Series 2 Report: Clone Detection and Visualization

Roman Körnig
Jakub Stanislaw Kaşıkcı

Contents

1 Introduction

In this assignment, we decided to implement the front-end route, focusing on creating visualizations to represent code duplication patterns. Our work involved detecting Type 1 and Type 2 clones using abstract syntax trees (ASTs).

This report outlines the design and implementation of our solution, presents the results, and evaluates its effectiveness.

2 Clone Detection

Our clone detection algorithm is inspired by the approach proposed by Baxter *et al.* [1], which utilizes abstract syntax trees (ASTs) to identify code duplication. The algorithm is designed to detect both Type 1 and Type 2 clones:

- **Type 1 clones** are exact duplicates of code, differing only in formatting or whitespace.
- **Type 2 clones** include modifications such as variable renaming or changes to literals, while retaining the same structural logic.

The following subsections describe the key steps in our clone detection process, including the construction of ASTs, subtree hashing, filtering by mass threshold, and the creation and refinement of clone classes.

2.1 AST Construction

The first step in our clone detection process involves extracting Abstract Syntax Trees (ASTs) from the source code. ASTs provide a structural representation of the code by abstracting away irrelevant details such as formatting and comments. This allows us to focus on identifying structural similarities between code fragments.

We use the Rascal M3 framework to extract ASTs for all files in the target project. The following ‘getASTs‘ function demonstrates how the ASTs are generated:

```
list[Declaration] getASTs(loc projectLocation) {
    M3 model = createM3FromMavenProject(projectLocation);
    list[Declaration] asts = [createAstFromFile(f, true)
        | f <- files(model.containment), isCompilationUnit(f)];
    return asts;
}
```

Code 1: Extracting ASTs from a project.

The ‘getASTs‘ function takes the project location as input and performs the following steps:

- Creates an M3 model of the project using `createM3FromMavenProject`, which captures the structure and dependencies of the project.
- Iterates through all files in the project's containment structure.
- Filters the files to include only valid Java compilation units using `isCompilationUnit`.
- Generates an AST for each compilation unit using `createAstFromFile`.

The generated ASTs form the foundation for detecting code clones by providing detailed structural representations of the source code.

2.2 Hash Bucket Creation

After constructing the Abstract Syntax Trees (ASTs), the next step in our clone detection process involves creating hash buckets. These buckets group structurally similar sub-trees by hashing their normalized representations. This step is crucial for efficiently identifying potential clones.

The process consists of two main parts: **normalization** (for detecting Type 2 clones) and **sub-tree hashing**. Each part is explained below.

2.2.1 Normalization for Type 2 Clones

To detect Type 2 clones, normalization is applied to the ASTs to replace variable identifiers, literals, and other non-structural attributes with placeholders. This ensures that minor differences, such as variable renaming or value changes, do not affect the hash.

Normalization is performed by visiting each node in the AST and replacing elements as follows:

```
if (type2) {
    ast = visit(ast) {
        case i:\id(_): {
            insert id("", src=i.src);
        }
        case n:\number(_): {
            insert number("", src=n.src);
        }
        case b:\booleanLiteral(_): {
            insert booleanLiteral("", src=b.src);
        }
        case s:\stringLiteral(_): {
            insert stringLiteral("", src=s.src);
        }
        case c:\characterLiteral(_): {
            insert characterLiteral("", src=c.src);
        }
        case t:\textBlock(_): {
            insert textBlock("", src=t.src);
        }
    }
}
```

Code 2: Normalizing AST for Type 2 clones.

This code snippet demonstrates how identifiers (*id*), numbers (*number*), booleans (*booleanLiteral*), string literals (*stringLiteral*), character literals (*characterLiteral*), and text blocks (*textBlock*) are replaced with generic placeholders. This normalization step ensures that only the structural role of these elements is retained in the AST, while differences such as variable names or literal values are abstracted away. This is essential for detecting Type 2 clones, which may contain such variations.

2.2.2 Subtree Hashing

Once the AST is normalized, the algorithm hashes each subtree to identify structurally similar fragments. Subtrees with a "mass" greater than a predefined threshold are hashed and added to a hash bucket.

The following code demonstrates sub-tree hashing:

```
top-down visit(ast) {
    case node subtree: {
        int mass = getMass(subtree);
        if (mass > threshold) {
            str subtreeHash = hash(unsetRec(subtree, {"src", "decl", "
                typ"}));
            if (subtreeHash in hashBucket) {
                hashBucket[subtreeHash] = hashBucket[subtreeHash] + [<
                    subtree, subtree.src>];
            } else {
                hashBucket[subtreeHash] = [<subtree, subtree.src>];
            }
        }
    }
}
```

Code 3: Hashing subtrees and grouping into buckets.

- The `getMass` function calculates the "mass" of the sub-tree based on the number of lines. While Baxter *et al.* [1] define "mass" as the number of nodes within the sub-tree, we opted to use the number of lines to reduce computational complexity. This decision ensures faster calculations while still providing a reliable approximation of the sub-tree's size. The value returned by the `getMass` function is used to filter out smaller, less significant sub-trees, improving efficiency by focusing only on larger, structurally meaningful sub-trees.
- The `unsetRec` function removes attributes like `"src"`, `"decl"`, and `"typ"` to eliminate meta-data that is not relevant for clone detection. This ensures that the hashing process is based solely on the syntactical properties of the sub-tree.
- The `hash` function is used to generate a unique hash for each subtree. It applies the `md5Hash` function to the subtree's structural representation, producing a compact and unique identifier (`subtreeHash`) based on its content. The MD5 algorithm, as defined by Rivest in RFC 1321 [6], ensures that structurally identical subtrees produce the same hash value.
- If the hash (`subtreeHash`) is already present in the `hashBucket`, the current subtree and its corresponding source location (`subtree.src`) are appended to the existing list. This ensures that all subtrees sharing the same structure (and thus the same hash) are grouped together. If the hash (`subtreeHash`) is not present, a new entry is created in the `hashBucket`, initializing it with a list containing the current subtree and its source location.
- The `hashBucket` is a map where the key is the hash (`str`) of a sub-tree, and the value is a list of tuples. Each tuple consists of the sub-tree itself (`subtree`) and its source location. This structure efficiently groups structurally identical sub-trees while retaining their locations for detecting sub-clones later on.

2.3 Creating Clone Classes

After grouping structurally similar subtrees into buckets based on their hashes, the next step is to identify and organize clone classes. Clone classes consist of subtrees that share the same structural hash and appear in at least two locations, indicating potential duplicates.

The following code demonstrates the process of creating clone classes:

```
map[str, list[tuple[node, loc]]] createCloneClasses(map[str, list[tuple
    [node, loc]]] hashBucket) {
    map[str, list[tuple[node, loc]]] cloneClasses = ();
    for(hash <- hashBucket) {
        if(size(hashBucket[hash]) < 2) {
            continue;
        }
        subtrees = hashBucket[hash];
        cloneClasses[hash] = subtrees;
    }
    println("Delete clone sub clone classes");
    cloneClasses = deleteSubCloneClasses(cloneClasses);
    return cloneClasses;
}
```

Code 4: Creating Clone Classes.

- The function iterates through each `hash` in the `hashBucket`. If the list of subtrees for a hash has fewer than two elements, it is skipped, as single entries do not form clones. For hashes with at least two subtrees, the list of subtrees and their source locations is added to `cloneClasses`.
- The function then calls `deleteSubCloneClasses` to remove sub-clones—redundant subtrees that are structurally contained within larger clones. This ensures that only the most meaningful clone classes remain. The process of sub-clone removal will be explained in the next subsection. Finally, `cloneClasses` is returned as the set of structurally similar and non-redundant clone classes.

2.4 Detelting Sub-Clone Classes

As mentioned in Baxter *et al.* [1], sub-clone classes are clone groups that are strictly contained within larger clone classes. These redundant sub-clones need to be removed to ensure that the final set of clone classes is non-overlapping.

The following code demonstrates the process of identifying and deleting sub-clone classes:

```
    map[str, list[tuple[node, loc]]] deleteSubCloneClasses(map[str,
        list[tuple[node, loc]]] cloneClasses) {
        map[str, list[tuple[node, loc]]] newCloneClasses = cloneClasses
            ;
        for (cloneClassHash <- cloneClasses) {
            cloneClass = cloneClasses[cloneClassHash];
            for (subCloneClassHash <- cloneClasses) {
                subCloneClass = cloneClasses[subCloneClassHash];
                if (size(cloneClass) == size(subCloneClass) &&
                    cloneClass != subCloneClass) {
                    for (i <- [0..size(cloneClass)]) {
                        if (isStrictlyContainedIn(cloneClass[i][1],
                            subCloneClass[i][1])) {
                            newCloneClasses = delete(newCloneClasses,
                                cloneClassHash);
                            cloneClass = subCloneClass;
                        }
                    }
                }
            }
```

```
        }
        return newCloneClasses;
    }
```

Code 5: Detelting Sub-Clone Classes.

- The `deleteSubCloneClasses` function takes as input a map `cloneClasses`, where each key is a hash representing a clone class, and the value is a list of tuples. Each tuple contains a subtree node and its corresponding source location.
- The function iterates through all pairs of clone classes. For each pair, it compares the `size` of the two clone classes. Only clone classes of equal size are further analyzed to identify a sub-clone relationship. Further explanation on this is provided below.
- The `isStrictlyContainedIn` function checks whether each node in a clone class is strictly contained within the corresponding node of another clone class.
- If a sub-clone is detected, the smaller clone class (sub-clone) is removed from the map using the `delete` operation, ensuring that redundant entries are eliminated.
- The refined map `newCloneClasses` is returned, containing only the non-redundant clone classes. This ensures that the final set of clone classes is distinct and free of unnecessary sub-clones.

2.4.1 Clone Deletion of same sized Clone Classes

The decision to only compare clone classes of equal size is based on the assumption that there can be sub-clone classes that are strictly contained within larger clone classes, although have a different size. Code1 and Code2 are two clone classes of different sizes, although the smaller clone Class 2 in `Code1.java` is strictly contained within the larger clone Class 1 in `Clone1.java`. If Class 2 would be subsumed by Class 1, the algorithm would not detect the additional clone in Class 2 in `Clone2.java`.

```
// Clone1.java
public static void main(String[] args) {
    if(args.length > 0) {
        hello("Alice");
        int x=0;
        int a=1;
        int b=2;
        int w=4;
    }
}
public static void hello(String args) {
    if(args.length > 0) {
        hello("Alice");
        int x=0;
        int a=1;
        int b=2;
        int w=4;
    }
}
```

Code 6: Example of Clone Class 1

```
// Clone1.java
    if(args.length > 0) {
        hello("Alice");
        int x=0;
        int a=1;
        int b=2;
        int w=4;
    }
    if(args.length > 0) {
        hello("Alice");
        int x=0;
        int a=1;
        int b=2;
        int w=4;
    }
// Clone2.java
    if(args.length > 0) {
        hello("Alice");
        int x=0;
        int a=1;
        int b=2;
        int w=4;
    }
```

Code 7: Example of Clone Class 2

3 Visualization

Understanding complex software systems is a fundamental challenge for maintainers, particularly during tasks such as code exploration, maintenance, and comprehension. According to Storey *et al.* [7], effective visualization tools can significantly support maintainers by aiding the construction of accurate mental models of software systems. These tools are designed to address cognitive challenges, such as navigating delocalized code, identifying relationships between components, and managing large-scale systems.

In this section, we present two techniques we explored to visualize the cloning data extracted from projects and go briefly through the implementation of the technique.

3.1 Clone Visualization Using Circle Packing

To effectively visualize cloning data extracted from software projects, we explored techniques inspired by ClonePacker, a tool developed by Murakami *et al.* [5]. ClonePacker uses the Circle Packing visualization technique to present clone sets in a clear, hierarchical manner.

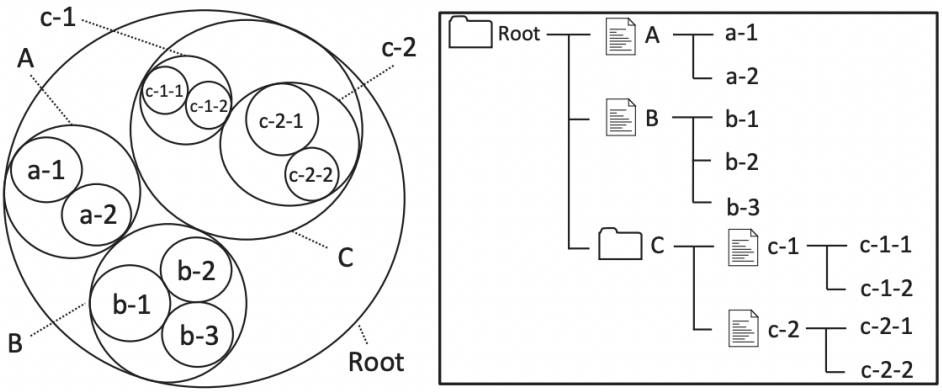

Figure 1: Circle Packing from Murakami *et al.* [5].

Circle Packing is a method for representing hierarchical data structures as nested circles. In the context of clone visualization:

- The outermost circles represent directories within the project structure.
- The intermediate circles represent files contained within these directories.
- The innermost circles represent individual methods, where the size of each circle corresponds to the lines of code (LOC) in the method.

For the Circle Packing visualization, we utilized the D3.js library [2]. The initial version effectively visualized smaller projects by representing directories, files, and methods as nested circles. However, we observed that the visualization did not scale well for larger codebases, becoming cluttered and difficult to interpret.

To address this limitation, we developed an enhanced version inspired by the Zoomable Circle Packing implementation demonstrated on ObservableHQ [3]. This version incorporates zoom-in and zoom-out features, allowing maintainers to interactively explore the project hierarchy. The following section will have detailed information about the implementation.

3.2 Implementation of Circle Packing Visualization

3.2.1 Data Structure and JSON Writing

The Circle Packing visualization is implemented using the D3.js library, combined with React.js and TypeScript. The clone has been written into two separate JSON files, one containing the pure clone class data with details of the clones in each leaf (e.g. length, src, and hash), and the other containing the clones ordered in a hierarchical structure. The hierarchical structure has the same structure as the project, with directories containing files. Each leaf has the hash of the clone itself and the hash of the clone class it belongs to.

```
{
    "name": "root",
    "children": [
        {
            "name": "src",
            "children": [
                {
                    "name": "main",
                    "children": [
                        {
                            "name": "java",
```

```
                                "children": [
                                    {
                                        "name": "Clone1.java",
                                        "children": [
                                            {
                                                "name": "Clone1.java",
                                                "hash": "hash1",
                                                "cloneClass": "cloneClass1"
                                            }
                                        ]
                                    }
                                ]
                            }
                        ]
                    }
                ]
            }
        ]
}
```

Code 8: Example of the hierarchical structure of the clone data.

```
{
    "cloneClasses": {
        "cloneClass1Hash": {
            {
                "hash": "hash1",
                "src": "Clone1.java",
                "length": 10,
            },
            {
                "hash": "hash2",
                "src": "Clone2.java",
                "length": 10,
            }
        },
        "cloneClass2Hash": {
            {
                "hash": "hash1",
                "src": "Clone1.java",
                "length": 30,
            },
            {
                "hash": "hash2",
                "src": "Clone2.java",
                "length": 30,
            },
            {
                "hash": "hash3",
                "src": "Clone3.java",
                "length": 30,
            }
        }
    }
}
```

Code 9: Example of the clone class data.

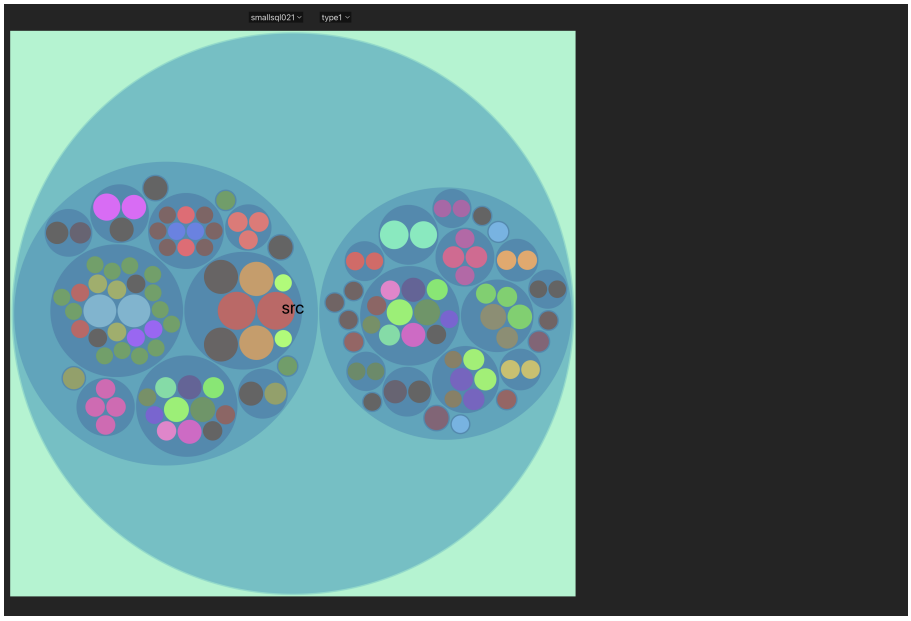

Figure 2: Circle Packing Tool initial screen

It was particularly difficult to write out the JSON files in the correct format. The hierarchical structure of the clone data was created by iterating through the clone classes and recursively adding the clones to the correct directory and file within the JSON structure. The clone class data was written out by iterating through the clone classes and adding the clones to the correct clone class.

3.2.2 Circle Packing Visualization

The Circle Packing visualization was implemented using the D3.js library, React.js and TypeScript. The visualization consists of nested circles representing directories, files, and clones. The size of each circle is proportional to the number of lines of code (LOC) in the corresponding clones. Each leaf circle is color-coded based on the clone class it belongs to, allowing maintainers to identify related clones at a glance.

The visualization allows maintainers to interactively explore the project structure by zooming in and out of the hierarchy. When a circle is clicked, the visualization zooms in on that circle, displaying the next level of the hierarchy. When a leaf circle is clicked, the view is zoomed out to the root level and the corresponding clones in the same classes are highlighted. Additionally, the detailed information of the clone class is displayed in the right of the circle packing visualization. The details include the location of the clone, the length of the clone, LOC, as well as the beginning and the end of the lines. The drop-down in the top also allows to switch between the different projects that have been analyzed and also switch between displaying only type I or a combination of type I and type II clones.

As shown in the figure below, the Circle Packing visualization provides an intuitive overview of the project structure, allowing maintainers to explore code duplication patterns and identify related clones efficiently.

3.3 Clone Visualization Using Tree Map

In addition to Circle Packing, we explored the Tree Map technique to visualize clone data directly inspired by Cyclone's Tree Map as described by Hammad *et al.* [4]. Tree Maps are particularly effective for representing hierarchical structures with weighted components. Each

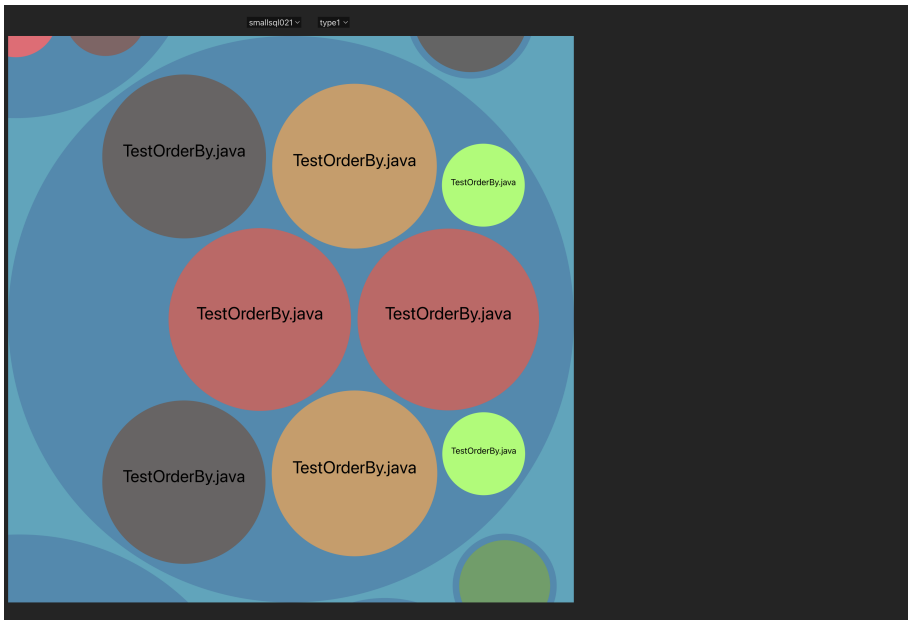

Figure 3: Circle Packing Tool after zooming into a folder. Clones with their located files are displayed.

Figure 4: Circle Packing Tool after clicking a leaf node. Clones from the same class are highlighted and details of the clones are displayed.

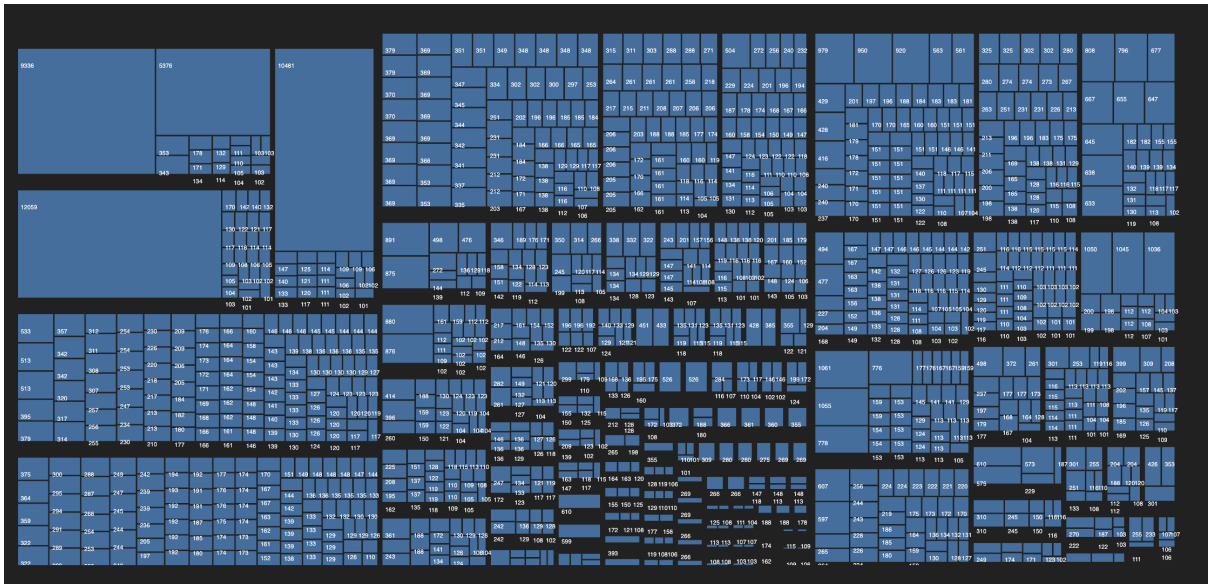

Figure 6: Early Implementation of the Tree Map Visualization

level of the hierarchy is represented as a nested rectangle, and the size and color of each rectangle can encode additional information.

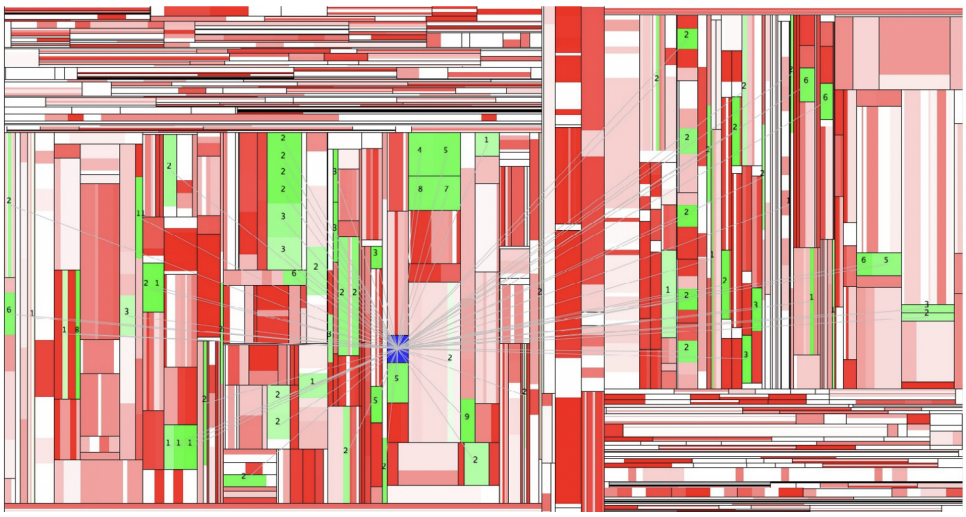

Figure 5: Cyclone's Tree Map view from Hammad *et al.* [4]

3.3.1 Difficulty in Implementation and Limits of Tree Mapping

Whereas Tree Mapping is effective for visualizing hierarchical structures, and has a high information density, it can be challenging to interpret at a glance. The dense layout of rectangles can make it difficult to distinguish relationships between components, especially in large-scale projects. To address this challenge, we explored different color-coding and linking strategies to enhance the visualization's clarity and reduce cognitive overhead. However, we found that the static nature of the Tree Map visualization limited its interactivity and navigation flexibility and scrapped the implementation in favor of the Circle Packing visualization with zooming capabilities.

4 Reflection on Clone Detection

4.1 Changes from the Original Algorithm

Major differences from the original algorithm by Baxter *et al.* [1] include the use of character count as a proxy for mass and the removal of the comparison by the similarity threshold that requires a full traversal of each AST node and the calculation of shared and different nodes. Similar to the reasoning of the removal of mass threshold, the similarity threshold was removed to simplify the algorithm and reduce computational complexity and make the algorithm adaptable for large-scale projects. Removing the threshold allowed us to run the algorithm on large projects such as hsqldb and compute a result within a reasonable time frame.

4.2 Mass Threshold Adjustment

Depending on the mass threshold, the result of the clone detection can vary significantly. A lower mass threshold will result in more fine-grained clones, while a higher mass threshold will group more code fragments together. The mass threshold is a crucial parameter that needs to be adjusted based on the project size and the desired granularity of the clone detection. Also depending on the complexity of the project, the mass threshold needs to be adjusted to avoid false positives. For example, for simple projects with few nesting of data in the code, a lower mass threshold can be used. For more complex projects with many nested data structures, a higher mass threshold is recommended to avoid false positives. An algorithm that could automatically adjust the mass threshold based on the project size and complexity would be beneficial.

5 Reflection on Visualization

5.1 Reflexion on Implementation

The Circle Packing visualization gives a clear overview of the project structure and allows maintainers to explore code duplication patterns effectively. The zooming feature enhances the visualization by providing interactive navigation and enabling maintainers to focus on specific components while maintaining context. The color-coding of clone classes helps identify related clones at a glance, and the detailed information displayed on click provides additional context for each clone.

Whereas the tool gives a good overview of the project structure and the clones, there are still many improvements that can be made. For larger scale project such as the hsqldb project, the visualization can become cluttered and difficult to interpret. On top of that, because of the high computational complexity of the rendering of the visualization, the tool can become slow and unresponsive when many clones are displayed.

To make the visualization more effective, we could implement a functionality that allows maintainers to filter the clones based on different criteria, such as clone length, clone class, or file location. This would enable maintainers to focus on specific subsets of clones and reduce visual clutter. Additionally, a display of the clone code can be implemented to allow maintainers to inspect the code directly from the visualization such as in the tool presented by Luiciano *et al.* [**?**].

5.2 Circle Packing

Storey *et al.* [7] identify cognitive design elements that visualization tools should provide to support program comprehension and reduce cognitive overhead. These elements are as follows:

- *Enhance bottom-up comprehension:* Visualizations should allow maintainers to explore code details, such as files and methods, while indicating *syntactic and semantic relationships* and reducing the effect of *delocalized plans*.
- *Enhance top-down comprehension:* Tools should provide an *overview of the system architecture* to support goal-directed comprehension at multiple abstraction levels.
- *Integration of bottom-up and top-down approaches:* Effective tools integrate both perspectives, enabling the construction of *multiple mental models* and cross-referencing between abstraction levels.
- *Facilitate navigation:* Visualizations must offer *directional* and *arbitrary navigation*, allowing maintainers to explore paths intuitively and jump between components.
- *Provide orientation cues:* Tools should track progress by highlighting the *current focus*, showing the *path leading to the focus*, and suggesting *further exploration* options.
- *Reduce disorientation:* Visualizations minimize *interface adjustment effort* and employ *effective presentation styles* to ensure clarity and consistency.

Both Circle Packing and Tree-Maps addressed the cognitive design elements outlined above. s. Below, we reflect on their effectiveness and highlight the improvements made to Circle Packing with the zooming feature:

- *Enhance bottom-up comprehension:* The Circle Packing visualization allows maintainers to explore individual methods (represented as innermost circles) and files within directories. The size of each circle reflects the lines of code (LOC), providing immediate insight into code details. The zoom-in and zoom-out feature enhances bottom-up comprehension by enabling maintainers to isolate and examine specific areas of the project without visual clutter. While the Tree Map also facilitates bottom-up exploration by representing code fragments as rectangles, the dense layout can make it harder to interpret relationships at a glance.
- *Enhance top-down comprehension:* Both visualizations offer a hierarchical representation of the system architecture. Circle Packing excels at presenting an intuitive overview: larger outer circles represent directories, while nested circles correspond to files and methods. The circular arrangement naturally conveys containment and hierarchy. In contrast, the Tree Map's rectangular layout, while precise, can appear overwhelming and visually dense in large-scale projects.
- *Integration of bottom-up and top-down approaches:* The Circle Packing visualization integrates bottom-up and top-down comprehension seamlessly through its zooming feature. Maintainers can begin with a high-level overview and progressively drill down to specific methods. The Tree Map also supports this integration but lacks interactivity in its static form, which limits exploration flexibility.
- *Facilitate navigation:* The interactive zoom functionality in Circle Packing provides smooth, *directional navigation*, allowing maintainers to focus on specific components while maintaining context. The Tree Map visualization, despite its ability to show relationships through color and links, does not inherently support arbitrary navigation or interactive zooming. Navigating dense regions can be challenging without additional features.
- *Provide orientation cues:* Circle Packing inherently highlights the current focus during zooming by enlarging the selected region while preserving the surrounding context. This

provides clear orientation cues and minimizes disorientation. The Tree Map, while color-coded to distinguish clones, does not dynamically adjust to maintainers' focus, making it less effective for tracking navigation paths.

- *Reduce disorientation:* The zoom-in and zoom-out features in Circle Packing reduce disorientation by allowing maintainers to navigate the hierarchy incrementally. This prevents information overload and ensures a clean presentation at each level. In contrast, the static Tree Map can quickly become overwhelming for large-scale systems, as it displays all components simultaneously without abstraction or interactivity.

6 Results

The clone detection analysis produced the following key observations across the analyzed projects:

- **Type 1 Clones:** Exact duplicates were identified across all projects. The results for the smallest project, *TestClone1*, which we created for testing purposes, came out as expected, reflecting controlled duplication patterns. Larger projects exhibited a higher absolute number of clones but lower relative percentages.
- **Type 2 Clones:** Near-duplicate code fragments were far more prevalent than exact clones, contributing significantly to the overall duplication. This highlights areas where minor variations, such as formatting or variable renaming, could be addressed.
- **Project Size:** Larger projects, such as *hsqldb-2.3.1*, contained the most extensive clone classes and duplicated lines, with the largest clone spanning 401 lines. While absolute duplication was high, the relative percentage remained moderate. In contrast, the smaller test project showed a higher percentage of duplication, as expected for its design.
- **Biggest Clones:** Notable clones of significant length were detected in larger projects. For example, the largest clone (401 LOC) in *hsqldb-2.3.1* highlights areas with substantial redundancy that could benefit from refactoring.
- **Mass Threshold:** The analysis applied a mass threshold of 50 lines to focus on structurally significant clones, ensuring that small and insignificant fragments were excluded.

Appendix

A Clone Detection Results

A.1 TestClone1

Clone Detection Report for Project: **TestClone1**

```
----------------------------------------
Total Lines of Code (LOC): 58
Mass Threshold for Clones: 50

Type 1 Clones:
----------------------------------------
Total Clone Classes: 1
Total Clones: 2
Total Duplicated Lines: 18
Percentage of Duplicated Lines: 31.03448276%
Biggest Clone (in LOC): 9
Location: Clone1.java: lines 15 - 23 (9 LOC)
Example Clones:
  - Clone1.java: lines 15 - 23 (9 LOC)
  - Clone1.java: lines 29 - 37 (9 LOC)

Type 2 Clones:
----------------------------------------
Total Clone Classes: 2
Total Clones: 5
```

```
Total Duplicated Lines: 45
Percentage of Duplicated Lines: 77.58620690%
Biggest Clone (in LOC): 9
Location: Example2.java: lines 15 - 23 (9 LOC)
Example Clones:
  - Example2.java: lines 15 - 23 (9 LOC)
  - Clone1.java: lines 15 - 23 (9 LOC)
  - Clone1.java: lines 29 - 37 (9 LOC)
```

A.2 smallsql0.21_src

Clone Detection Report for Project: **smallsql0.21_src**

```
----------------------------------------
Total Lines of Code (LOC): 24016
Mass Threshold for Clones: 50

Type 1 Clones:
----------------------------------------
Total Clone Classes: 51
Total Clones: 126
Total Duplicated Lines: 750
Percentage of Duplicated Lines: 3.122918055%
Biggest Clone (in LOC): 29
Location: TestOrderBy.java: lines 698 - 726 (29 LOC)
Example Clones:
  - Utils.java: lines 205 - 212 (8 LOC)
  - BasicTestCase.java: lines 102 - 109 (8 LOC)
  - TestOrderBy.java: lines 329 - 352 (24 LOC)

Type 2 Clones:
----------------------------------------
Total Clone Classes: 164
Total Clones: 557
Total Duplicated Lines: 3928
Percentage of Duplicated Lines: 16.35576282%
Biggest Clone (in LOC): 151
Location: Language_it.java: lines 35 - 185 (151 LOC)
Example Clones:
  - RowSource.java: lines 151 - 155 (5 LOC)
  - RowSource.java: lines 157 - 161 (5 LOC)
  - RowSource.java: lines 188 - 192 (5 LOC)
```

A.3 hsqldb-2.3.1

Clone Detection Report for Project: **hsqldb-2.3.1**

```
----------------------------------------
Total Lines of Code (LOC): 172150
Mass Threshold for Clones: 50

Type 1 Clones:
```

```
----------------------------------------
Total Clone Classes: 805
Total Clones: 2025
Total Duplicated Lines: 16917
Percentage of Duplicated Lines: 9.826895150%
Biggest Clone (in LOC): 73
Location: JDBCBlobFile.java: lines 516 - 588 (73 LOC)
Example Clones:
  - TestStoredProcedure.java: lines 447 - 448 (2 LOC)
  - TestStoredProcedure.java: lines 467 - 468 (2 LOC)
  - StatementSchema.java: lines 1307 - 1310 (4 LOC)

Type 2 Clones:
----------------------------------------
Total Clone Classes: 1921
Total Clones: 6778
Total Duplicated Lines: 64452
Percentage of Duplicated Lines: 37.43944235%
Biggest Clone (in LOC): 401
Location: ResultConstants.java: lines 258 - 658 (401 LOC)
Example Clones:
  - ClosableCharArrayWriter.java: lines 276 - 291 (16 LOC)
  - ClosableByteArrayOutputStream.java: lines 239 - 258 (20 LOC)
  - DatabaseManager.java: lines 504 - 507 (4 LOC)
```

References

[1] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998. doi: 10.1109/ICSM.1998.738528.

[2] Mike Bostock. D3.js - data-driven documents. URL `http://d3js.org/`.

[3] Mike Bostock. Zoomable circle packing, 2019. URL `https://observablehq.com/@d3/zoomable-circle-packing`. Accessed: 2024-06-15.

[4] Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. A systematic mapping study of clone visualization. *Computer Science Review*, 37:100266, 2020. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2020.100266. URL `https://www.sciencedirect.com/science/article/pii/S1574013719302679`.

[5] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. Clonepacker: A tool for clone set visualization. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 474–478, 04 2015. doi: 10.1109/SANER.2015.7081859.

[6] Ronald Rivest. The MD5 Message-Digest Algorithm. RFC 1321, 1992. URL `https://www.rfc-editor.org/rfc/rfc1321`.

[7] 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: https://doi.org/10.1016/S0164-1212(98)10055-9. URL `https://www.sciencedirect.com/science/article/pii/S0164121298100559`.