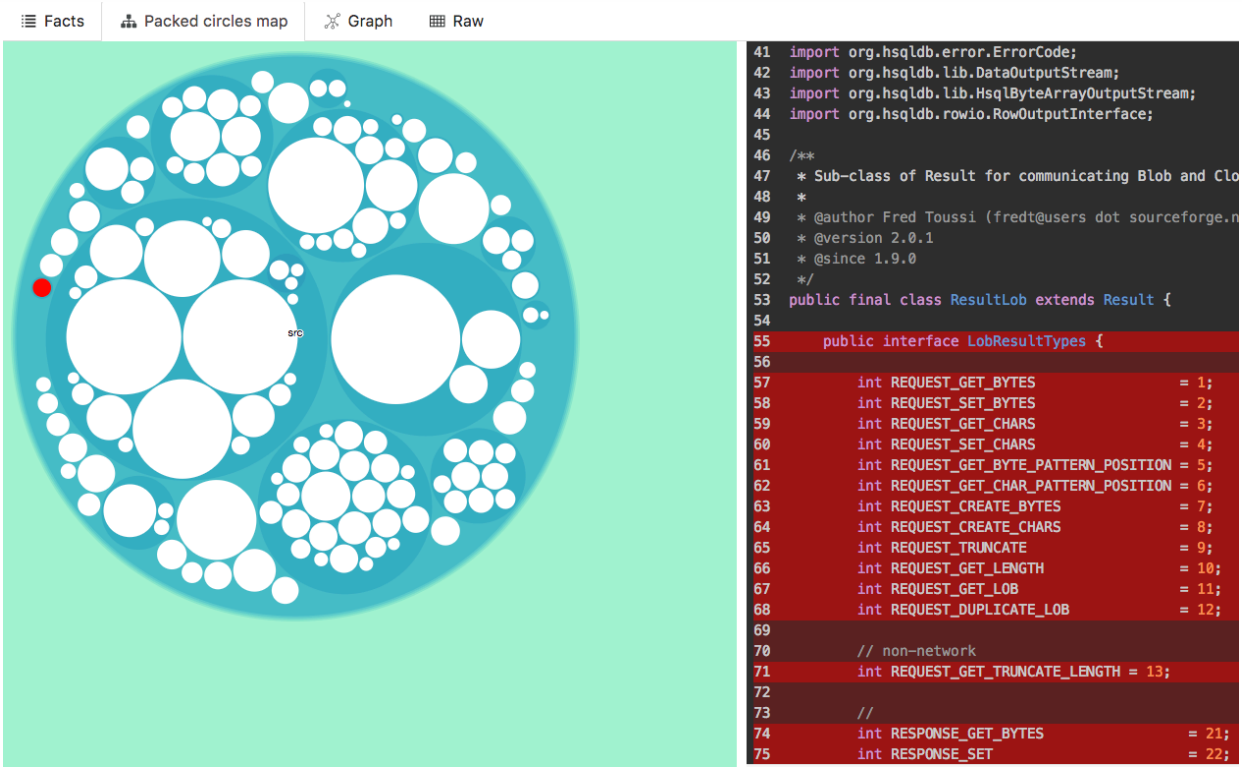
What a maintainer can learn from this view:

- There are many files that have clones in themselves - which should be an easy fix
- Most of the clusters are fully interconnected, which shows that there is some behavior that all of these files could have in common.
- Removing the largest 4 clone classes could reduce the number of files that contain clones drastically.

Code preview

Instead of showing clones only we decided to show the whole code of a selected class next to the enclosure diagram.

Figure: Code preview panel next to enclosure diagram

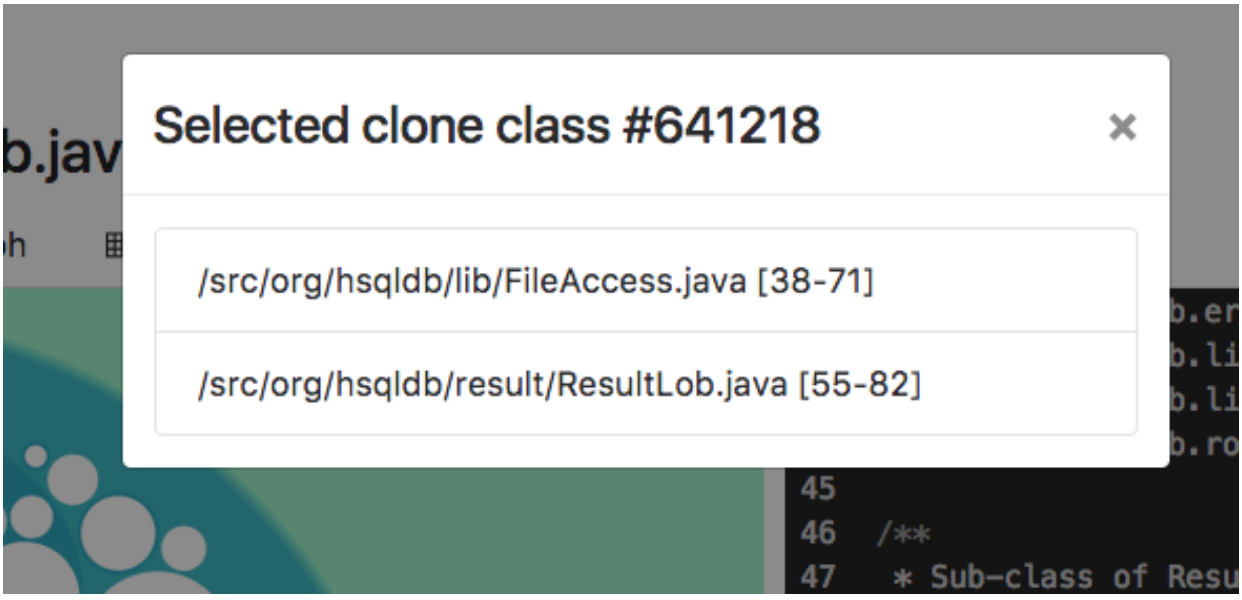


There are three diffrent levels of highlighting:

- Regular: No clone was detected in this area. The code is highlighted like the contents of an IDE.
- Dark red: Inside a clone, but not counted as a line of code (empty line or comment)
- Bright red: Lines that are inside a clone and contain actual code.

By clicking on an area which is highlighted red a maintainer can see which other parts of his codebase contain the same clone (see figure below).

Figure: Modal to show related clones

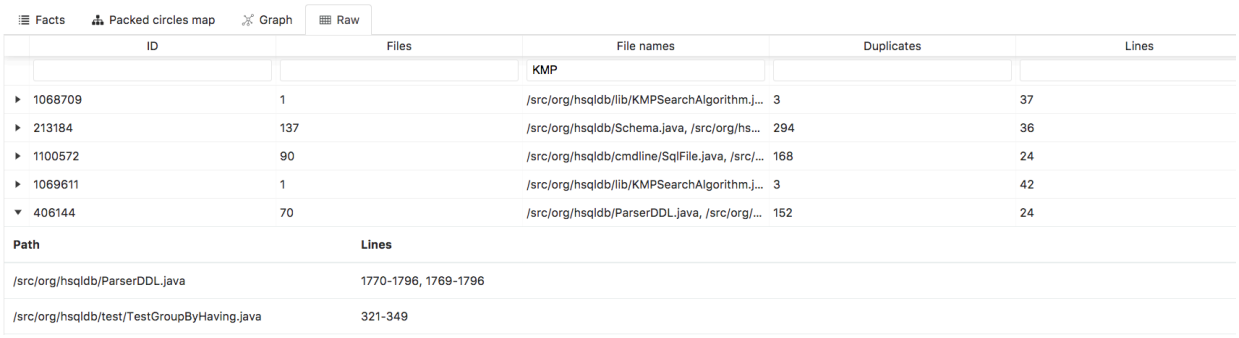


Tabular information

To give the maintainer a proof of our numbers we included a searchable raw data table.

In the example below only files that contain the keyword `KMP` are listed. A maintainer can then inspect the clone class by opening the arrow to the left.

Figure: Interactive raw data table



If a maintainer does not want to explore raw data but get a quick overview of the system we added a "facts" table that shows the worst hotspots and other metrics:

Figure: Quick facts table

Project name	smallsql
Project lines of code	24050
Duplicate lines	375 (1.56% of total project)
Clone classes	23
Largest clone class by nodes	#105869 (nodes: 6, lines: 6, lines total: 36)
Largest clone class by lines	#186775 (nodes: 2, lines: 11, lines total: 22)
Largest clone class by lines total	#105869 (nodes: 6, lines: 6, lines total: 36)
Files in project	186
Files with clones	30
File with most cloned lines	/src/smallsql/junit/TestTransactions.java (duplicate lines: 36, clones: 6)
File with most distinct clones	/src/smallsql/junit/TestTransactions.java (duplicate lines: 36, clones: 6)

Testing

We provided a test suit for this projects. Tests can be executed when all test files are imported:

```
1 import series2::Tests::CloneDetectionTest;
2 import series2::Tests::LinesOfCodeTest;
3 import series2::Tests::CloneClassesTest;
```

Clone detection

We created a framework that makes testing easy. You can specify in the test file what project has to be tested. The following tests are supported:

```
//getClonesType accepts a class that has to be analyzed, and the clone type
//It returns the duplicate lines
set[int] lines = getClonesType("ifChange", 1); //Test the class ifChange for type 1 clone
//doT3CloneDetect accepts a class that has to be analyzed, the amount of minimal sub nodes, and a similarity percentage.
//It returns the duplicate lines
set[int] lines = doT3CloneDetect("codeSwap", 16, 95.0); //Test the class codeSwap, test nodes with minimal 16 sub nodes and a similarity of 95%
//doT3CloneDetect accepts a class that has to be analyzed, the amount of minimal sub nodes, and the clone type
set[int] lines = getClonesTypeWithV("codeSwap", 1, 16); //Test the class codeSwap, detect type 1 clones, and each node that is compared needs to have 16 sub nodes or more.
//There are multiple ways of testing the output. You can check if the lines are equal to a set of lines, or you can check how many lines are found. For example:
size(lines) == 6;
```

We execute 25 different test cases:

```
Running tests for series2::Tests::CloneDetectionTest
Test report for series2::Tests::CloneDetectionTest

        all 25/25 tests succeeded
```

Lines of code

As mentioned earlier, we have test the lines of code functionality. We created a framework that makes testing easy. You can specify in the test file what project has to be tested. The following tests are supported:

```
//doT3CloneDetectionLoc accepts a class that has to be analyzed, the amount of minimal sub nodes, and a similarity percentage
//It returnes a file overview with lines that are counted as duplicates
cloneDetectionResult result = doT3CloneDetectionLoc("dupTest", 1, 90.0);
//Test the class dupTest, test nodes with minimal 1 sub node and a similarity of 90%
//You can caludate the output like this:
result.duplicateLines[fn] == toSet([4,5,6,8,10,11,21,22,25,31]);
//With this you check that [4,5,6,8,10,11,21,22,25,31] are the found duplicate lines.
```

We execute 2 different test cases:

```
Running tests for series2::Tests::LinesOfCodeTest
[4,5,6,8,10,11,21,22,25,31] success

Test report for series2::Tests::LinesOfCodeTest

        all 2/2 tests succeeded
```

For these test we choose to add multiple side cases per test instead of a large amount of different tests.

Design decision

The following list illustrates our most important and influencial design decisions. We tried to base them on the initial stakeholder requirements and explain them in the sections above:

- **Make report configurable**: As you can see in the `Parameters` section, we included a set of options that a user of the tool can use to change the report to his own liking. There are many variables that could make it hard to reproduce results, so we decided to use sensible default values based on literature (see parameters section).
- **Loading whole code files on demand**: Instead of showing snippets of clones with little to no context we decided to load and show whole code classes on demand. Other duplication tools only show the part of the code which is duplicate, but with our approach, a maintainer can get a quick overview of the whole class and can maybe already see the reason for the clone.
- **Using AST over text based approach**: Our text-based approach from series1 was already able to detect Type1 clones and deliver clone classes. However, we used an AST based approach in this assignment to create Type2+ clones. Without the AST it is hard to normalize the vocabulary of each programming language. Java has a set of well-defined keywords, but for other languages, we would need to add a tokenizer to understand in which context we have to normalize token values.
- **Including multiple diagrams**: We decided to use multiple diagrams to show different aspects of clones: their distribution across the project (enclosure diagram) and their relation towards each other / how clone classes are built (graph diagram).
- **Using JavaScript for visualisation**: Rascal has a powerful visualization framework, but a

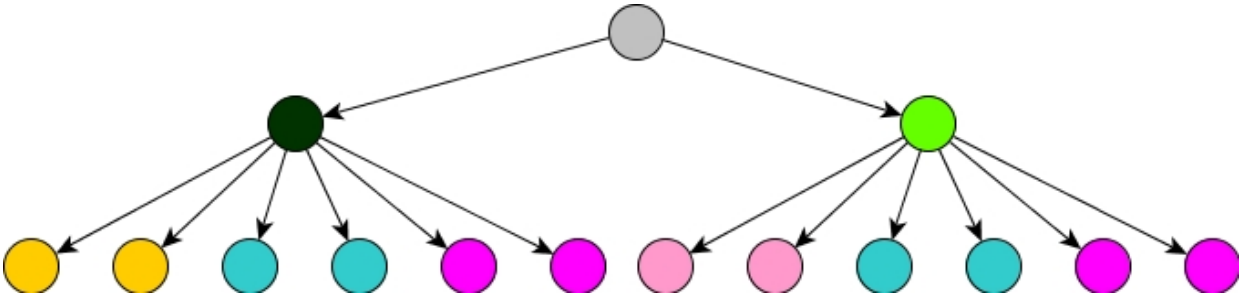
rather small community and libraries that one can build on. We decided to use JavaScript together with React, D3.js, and vis.js to quickly prototype different diagrams. By saving time on basic problematics like layout algorithms we invested more time in usability and the integration of the diagrams in the dashboard (e.g. making enclosure and graph diagram zoomable and selectable).

Future work

Comparing neighbouring nodes

We cannot compare nodes, or groups of nodes that are on the same level to other nodes or group of nodes. For example, the tree below where nodes with a similar color are the same:

Figure: Neighbouring nodes with clones



- The two large green trees are not seen as a type one or two clone, this because the sub nodes are different, what is correct.
- The two yellow sub nodes of the first green tree are not seen as duplicates. This because our tool only compares whole nodes.
- The two yellow sub nodes of the first green tree are also not compared with any combination of sequential nodes of the second green sub tree. This because our approach does not support comparing sequential nodes with other nodes in the system. It also does not support comparing these groups to groups under the same parent node.

We did not add support for this because it adds a lot of complexity. The clone detection will also take a lot more times because of the high amount of additional checks that have to be preformed. Large clones are unlikely to be affected because in real-live projects, they normally are not on a single layer in the AST. For this project, and our goal is to find large clones; small codes have a smaller impact to the system.

Show relationship in enclosure diagram

Instead of using two diagrams, enclosure, and graph, to give an overview and the relations we could unit them in a single view. One approach could be a connected bubble diagram in which bubbles are connected with lines to represent the number of shared clones, the stronger the line, the more duplicates they share.

Furthermore, we could add color codes or shapes to visualize multiple properties like lines of code, complexity or analyzability at once.

Literature

[Tornhill, 2015] Adam Tornhill. (2015). Your Code as a Crime Scene. https://doi.org/10.1017/CBO9781107415324.004

[Koschke, 2008] R. Koschke. (2008). Identifying and Removing Software Clones.

[Storey, Fracchia, Müller, 1999] Storey, M. A., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, *44*(3), 171-185.

[Baxter, Yahin, Moura, Sant'Anna, Bier, 1998] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998, November). Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on* (pp. 368-377). IEEE.

[Khatoon, Singh, Shukla, 2012] Khatoon, T., Singh, P., & Shukla, S. (2012). Abstract Syntax Tree Based Clone Detection for Java Projects. *IOSR Journal of Engineering*, *2*(12).

[Roy, Cordy, 2007] Roy, C. K., & Cordy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR*, *541*(115), 64-68.