Improving Code Reviews using the Envision IDE

Master Thesis Report

Manuel Galbier

Supervisor: Dimitar Asenov
ETH Zürich

01.09.2016

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Code review is a widely applied, essential practice for achieving successful evolution of software systems. Besides finding defects in software, it also leads to improved code and knowledge transfer between developers, ultimately leading to higher software quality [3, 6]. Thus, most software companies have their developers participate in code reviews. The process itself is often time-consuming, and although modern code review tools like GitHub[1] or Gerrit[2] try to aid the developers in the process, current research [18] shows that developers are not satisfied with the state of the art. The key technical challenge of code review is code and change understanding. In this area most current tools are limited and do not provide a lot of support to the reviewer. The features provided by the tools consist in general of a line-based diff between the old and the new version of the software as well as the ability to comment on changes. Interviews with developers indicate that it would help having the ability to see the diff in the context of the entire class and compare the difference between multiple consecutive patches.

We improve the code review process by designing features which help developers in their code and change understanding needs. By using a fine-grained version control system as a basis we can provide developers with more detailed information about a change set. Instead of distinguishing only insertions and deletions the system used is able to distinguish modifications and moves of code. By designing intuitive visualizations for the diff we are able to communicate this additional information, which developers would otherwise have to manually identify, leading to more efficient change understanding. These visualizations, coupled with a mechanism to quickly access the context of a change using a zoom function, form the basis of our project. To further help the developers gain familiarity with a change set, we designed three advanced diff features. An overview feature displays changes in the context of the entire software project. A name change summary option summarizes changes related to, e.g., the renaming of a field. A history features allows the developer to inspect the evolution of a code component of the software project over multiple revisions. It is possible to organize the change sets of a code review automatically by using strategies which, for example, group related changes. This, combined with a feature that allows the author of a commit to define steps in which changes should be inspected, provides guidance to reviewers and helps in understanding change sets.

These features help developers in tasks which they otherwise need to execute manually and facilitate code inspection during code reviews. By providing more information about changes compared to current code review tools we improve code and change understanding and ultimately the code review process.

[1]`https://github.com`
[2]`https://www.gerritcodereview.com/`

Acknowledgments

I would like to thank Dimitar Asenov for his support and input during the thesis. I very much appreciate the time he took to provide valuable feedback and advice during code reviews and meetings. His experience with designing software systems and visualizations helped tremendously during the work on the thesis.

Also, I would like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this interesting project.

Finally, I would like to thank my girlfriend and family for their patience and support during the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

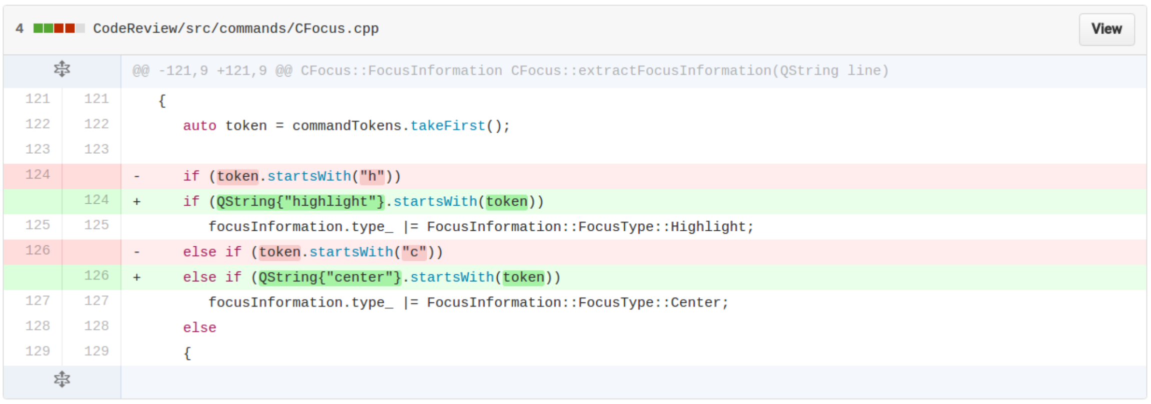
In this chapter we detail the motivation of our project and give an overview of the designed features. Further, we give an introduction to the Envision IDE in which these features were implemented.

1.1 Motivation & overview

During recent years the software industry has seen tool-based code review become an almost universally adopted process to help maintain high quality in software projects. Many well-known and successful companies such as Microsoft [3], Google [22] and Facebook [7] employ the process to reap the benefits of knowledge transfer, increased team awareness and improved code [3]. Performing code reviews takes time and effort. In a recent survey the time spent in the code review process was found to be on average six hours per developer per week [5]. This, combined with the fact that an increasing number of developers (e.g., more than 50'000 Microsoft employees were involved in code reviews in January 2015 alone [6]) participate in code reviews, illustrates the importance of code reviews today.

Despite the fact that code review plays an important role in today's software development process, multiple recent surveys found that developers are not satisfied with the state of the art in code review tool support. The main difficulty developers face during code review is code and change understanding [3] or in a broader sense to gain familiarity with the code [18]. One specific area where developers would want to see improvements is in the diff of the changes. Instead of comparing changes "line-by-line", reviewers mentionend during a recent survey [18] the importance of having the ability to compare them on a "file-by-file" basis. Making diffs aware of object relationships and allowing for IDE-like navigation, e.g., inspecting definitions not part of the current change set, would further help in the change-understanding process [23].

In Figure 1.1 the line-based context approach of GitHub is shown. Very little of the code surrounding the changes is displayed by default. The amount of context supplied is exactly three lines of code before and after the location of a change in a file, including empty lines. This means that it is often necessary to iteratively request more context to get an idea of the code surrounding a change. To be able to judge the correctness of a change it is often times necessary to inspect the neighboring code. For example, if a reviewer is not familiar with the method containing a change and the change is interacting



```
4  CodeReview/src/commands/CFocus.cpp                                         View

@@ -121,9 +121,9 @@ CFocus::FocusInformation CFocus::extractFocusInformation(QString line)
    {
        auto token = commandTokens.takeFirst();

-       if (token.startsWith("h"))
+       if (QString{"highlight"}.startsWith(token))
            focusInformation.type_ |= FocusInformation::FocusType::Highlight;
-       else if (token.startsWith("c"))
+       else if (QString{"center"}.startsWith(token))
            focusInformation.type_ |= FocusInformation::FocusType::Center;
        else
        {
```

Figure 1.1: Default context shown for a change in GitHub.

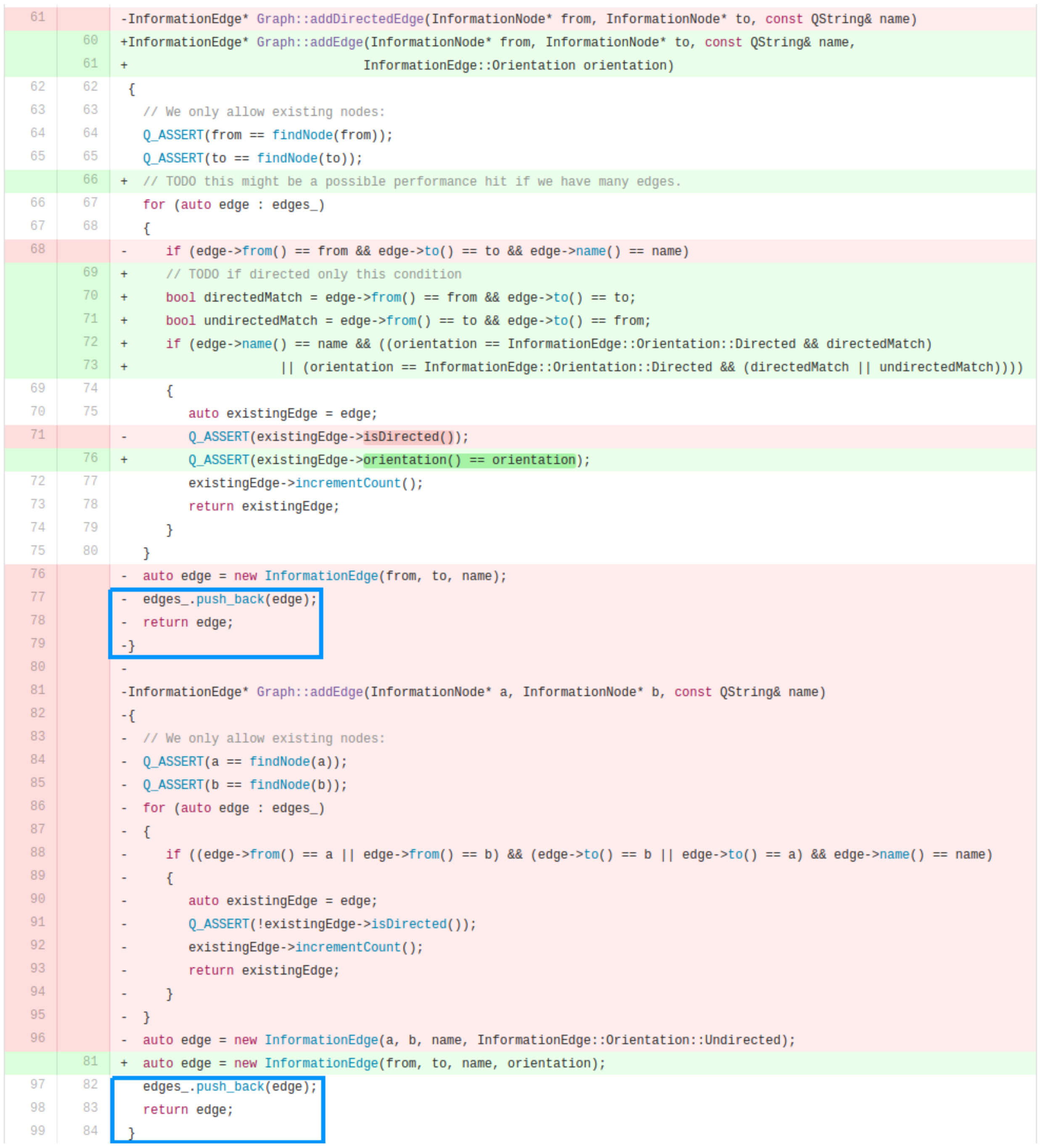
with other parts of this method, it can be important to inspect more of the code to reason about the correctness of the change. The diff itself can sometimes be hard to understand, for example, in Figure 1.2 incorrect matches and line-based differences lead to situations where some effort is needed to understand what actually happened.

Further difficulties faced by developers involve judging the risk imposed by a change on other software components (e.g, classes, methods, etc.) as well as deciding whether a change is complete and consistent [23]. Imagine a developer inspecting a change which adds a new option to an enum and uses it in different places. A question developers might ask themselves is whether the option is used in all the necessary places and whether, for example, all existing switches operating on this enum are handling the new option appropriately. To answer such a question all cases of such switches would need inspection.

The goal of this project is to tackle some of the areas where developers have expressed difficulties fulfilling their information needs and ultimately help them to perform more efficient code reviews. Achieving this is challenging and requires efficient communication of change information and additional tools which aid in the comprehension of code changes. Using the Envision IDE and its fine-grained version control system as a basis we designed different extensions to facilitate change understanding. We designed a visualization for Envision's diff mechanism which serves as a basis for most of the features. Additionally, we designed advanced diff features which aim to help the developer in understanding the changes and gaining familiarity with the code.

First, the history feature allows developers to inspect the evolution of a specific software component like, for example, a class or method. This is different to the common history feature available in current line-based VCS which instead show the history of an entire file. Having the possibility to inspect the changes made to the selected component individually for each commit involved can help to break down the process of understanding it into smaller steps. It allows developers to focus on changes specifically made to the selected component by omitting all non-related changes. Since it also tracks modifications and moves across files it is more precise than line-based diffs.

Second, the name change summary helps in the task of inspecting changes by summarizing changes related to a renaming. Imagine a big change set which also includes the renaming of a variable that is used in various places. Lots of the changes displayed in,



```
-InformationEdge* Graph::addDirectedEdge(InformationNode* from, InformationNode* to, const QString& name)
+InformationEdge* Graph::addEdge(InformationNode* from, InformationNode* to, const QString& name,
+                                InformationEdge::Orientation orientation)
 {
   // We only allow existing nodes:
   Q_ASSERT(from == findNode(from));
   Q_ASSERT(to == findNode(to));
+  // TODO this might be a possible performance hit if we have many edges.
   for (auto edge : edges_)
   {
-      if (edge->from() == from && edge->to() == to && edge->name() == name)
+      // TODO if directed only this condition
+      bool directedMatch = edge->from() == from && edge->to() == to;
+      bool undirectedMatch = edge->from() == to && edge->to() == from;
+      if (edge->name() == name && ((orientation == InformationEdge::Orientation::Directed && directedMatch)
+                 || (orientation == InformationEdge::Orientation::Directed && (directedMatch || undirectedMatch))))
       {
          auto existingEdge = edge;
-         Q_ASSERT(existingEdge->isDirected());
+         Q_ASSERT(existingEdge->orientation() == orientation);
         existingEdge->incrementCount();
         return existingEdge;
      }
   }
-  auto edge = new InformationEdge(from, to, name);
-  edges_.push_back(edge);
-  return edge;
-}
-
-InformationEdge* Graph::addEdge(InformationNode* a, InformationNode* b, const QString& name)
-{
-  // We only allow existing nodes:
-  Q_ASSERT(a == findNode(a));
-  Q_ASSERT(b == findNode(b));
-  for (auto edge : edges_)
-  {
-      if ((edge->from() == a || edge->from() == b) && (edge->to() == b || edge->to() == a) && edge->name() == name)
-      {
-         auto existingEdge = edge;
-         Q_ASSERT(!existingEdge->isDirected());
-         existingEdge->incrementCount();
-         return existingEdge;
-      }
-  }
-  auto edge = new InformationEdge(a, b, name, InformationEdge::Orientation::Undirected);
+  auto edge = new InformationEdge(from, to, name, orientation);
   edges_.push_back(edge);
   return edge;
 }
```

Figure 1.2: Imprecise textual matchings in GitHub. On a first look it may seem, that the method addEdge in the new version of the code was put together using parts of the methods addDirectedEdge and addEdge of the old version. Inspecting the change more closely reveals that the three lines marked with rectangles are exactly the same. This means it is possible to mark the bottom lines as deleted instead of the ones further up, which would lead to the method in the bottom being marked as deleted and allow the reviewer to focus on the changes in the top.

for example, GitHub, would relate to renamed references to this variable. By summarizing all changes related to a name change it is possible to reduce the amount of changes developers have to inspect manually.

Third, the overview feature allows developers to see changes in the context of the entire project, instead of the more localized approach of showing changes used by current tools. This provides an overview of where the changes are located in the project and which other components could be affected by the change.

Further, we provide developers with the ability to write code review comments on the changes displayed. Since comments are considered the main building block of code reviews [6] we give developers the possibility to leave fine-grained comments that can be associated with any of the displayed nodes of the visualization, as compared to the line-based comments used in most of the current code review tools. The author of a change set has the possibility to add focus information to the comments. This can be used to, for example, define a sequence of steps in which comments should be shown to the reviewer to provide some guidance during the review of the changes.

Lastly, we designed an automated grouping and ordering function, which allows reviewers to automatically have the changes grouped and ordered according to some user-specified strategies. Imagine a large change set, which mixes changes related to different features. Instead of showing the reviewer the changes in an arbitrary order, it is possible to show the changes in groups. For example, grouping all changes which have some kind of relation (e.g., use or depend on each other). This gives structure to the changes and allows the reviewers to inspect the changes in smaller chunks by inspecting, for example, one group after another.

The report is structured as follows: Chapter 2 details the design process of our diff visualizations as well as the advanced diff features and shows the final design. We then discuss the design of features aimed to improve the code review process further in chapter 3. The implementation in Envision is discussed in Chapter 4. An evaluation of the designed features can be found in Chapter 5. Chapter 6 discusses related work, while Chapter 7 concludes the report and discusses future work. In Appendix A, we give an overview of the new commands and keyboard shortcuts that we introduced to Envision.

1.2 The Envision IDE

The features designed during this project were implemented within the Envision IDE[1]. Envision is an open-source IDE prototype, mainly developed at ETH Zurich [1]. It features a structured visual editor for object-oriented languages to improve productivity of developers working on large projects. To abstract from the syntactic rules of languages it uses abstract syntax trees to internally represent the software projects.

Figure 1.3 shows an example project loaded in the Envision IDE. It draws the components (e.g., classes, methods etc.) of the software project in a 2D canvas, which enables showing the entire project at once instead of using a per-file approach. By using colors, icons and other visualization techniques, it allows developers to efficiently comprehend code features [2]. On the bottom left it features a mini-map of the software project for additional orientation and fast navigation.

[1]`http://www.pm.inf.ethz.ch/research/envision.html`

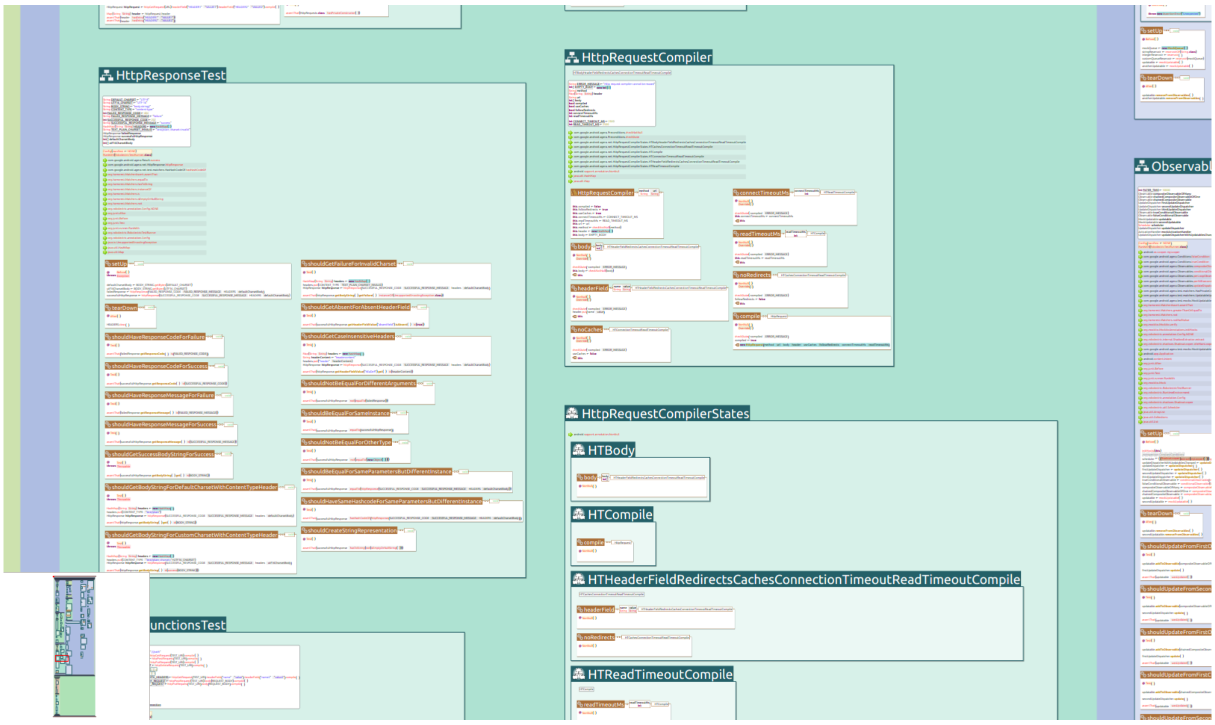


Figure 1.3: A project loaded and displayed in Envision.

Envision has a fine-grained version control system which is built on top of Git[2]. It was developed during the master thesis of Otth [21] and the bachelor thesis of Guenat [15]. The system is fine-grained because, unlike traditional text-based solutions, it tracks the changes on the basis of nodes in the abstract syntax tree of the software. Due to the structure-aware nature of this system it is not dependent on the syntax of the program and allows it to distinguish four types of changes: insertions, deletions, moves and modifications. These initial implementations were concerned with the back-end of the system only and did not focus on providing a graphical user interface, which we do in this project.

[2]`https://git-scm.com/`

Chapter 2

Visualizing differences between software versions

To facilitate change understanding it is vital to display the differences between software versions clearly. During a recent survey developers requested diffs tools, which provide more context and do not only operate on a per-line basis [18]. In this chapter we discuss the design of the visualization for a diff which differentiates between four change types: insertions, deletions, moves and modifications. Since this will be the basic tool to understand the changes of the code it is of great importance during the code review process.

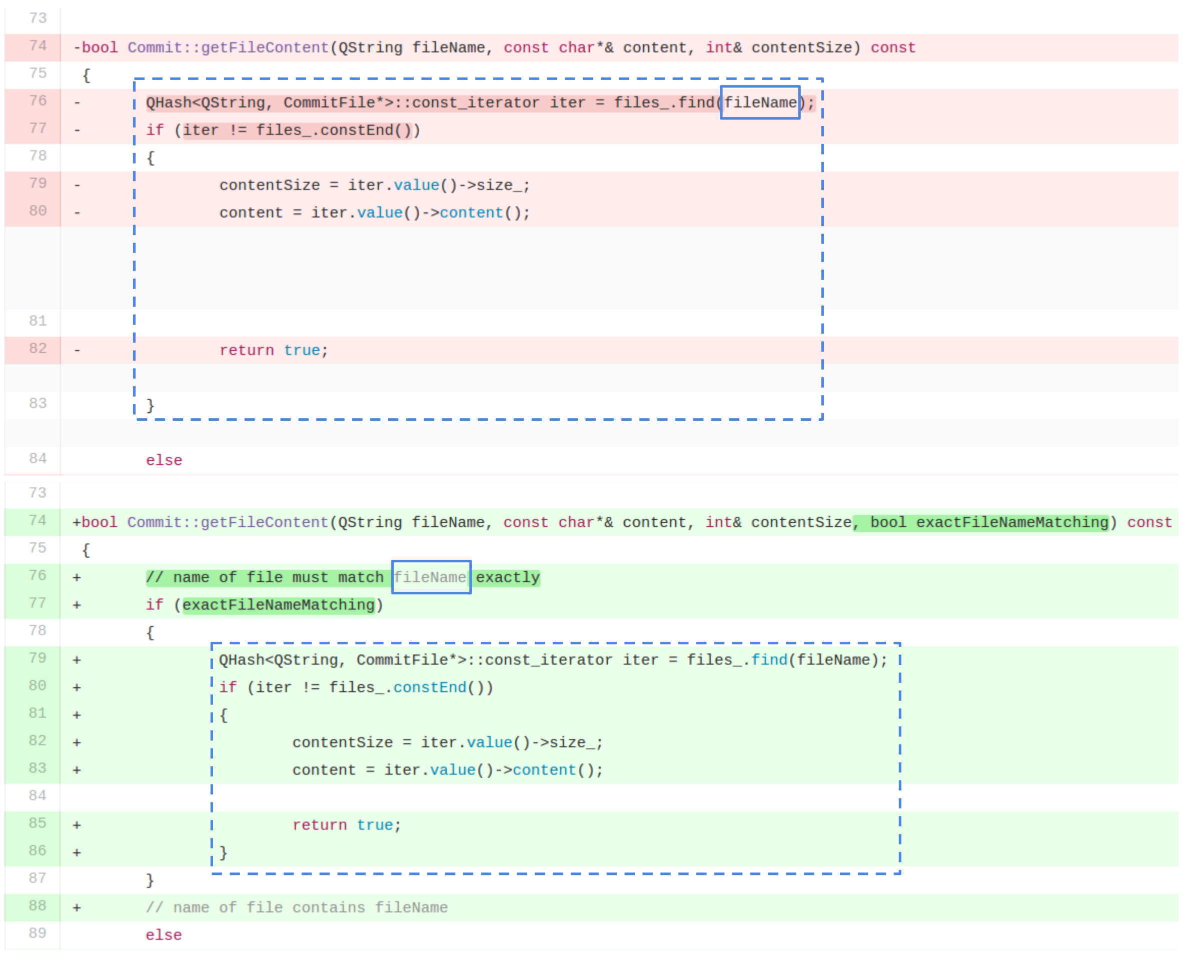
2.1 Issues of current tools

In current tools, which mainly display line-based diffs, changes can sometimes be difficult to interpret correctly. Most diffs are not aware of the structure of the code and are limited by textual constraints [13], which can lead to misleading matchings. In Figure 2.1 we provide an example which came up during the development process of this project and illustrates a case of such misleading matchings.

Such cases of inaccurate text-matchings can lead to confusion and hinder the understanding of the changes instead of facilitating it. Another problem is how current tools present the context of changes. Most of them operate on a line-number basis and have no mechanisms to intelligently show context. It can be cumbersome for developers to use such line-based context mechanisms if a lot of actions are required to display the amount of context needed to understand a change. Figure 2.2 shows an example where the amount of context shown by default is not enough.

2.2 Visualizations for improved diff understanding

Keeping the difficulties and requests for improvements by developers in mind, we describe in this section the process of designing diff visualization, which help developers to understand fine-grained code changes.



```
-bool Commit::getFileContent(QString fileName, const char*& content, int& contentSize) const
 {
-        QHash<QString, CommitFile*>::const_iterator iter = files_.find(fileName);
-        if (iter != files_.constEnd())
         {
-                contentSize = iter.value()->size_;
-                content = iter.value()->content();

-                return true;
         }
         else
```

```
+bool Commit::getFileContent(QString fileName, const char*& content, int& contentSize, bool exactFileNameMatching) const
 {
+        // name of file must match fileName exactly
+        if (exactFileNameMatching)
         {
+                QHash<QString, CommitFile*>::const_iterator iter = files_.find(fileName);
+                if (iter != files_.constEnd())
+                {
+                        contentSize = iter.value()->size_;
+                        content = iter.value()->content();

+                        return true;
+                }
         }
+        // name of file contains fileName
         else
```

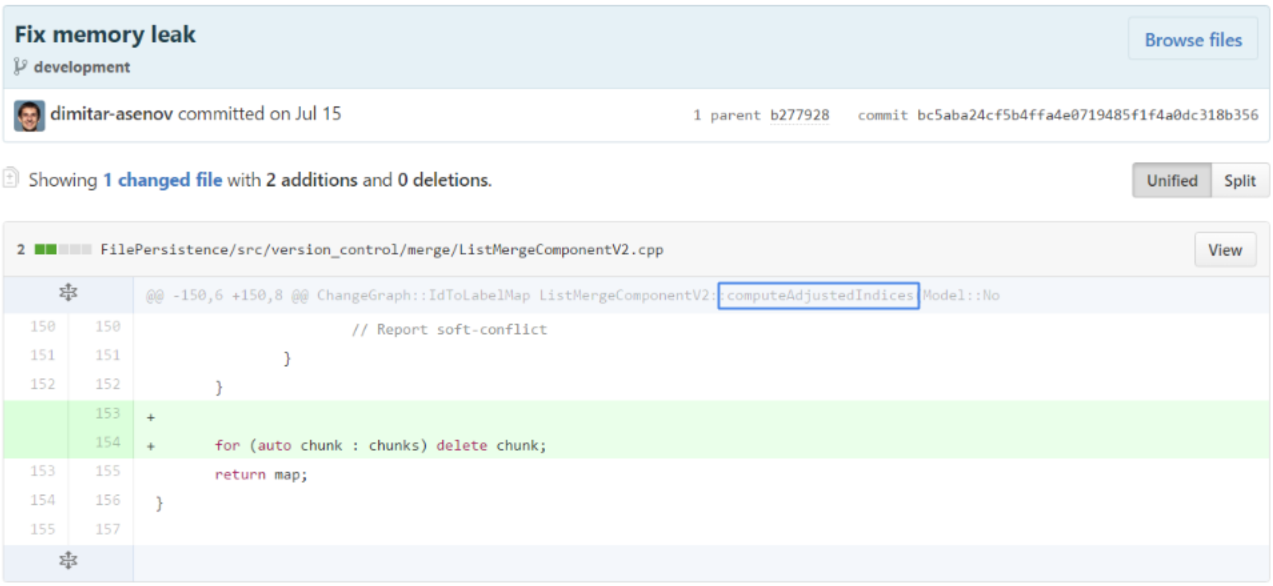
Figure 2.1: Imprecise textual matching between old version (top) and new version (bottom). Confusingly the variable $fileName$, indicated by the rectangle with the solid line, was matched with an occurrence of the same text in a comment of the new version. Although on a first glance it might seem that the entire code in the rectangle with the dashed line was removed, it actually was only moved inside a new if.

2.2.1 Initial design steps

The first iteration of the design process consisted of sketching visualizations for the different change types and exploring design options for various areas of a potential diff visualization. The initial design sketches served as a basis to discuss possible options and gave us an idea of possible visualizations. By analyzing the initial sketches and examining diff use-cases we extracted a number of design dimensions. To help iterating a variety of design options we explored answers to questions related to these design dimensions. In this section we present a selection of the more interesting questions and answers, as well as some of the rejected sketches of possible visualizations for different change types.

Q1: How many versions should be displayed?

A basic question to ask is how many versions of the software should be displayed. When comparing two versions of a software system, there are naturally a new and an old version, which could be displayed. If one considers existing solutions there are mainly two variants which seem to prevail: A unified view is showing one visualization of the code, which is a mixture between old and new version. A split view shows the old and new version of



Fix memory leak

development

dimitar-asenov committed on Jul 15   1 parent b277928   commit bc5aba24cf5b4ffa4e0719485f1f4a0dc318b356

Showing 1 changed file with 2 additions and 0 deletions.

2 FilePersistence/src/version_control/merge/ListMergeComponentV2.cpp

```
@@ -150,6 +150,8 @@ ChangeGraph::IdToLabelMap ListMergeComponentV2::computeAdjustedIndices(Model::No
                                // Report soft-conflict
                        }
                }
+
+               for (auto chunk : chunks) delete chunk;
                return map;
        }

```

Figure 2.2: Example for insufficient amount of context displayed per default in GitHub. As can be seen in the section marked by the rectangle, the deletion of chunks *does not take place in a destructor. The surrounding code would be required to get more information about* chunks *and to investigate why the deletion is done here.*

the code side by side.

- **Display one version:** Show all changes in one version.
- **Display two versions:** Shows both, the old and the new version.
- **Display 1.5 versions:** Show one version completely while showing only parts of the other.

Q2: How much context should be shown?

It is necessary to show some amount of context related to a change to get an idea what the surrounding code is and where that change is located in the software project. There is a trade-off between the amount of context to show and the conciseness of displaying the changes.

- **Show context according to code structure:** Use the structure of the code to determine the amount of context to show, e.g., if a change is located inside a method, show the content of the entire method as context.
- **Show everything:** Give as much context as possible.
- **Show only changed parts:** Show only changes, do not include any context.
- **Show surroundings:** Include limited context surrounding the change for the visualization. Similar to context presentation in, e.g., GitHub, which shows by default three lines of context surrounding the change.

Q3: How to make information stand out?

The attention of the user should be drawn to certain parts of the visualization. This is important to communicate, for example, the different change types in the diff.

- **Highlight information:** Using colored highlightnings is an effective way to focus the attention to certain parts of the visualization.
- **Blur other information:** Instead of using focus mechanisms on the information which should stay out, another approach would be to weaken the other parts of the visualization by, e.g., blurring them.
- **Center view on information:** Center the view of the visualization on an element in order to focus it without adding other visual elements to indicate its importance.
- **Visually group information:** Have visual separation, e.g., distance, between the important components and other elements.
- **Scale information up:** Have the important information drawn bigger than other parts of the visualization.
- **Annotate information:** Use visual elements to indicate important information. For example, use an arrow to point to important information.

Q4: How to display the change type of a component?

The diff is used to show the changes between different software versions to the user. Thus, it is important to find an intuitive and efficient way to distinguish the four possible change types (i.e., insertions, deletions, modifications and moves).

- **Use change type specific highlight:** Use highlights with specific colors to identify the change type of an element.
- **Use change type specific icon:** Display icon associated with change type next to the element.
- **Have a list of each change type:** Display a list which shows each change for a specific change type.
- **Show different views for each change type:** Display a separate view for each change type.

See also: Q3

Q5: How to display a deleted component?

This question is a specific instance of *Q4* for deleted components. Deleted components are only part of the old version of the code. This question only considers visualizations that we think are somehow inherently representative of deletions and not other change types.

- **Draw a line:** Use a line as a representation for a removed element.

- **Grey out the component:** Visually weaken the element to indicate removal.
- **Scale the component:** Shrink removed components.

Sketch:

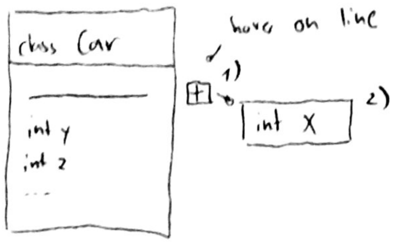


Figure 2.3: Sketch showing a change of type deletion. The deleted component is replaced with a line. By hovering over the line a box becomes visible which reveals the deleted component on click.

Figure 2.3 shows a possible visualization of a deletion change. The idea is to remove the deleted component from the visualization. Generally the user wants to know what exactly was deleted, so we added a mechanism to supply this information. Here the solution was to add interactivity to the design. Problematic in this approach is the amount of interactions needed by the user to see all the deletions in a big change set, which is the reason why this approach was rejected.

See also: Q3, Q4

Q6: How to display an inserted component?

This question is a specific instance of *Q4* for inserted components. Inserted components are only part of the new version of the code. This question only considers visualizations that we think are somehow inherently representative of insertions and not other change types.

- **Draw the component using bold stroke:** Use a bold stroke during drawing of the element.
- **Scale the component:** Display the inserted components in a bigger size than other components.

Sketch:

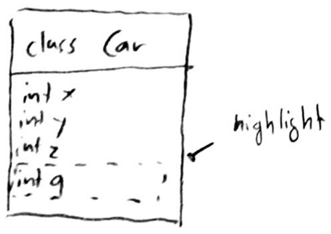


Figure 2.4: Sketch showing a change of type insertion. The inserted component is marked using a highlight.

The sketch shown in Figure 2.4 illustrates a possible visualization of an insertion change using highlights. If combined with colors this approach can aid the fast identification of insertions. This stems from the fact that recognition of a small number of colors is selective [8] and therefore easily identifiable for humans.

See also: Q3, Q4

Q7: How to display a moved component?

This question is a specific instance of *Q4* for moved components. Moved components relate code from the old version with code in the new version. This connection should be communicated. This question only considers visualizations that we think are somehow inherently representative of moves and not other change types.

- **Next to the new location:** Show a box next to the destination of the move which gives information about the origin of the element.
- **Using an animation:** Have an animation showing the move from the new to the old location or vice-versa.
- **Using an arrow:** Display an arrow which indicates the origin and destination of a move.
- **Using proximity:** Put the old and the new versions visually close.

Sketch:

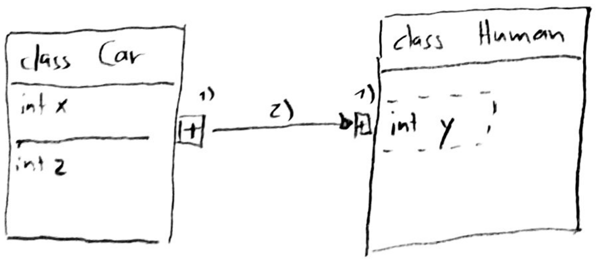


Figure 2.5: Sketch showing a change of type move. The visualization uses a combination of deletion and insertion to indicate a move. By hovering over the line for the deletion or the highlight for the insertion and clicking the appearing box a line is displayed. The line connects the origin and destination of the move.

In the sketch shown in Figure 2.5 a change of type move is visualized as a combination of a deletion and an insertion as described in Figures 2.3 and 2.4 respectively. This design was rejected due to the amount of actions needed to display the connection between old and new version of a change of type move.

See also: Q3, Q4, Q11

Q8: How to display a modified component?

This question is a specific instance of *Q4* for modified components. Similar to moved components, modified components also hold a relation from the old to the new version which should be communicated. This question only considers visualizations that we think are somehow inherently representative of modifications and not other change types.

- **Next to the original:** Show the modified element next to the original.
- **Next to the change:** Show the original element next to the modified element.
- **Only display it on request:** Only show the original element if the user requests seeing it.

Sketch:

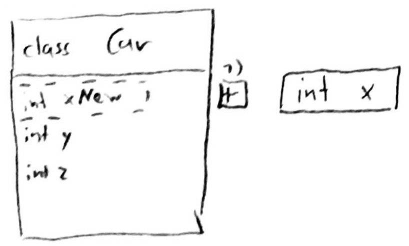


Figure 2.6: Sketch showing a change of type modification. The modified component is marked using a highlight. By hovering over the highlight and clicking the appearing box the original component is displayed.

Figure 2.6 shows how a modification change could be visualized. Like some of the other designs, this design was rejected because too many actions are required to display all information related to the modification.

See also: Q3, Q4, Q11

Q9: How many changes to display at once?

The question is of interest because the amount of changes which need to be shown in the diff can be very large. For example, if the user needs to compare revisions of a software system which have a big number of commits between them.

- **Display a single change:** Only show a single change at a time.
- **Display all changes:** Show all the changes at once.
- **Display related changes:** Show changes which stand in a relation to each other.
- **Display user selection of changes:** Let the user decide which changes should be shown.

See also: Q10, Q12

Q10: How to handle big change sets?

Given a big set of changes it is beneficial to not show everything at once to users. This helps in not overwhelming them with information related to changes and can provide some guidance. For example, code review tools like GitHub often use an alphabetically sorted list of the changed files to organize the changes.

- **Give overview of changes:** Have a zoomed out view, which only shows some information about the changes. If the user wants more information relating to a specific change, a zooming mechanism can be employed to zoom in on the specific change
- **Use a list:** Have a "list" similar to GitHub, with each of the changes below each other, only part of the changes are visible at a given time. Scrolling is used to see all the changes.
- **Define a sequence of changes:** Have the ability to step through the changes in a pre-defined order. Only show the change assigned to the current step.
- **Distribute changes to different views:** Find a way to distribute changes over multiple views instead of showing everything at once.

See also: Q9, Q12

Q11: How to express connection between changes?

Changes of type move and modification relate components of the old code to components of the new code. In order to help the user understand such changes this connection needs to be communicated.

- **Using arrows:** Have arrows joining the element from the old and new version to indicate the connection.
- **Using numbering:** Give each element of a connection group the same unique number.
- **Using color coding:** Use the same color for elements that have some connection.
- **Using distance:** Use spatial arrangement to have connected elements near each other.
- **Don't express the connection**

See also: Q7, Q8

Q12: How to arrange displayed components?

The displayed changes should be arranged in some way in order to give them a structure. This can help users to stay oriented and may provide guidance.

- **By stacking changes above each other:** Have the components arranged vertically above each other.

- **By arranging changes next to each other:** Have the components arranged horizontally next to each other.
- **By grouping changes:** Group the components according to some properties.
- **By showing them individually:** Show one change at a time and provide a mechanism to switch to the next after finishing inspection of the current one.

See also: Q9, Q10

2.2.2 Prototyping of selected possible solutions

In this section we present digital sketches of some possible visualizations using a selection of the design options explored during the previous section. If colors are used to specify change types we use the following convention, if not mentioned otherwise:

Green	$\longleftrightarrow$	Insertion
Red	$\longleftrightarrow$	Deletion
Yellow	$\longleftrightarrow$	Modification
Blue	$\longleftrightarrow$	Move

One version

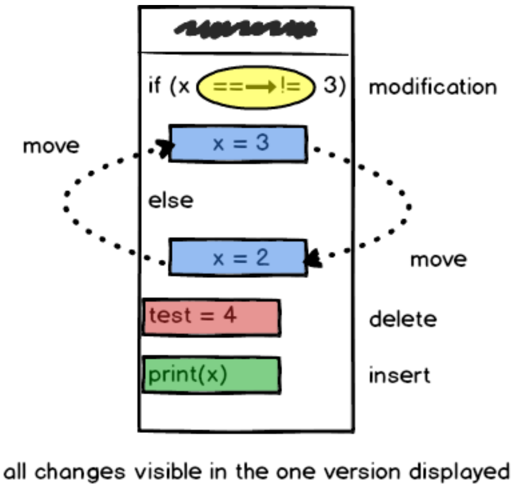


Figure 2.7: One-version prototype of a change diff of a single method. Uses only one version to display all changes. Uses highlights to mark changed components.

The prototype in Figure 2.7 displays one version of the code, which is a merge of the old and the new version of the software. The changes are highlighted using different colors to identify the change type. Modifications show the original and the modified entry next to each other in a highlighted box. Changes of type move use arrows to indicate origin and destination, while the component (e.g., class, method, etc.) which was moved remains at the origin. This prototype shows a concise way of showing all the changes. Using colored highlights helps in identifying the change types quickly [8]. A downside of this prototype is that it requires effort to read the old or the new version of the code and to compare them. For example, to read the new code moved components must be imagined at their destination, which can get difficult if the change is big or there are a lot of moved components.

One version using icons

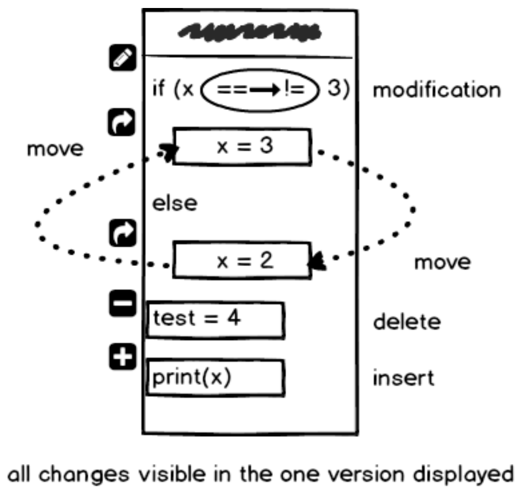


Figure 2.8: One-version prototype of a change diff of a single method using icons. Each change type features a specific icon.

Figure 2.8 shows a variant of the prototype *One version*. Instead of using colored highlights, this prototype uses icons to indicate change types. This avoids clutter which can be caused by various highlights on the code components. The prototype displays the user a familiar visualization of the code, except for the icons the code is displayed like during coding work. Similar to *One version* it is a concise way to show the changes, but it is not as easy to identify the different change types. Since it is a variant of *One version* it also requires effort in distinguishing the old and the new version of the code.

One and a half versions

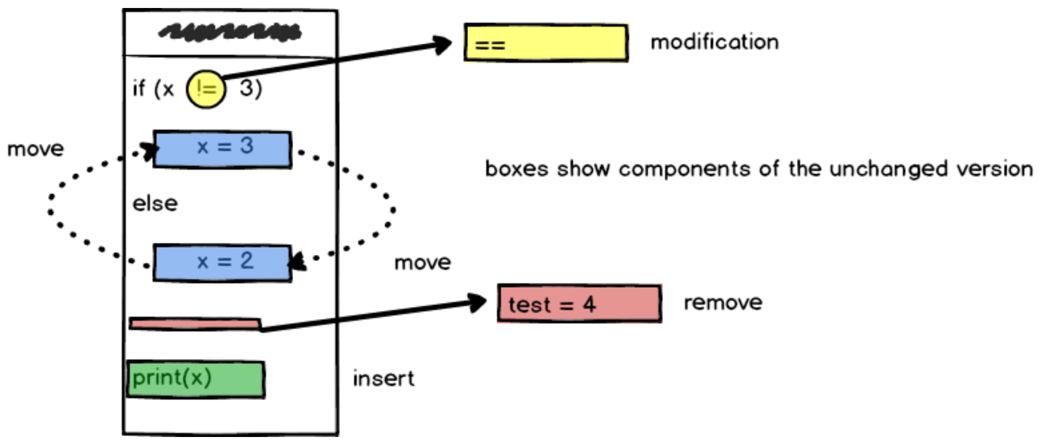


Figure 2.9: One-and-a-half-versions prototype of a change diff of a single method. The component shown on the left represents the new version of the code. The floating boxes on the right provide additional information about changes, for example, what the original component of a modification change was.

Instead of merging the two versions of the code the prototype in Figure 2.9 displays mainly the new version, the old version is only displayed in its changed parts, where additional information could be needed. Take as an example the modification change in the figure, the original component of the old version is presented on the side of the code version displayed. It is linked to the modification it belongs to with an arrow. Compared to *One version* it displays the moved component at its destination. This allows users to read the new version of the code by just looking at the displayed version. It is however more difficult to read the old version, since the information about it is displayed next to the new version displayed. A problem in this approach is, that it does not scale well with bigger changes. It would become increasingly difficult to arrange these boxes while still having a clear visualization of the information needed.

One version using decomposition

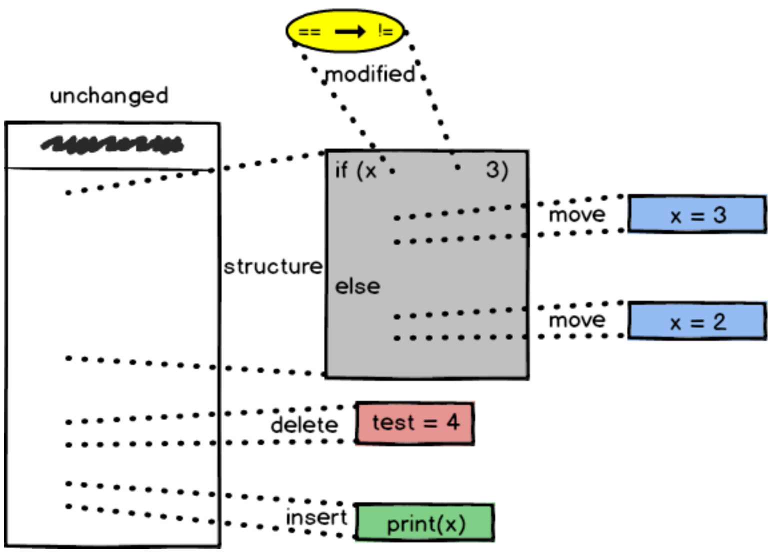


Figure 2.10: One-version prototype of a change diff of a single method using decomposition. The unchanged components remain in the version displayed on the left. The changed components on the other hand get extracted and are displayed by themselves using highlights. If the children of a component have changed, then this component will be highlighted in grey.

In the prototype shown in Figure 2.10 the changes are visually separated. In the left container all the unchanged elements are displayed. By dissecting the changes from the unchanged elements they can be analyzed in a hierarchic fashion. A change in the children of a component is indicated by coloring and highlighting it grey. Since there is the possibility that not everything of this component has changed, the unchanged parts are located in this grey box. The modification of the if-condition is extracted from this component and shown in a separate location. While visually interesting, this variant does not offer any clear advantages over the other prototypes.

One version with selection

The prototype in Figure 2.11 features interactivity. Initially the developer is presented with a list of all the changed components, bar any context. By selecting a change the element will be displayed in its appropriate context. Depending on the type of the selected change some additional information may be displayed. In the sketched example the developer selects the change of type move $x = 2$, this shows the component with context and since the change is of type move an additional arrow which indicates the origin of the selected component is displayed. While the initial step can provide an

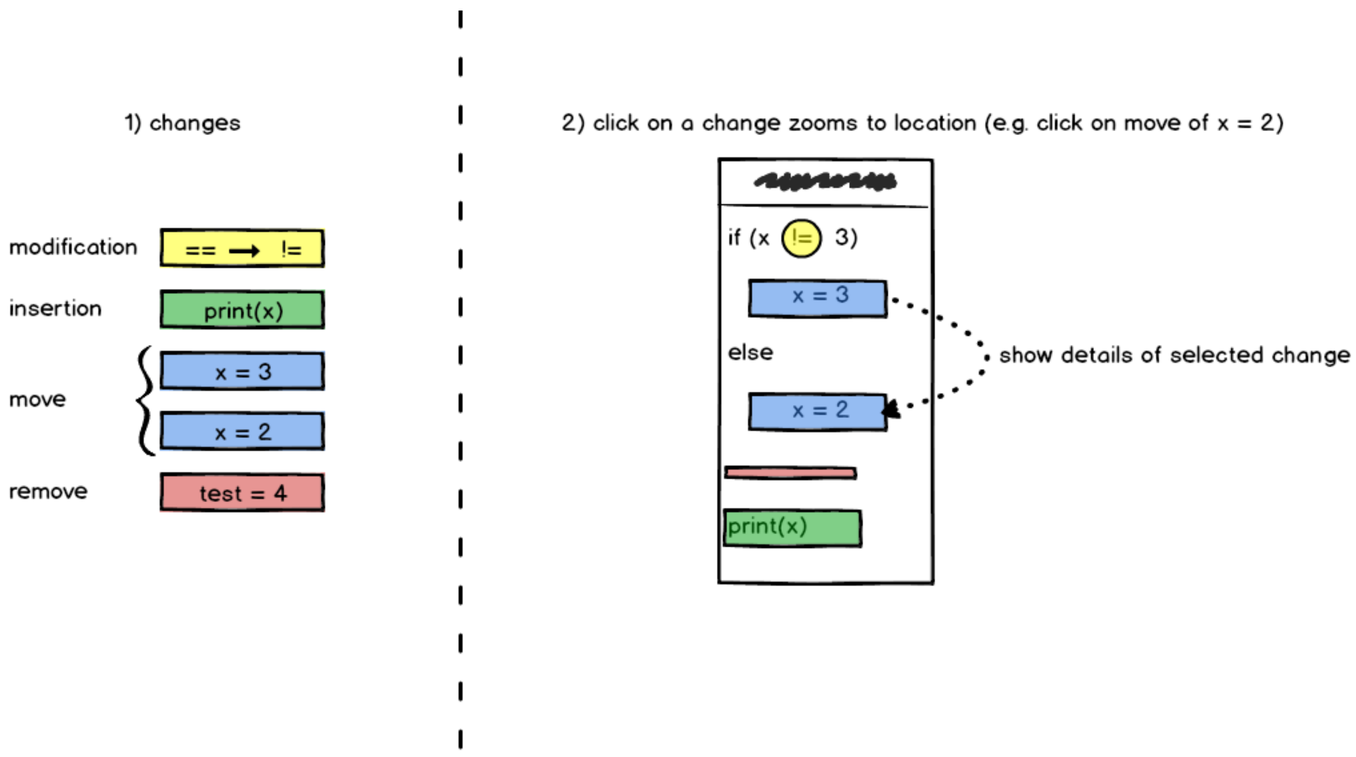


Figure 2.11: One-version prototype of a change diff of a single method using selection. In the initial step displayed on the left all the changes are listed without any context. Users can then select a change to get to the visualization of the change with additional context. In this figure the change $x = 2$ was selected by a user.

overview of the changes the approach in general is cumbersome for users. To inspect the changes users have to individually select a change that they are interested in, in order to continue to the visualization of the change with context. Because usually every change has to be inspected this leads to a large number of actions required by the user to see all the available information.

Two versions

The prototype in Figure 2.12 displays the old and the new version of the software side-by-side. Highlights and arrows are used similarly to the *one version* prototype. The arrow of a move now spans between the two versions, connecting the origin and destination of the move. Displaying of modifications is now split as well between the two versions, i.e., the original component is in the old version and the modification in the new one. The two components of the modification are, like for changes of type move, connected with an arrow. The separation between the two versions allows users to read the old or the new version of the code without additional effort. While the information displayed is not as dense as in, for example, *One version*, it features better readability of the different versions.

Two version variant

Figure 2.13 shows a variant of the *Two versions* prototype. It uses the old version displayed mainly as a reference to how the code looked before the change and to mark deletions. All other change types are solely displayed in the new version. The origin

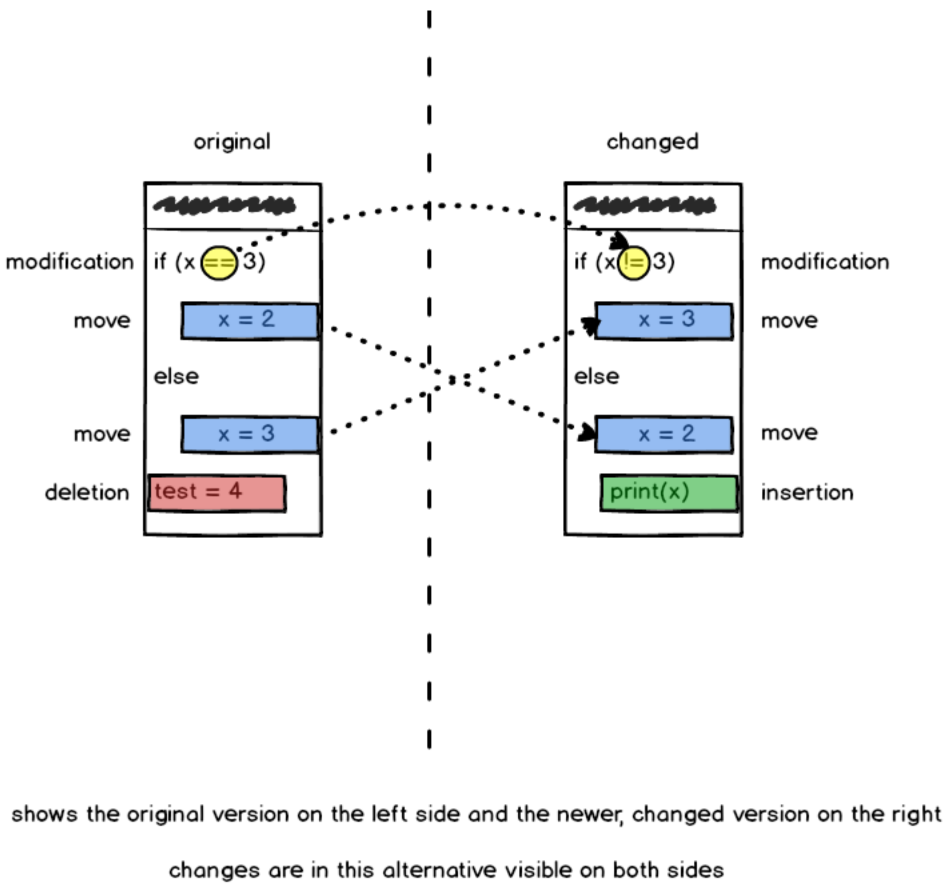

original
changed
modification
if (x == 3)
if (x != 3)
modification
move
x = 2
x = 3
move
else
else
move
x = 3
x = 2
move
deletion
test = 4
print(x)
insertion
shows the original version on the left side and the newer, changed version on the right
changes are in this alternative visible on both sides

Figure 2.12: Two-versions prototype of a change diff of a single method. The old version of the code is displayed on the left, while the new version of the code is displayed on the right. Arrows connect changes of type move and modification to relate the origin and destination of a move or original and modified component of a modification.

of a move is displayed in the new version itself. Changes of type modification are only highlighted in the new version. This variant still features the advantage of having clear separation between the old and the new version of the code. The relation between the old and the new version of the code for changes of type modification and move are not as clear as in *Two versions.*

Two versions using decomposition

The prototype shown in Figure 2.14 is similar to the *One version using decomposition* prototype. The old version uses the same highlights as the decomposed new version. Different move changes are highlighted using a unique color to facilitate the matching of origin and destination for each of the different moves. Like for *One version using decomposition* there is no clear advantage in extracting the changes and displaying them in the visualized way.

2.3 Evaluating candidates for the final design

After analyzing the prototypes described in the last section we excluded the following prototypes:

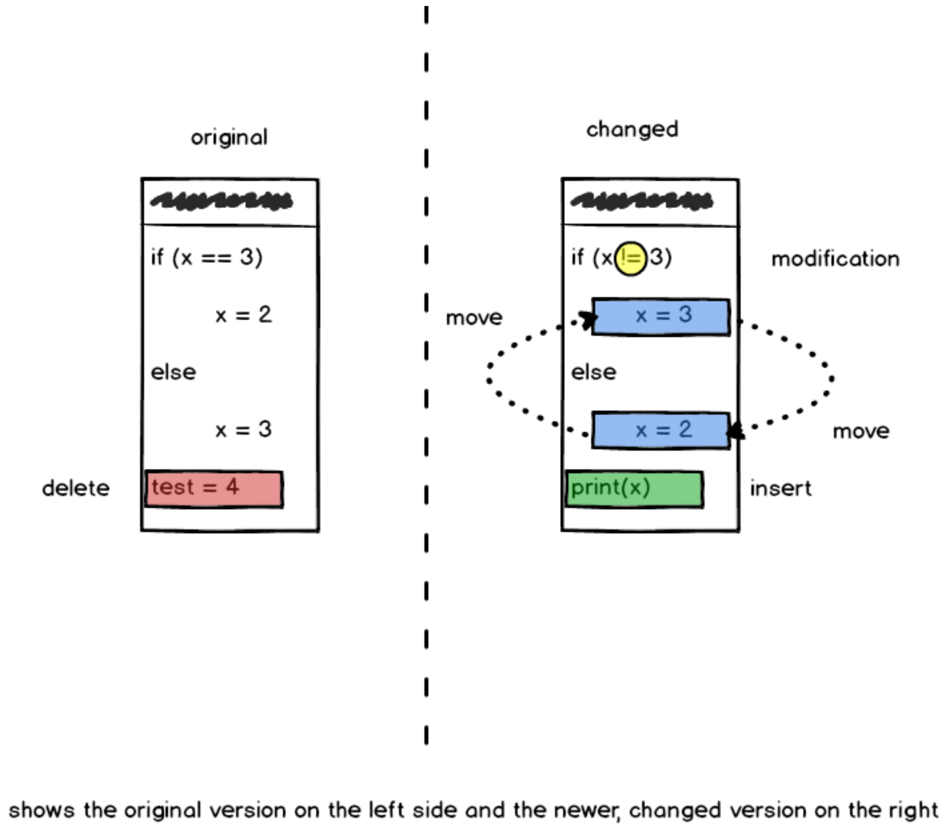


Figure 2.13: Two-versions variant prototype of a change diff of a single method. The old version of the code is displayed on the left, while the new version of the code is displayed on the right. In the old version only deletions are highlighted. The origin of a move is indicated in the new version itself. Modifications are only highlighted in the new version.

- **One and a half versions:** It would become difficult to arrange the visualizations for big change sets.
- **One version using decomposition:** Provides no clear advantage over the other prototypes.
- **One version with selection:** Too many actions required by developers to see all available information.
- **Two version variant:** Relations for changes of type move and modification not as clearly visible as in *Two versions*.
- **Two versions using decomposition:** Like *One version using decomposition* it provides no clear advantage over other prototypes.

To come to a decision, which approach should be selected as a basis for the design and implementation in Envision we evaluate the remaining approaches using a mixture of parts of the *cognitive dimensions* as described by Green and Petre [14] and our own criteria. The results of the evaluation are detailed in table 2.1.

We could not determine a clearly advantageous design between one version or two versions with highlights. We decided to use a two version approach for the design in Envision

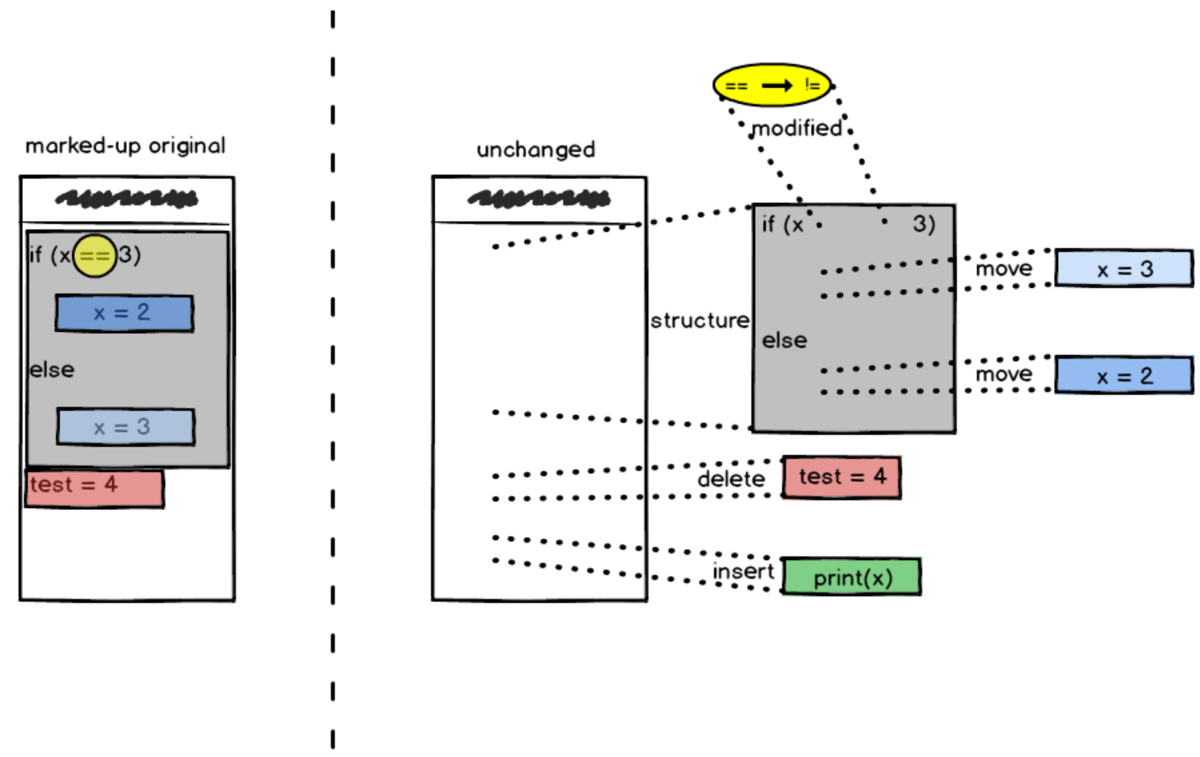


Figure 2.14: Two-versions prototype of a change diff of a single method using decomposition. The old version of the code is displayed on the left, while the new version with the changes extracted is displayed on the right. The old version shows the same highlights like the new version except for insertions. Decomposition is handled the same way like in the One version using decomposition *prototype.*

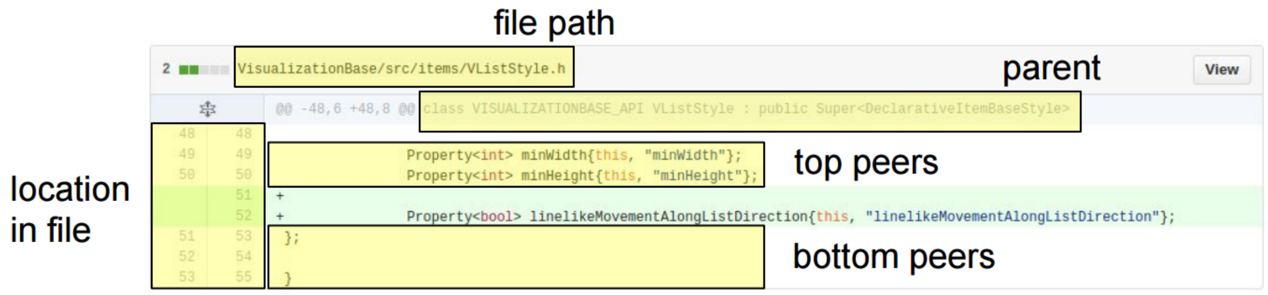
because it avoids the need of merging two versions of code and is therefore more feasible to implement in the current state of Envision.

	1 version icons	**1 version highlights**	**2 versions highlights**
Visual variety	Lacking, icons would need different colors to feature greater visual variety.	Both designs offer visual variety thanks to colors used for the highlights. Could be improved by using different shapes for the highlights.	
Information density	The single version approach offers higher information density, since the old and new version are merged and displayed together.		Having the two versions split and shown separately leads to lower information density.
Clutter	The only visualizations added to the code are the icons displayed next to the changed components. This leads to minimal additional clutter.	The move change arrows which point to different components, as well as the modification highlights containing information about the original and modified component increase clutter.	Similar to *1 version highlights* but less clutter in since information is distributed over the two displayed versions.
Readability of versions	Hard to read the old or the new version by itself. The icons indicate to which version a change belongs.	Colored highlights indicate which change belongs to which version of the code. To read the new version the destination of moved components needs to be considered by following the arrows.	Old version located on one side, new version on the other. This distinction offers the best readability.
Diffuseness	Every change type needs its own icon or graphic.	Since colored highlights are used, only different colors have to be chosen.	
Dependencies	Moves and Modification would need better displaying of their dependencies.	Clear dependencies for modifications. Distinguishing origin and destination of moves could be improved.	Arrows connecting related components for modifications and moves clearly show dependencies.
Role expressiveness	User needs to scan the icons to identify the change type.	Selectiveness of colors leads to strong role expressiveness.	

Table 2.1: Comparing candidates for design in Envision. Cells highlighted in grey indicate the best option for a specific criteria.

2.4 Presenting context for changes

An important decision to make is how to present the context for a change, i.e., how much context should be provided for a change. If we consider how existing solutions like GitHub present context we mostly see the approach that a block which has changed is surrounded by a few extra lines of stable code. If developers needs more context they can repeatedly click the appropriate button to iteratively add more lines to the context of the change.



```
2 VisualizationBase/src/items/VListStyle.h
@@ -48,6 +48,8 @@ class VISUALIZATIONBASE_API VListStyle : public Super<DeclarativeItemBaseStyle>
48 48
49 49             Property<int> minWidth{this, "minWidth"};
50 50             Property<int> minHeight{this, "minHeight"};
   51  +
   52  +           Property<bool> linelikeMovementAlongListDirection{this, "linelikeMovementAlongListDirection"};
51 53  };
52 54
53 55  }
```

Figure 2.15: Context as it is presented in GitHub. The file path details, in what file the change took place and where it is located in the software project. In the parent box, we can see which parent component (e.g., class, method, etc.) contains the displayed changes. The top and bottom peers show surrounding code of the change. The numbers on the left show, where in the file the change is located.

Besides the neighboring lines, called peers in Figure 2.15, there is also the path to the file as well as the parent component of the change indicated. The line numbers should help the developer to get an idea where in the file the change is located.

It would be sensible to improve context presentation and think about better ways to provide developers with context for changes. There exist approaches which use data and control dependencies of changes to define the context useful for a change [26]. For our approach of context presentation we considered different options:

- **Reaching definition analysis:** One option would be to include into the context everything which is obtained by a reaching definitions analysis for the variables affected by the change. The idea is to encompass everything which somehow is connected to the affected variables.
- **Expand to next component:** This approach uses the structure inherent to the components to present context. Assuming a change resides inside a method, then in the first step the whole method is displayed as context. If more context is needed it can be expanded to the next component which in this case may be, e.g., a class. Now the class is shown in its entirety. This enables a more structural approach to expanding context than showing an additional arbitrary amount of code lines.
- **Scaling:** In this more visual approach important components, i.e., the changed components, are highlighted by scaling them up. Figure 2.16 shows an illustration of this approach. In the default view only the scaled-up components are clearly visible. If more context is needed, the developer can use a zooming mechanism to zoom in on the changes which gradually delivers more context. At first only high

level components, e.g., modules, may become visible but the higher the zoom level the more details are visible.

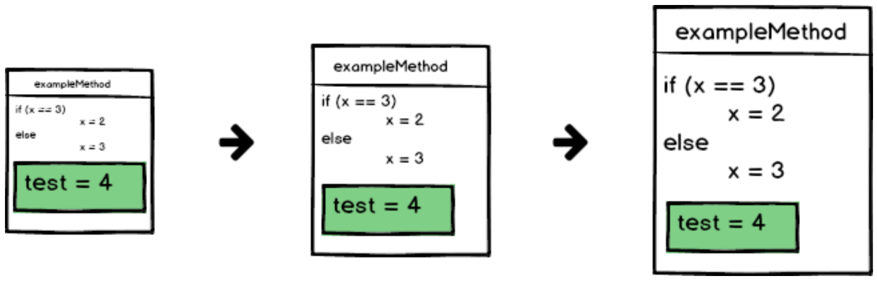


Figure 2.16: Highlighting changes by scaling them up. The insertion test $= 4$ *is scaled-up which leads to it being the only code fragment clearly visible if the view is zoomed out. By zooming in gradually more of the context becomes visible until everything is scaled normally and the change is distinguishable from unchanged components by its highlight.*

We decided to experiment with a design which uses a mixture of the *Expand to next component* and *Scaling* approaches to present context. We define the amount of context per change by showing a component of a specific type (e.g., class, method, etc.) which is a parent of the changed component or the component itself. We then use the scaling approach to show changed components clearly in a zoomed-out view and provide the context by zooming-in on a change. We discarded the *Reaching definition analysis* approach because if a changed variable is used in many different places (e.g., different classes) the amount of code which would be considered context could get very big and difficult to manage.

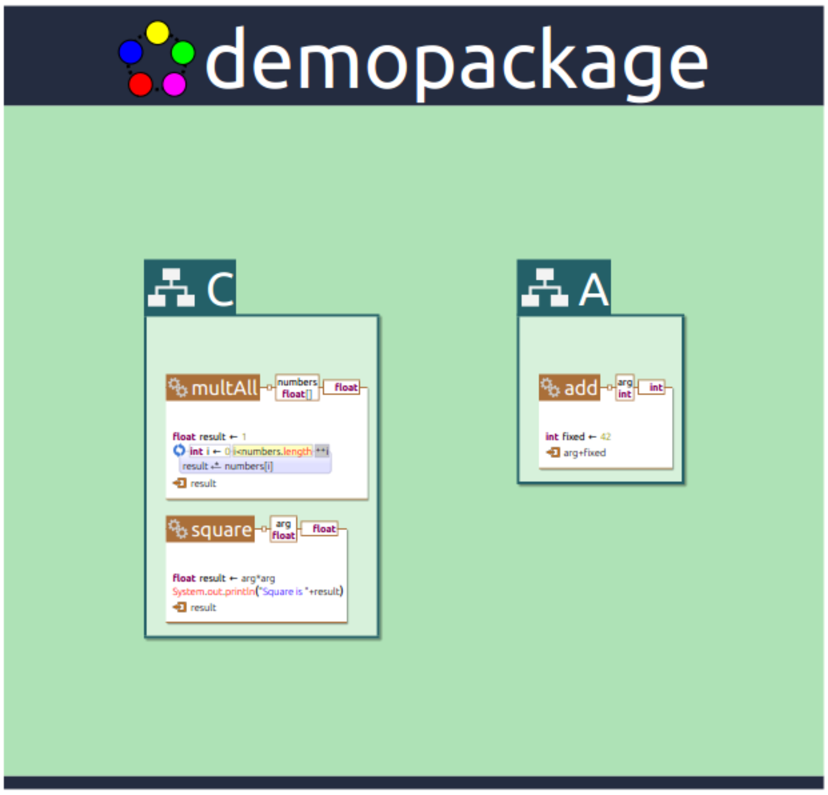
2.5 Final design in Envision

2.5.1 Diff visualization in Envision

In this section we present the final design of the visualization for the diff. We implemented this design in Envision. To illustrate the design we use the example project in Figure 2.17.

The project view displayed in Figure 2.17 shows the latest revision of the *DiffExample* project, which contains a package called *demopackage* and two classes *C* and *A*. We execute the `diff` command to compare the differences between the initial revision *6caf* of the project and the current revision *a8c9*. The `diff` command provides a list of possible commits and shows an abbreviated version of the sha1 hashes, i.e., only as many characters as are needed to avoid ambiguity between commits and to still fulfill the minimum length requirement by git. This enables a more concise and readable identification of the commits. The commit message displayed provides the description for each of the commits and allows developers to get an idea what the content of a specific commit is.

We execute the `diff` command shown on the right side in Figure 2.18. This switches to a diff view, which is shown in Figure 2.19. This view shows three grey containers called *DiffFrames*. The amount of context shown per change can be configured by the developer

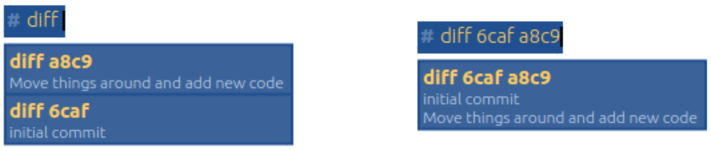


```
float result ← 1
int i ← 0  i<numbers.length  ++i
    result *= numbers[i]
result
```

```
float result ← arg*arg
System.out.println("Square is "+result)
result
```

```
int fixed ← 42
arg+fixed
```

Figure 2.17: Example project with a module demopackage which contains two classes C and A.



```
# diff
diff a8c9
Move things around and add new code
diff 6caf
initial commit
```

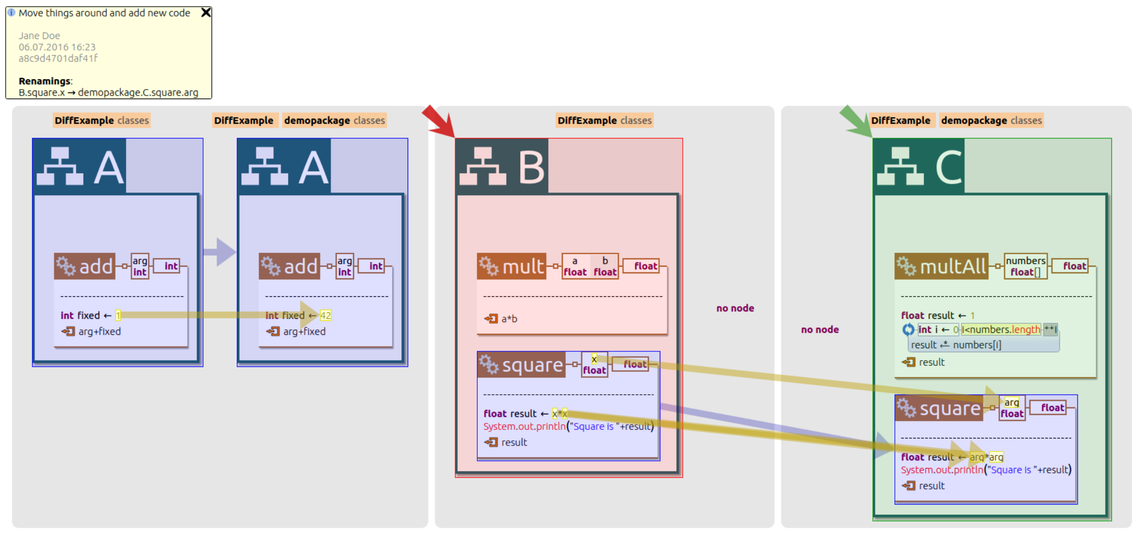
```
# diff 6caf a8c9
diff 6caf a8c9
initial commit
Move things around and add new code
```

Figure 2.18: The `diff` command providing auto-completion and information about commits. The command shows possible commit ids and the related commit messages on Envision's standard command prompt.

(see section 2.5.2 for more details). In this example the context level is set to *method*. This means that for a change the first parent method encountered by walking up the AST will be displayed. If there is no such method, only the component (e.g., class, method, etc.) itself will be displayed. If there are two changes to the same method, then they will be displayed together in the same *DiffFrame*.

To keep the number of displayed *DiffFrames* as small as possible and avoid showing components in multiple *DiffFrames* at once we implemented a mechanism that puts changed components in the *DiffFrame* of, if available, a changed component located further up in the AST. Consider, for example, in Figure 2.20, the change modifying the value of variable *fixed*: If class *A* would not have been moved, a *DiffFrame* showing method *add* as context for the change would be displayed. But since a *DiffFrame* showing class *A* (and therefore also method *add*) is available, the change is instead displayed in this *DiffFrame*.

The top left corner of the diff view shown in Figure 2.19 shows meta information and provides a summary of all name changes part of the current diff. The summary of renam-


Move things around and add new code

Jane Doe
06.07.2016 16:23
a8c9d4701daf41f

Renamings:
B.square.x → demopackage.C.square.arg

Figure 2.19: *Diff view in Envision. In the top left corner of the view a box with additional information about the commit is displayed. The three grey containers represent individual DiffFrames. Each DiffFrame shows the appropriate context for the changed components (e.g., classes, methods, etc.) it is displaying. Changed components are marked using colored highlights.*

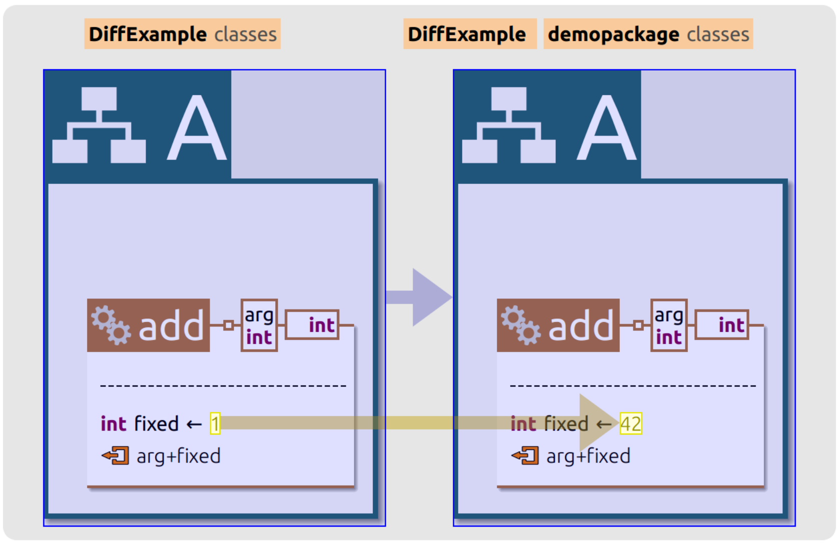
ings is a feature which other code review tools lack [19]. The information box consists, from top to bottom, of:

- commit message
- commit author
- commit sha1 id
- Renamings (if available)

Figure 2.20 shows the *DiffFrame* of class *A*. The top of the *DiffFrame* features the object path to the displayed component. The object path allows developers to see, where in the context of the project the displayed component is located. The left side of the *DiffFrame* contains the old version of the component, while the right side holds the new version. Since class *A* is highlighted in blue it is indicated that the class was moved. Because the whole class was moved, we have to consider the object path in the top of the frame for more information. It states that in the old version the class *A* was located directly in the project *DiffExample*, while in the new version it is located in *DiffExample.demopackage*. The only other change to class *A* was a modification of the value of the integer *fixed* in the *add* method. The figure also shows the use of arrows for modifications (yellow) and moves (blue). They help to show the relation between original and modifications, as well as origin and destination.

By hovering over such an arrow with the mouse, the arrow gets drawn with a bolder stroke. This helps in distinguishing the selected arrow from other arrows in the view. Figure 2.21 shows this mechanism in an example.

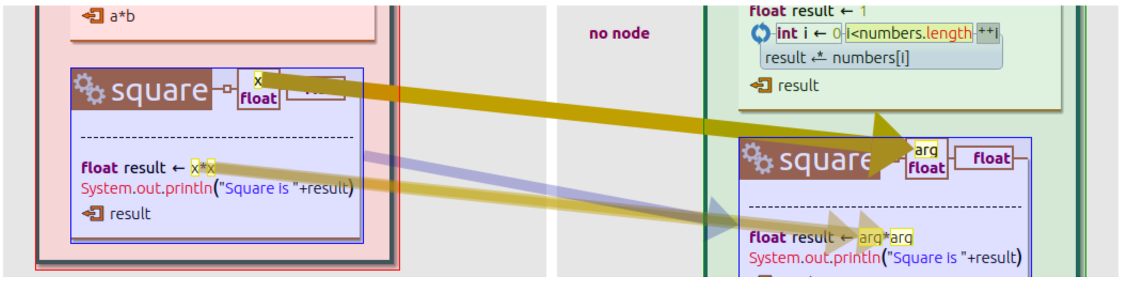
Figure 2.22 shows the *DiffFrames* of class *B* and *C*. The entire class *B* is highlighted in red which indicates that the class was removed. This is also the reason why in the



```
int fixed ← 1
arg+fixed
```

```
int fixed ← 42
arg+fixed
```

Figure 2.20: *DiffFrame showing two changes related to class A. First, the entire class A is highlighted in blue, which means that it was moved. The object path in the top reveals that class A was moved into the demopackage module. Second, the value of fixed is highlighted in yellow which indicates a modification.*



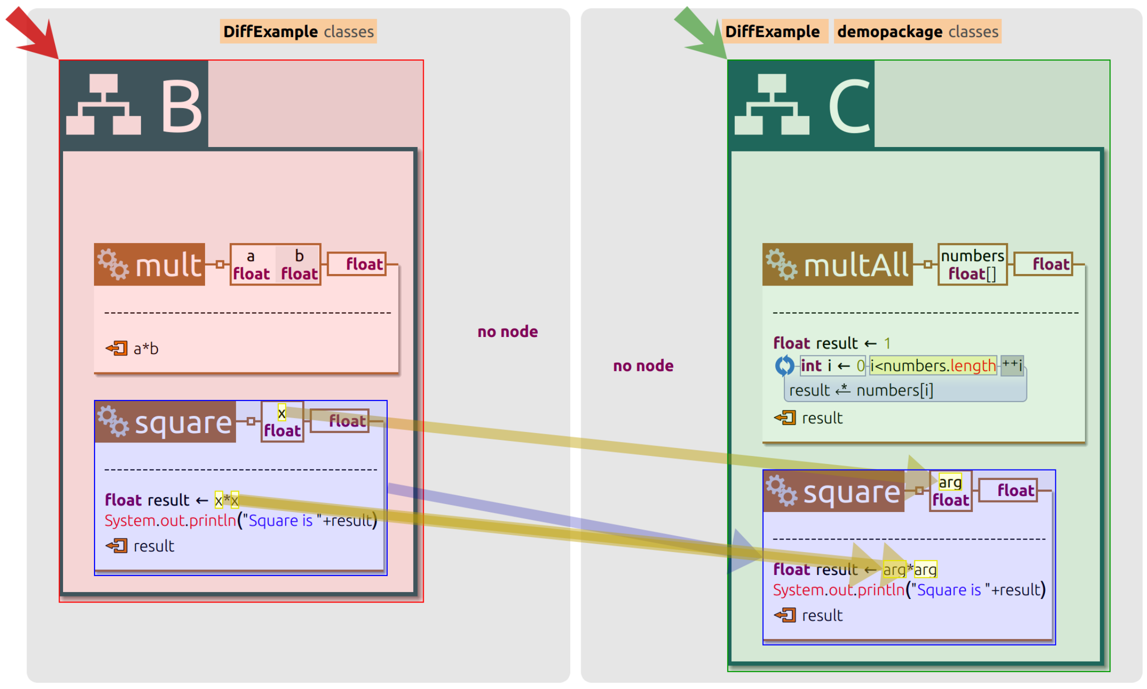
```
a*b
float result ← x*x
System.out.println("Square is "+result)
result
```

```
float result ← 1
int i ← 0 i<numbers.length ++i
result += numbers[i]
result
float result ← arg*arg
System.out.println("Square is "+result)
```

Figure 2.21: *Example of a highlighted modification arrow. The arrow connecting the modification of the name change from x to arg was highlighted by hovering over it with the mouse cursor.*

DiffFrame showing class *B* no new version of the component is shown. The opposite is the case in the *DiffFrame* showing class *C*. Class *C* was newly added in this commit and therefore no old version of class *C* is available. To help identifying insertions and deletions more efficiently, we added *indicator arrows* to the design. They are located in the top left corner of the highlight of a change. Since moves and modifications are easily identifiable because of their relationship arrows, they feature no such *indicator arrows*. Other than the two changes affecting the entire classes, we see a blue arrow which indicates that the method *square* of class *B* was moved into the new class *C*. Further, there is a yellow arrow which indicates a modification change. In this example it is a renaming of the argument from *x* to *arg* in the *square* method. From the bread crumb bar which represents the object path we gather the information, that class *B* was located in *DiffExample*, while class *C* was inserted into *DiffExample.demopackage*.

Thanks to the fine-grained diff changes additional information (i.e., moves and modifications) can be shown to developers, which relieves them from the task of manually identifying changes of those types. This precise diff can help developers to effectively understand changes [25].



```
mult  a float  b float  float
a*b

square  x float  float
float result ← x*x
System.out.println("Square is "+result)
result
```

```
multAll  numbers float[]  float
float result ← 1
int i ← 0  i<numbers.length  ++i
    result *= numbers[i]
result

square  arg float  float
float result ← arg*arg
System.out.println("Square is "+result)
result
```

Figure 2.22: Two DiffFrames showing the changes to classes B and C. The method square is highlighted in blue, which indicates a move change. Following its arrow, we can see that it was moved from the deleted class B in the old version to the newly created class C. The yellow modification changes highlight the renaming of the argument x to arg.

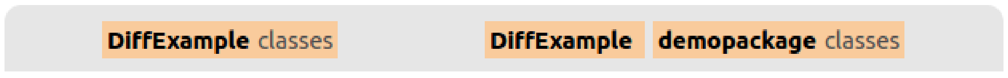


Figure 2.23: Object path of a DiffFrame. The left side shows the object path of the old version. On the right side the object path of the new version is displayed. Each breadcrumb of the object path features interactivity: by clicking on it, the appropriate component is displayed next to the DiffFrame.

The object path, as shown in Figure 2.23, of each *DiffFrame* offers interactivity. The path consists of individual breadcrumbs, by clicking the breadcrumbs it is possible to open the respective component next to the *DiffFrame*. This allows developers to quickly inspect other components of the project, which may not be shown in the diff view by default. Such a feature which allows developers to navigate and explore the code base was requested during surveys [18, 23].

2.5.2 Presenting context in Envision

In this section we discuss how we present context in Envision. In the diff view we avoid showing the entire project, but instead use the available space to focus on the changes done to the project. Since context is important for change understanding [3, 18, 23] we provide different mechanisms to access more context if needed.

Context units

The first step in presenting context is done by configuring context units. The developer can decide which components (e.g., class, method, etc.) of a project should be used as a "frame" to provide context to the changes. This allows developers to define the level of context they are accustomed to work with. During development we used *method* as a context unit. This has the effect that for every change the closest enclosing component (including the changed component itself) of type method is displayed, if available. It is possible to define a list of such context units with decreasing priorities, e.g., first priority is to use and display method, if this is not possible try displaying an ancestor class, etc. If a changed component is a child of a changed parent component, then the changed component will be displayed in the same context as its parent component. See for example Figure 2.20 where the modified component is displayed within their classes which was moved.

Providing the entire component as context for a change, removes the need to iteratively request more lines of context like in other code review tools as, e.g., GitHub.

Scaling of changes

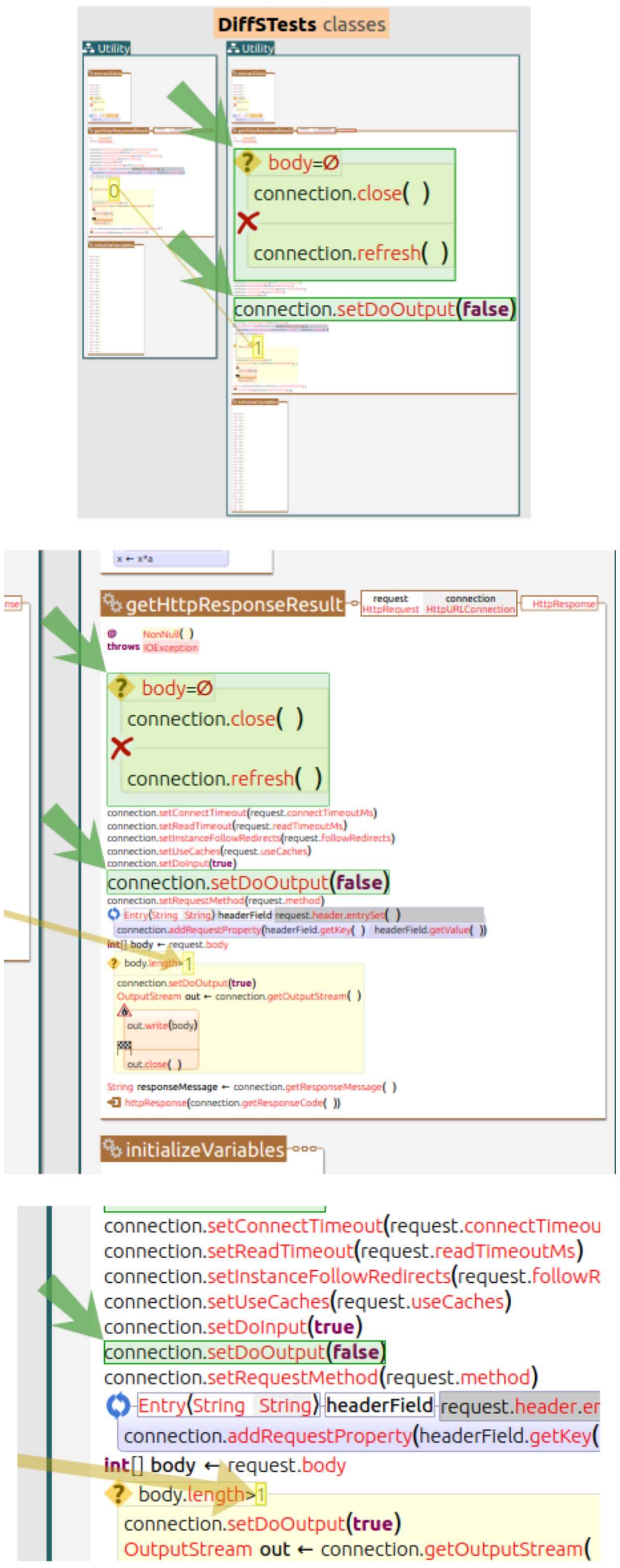
Using the discussed context units, sometimes a lot of context is already displayed in the diff view. This makes a mechanism to clearly indicate the actual changes a necessity. Inside a *DiffFrame* the changed components are scaled differently during a zoom compared to the unchanged components, leading to the effect, that in a zoomed-out view only changes are clearly visible. This helps in identifying all the changes and get an overview over the single changes in the different *DiffFrames*. If developers needs more context for a specific change they can move the mouse cursor over the appropriate change and zoom in. Gradually the unchanged components become visible. On the nearest zoom levels the scaling of the changed and unchanged components is the same and it is possible to read the code with the highlighted changes normally. Figure 2.24 shows the behavior on an example.

Adding additional context

It is possible to display more context by adding components which are not shown in the diff view by default. One way to add components is to click on the component names in the object path. This displays the respective component in the diff view. Another way to add components to the diff view is using Envision's command prompt, which gives the possibility to add components by name or, e.g., show all callers of a method. This enables IDE-like navigation during the diff view, which can improve change understanding and was requested by developers during surveys [18, 23, 25].

2.6 Advanced features

In this section we introduce advanced features of the diff, which aid the developer in the task of understanding code changes.



```
body=∅
    connection.close( )
    connection.refresh( )
connection.setDoOutput(false)
```

```
x ← x*a

getHttpResponseResult   request HttpRequest   connection HttpURLConnection   HttpResponse
@ NonNull( )
throws IOException
body=∅
    connection.close( )
    connection.refresh( )
connection.setConnectTimeout(request.connectTimeoutMs)
connection.setReadTimeout(request.readTimeoutMs)
connection.setInstanceFollowRedirects(request.followRedirects)
connection.setUseCaches(request.useCaches)
connection.setDoInput(true)
connection.setDoOutput(false)
connection.setRequestMethod(request.method)
Entry(String String) headerField request.header.entrySet( )
    connection.addRequestProperty(headerField.getKey( ) headerField.getValue( ))
int[] body ← request.body
body.length>1
    connection.setDoOutput(true)
    OutputStream out ← connection.getOutputStream( )
        out.write(body)
        out.close( )
String responseMessage ← connection.getResponseMessage( )
httpResponse(connection.getResponseCode( ))

initializeVariables
```

```
connection.setConnectTimeout(request.connectTimeou
connection.setReadTimeout(request.readTimeoutMs)
connection.setInstanceFollowRedirects(request.followR
connection.setUseCaches(request.useCaches)
connection.setDoInput(true)
connection.setDoOutput(false)
connection.setRequestMethod(request.method)
Entry(String String) headerField request.header.er
    connection.addRequestProperty(headerField.getKey(
int[] body ← request.body
body.length>1
    connection.setDoOutput(true)
    OutputStream out ← connection.getOutputStream(
```

Figure 2.24: Zooming in on a single inserted statement. In the top the only components clearly visible are the changed components. By hovering over a change and zooming in more information becomes visible

2.6.1 History of a component

To help developers understand the evolution of a specific component (e.g., class, method, etc.) in a software project we designed the history feature. By selecting any AST node (e.g., a method or an individual statement) in the project view and executing the `history` command the developer will be presented with a “film strip” view of the evolution of the selected node. A similar feature was requested by developers during a recent survey [18]. To illustrate this feature we will use the example project shown in Figure 2.25.

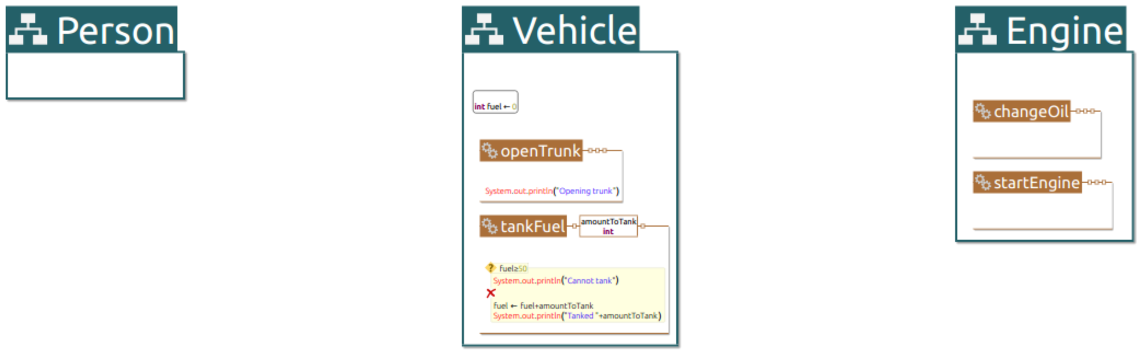


Figure 2.25: Example project for the `history` *command. The project features three classes Person, Vehicle and Engine. In this example we are interested in the Vehicle component.*

By selecting the class *Vehicle* and executing the `history` command, we are able to see the complete evolution of this class.

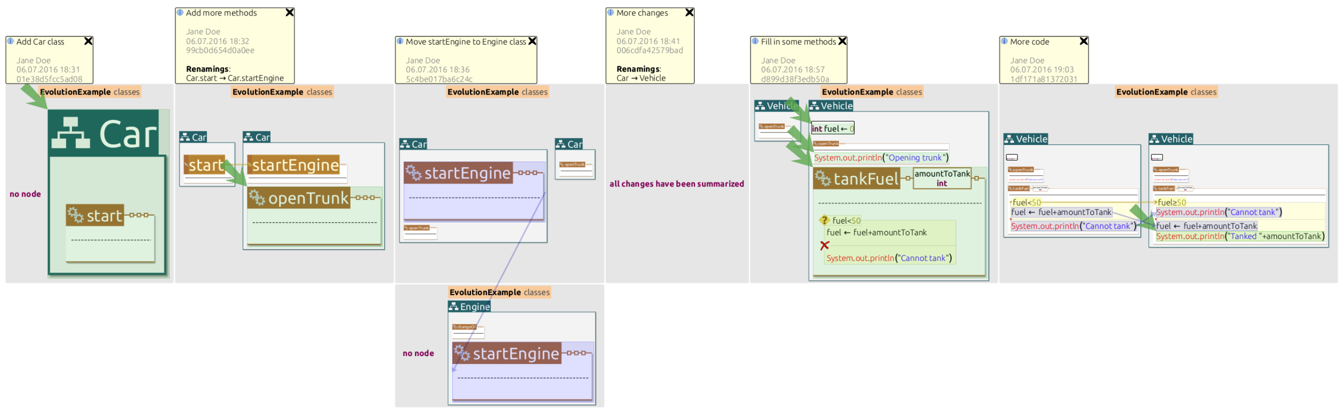


Figure 2.26: History view in Envision. Displayed is the evolution of the Vehicle component shown in Fig. 2.25. The history starts on the left side with the creation of the component and continues to the right to the current version of the Vehicle component. Each DiffFrame represents a commit involved in the shaping of the Vehicle component.

Figure 2.26 shows the history view, detailing the evolution of the *Vehicle* component. This view shows what changes were applied to the specified component and in which commits. Like in the normal diff view, changes appear bigger when the view is zoomed out.

In Figure 2.27 we can see a snippet of the entire history view. To display the target of the move change, an additional *DiffFrame* is displayed to enable the drawing of the move arrow. The next change features an empty *DiffFrame*. This happened, because all of the changes were name changes, which were summarized by the name change summary feature. The renaming is declared in the info box on top of the *DiffFrame*.

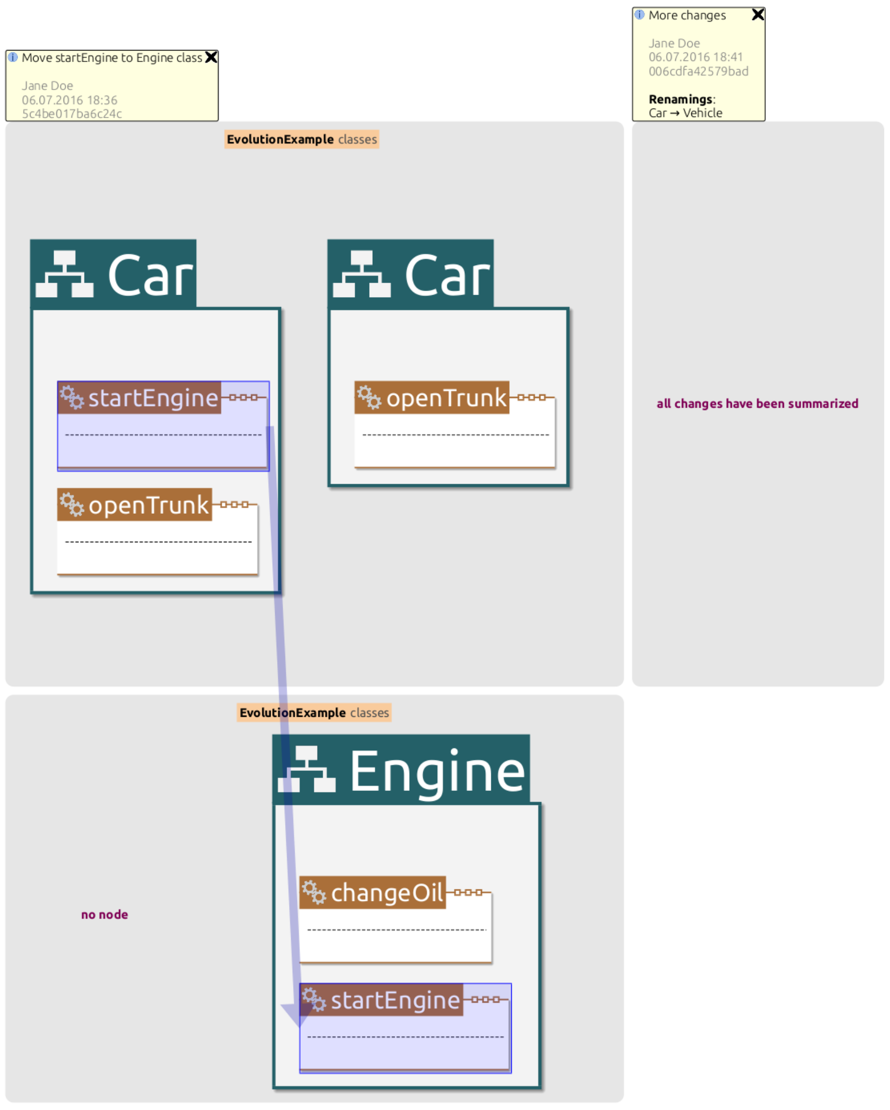


Figure 2.27: Snippet of history view. Displayed are three DiffFrames of the evolution of the Vehicle *component. The two DiffFrames located on top of each other show the move of method startEngine. To be able to display the destination, which is in another class, the additional DiffFrame in the bottom is shown. In the DiffFrame on the right all changes were summarized and nothing is shown. The summary about the change is visible in the DiffFrames information box.*

The history view snippet displayed in Figure 2.28 illustrates how the developer can inspect the evolution of the component. In the first *DiffFrame* we can see, that in the old version only the *openTrunk* method existed. The new version added some statements and a new method *tankFuel*. By looking at the next *DiffFrame* we can follow the evolution of the component. The component displayed in the new version of the last *DiffFrame* is the same, which is displayed in the old version of the current *DiffFrame*. An alternative would be to show each component only once, which would lead to too much clutter because of the amount of change highlights which need to be displayed. This is the reason why we instead decided to show two identical components next to each other.

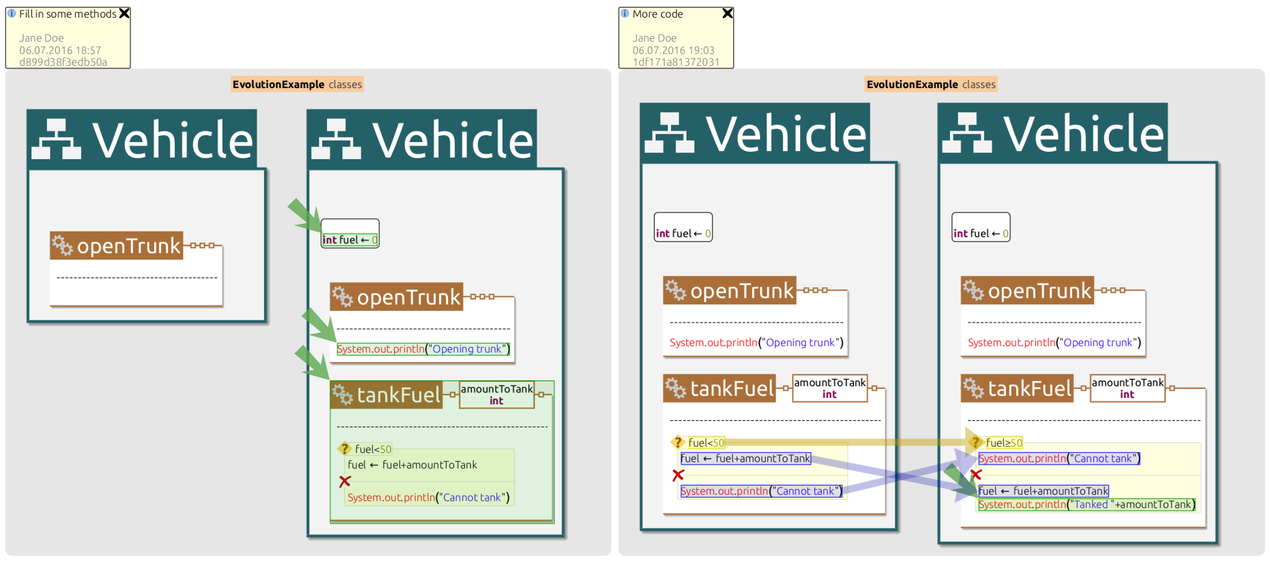


Figure 2.28: *Different snippet of history view. Displayed are two DiffFrames involved in the evolution of the Vehicle component. In the left DiffFrame various components were added to Vehicle. In the right DiffFrame the condition of the if was modified and the branches switched. In the else-branch an additional statement was inserted.*

2.6.2 Overview of changes

Using state of the art code review tools, it can be difficult to get an overview over which components have changed and to identify regions of a project which could possibly be affected by the changes [23]. By displaying a view which shows a map of all the components of the software project and highlighting the changed ones, it is possible to identify the regions or modules of the project, where the changes take place. This allows developers to get an idea of which areas of the software project could be impacted by the changes.

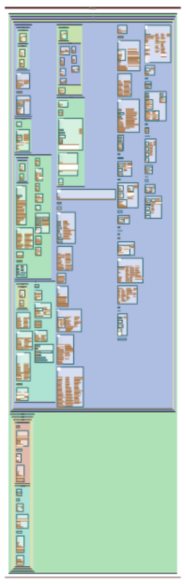


Figure 2.29: *Large example project in Envision in a zoomed-out view. In this view different regions of the software project can be identified. A developer working on this project may have a good idea which modules are located in these regions.*

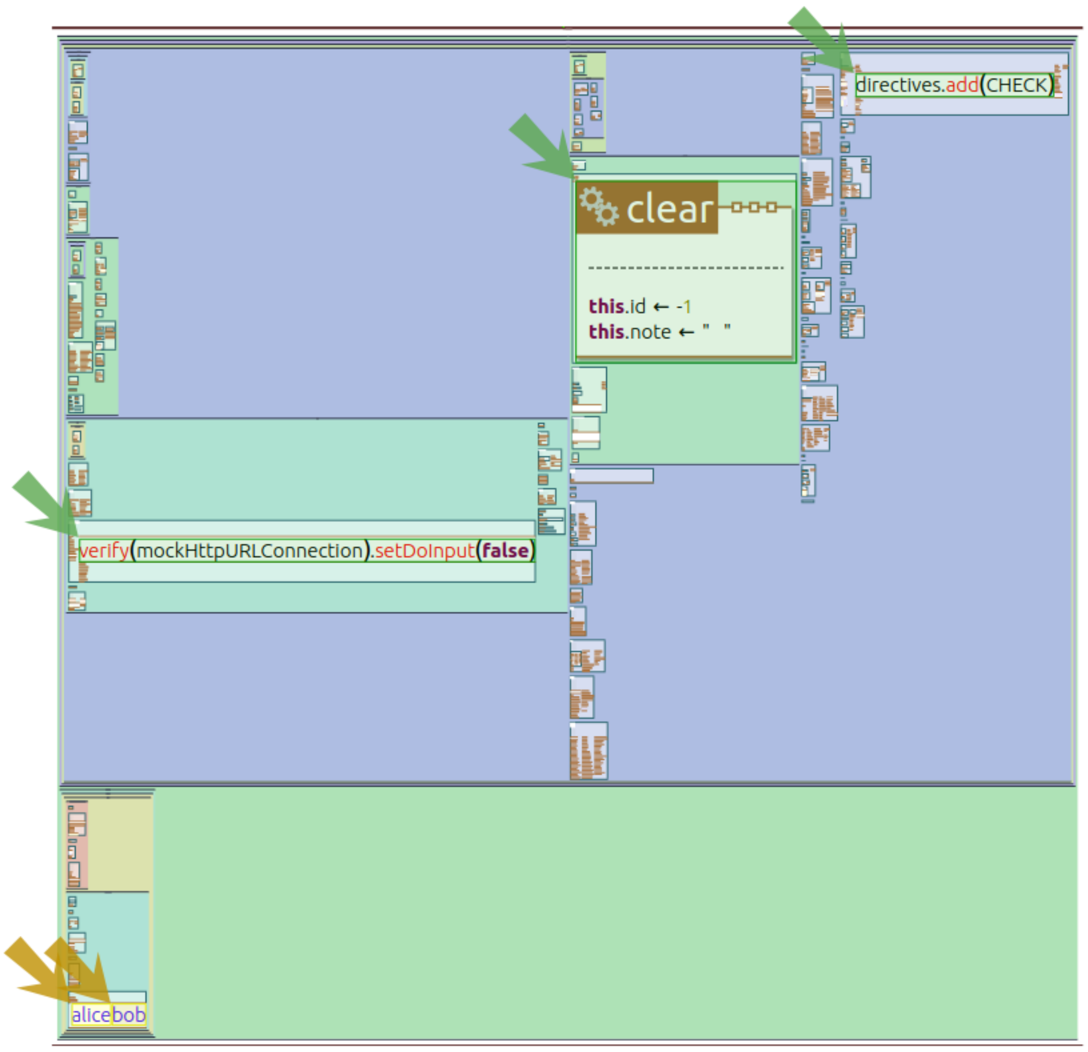


Figure 2.30: The same view as shown in 2.29 after executing the `diff` *command using the* `overview` *option. The highlighted changes are the only code which is readable. Developers with knowledge of the project can get a first idea where the changes are located in the project and what components could possibly be affected by them.*

For the implementation in Envision, we use the zoom feature as described in section 2.5.2. In Figure 2.29 a large project, which was loaded in Envision, is displayed in a zoomed out view.

The `overview` option for the `diff` command uses this view to highlight and scale changes in the project. Since this is also the view developers are accustomed to when working on this project, they may have a good idea about the location of the individual modules in this zoomed-out view.

Figure 2.30 shows the view after executing the `diff` command using the `overview` option. It allows developers to get a first overview over where the changes are located in the project and components that could be potentially affected by the changes. If developers require more details about the context of a particular change they can zoom in on the change. Unlike in the standard diff view, changes of type deletion are not shown in this view. The reason is that this view only shows the current state of the project and the deleted components are no longer part of this visualization.

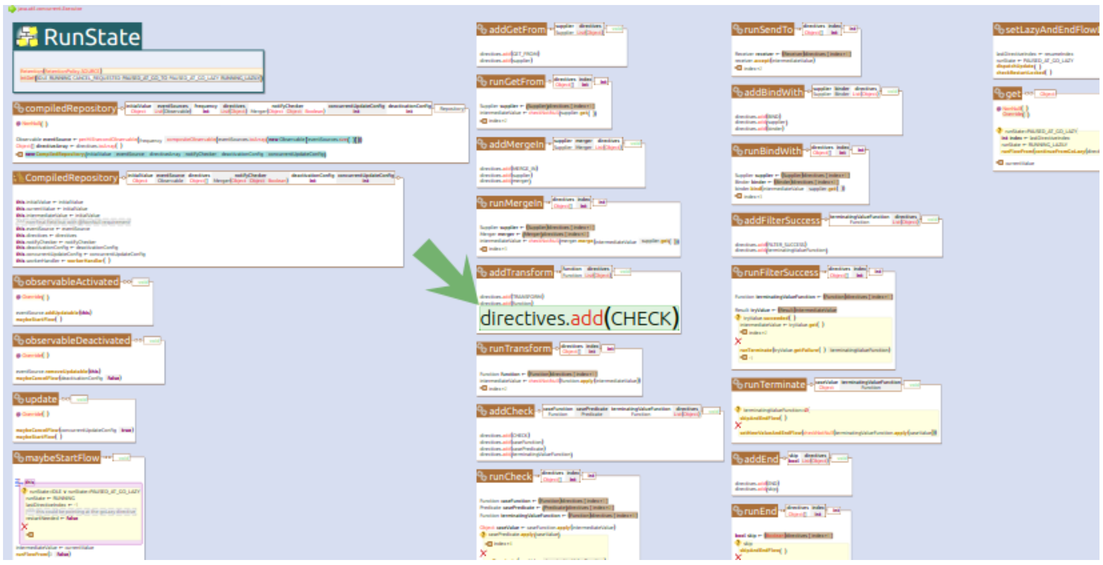
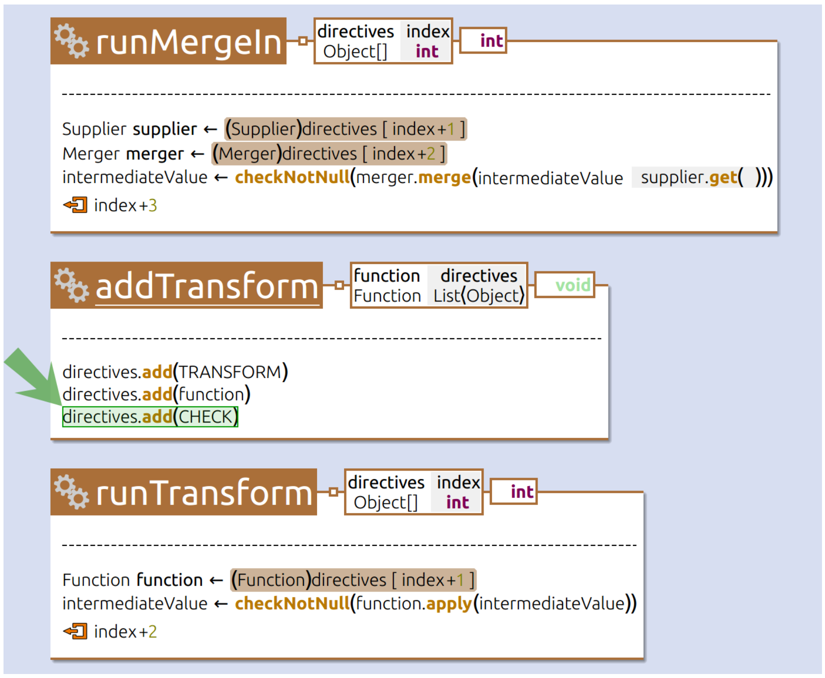


Figure 2.31: The same view as shown in 2.30 after zooming in on a change. Zooming in reveals more context about a change and gives more accurate information about its location in the project.

Figure 2.31 shows how zooming in on a change provides developers with more context. At one point everything is scaled normally and the only difference to the normal view are the highlighted changes, which can be seen in Figure 2.32.



```
runMergeIn  directives  index   int
            Object[]    int
-------------------------------------------------
Supplier supplier ← (Supplier)directives [ index+1 ]
Merger merger ← (Merger)directives [ index+2 ]
intermediateValue ← checkNotNull(merger.merge(intermediateValue  supplier.get( )))
index+3

addTransform  function  directives    void
              Function  List(Object)
-------------------------------------------------
directives.add(TRANSFORM)
directives.add(function)
directives.add(CHECK)

runTransform  directives  index   int
              Object[]    int
-------------------------------------------------
Function function ← (Function)directives [ index+1 ]
intermediateValue ← checkNotNull(function.apply(intermediateValue))
index+2
```

Figure 2.32: The same view as shown in 2.31 after zooming in further on the change. Everything is scaled normally and the only difference between the changed and the unchanged code it the highlight.

2.6.3 Name change summary

If a change set contains, e.g., a refactoring of a field name, it can be hard to distinguish changes which are part of the renaming and changes which are more substantial, e.g., semantic changes in algorithms. Lentz et al. mentioned this difficulty in their work on improving code reviews using fine-grained IDE events [19]. To help the developer in such situations we designed a name change summary feature, which allows them to filter all changes related to a name change and instead display a summary of all name changes in the change set. A sketch of this idea can be seen in Figure 2.33.

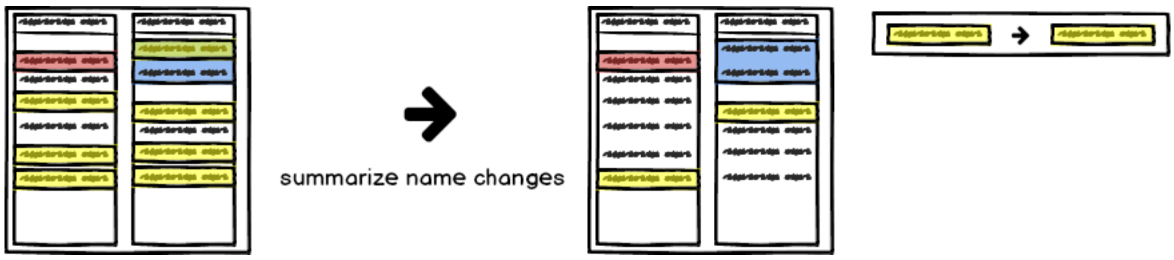
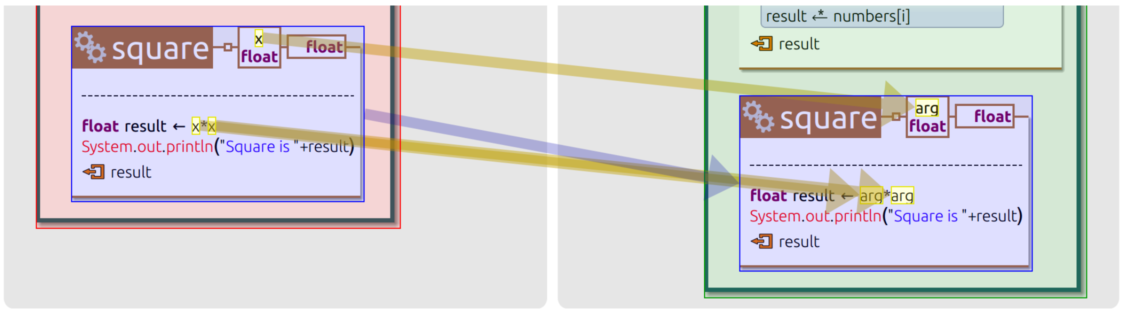


Figure 2.33: Sketch of the name change feature. On the left all changes are highlighted. On the right only changes, which are not related to name changes are highlighted. The information about the name change is provided in the separate box. This leads to less clutter from the highlights and to less highlighted changes which need inspection.

This feature can reduce the amount of clutter in the view due to refactoring changes. While we designed it to help reduce cluttering due to name changes, it is in theory possible to extend this to other refactorings, as long as they can be detected reliably from the changes. We illustrate the design in Envision on an example, showing the different options possible in displaying name changes in Figures 2.34, 2.35, 2.36.



```
square  x Float  Float
--------------------------------
Float result ← x*x
System.out.println("Square is "+result)
result
```

```
result ← numbers[i]
result
square  arg Float  Float
--------------------------------
Float result ← arg*arg
System.out.println("Square is "+result)
result
```

Figure 2.34: No filtering of name changes applied. In this example the changes relating to name changes are highlighted like any other changes.

Users can define the amount of filtering on name changes according to their liking. A textual summary of the name changes can be provided in the displayed information box for the current diff view, as shown in Figure 2.37.

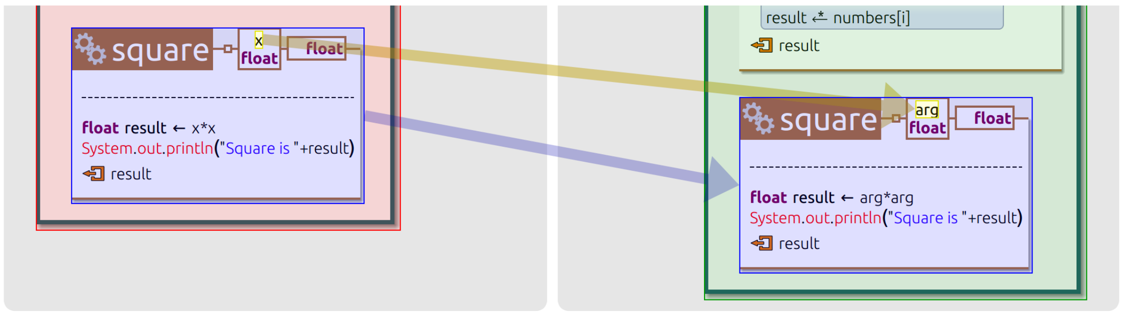


Figure 2.35: Filter highlights of references modified because of a name change. Only the definition of the name change is highlighted.

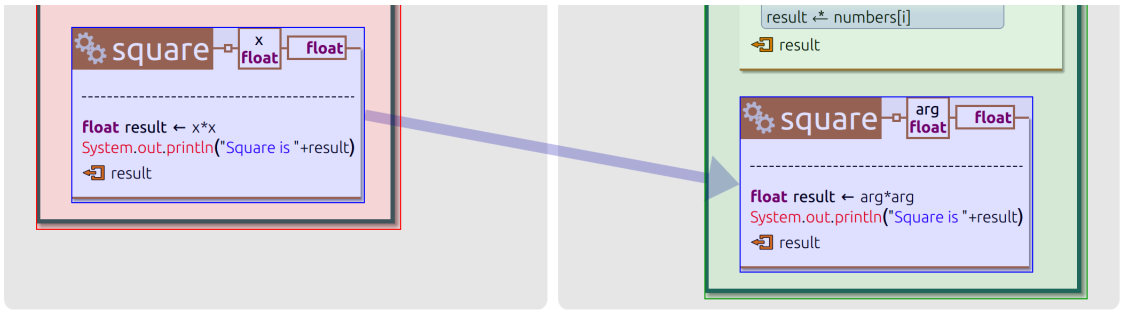
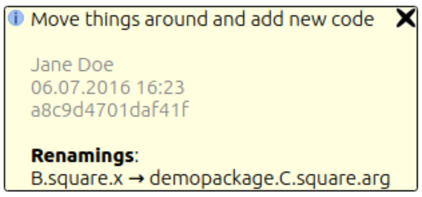


Figure 2.36: Filter all highlights related to name changes. Only changes, which are not related to name changes are highlighted.



Move things around and add new code

Jane Doe
06.07.2016 16:23
a8c9d4701daf41f

Renamings:
B.square.x → demopackage.C.square.arg

Figure 2.37: Textual summary of a renaming. The entries under the title *Renamings* list the different renamings in the current diff view. On the left side of the arrow is the old name and on the right side the new, changed name.

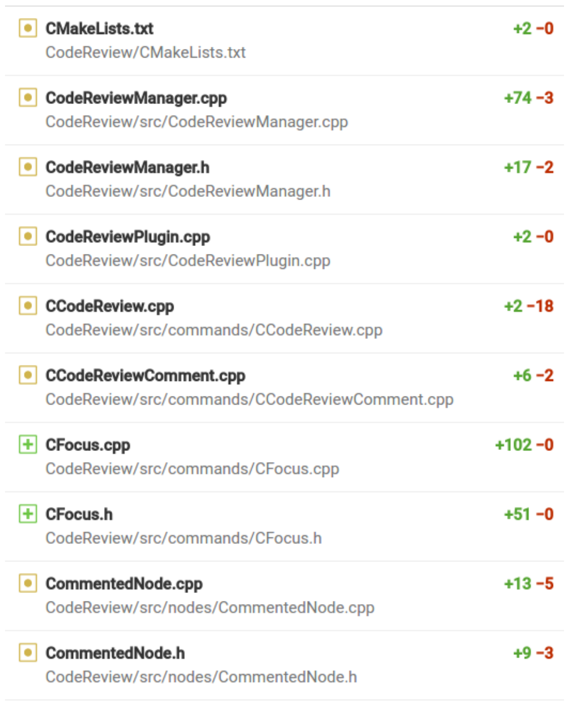
Chapter 3

Facilitating the code review process

Building on the diff visualizations from chapter 2, in this chapter we describe features specifically designed to help during the code review process.

3.1 Difficulties during the code review process

Research surveys [3, 18, 23] have shown that the process of code review is a non-trivial task for developers. A problem which is often mentioned is the fact that commits can contain multiple fixes or feature implementations at once. If the reviewer then is presented with just an alphabetically ordered list of the files, like, for example, in GitHub it can be hard to figure out where to actually start inspecting the changes.



```
CMakeLists.txt                                    +2 -0
CodeReview/CMakeLists.txt
CodeReviewManager.cpp                             +74 -3
CodeReview/src/CodeReviewManager.cpp
CodeReviewManager.h                               +17 -2
CodeReview/src/CodeReviewManager.h
CodeReviewPlugin.cpp                              +2 -0
CodeReview/src/CodeReviewPlugin.cpp
CCodeReview.cpp                                   +2 -18
CodeReview/src/commands/CCodeReview.cpp
CCodeReviewComment.cpp                            +6 -2
CodeReview/src/commands/CCodeReviewComment.cpp
CFocus.cpp                                        +102 -0
CodeReview/src/commands/CFocus.cpp
CFocus.h                                          +51 -0
CodeReview/src/commands/CFocus.h
CommentedNode.cpp                                 +13 -5
CodeReview/src/nodes/CommentedNode.cpp
CommentedNode.h                                   +9 -3
CodeReview/src/nodes/CommentedNode.h
```

Figure 3.1: Order of files in a Pull Request of GitHub. The files are ordered by name. Since header and implementation files in C++ should use the same name, implementation files are shown before the respective header.

Figure 3.1 shows an example in GitHub. The alphabetical list shows implementation files of classes before the related header files. This is not a sensible order in which to review files. Even if the commit would be untangled (e.g., only contain changes relating to one bug fix) one would inspect header files before implementation files.

Developers stated that it would help if there was some decomposition of such tangled changes into subsets of related changes [4]. Another way to facilitate the code review process would be if the author of the changes could provide some guidance to the reviewers [3].

Besides the obligatory comment feature, we provide two additional features to help with the two cases mentioned by developers.

3.2 Review comments

An essential feature for code reviews is the possibility to communicate over the tool. Reviewers want to give feedback to the code or ask the committer for additional input regarding some changes. Our requirements for the comments are as follows:

- **Identification:** it should be clear who wrote a comment.
- **Date:** It should be possible to easily see how old a certain commit is.
- **Discussion:** There should be a mechanism which allows developers to have discussions for a certain feature, i.e., comments which belong to the same discussion should be identifiable as such.
- **Individual commenting:** The author of a comment should have the ability to comment on any component displayed during a review.

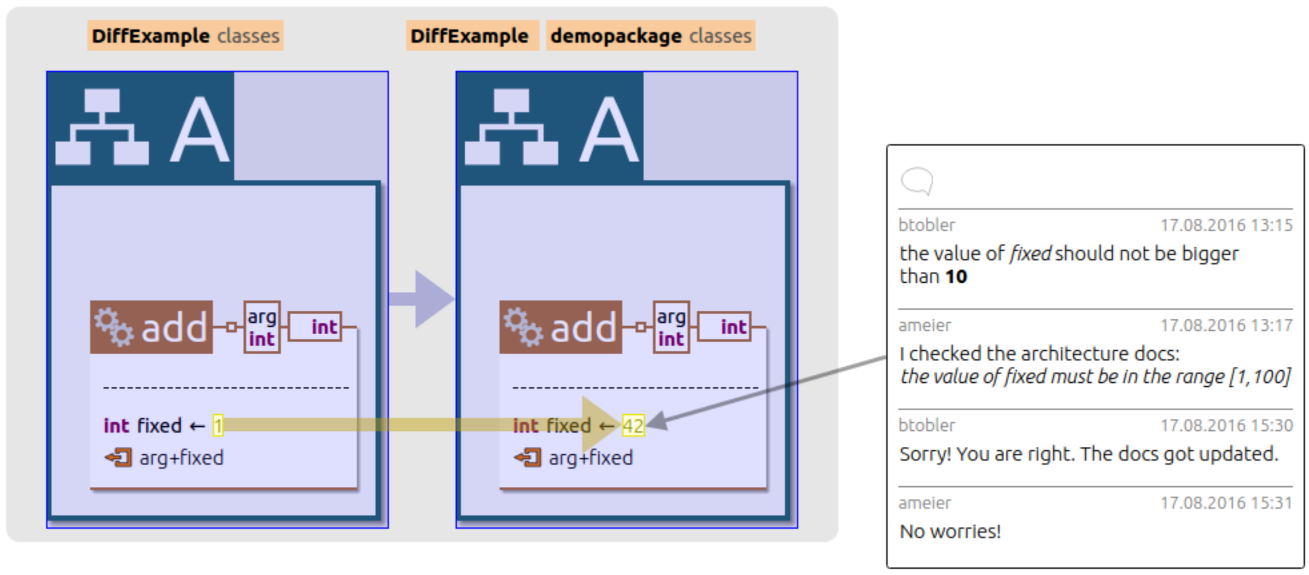
We use the rich comments of Envision, designed during the work of Trappenberg [24] and Dinkel [10] as a basis for the review comments. These allow developers to have mark-up in the comments as well as rich components like images, tables or even embedded HTML and JavaScript code.

Figure 3.2 shows an example of a review comment in Envision. Since Envision does not yet support a multiple-user system we use the current user of the system to generate the usernames. It is possible to move the comments around using drag and drop interaction with a mouse. Persistence of the comments and their location in the view is supported.

3.3 Automated grouping and ordering of changes

Participants in the survey by Tao et al. [23] called for a feature that could automatically decompose a composite change into separate sub-changes. We designed a feature that groups changes according to strategies. Users can choose the strategies, which should be used for the ordering.

The sketch in Figure 3.3 shows the possible process during the code review preparation. Using different strategies, like for example grouping related changes, the diff will be grouped and ordered according to some strategy-depending criteria. The author of the



btobler 17.08.2016 13:15
the value of *fixed* should not be bigger than **10**

ameier 17.08.2016 13:17
I checked the architecture docs:
the value of fixed must be in the range [1,100]

btobler 17.08.2016 15:30
Sorry! You are right. The docs got updated.

ameier 17.08.2016 15:31
No worries!

Figure 3.2: Example of a review comment in Envision. The comment is associated with the statement int fixed = 42. The arrow connecting the comment with the statement indicates this association. It is possible to have a discussion consisting of multiple comments talking about the same node. The top left corner of a single comment shows the username of the author, while in the top right area the date of the comment is displayed. As can be seen the comments allow marked-up text like, for example, bold and italic fonts.

changes have the change to provide further guidance to the reviewer by using the focus feature. See section 3.4 for information about the focus feature.

The user can choose strategies for two categories: one strategy for grouping the changes, and a second one to order the changes in the group. If the order of the groups should also be automatically arranged the system needs to be extended to handle a third strategy which defines the ordering of groups. In the following example we used an use-analysis strategy to group the changes. This analysis puts two *DiffFrames* together if one *DiffFrame* references components, e.g., methods, which are part of a change displayed in the other *DiffFrame*. This leads to a grouping where related changes are displayed together.

Figure 3.4 shows an example where the mentioned strategies were used. The layout of the changes uses the 2D-plane to indicate the groups. We designed the function which uses the strategies flexible enough, so that the system can be extended with any number of other strategies.

3.4 Author-provided guidance

Interviewed developers explained that having the author of the changes providing some sort of guidance can be of great value during the review [3]. We designed a feature, where the author of the changes can prepare the code review by adding comments with focus annotations, which provide guidance to the reviewer and can help in understanding the reasoning of the changes. These focus annotations provide information about how and in what order the comments should be presented to the reviewer. It is possible to mark comments with highlights or centering the view on a specific comment. Further it is pos-

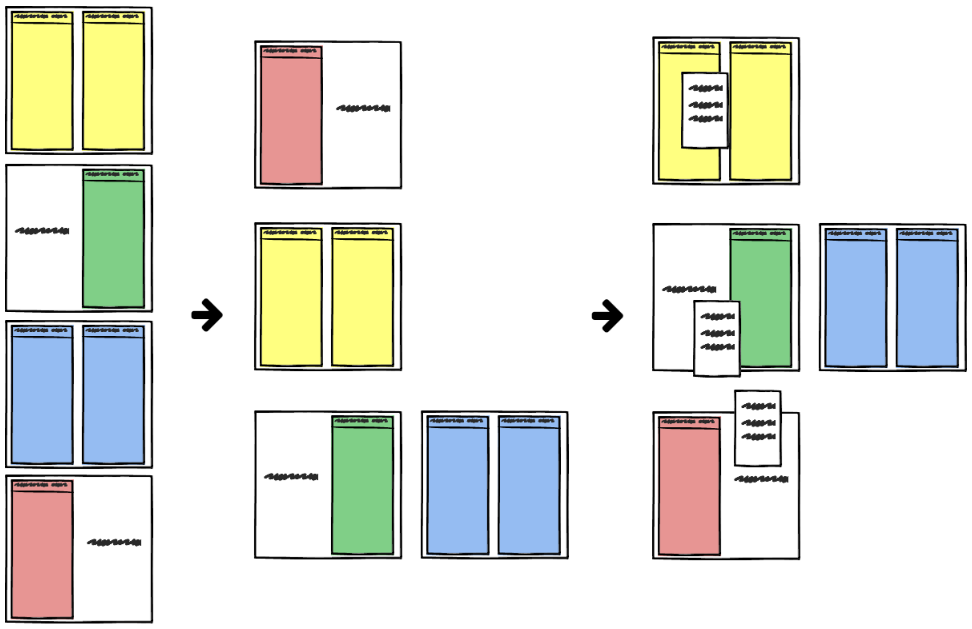


Figure 3.3: Sketching the process for automated ordering. On the left the list of arbitrarily ordered changes is forwarded to the automated grouping and ordering feature. The changes are grouped and ordered, which can be seen in the middle step. In the last step the author of the changes has the possibility to annotate the changes with additional information and define a specific path through the changes to guide the reviewer.

sible to define at which step a comment should be marked or centered. This allows the author to define a "review story" by defining such a sequence of focus steps. Reviewers can then choose to follow this pre-defined sequence of the author or inspect the changes and comments on their own accord.

The focus information can be specified in the top right corner of a review comment. One information which needs to be specified is whether the comment should be centered, highlighted or both. Such an option is specified by writing a substring of *highlight* or *center* with minimum length 1 in the focus information field. After specifying the type of focus for the comment, it is mandatory to define the step at which it should be focused by providing a number. It is possible to have multiple comments with the same number, which allows developers to, for example, highlight a group of comments at the same time. If *center* is activated for multiple comments at the same time, the view will be centered on only one of the comments. Figure 3.5 shows an example of a comment annotated with focus information.

Reviewers can use the focus step key combination to step through the comments which are defined in the focus sequence. This means, that the view gets centered on specific comments, describing the changes which connect to the centered comment. Another possibility is that highlights are shown to focus the attention of the reviewer to specific comments.

Figure 3.6 gives an example how a review could look from the perspective of a reviewer. The changes were automatically grouped and ordered and the author provided some guidance in form of review comments annotated with focus information.

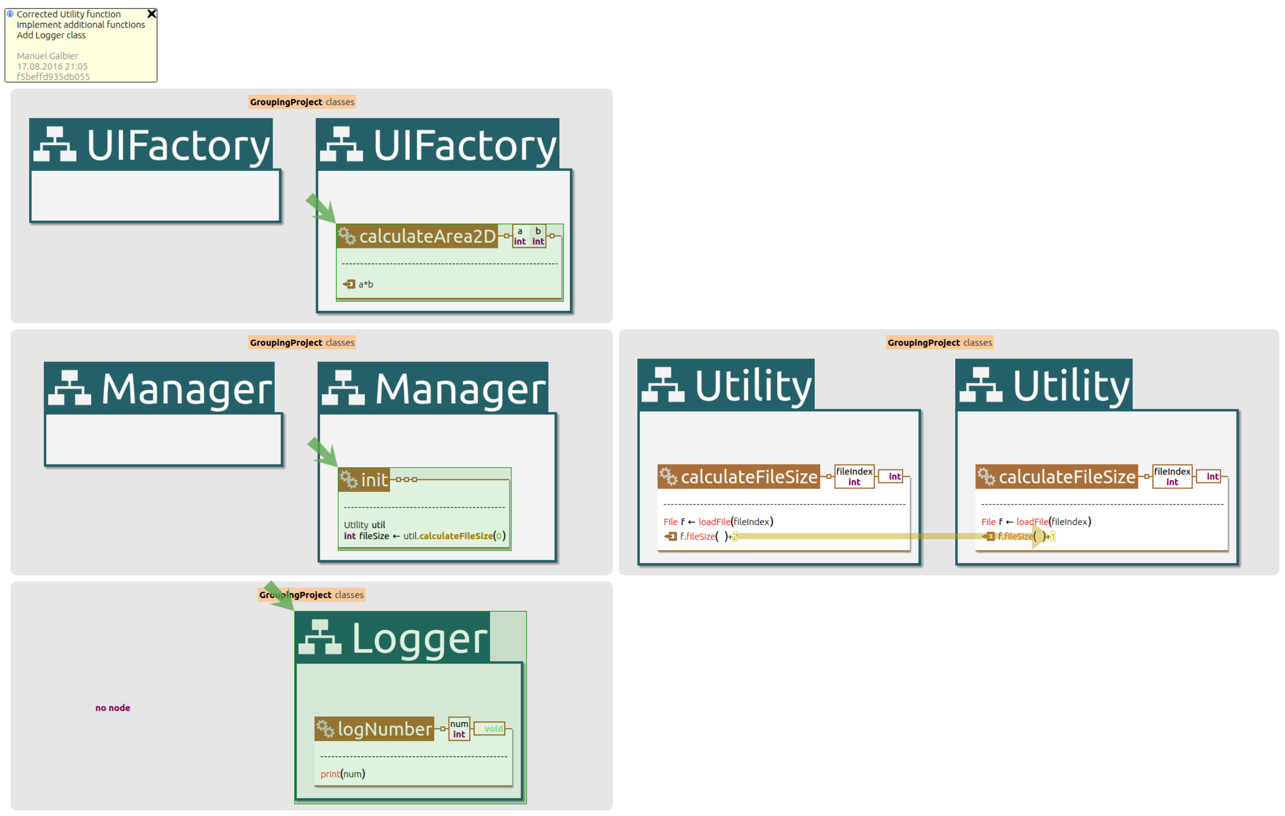
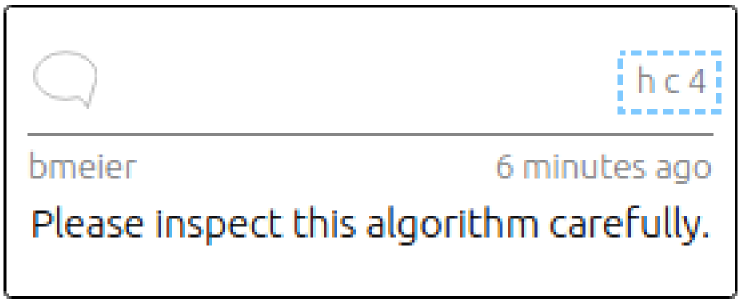


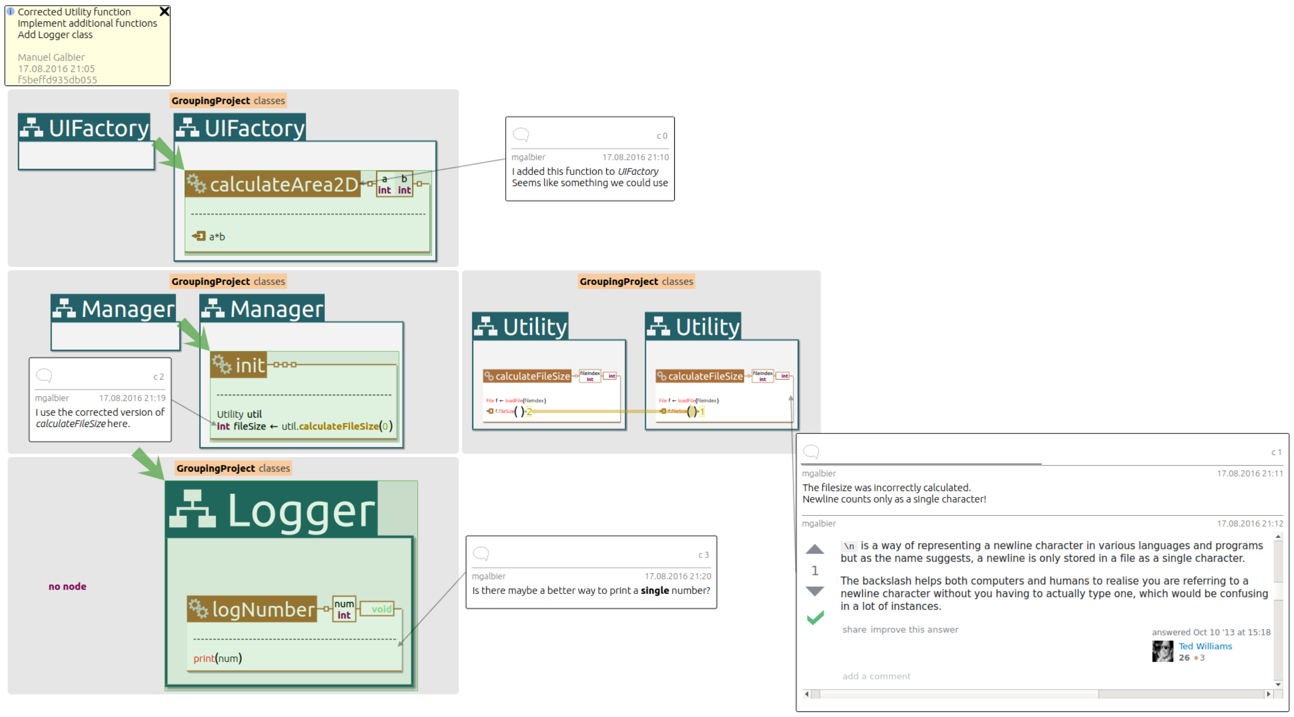
Figure 3.4: Simple example showing grouping using use-analysis. Each row represents a group of changes. The first and third row display independent changes and are therefore displayed by themselves. Row two shows two DiffFrames grouped together because the init method in the first DiffFrame calls the calculateFileSize method which was modified as can be seen in the second DiffFrame.



bmeier 6 minutes ago

Please inspect this algorithm carefully.

Figure 3.5: Review comment with focus information. In the top right corner of the comment the focus information is highlighted. It defines, that this comment should be ***h****ighlighted and* ***c****entered at the* ***4****th step of the focus sequence.*



Corrected Utility function
Implement additional functions
Add Logger class

Manuel Galbier
17.08.2016 21:05
f5beffd935db055

mgalbier 17.08.2016 21:10
I added this function to *UIFactory*
Seems like something we could use

mgalbier 17.08.2016 21:19
I use the corrected version of *calculateFileSize* here.

mgalbier 17.08.2016 21:20
Is there maybe a better way to print a **single** number?

mgalbier 17.08.2016 21:11
The filesize was incorrectly calculated.
Newline counts only as a single character!

mgalbier 17.08.2016 21:12
`\n` is a way of representing a newline character in various languages and programs but as the name suggests, a newline is only stored in a file as a single character.

The backslash helps both computers and humans to realise you are referring to a newline character without you having to actually type one, which would be confusing in a lot of instances.

answered Oct 10 '13 at 15:18
Ted Williams
26 ● 3

Figure 3.6: Example of code review featuring author-provided comments with focus annotation. A reviewer can use the focus step keyboard shortcut to step through the comments featuring focus information. Depending on the focus information defined by the author this can center the view on different comments and display some comments with highlights.

Chapter 4

System design

This chapter gives an overview of the implementation of the presented features in Envision. Some selected features are discussed in more detail.

4.1 Component overview

The main implementation work consists of two new plugins for the Envision project: *VersionControlUI* and *CodeReview*. The plugin *VersionControlUI* contains functionality which is directly related to the diff and its visualizations. In the future it will be used to also contain visualizations for merging and other version control related tools. *CodeReview* is the plugin responsible for functionality directly concerned with code reviews. The review comments, automated ordering and grouping, and focus functions are located in this plugin. The *CodeReview* plugin depends on the *VersionControlUI* plugin.

4.1.1 VersionControlUI plugin

In this section we present an overview of the most important classes in the *VersionControlUI* plugin. Figure 4.1 shows a diagram of these classes.

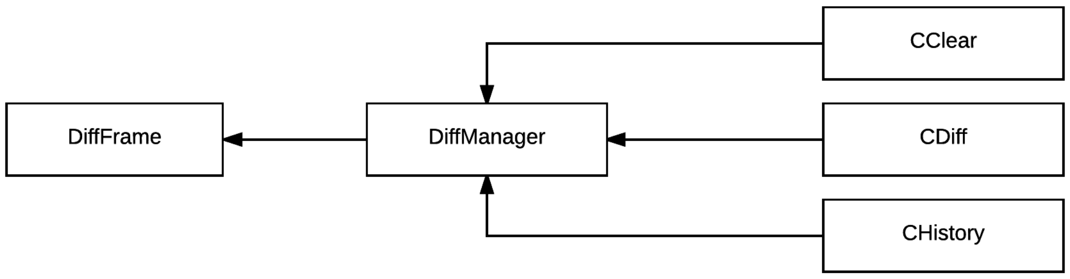


Figure 4.1: Diagram showing important classes in the VersionControlUI plugin. The commands CClear, CDiff and CHistory invoke DiffManager. The class DiffFrame represents the DiffFrames used for every view, that shows changes. DiffManager handles the creation of DiffFrames.

CClear implements the command to clear the view of highlights and scalings related to the diff. Its main use is to clean up the project view after executing `diff` with the `overview` option.

CDiff is the class which implements all of the `diff` commands. It creates abbreviated, unambiguous identifier for the different commits available. To show a diff view or showing the changes in the current view using the `overview` option it invokes *DiffManager*.

CHistory implements the `history` command. It uses *DiffManager* to create and display the history view.

DiffManager is the entry point for most of the presented features related to the diff. Preparing the diff and history, instantiating the required *DiffFrames*, scaling and highlighting of changed components and identifying name changes and processing them are part of the tasks handled by this class.

DiffFrame represents the displayed *DiffFrames* to manage the showed changes. They serve as context containers for the displayed changes. The object paths feature interactivity and allow developers to display the appropriate component (e.g., class, method, etc.) next to the displayed *DiffFrame*. Contains the logic to extract the object path of a component and provides the breadcrumbs. See Figure 4.2 for an example of a *DiffFrame*.

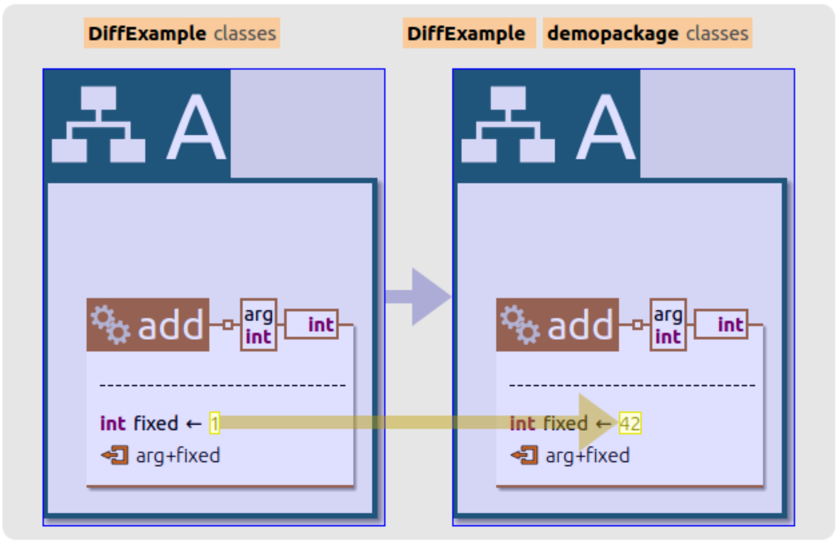


Figure 4.2: Example of a DiffFrame. Shows the object paths consisting of individual bread crumbs in the top area. The old version of the displayed class A is displayed on the left side and the new version on the right side.

4.1.2 CodeReview plugin

In this section we present an overview of the most important classes in the *CodeReview* plugin. Figure 4.3 shows a diagram of these classes.

CCodeReview implements the `review` command used to open the review view. It invokes the *DiffManager* class from the *VersionControlUI* plugin to get the *DiffFrames* for a change set. The class *CodeReviewManager* is called in order to automatically order and group the *DiffFrames*. If available, existing code review

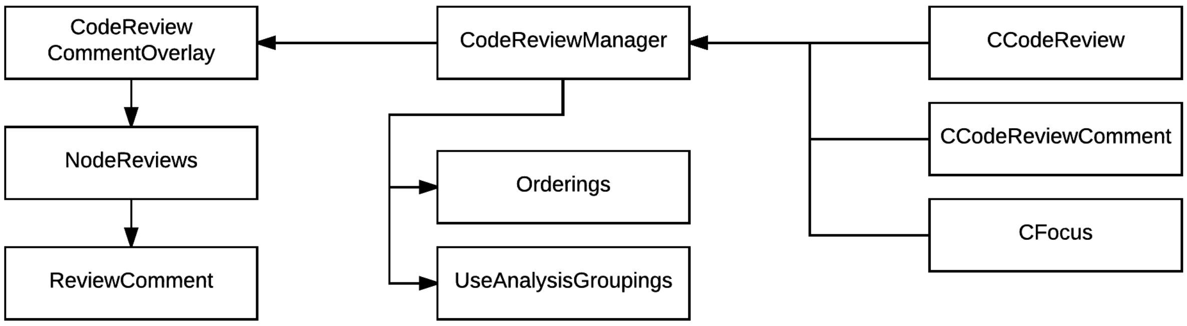


Figure 4.3: Diagram showing important classes in the CodeReview plugin. The commands CCodeReview, CCodeReviewComment and CFocus invoke CodeReviewManger. The class CodeReviewCommentOverlay is used to display code review comments, it contains NodeReviews which itself consists of ReviewComments. Orderings stores ordering strategies. UseAnalysis-Groupings provides strategies for groupings using use-analysis. CodeReviewManager uses both of these classes to achieve automated ordering and grouping for changes.

comments are then loaded before displaying the review view with the ordered and grouped *DiffFrames*.

CCodeReviewComment implements the `comment` command. It allows developers the creation of code review comment for nodes of displayed software project. It uses *CodeReviewManager* to manage the code review comments.

CFocus implements the `focus` command and provides the logic to step through the focus sequence defined by review comments annotated with focus information.

CodeReviewManger handles the saving, loading and managing of code review comments and uses grouping and ordering strategies to order list of *DiffFrames*.

CodeReviewCommentOverlay is the overlay class which displays *NodeReviews* for the associated node.

NodeReviews represents the relation between a node and its code review comments. Stores a list of *ReviewComments* associated with the node.

ReviewComment represents the single code review comments.

Ordering implements two ordering strategies. One is the alphabetical strategy which orders the *DiffFrames* according to the name of the displayed component, the other just returns the same list of *DiffFrames* and therefore does not change the order.

UseAnalysisGroupings implements strategies for grouping, which use an use-analysis. It checks for relations between *DiffFrames* by inspecting the components displayed and referenced by a *DiffFrame* and comparing it to the others.

4.2 Structure-aware zooming

Very early in the development process it became clear that our approach to use different scalings for the changed components on different zoom levels leads to problems with the original zoom function of Envision. To explain the problem, it is necessary to differentiate between geometrically and dynamically scaled items. For example, everything displayed on Google Maps[1] is geometrically scaled. That means if we were to zoom in or out of the map everything we see gets bigger or smaller at the same speed. A component that is scaled dynamically is one where its scale is not simply changed linearly during zooming. Its appearance might change arbitrarily (e.g., non-linear scale). If nothing is scaled dynamically the normal mechanism, which just zooms in on a particular pixel of the scene works fine. Because everything stays the same size, only the distance of the view to the scene changes. The problem stems from the fact, that if visual items are scaled differently the scene and its layout can change during different zoom levels and zooming on the same pixel of the last zoom level can lead to confusing results and an unexpected behavior. To counteract this problem we designed the structure aware zoom. If we zoom in on the scene, Envision checks if an item is under the cursor. If there is one, the items local coordinates are used to remember where we zoomed in/out and the view is moved in a way, that the zoom behaves as expected, i.e., the item which was zoomed in on is under the cursor roughly at the same position as before.

This feature was required to help the user stay oriented during zooming in views, which contain items that are scaled dynamically, i.e., changed components.

4.3 Name change summary

Thanks to Envision's fine-grained version control system we could use changes of type modification to identify name changes. The definition of a name change always involves a modification of a node of type *NameText*. We scan all the changes in an initial step to identify such *NameText* modifications. Using this information we can in a next stage iterate over the changes and identify whether modifications on *Reference* nodes are part of a name change or not. By defining an enum which enables to control whether *NameText* or *Reference* nodes related to a name change should be shown in the diff visualization, we can control the level of detail in which such name changes should be displayed. The renaming summary in the info box can be computed by using information stored during the identification of the name change definitions.

4.4 Automated ordering and grouping

A goal of our design of the automated ordering and grouping feature was, to have it flexible enough that developers can add new strategies to group and order changes without too much effort. To provide new strategies the developer has to implement a *Grouping-Function* or an *OrderingFunction*:

```
using GroupingFunction =
  std::function<QList<QList<VersionControlUI::DiffFrame*>>
                (QList<VersionControlUI::DiffFrame*> diffFrames)>;

using OrderingFunction =
  std::function<QList<VersionControlUI::DiffFrame*>
```

[1] https://www.google.ch/maps

```
                (QList<VersionControlUI::DiffFrame*> diffFrames)>;
```

Listing 4.1: Shows the definitions of the Grouping and Ordering functions located in CodeReviewManager.

A *GroupingFunction* takes a list of all the *DiffFrames* of the change. This information can then be analyzed to group the changes into a list of list, i.e., each list represents a single group of changes. To decide which *DiffFrames* should be grouped together, the *DiffFrames* provide information about the changes they display.The *OrderingFunction* on the other hand is used to order a single group. This means that it takes a list of *DiffFrames*, which represents a group of changes, grouped in a previous step. This group can then be ordered according to some strategy, which returns an ordered list of the group. The rest is handled by the *CodeReviewManager*, which uses the described functions to group and order the *DiffFrames* before displaying the code review view.

Chapter 5

Evaluation

We did not perform a quantitative user study since it was out of scope for this thesis. We did however talk to two developers for qualitative feedback on the features we designed. We present the feedback in the following section. We then analyze our features by comparing them to other tools. In the last section of this chapter we discuss if the features are applicable to other tools.

5.1 Developer feedback

We presented our work to two master students in computer science at ETH Zurich and conducted separate interviews in order to collect qualitative feedback. Both have more than 10 years programming experience and were employed as professional developers in the past. Developer one (D1) had some experience with code reviews in his professional career. He is experienced with the use of the command line `diff` and knows the diff visualizations of GitHub. He had not worked with Envision prior to our interview. Developer two (D2) is experienced with code reviews in a professional setting. He worked in a team where code reviews were a core part of the development process and every change had to be reviewed before it was committed. He has experience with GitHub and has seen (but nor used) Envision before.

The developers were interviewed individually. During the interview we introduced one feature at a time using the example projects discussed in Sections 2.5, 2.6, 3.3 and 3.4 of the report and let the developers interact with them. We then asked for feedback related to positive and negative aspects of the introduced features and other general comments.

Both developers mentioned that the visual approach of Envision was unfamiliar since they are accustomed to having files and folders and a textual representation the code. They both found that our features could help during the code review process, because they provide additional information which can help to understand the changes.

First, we presented the diff visualizations. Both developers mentioned that it would be beneficial to highlight the object path if the entire displayed component of a *Diff-Frame* was moved (for an example of this situation see Envision's diff visualization in Figure 5.1), since it was not immediately clear where to look for the information about the move. They both liked the arrows used to connect changes of type move and modification, because it allowed them to easily identify and match the related changes. D1 liked the fact that the diff visualization shows information about modifications and moves as compared to other diff tools, where he would have to manually identify such changes. D2 mentioned that he would also like to have such arrows in GitHub's diff visualization. Further, D1 would prefer if related changes, in Envision, are displayed on the same height visually, like in GitHub. D2 liked our approach of presenting context and the fact that changed child components share their context with a changed parent component instead of displaying two individual *DiffFrames* for such cases. D2 did not like the visual appearance of the indicator arrows for changes of type insert and delete. Both developers stated that, after an introduction to the colors of the highlights, the visualizations are easy to read and would help in better understanding the displayed changes compared to other diff tools.

Second, we introduced the name change summary. Both D1 and D2 liked that the changes of a refactoring can be summarized using this feature. D2 liked that he could toggle between the different levels of visualization for changes related to name changes and stated that he could see the advantage of this feature if, for example, a large refactoring took place in a change set. It would allow him to focus on the other changes instead of inspecting the individual name changes.

Third, we introduced the overview feature. Both developers found that the feature could be useful. D2 mentioned that he likes that the feature provides him a high-level overview over the changes directly in the project view. D1 argued that if a lot of changes have to be displayed, the overview could become cluttered. He mentioned that he could imagine using this feature while working in a team and coming back after some days of absence to see in what changes were made in which module. D2 mentioned that he finds the colors of the modules/packages in Envision sometimes too similar to the colors used for the highlighting of the changes. Although he originally disliked the change indicator arrows, he thought that they are useful for this feature, since they allow him to easily identify changes. D1 stated that he liked the zoom, because the view stays stable and the mouse pointer remains in the same position.

Fourth, we presented the history feature. D1 mentioned he would prefer to not have a second *DiffFrame* which shows the destination of a moved component in another class. He would instead place the information about the move in the information box similar to the renaming information. D2 on the other hand liked the move arrow and the presentation of the destination of the move in an additional *DiffFrame*. Both developers appreciate the history feature for the ability to inspect changes made to a component on a per-commit basis. D1 stated, that if he would have such a feature available in his IDE he would use it, but he would not use it if it would require opening up an external tool. D2 would prefer to have the empty *DiffFrame* show the summarized information of the change instead of only displaying "All changes have been summarized".

Fifth, we introduced the automated grouping and ordering feature. Both D1 and D2 mentioned that they find it useful to have the ability to group changes which should be inspected together, as compared to other tools where they only have a list of all the changes. D1 argued that a group could grow very large and therefore use a lot of horizontal space if there are many changes which should be grouped together.

Sixth and last, we demonstrated the ability to add review comments with focus information to a code review. D1 stated that he likes the idea of having the ability to define an order in which the review comments should be inspected. He argued that if a code review is performed with a reviewer and the author both in front of the same display, the author would just guide the reviewer through the changes in a certain fashion, but asynchronously this is not easily possible. D2 also stated that it can be useful to define a path through the changes in order to show them in a sensible order. D1 mentioned that this feature could also be used as a reviewer to provide feedback. Using the highlight option it would be possible for a reviewer to mark multiple comments which point to the same type of proposed change.

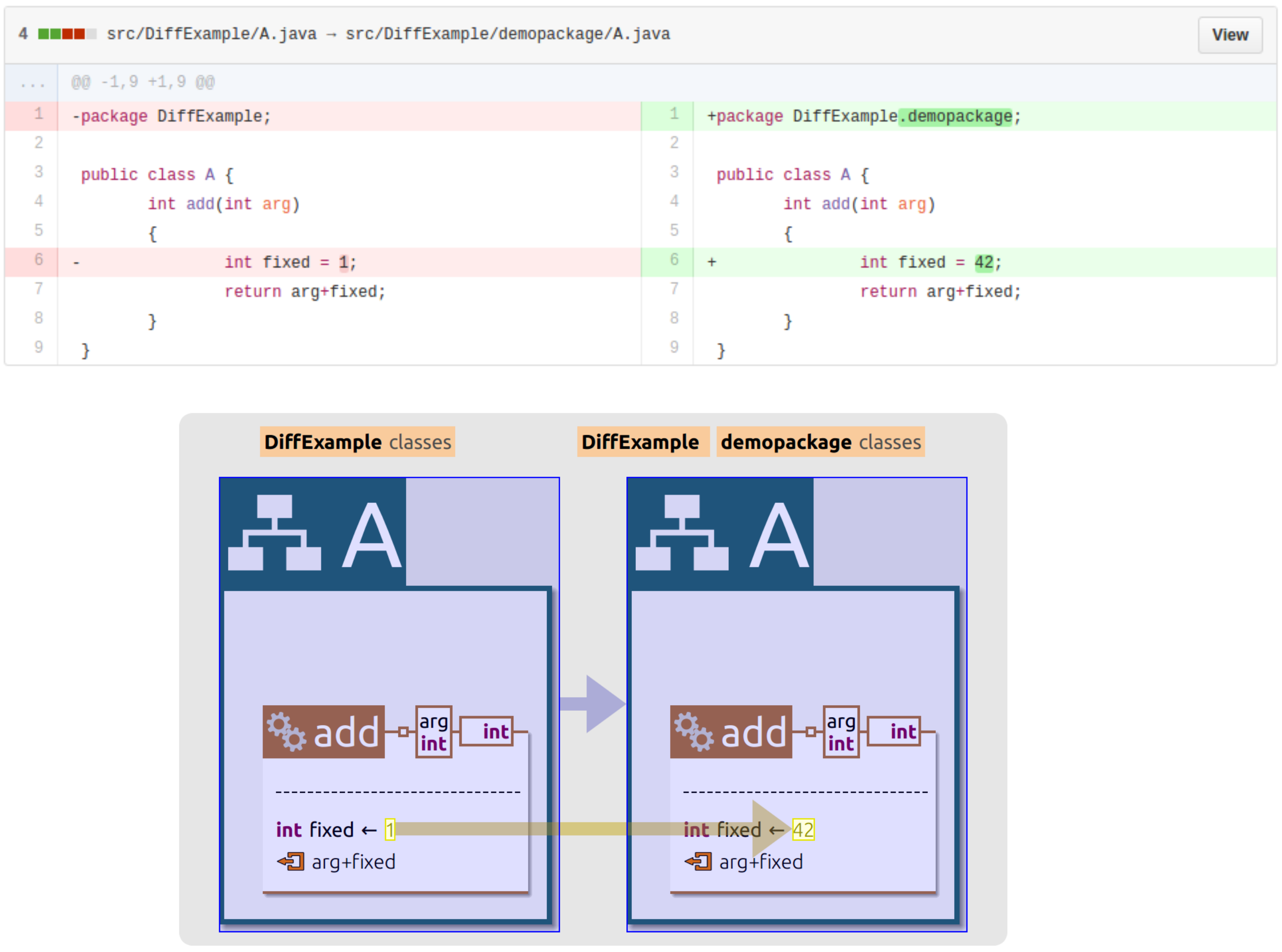
5.2 Diff visualizations

To complement our developer interviews we evaluated the diff visualizations by comparing them to the side-by-side view in GitHub, which is a typical and widely used code review tool. In comparison to GitHub's diff, the diff of Envision can profit from the more fine-grained information provided by Envision's version control system. We discuss the visualizations using simple example projects, which feature the same changes in Envision and GitHub.

Figure 5.1 shows changes to a class *A*. Both visualizations are easy to read and GitHub's line-based diff provides detailed information about the part which was actually changed, e.g., the value of *fixed* which was modified. Envision also highlights the modifications very clearly. The move of the class *A* into another package is displayed more prominently in Envision's diff visualization. Since the entire class *A* is highlighted in blue it becomes immediately clear that the class was moved. The move of the class is also detected by GitHub, but the visualization is more discrete in the header of the displayed change container. It has to be noted that this change looks nice in GitHub because the code was only modified slightly. The matching of moved files in GitHub is heuristic and uses a similarity index to decide whether a file was renamed/moved or not. The default similarity index threshold of git-diff is 50% [1]. In Envision on the other hand, moves are tracked precisely and it does not rely on thresholds on heuristics to identify them. In Envision it is possible to track fine-grained moves which GitHub's diff does not support. This means Envision is able to track moves not only for entire files but also for any node like, for example, individual statements.

Another clear advantage of Envision is the possibility to click on the "breadcrumbs" in the object path, located at the top of the *DiffFrame*, to quickly open other parts of the project for inspection.

[1] `https://git-scm.com/docs/git-diff`



```
4 ■■■■■ src/DiffExample/A.java → src/DiffExample/demopackage/A.java                                   View
...  @@ -1,9 +1,9 @@
-package DiffExample;                          +package DiffExample.demopackage;

 public class A {                               public class A {
         int add(int arg)                               int add(int arg)
         {                                              {
-                int fixed = 1;                +                int fixed = 42;
                 return arg+fixed;                              return arg+fixed;
         }                                              }
 }                                              }
```

Figure 5.1: *The same changes displayed in GitHub's split diff view on top and Envision's diff view below.*

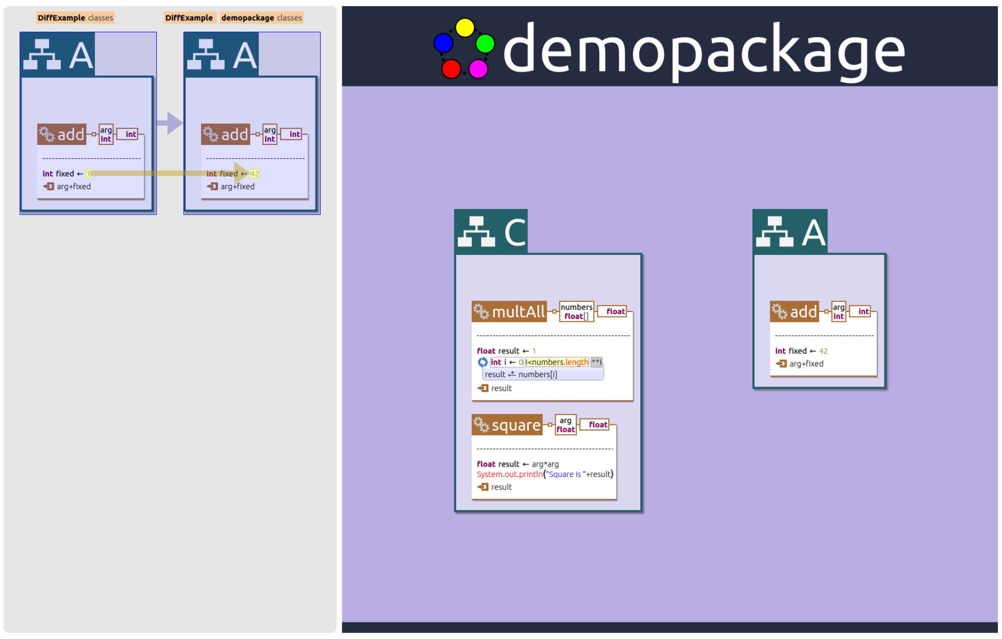
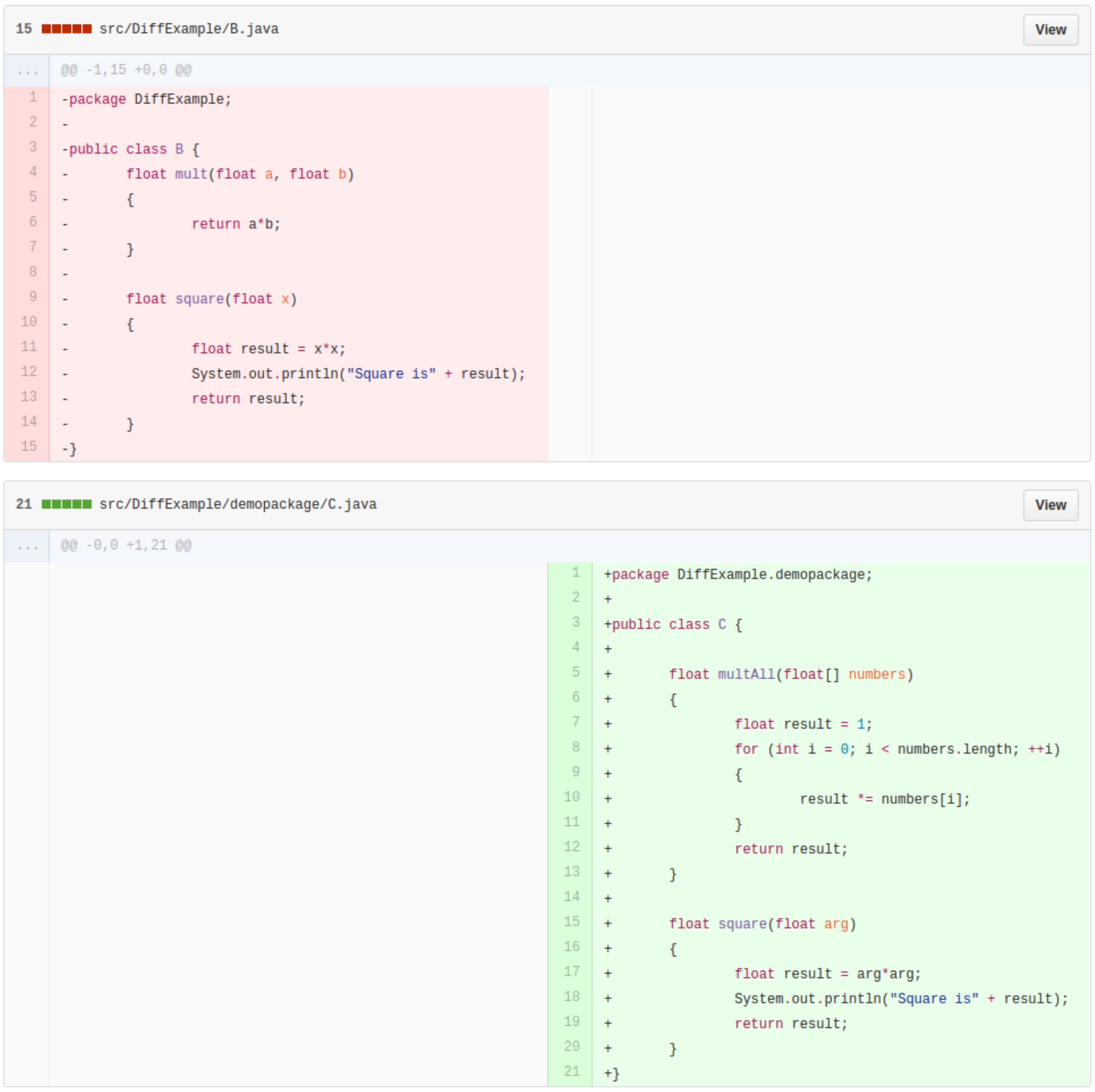


Figure 5.2: *Using object path items to add additional context. In this example the user clicked on the demopackage breadcrumb item, which opened the corresponding component to the right of the DiffFrame.*

In Figure 5.2 an example of an use of the object path is shown. By clicking on *demopackage* it shows the corresponding component to the right of the *DiffFrame*. It is not easy to get this kind of context in other code review tools like GitHub. Since they are file-based it is not possible to show a package like this. The individual classes need to be searched and inspected per file. Envision's command prompt also allows developers to add any other component of the software project by name.

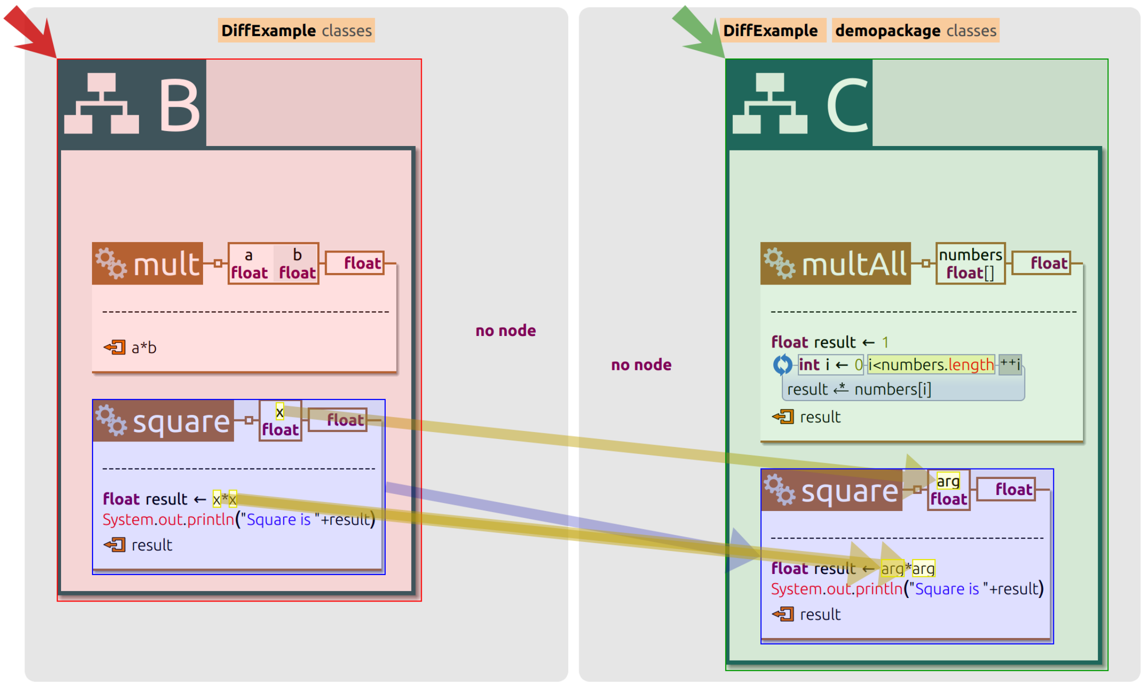


```
15 ■■■■■ src/DiffExample/B.java                                    View
... @@ -1,15 +0,0 @@
-package DiffExample;
-
-public class B {
-        float mult(float a, float b)
-        {
-                return a*b;
-        }
-
-        float square(float x)
-        {
-                float result = x*x;
-                System.out.println("Square is" + result);
-                return result;
-        }
-}
```

```
21 ■■■■■ src/DiffExample/demopackage/C.java                        View
... @@ -0,0 +1,21 @@
+package DiffExample.demopackage;
+
+public class C {
+
+        float multAll(float[] numbers)
+        {
+                float result = 1;
+                for (int i = 0; i < numbers.length; ++i)
+                {
+                        result *= numbers[i];
+                }
+                return result;
+        }
+
+        float square(float arg)
+        {
+                float result = arg*arg;
+                System.out.println("Square is" + result);
+                return result;
+        }
+}
```

Figure 5.3: Changes to class B and C in GitHub's split diff view.

Figures 5.3 and 5.4 show the removal of class *B* and the insertion of class *C* in GitHub and Envision, respectively. In both versions it is easy to see that the whole classes were deleted or inserted respectively. This example shows the advantage of having the differences supplied by a fine-grained version control system, like Envision's. Using the information we are able to visualize the move of the method *square*, which in GitHub's



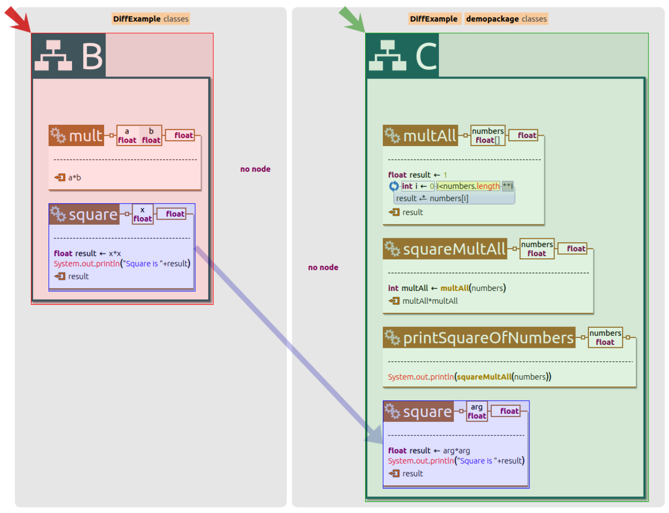
```
mult  a float  b float  float
⟲ a*b

square  x float  float
float result ← x*x
System.out.println("Square is "+result)
⟲ result
```

```
multAll  numbers float[]  float
float result ← 1
int i ← 0  i<numbers.length  ++i
    result ←* numbers[i]
⟲ result

square  arg float  float
float result ← arg*arg
System.out.println("Square is "+result)
⟲ result
```

Figure 5.4: Changes to class B and C in Envision's diff view.

visualization has to be identified by the developer.

Presenting changes which should be inspected together like modifications and moves could be improved. If class *C* would add more methods, the distance between the moves would already begin to increase, which Figure 5.5 illustrates.



```
mult  a float  b float  float
⟲ a*b

square  x float  float
float result ← x*x
System.out.println("Square is "+result)
⟲ result
```

```
multAll  numbers float[]  float
float result ← 1
int i ← 0  i<numbers.length  ++i
    result ←* numbers[i]
⟲ result

squareMultAll  numbers float  float
int multAll ← multAll(numbers)
⟲ multAll*multAll

printSquareOfNumbers  numbers float
System.out.println(squareMultAll(numbers))

square  arg float  float
float result ← arg*arg
System.out.println("Square is "+result)
⟲ result
```

Figure 5.5: Increasing distance between a moved components origin and destination.

It would be beneficial for the visualization and its readability to find an approach to try and keep such related components close.

The advanced features, namely overview, name change summary and history of changed components are in our opinion valuable tools to facilitate the understanding of changed components and their context. Compared to Git's feature `git-blame`, our history feature allows to track changes to a specific component (e.g., a method) precisely, whereas `git-blame` only allows to track code snippets and has no concept of classes, methods, etc. Further, `git-blame` has to rely on heuristic for changes like, for example, line moves and is therefore not as precise as the `history` command in Envision. The overview feature could be improved by not only showing changes relating to the new version, but also changes relating to the old version, i.e., show removed components somehow during overview and add origin of moves, as well as original components of modification changes.

5.3 Code review features

Code review comments are considered the main building block of code reviews [6] and essential to discuss the changes. Our code review comments enable developers to use mark-up to structure the comment text or insert images and tables, which provide additional possibilities to discuss issues of the code. By not limiting developers to express issues or show information in plain text only, we allow them to comment more effectively. Tymchuk et al. [25] state that effective commenting should be part of an improved code review approach. Further, we enable developers to define fine-grained comments on any individual component in the review view. This allows developers to more precisely discuss issues in the code by associating the comment directly with specific components instead of commenting on the entire line like in other code review tools.

Developers stated that author-provided guidance would help them in the change understanding task [3, 4]. We enable authors to provide guidance to the reviewers by annotating changes with comments featuring focus information. This helps reviewers to get an idea in what order to inspect a change set and better understand the reasoning for its changes. In our opinion it can help decrease the time needed to review code, if the author of the changes takes the time to add comments where needed and define steps to look at the changes.

5.4 Applying designed features to other tools

The visualization concepts we designed are applicable to any tool that provides fine-grained change information. One advantage of using Envision is that it is a visual code editor and therefore already provides a strong visual basis, unlike other IDEs.

Name change summary requires a robust way to identify name changes. Any code review tool which provides a robust refactoring detection should be able to add a name change summary feature.

Overview feature requires a visual overview of a software project. Is this provided by a code review tool, it can use change information to highlight the changes in such a view.

History feature requires a mechanism which retrieves the evolution of a specific component. Most current code review tools have no concept of components like, for example, methods and statements. To be able to provide a feature similar to our history feature it is necessary to track changes of components robustly across files.

Automated grouping and ordering requires a mechanism which analyzes the changes to group and sort them. Further, a visualization of the groupings is necessary. One option could be a canvas showing simple boxes containing the normal diff views grouped and ordered with each other. Another option would be to just group the changes in a list using,for example, distance between entries which belong to different group.

Author-provided guidance requires at least the ability to comment on changes. Since this is possible in every code review tool it is feasible to implement a similar feature in order to provide guidance. One possibility would be to allow developers to define an order on the comments. That way the author could provide a description of the changes and use the ordering to present reviewers with a sensible sequence in which they could inspect the changes. The tool would need a mechanism which uses the information about the order of comments and show them to reviewers in this defined sequence. Another way would be to let authors sort line-comments by numbering them and display a list of comments using this order.

Generally speaking, the main difficulty in applying the designs of our features to other tools is their lack of support for fine-grained changes and robust identification of refactorings. If tools support this the designs are applicable, provided they support visual views of the software projects, classes, etc.

Chapter 6

Related work

6.1 Code review process

Bacchelli and Bird [3] investigated the motivations, challenges and outcomes of tool-based code reviews. They observed and interviewed developers and managers and found that the key aspect in code reviews is code understanding. This crucial part also takes most of the time of a review. They came to the conclusion that modern code review tools do not help the developers enough in the process of code understanding. While many modern IDEs integrate tools which aid context and understanding, all the current code review tools they knew of only showed a highlighted diff of the changed files and would not support any of these tools. Their suggestion is that code review tools should help more in the area of code comprehension. Combined with the possibility for the developer to provide the reviewer with context and direction regarding changes they expect it to lead to faster and better code reviews. Czerwonka et al. [9] state that code reviewing is often the longest part of the code integration activities. They found that code reviews often do not find functional defects which should block submission of the code. Further it is important to assign a code reviewer who has the right set of skills and knowledge to inspect the code for better results. They suggest that the quality of code reviews seems to be higher if the number of changes, i.e., the number of included files, for a single review is not too big. In general they state that due to its cost code review is a topic which should be better understood and integrated in the software engineering workflow. Kononeko et al [18] investigated how developers perceive code review quality. They performed a survey of 88 Mozilla core developers. They found that the review quality is influenced by mainly three factors: first, the familiarity of the reviewer with the code. Second, with quality of feedback. Third how the reviewer perceives the quality of the code. Developers stated that gaining familiarity with the code is one of the main challenges during code reviews. Developers further stated interest in better development environments, which offer the ability to easily get the patch from the issue tracking system into the local editor for analysis. Generally, developers want improved support for diff tools. Due to the importance of change understanding and the request for better diff tools which support IDE-features, our approach provides code reviews directly in the IDE with support of all the standard IDE tools and interactions. This eliminates the need to switch to other tools in order to perform such tasks.

McIntosh et al. [20] studied the relationship between software quality, code review coverage and code review participation. By examining the Qt[1], VTK[2] and ITK[3] projects they found that code review coverage and participation share a significant link with software quality. They showed that poorly reviewed code has a negative impact on software quality in large software systems using modern code review tools. They suggest that the amount of discussion generated during reviews should be considered when making integration decisions.

Tao et al. [23] performed a large-scale study at Microsoft to learn how developers try to understand changes made to the source-code of software. They found that it is difficult to judge a change's completeness, consistency and the risk it imposes on other software components. During interviews developers requested to have the possibility to use common IDE-features like, for example, find references and go to definition, in a diff view. This inspired us to design a system that allows developers to quickly access other parts of the software project in the diff view.

6.2 Code review tools

In this section we give an overview of code review tools used in practice as well as some research projects.

6.2.1 Tools designed by researchers

In this section we present some tools designed by researchers with the goal to improve the code review process directly or help with understanding software changes.

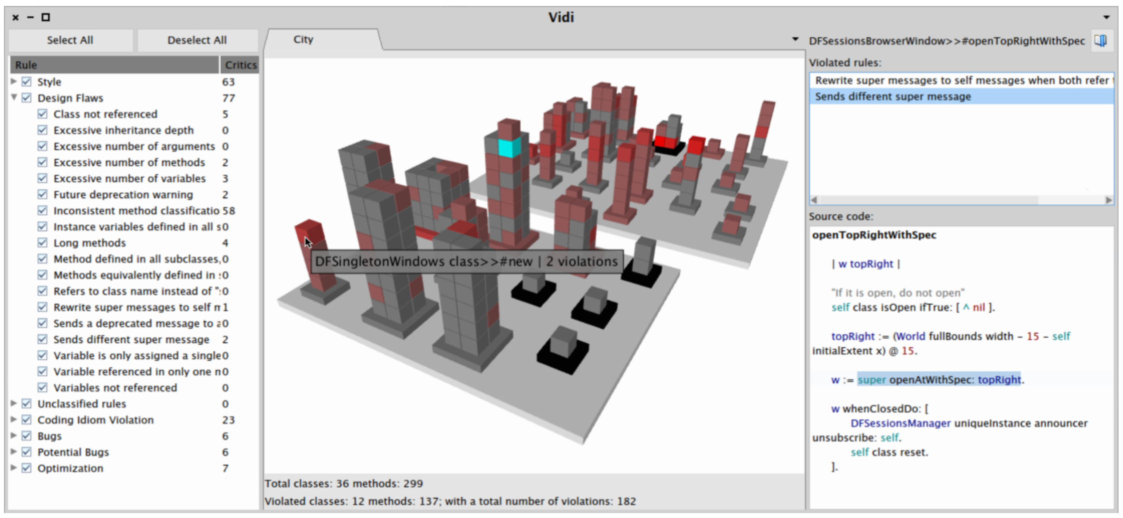
Visual Design Inspection

Visual Design Inspection (ViDI) [25] is a work in progress prototype of a code review tool, which uses static code analysis and visualizations techniques to allow the user to inspect the design of the software. Figure 6.1 shows the main view of the tool. They aim to improve change inspection by providing static analysis information which can hint to problems in the code. Our approach is focused on visualizing changes and provide additional information which facilitates change understanding. It would be interesting to combine both approaches to further facilitate change understanding and inspection.

Tree matching algorithms

Falleri et al. [11] designed GumTree - an algorithm that computes fine-grained edit scripts on ASTs, while also taking move actions into account. Their evaluation shows that the results of the algorithm are good and often do a better job in helping to understand changes than compared to the `diff` tool. ChangeDistiller is a similar algorithm for computing fine-grained change information [12]. It allows to categorize changes in more detail than the usual diff which is only text based and detects insertions and deletions. These algorithms enable more precise diffs which may help in code reviews. Envision does not need such algorithms, since unlike other tools it uses the AST to represent code and

[1]`http://www.qt.io/`
[2]`http://www.vtk.org/`
[3]`http://www.itk.org/`


Vidi
Select All
Deselect All
City
DFSingletonWindows class>>#new | 2 violations
DFSessionsBrowserWindow>>#openTopRightWithSpec
Violated rules:
Source code:

Rule	Critics
☑ Style	63
☑ Design Flaws	77
☑ Class not referenced	5
☑ Excessive inheritance depth	0
☑ Excessive number of arguments	0
☑ Excessive number of methods	2
☑ Excessive number of variables	3
☑ Future deprecation warning	2
☑ Inconsistent method classificatio	58
☑ Instance variables defined in all s	0
☑ Long methods	4
☑ Method defined in all subclasses,	0
☑ Methods equivalently defined in :	0
☑ Refers to class name instead of ":	0
☑ Rewrite super messages to self n	1
☑ Sends a deprecated message to a	0
☑ Sends different super message	2
☑ Variable is only assigned a single	0
☑ Variable referenced in only one n	0
☑ Variables not referenced	0
☑ Unclassified rules	0
☑ Coding Idiom Violation	23
☑ Bugs	6
☑ Potential Bugs	6
☑ Optimization	7

Total classes: 36 methods: 299
Violated classes: 12 methods: 137; with a total number of violations: 182

Rewrite super messages to self messages when both refer
Sends different super message

```
openTopRightWithSpec

    | w topRight |

    "If it is open, do not open"
    self class isOpen ifTrue: [ ^ nil ].

    topRight := (World fullBounds width - 15 - self
initialExtent x) @ 15.

    w := super openAtWithSpec: topRight.

    w whenClosedDo: [
        DFSessionsManager uniqueInstance announcer
unsubscribe: self.
        self class reset.
    ].
```

Figure 6.1: Main window of ViDI. On the left side an overview over potential issues in the software is displayed. The software project with annotated critics is displayed in the middle. The right side of the view shows detailed information about the selected entity: the top shows critics of the selected component and the bottom shows the accompanying source code.

is able to uniquely identify every node. This allows it to compute and track fine-grained change information for the different nodes.

Diff/TS

Diff/TS [16] is a tool which, similar to ChangeDistiller and GumTree, allows to analyze structural changes made to program code. The tool provides the fine-grained edit operations, including moves, made on source code to transform the old version of the code to the new changed version.

Griotte

Lentz et al. [19] developed Griotte, which exploits the information provided by fine-grained IDE events to help developers overcome difficulties during code review. They mention the difficulties of having inaccurate or wrong commit messages, scattered changes, tangled commits and shadowed changes. They also propose an approach using fine-grained IDE events to group mutually related changes. Our implementation of the automated ordering and grouping is a similar approach, which allows to use all the available fine-grained information of the changes to design strategies which order and group changes.

Torch

Gomez et al. [13] designed Torch. A tool to characterize changes by considering structural information as well as authors and symbolic information. The information provided by the tool uses a mixture consisting of textual diff information and visual representations. They also include other metrics characterizing the displayed changes. This should help release masters to decide, whether new changes should be merged into the system release of a software.

6.2.2 Tools used in industry

There are a multitude of code review tools available which are used by thousands of developers. We focus our discussion on GitHub, which is the largest open-source code-hosting platform. We also give a short overview of some other well-recognized tools.

GitHub

GitHub[4] is a hosting service for git repositories. It employs a Pull Request based review process: new features or fixes are usually developed by branching off the main branch, so as not to interfere with it during the implementation of the new features or bug fixes. After implementing and committing all the changes on this new feature branch, a Pull Request starts the review process, where the proposed changes are discussed and potential fixes are committed. After finishing the code review and fixing all the issues the feature branch is merged into the main branch. Figure 6.2 shows an illustration of this process.

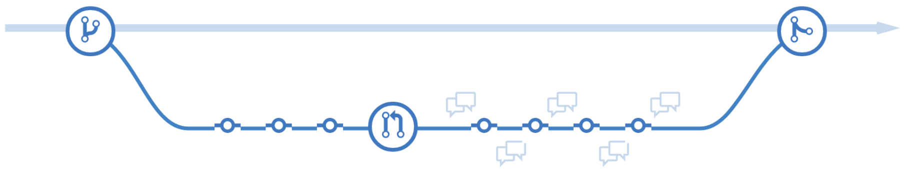
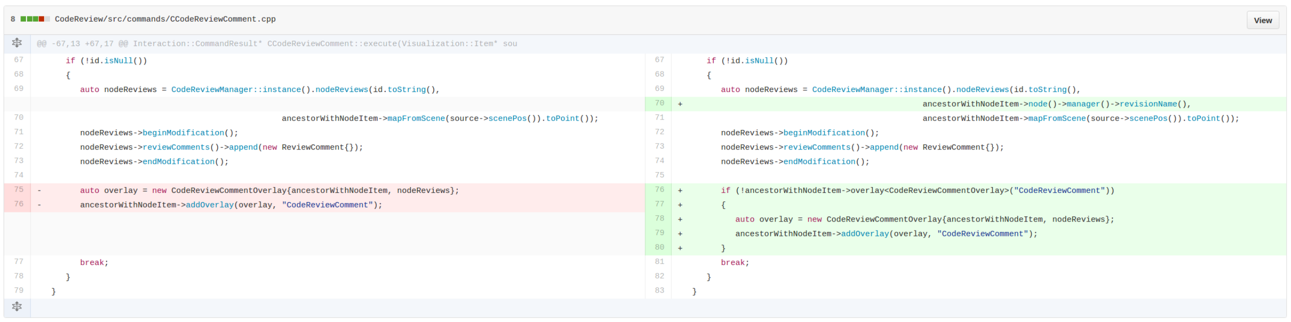


Figure 6.2: Review process in GitHub. After creating a new branch and adding commits to it a Pull Request is opened. After discussion the changes and, if needed, adding more commits to fix issues this branch is merged into the main branch.

Reviewers can choose to see the diff of all changes at once or only per commit of the Pull Request. GitHub supports two visualizations of the diff: an unified view and a split view.



8 CodeReview/src/commands/CCodeReviewComment.cpp

```
@@ -67,13 +67,17 @@ Interaction::CommandResult* CCodeReviewComment::execute(Visualization::Item* sou
```

```
      if (!id.isNull())
      {
         auto nodeReviews = CodeReviewManager::instance().nodeReviews(id.toString(),

                                                ancestorWithNodeItem->mapFromScene(source->scenePos()).toPoint());
         nodeReviews->beginModification();
         nodeReviews->reviewComments()->append(new ReviewComment{});
         nodeReviews->endModification();

-        auto overlay = new CodeReviewCommentOverlay{ancestorWithNodeItem, nodeReviews};
-        ancestorWithNodeItem->addOverlay(overlay, "CodeReviewComment");



         break;
      }
   }
```

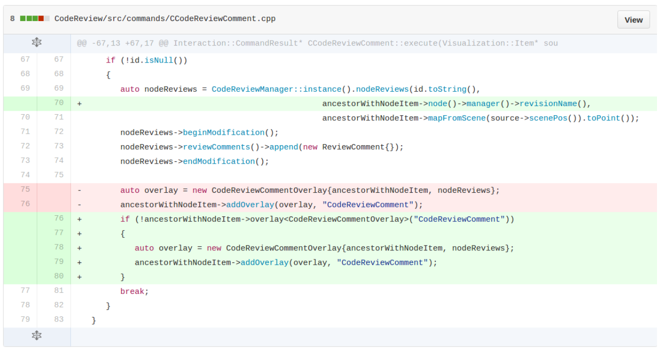
```
      if (!id.isNull())
      {
         auto nodeReviews = CodeReviewManager::instance().nodeReviews(id.toString(),
+                                               ancestorWithNodeItem->node()->manager()->revisionName(),
                                                ancestorWithNodeItem->mapFromScene(source->scenePos()).toPoint());
         nodeReviews->beginModification();
         nodeReviews->reviewComments()->append(new ReviewComment{});
         nodeReviews->endModification();

+        if (!ancestorWithNodeItem->overlay<CodeReviewCommentOverlay>("CodeReviewComment"))
+        {
+           auto overlay = new CodeReviewCommentOverlay{ancestorWithNodeItem, nodeReviews};
+           ancestorWithNodeItem->addOverlay(overlay, "CodeReviewComment");
+        }
         break;
      }
   }
```

Figure 6.3: Split diff view in GitHub. The old version of the including with possible deletions is displayed on the left, the new version of the code featuring possible insertions on the right.

Figure 6.3 shows the split view of GitHub. It shows the old version of the code on the left and the new version on the right. This means deletions are located on the left side and insertions on the right. The color of the highlights indicates the change type: green for insertions and red for deletions.

In the unified view, shown in Figure 6.4, only one version of the code is displayed, showing insertions and deletions in the same version, providing a more dense view of the changes.

[4] https://github.com

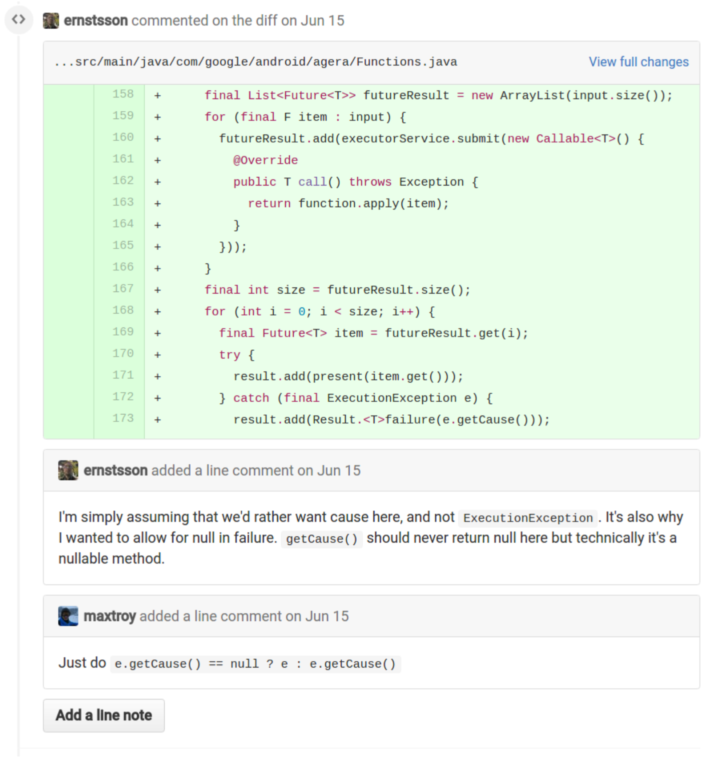


8 CodeReview/src/commands/CCodeReviewComment.cpp View

```
@@ -67,13 +67,17 @@ Interaction::CommandResult* CCodeReviewComment::execute(Visualization::Item* sou
          if (!id.isNull())
          {
             auto nodeReviews = CodeReviewManager::instance().nodeReviews(id.toString(),
+                                                       ancestorWithNodeItem->node()->manager()->revisionName(),
                                                        ancestorWithNodeItem->mapFromScene(source->scenePos()).toPoint());
             nodeReviews->beginModification();
             nodeReviews->reviewComments()->append(new ReviewComment{});
             nodeReviews->endModification();

-            auto overlay = new CodeReviewCommentOverlay{ancestorWithNodeItem, nodeReviews};
-            ancestorWithNodeItem->addOverlay(overlay, "CodeReviewComment");
+            if (!ancestorWithNodeItem->overlay<CodeReviewCommentOverlay>("CodeReviewComment"))
+            {
+               auto overlay = new CodeReviewCommentOverlay{ancestorWithNodeItem, nodeReviews};
+               ancestorWithNodeItem->addOverlay(overlay, "CodeReviewComment");
+            }
             break;
          }
       }
```

Figure 6.4: Unified diff view in GitHub. The displayed version is a merge of the old and new version. This allows GitHub to display all the changes in this single version.

It is possible to switch between those two visualizations at any point in time, letting the user decide which view is preferred.



ernstsson commented on the diff on Jun 15

...src/main/java/com/google/android/agera/Functions.java View full changes

```
+      final List<Future<T>> futureResult = new ArrayList(input.size());
+      for (final F item : input) {
+        futureResult.add(executorService.submit(new Callable<T>() {
+          @Override
+          public T call() throws Exception {
+            return function.apply(item);
+          }
+        }));
+      }
+      final int size = futureResult.size();
+      for (int i = 0; i < size; i++) {
+        final Future<T> item = futureResult.get(i);
+        try {
+          result.add(present(item.get()));
+        } catch (final ExecutionException e) {
+          result.add(Result.<T>failure(e.getCause()));
```

ernstsson added a line comment on Jun 15

I'm simply assuming that we'd rather want cause here, and not `ExecutionException`. It's also why I wanted to allow for null in failure. `getCause()` should never return null here but technically it's a nullable method.

maxtroy added a line comment on Jun 15

Just do `e.getCause() == null ? e : e.getCause()`

Figure 6.5: In-line comments in GitHub's diff view. The button Add a line note allows users to add comments to the current discussion.

GitHub, like every code review tool, provides the possibility to add in-line review comments to the changes in the diff view. Figure 6.5 shows an example of this. It is also possible to add general comments relating to the entire Pull Request. This is the main communication tool to give feedback and discuss the changes in GitHub.

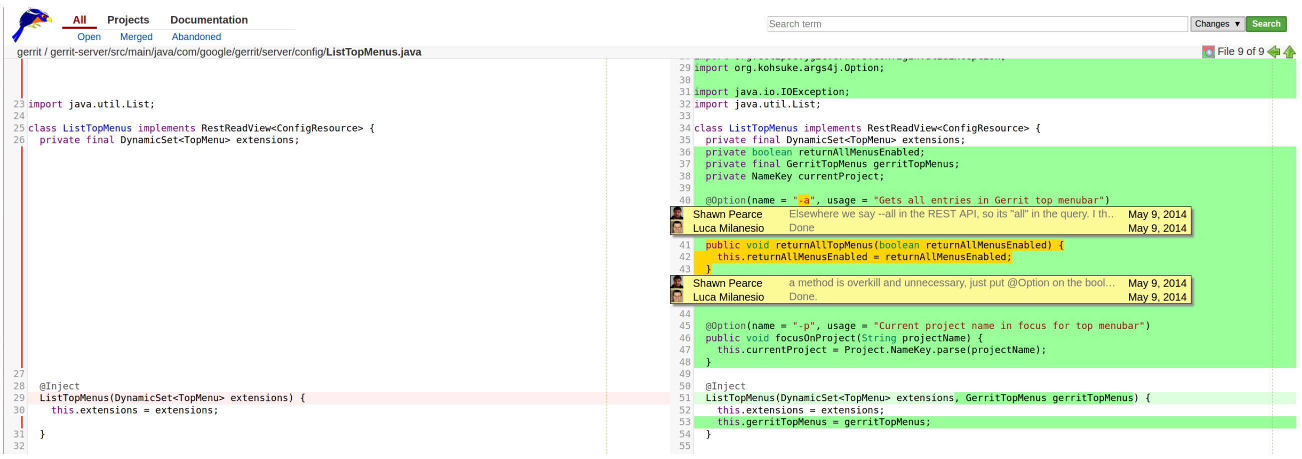
Thanks to the Pull Request feature it provides a clear work flow for code reviews. The possibility to add commits to the same Pull Request allows developers to discuss everything related to a specific feature, i.e., the initial commits as well as later commits which fix issues that came up during review, in the same place. Sometimes the textual matchings in the diffs of GitHub are not intuitive and can hinder change understanding.

This combined with, at times, lacking context in the default view of the changes could be improved. We discuss these difficulties in more detail in Section 2.1.

Other review tools

We present some further code review tools, while mainly focusing on their diff visualization or other interesting abilities.

Gerrit[5] is an open-source web-based code review tool developed at Google for the Android project. It uses the Git version control system and offers a side-by-side view, which can be seen in Figure 6.6, for the diff of the changes.



gerrit / gerrit-server/src/main/java/com/google/gerrit/server/config/**ListTopMenus.java**

```
import java.util.List;

class ListTopMenus implements RestReadView<ConfigResource> {
  private final DynamicSet<TopMenu> extensions;

  @Inject
  ListTopMenus(DynamicSet<TopMenu> extensions) {
    this.extensions = extensions;

  }

```

```
import org.kohsuke.args4j.Option;

import java.io.IOException;
import java.util.List;

class ListTopMenus implements RestReadView<ConfigResource> {
  private final DynamicSet<TopMenu> extensions;
  private boolean returnAllMenusEnabled;
  private final GerritTopMenus gerritTopMenus;
  private NameKey currentProject;

  @Option(name = "-a", usage = "Gets all entries in Gerrit top menubar")
```

Shawn Pearce	Elsewhere we say --all in the REST API, so its "all" in the query. I th...	May 9, 2014
Luca Milanesio	Done	May 9, 2014

```
  public void returnAllTopMenus(boolean returnAllMenusEnabled) {
    this.returnAllMenusEnabled = returnAllMenusEnabled;
  }
```

Shawn Pearce	a method is overkill and unnecessary, just put @Option on the bool...	May 9, 2014
Luca Milanesio	Done.	May 9, 2014

```

  @Option(name = "-p", usage = "Current project name in focus for top menubar")
  public void focusOnProject(String projectName) {
    this.currentProject = Project.NameKey.parse(projectName);
  }

  @Inject
  ListTopMenus(DynamicSet<TopMenu> extensions, GerritTopMenus gerritTopMenus) {
    this.extensions = extensions;
    this.gerritTopMenus = gerritTopMenus;
  }

```

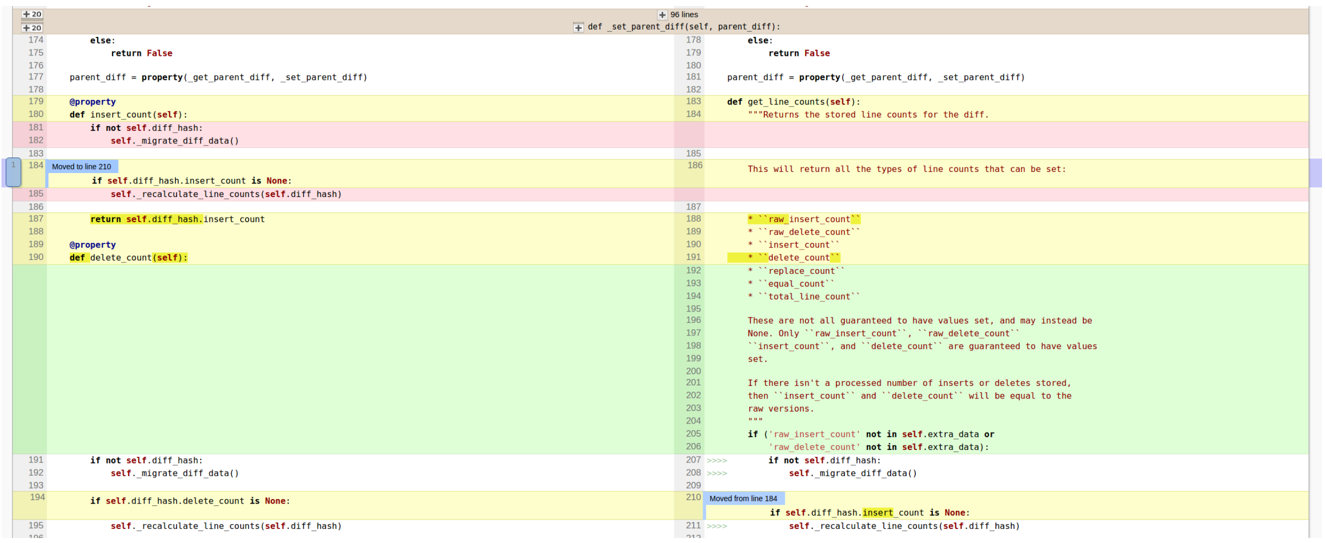
Figure 6.6: Diff view in Gerrit. The line-comments are displayed in between the code lines.

It employs a similar process like GitHub's Pull Request system.

ReviewBoard[6] provides a side-by-side diff view shown in Figure 6.7. Unlike most of the other tools it provides some mechanisms to detect moved lines.

Upsource[7] is, according to JetBrains, the only code review tool which enables code-aware navigation for Java, PHP, JavaScript and Kotlin [17]. Figure 6.8 shows some of the code-aware function available for the inspection of a class. It allows users to display the diff in a side-by-side or unified view similar to GitHub.

[5]`https://www.gerritcodereview.com/`
[6]`https://www.reviewboard.org/`
[7]`https://www.jetbrains.com/upsource`

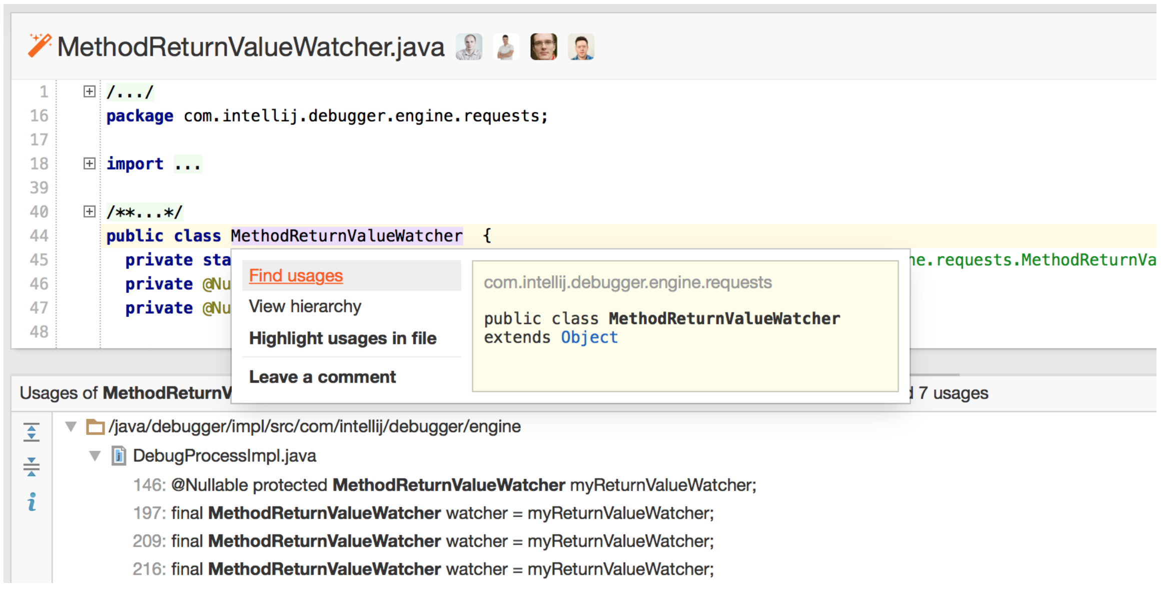


```
+20
+20                                                   + 96 lines
                                                      + def _set_parent_diff(self, parent_diff):
```

```
        else:
            return False

    parent_diff = property(_get_parent_diff, _set_parent_diff)

    @property
    def insert_count(self):
        if not self.diff_hash:
            self._migrate_diff_data()

Moved to line 210
        if self.diff_hash.insert_count is None:
            self._recalculate_line_counts(self.diff_hash)

        return self.diff_hash.insert_count

    @property
    def delete_count(self):
        if not self.diff_hash:
            self._migrate_diff_data()

        if self.diff_hash.delete_count is None:
            self._recalculate_line_counts(self.diff_hash)
```

```
        else:
            return False

    parent_diff = property(_get_parent_diff, _set_parent_diff)

    def get_line_counts(self):
        """Returns the stored line counts for the diff.

        This will return all the types of line counts that can be set:

        * ``raw_insert_count``
        * ``raw_delete_count``
        * ``insert_count``
        * ``delete_count``
        * ``replace_count``
        * ``equal_count``
        * ``total_line_count``

        These are not all guaranteed to have values set, and may instead be
        None. Only ``raw_insert_count``, ``raw_delete_count``
        ``insert_count``, and ``delete_count`` are guaranteed to have values
        set.

        If there isn't a processed number of inserts or deletes stored,
        then ``insert_count`` and ``delete_count`` will be equal to the
        raw versions.
        """
        if ('raw_insert_count' not in self.extra_data or
            'raw_delete_count' not in self.extra_data):
>>>>        if not self.diff_hash:
>>>>            self._migrate_diff_data()

Moved from line 184
            if self.diff_hash.insert_count is None:
>>>>            self._recalculate_line_counts(self.diff_hash)
```

Figure 6.7: *Diff view in ReviewBoard. Within a file, it displays the origin and destination of unchanged, moved lines using blue highlights.*



MethodReturnValueWatcher.java

```
/.../
package com.intellij.debugger.engine.requests;

import ...

/**...*/
public class MethodReturnValueWatcher  {
  private sta                                                  ne.requests.MethodReturnVa
  private @Nu
  private @Nu
```

```
com.intellij.debugger.engine.requests

public class MethodReturnValueWatcher
extends Object
```

Usages of MethodReturnV   7 usages

/java/debugger/impl/src/com/intellij/debugger/engine
DebugProcessImpl.java
146: @Nullable protected **MethodReturnValueWatcher** myReturnValueWatcher;
197: final **MethodReturnValueWatcher** watcher = myReturnValueWatcher;
209: final **MethodReturnValueWatcher** watcher = myReturnValueWatcher;
216: final **MethodReturnValueWatcher** watcher = myReturnValueWatcher;

Figure 6.8: *Code insight for Java in Upsource. The displayed box shows different options available for inspecting other components related to the selected class. The bottom list shows the result of selecting Find usages.*

Chapter 7

Conclusion & Future Work

In this thesis we laid the foundation for a code review tool, that improves developers' experience. We designed visualizations for fine-grained changes, which can be divided into four categories: insertions, deletions, moves and modification. We presented advanced diff features to help improve change understanding further. The history feature allows developers to inspect the evolution of components of the software system. The overview feature allows developers to display changes in the context of an entire software project. To improve readability of the changes during a review and allow reviewers to focus on more substantial changes, we designed a name change summary function, which can filter changes related to a name refactoring from the view and summarize the refactoring as a textual information.

To allow users to communicate during the code review process we designed code review comments, which can be annotated with focus information. Focus information allows the author of the changes to provide guidance for the reviewer by defining the order of steps in which a change set should be inspected. Additionally we created a mechanism, which allows users to choose a strategy for automatically grouping and ordering changes, e.g., group changes which are related together. according to strategies chosen by the user. The mechanism is extensible, making it possible to add new strategies to their likening.

To test the effectiveness of our approach for diff visualizations and code review improvements it would be important to conduct an evaluation by performing a user study. It would be interesting to test the tool during real code reviews to get an idea, in which areas its strength and weaknesses are.

Further work should be invested to add more tools to help the developer answer questions related to the quality of code changes. Static analysis tools could help developers to assert the correctness and completeness of changes. Such static analysis tools could also help in providing automatic decomposition of changes: instead of having a large changeset containing a mixture of bug-fixes and features related to different areas it would be advantageous to decompose such changes in smaller sets handling individual features or bug-fixes.

The user interface for the code reviews in general could need more polish. One area concerned would be to improve the organizing of changes in order to avoid too large spatial distance between related changes (e.g., origin and destination of a move).

In order to perform code-reviews asynchronously it would be important to introduce a proper user-system in Envision and provide a way of synchronizing code review comments over the internet.

Envision features mechanisms which allow users to merge changes. To enable users to decide how to merge conflicting areas of the code it is necessary to provide a user interface to interact with the merging tools.

Appendices

Appendix A

Overview of Commands

Command / Keyboard shortcut	Description
diff	Display diff of HEAD revision against working directory.
diff *rev*	Display diff of the specified revision *rev* against the working directory.
diff *old_rev new_rev*	Display diff of *old_rev* against *new_rev*.
diff *old_rev new_rev* **overview**	Display diff *old_rev* against *new_rev*, but display changes in current view. The **overview** flag always needs to follow two specified revisions.
history	Open history view for the selected component. Show history starting with the first commit involved with the selected component.
history *rev*	Open history view for the selected component. Show history starting at *rev* involved with the selected component.
review	Display review view using diff of HEAD revision against working directory.
review *rev*	Display review view using diff of the specified revision *rev* against the working directory.
review *old_rev new_rev*	Display review view using diff of *old_rev* against *new_rev*.
review save	Persist review comments of current review view. Review comments are automatically loaded if available by using **review** commands.
comment	Used in review view to add a review comment to the selected node.
clear	Removes scaling and highlights added through diff commands. Used, e.g., to clean project view after using **overview** option for diff command.
focus **<CTRL> + <F12>**	Step through focus information in review view, if available.
<CTRL> + <SHIFT> + <F10>	Toggle through different options of name change highlights to shown. There are three options: First, show everything related to name changes. Second, show only name change definitions. Third, show nothing related to name changes.

Table A.1: Commands related to diff and reviews.

Bibliography

[1] Dimitar Asenov and Peter Muller. Envision: A fast and flexible visual code editor with fluid interactions (overview). In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 9–12. IEEE, 2014.

[2] Dimitar Asenov, Otmar Hilliges, and Peter Müller. The effect of richer visualizations on code comprehension. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5040–5045. ACM, 2016.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144. IEEE, 2015.

[5] Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142. IEEE, 2013.

[6] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: an empirical study at microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 146–156. IEEE Press, 2015.

[7] Joe Brockmeier. A look at phabricator: Facebook's web-based open source code collaboration tool, 2011. URL `http://readwrite.com/2011/09/28/a-look-at-phabricator-facebook/`. Last visited 2016-08-23.

[8] Stéphane Conversy. Unifying textual and visual: a theoretical account of the visual perception of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 201–212. ACM, 2014.

[9] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 27–28. IEEE Press, 2015.

[10] Sascha Dinkel. Enhancing the visual documentation artifacts in envision. ETH Zürich, 2014.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[12] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[13] Veroonica Uquillas Gomez, Stéphane Ducasse, and Theo D'Hondt. Visually supporting source code changes integration: the torch dashboard. In *2010 17th Working Conference on Reverse Engineering*, pages 55–64. IEEE, 2010.

[14] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[15] Balz Guenat. Tree-based version control in envision. ETH Zürich, 2015.

[16] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288. IEEE, 2008.

[17] Maria Khalusova. What's new in upsource 3.0, 2016. URL `https://www.youtube.com/watch?v=9kbUn5ikhNo`. Last visited 2016-08-18.

[18] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1028–1038. ACM, 2016.

[19] Skip Lentz, Martín Dias, and Damien Cassou. Griotte: Improving code review with fine-grained ide events. In *BENEVOL'15: 14th BElgian-NEtherlands software eVOLution seminar*, 2015.

[20] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[21] Martin Otth. Fine-grained software version control based on a program's abstract syntax tree. ETH Zürich, 2014.

[22] Guido Van Rossum. Mondrian code review on the web, 2006. URL `https://www.youtube.com/watch?v=sMql3Di4Kgc`. Last visited 2016-08-15.

[23] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[24] Jonas Trappenberg. Supporting documentation artifacts in envision. ETH Zürich, 2013.

[25] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. Code review: Veni, vidi, vici. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 151–160. IEEE, 2015.

[26] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 111–122. IEEE Press, 2015.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Improving Code Reviews using the Envision IDE

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Galbier	Manuel

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 01.09.2016	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.