Interactive Visualizations for Software Duplication

Master Thesis

Jonas Richner

from

Gränichen AG, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

January 2021

Prof. Dr. Oscar Nierstrasz

Dr. Nataliia Stulova

Software Composition Group

Institut für Informatik und angewandte Mathematik

University of Bern, Switzerland


b
UNIVERSITÄT
BERN


UNIVERSITÉ DE
NEUCHÂTEL


UNI
FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

In large software systems usually about 5%-20% of the code is duplicated. Duplicated code can increase maintenance costs because it has to be maintained in multiple locations. There is a significant amount of research on visualizing software duplication to help reduce these costs. But in practice mostly basic visualizations are used and the more advanced visualizations proposed by researchers are not adopted by the software industry. We believe the reason for this is that visualizations from academic research rely on single stand-alone views that only support simple analysis tasks. To support more complex tasks, we propose a set of connected multi-view visualizations for inspecting software duplication. We follow the systematic approach of Bret Victor for building interactive visualizations to gain insight into a system. Results from our user study indicate that our prototype is easy to use in various clone analysis tasks, and helps users reason about the code at multiple levels of abstraction.

Contents

1 Introduction

Code clones are fragments of code that are identical or very similar and occur multiple times in the codebase. They are usually introduced when developers copy existing code to implement new functionality. This duplicated code can lead to increased maintenance costs, because the same code has to be maintained in multiple locations.

In large software systems usually about 5%-20% of the code is duplicated [19]. There is a significant amount of research on both the good and bad effects of software duplication. Some duplicates may be less expensive to maintain than a complex generic solution [13], whereas other duplicates can increase the number of bugs as well as the time it takes to implement new features [15].

The high maintenance cost of code duplication drives the development of tools that help to manage duplicates. Given the complex nature of code duplication, where some duplicates are worthwhile to remove while replacing other duplicates with a generic solution may lead to increased maintenance costs, visualizations can help developers get a detailed understanding of the duplication in their codebase. A comprehensive mental model of code duplication allows developers to make informed decisions on which duplicates are worthwhile to remove.

There are a number of studies that propose duplication specific code visualization tools, but they often rely on single stand-alone views that only support simple analysis tasks, such as finding which files contain the most duplicated code. For more complex tasks, such as figuring out why cloning is prevalent in the codebase, we believe that users need to be able to freely explore the duplication in the code at different levels of abstraction that help them tie their findings together. To support more complex tasks, we propose a set of interconnected interactive visualizations for inspecting software duplication. They make it easy to freely explore the system by stepping between the concrete source code and higher levels of abstraction. This allows users to see high-level patterns by taking a bird's-eye view and to find the explanation for those patterns by inspecting the concrete instances of duplicated code.

The rest of the thesis is structured as follows: chapter 2 gives an overview of related work and how our contribution fits in, chapter 3 outlines the objectives, chapter 4 describes the implementation, chapter 5

describes the design of the tool, and in chapter 6 we validate the tool through user studies. We discuss areas for future research in chapter 7, before we conclude with chapter 8.

2

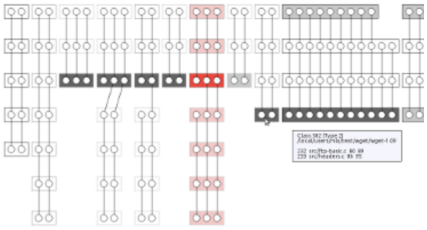
Related Work

The first paper that mentions clone visualization was published in 1992 by Brenda Baker [2]. In the paper, she describes the usage of dotplots to visualize duplication patterns. Since then 40 papers have been published with clone visualization as one of their primary focus points. A systematic mapping study on clone visualization by Hammad *et al.* finds that these 40 studies present the following visualizations as research prototypes [8]:

- **Standard visualizations:** These are classical data visualizations such as pie charts, bar charts, and line charts. For example, *VisCad* uses pie charts to show clone rates in different subsystems of a codebase [1] and *Cyclone* uses line charts to visualize the lifetime of clone families [9].
- **Textual visualizations:** These are visualizations that show the characteristics of the code at the level of detail of the source code text. For example, Sano *et al.* try to understand why a duplicate occurs using a tag cloud that uses identifier names of duplicated sections [20], as shown in Figure 2.1(a). Forbes *et al.* use vertical bars beside the code to highlight duplicated sections. They color the bars based on the impact factor, which is the product of the number of clone occurrences and their similarity [7], as shown in Figure 2.1(b).

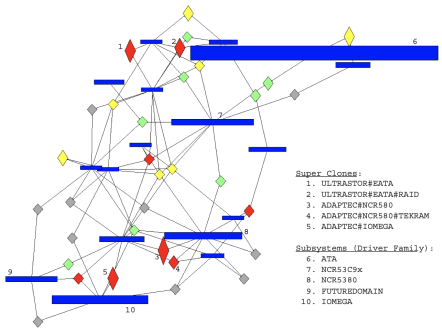
(a) Tag Cloud visualization by Sano *et al.*. (b) Vertical bars visualizations by Forbes *et al.*.

- **Temporal data visualizations:** These are visualizations where the data is organized along a timeline. For example, the clone evolution view of *Cyclone* can be used to analyze how clones evolved [9], as shown in Figure 2.1(c).



(c) Clone evolution view by Harder *et al.*.

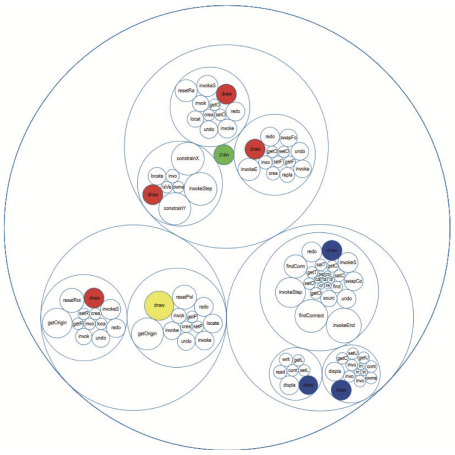
- **Node link visualizations:** These are visualizations where nodes represent entities and the edge between them represent their relations. For example, Jiang *et al.* show duplication across subsystems and within subsystems using a node-link diagram [12], as shown in Figure 2.1(d).



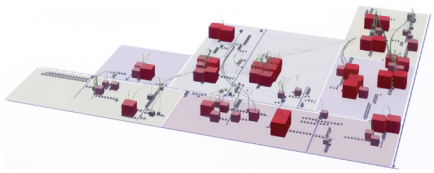
(d) Duplication across subsystems graph by Jiang *et al.*.

- **Hierarchical visualizations:** These are visualizations that focus on showing hierarchical data

structures. For example, Murakami *et al.* use circle packing to show where duplicates occur in the directory hierarchy [18], as shown in Figure 2.1(e). Steinbrückner and Lewerentz propose a code city visualization where the hierarchical relationships are depicted as branching streets [21], as shown in Figure 2.1(f).

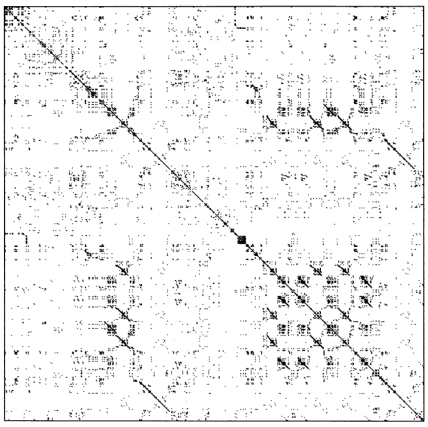


(e) Circle packing graph by Murakami *et al.*.

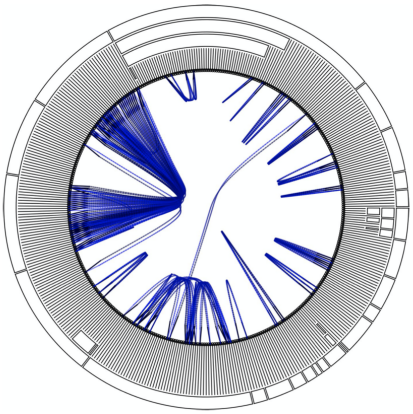


(f) EvoStreets by Steinbrückner and Lewerentz.

- **Other visualizations:** There are also other visualizations that do not fit the categories mentioned above. For example, Church *et al.* use dotplots to visualize complex duplication patterns [4], as shown in Figure 2.1(g). Hauptman *et al.* use hierarchical edge bundles in a circular layout to show how much code is duplicated between different parts of the system [11], as shown in Figure 2.1(h).



(g) Dotplot graph by Church *et al.*.



(h) Hierarchical edge bundling graph by Hauptman *et al.*.

Hammad *et al.* also investigated visualizations in commercial tools such as *Atomiq*, *Axivion Bauhaus Suite*, *Clone Doctor* by *Semantic Designs*, *JetBrain's dupFinder*, *Microsoft Visual Studio*, *Pattern Insight*, *Solid Source*, *SonarQube*, and *Teamscale*. They find that these tools use only basic types of clone visualizations, whereas most of the more advanced visualizations from academic research are not used by the software

industry. This suggests that there may be problems with these academic visualizations that hinder their adoption.

Mondal *et al.* recognized that a major problem with existing duplication visualizations is that they often only support simple analysis tasks. Their solution to allow for more complex analysis tasks consists of combining multiple different visualizations to balance the strengths and weaknesses of each individual visualization [17], as shown in Figure 2.1.

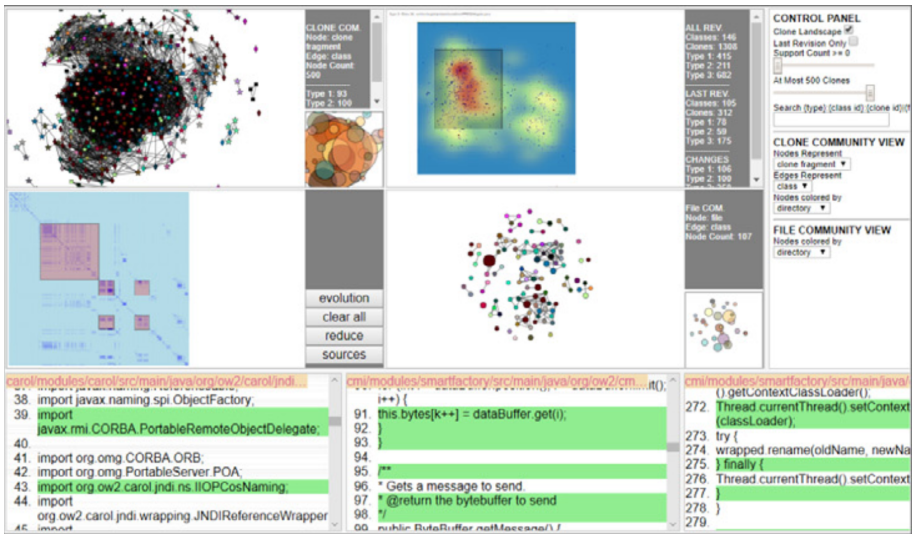


Figure 2.1: CloneWorld by Mondal *et al.*.

We agree with Mondal *et al.* that the way towards making visualizations of duplication more useful is to combine multiple visualizations that offer different perspectives. We improve on the approach of Mondal *et al.* in two ways:

- We construct visualizations that make use of existing structures for organizing code, such as files and directories, rather than more abstract node-link diagrams that break people's existing knowledge of how the code is organized. We significantly improve the *SeeSoft* graph for visualizing duplication patterns between files [6]. We also make use of and improve the icicle chart, which is usually used for performance monitoring and has not previously been used for visualizing software duplication.
- We develop new ways of connecting visualizations together that enable quick navigation across multiple levels of abstraction and allow users to reason across multiple dimensions.

3

Objectives

The objective of the tool is to provide insight into code duplication to help reduce maintenance costs associated with code clones. Although the objective is clear, it is not specific enough to be actionable. To decide which features should be supported by our tool, we turn to questions about duplication that arise in practice. We use questions from our own experience when developing and analyzing software, as well as questions from a literature survey of Basit *et al.* [3]. A few examples of questions that can arise in practice are the following:

- Where are the clones located in the system directory structure?
- What is the reason for the clones?
- What is the benefit of refactoring a clone?
- How can future code duplication be prevented?
- Where in the codebase is it most cost-effective to remove duplication?
- Which duplicates occur most often?

Obviously, any tool can help to answer these questions as long as it offers a view of the code. What matters is how hard it is to answer these questions with the tool, and how well it assists developers in getting a mental model of the duplication in the codebase that helps them tie their findings together to form a complete picture. Our tool addresses these needs by offering a multi-perspective view at different levels of abstraction so that users can answer a variety of questions. The views are linked together so that users can perform more complex analysis tasks that require reasoning across multiple dimensions.

4 Implementation

4.1 Technologies

We use a set of standard technologies to build our tool. We use *Java* in the backend because it is a common language with a large ecosystem of libraries and frameworks. It is also cross-platform and performant. We use *Spring* to build a *REST* API in the backend. In the frontend, we use *Javascript*, *HTML*, and *CSS*. These are all standard web technologies with large ecosystems of libraries to support them. We also use *d3*, which is the most common library that is used for building custom data visualizations in *Javascript*.

4.2 Architecture

Our tool consists of a backend and a frontend that both run on *HTTP* servers, as seen in Figure 4.1. The backend is responsible for scoping, code cleaning, duplicate detection and persistence. The frontend serves the *Javascript*, *HTML*, and *CSS* code to the web browser, which sends *REST* requests to the backend. For example, when the user clicks the button to run the analysis, the browser sends a *REST* request to run the analysis to the backend. When the backend is done with the analysis it returns the data to the browser. The backend saves duplication data and scope configuration options directly to the file system of the host computer.

4.2.1 Backend

The main responsibility of the backend is to detect code clones.

There are four different types of code clones [19]:

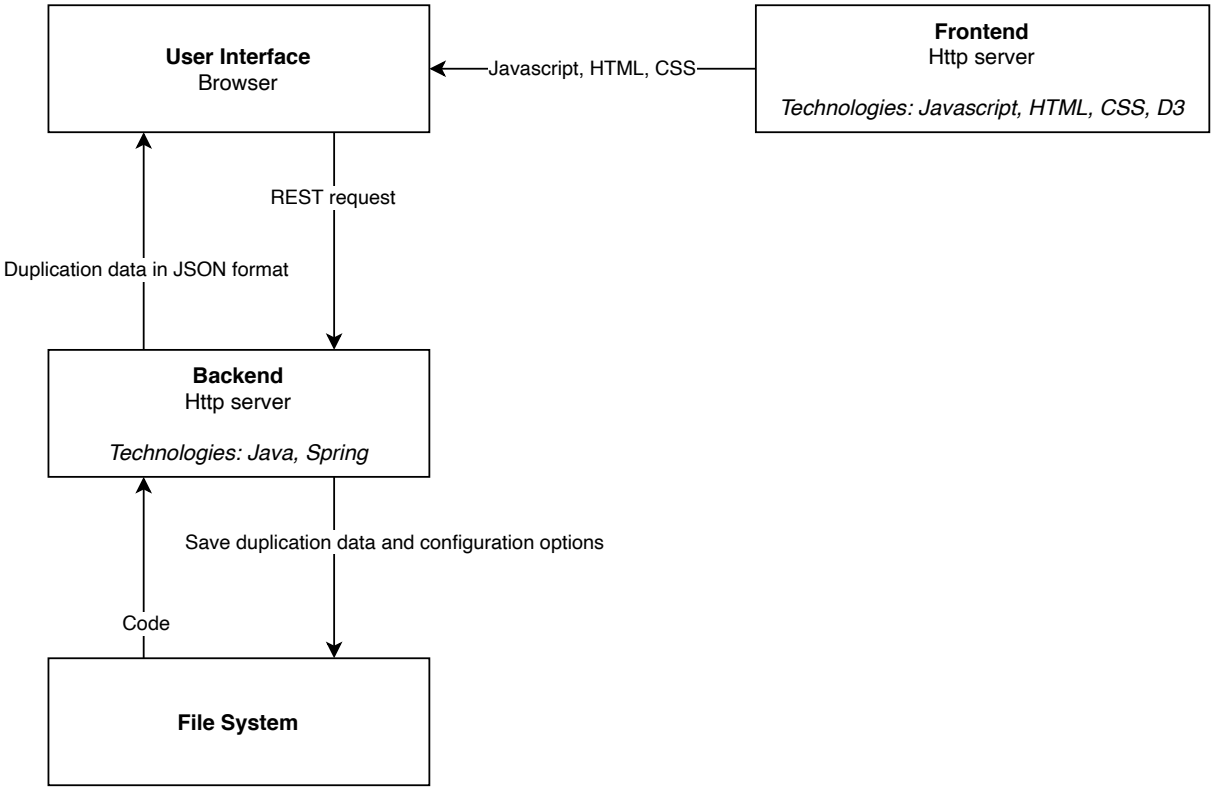


Figure 4.1: Shows the high-level system architecture.

- **Type I:** Syntactically identical code fragments except for differences in white space, layout, and comments.
- **Type II:** Syntactically identical code fragments except for variations in identifiers, literals, types, white space, layout, and comments.
- **Type III:** Copied fragments of code with further modifications. There may be added, removed, or changed statements in addition to variation in identifiers literals, types, layout, white space, and comments.
- **Type IV:** Syntactically dissimilar code fragments that implement the same functionality.

Approaches that measure non Type I clones have limited application in industrial contexts because they require a parser for every programming language [5]. Additionally, programming languages are continuously evolving which requires constant maintenance of the parsers. But many software systems use multiple languages because it is often better to use specialized languages for different tasks than a single general-purpose one. For example, a software system might have a user interface built in *Javascript*, a backend in *Scala* with performance sensitive code written in *Rust*. So we stick to a mostly language independent approach which measures Type-1 clones. Type-1 clone detection finds a significant number of clones and has the additional benefit that it does not detect any false positives, such as getters and setters which are found by the other approaches [5].

We decided to build our own duplicate detector because we believe that using a third-party solution would lead to more complexity and workarounds, especially given that duplicate detection itself only requires little code.

We detect duplicates in four consecutive steps, as shown in Figure 4.2:

1. **Scoping:** We remove files from the code that are not manually maintained, for example, third-party files or auto-generated files. The user can choose manually which files to remove.
2. **Code cleaning:** We remove whitespace, empty lines, comments, import statements, and curly braces from the code.
3. **Duplicate detection:** We detect fragments of duplicated code that are six lines or longer. We chose six lines because it is a common configuration choice in other code analysis tools such as Simian and Sokrates [10] [23]. We use the Rabin-Karp algorithm to quickly filter out fragments of code that do not match and then perform a string comparison on the rest [14].
4. **Merge duplicates:** We merge the duplicated fragments of six lines of code into larger duplicated sections where possible. There may be many overlapping duplicates. For example, there might be a duplicated section that is ten lines long that occurs in four places in the codebase, and a subsection of those duplicated ten lines that occurs in five places in the code. These would correspond to two separate duplicate classes.

4.2.2 Frontend

We built the frontend in a way that decouples the visualizations from each other. Although the visualizations are connected, changes to one visualization will not affect the others. One can also easily add new visualizations or remove existing ones. The decoupling is done with an event system, as shown in Figure 4.3. For example, when the duplicate detection has finished in the backend, it returns the duplicate data to the scoping section where the user first clicked the run analysis button. The scoping section then sends an event that the duplication data has changed and all visualizations that subscribed to that event update themselves. If the user then hovers over a rectangle that represents a file in the file graph, the file graph will send an event that a certain file was selected. The hierarchy graph is subscribed to that event and will update itself to show the selected file.

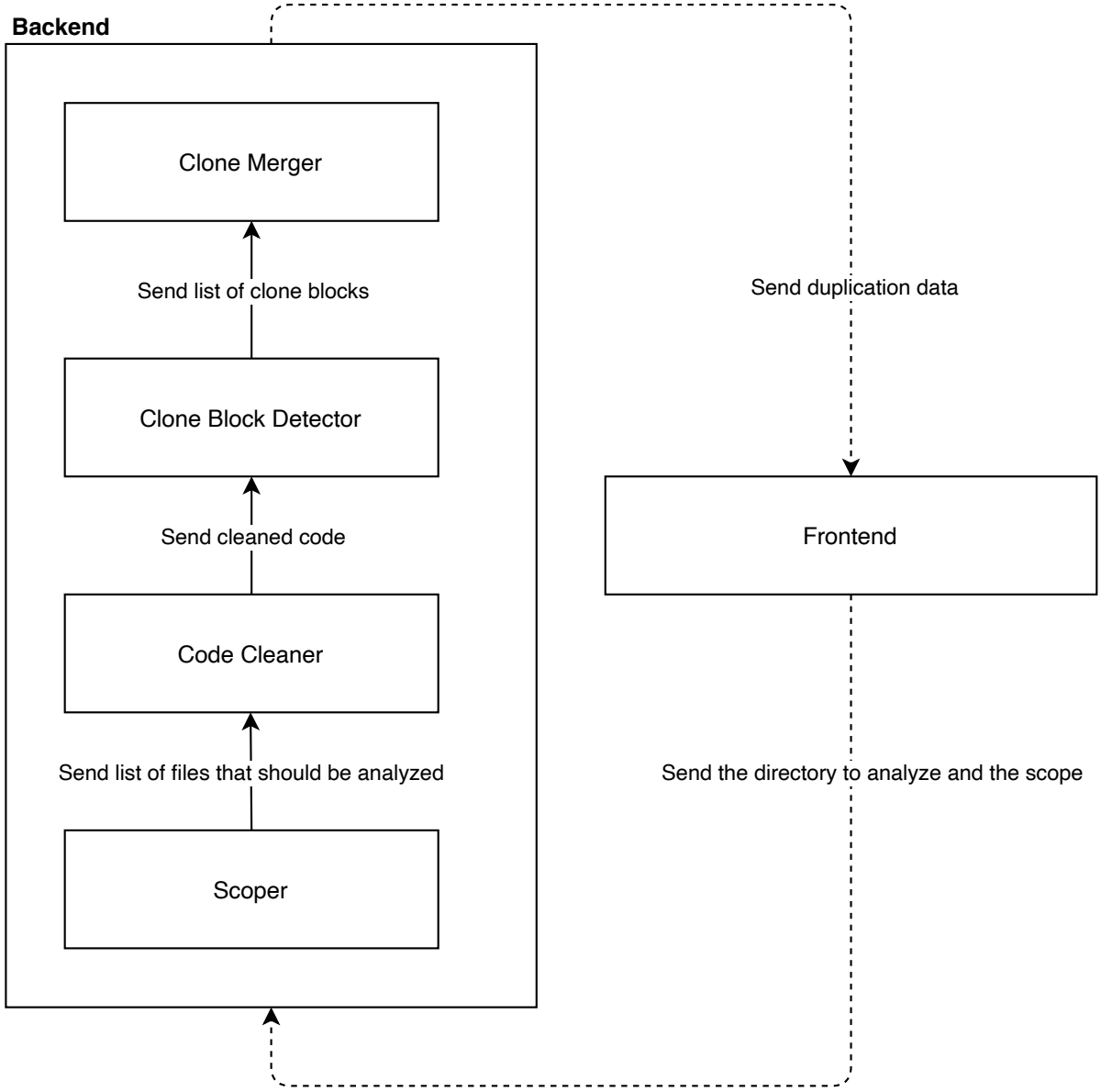


Figure 4.2: Shows the high-level backend architecture. The dotted arrows represent data transported over *HTTPS*. The full arrows represent data transported through method calls in *Java*.

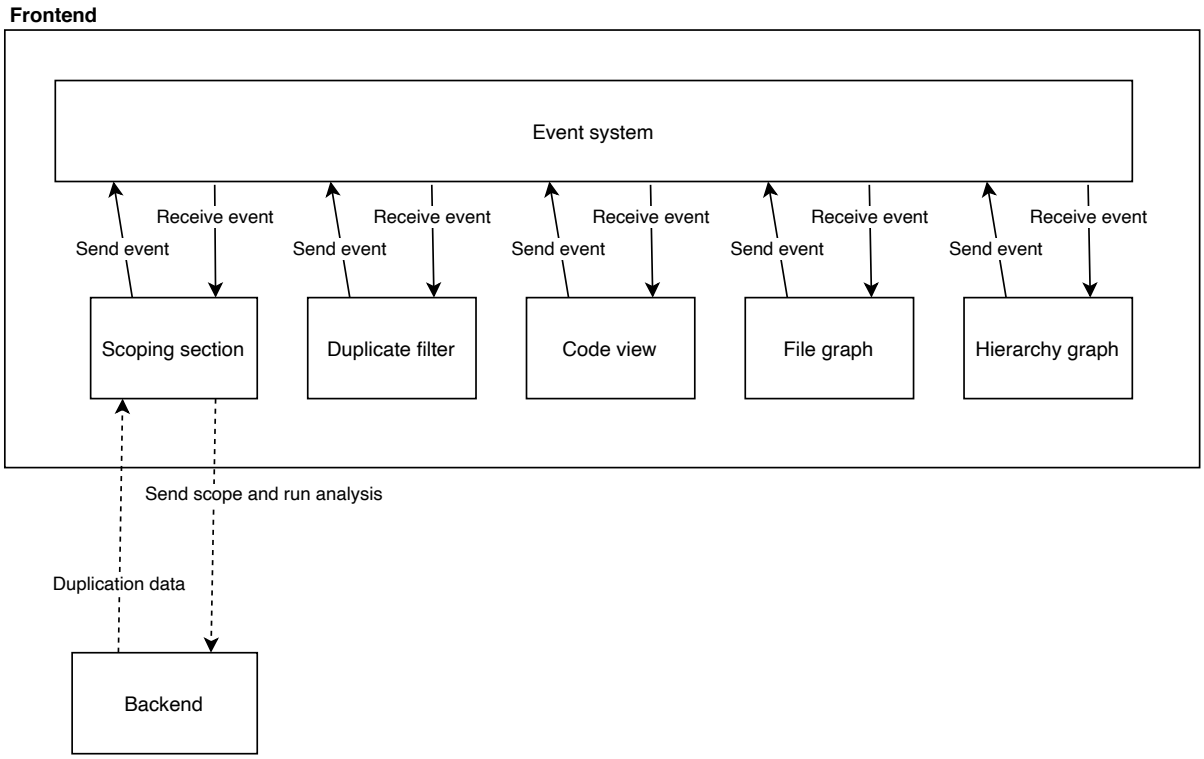


Figure 4.3: Shows the high-level frontend architecture. The dotted arrows represent data transported over *HTTPS*. The full arrows represent data transported through function calls in *Javascript*.

5
Tool Design

Much of the tool design has been influenced by Bret Victor's essay, *"Up and Down the Ladder of Abstraction"*, where he presents a systematic approach for building interactive visualizations [22]. He argues that moving between levels of abstraction is the most powerful way to gain insight into a system, because abstract representations help us see high-level patterns and concrete instances allow us to discover the explanation for the patterns.

Using this approach we can reason about the different abstraction levels that are used for organizing code. The smallest named fragment of code that can be unit tested is usually a *unit*. For example, in Java a unit would correspond to a method. As seen in Figure 5.1, files group related units together, and directories group related files together.

To construct our visualizations we reason about the properties of these abstractions which should be reflected in them. A first common property is that directories, files, and units have names. Visualizations should show those names because they have meaning that can be crucial for understanding the code. Directories can be nested within each other, so we should show which subdirectories they are composed of and in which subdirectories the duplication is located. In contrast to directories, the content in files is laid out from the first line to the last line. So there is meaning to where a duplicate occurs in a file that should be shown in our visualizations.

Our tool supports three consecutive levels of abstraction with three visualizations, as shown in Figure 5.2:

1. **Code view**: A view of the concrete source code that allows us to explore why the code is duplicated. It does not support a breakdown of duplicated units because that would require language specific parsing.
2. **File graph**: One abstraction level higher is a view where we can see what kind of duplication there is. It allows us to see duplication patterns across multiple files.

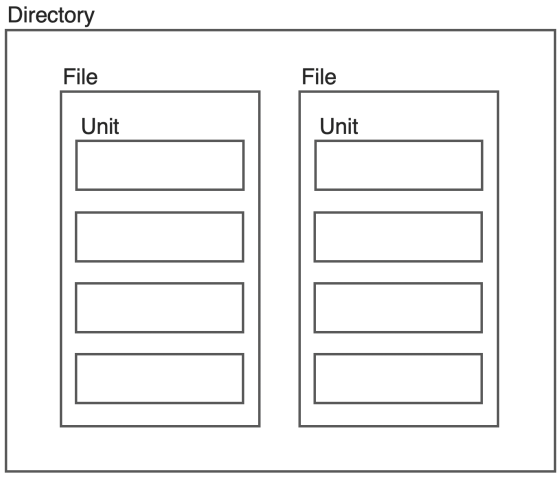


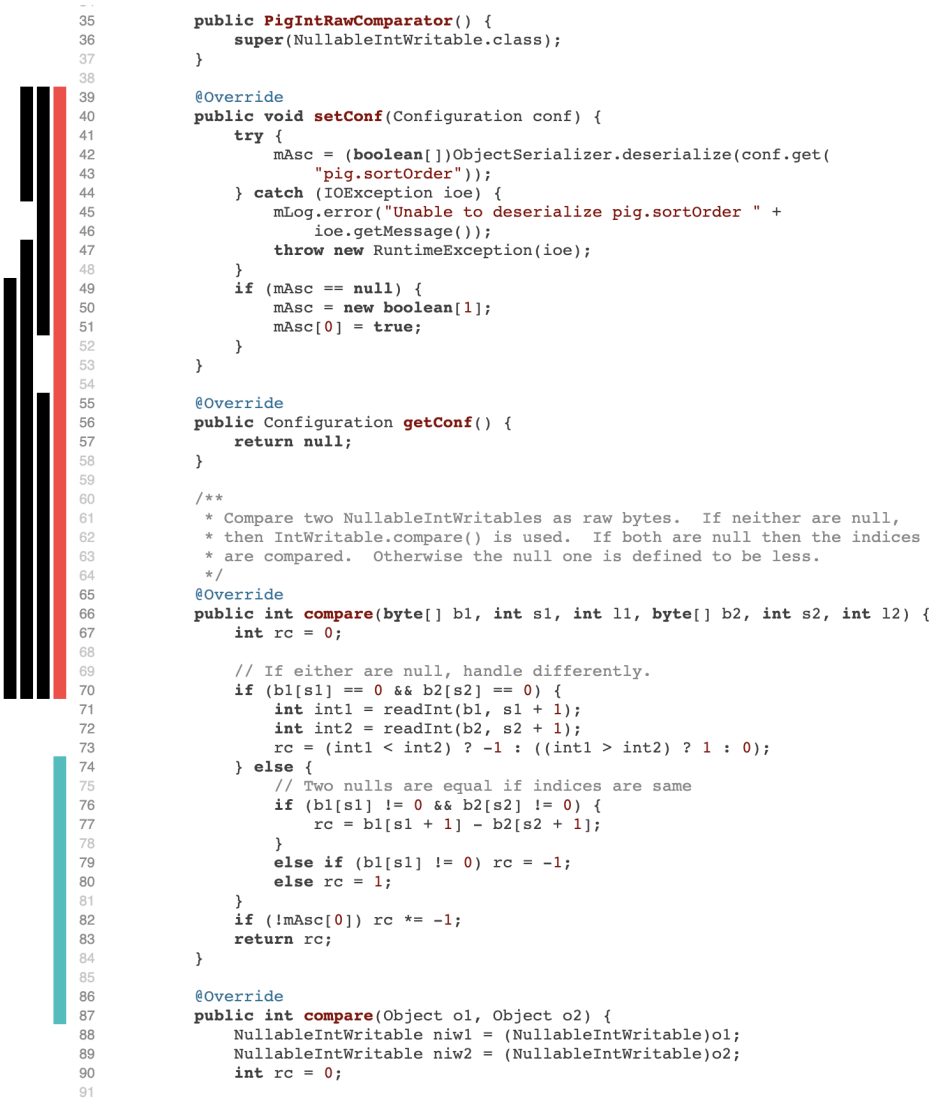
Figure 5.1: Shows the different levels of abstraction used for code organization.

3. **Hierarchy graph**: One abstraction level higher is a view that allows us to see where the duplication is located. We can see duplication patterns across all levels of the directory hierarchy.

Figure 5.2: The detail tab of our tool shows the file graph *(upper left)*, hierarchy graph *(lower left)* and the code view *(right)*. The codebase being analyzed in the picture is apache pig. Checked out at commit 59ec4a326079c9f937a052194405415b1e3a2b06 at https://github.com/apache/pig/commits/trunk

5.0.1 Code View

The code view shows the source code of a file. The line numbers of the lines that were removed in the code cleaning step are greyed out to make it transparent what the tool is doing, as shown in Figure 5.3.



```
35          public PigIntRawComparator() {
36              super(NullableIntWritable.class);
37          }
38
39          @Override
40          public void setConf(Configuration conf) {
41              try {
42                  mAsc = (boolean[])ObjectSerializer.deserialize(conf.get(
43                      "pig.sortOrder"));
44              } catch (IOException ioe) {
45                  mLog.error("Unable to deserialize pig.sortOrder " +
46                      ioe.getMessage());
47                  throw new RuntimeException(ioe);
48              }
49              if (mAsc == null) {
50                  mAsc = new boolean[1];
51                  mAsc[0] = true;
52              }
53          }
54
55          @Override
56          public Configuration getConf() {
57              return null;
58          }
59
60          /**
61           * Compare two NullableIntWritables as raw bytes.  If neither are null,
62           * then IntWritable.compare() is used.  If both are null then the indices
63           * are compared.  Otherwise the null one is defined to be less.
64           */
65          @Override
66          public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
67              int rc = 0;
68
69              // If either are null, handle differently.
70              if (b1[s1] == 0 && b2[s2] == 0) {
71                  int int1 = readInt(b1, s1 + 1);
72                  int int2 = readInt(b2, s2 + 1);
73                  rc = (int1 < int2) ? -1 : ((int1 > int2) ? 1 : 0);
74              } else {
75                  // Two nulls are equal if indices are same
76                  if (b1[s1] != 0 && b2[s2] != 0) {
77                      rc = b1[s1 + 1] - b2[s2 + 1];
78                  }
79                  else if (b1[s1] != 0) rc = -1;
80                  else rc = 1;
81              }
82              if (!mAsc[0]) rc *= -1;
83              return rc;
84          }
85
86          @Override
87          public int compare(Object o1, Object o2) {
88              NullableIntWritable niw1 = (NullableIntWritable)o1;
89              NullableIntWritable niw2 = (NullableIntWritable)o2;
90              int rc = 0;
91
```

Figure 5.3: The code view shows the source code of a file and marks the duplicated sections with colored rectangles on the left hand side.

We mark duplicated sections with rectangles beside the code as shown in Figure 5.3. The rectangles are colored the same as in the file graph to make it easier to see which fragments of code correspond to which sections of a file in the file graph. The overlapping duplicates are colored black. These are the duplicates that are not shown in the file graph because they are not part of the minimum spanning duplicates.

Hovering over a rectangle that represents a duplicated section will highlight all its occurrences in the hierarchy graph and in the file graph.

5.0.2 File Graph

The file graph allows us to see duplication patterns at file level. Each grey rectangle represents a file that starts on the left and ends on the right. The colored rectangles represent duplicated sections of the file.

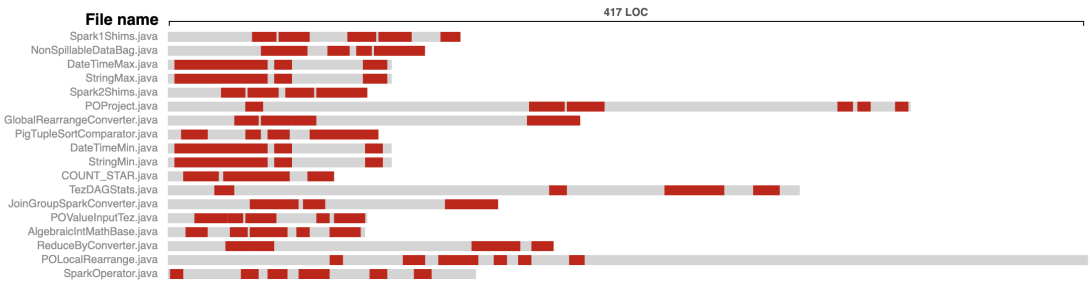


Figure 5.4: A small section of the file graph.

This zoomed out view of the code enables us to see duplication patterns across multiple files as well as within a file. We can easily identify different kinds of duplication patterns:

- Are there many small duplicates in a file that might make it difficult to refactor, or are there only a few large ones?
- Are duplicates spread over multiple files or are they only duplicated between two files?
- Are the same sections of a file always duplicated?

Layout To construct the layout shown in Figure 5.4 we represent the lines of code, which is code without comments and whitespace, of each file with a horizontal grey bar. We decided against a vertical layout. Although a vertical layout would require less mental mapping between the code view and the file graph, the file names would have to be vertical, which makes them harder to read.

Another issue is that a single huge file can dwarf all of the other files in view, making it hard to see any details of the duplicates. We fix that issue by wrapping the large files into multiple grey bars when they hit the right margin as shown in Figure 5.5. A problem that arises with this solution is that it is more difficult to recognize multiple bars of a wrapped file as one single file, because users are used to seeing most files as a single bar. To make it clearer that multiple bars represent one single file, we introduce some extra spacing between the first and the last bar and condense the spacing between the bars of the wrapped file. We decided against using a logarithmic view for showing the file size because it would also warp the size of the duplicates. With a logarithmic view, duplicates with the same number of lines would have different sizes depending on where they are in the file.

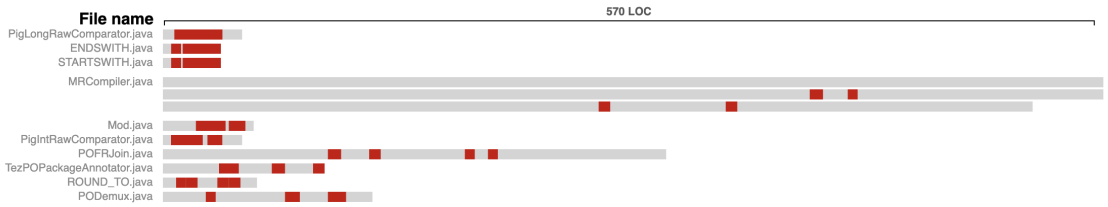


Figure 5.5: One large file and many small files in the file graph.

Interaction There are three main ways of interacting with the file graph.

Hover: When you hover over a file it marks all of the duplicates in the file with a different color. It also highlights the duplicates in other files that are duplicated with the selected file as shown in Figure 5.6. Each duplicate is represented with a colored rectangle. The corresponding duplicates in other files are marked with the same color.

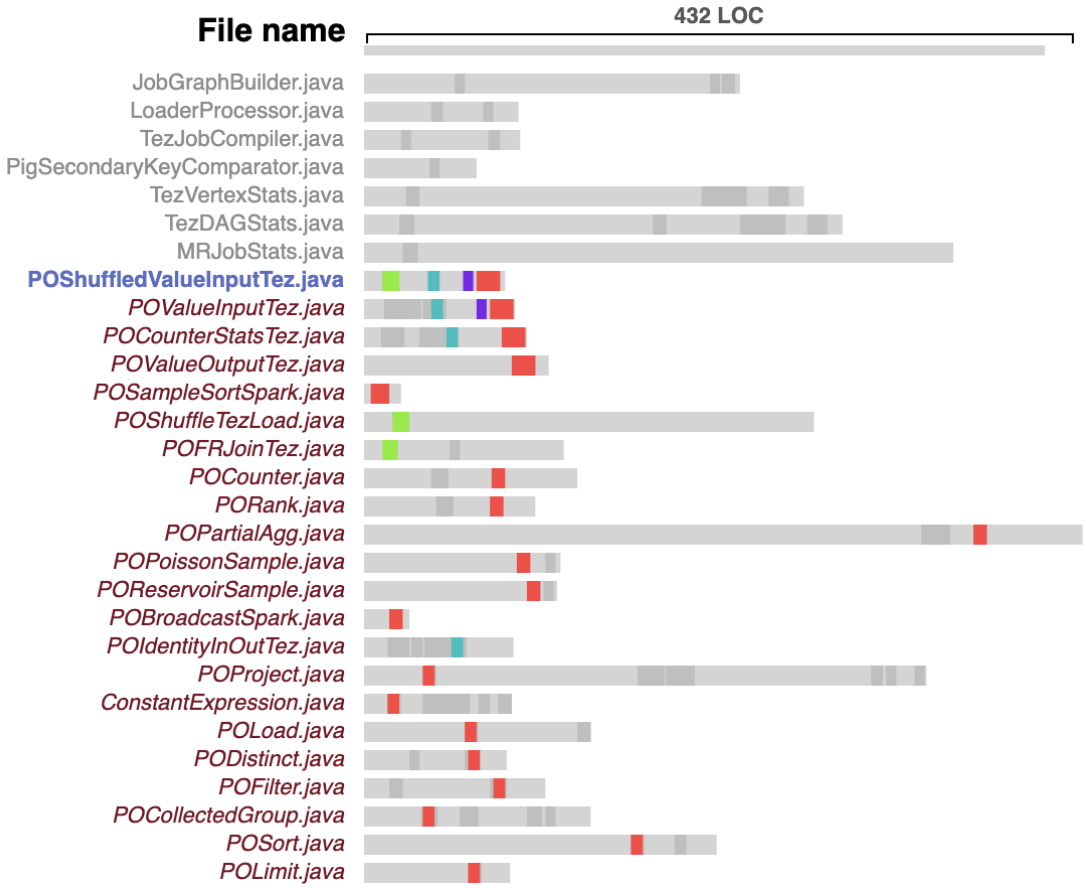


Figure 5.6: Shows where the duplicates of *POShuffledValueInputTez.java* occur in other files.

When hovering over a file users can also see where the file is located in the hierarchy graph as well as

which directories it is duplicated with, as shown in Figure 5.7. This helps users see where they are in the codebase and gives them an aggregated view of where the duplication with the selected file is coming from.

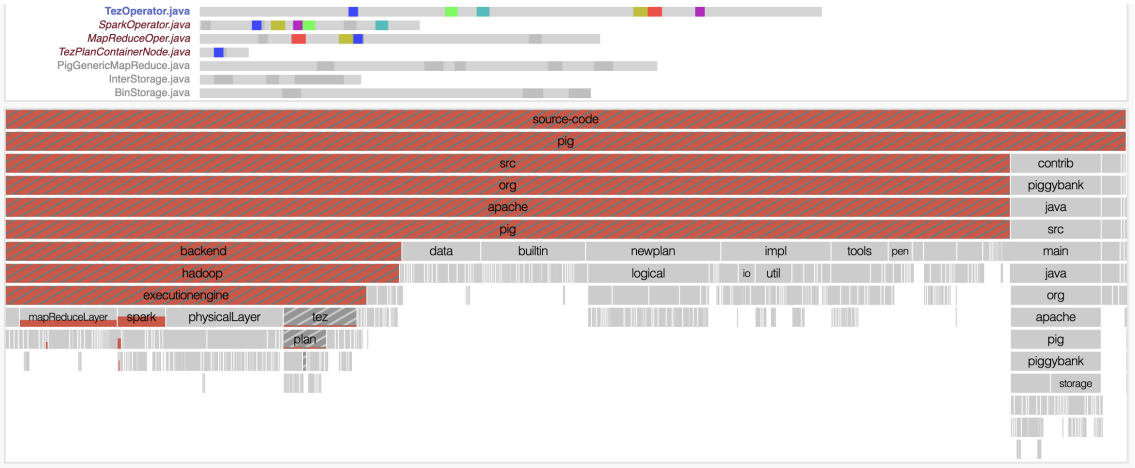


Figure 5.7: Shows where the duplicates of *TezOperator.java* occur within other files as well as where they occur in the hierarchy.

Left click & drag: Users can click on a rectangle and drag the mouse to scroll through the code, as shown in Figure 5.8. This makes it easy to step down a level of abstraction and view the concrete source code to find explanations for the patterns in the code view.

Figure 5.8: Shows how to click on a rectangle and drag the mouse to scroll through the code.

Right click: When users right click on a file it will re-arrange the ordering of the files in the file graph. It

moves all of the files that contain matching code duplicates with the selected file below the selected file, as shown in Figure 5.9. This makes it easy to see duplication patterns across files. An alternative design would have been to filter the other files instead of re-arranging the graph, but we decided against it because that way users would have to navigate back out of the filter. It would also make it harder to see patterns across duplicated files of duplicated files, which is possible by re-arranging the files multiple times when exploring.

Figure 5.9: Shows the file graph before right clicking on a rectangle that represents a file on the left and after right clicking on the right.

Colors When users hover over a file, the duplicates are colored based on maximal spacing from each other in the color spectrum. We use the Sinebow color scheme in d3 and find that this works well enough in practice to differentiate duplicates and see patterns.

There are several ways in which the duplicate colors could still be improved. Currently when hovering over different files in succession, the duplicates might change color because different files contain a different number of duplicates. To keep color constancy one could choose colors based on all duplicates of all files duplicated with each other. If this group of duplicates turns out to be too large in practice one could also calculate the color of the duplicates based on user interaction, which would lead to a smaller duplicate group. But the tradeoff is that you cannot compute the most optimal color spacing upfront.

Overlapping duplicates In an earlier design the file graph showed all of the the overlapping duplicates. One issue with this approach is that it does not scale to a large number of overlapping duplicates. The rectangles that represent duplicated sections can become too thin to see. But an even greater problem was that there was too much information on the screen, which made it hard to see patterns.

The idea when making the file graph was to enable users to see what kind of duplication there is. So we decided to only show the minimum spanning duplicates, which are the minimal set of duplicates and fragments of duplicates that span all of the duplicated lines of the file.

When hovering over a file we show all the duplicated sections in other files that are duplicated with the selected file. Because we only show the minimum spanning duplicates, some duplicates in the other files may be split into multiple duplicated sections.

Options There are two options for sorting the file graph. You can sort the graph by files with most duplicated lines and you can sort it by files with the largest percentage of duplication.

Users can also adjust how many lines of code are represented with each rectangle. This way users can zoom in on small files and zoom out on large files.

5.0.3 Hierarchy Graph

The hierarchy graphs allows users to see where in the directory hierarchy the duplication is located. Each bar represents a directory. As shown in Figure 5.10 the length of each bar corresponds to the lines of code in that directory. The amount of duplication is aggregated per directory and is shown in red.

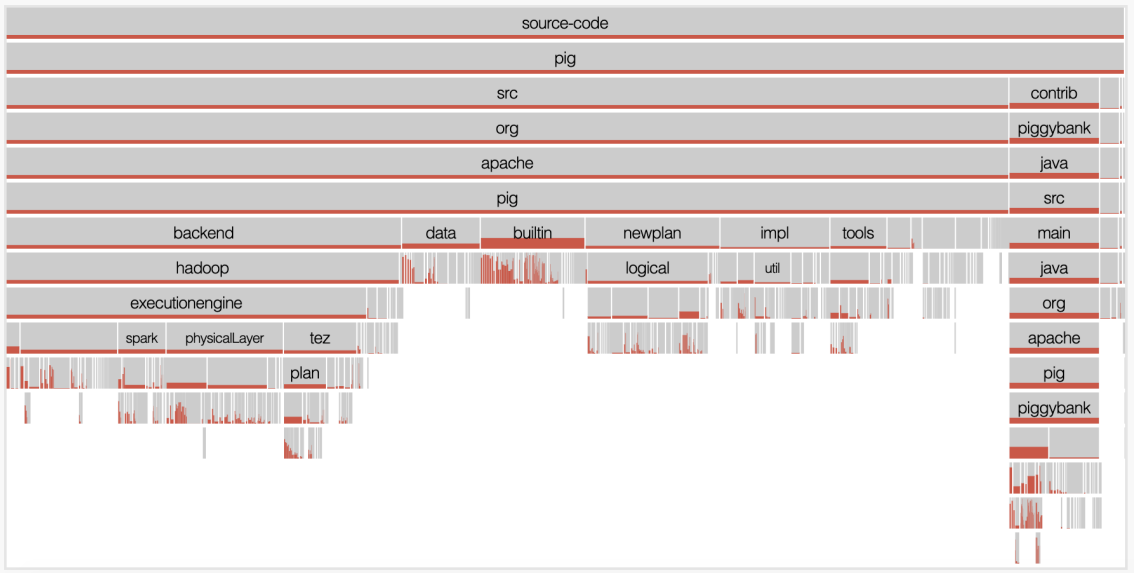


Figure 5.10: The hierarchy graph of the *apache pig* codebase.

Layout The most common way to represent this kind of aggregated hierarchy is with a sunburst graph, as shown in Figure 5.11. We decided against using a sunburst for three reasons. First, in the sunburst graph the size of the bar differs based on how far from the center it is, thus one cannot compare sizes across hierarchy levels. Second, the sunburst is circular which makes it space inefficient in a rectangular layout. Third, to label the directories we would have to use circular text, which is hard to read. To fix these problems we chose an icicle graph. Compared to the sunburst, one drawback is that the further down that the directories are in the hierarchy, the smaller and harder to read they will become.

An improvement to the hierarchy graph would be to remove redundant information by only showing directories if they contain more than one subdirectory or file. In the example seen in Figure 5.10 the directories *src*, *org*, *apache*, pig only contain one subdirectory, so they all have the same amount of duplication thus there is no new information added when we include them in the graph.

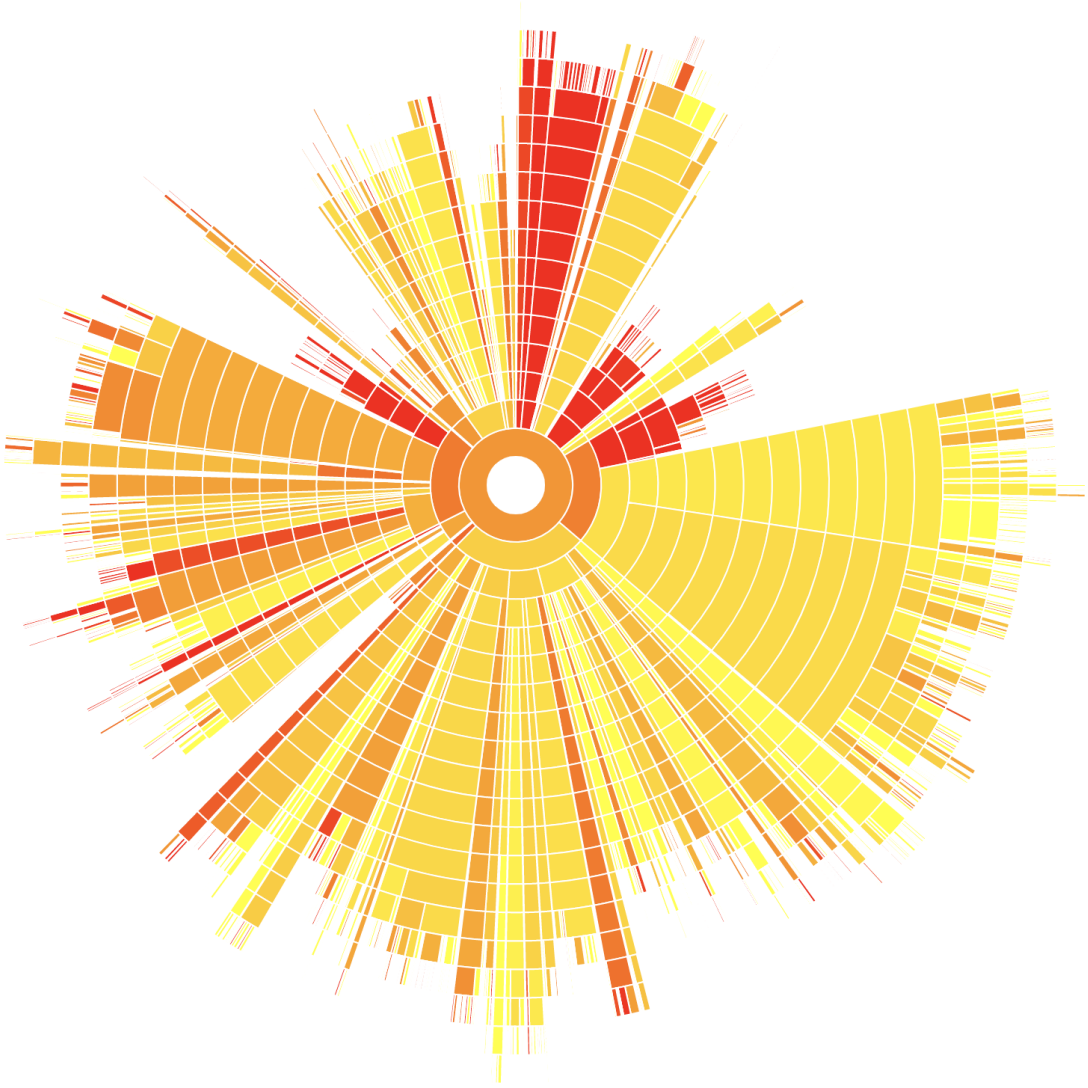


Figure 5.11: Example of a sunburst graph.

Colors The red bar shows how much of a given directory is duplicated in percent. The lines of code in a directory define the length of a bar and the width of the red bar is defined in percentage of duplicated code. So the total red area will correspond to the amount of duplicated code.

An alternative would have been to use color gradients to show the amount of duplication in each directory. But colors are less intuitive because they have to be learned. They also allow for less visual acuity in most cases. We could use a large range of colors that would allow for more visual acuity for certain ranges. For example, if we want more visual acuity for the 0%-15% range of duplication, we could encode that in the color range. This would require more upfront learning though.

Interaction There are three main ways of interacting with the hierarchy graph.

Click: Clicking on any bar in the graph will zoom in to that directory.

Click & Hold: Shows how much of the directories are duplicated with the selected directory. Rather than absolute numbers, these numbers are relative. For example, if 50% of the duplication of a selected directory comes from another directory then 50% of the bar will be red.

Right Click: Rearranges the file graph so that all of the files in the selected directory are shown.

5.1 Filtering

The filtering tab offers a way to filter files in the scoping section and a way to filter duplicates in the duplicate filter section.

5.1.1 Scoping

When measuring duplication in the source code we only want to analyze code that has to be maintained, which is usually manually written code.

This works in a similar way to the ignore rules that version control systems use to make sure that no irrelevant content is versioned. A good rule for version control systems is that they should only contain manually edited files and no data or code that is automatically generated during the build. But even when analyzing code from a version control system it is necessary to exclude additional files, because files in the versioning system often include bundled code, third-party code, or even generated code.

Users can decide what to exclude from the analysis using regular expressions as shown in Figure 5.12.

We split the scoping into two steps. First, we allow users to filter content using regular expressions. All of the file paths matching at least one expression will be excluded from the analysis. These files are excluded in the first step to make the analysis faster.

After the analysis has run the user can still exclude more directories or choose to only include certain ones. This allows for faster scoping as the user does not have to re-run the tool when he finds something that should be excluded. One can also comment or uncomment rules while investigating the codebase. The changes are then applied immediately.

5.1.2 Duplicate Filter

We allow the user to filter duplicates by length as well as by the number of times the duplicate occurs in the code, as shown in Figure 5.13. While filtering users can see a visual distribution of the number of duplicates with specific lengths as well as the number of duplicate classes that occur a specific number of times.

Figure 5.12: The scoping section of the tool.

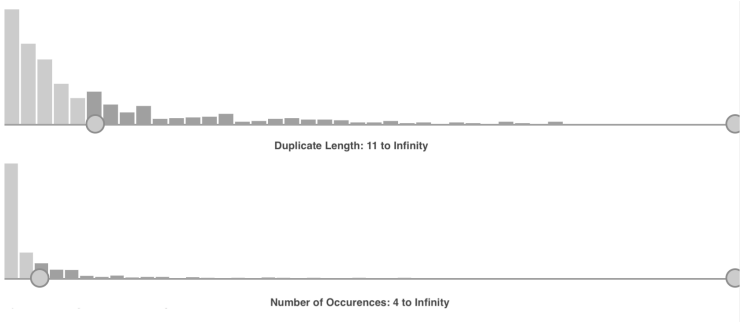


Figure 5.13: The duplicate filter section.

6 Validation

To validate our tool, we first perform a pilot usability test with two participants and then do a user study with five participants. We use *apache pig* as the codebase for the usability test and the user study. It is an open source system written in *Java* where 15% of the code is duplicated, which is typically a system that could benefit from an analysis of its duplication.

6.1 Pilot Usability Test

The goal of the usability test is to find any major usability bugs before we do the user study. We recruited two participants. Both have more than two years of industry experience as software developers.

We first introduced participants to the tool and the visualizations and then asked them to come up with recommendations that would reduce costs related to duplication for the codebase. We asked them to think aloud and observed them using the tool.

The first participant took around ten minutes to play around with the tool and explore different directories and files to get a general overview of what was in the code. She then eventually honed in on the *built-in* directory and the *piggybank/evaluation* directory. The main usability problem was that she had trouble comparing multiple files with each other, because we did not implement a file diff for the first evaluation. She diffed the files using a third-party command line tool, but the context switching and losing the place in the file was a major obstacle. But she eventually concluded that much of the duplication comes from math related classes where code was just copy-pasted to use a different data type. For example, there are files that are almost fully copy-pasted just to substitute all *integers* with *longs*. She found and that these files are mostly in *built-in* and in *piggybank/evaluation*. The participant also remarked that it would be nice to see duplicates of specific methods.

The second participant honed in quickly on the *built-in* and the *piggybank/evaluation* directory, where much of the duplication is. He also quickly found that there is a lot of duplication between the *built-in* directory and the *piggybank/evaluation* directory. He then filtered the duplicates by the number of times they occurred in the codebase and noted that there are some missing abstractions because some getters are duplicated more than ten times. He also had trouble comparing the differences between heavily duplicated files.

Both participants said that it was hard to come up with recommendations for a codebase that is not their own. But they both remarked that it is easy to move around the code. We observed the participants quickly jumping between the hierarchy graph, the file graph and the code view when trying to figure out why specific directories are heavily duplicated. We believe that this pilot study confirmed that the tool is usable, but could be improved with an interface for comparing files.

6.2 User Study

We recruited five participants. Two of them had more than two years of software development experience in industry. One of them had one year of experience. The other two participants were master students in Computer Science.

We first explained the visualizations of our tool, and then asked them to perform three tasks. There was no time limitation for the tasks. After completing the tasks, we asked them to rate each task depending on how much effort it took to perform the task on a scale from 1 (very low) to 5 (very high). In the end, we asked for qualitative feedback on the tool.

Instead of asking the participant's directly to perform complex tasks such as finding how to reduce maintenance costs related to duplication, we broke this task down into more focused tasks that are typical when looking for ways to reduce costs of software clones.

- **Task 1:** Find which directories contain the most duplication. Report these directories.
- **Task 2:** Find the clones that appear most often in the codebase. Report these clones.
- **Task 3:** Find files where the duplication is spread throughout a few large duplicates. Report these files.

The answers were consistent among all five participants. The answers only varied in how many of the directories, files or clones they reported. The participants reported that the tasks required low effort with our tool, as shown in Figure 6.1.

When asked for qualitative feedback, the participants said that the tool is easy to use. They said that "it is easy to navigate to any location in the code", "the view with the directories and files is great", "the filter is easy to use and very useful". When giving suggestions what could be improved one of the participants said that the duplicate filter could give more information on how many duplicates are still included after filtering. Another participant said that the right-click on the file graph can be a bit confusing, but maybe one would get used to it.

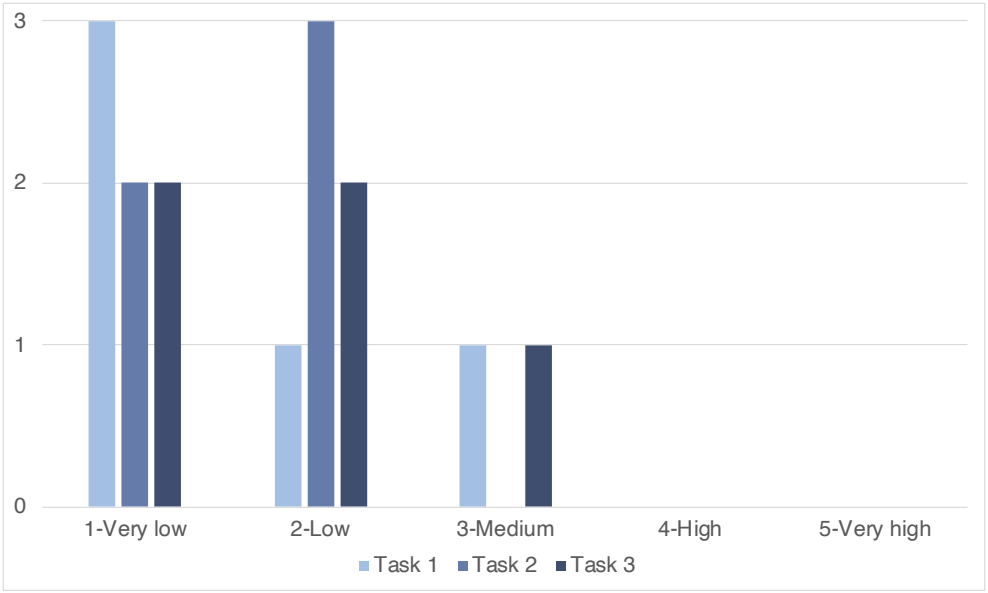


Figure 6.1: Shows the number of participants per effort category distributed per task.

6.3 Threats to Validity

We believe that the main threat to validity from the user study comes from the construct validity. We measure a few specific tasks that are typical when finding ways to reduce maintenance costs that are associated with code clones. But success in these narrow tasks might not necessarily prove that the tool is also successful in letting users gain a mental model of the duplication in the codebase.

7 Discussion

In this section, we reflect on the challenges and limitations that we encountered when working on the tool. We elaborate on which parts worked well and and what could be improved in the future.

7.1 What worked well

When visualizing geographic data with a world map, we can rely on everyone having the same map. And tectonic plates do not shift the continents halfway around the globe every few weeks. This is not the case in software. With software each developer has their own mental model of the code. And the mental model is hard to document because it is always changing.

When visualizing software, we do not have access to detailed geographic maps upon which we can plot our data. But we have other kinds of maps. There are few almost universal code organization principles that we can use. For example, developers organize their code in directories and files. So we focus on the existing abstractions that developers use to organize their code instead of inventing new layouts. We also show the names of the files and the directories in each visualization. Showing those names can be challenging because they have to be laid out in a readable manner. But we believe it is crucial, because without them users can quickly get lost in a sea of abstract bars and colors that have no meaning attached.

In software, there can be many different maps. The directory and file structure is just one of them. One can, for example, also use a map of execution traces or dependencies between software entities. The more use case specific we can construct this map, the better we can visualize the results. For example, visualizing performance works much better with call graphs than with directory structures. So in our case, we believe that making visualizations specifically for duplication was the right approach.

7.2 What could be improved

Deciding which features should be supported by which visualization was hard. Each visualization can only support a certain number of features until it starts to break, and when that starts to happen it might be better to build a new visualization that is more specialized. For example, in the hierarchy graph we added some functionality that might have been better suited for a more specialized graph. In the hierarchy graph, users can see which directories are duplicated with each other through interaction. But the structure of the graph makes it impossible to show all of this information simultaneously. We believe that adding more types of visualizations, each with their specific use cases, would make the tool better.

There is also a difference between visualizations meant for exploring versus visualizations meant for communicating. Our tool focuses mostly on the former. But communicating results to stakeholders might be equally important. We believe that there is also opportunity for interactive visualizations that enable users to condense what they found in the exploratory interface and construct a visualization that highlights what they want to communicate.

We left out questions concerning system evolution for future work. We believe that using a similar approach one could build effective visualizations for viewing the duplication change over time.

8 Conclusion

We developed a prototype tool offering a set of connected interactive visualizations for inspecting software duplication. They make it is easy to freely explore the codebase by stepping between the concrete source code and higher levels of abstraction. We followed Bret Victor's systematic approach for building interactive visualizations, which enables users to see high-level patterns by taking a bird's-eye view and to find the explanation for those patterns by inspecting the concrete instances of duplicated code. In contrast to existing stand-alone duplication visualizations, our tool supports complex tasks that require users to build a mental model of the duplication in the codebase to tie their findings together.

Software maintenance makes up more than half of all software development costs [16]. Our objective is to advance research that can help reduce maintenance costs associated with code clones. We hope that we could contribute to the research towards making effective solutions for code duplication that can be more widely adopted in the industry.

Bibliography

[1] Muhammad Asaduzzaman, Chanchal K Roy, and Kevin A Schneider. VisCad: flexible code clone analysis support for NiCad. In *Proceedings of the 5th International Workshop on Software Clones*, pages 77–78, 2011.

[2] Brenda S Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[3] Hamid Abdul Basit, Muhammad Hammad, and Rainer Koschke. A survey on goal-oriented visualization of clone data. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 46–55. IEEE, 2015.

[4] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993.

[5] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[6] Stephen G Eick, Joseph L Steffen, Eric E Sumner Jr, et al. Seesoft — a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[7] Christopher Forbes, Iman Keivanloo, and Juergen Rilling. Doppel-code: A clone visualization tool for prioritizing global and local clone impacts. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 366–367. IEEE, 2012.

[8] Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. A systematic mapping study of clone visualization. *Computer Science Review*, 37:100266, 2020.

[9] Jan Harder and Nils Göde. Efficiently handling clone data: RCF and cyclone. In *Proceedings of the 5th International Workshop on Software Clones*, pages 81–82, 2011.

[10] Simon Harris. Simian: Features. https://www.harukizaemon.com/simian/features.html, Accessed: 2021-01-09.

[11] Benedikt Hauptmann, Veronika Bauer, and Maximilian Junker. Using edge bundle views for clone visualization. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 86–87. IEEE, 2012.

[12] Zhen Ming Jiang, Ahmed E Hassan, and Richard C Holt. Visualizing clone cohesion and coupling. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 467–476. IEEE, 2006.

[13] Cory Kapser and Michael W Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 19–28.

[14] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987.

[15] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[16] Jussi Koskinen. Software Maintenance Costs. https://web.archive.org/web/20120313070806/http://users.jyu.fi/ koskinen/smcosts.htm, Accessed: 2021-01-11.

[17] Debajyoti Mondal, Manishankar Mondal, Chanchal K Roy, Kevin A Schneider, Yukun Li, and Shisong Wang. Clone-world: A visual analytic system for large scale software clones. *Visual Informatics*, 3(1):18–26, 2019.

[18] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. Clonepacker: A tool for clone set visualization. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 474–478. IEEE, 2015.

[19] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[20] Manamu Sano, Eunjong Choi, Norihiro Yoshida, Yuki Yamanaka, and Katsuro Inoue. Supporting clone analysis with tag cloud visualization. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, pages 94–99, 2014.

[21] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, 2013.

[22] Bret Victor. Up and Down the Ladder of Abstraction: A Systematic Approach to Interactive Visualization. http://worrydream.com/LadderOfAbstraction/, Accessed: 2021-01-09.

[23] Željko Obrenović. Examined Line: The Art of Source Code Analysis with Sokrates. https://www.sokrates.dev/book/duplication, Accessed: 2021-01-09.