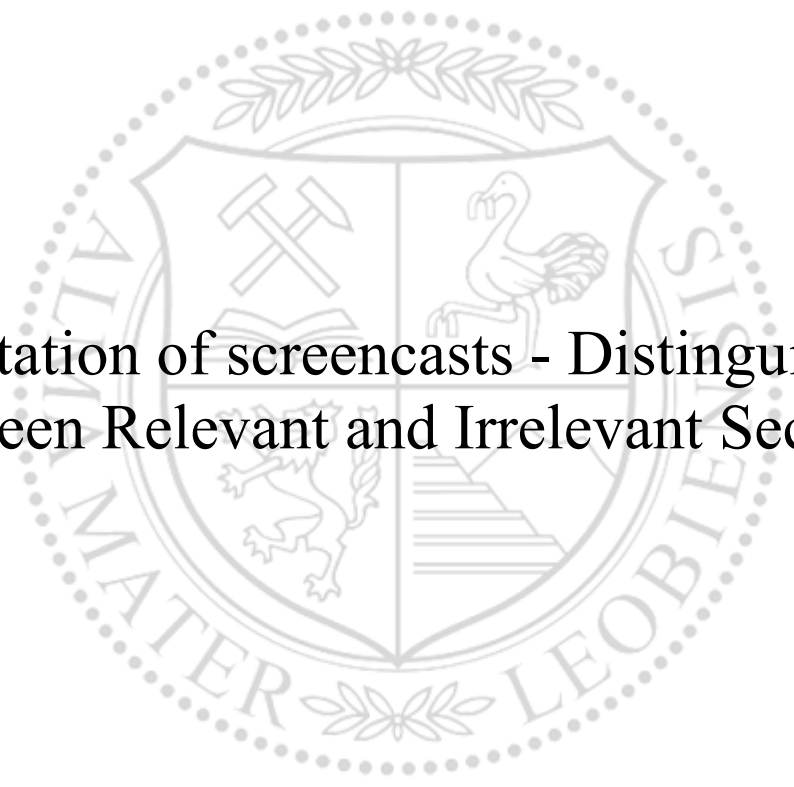




Chair of Information Technology

Master's Thesis



Annotation of screencasts - Distinguishing  
Between Relevant and Irrelevant Sections

Tabea Ulm, BSc

May 2022

# Affidavit

I declare in lieu of oath, that I wrote this thesis and performed the associated research myself, using only literature cited in this volume.

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Leoben, am \_\_\_\_\_

Datum

\_\_\_\_\_

Unterschrift

# Abstract

This thesis proposes a method to annotate screencasts, in order to identify sections of significance. The proposed approach quantifies the relevance frame by frame over the duration of the recording, making it easier for an external observer to navigate to sections of interest. Within this work, we implemented an approach for annotating screencasts of programming activities. Given a recording of screencasts only, the proposed method measures the amount of written code between each pair of subsequent frames. The approach is divided into three steps: extracting the code editor of a development environment, separating individual characters within those regions, and finally analyzing changes of those characters between subsequent frames. The detection of code editors is performed using computer vision methods that detect features characteristic for those regions. Character segmentation algorithms are then applied to the detected regions, in order to decide whether it contains a monospaced font, as this is a distinct attribute for fonts used in code editors. Changes in those characters are then analyzed, taking into account possible disturbances. The results were evaluated using 56 screencasts. The recordings originated from three different programming exercises, completed by 20 different students, each student using one of two development environments. The evaluation of those recordings result in a median accuracy of 83.4% with a median  $F_2$  score of 81.5%.

# Kurzzusammenfassung

Diese Arbeit beschreibt eine Methode, um Screencasts zu annotieren und signifikante Abschnitte zu identifizieren. Der beschriebene Ansatz definiert ein Maß über die Dauer der Aufzeichnung, das die Relevanz von Abschnitten quantifiziert. Ziel ist es, einem externen Betrachter zu ermöglichen, schnell zu relevanten Stellen in der Aufzeichnung zu navigieren. Dieser Ansatz wurde für Screencasts von Programmieraktivitäten implementiert, indem ein Maß für die Menge an getippten Text zwischen Paaren von benachbarten Frames ausgegeben wird. Der hier beschriebene Ansatz lässt sich in drei Unteraufgaben teilen: Extrahieren des Code-Editors, Erfassen einzelner Zeichen und schließlich die Bewertung der Unterschiede an Zeichen zwischen aufeinanderfolgenden Frames. Die Detektion der Code-Editoren erfolgt durch Anwendung von Computer Vision Methoden. Dabei werden charakteristische Merkmale detektiert und zur Rekonstruktion der Editor-Fenster verwendet. In den detektierten Regionen werden anschließend Algorithmen zur Zeichensegmentierung angewandt und anhand der Ergebnisse beurteilt, ob sich der Inhalt, um Text in einer Monospace-Schriftart handelt. Diese sind charakteristisch für Text Editoren von Entwicklungsumgebungen und werden deswegen zur Klassifizierung der detektierten Bereiche verwendet. Abschließend werden die geänderten segmentierten Zeichen evaluiert. Die Ergebnisse wurden anhand von 56 Screencasts evaluiert. Die Aufzeichnungen stammen von drei unterschiedlichen Programmieraufgaben und wurden von 20 Studierenden gelöst. Die Studierenden verwendeten dafür eine von zwei Entwicklungsumgebungen. Die Evaluierung dieser Screencasts ergab einen mittleren Genauigkeitswert von 83.4% und ein mittleres  $F_2$ -Maß von 81.5%.

# Contents

<b>Affidavit</b>	<b>I</b>
<b>Abstract</b>	<b>II</b>
<b>Kurzzusammenfassung</b>	<b>III</b>
<b>List of Figures</b>	<b>VII</b>
<b>List of Tables</b>	<b>IX</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Motivation . . . . .	1
1.2 Research Theme . . . . .	1
1.2.1 Relevant Image Sections . . . . .	2
1.2.2 Relevant Actions . . . . .	2
1.3 Evaluation Method . . . . .	3
1.3.1 Evaluation of Available Screencasts . . . . .	3
1.3.2 Separation Into Implementation and Evaluation Data . . . . .	4
1.3.3 Method of Performance Evaluation . . . . .	6
<b>2 Related Work</b>	<b>9</b>
2.1 Transcribing Code . . . . .	9
2.1.1 CodeTube . . . . .	9
2.1.2 ACE . . . . .	10
2.1.3 psc2code . . . . .	10
2.1.4 Codemotion . . . . .	11
2.2 Locating Source Code . . . . .	11
2.2.1 Identifying Source Code . . . . .	11
2.2.2 Identifying Code Fragments . . . . .	12
2.3 Event Detection . . . . .	12
2.3.1 Classifying Actions . . . . .	12
2.4 Significance of This Work . . . . .	13

<b>3</b>	<b>Algorithmic Approach</b>	<b>15</b>
3.1	Initial Definitions . . . . .	15
3.2	Frame Layout Detection . . . . .	16
3.2.1	Emphasize Relevant Features . . . . .	18
3.2.2	Extract Border Parallel Lines . . . . .	21
3.2.3	Construct Possible ROIs . . . . .	23
3.2.4	Analyze Geometric Properties . . . . .	25
3.2.5	Group Frames According to Frame Layout . . . . .	28
3.3	Grid Analysis . . . . .	29
3.3.1	Row and Column Period . . . . .	30
3.3.2	Period per Frame . . . . .	36
3.3.3	Period per Section . . . . .	36
3.3.4	Text Editor Classification . . . . .	39
3.4	Analysis of Textual Changes . . . . .	40
3.4.1	Preprocessing . . . . .	41
3.4.2	Determining Change Values . . . . .	42
3.4.3	Processing of Larger Textual Changes . . . . .	44
3.5	Discarded Approaches . . . . .	49
<b>4</b>	<b>Implementation</b>	<b>50</b>
4.1	Usage Specifications . . . . .	50
4.1.1	Requirements for Input . . . . .	50
4.1.2	Resulting Data . . . . .	51
4.2	Project Structure . . . . .	52
4.2.1	Implementation of Frame Layout Detection . . . . .	53
4.2.2	Implementation of Grid Analysis . . . . .	53
4.2.3	Implementation of Analysis of Textual Changes . . . . .	54
<b>5</b>	<b>Evaluation</b>	<b>55</b>
5.1	Setup . . . . .	55
5.1.1	Ground Truth . . . . .	55
5.1.2	Technical Setup . . . . .	56
5.2	Results . . . . .	56
5.2.1	Evaluation of Accuracy . . . . .	56
5.2.2	Evaluation of Precision and Recall . . . . .	58
5.2.3	Analysis of <i>a-typing</i> . . . . .	60
5.2.4	Evaluation of Larger Textual Changes . . . . .	63
5.2.5	Exemplary Study for Night Mode . . . . .	64

<b>6 Conclusion</b>	<b>66</b>
6.1 Summary . . . . .	66
6.2 Further Research . . . . .	66
<b>Appendix</b>	<b>XV</b>
I Input Data Analysis . . . . .	XV
II UML class diagramm . . . . .	.XVII
III Code Documentation . . . . .	.XVII

# List of Figures

1.1	Distribution of available inputs according to IDE and UI settings. . . .	4
3.1	Visualization of the coordinate system. . . . .	15
3.2	Input image for ROI detection. . . . .	17
3.3	Examples of one-dimensional edge detection. . . . .	19
3.4	Pixels considered for derivative calculation. . . . .	19
3.5	Input images; Estimated vertical derivatives; Estimated horizontal derivatives. . . . .	21
3.6	Example image for Hough Transform. . . . .	22
3.7	Representation of Hough space for the example image. . . . .	22
3.8	Right: corner region; Left: all detected lines. . . . .	24
3.9	Right: all detected lines; Left: reduced set of lines. . . . .	24
3.10	Example of partial and full overlap. . . . .	25
3.11	Input image; Green: all detected ROIs; Red: resulting ROIs . . . . .	26
3.12	Remaining ROIs for different IDEs. . . . .	27
3.13	All detected possible ROIs; Remaining possible ROIs with hierarchy. . . . .	28
3.14	Relevant ROI for different IDEs. . . . .	30
3.15	Input image for row and column detection. . . . .	31
3.16	Example image after applying an adaptive threshold. . . . .	32
3.17	Average pixel values per row (left) and column (right). . . . .	33
3.18	Autocorrelation results per row (left) and column (right). . . . .	35
3.19	Detected grid for example image. . . . .	35
3.20	Comparison of all detected ROIs (left) and the resulting ROIs of this method (right). . . . .	36
3.21	Example ROI structure for determining period values per section. . . . .	37
3.22	Demonstration of voting algorithm. . . . .	38
3.23	Resulting ROI structure for the example. . . . .	39
3.24	Example of additive and subtractive change images. . . . .	42
3.25	Example of grid cells for which a change occurred. . . . .	43
3.26	Example of change images with horizontal shift and area of largest coherent change. . . . .	45
3.27	Representation of subparts for subsequent images with a shift. . . . .	46



---

3.28	Example of change images with autocomplete box. . . . .	47
3.29	Possible rectangles for further processing. . . . .	48
4.1	Interactions between logic classes. . . . .	53
5.1	Box plot of accuracy. . . . .	57
5.2	Scatter plot of precision and recall. . . . .	59
5.3	Box plot of $F_2$ score. . . . .	60
5.4	ROC curve. . . . .	62
5.5	Precision-Recall curve. . . . .	63
I	Package <code>videoanalysis</code> : classes and their dependencies . . . . .	XVII

# List of Tables

3.1	Overview of methods used for ROI detection. . . . .	18
3.2	Overview of methods used for period detection. . . . .	31
5.1	Confusion matrix. . . . .	56
5.2	Resulting $p$ -values for group pairings. . . . .	58
5.3	Relative rank of larger textual changes. . . . .	64
5.4	Results of exemplary study. . . . .	64

# 1. Introduction

## 1.1. Problem Motivation

Due to the exceptional situation caused by the COVID-19 pandemic, lectures are increasingly held remotely, making interactions between students and teachers more difficult. Therefore, this situation calls for unconventional teaching methods. One possibility for distance learning is that students record their screens, and the instructors then evaluate the screencasts. Compared to other monitoring techniques, such as tracking key board inputs, display recordings have the advantage of being non-invasive. Starting a recording takes little to no effort for students and teachers and may not even require additional applications on students' devices. However, evaluation of these recordings is very labor-intensive, since every recording has to be scanned for every student who takes part in the course.

For this reason, it would be very convenient to have a program that recognizes relevant time codes in a video recording in order to reduce the manual effort required for evaluation. Relevant time codes are those in which the user carries out defined activities. The time codes of these activities are to be made available to the evaluating persons. Based on these timestamps, it should be possible to better navigate a screen recording.

## 1.2. Research Theme

Within this work the following question shall be answered:

**RQ:** How can relevant semantic content in screencasts be detected?

The input data for this work consists of screencasts from programming activities. The relevant content to be detected for this use case is the modification of text inside a development environment. We therefore want to identify text editors and analyze the actions within. To achieve this goal, the research question can be divided as follows:

**RQ1:** How can semantically significant areas for programming activities in a screen recording be detected?

**RQ2:** Within those significant areas, which methods can be applied to identify text changes by users?

### 1.2.1. Relevant Image Sections

The screencasts used for this work are evaluated as a consecutive sequence of images. Each image is denoted as a frame. Within each frame we want to define the areas that are considered to be semantically relevant.

For this particular use case, we are only focusing on programming activities. Therefore, semantically relevant sections are text editors in which code can be written. As each user has different preferences regarding development environment and user interface, no additional assumptions regarding the representation of the code like fonts or coloring can be made.

The default setting in all prevailing operating systems is, that text from the keyboard can only be entered into windows that are located at the topmost layer. Consequently, we can restrict the analysis of textual changes to such windows.

### 1.2.2. Relevant Actions

Within the marked areas, we want to identify times in which a user performs relevant actions. For solving a programming exercise, semantically relevant actions are foremost the modification of text.

For further description of those relevant actions, we need to understand how changes within a video can be detected. As mentioned, a video is described as a consecutive sequence of images. We will refer to the number of images as  $n$ . Each of those frames consists of a fixed number of pixels. Whenever an action is performed, it results in the change of pixel values between two consecutive frames.

Of course, not all changes in pixel values are semantically relevant. As mentioned in Section 1.2.1, we restrict the area of evaluation to code editors, which are located on the topmost layer. When programming, not semantically relevant actions will occur inside a code editor as well, for example scrolling, moving the cursor or the pop-up of windows and auto-complete-boxes. Therefore, it becomes necessary to distinguish between relevant and irrelevant actions. As mentioned, we want to restrict this analysis to textual changes. We need to classify all changes in pixel values into relevant and

irrelevant actions.

**Definition 1.2.1.** We define a *typing function* as a discrete function  $t : \{1 \dots n - 1\} \rightarrow \mathbb{Z}$  which measures the textual changes in a fixed time interval.

This function will give a measure for the number of characters which have been changed between two consecutive frames. Notably, the results of the typing function can be positive or negative. A positive value corresponds to additional textual changes like typing characters. Likewise, a negative value is related to a subtractive action like the deletion of text. The replacement of preexisting text results in a change value of zero, as the amount of added and subtracted characters is equal.

**Definition 1.2.2.** Given a frame interval  $[j, k]$ ,  $j < k$ , we say that a user is *a-typing* over this interval, if the sum of the absolute values of  $t$  in this interval lies above a certain threshold  $a$ :

$$\sum_{i=j}^k |t(i)| > a, j, k \in \{1 \dots n - 1\}$$

## 1.3. Evaluation Method

### 1.3.1. Evaluation of Available Screencasts

The available screencasts are from six different Java programming exercises. Each exercise has been done by 28-44 students on their personal computers. Most students participated in more than one exercise. In total there are 225 screencasts available, 8 of which can not be evaluated as the code editor is not visible or not fully visible in these screencasts due to technical difficulties when recorded. In the following analysis those screencasts have been excluded.

As stated in our research theme, the method should detect text editor windows in which code is written. As the detection of editors from screencasts depends on their appearance, our available data will be analyzed in terms of used development environment (IDE) and user interface (UI) settings.

There were no specifications regarding development environment or user interface for the exercises, resulting in a large number of possible settings. However, in all screencasts the students used one of two IDEs. In 154 screencasts (71,0%), the exercises have been solved with BlueJ [13]. In the remaining 63 screencasts (29,0%) the students used Eclipse [8].

Regarding the user interface, all except one student have not made any relevant changes to the default settings of their IDE. Only one student, who used Eclipse, has changed the user interface to night mode.

For further analysis, we group the screencasts according to the used IDE and UI-settings. The following list represents the number of screencasts in each category:

1. 154 (71,0%) screencasts BlueJ  
all with default UI-settings
2. 63 (29,0%) screencasts Eclipse  
57 (26,3%) screencasts with default UI-settings  
6 (2,8%) screencasts with night mode

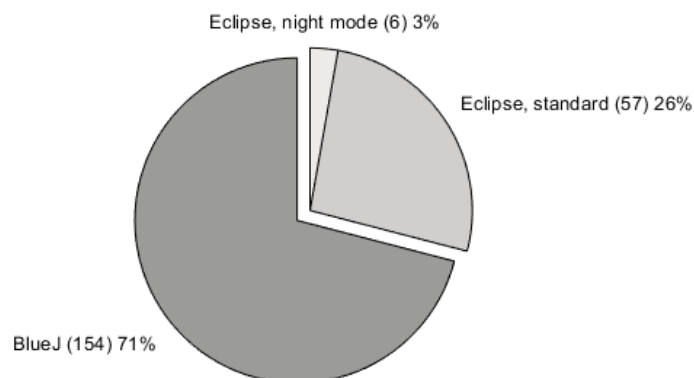


Figure 1.1.: Distribution of available inputs according to IDE and UI settings.

### 1.3.2. Separation Into Implementation and Evaluation Data

The screencasts will be divided into one group for development and a separate group for evaluation. The screencasts in the development group will be used to implement the proposed method and test the code in the implementation phase. The screencasts

in the evaluation group are used to determine the performance of the proposed method. A detailed description of the evaluation method will be given in chapter 1.3.3. Because of the strong dependency on the appearance of code editors, we will include all variations of IDE and UI settings in both groups.

Notably from the data evaluation above, the sample sizes of each subgroup of IDE and UI settings vary significantly in size. In order to make any predictions about the behavior of the proposed method on new students, we need at least four people in each subgroup - two for development and two for evaluation. To ensure a scientific approach, subgroups with less than four students will therefore not be included in development. This restriction concerns the screencasts, which use Eclipse in night mode. For evaluation, those screencasts will be analyzed separately in a qualitative study. This restriction to sample size effects 2,3% of the users and 2,8% of the number of screencasts.

For development and a detailed evaluation of the results, only the following two subgroups of user remain: BlueJ user with default UI settings and Eclipse user with default UI settings. Those two subgroups contain 97,7% of all students and 97,2% of the total number of screencasts. The division of those screencasts will be done to meet the following criteria:

- Half of the exercises (three) must only be used for evaluation.
- Half of all BlueJ users with default UI settings must only be used for evaluation.
- Half of all Eclipse users with default UI settings must only be used for evaluation.

When applying those criteria on the available data, 49 screencasts of three different exercises remain for development. The screencasts are from 22 different students. In 34 of the screencasts, the exercises have been solved using BlueJ and in 15 screencasts Eclipse has been used.

The remaining group of 162 screencasts does also contain students and exercises, which have been used to develop the method and are therefore "known" to our program. Thus, when evaluating the proposed method, we have to distinguish between three groups:

- screencasts of known exercises, done by new students (42)
- screencasts of new exercises, done by known students (64)
- screencasts of new exercises, done by new students (56)

As specified in the introduction, our proposed method will be applied to future exercises and new students. The most conclusive results can therefore be obtained when evaluating the screencasts of new exercises and new students. Consequently, we will restrict the evaluation group to the 56 screencasts in this subgroup.

The appendix includes a detailed list of the available screencasts. All information regarding the number of students per exercise for different IDE and UI settings can be found there (I).

### 1.3.3. Method of Performance Evaluation

As stated in chapter 1.2.2, the goal of our method is to identify textual changes. According to the definition 1.2.2, the detection of *a-typing* depends on the adjustable constant  $a$ . This constant is a threshold for detecting the respective action in the specified interval. The lower the threshold is, the more time intervals will be classified as containing the relevant action. As our goal is to assist people in finding semantically relevant parts of a recording, we have to assume that each person has a subjective view of what they consider to be relevant. Therefore, this constant will remain an input parameter for our method.

When evaluating the proposed method, we first split all given screencasts into one-minute long time intervals. We then manually define for each time interval if typing actions are detected, describing the ideal outcome of our proposed method. Those detection values will be compared to the output function of our program.

For evaluation, we need to calculate the *a-typing* value as specified in the definition 1.2.2 for each interval. Those results will be compared to the manually defined typing activities. Both events are described as a binary outcome: the detection of typing or non-typing in a certain interval and whether a user is actually typing or not. To represent all possible combinations of detected and actual activities, we will use a confusion matrix to visualize the outcome of the proposed method. Within the confusion matrix, the absolute and relative number of intervals in each category are displayed.

For a given value of  $a$  we use, as an initial estimate for the performance of our method, the percentage of accurately detected intervals. The performance of the proposed method is adequate, if for every screencast in the evaluation group, the accuracy is at least two thirds. In addition to accuracy, we will use an analysis of precision and recall as further performance indicators.



When evaluating falsely predicted intervals, we need to consider them in terms of our specific use case. As stated in the research theme, our proposed method should detect typing, with the goal that a tutor can find semantically relevant parts of a screen recording. Typically, a tutor will watch all sections which have been classified as containing typing. If they come across a section which is a false positive, they can easily identify the video section as not relevant and will move on to the next interval. On the other hand, a false negative interval might lead to an extensive search as to when a student is writing certain parts of a code. If a false negative interval directly follows a true positive though, a tutor will most likely resume watching the recording as they can identify this section as relevant.

When evaluating our proposed method, those user behaviors effect the interpretation of our performance indicators. The autonomous identification of false positives means that a high precision rate is not as significant as a high recall rate. When evaluating false negatives, the severity of a wrong classification mostly depends on the classification of subsequent intervals. As mentioned all false negative intervals which directly follow a true positive will have virtually no impact.

So far, the described evaluation methods for accuracy, precision and recall use typing as a binary action and do not take into account, how many characters have been typed during a specific interval. To measure the accuracy of the typing function, we will further compare the outcome of the function in specific intervals with the total number of characters which have been altered in that time span. It is expected that the value of the proposed method will be higher, the more characters have been added during that time.

For this evaluation, we will use time intervals in which the user performs copy and paste actions. A distinctive feature of our input data is, that at the beginning of the implementation phase, the user is performing at least one copy and paste action. We restrict the analysis of larger textual changes to the intervals, in which this action occurs. For evaluation, we will compare the typing rates of those intervals to the typing rates of all positively identified intervals.

The number of characters which are added, will be identified manually and compared to the output of the proposed method. Using the correlation coefficient, we can estimate the accuracy of the typing function.

In addition to these evaluations, an exemplary study will be done on the behavior

of the proposed method on unknown IDEs and UI settings. The data for this study consists of all screencasts with night mode settings as they have not yet been included due to their small user sample sizes. The goal of this study is to give an idea on how the program will perform under unknown settings and lay a foundation for further research and development.

## 2. Related Work

Since programming screencasts, such as online tutorials, have become a popular resource for developers, scientific research has been conducted in order to extract semantically relevant content of a recording. Within this section, related papers are grouped according to their research question and therefore the content, they wish to extract.

### 2.1. Transcribing Code

A common research question for analyzing programming screencasts is the extraction of written code from image data. Although our method should not result in the code itself, it is possible to apply similar methods for obtaining the change rates of text.

#### 2.1.1. CodeTube

One paper from 2016, which proposes a method for extracting code from screencasts, has the descriptive title "Too long; didn't watch! Extracting Relevant Fragments from Software Development Video Tutorials" [18]. As the title suggests, the authors introduce a method for obtaining semantically relevant sections of a screencast. Similarly to our use case, they define relevant sections as frames, in which source code is visible within an IDE. The paper then proposes a method for obtaining a transcript for a coherent segment of code. This method has been implemented in a web-application called **CodeTube**.

In order to determine, whether a frame contains code, **CodeTube** uses the geometric properties of text editors as well as key words of the respective programming language. The distinctive attributes for text editors, which are used for this detection, are that they can be described as rectangles with border-parallel lines. Each side has to have a minimum length relative to the height and width of the screen. In addition to those properties, an OCR tool is used to extract the text within each frame. The extracted text is then used as an indicator for semantically relevant frames. Only frames, which

contain key words, are further considered. In contrast to our use case, `CodeTube` then uses a combination of noisy text fragments to reconstruct the original text. We on the other hand are primarily interested in the textual differences between two frames.

In 2019, the authors of `CodeTube` published an extension to their original work [17]. In addition to the described OCR approach, this paper proposes a method to classify the coherent fragments into seven categories using machine learning. The categories are introduction to a tutorial topic, theoretical concepts, code implementation, working environment setup, execution of implemented code, dealing with errors and closing of a tutorial. This classification has then been used to further identify relevant sections.

### 2.1.2. ACE

A similar application, which also focuses on the reconstruction of dynamically written text, has been described by Shir Yadid and Eran Yahav in their paper "Extracting code from programming tutorial videos" [22]. The method has been implemented in a tool called `ACE`. Similar to `CodeTube`, this application firstly identifies regions of interest before extracting their contents using an OCR.

For this application, the region of interest within a frame is defined as the smallest detectable rectangle, which covers the majority of the code in the image. The detected rectangles must further be visible within multiple frames of the screencast.

### 2.1.3. psc2code

In 2020 Bao, et al. proposed their method for transcribing code form screencasts [3], [4]. Their approach deals specifically with the exclusion of noisy frames, which improves the results of the OCR. They specify noisy code regions as any frames, which either contain no code at all, or in which code is not clearly visible due to dialog windows and similar pop ups. Their approach classifies each frame as either relevant or noisy, using a Convolutional Neural Network.

For all frames with clearly visible code, the regions of code are extracted by applying computer vision techniques. Within those regions of interest, the text is then extracted using an OCR.

### 2.1.4. Codemotion

In contrast to the previously discussed tools, `Codemotion` [12] focuses specifically on the differences in code over time. Similarly to the previous discussed tools, `Codemotion` uses feature detection to determine possible text editors of development environments. Then, an OCR is applied to each region of each frame. First, the results are used to determine, if the extracted text might be from a programming language. Afterwards, the code is reconstructed using the extracted text fragments.

An addition to previously discussed tools, `Codemotion` will determine time codes, for which the extracted text varies significantly from the previous frame. They split the screencast into different edit intervals. For each interval, the code is reconstructed separately. This paper also mentions the challenge of larger textual changes due to fast scrolling. In their approach, such a change is considered to be a separate interval.

## 2.2. Locating Source Code

All methods discussed in Section 2.1 use code editors as their regions of interest. Those regions might also include white-space, or, depending on the implementation, even toolbars and other sections irrelevant for this analysis. In recent years, studies have been conducted, in order to further restrict the location of the source code in image and video data.

### 2.2.1. Identifying Source Code

As OCR results are extremely sensitive to noise, Ott, et al. propose a method to better restrict the area for which OCR is applied [16]. Their approach uses a convolutional neural network (CNN) to classify frames of a programming tutorial. In contrast to all previous discussed tools, they do not restrict their analysis to screencasts and therefore computer generated images. Their model distinguishes between frames containing typeset code, partially visible typeset code, handwritten code, and frames without visible code. In addition to accurately identifying frames containing code, their approach also predicts the location of the source code. In a second step, they group frames, according to similarities in the detected regions of interest by applying deep autoencoders.

## 2.2.2. Identifying Code Fragments

A similar approach has been published by Alahmadi, et al. in 2018. Like Ott, et al., they propose a deep convolution neural network to identify regions, which contain source code in videos. In contrast to the previously discussed work, they focus on typeset code. Within their paper "Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning" [1], they evaluate their approach using 4,000 frames from Java programming screencasts. Their results show that the approach can accurately predict the location of source code.

In a follow-up paper from 2020 [2], they extended their analysis. They increased their data set to include three different programming languages. Furthermore, they evaluated the accuracy of OCR-extracted text and compared their results to the results obtained by CodeTube (2.1.1). Through their approach, the code of programming screencasts could be extracted with a significantly higher accuracy. Also, they directly compare their results with the approach from Ott et al. and concluded that their method outperforms the previous work.

## 2.3. Event Detection

So far, all discussed methods had either the goal to transcribe code from a programming screencast, or to increase the accuracy of the code extraction. But even though similar methods can be applied in order to answer our research question, we are less interested in the code itself, but in the typing process, which leads to the code. Our research shows, that questions related to events in programming screencasts, are less researched than questions related to transcribing the code.

### 2.3.1. Classifying Actions

In 2019 Zhao, et al. published the first paper, which describes a method to detect the actions performed within programming screencasts [23]. In contrast to the previously discussed approaches, their method processes not the individual frames, but the difference between subsequent frames, which gives an indication of the performed action. Those differences are used to identify a region of interest, which is then further processed.

For each pair of subsequent frames, a convolutional neural network is applied in order

to classify the actions, which happen between those frames. Within this work, Zhao, et al. distinguish between four groups of actions, each group is further divided into two to four subgroups. The main groups defined as follows: control cursor, edit content, interact with app, and other. Within their evaluation group, they achieved to classify the correct action with an average recall rate of 79% and 80% accuracy.

## 2.4. Significance of This Work

Most of the discussed papers, define the relevance of a frame with the visibility of code. We, on the other hand, are concerned with the amount of altered text as a measure of relevance.

The first step for all discussed papers is to define a region of interest, which in most cases consists of the region, which contains code. This aligns with our first part of the research question. Within all methods, which were presented in Section 2.1, this has been achieved by detecting characteristic features. The main objective of Section 2.2 was, that for obtaining reliable OCR results, you need to further restrict the area. They achieved that goal, by applying deep learning methods. In contrast, the approach described in Section 2.3 did not use the frames themselves as an indication for regions of interest. As they were interested in detecting changes, they extracted the regions with the largest change between two subsequent frames.

For this work, we are not focusing on the written code itself, but on the amount of textual changes. Transcribing the code will therefore not be necessary. As we are therefore not applying an OCR, we will reconstruct the region of interest by detecting characteristic features.

The second part of our research question concerns the quantification of written code. Here, the existing literature has fewer examples, of how this could be achieved. Most notably, the work by Zhao, et al. categorizes changes in frames, including categories for adding and deleting characters using a convolutional neural network. Their method could also be applied to our research question. However, a significant drawback to their method is the required manual labor for establishing a ground truth. Within their study, they analyzed 73,725 pairs of frames, for which a change was detectable. According to their work, three people worked on the labeling process for one month [23].

As mentioned in section 1.3.1, we are working with 109 screencasts. On average, each

screencast consists of 4,100 frames. Even when excluding pairs of subsequent frames, in which no change is detectable, it will require immense manual effort, in order to adapt a similar method to Zhao, et al. So, the selected approach for this work will not apply a convolutional neural network. Instead, we are extracting distinct features of code editors within IDEs, in order to correctly quantify typing activities.



## 3. Algorithmic Approach

### 3.1. Initial Definitions

For further description of the proposed method, we need a more formal definition of a video. The notations for image processing within this chapter are based on the notation used in the book „Digital image processing” [9].

We will describe a video  $\mathcal{V}$  as a consecutive sequence of images  $\mathcal{V} = \{\mathcal{I}_1, \dots, \mathcal{I}_n\}$ . Each image  $\mathcal{I}_k$   $k \in \{1, \dots, n\}$  can be defined as a two-dimensional function  $\mathcal{I}_k(i, j) \rightarrow p_{i,j}$ , where  $p_{i,j}$  denotes the gray level of the image at the pixel  $(i, j)$ . The values of  $p_{i,j}$  can range from 0 (black) to 255 (white).

In image processing, a spatial coordinate system is used, whereas  $i$  denotes the index of a column and  $j$  denotes the index of a row. The pixel value  $p_{1,1}$  does therefore always refer to the gray level at the left upper corner. We will use the notation  $x$  to specify the total number of columns and  $y$  for the total number of rows respectively. The function  $\mathcal{I}_k(i, j)$  is therefore defined over the intervals  $i \in \{1 \dots x\}$  and  $j \in \{1 \dots y\}$  respectively.

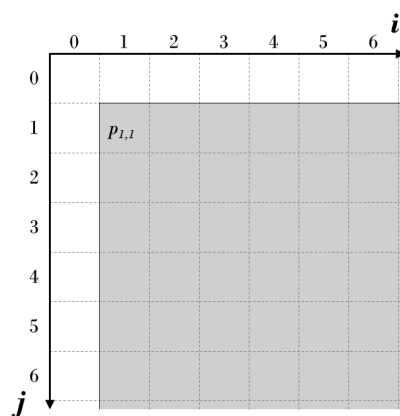


Figure 3.1.: Visualization of the coordinate system.

For a specific image  $\mathcal{I} \in \mathcal{V}$ , the values  $(x, y)$  are known as the resolution of the image.

Notably, for one screencast, the resolution of all images  $\mathcal{I}_k$  stay constant for all frames within the video.

## 3.2. Frame Layout Detection

As specified in section 1.2, we want to detect textual changes over the duration of the recording. We defined, that semantically relevant text changes are those, which happen inside text editors within IDEs. Regardless of the used IDE or the UI preferences, all editors have the following properties:

1. The editor region has a rectangular shape, with edges parallel to the borders of the screen.
2. While typing, the IDE is located at the foreground of the screen. So, the edges are not covered by similarly large rectangles.

Using those properties of text editors, we will extract all rectangles within a frame, which fulfill those criteria. Notably, those properties are also true for most other computer windows. The outcome of this method is therefore a set of possible regions of interest (ROIs), which need to be analyzed further.

As a preprocessing step, we will evaluate the variance of the average pixel value per row over all frames. If the variance for rows at the bottom area of the screen lies below a certain threshold, we will exclude this area for subsequent processing.

The first step to obtaining the set of possible ROIs for an individual frame, is to extract all rectangles, which fulfill the specified criteria. This method is described in sections 3.2.1, 3.2.2 and 3.2.3. The resulting set of possible ROIs might include rectangles, which can be discarded due to their geometric position to other detected ROIs. By analyzing all pairs of detected rectangles, we can reduce the set of possible ROIs. This process is described in section 3.2.4. After identifying the frame layout for an individual frame, we will group consecutive frames with similar layouts. This process is described in section 3.2.5.

Table 3.1 gives an overview of the performed operations. Further details can be found in the respective chapters. Furthermore, the table shows a visual interpretation of the outcome of each step. The following image has been used as input for this example:

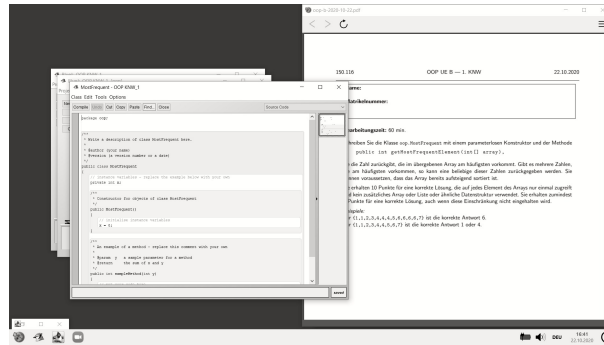


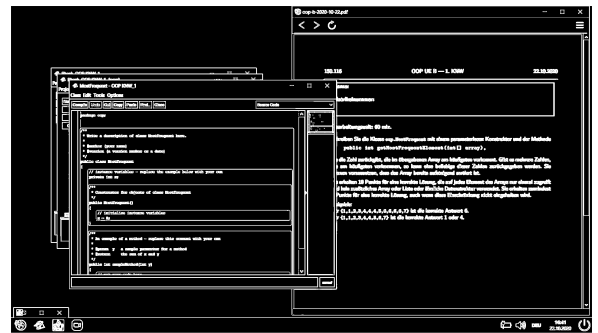
Figure 3.2.: Input image for ROI detection.

Section

Visualized Output

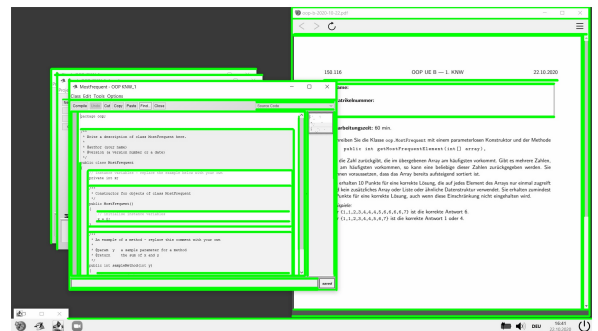
Feature Extraction(3.2.1)

- **Scharr operators:** Those operators are applied to the image in order to emphasize edges.
- **Binarization:** By binarizing an image, relevant edges will be highlighted.



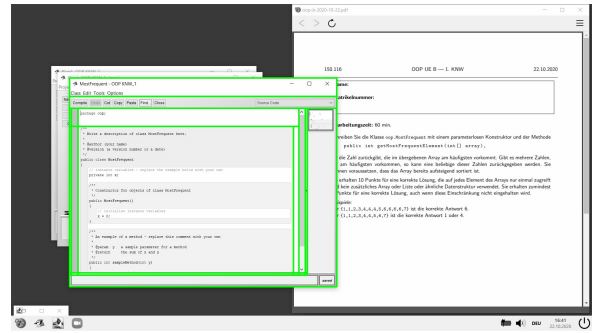
Line Detection (3.2.2)

- **Probabilistic Hough Transform:** For an image of highlighted regions, this method is applied to extract all lines, which fulfill predefined criteria.



### Construction of ROIs (3.2.3)

- **Non-maximum suppression for lines:** For lines, which lie in close proximity to each other, only the longest line will be used for further processing.
- **Hough Transform for rectangles:** Using the set of line, this method constructs possible rectangles. All rectangles, which fulfill predefined criteria are extracted for further processing.



### Geometric Analysis (3.2.4)

- **Analysis of overlap:** By analyzing the overlap between two rectangles, we can reduce the set of possible ROIs.
- **Analysis of structure:** All remaining ROIs are separated into topmost rectangles (red) and their nested ROIs within (green).

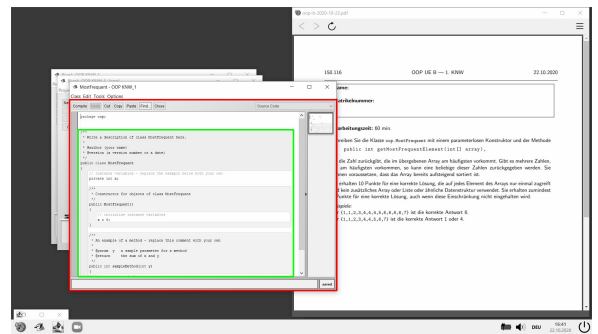


Table 3.1.: Overview of methods used for ROI detection.

### 3.2.1. Emphasize Relevant Features

In computer vision, edges are detectable due to the high contrast in comparison with their surroundings. In pixel values, this corresponds to a relatively large change in the gray level between subsequent pixels. This change can be measured by calculating the derivative of the image at each pixel  $p_{i,j}$ . A high derivative corresponds to a high change in the intensity level and might therefore indicate an edge.

For further description of the derivative in image processing, we will first look at a one-dimensional signal. For each point  $p_i$  of that signal, the derivative can be estimated by calculating the difference of the surrounding points  $p'_i = p_{i+1} - p_{i-1}$ . A high value of  $|p'_i|$  means that the signal has a high intensity change around  $p_i$ . Notably, a high value  $|p'_i|$  correlates to a high derivative at  $p_i$ .

The following images are visualizations of one-dimensional signals and the calculated differences at each point. The first example shows a clear border between the high and low intensity regions. The second example displays a blurrier transition.

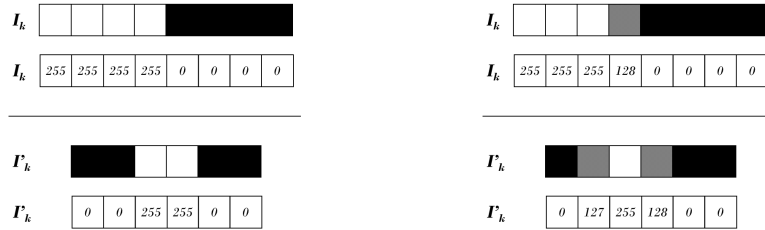


Figure 3.3.: Examples of one-dimensional edge detection.

When working with two-dimensional images, this calculation is performed separately for the horizontal and vertical directions. In contrast to the one-dimensional signal processing, we will consider a two-dimensional neighborhood for this calculation. The size of these surroundings can be chosen according to the input image. A larger neighborhood means that more values are taken into account when calculating the differences. Thus, values with a high contrast will influence the derivative within a larger area. The resulting image of the calculation will therefore look blurry around high contrasting regions.

The input images for this work are computer generated images. We can therefore assume that the edges in our input images are well defined and will easily be detectable. Consequently for our calculation, we will use a  $3 \times 3$  surrounding, with the pixel  $p_{i,j}$  at its center.

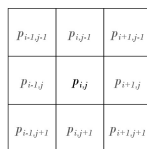


Figure 3.4.: Pixels considered for derivative calculation.

Within this region, the derivative is estimated by applying the discrete form of the two-dimensional convolution between an image  $\mathcal{I}$  and a convolution kernel  $h$ . We will denote the convolution operation as  $\mathcal{I} * h$ . The convolution kernel defines, how the surrounding pixel values of  $p_{i,j}$  are taken into account in the calculation. As we

want to use a  $3 \times 3$  surrounding, the kernel will have the same dimension. For a  $3 \times 3$  convolution kernel, the formula for the two-dimensional discrete convolution is as follows.

$$I * h := p'_{i,j} = \sum_{l=-1}^1 \sum_{m=-1}^1 h_{l,m} \cdot p_{i-l,j-m}$$

$$i \in \{1, \dots, x\}, j \in \{1, \dots, y\}$$

As becomes clear from this definition, the values of the convolution kernel are the coefficients with which the surrounding pixel values are taken into account. For the one-dimensional signal processing, the convolution kernel could be described as a  $3 \times 1$  kernel with the following values.

$$h_{1D} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

For images, we calculate the directional derivatives. Therefore, we need two separate kernels  $h_x$  and  $h_y$  estimate the derivative in the respective direction. Commonly used kernels are the Prewitt filters or the Sobel operators. With the Prewitt filter, all pixel values in the surrounding of  $p_{i,j}$  are equally considered for the derivative calculation. By applying the Sobel operator, pixels closer to  $p_{i,j}$  have a higher influence. For our use case, we will use a optimized form of the Sobel filters, the Scharr operators [20].

$$h_x = \begin{bmatrix} 47 & 0 & -47 \\ 162 & 0 & -162 \\ 47 & 0 & -47 \end{bmatrix} \quad h_y = \begin{bmatrix} 47 & 162 & 47 \\ 0 & 0 & 0 \\ -47 & -162 & -47 \end{bmatrix}$$

For commonly used computer vision libraries like OpenCV, the Scharr operators are the recommended kernels to calculate the derivatives within a  $3 \times 3$  neighborhood [15].

The figures below show the results of  $\mathcal{I} * h_y$  and  $\mathcal{I} * h_x$  for two different input images. In order to display the outcome of this operation, we have taken the absolute values for each pixel  $p_{i,j}$  and scaled them between 0 and 255. The images are from two different IDEs. Notably, the horizontal and vertical borders of the text editors are clearly visible.

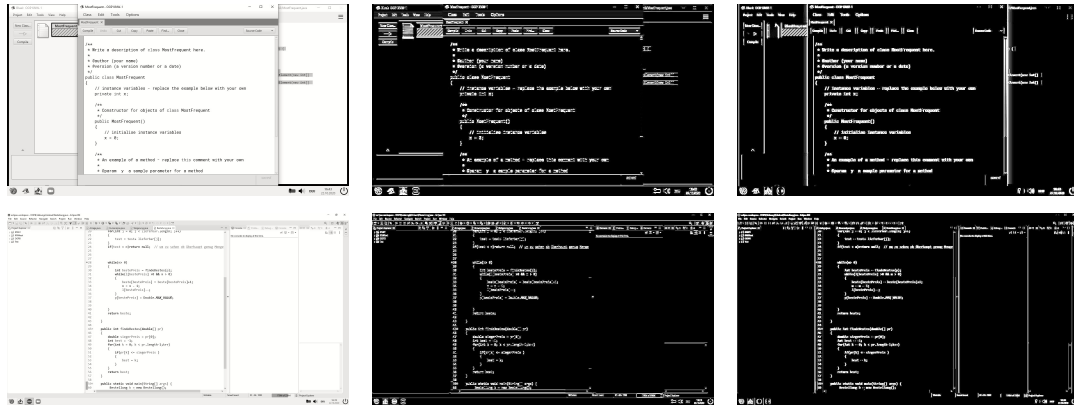


Figure 3.5.: Input images; Estimated vertical derivatives; Estimated horizontal derivatives.

For further processing, the images are then converted into binary images. For our input images, the operation results in a high contrast between black and white regions. We can therefore apply an absolute threshold  $th_{bin}$  to all pixels. If a pixel  $p_{i,j}$  is below this threshold, it is considered to be black. Otherwise, we consider it to be white.

$$p_{i,j} = \begin{cases} 0 & p_{i,j} < th_{bin} \\ 255 & p_{i,j} \geq th_{bin} \end{cases} \quad i \in \{1 \dots x\} \text{ and } j \in \{1 \dots y\}$$

### 3.2.2. Extract Border Parallel Lines

A commonly used method for detecting lines in images, is known as the Hough Transform [19]. This method determines, how many pixels within an image support a certain line. To represent all lines within an image, we need to use the normal representation of a line, where  $\rho$  is the orthogonal distance of the line to the origin and  $\theta$  represents the distance from the  $i$  axis:

$$\cos(\theta) * i + \sin(\theta) * j = \rho$$

Given a binary image, each white pixel  $p_{i,j} = 255$  supports all lines, for which the following statement is true:

$$\cos(\theta) * i + \sin(\theta) * j - \rho = 0$$

To identify all lines, we transform the image space into the Hough space, where the axis are  $\rho$  and  $\theta$ . For each white pixel, the  $\rho$  and  $\theta$  values for all possible lines, which

go through this point, are then represented within this space.

The following figure shows an example image, for which we will perform a Hough Transform.

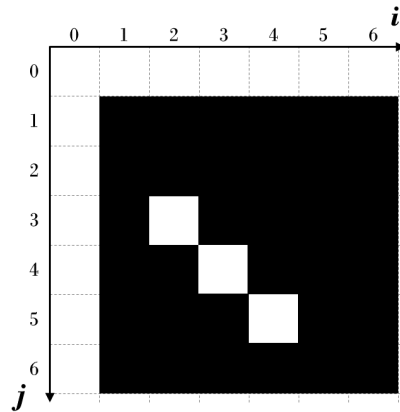


Figure 3.6.: Example image for Hough Transform.

For each white pixel, the  $\rho$  and  $\theta$  values of all possible lines are represented as curves within this space. The interception points of those curves mean, the respective  $\rho$  and  $\theta$  values result in lines which are supported by all three white pixel.

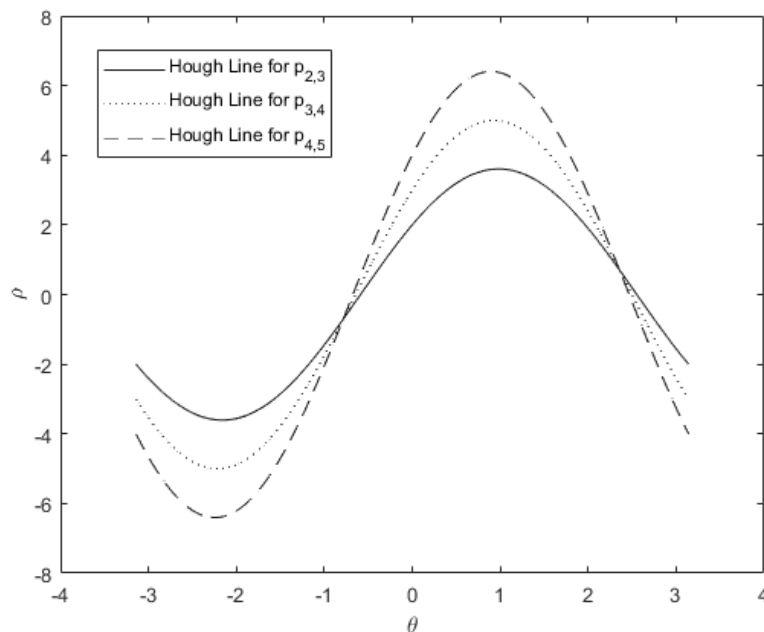


Figure 3.7.: Representation of Hough space for the example image.



In general, we are interested in points within the Hough space, where the number of supporting pixel lie above a certain threshold.

The standard Hough Transform counts the white pixels for all possible values  $-\frac{\pi}{2} \leq \theta < \frac{\pi}{2}$  and  $1 \leq \rho \leq \rho_{max}$ . The value  $\rho_{max}$  denotes the outermost pixel of an image for a given  $\theta$ -value. We can then analyze the result for maxima. For our use case, this process can be optimized by reducing the space of possible values. As we are only interested in border parallel lines, we can define the space of possible values for  $\theta$  and  $\rho$  as follows:

$$\begin{aligned}\theta &\in \{-\frac{\pi}{2}, 0\} \\ 1 &\leq \rho \leq \rho_{max}\end{aligned}$$

An optimization of the standard Hough Transform is known as the Progressive Probabilistic Hough Transform [14]. The implementation of this method does not include all pixels of an image but reduces the number to a random subset. Additionally, this method returns the extremes of a detected line, which represent the start and end point.

Additional inputs for this method are a minimum line length and a maximal gap within a line. For this use case, the minimum line length must include the minimum side length of the text editor.

### 3.2.3. Construct Possible ROIs

For all extracted lines, we need to evaluate, which ones could be the borders of a text editor within an IDE. When looking at the set of all possible lines, we will observe, that some regions contain clusters of detected lines in close proximity to each other. The figure below shows a close up of a corner region of a window. Due to the content of the window as well as some graphical elements, we extract not only the borders but also other lines.

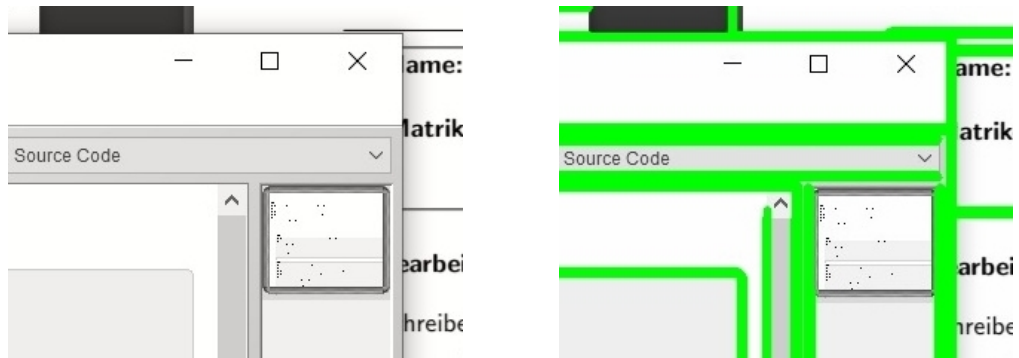


Figure 3.8.: Right: corner region; Left: all detected lines.

To reduce the number of possible lines and to better extract the border of a region of interest, we will apply a non-maximum suppression algorithm. For each set of horizontal and vertical lines, we will firstly identify clusters by analyzing their geometric distance. For each cluster, only the longest line will remain in the set of all possible lines.

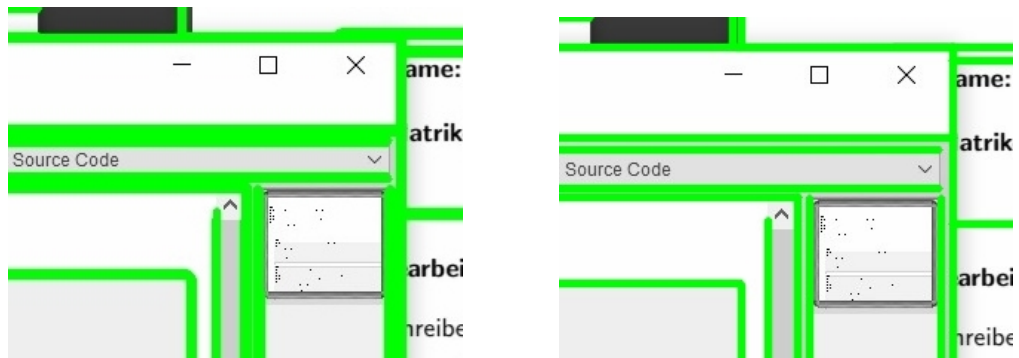


Figure 3.9.: Right: all detected lines; Left: reduced set of lines.

Given the reduced set of horizontal and vertical lines, we will determine a set of all possible ROIs. We will consider all rectangles, which fulfill the following criteria to be relevant:

1. The length of all sides must be above a given threshold.
2. Each side must be supported by the detected lines.

For the first criterion, we will use the same minimum length, which was also applied to the Hough Transform in Section 3.2.2. For all possible rectangles with this minimum side length we will then analyze, whether the rectangle is also supported by the extracted border parallel lines.

We define a minimum ratio of supported pixels to the total number of pixels for a specific side. In case, that all four sides of a rectangle with minimum side length fulfill that criterion, it is considered to be a possible region of interest.

### 3.2.4. Analyze Geometric Properties

Given the set of all detected ROIs, we need to identify those, which are located in the foreground. We can observe from the visually represented output of method 3.2.3, that the detected rectangles might be overlapping and therefore not fulfilling this criterion. In order to reduce the set to the topmost ROIs, we will look at the geometric relationships between all detected rectangles.

Firstly, we will look at overlapping rectangles. Given two overlapping rectangles, at least one of them cannot be a topmost ROI and should therefore be excluded from the resulting set. We will say, that two rectangles  $roi_i, roi_j$  are overlapping, if they share some pixels. We can measure this overlap  $o_{i,j}$  of  $roi_j$  in  $roi_i$  by calculating the ratio between shared pixels to total pixels of  $roi_i$ .

To analyze pairs of overlapping rectangles, we need to distinguish between partially and fully overlapping. Two rectangles  $roi_i, roi_j$  are said to be partially overlapping, if  $1 > o_{i,j} > 0$  and  $1 > o_{j,i} > 0$ . In contrast, we will say that  $roi_j$  is fully overlapping with  $roi_i$ , if  $1 > o_{i,j} > 0$  and  $o_{j,i} = 1$ . The following figure gives examples of partially and fully overlapping rectangles as well as their corresponding overlaps  $o_{i,j}$  and  $o_{j,i}$ .

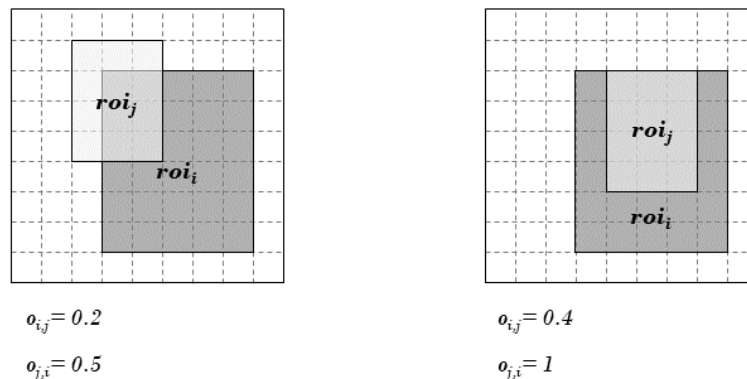


Figure 3.10.: Example of partial and full overlap.

For our proposed method, we will first look at all pairs of partially overlapping rectangles. We know by definition, that at least one of them does not belong to the topmost

layout of ROIs. So, we need to decide for each pair, which rectangle should be discarded.

In Section 3.2.3 we introduced a method to indicate how well a given rectangle is represented in the image of relevant lines. The method results in a value between 0 and 1, whereas a higher value represents a higher support of the rectangle in the image. We determined the set of possible ROIs by applying a threshold to the resulting values. For a pair of partially overlapping rectangles, we therefore know, that for both rectangles, the method results in a value above the threshold. But furthermore, this measure gives us a relative value of how good the respective rectangle can be observed in the picture of possible lines. We can therefore conclude that the rectangle with the lower value should be excluded from the set of possible ROIs. In the rare case, that both rectangles result in the same value, we will discard the smaller one. We will use this decision mechanism for all pairs of partially overlapping rectangles.

This method is represented in figure 3.11. The input image shows two overlapping rectangles. Because the overlap occurs only in a relatively small area, both rectangles are considered to be possible ROIs. An analysis of the actual representation of the rectangles in the input image shows, that only the lower rectangle can be in the foreground.

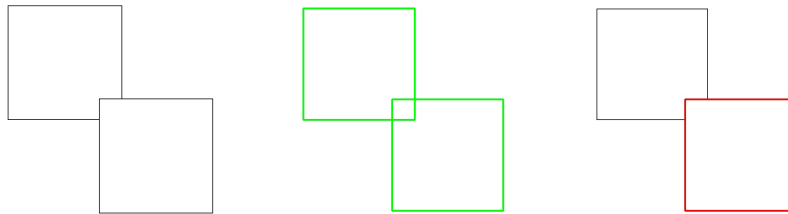


Figure 3.11.: Input image; Green: all detected ROIs; Red: resulting ROIs

After reducing the set of possible ROIs to a set with no pair of partially overlapping rectangles, we will now analyze pairs of fully overlapping ROIs. For partially overlapping ROIs, we could use the visual representation of the rectangles in the image to decide, which one is more likely to be part of the topmost layout. For fully overlapping rectangles, this method can not be applied, as the borders of both rectangles can be fully visible.

So, instead of the visibility of the borders, we will analyze the overlap  $o_{i,j}$  of a smaller ROI  $roi_j$  in  $roi_i$ . As  $roi_j$  lies fully inside  $roi_i$ , the overlap gives us a measure, of how similar those rectangles are. In case this overlap is above a certain threshold, we will

remove the larger ROI  $roi_i$  from the set of possible ROIs, as we do not expect that the image difference holds any semantically relevant information.

Figure 3.12 shows the remaining set of possible ROIs for two different IDEs. Ideally, we want to extract the text editor within the IDE. In both cases the outline of the editor has been detected but is fully overlapping with larger rectangles. In the second picture, the method also detects a smaller rectangle within the text editor.

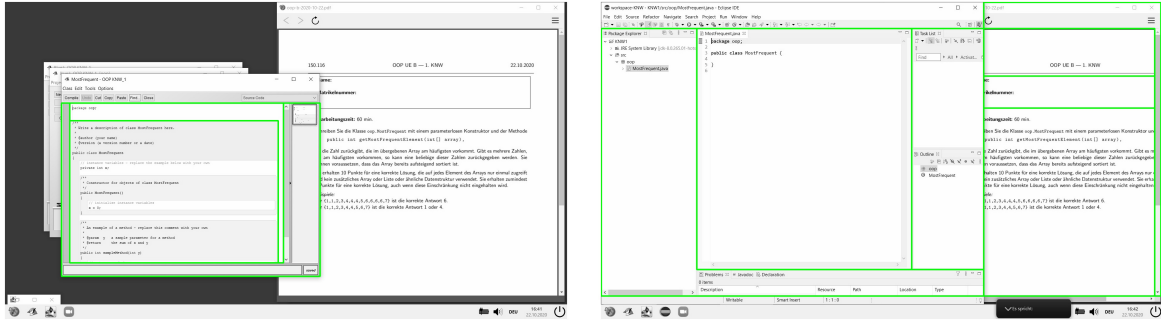


Figure 3.12.: Remaining ROIs for different IDEs.

As becomes obvious, we are not able to decide, which rectangle will be our actual region of interest by just analyzing their geometric position to each other. Instead, we will construct and examine the hierarchy of all remaining ROIs.

We will say, that a ROI  $roi_i$  belongs to the topmost layer, if the overlap of  $roi_i$  to all other ROIs  $roi_j$  in the set of all remaining ROIs is  $o_{i,j} < 1$ . Therefore, all other rectangles are either smaller fully overlapping rectangles or not overlapping with  $roi_i$ .

For further processing, all ROIs, which belong to the topmost layer, are grouped into a separate set of topmost ROIs. For each ROI, we define a set of directly nested ROIs. A ROI  $roi_j$  is said to be a directly nested ROI of  $roi_i$ , if  $roi_i$  it is the smallest ROI with which  $roi_j$  is fully overlapping. As ROIs, which are surrounded by multiple ROIs, are less likely to be relevant, we define a maximum depth and discard all possible ROIs where the nested structure exceeds this depth.

The following figure shows the result of this method applied to screenshots of different IDEs. The left images show all detected possible ROIs. On the right are images of the visual representation of all remaining ROIs. The topmost ROIs are marked in red.

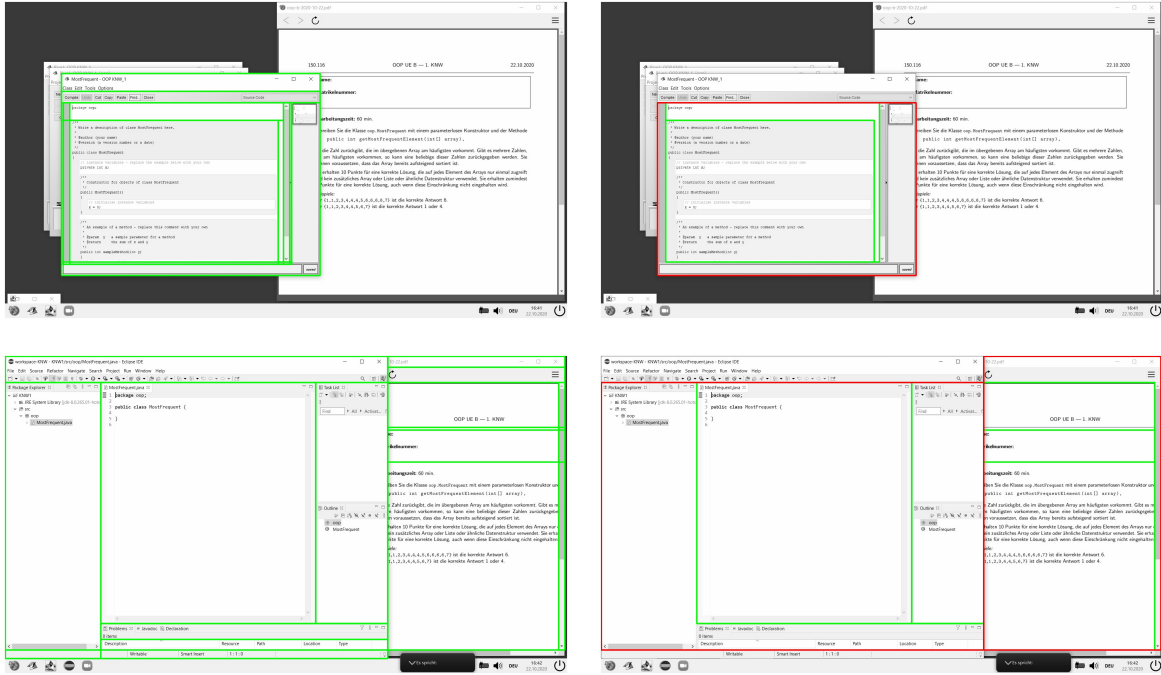


Figure 3.13.: All detected possible ROIs; Remaining possible ROIs with hierarchy.

### 3.2.5. Group Frames According to Frame Layout

So far, the process results in the window layout of a single frame. Within this section, we will describe a method for grouping frames with a similar window layout. As we established in Section 3.2.4, the nested regions of interest might be part of the user interface and can therefore change their position inside the topmost ROIs. When we are comparing the detected layout of subsequent frames, we will therefore only compare the topmost ROIs.

When provided with a frame  $\mathcal{I}_k$  and a corresponding layout of topmost ROIs, we can determine, whether a subsequent frame  $\mathcal{I}_{k+m}$  contains the same topmost ROIs, by evaluating the emphasized lines of  $\mathcal{I}_{k+m}$  (as defined in Section 3.2.1). Each border of the detected topmost ROIs for  $\mathcal{I}_k$  must be found as an emphasized line of  $\mathcal{I}_{k+m}$ . For detection, we can use the same minimum ratio of white pixel to side length of Section 3.2.3 to evaluate whether  $\mathcal{I}_{k+m}$  contains a specific topmost ROI.

Notably, we will only need to calculate the layout once for  $\mathcal{I}_k$ . For all subsequent images, we only need to apply the first step in the process described in 3.1. Via this method, we know that all detected ROIs of  $\mathcal{I}_k$  are present within  $\mathcal{I}_{k+m}$ . But we can not determine whether all topmost ROIs of  $\mathcal{I}_{k+m}$  are present in  $\mathcal{I}_k$ . All new detectable ROIs in  $\mathcal{I}_{k+m}$ , which do not cover any borders of topmost ROIs in  $\mathcal{I}_k$  will not be de-

tected.

To avoid a loss of information, we will compute the layout of detected topmost ROIs at fixed intervals and compare them to the original layout of  $\mathcal{I}_k$ . When provided with the layouts of detected topmost ROIs for two frames  $\mathcal{I}_k, \mathcal{I}_{k+m}$ , we will use the overlap as a measure for similarity. For each topmost ROI of  $\mathcal{I}_k$ , there must be a topmost ROI of  $\mathcal{I}_{k+m}$ , so that the overlap of those two ROIs is above a given threshold. Similarly, each topmost ROI of  $\mathcal{I}_{k+m}$  must fulfill the same criterion with a topmost ROI of  $\mathcal{I}_k$ .

As soon as an image  $\mathcal{I}_{k+m}$  does not contain the topmost ROIs of  $\mathcal{I}_k$  or the topmost ROIs fail to overlap sufficiently, we define a new section from  $k$  to  $k + m - 1$ .

### 3.3. Grid Analysis

We currently have a division of the screencast into sections with a similar layout. The topmost ROIs of all frames within this section match, but we still need to consider the nested ROIs, as they might be a better indicator for the location of the actual text editor.

We can assume, that the further analysis of textual changes works more accurately, if the region of the text editor has been identified more narrowly, as changes in the irrelevant parts of the ROI might falsely be identified as typing activities. On the other hand, we need to be aware, that ROIs could also be detected, due to highlighting and markings within a text editor.

The following figure shows the topmost ROI, which contains a text editor, for two different IDEs with the respective nested windows. The first picture shows a case, in which the smallest possible ROI excludes part of the code. In contrast, the topmost ROI of the second picture includes various other regions of the IDE in addition to the text editor.

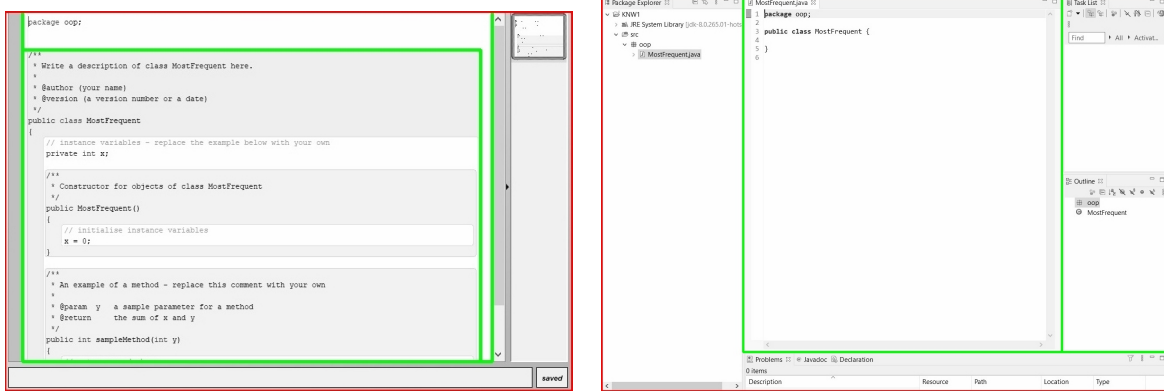


Figure 3.14.: Relevant ROI for different IDEs.

In order to decide, which of the possible ROIs gives us the best solution, we will examine the content within. As we are interested in textual changes, we will use properties of the relevant text to decide, in which ROI those properties are represented best.

One distinct property of text editors within IDEs is, that all commonly used font types are monospaced fonts. All characters for such fonts can occupy the same width for a given height. Furthermore, the font size remains constant within the text. For code editors, it must therefore be possible to define a regular grid, which separates all characters.

Section 3.3.1 describes the process of obtaining the relevant properties. Using those properties we will decide, which ROIs should be analyzed further. For this process, we are interested in one ROI for each topmost ROI that is most likely to be the code editor. Section 3.3.2 proposes a method to decide for a single frame, which ROIs fulfill the specified criteria best. This information will be used in Section 3.3.3 to determine the ROIs for the respective section of the screencast.

Throughout this section, we will only consider ROIs instead of entire frames. Will will therefore use the term image synonymous with ROI. To better describe the proposed method, we will use the indices  $i_{roi}$  and  $j_{roi}$  as pixel coordinates for a specific ROI, with an origin in the left upper corner.

### 3.3.1. Row and Column Period

For a given ROI, we now want to decide, whether we can identify a periodic pattern for the rows and columns. The performed steps for this approach are described in table



3.2. In addition, the table also includes a visual representation of the output. The following figure has been used as input for this example.

```
package oop;

/**
 * Write a description of class MostFrequent here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MostFrequent
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class MostFrequent
     */
    public MostFrequent ()
    {
        // initialise instance variables
    }
}
```

Figure 3.15.: Input image for row and column detection.

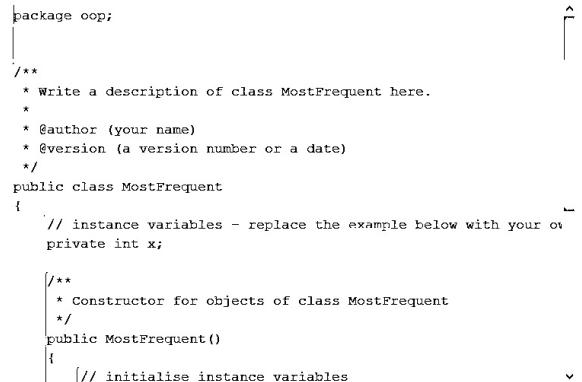
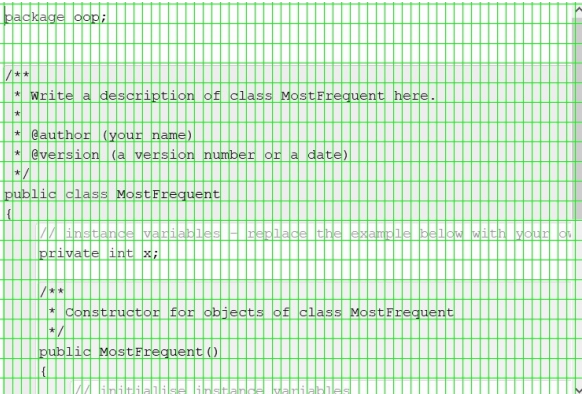
Step	Visualized Output
<p><b>Preprocessing</b></p> <ul style="list-style-type: none"> <li>• <b>Binarization:</b> Using an adaptive threshold, the background can be separated from the text.</li> <li>• <b>Long line exclusion:</b> By applying a probabilistic Hough transform, we can exclude irrelevant (long) lines.</li> </ul>	
<p><b>Period Detection</b></p> <ul style="list-style-type: none"> <li>• <b>Reduction:</b> The two-dimensional image will be reduced to the horizontal and vertical one-dimensional signals.</li> <li>• <b>Autocorrelation:</b> Using those signals, we will determine the most likely values for the period in horizontal and vertical directions.</li> </ul>	

Table 3.2.: Overview of methods used for period detection.

The first step for this approach is to simplify the input in such a way, that characters are emphasized. We therefore want to convert the grayscale image into a binary image, in which characters and background are clearly distinguishable.

A challenge for binarizing the input images is the syntax highlighting within IDEs. As can be observed in figure 3.15 the pixel values of character and background colors vary significantly. In order to establish a robust method for emphasizing characters, we will apply an adaptive threshold to the input images. For such thresholds, we consider a defined neighborhood of a pixel. The gray levels of the pixels within this neighborhood are used to calculate a threshold. The threshold for  $p_{i,j}$  is then defined as the mean of all pixel values in its surrounding minus a constant value. If  $p_{i,j}$  is smaller than the threshold, we will set its value to 0, otherwise to 255 [7]. The resulting image for our input image can be seen below.

```
package oop;

/**
 * Write a description of class MostFrequent here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MostFrequent
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class MostFrequent
     */
    public MostFrequent()
    {
        // initialise instance variables
    }
}
```

Figure 3.16.: Example image after applying an adaptive threshold.

Notably, in addition to characters, this method also emphasizes straight lines, which either belong to UI elements like scroll bars, or to syntax highlighting. As straight, consistent lines over a certain length are never part of characters, we filter them out. The method described in Section 3.2.2 can also be used to identify lines within our current binarized images. The only difference to the predefined method is, that we now want to identify black lines on a white background. We will therefore not perform the Hough Transform on the original image, but on an inverted copy. For all identified lines, the pixel values within the original binarized image will then be set to 255. an example of the graphical representation of this method can be seen in table 3.2.

For the resulting images, we can now analyze, whether a periodic pattern in horizontal and vertical direction can be detected. First of all, we will reduce the complexity of the two dimensional image into two one-dimensional signals. We obtain the vertical signal by calculating the average pixel value per row. Likewise, the horizontal signal is obtained by calculating the average pixel value per row for each column.

The average pixel values for all rows and columns of the example image are displayed below. Especially for the average pixel values per row, we can observe, that the data has visible notches. Those represent rows, in which many black pixels exist. This indicates that those rows might contain text.

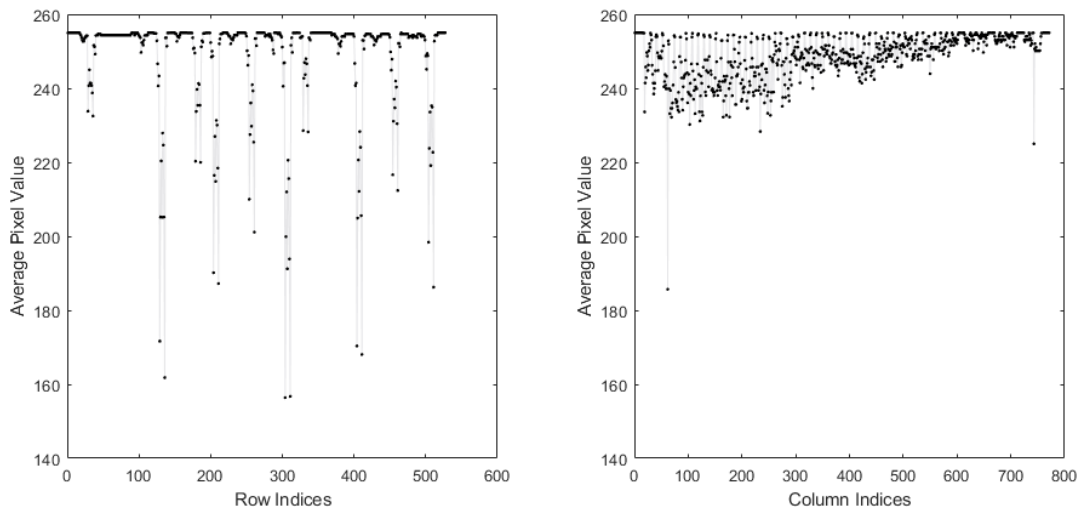


Figure 3.17.: Average pixel values per row (left) and column (right).

We now want to analyze, whether we can identify an integer periodicity within those signals. A method to determine repetitive patterns for data containing noise is autocorrelation [5]. For a signal  $x_i, i \in \{1, \dots, N\}$ , the autocorrelation for a specific period  $a(p)$  is calculated as follows:

$$a(p) = \frac{\sum_{i=1}^{N-p} (x_i - \bar{x}) * (x_{i+p} - \bar{x})}{\sum_{i=1}^N (x_i - \bar{x})^2}$$

For the horizontal and vertical signals, we will calculate the autocorrelations  $a_{hor}(p)$  and  $a_{ver}(p)$  for all values  $p \in \{1, \dots, N\}$ . As we want to identify a character grid, the only period values, in which we are interested, are possible widths of characters and heights of lines. So as the periodicity of the character grid we will only accept values

$a_{hor}(p_{hor})$  witch lie within  $p_{hor} \in \{width_{min}, \dots, width_{max}\}$ . Likewise, the acceptable periods for the horizontal grid are  $p_{ver} \in \{height_{min}, \dots, height_{max}\}$ .

As a high autocorrelation value indicates a possible period, we are particularly interested in local maxima within acceptable period values. As we are dealing with periodic values, it is important to keep in mind, that multiples of a maximum should also be a local maximum. In reverse, integer divisors of a local maximum might better represent the character grid. To consider those effects, we will define the strategy for identifying the period value  $p_{best}$  as follows:

1. Identify the highest local maximum  $p_{best}$  of  $a(p)$  within  $p \in \{min, \dots, 3 * max\}$ .  
If there is no local maximum within the specified range, no periodic value is found.
2. Identify all divisors of  $p_{best}$ , which are greater or equal than  $min$ .
3. Set  $p_{best}$  to the smallest divisor, which is also a local maximum.
4. If  $p_{best} > max$ , no periodic value is found.
5. If either  $2 * p_{best}$  or  $3 * p_{best}$  is a local maximum,  $p_{best}$  is an acceptable value.  
Otherwise, no periodic value is found.

If no period can be found, which fulfills the specified criteria, we will return the value  $-1$  as an indication, that no periodic value was detectable.

For the implementation of this method, we include a tolerance for the decision if a divisor is a local maximum (step 3). Furthermore, we found that the period values could be detected more reliably, if we exclude the outermost rows and columns for an autocorrelation calculations. Also, as the number of rows and columns stay constant, we will calculate a non-normalized version of the autocorrelation.

We experimentally established, that acceptable character widths lie within  $p_{hor} \in \{7, \dots 30\}$  and acceptable line heights within  $p_{ver} \in \{13, \dots 40\}$ . The figure below shows the calculated autocorrelations  $a_{ver}$  and  $a_{hor}$  of our example image for relevant

$p$  values. All local maxima are highlighted.

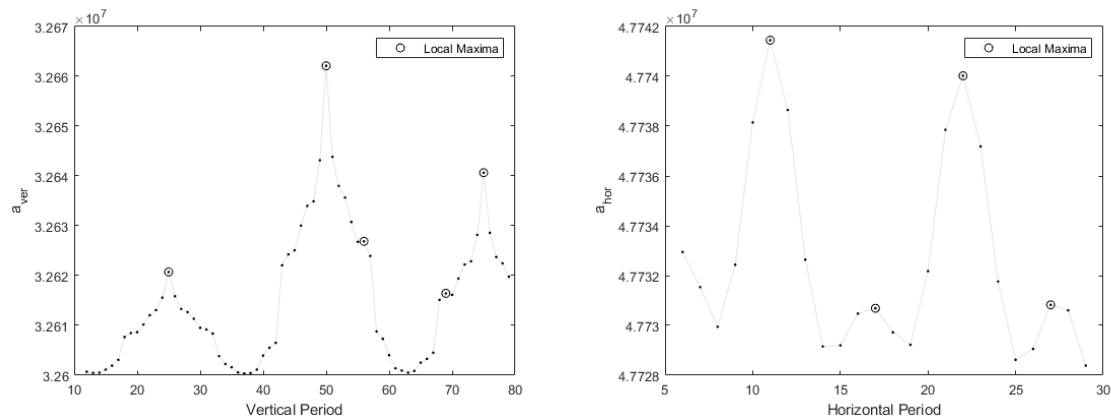


Figure 3.18.: Autocorrelation results per row (left) and column (right).

For the vertical period, the highest local maximum lies at 50. The only possible divisors for 50 is 25, which is also a local maximum. Therefore, the resulting value for the vertical period is 25. The highest local maximum for the horizontal period lies at 11. As this value has no acceptable divisors, we need to evaluate the autocorrelation for 22 and 33. At 22 we can also observe a peak, which means that 11 is an acceptable period. We can see the graphical interpretation of those values in the following figure.

```
package oop;

/**
 * Write a description of class MostFrequent here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MostFrequent
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class MostFrequent
     */
    public MostFrequent()
    {
        // initialise instance variables
    }
}
```

Figure 3.19.: Detected grid for example image.

### 3.3.2. Period per Frame

So far, we established a method, which results in a best fitting period value for a specific ROI, as long as such a value exists. As mentioned, the layout of all possible ROIs, might include regions with a nested structure. We will use the detected period values, in order to decide, which structure should be analyzed further.

We assume, that for a given nested ROI, the surrounding ROI will be a better bound for a text editor, when the detected horizontal and vertical periods are identical. In contrast, all nested ROIs are preferred to the surrounding ROI, in case one of the detected periods for the nested ROIs does not match the detected surrounding period values. Therefore, we will process the smallest ROIs, for which the period values do not match the surrounding values for all topmost ROIs.

The following figure compares the detected ROIs of an example image to the resulting ROIs of this process.

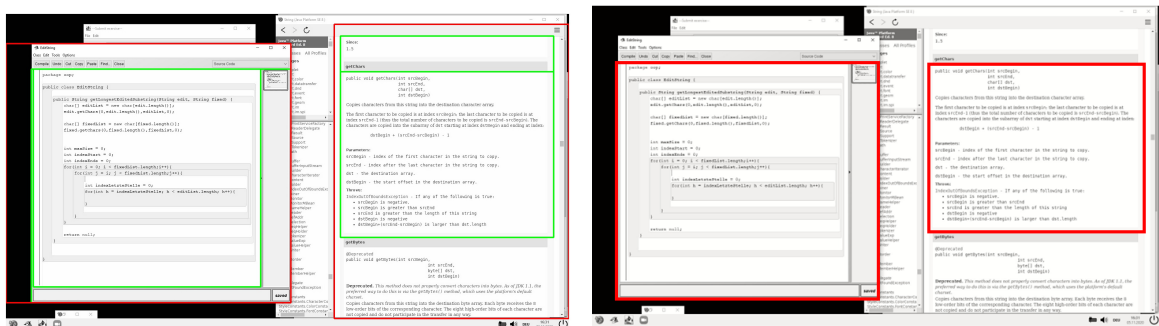


Figure 3.20.: Comparison of all detected ROIs (left) and the resulting ROIs of this method (right).

### 3.3.3. Period per Section

As some individual frames may contain noise, we analyze the period values within the ROI structure over the entire section. For each frame we obtain a ROI structure with corresponding period values. Using this information of each frame within the section, we determine the best ROI structure with corresponding period values for the entire section.

For one individual ROI we determine the number of frames, in which this ROI was part of the resulting ROI structure. If the respective ROI was part of at least one resulting ROI structure, we can determine the period values, which were most often detected

for this ROI. The number of frames in which those period values were detected, will be used for a voting algorithm.

We recursively compare the vote count of the grid, which was most often detected for a specific ROI, to the average of all vote counts of the nested ROIs. This method results in a ROI structure and corresponding period values, which were most often detected within a specific section.

The following example demonstrates this voting algorithm for a section with ten frames. The figure below displays the structure of all detected ROIs for this example.

ROI layout for section with 10 frames

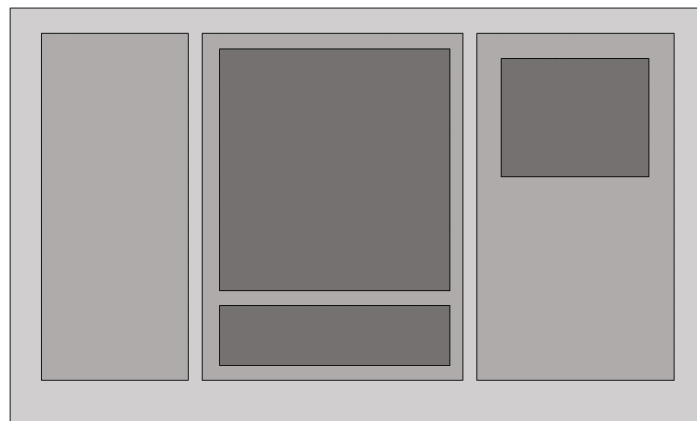


Figure 3.21.: Example ROI structure for determining period values per section.

For each of the ten frames, the best ROI structure with corresponding period values is calculated. Each ROI within the detected structure and their period values count as a vote within the voting algorithm. The following figure shows the results for this example. The vote counts per period values are displayed on the right of the respective ROI.

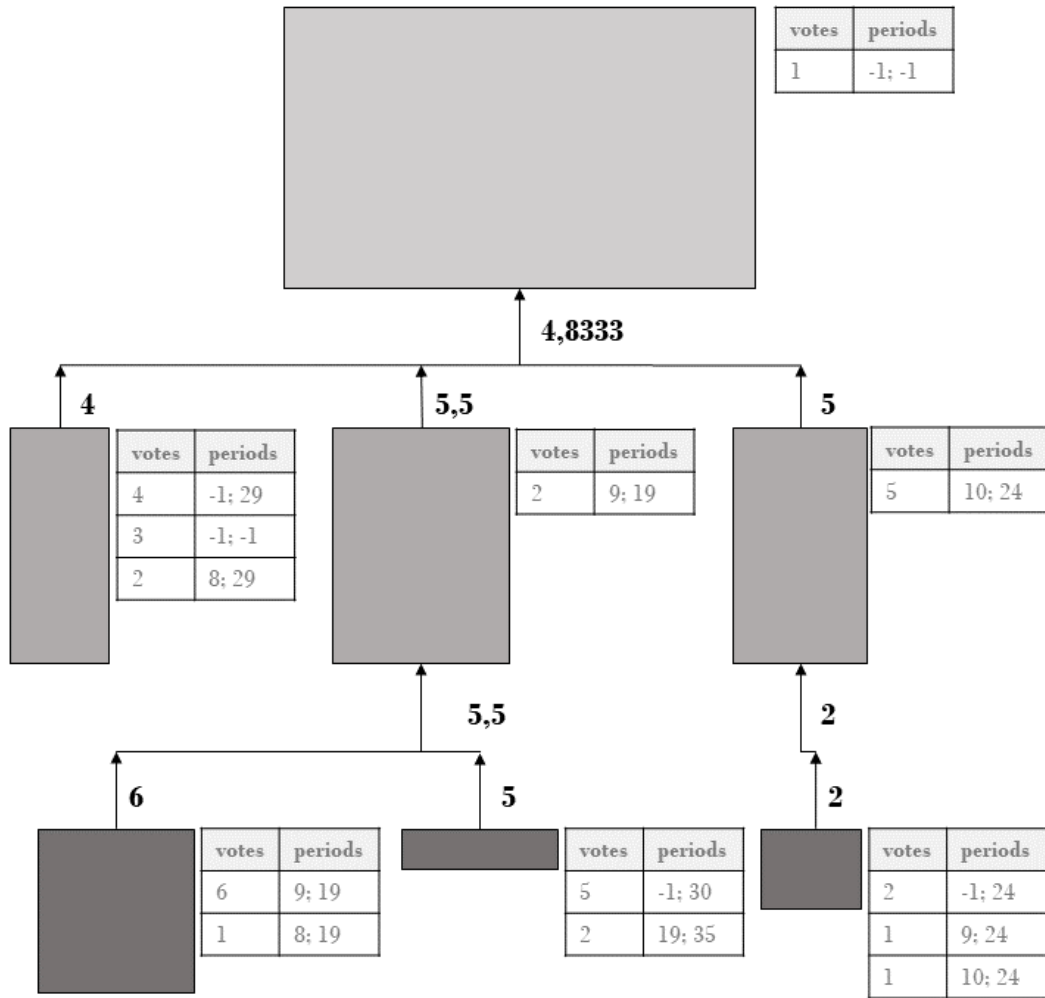


Figure 3.22.: Demonstration of voting algorithm.

Through this voting algorithm, the following structure is obtained.



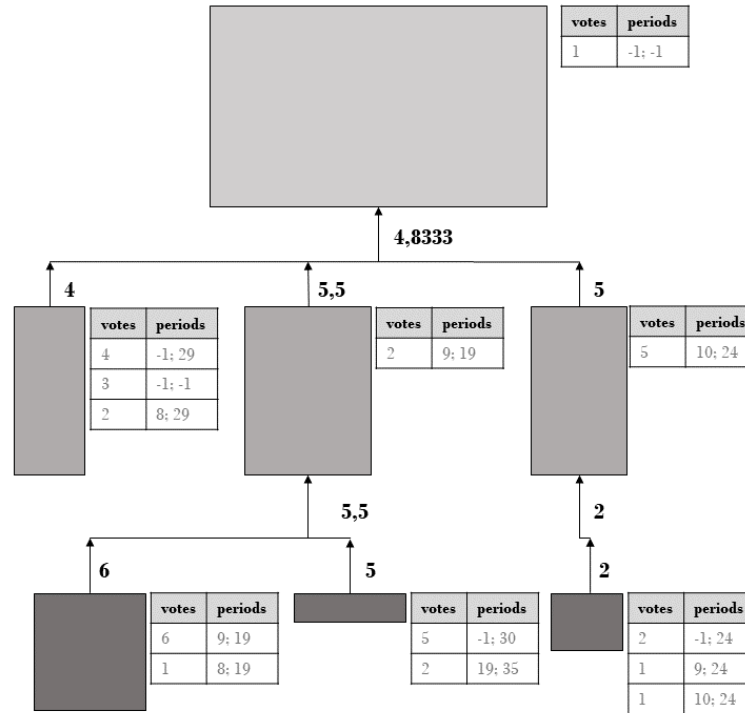


Figure 3.23.: Resulting ROI structure for the example.

### 3.3.4. Text Editor Classification

The resulting set of ROIs consists of all ROIs, which are most likely to be text editors for a specific section and their respective period values. Within this step we want to further analyze the proposed grids, in order to classify the detected ROIs into text editors and other ROIs.

Throughout this process, we have considered a period of -1 (i.e., no period found) as being an acceptable value. Especially within Section 3.3.3 the information of the amount of frames, for which no grid could be detected is preserved. As we are now classifying the ROIs, we take this information into consideration. All ROIs, for which the obtained horizontal or the vertical period equals -1, are classified as a non-text editor and excluded from further analysis.

For all remaining ROIs, we evaluate, how well the identified grid fits the respective ROI. So far, we have only defined the period values. To set up a grid we also need a parameter for the offset in each direction. The offset values  $o_{ver}$ ,  $o_{hor}$  define the shift of the first grid line from the top and left border, respectively. The offset is therefore an integer value between 0 and  $p$ . A row  $j_{roi}$  is said to be a segmentation row of the

grid if  $j_{roi} + o_{ver}$  is divisible by  $p_{ver}$ . Similarly, segmentation columns will be denoted as columns  $i_{roi}$  for which  $i_{roi} + o_{hor}$  is divisible by  $p_{hor}$ .

The period value stays constant throughout all frames. In contrast, the offset might change due to scrolling and changes within the text layout. We therefore need to identify the offset values for each frame individually.

Starting point for the offset detection are the average pixel values. An example of those values can be seen in figure 3.17. The segmentation lines should be in between the characters. We select the offset as the value for which the difference between the sum of pixel values of segmentation lines and the sum of pixel values of intermediate lines is highest.

This difference is also a measure of the quality of our grid. We calculate this difference for each frame. If the average difference lies above a given threshold for both the horizontal and vertical direction, we classify this ROI as a text editor. Otherwise, it is classified as a non editor and therefore excluded from further analysis.

### 3.4. Analysis of Textual Changes

We analyze the changes within all ROIs, which were classified as text editors. In comparison to all previous steps, we will not analyze individual images, but look at two subsequent frames and their differences. As we are only considering ROIs instead of frames, we will use the term image synonymous with the current ROI.

For each pair of subsequent frames, we generate change images as described in Section 3.4.1. Those images are then evaluated with respect to character changes within Section 3.4.2. In case the change values indicate that there might be a non textual change, we analyze the images further as described in Section 3.4.3.

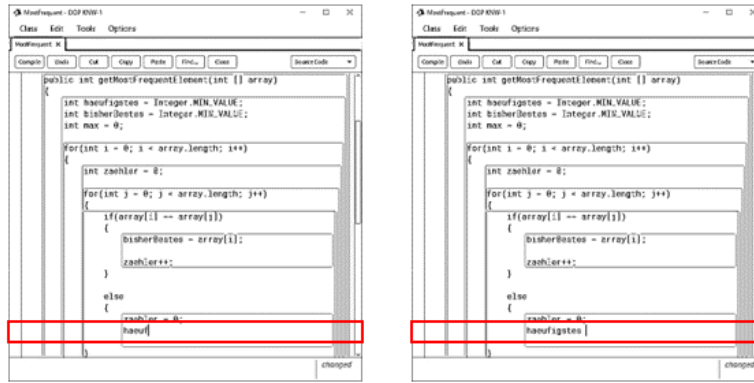
The described methods of Section 3.4.2 and Section 3.4.3 evaluate the subtractive and additive changes separately. Those methods will therefore result in a change value for the added amount of characters and a change value for subtracted amount of characters respectively. To calculate the typing rate as described in Definition 1.2.1, we have to subtract the subtractive change value from the additive change value.

### 3.4.1. Preprocessing

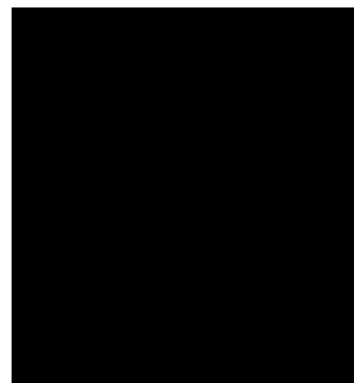
Within the preprocessing step, we emphasize the differences between ROIs of subsequent frames, by calculating the image differences. We obtain those images by pixel wise subtraction.

As inputs, we use the binarized ROIs from Section 3.3.1, where long lines have not yet been excluded. We construct the difference image by subtracting the pixel values of the two subsequent images. As we are processing binarized images, the pixel wise subtraction can only result in one out of three values. The pixel value within the difference image will be 0 if there was no change. The result will be 255 if there was a subtractive change and -255 if there was an additive change.

As the pixel value of -255 cannot be graphically represented, we construct two images. One image represents all additive changes as white pixels. Likewise, the subtractive change image represents all subtractive changes as white pixels. An example of such change images is displayed in the figure below.



additive change



subtractive change

Figure 3.24.: Example of additive and subtractive change images.

Depending on the selected ROI, the change images might also detect changes of non textual elements, for example the BlueJ syntax highlighting or UI elements like the scroll bars. Similar to the grid detection, we can also exclude resulting changes, which cannot represent characters. We therefore apply the Hough Transform, described in Section 3.2.2 to the change images. We can determine the minimum line length, as the diagonal of a character grid  $\sqrt{p_{ver}^2 + p_{hor}^2}$ .

### 3.4.2. Determining Change Values

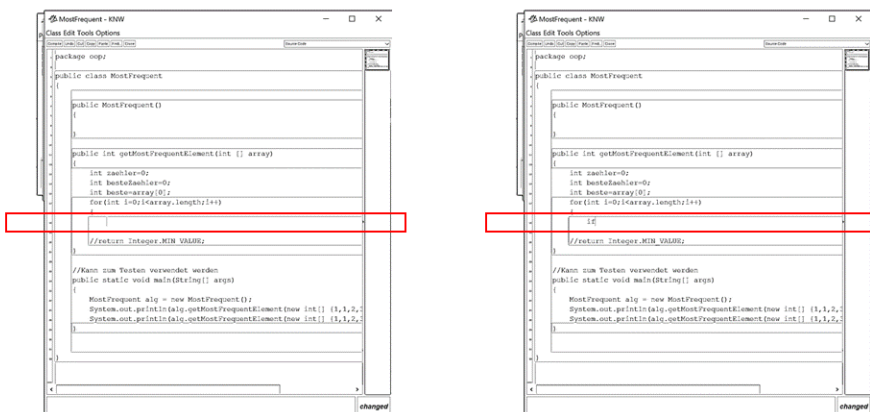
Our aim is to convert the pixel-wise changes into changes of characters. Notably, when typing different characters, the amount of added pixel depends on the appearance of the character. To convert changes of pixel into changes of characters, we evaluate the change images with respect to the obtained grids.

Within Section 3.3, we determined the parameters of the character grid. For a specific

ROI, the period values  $p_{hor}, p_{ver}$  stay constant throughout the section. In contrast, we determined offset values  $o_{hor}, o_{ver}$  for each individual ROI within the section.

All white pixels within the additive change image correspond to characters, which are only present within the second image in the pair of subsequent images. Therefore, we apply the grid parameters of the second image of the pair of frames to the additive change image. Likewise, the grid parameters of the first image are used for the subtractive change image.

The obtained changes are now analyzed with respect to the character grids. For each grid cell, we can analyze, whether a relevant change has occurred or not. We only consider complete grid cells and therefore disregard incomplete grid cells at the border of the ROI. For any given grid cell of a change image, we determine, whether a relevant change occurs. The amount of cells with a relevant change of the additive change image equals the amount of added characters. Similarly, we determine the number of subtracted characters as the amount of grid cells with relevant changes for the subtractive change image.



Corresponding grid cells with additive change:

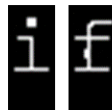


Figure 3.25.: Example of grid cells for which a change occurred.

We observe, that characters lie within the center of a grid cell. Therefore, we can define a padding within each grid cell. A specific cell does only contain a relevant change if

there is at least one white pixel within the inner area of it. Through this criterion, we can exclude noise, which might occur due to syntax highlighting or selection of text.

As a second criterion, we can observe, that characters always contain more than one pixel, which are connected to each other. We therefore only consider white pixel values with an immediately adjacent white pixel in horizontal or vertical direction.

To summarize, all grid cells within a change image are considered relevant, if the following criteria are met:

- At least one change lies within the padded grid cell.
- For at least one change within the padded grid cell, there is an immediately adjacent change.

Notably, this described approach does not consider non textual changes, which happen within text editors. In case the additive change value or the subtractive change value exceeds a certain threshold, we will further analyze the change images as described in Section 3.4.3.

### 3.4.3. Processing of Larger Textual Changes

Within this section, we propose approaches for considering common non textual changes within text editors. For this work we will restrict the analysis to scrolling activities and the appearance of dialog boxes. Notably, both events will result in a high additive or a high subtractive change value.

For change images with larger detected changes, the following steps are performed:

1. In case a vertical shift can be detected, return the change values which include a vertical shift.
2. In case a horizontal shift can be detected, return the change values which include a horizontal shift.
3. In case a dialog box can be detected, return the change values which include the dialog box.

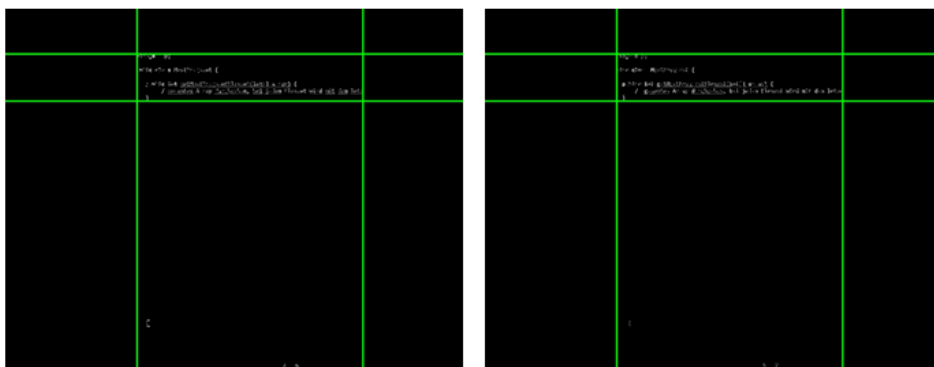
In case neither a change nor a dialog box can be detected, we will either return the initial change values obtained in Section 3.4.2 or 0 as the additive and subtractive change. For this decision, we consider, the similarity of the additive and subtractive

change values. For this analysis we calculate the ratio of the smaller change value to the larger change value. If this ratio lies above a certain threshold, both change values are similarly large. As this might indicate disregarded events within the screen, we will return 0 as the additive and the subtractive change. In contrast, if the ratio is below the threshold, the initial change values are returned.

### Shift Detection

To identify a potential shift, we apply template matching to the subsequent images. This method compares a template image to all potential regions of an input image. This approach results in a comparison value for each location in the input image. [10]

To evaluate whether the larger textual change has been caused by a shift in vertical or horizontal direction, we need to detect the region within the images which is potentially affected by the shift. For this evaluation, we identify the largest rectangular cluster of white pixels within the combined subtractive and additive change image.



additive change

subtractive change

Figure 3.26.: Example of change images with horizontal shift and area of largest coherent change.

For shift detection, we only consider the region within the area of largest coherent change. The templates are extracted from the first of the pair of images and compared with the second image. Within the largest region of coherent change, we use all lines as templates for vertical shift detection and columns for the horizontal shift detection. For each line or column respectively, we identify the maximum comparison value. If this value lies below a certain threshold, we mark this region as not detected.

The result per template can be described as the potential shift for this line or column respectively. We accept a shift, if at least half of all lines or columns result in the same shift value.

In case a shift exists, we evaluate the affected area as three distinct subparts: regions which are visible in both frames, regions which are only visible in the first image, and regions which only appear in the second image. For each subpart, the change values are calculated individually. The result of this method is the sum of the individual additive and subtractive change values.

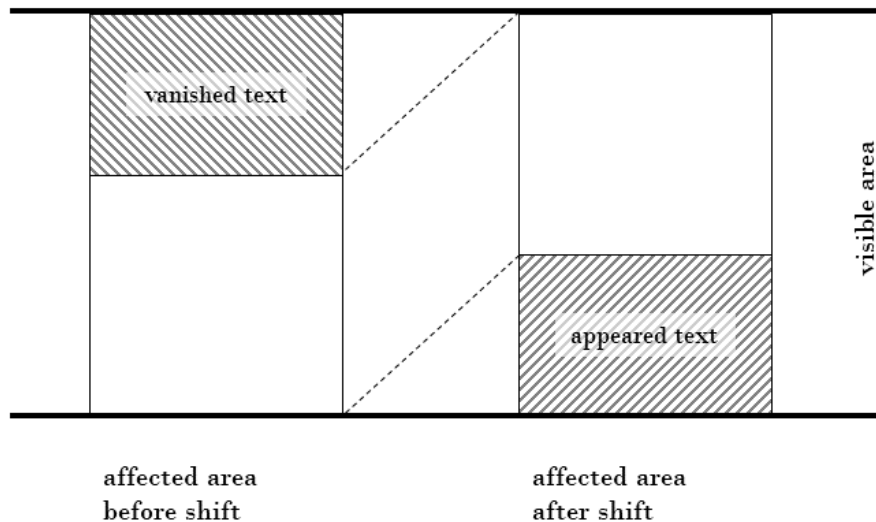


Figure 3.27.: Representation of subparts for subsequent images with a shift.

Regions which appear in both frames, are extracted. The change values within those regions are then calculated as described in Sections 3.4.1 and 3.4.2. Per default, the vanished and appeared texts are only considered, in case the region of the largest coherent change does not span over the entire ROI.



## Dialog Box Detection

We define all rectangular shapes, which might appear within the region of the code editor as a dialog box. This includes menus, auto complete boxes, and pop up windows. A main characteristic of all those boxes is, that after a number of frames, those boxes will vanish again. The area which was covered by those boxes will often stay unchanged.

As for the shift detection, we first of all determine the region of the largest coherent change within the change images. This region is then extracted from the first image  $\mathcal{I}_k$  for which the change value is determined. We know that the next image  $\mathcal{I}_{k+1}$  contains some change within this region. We now want to evaluate whether the change of this region will be reversed within the next  $n$  frames.

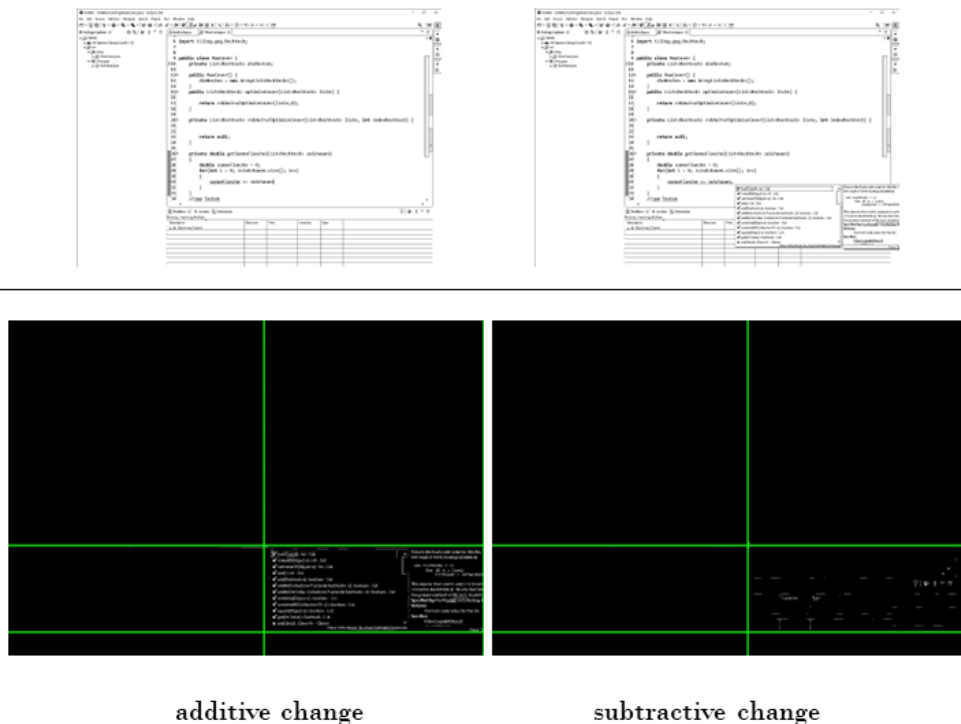


Figure 3.28.: Example of change images with autocomplete box.

Sequentially, we apply template matching to the images  $\mathcal{I}_{k+l}$   $l \in \{2, \dots, n\}$ . The template for this method is the extracted region from image  $\mathcal{I}_k$ . If the result of the template matching lies above a certain threshold, we assume, that the initial text has reappeared in image  $\mathcal{I}_{k+l}$ . In case no such value  $l$  is found, no dialog box is detected.

When a dialog box has been detected within images  $\mathcal{I}_{k+1}$  and  $\mathcal{I}_{k+l-1}$ , we generate the

change images of the given ROI between  $\mathcal{I}_k$  and  $\mathcal{I}_l$ . If no change can be detected within the region of the largest coherent change between  $\mathcal{I}_k$  and  $\mathcal{I}_{k+l}$ , we set all change values within that region for all pairs of subsequent images to zero. We can then recalculate the change values for  $\mathcal{I}_k$  and  $\mathcal{I}_{k+1}$  and proceed.

In case a change could be detected within the region of largest coherent change in between  $\mathcal{I}_k$  and  $\mathcal{I}_{k+l}$ , we only set a sub-region to zero. The following figure displays the possible sub-regions. The white rectangle represents the area of largest coherent change and the gray rectangle within represents the area, for which a change is still detectable between  $\mathcal{I}_k$  and  $\mathcal{I}_{k+l}$ . The area, which is set to zero is the one rectangle of  $rect_{up}$ ,  $rect_{down}$ ,  $rect_{right}$ , and  $rect_{left}$  with the largest area.

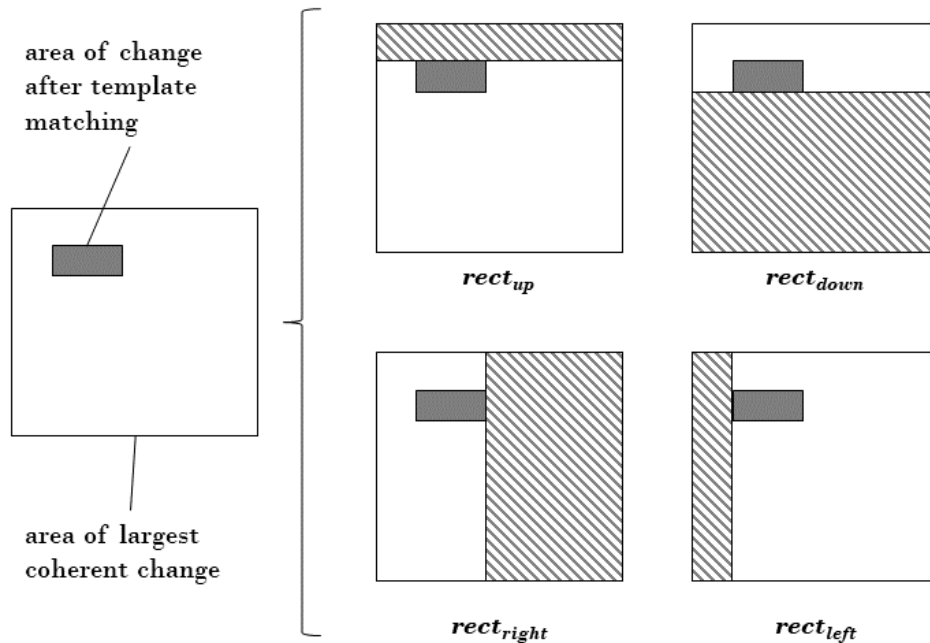


Figure 3.29.: Possible rectangles for further processing.

Within the resulting region, we set all change values for all pairs of subsequent images between between  $\mathcal{I}_k$  and  $\mathcal{I}_{k+l}$  to zero. Afterwards, we can recalculate the change values for  $\mathcal{I}_k$  and  $\mathcal{I}_{k+1}$  and proceed.

## 3.5. Discarded Approaches

The method as described in the sections above is the result of constant trial and error. Within this section, we want to discuss selected approaches, which were considered for this project, but ultimately discarded for various reasons.

As most of the comparable projects use OCRs in order to extract text (see Section 2), we initially tried similar approaches for detecting the change rates. It was quickly discovered that the extracted text by commonly used OCRs, did not result in the quality needed to identify textual changes between frames. For OCRs, which could produce accurate results, it was not viable to process the amount of frames, primarily due to the required financial costs.

For the detection of grid values (see Section 3.3.1), we implemented the method using autocorrelation as well as a Discrete Fourier Transform (DFT). We discarded the DFT, as the detected peaks were not as easily convertible into integer period values.

For the classification of ROIs into text editors and non text editors (see Section 3.3.4), we considered the alignment of text as an additional criterion. It was possible to evaluate, whether a ROI might contain left aligned text. However, since left aligned text is used in many computer windows, this criterion was not suitable for classifying code editors, as most ROIs were classified as such.

Within Section 3.4.3 we discussed a method for dealing with larger textual changes. The proposed method is restricted to the analysis of shifts, as well as the appearance of text boxes. This excludes many other events, which might happen within text editors. We initially implemented an approach that would store text segments throughout a screencast. Through this method, we could have distinguished newly written text from reappeared text. This approach was discarded due to the required storage and runtime complexity for such a global repository.

## 4. Implementation

### 4.1. Usage Specifications

This project has been designed as a Java console application. We used Java 15.0.1 for implementation. The only dependency is the OpenCV library [6], version 4.5.3.

This application requires three arguments, which are described in more detail below. To run this application successfully, the three arguments must be entered in the specified order.

1. **Path to Input Image Directory:** path to the directory, in which the input images are located. A detailed description of the requirements for the input can be found in Section 4.1.1.
2. **Path to Output Directory:** path to the directory, in which the resulting CSV file will be stored. A detailed description of the output of this application can be found in Section 4.1.2.
3. **Name:** a string, which will be used as the filename for the output file. Furthermore, this string will be used within log messages.

#### 4.1.1. Requirements for Input

All frames of the screencast need to be stored within the same directory. The directory must only contain frames from one screencast. The path to the directory is stated as the first input argument. The images within this directory must fulfill the following requirements:

- The file format of the images must be supported by OpenCV. A detailed list of all supported formats can be found in the OpenCV API [11]. All images, which

were used for implementation and evaluation were of type JPEG (.jpg).

- The file names of the frames must be in sequential order. All retrieved images will be sorted alphabetically according to the file name and processed in that order. The frames of all used screencasts within this work, were named with a display ID followed by a sequential number ("D00\_img000000001", "D00\_img000000002", ...).
- The resolution for all images within a screencast must stay constant. If the resolution changes for subsequent images, they will be considered as different displays.

A further requirement for all screencasts, which were used for this project, is, that the text editor of the IDE is fully visible in at least one display. We had to discard eight screencasts, as they did not meet this requirement.

### 4.1.2. Resulting Data

The output of this application is one CSV file per detected display. Those tables contain the detected typing rates for each pair of subsequent frames. The output files will be written into the specified output directory. The file name of each resulting CSV file will be the specified name plus an extension for the display number "\_Dxx", where "xx" denotes the number of the respective display. If a CSV file with the same name already exists within the specified directory, it will be overwritten.

The resulting CSV table contains the two columns "frame" and "change". The number of rows will be one less than the number of frames per detected display. The first value in each row gives the index of the pair of subsequent frames that are compared. The indices are consecutive numbers starting from 0. The corresponding typing rate for this pair of subsequent frames is stated in column "change". The typing rate between image  $\mathcal{I}_k$  and  $\mathcal{I}_{k+1}$  can therefore be found in the second column of the table entry, in which the first column equals the values  $k - 1$ .

In addition, the application will print log messages, containing information on the current state of the workflow.

## 4.2. Project Structure

The application contains the class `Main.java` in the root directory as the main entry point of the program. Furthermore, the root directory contains of the following packages:

- `videoanalysis`  
This package consists of all classes, in which the algorithmic approach is implemented. Furthermore, this package contains the class `Config.java` in which all constant values and parameters, which are used within this project, are contained.
- `videoanalysis.content`  
This package contains all classes, which represent geometric features of an image.
- `videoanalysis.input`  
This package contains one class, which provides the frames necessary for image processing.
- `videoanalysis.output`  
The class within this package stores the detected typing rates per frame and writes the resulting CSV file.

For further details, the UML class diagrams and class documentation can be found in the appendix (II, III).

The central component of this project is the class `videoanalysis.ScreencastAnalyzer.java`. Given a path to the directory in which the frames of a screencast are stored, this class coordinates the workflow of the proposed method. The following figure gives an overview over the classes in which the individual steps of the proposed method are implemented.

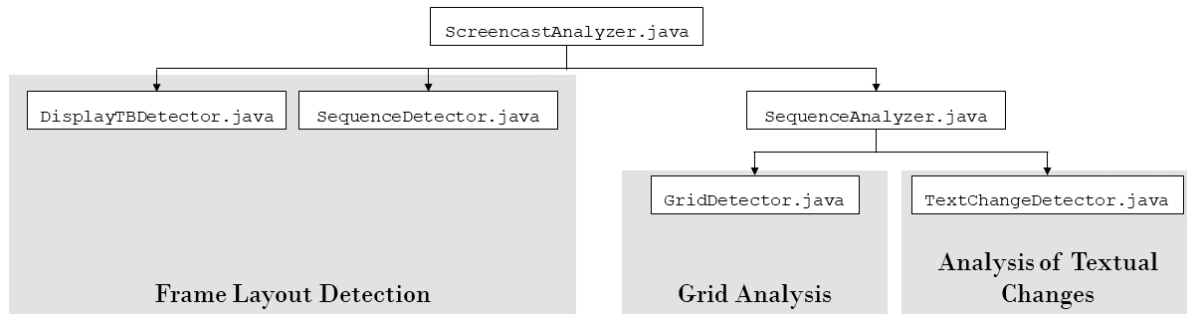


Figure 4.1.: Interactions between logic classes.

### 4.2.1. Implementation of Frame Layout Detection

The ROI layout detection, which has been described in Section 3.2, is implemented within the classes `DisplayTBDetector.java` and `SequenceDetector.java`.

`DisplayTBDetector.java` stands for Display-Taskbar-Detector. It contains all methods which are used to evaluate the resolutions of the images within the input directory. Furthermore, it contains methods for detecting regions at the bottom of each individual screen, in which no or few changes occur over the duration of the recording.

The methods within the class `SequenceDetector.java` group frames into sequences with a similar layout of topmost ROIs. All common feature detection techniques for single images are implemented within the class `FeatureDetector.java`.

### 4.2.2. Implementation of Grid Analysis

The further analyses of sequences with similar ROI layout are coordinated by the central class `SequenceAnalyzer.java`. Within this class, an instance of `GridDetector.java` is created, in which the approach described in Section 3.3 is implemented.

The aim of this class is to find the parameters of a character grid. Furthermore, a method for evaluating the fit of the grid is implemented. For the applied feature detection techniques, an instance of the class `FeatureDetector.java` is created.

### 4.2.3. Implementation of Analysis of Textual Changes

The approach for analyzing textual changes as described in Section 3.4 is implemented in the class `TextChangeDetector.java`. All methods within this class are invoked by the class `SequenceAnalyzer.java`.



# 5. Evaluation

## 5.1. Setup

### 5.1.1. Ground Truth

In section 1.2.2, we defined *a-typing* as a method to convert the typing rate into a binary output over a given time interval. A user is said to be *a-typing*, if the amount of altered characters within a given frame interval is greater than  $a$ .

The frame intervals for the manual annotation are chosen, so that they span over one minute long time intervals. For each time interval, it was manually decided, whether the user was typing or not. Within this chapter, we will mainly evaluate the screencasts as *4-typing*.

All methods used for this evaluation can be found for example in [21].

For each of the 56 evaluation screencasts, we compared the manually identified activity to the results of our method. Depending on the actual identified value and the resulting value of our method, we will categorize each minute into one of four categories: true positives (TP), false negatives (FN), false positives (FP) and true negatives (TN). The total amount of identified intervals with typing activities of one screencast is denoted as  $P$ , and the amount of non-typing activities as  $N$ . This classification is known as the confusion matrix and will be the basis for this evaluation.

		<b>method outcome</b>	
		P	N
<b>annotated values</b>	P	TP	FN
	N	FP	TN

Table 5.1.: Confusion matrix.

## 5.1.2. Technical Setup

The technical details of the server, which we used for evaluation, are as follows:

- 128GB RAM
- 4 processors of type AMD EPYC Processor, with 16 cores each
- CPU tact rate of 2495.312 MHz

## 5.2. Results

### 5.2.1. Evaluation of Accuracy

As a first measure for the performance of the proposed method, we will analyze the screencasts according to accuracy. This performance indicator is defined as the ratio between correctly classified sections to the total amount of all sections.

$$accuracy = \frac{TP+TN}{P+N}$$

The results of the accuracy values per screencast are displayed below.

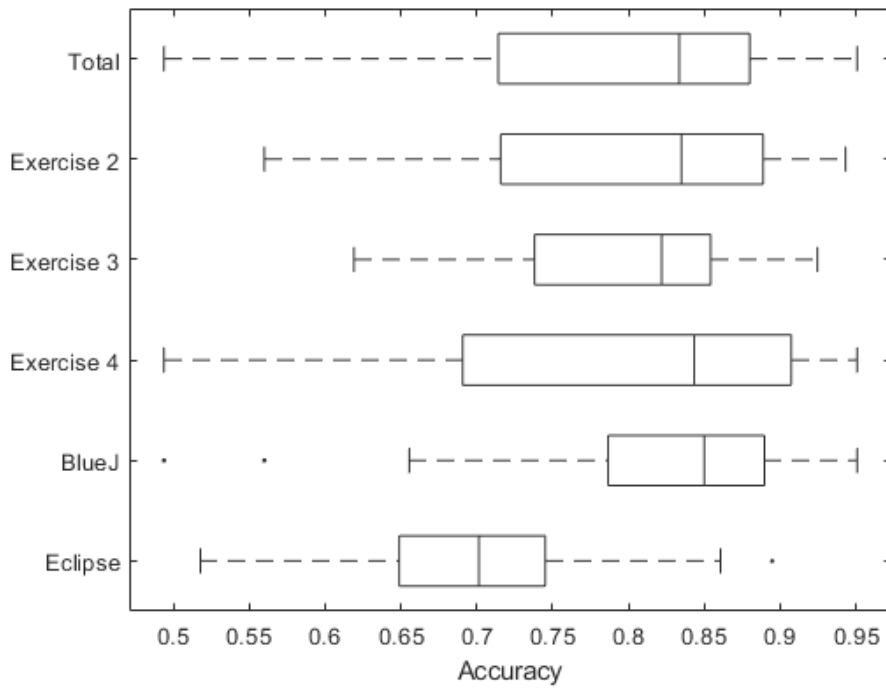


Figure 5.1.: Box plot of accuracy.

The overall median accuracy lies at 83.4%. To further evaluate the differing performances within each subgroup, the individual exercises as well as IDE-preferences, we perform a Mann Whitney U test for all pairs within a subgroup. This test has been chosen, because the resulting data is ordinal, both groups are independent of each other, and the results are not normally distributed. For two samples  $X, Y$  with sample sizes of  $n, m$  respectively, the  $U$  statistic is computed as follows:

$$U = \sum_{i=1}^n \sum_{j=1}^m S(X_i, Y_j)$$

with

$$S(X, Y) = \begin{cases} 1 & X > Y \\ 0.5 & Y = X \\ 0 & X < Y \end{cases}$$

As the sum of our sample sizes is always greater than 20, we can transform  $U$  into a standardized normal distribution using the following formula:

$$Z = \frac{U - \frac{n*m}{2}}{\sqrt{\frac{n*m*(n+m+1)}{12}}}$$

The resulting  $p$ -values for all pairings of exercises as well as the IDEs are displayed in the table below.

Group 1 0	Group 2	$p$ -value
Exercise 2	Exercise 3	0.5269
Exercise 2	Exercise 4	0.8953
Exercise 3	Exercise 4	0.4648
BlueJ	Eclipse	0.0004

Table 5.2.: Resulting  $p$ -values for group pairings.

For all pairs of exercise results, the null hypotheses that the values are samples from distributions with equal means is with a certainty of 99% not rejected. On the other hand, when applying this test to the results for the grouping by IDEs, the hypotheses is rejected with the same level of certainty.

We can conclude that the proposed method performs similarly for different exercises. In contrast, the obtained results for BlueJ screencasts are better than for Eclipse. We therefore conclude that the selected Eclipse screencasts in the development group were less representative of all Eclipse screencasts.

### 5.2.2. Evaluation of Precision and Recall

As further analysis of the behavior of this method, we will analyze each screencast with respect to precision and recall. Precision is the ratio of true positives to all positively identified values. It is therefore an indication of how many of the sections classified as typing, are actually relevant.

$$precision = \frac{TP}{TP+FP}$$

Recall is defined as the ratio of true positives to all positive values within the ground truth. This value therefore specifies the ratio of retrieved typing sections to the total amount of manually identified typing sections.

$$recall = \frac{TP}{TP+FN}$$

The following figure shows the results of precision and recall for all screencasts within the evaluation group. As the performance varies significantly between used IDEs, the

data within this figure is distinguished by the used IDE.

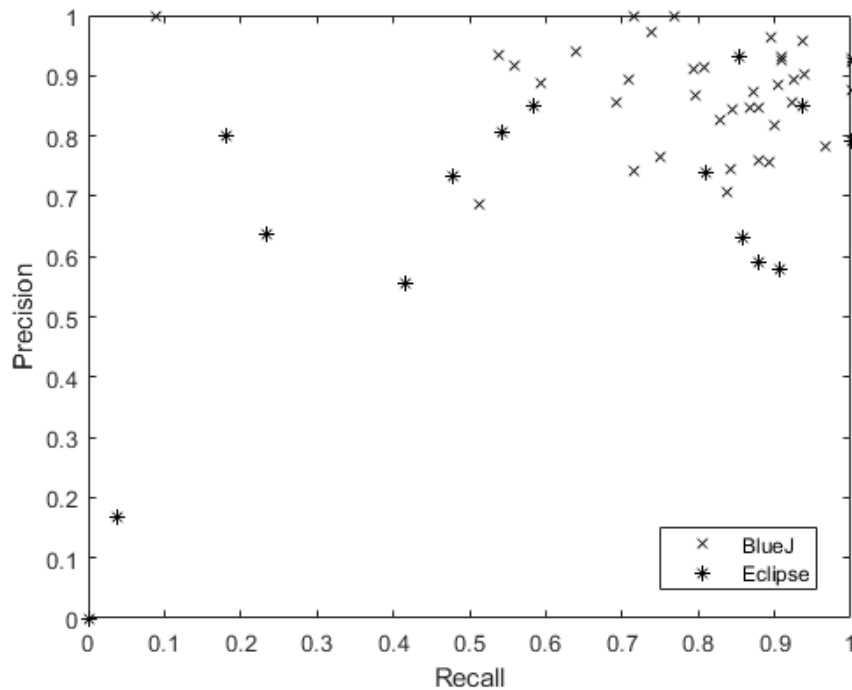


Figure 5.2.: Scatter plot of precision and recall.

A commonly used measure to combine precision and recall is the  $F_\beta$  score. This performance indicator is dependent on the value  $\beta$ , which controls the balance between precision and recall. As becomes apparent from the formula,  $\beta = 1$  means, that precision and recall are weighted equally. By applying a smaller  $\beta$ -value, precision is weighted higher. When  $\beta > 1$ , recall will be weighted more.

$$F_\beta = \frac{(1+\beta^2)*precision*recall}{\beta^2*precision+recall}$$

As we specified Section 1.2.2, for our use case, a high recall value is more essential than precision. Therefore, we will be analyzing the  $F_2$  score for our results. Using the precision and recall values for each screencast, we calculated the respective  $F_2$  score. The box plot below shows the resulting values for all screencasts as well as grouped according to exercises and IDEs.

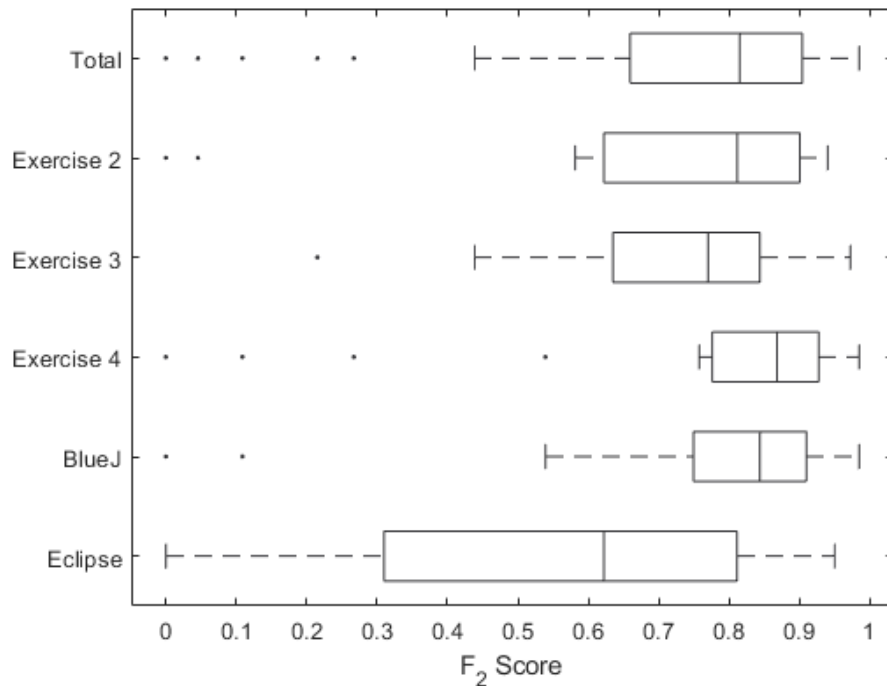


Figure 5.3.: Box plot of  $F_2$  score.

For all screencasts, the median  $F_2$  score lies between 43.9% and 98.5%, excluding the identified outliers, which each have an  $F_2$  score of less than 27%. The median for all screencasts within the evaluation group lies at 81.5%. The median  $F_2$  scores grouped by exercises lie at 81.1% for Exercise 2, 77.0% for Exercise 3, and 86.8% for Exercise 4. The  $F_2$  scores grouped by IDE lie further apart with 84.4% for BlueJ and 62.2% for Eclipse.

### 5.2.3. Analysis of *a*-typing

So far, all analyses were performed for *4*-typing as binarization classifier. Within this section, we analyze, how the binary performance indicators are affected by different  $a$  values. In general, a low  $a$  value will result in less typing intervals, which are incorrectly classified as true negatives. As a result, there might be more sections falsely classified as positives. An  $a$  value of -1 means, that every interval will be classified as containing typing. In contrast, a high  $a$  value is expected to result in more false negative values but in less sections, which are false positives.

A receiver operating characteristic (ROC) curve is a tool to evaluate the performance of an application for different parameters. It displays two binary classifiers for various classification parameters. To construct the ROC curve, we need to calculate the recall

and false positive rate (FPR) for each  $a$  value within a specified interval.

The false positive rate is the ratio between false positives to all negative values within the ground truth. This value therefore indicates, how many non-typing sections have been falsely classified as containing typing.

$$FPR = \frac{FP}{FP+FN}$$

We will calculate recall and FPR for each screencast within the evaluation group for  $a$  values in between -1 and 90. For any given  $a$  value, we will therefore obtain 56 values for recall and FPR respectively. For this evaluation, we will use the median recall and FPR for each  $a$  value. The following figure shows the resulting ROC curve. The solid line displays the median values for all 56 screencasts. In addition, the plot shows the median recall and FPR per IDE. As a comparison, the figure also displays the base curve of the ROC curve as a dashed line. This curve marks the expected recall and FPR, if each section would be randomly classified with equal chances as containing typing or non-typing.

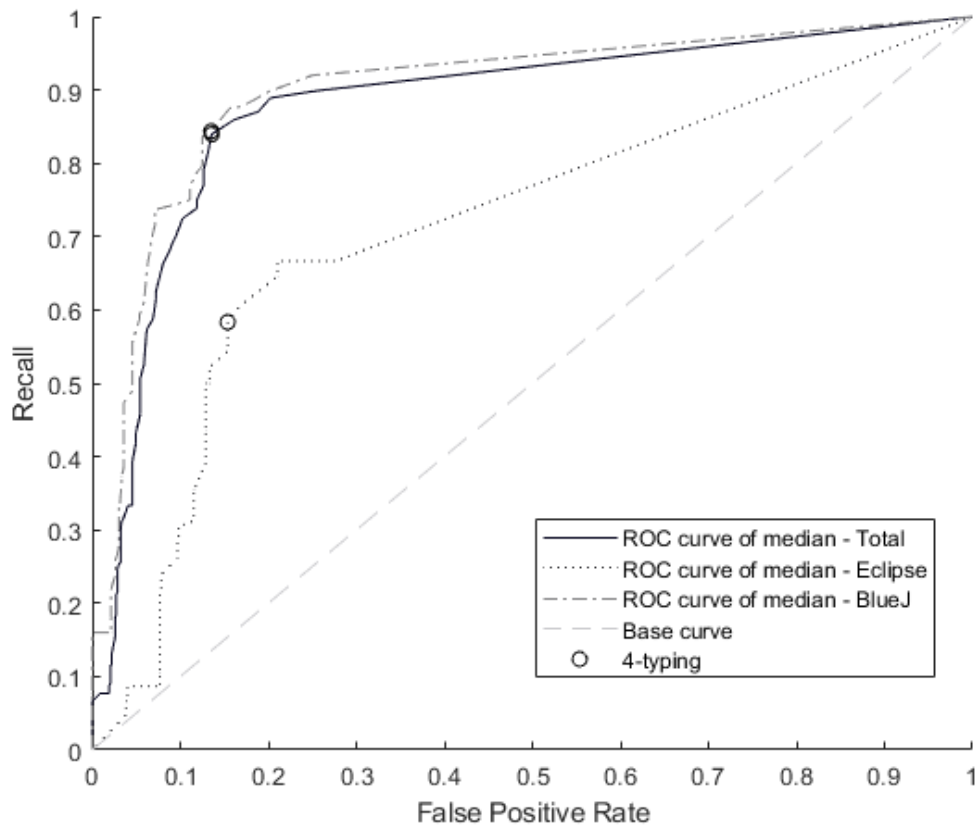


Figure 5.4.: ROC curve.

To interpret this figure, we will use the area under the curve (AUC). This measure calculates the integral of the ROC curve in between 0 and 1. The AUC value for all screencasts within the evaluation group is 0.880. The AUC value for BlueJ screencasts lies at 0.900. For Eclipse, the value is lower with 0.715.

Similar to the ROC curve, we can evaluate the precision and recall values for different  $a$  values using the Precision-Recall curve. For this analysis, the recall values are displayed on the horizontal axis and the precision values on the vertical axis, respectively. For a specific  $a$  value, we calculated precision and recall for all 56 screencasts within the evaluation group. The curve was then constructed using only the median values. The solid line shows the results of all screencasts within the evaluation group. Additionally, the figure displays the median values per IDE.



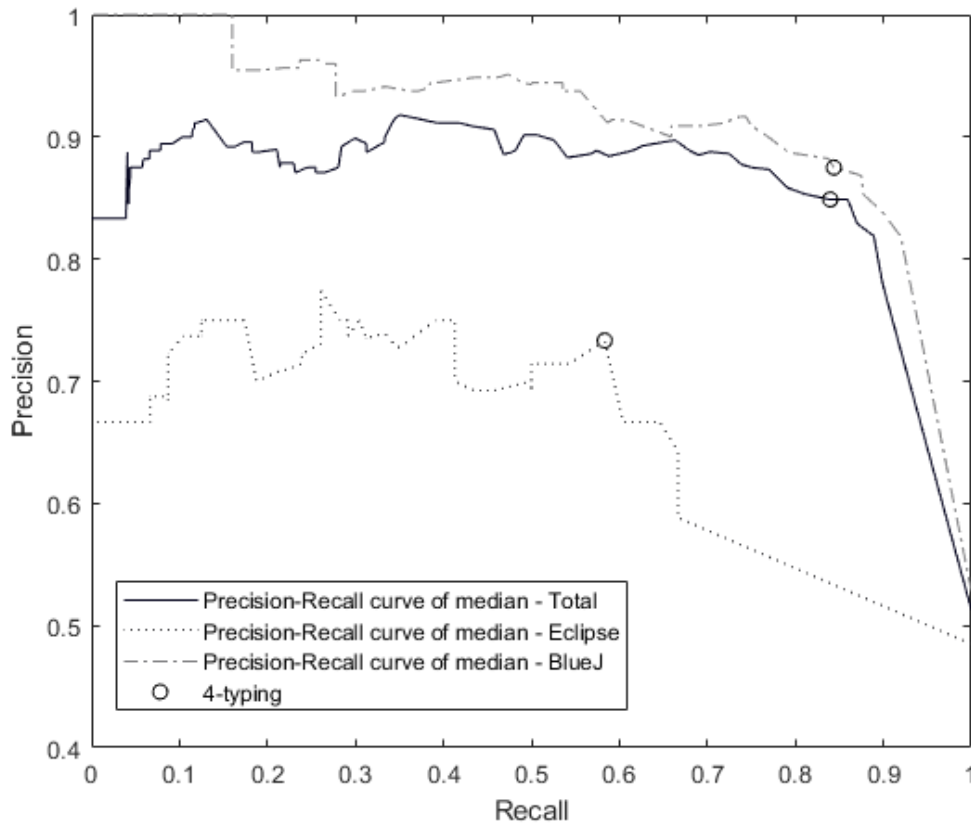


Figure 5.5.: Precision-Recall curve.

The resulting curves can also be evaluated using the AUC measure. For all screencasts within the evaluation group, the AUC is 0.859. For BlueJ screencasts, the value lies slightly higher at 0.914. The AUC measure for Eclipse screencasts is 0.654.

#### 5.2.4. Evaluation of Larger Textual Changes

So far, we have only analyzed the binary output but have not yet quantified the behavior of the typing rate itself. As the typing rate measures the amount of altered characters, we will evaluate the typing rate between frames, which contain a larger textual change.

For this evaluation, we will look at the first copy and paste activity over multiple lines within selected screencasts. We restrict this analysis to recordings of exercises 3 and 4, as users always perform at least one such activity within those recordings. This restricts our analysis to 37 screencasts. For those screencasts, we extracted the frame indices, in between which the first copy and paste activity over multiple lines is performed.

Notably, a copy and paste activity is often performed to replace preexisting text. Depending on the amount of new characters to preexisting characters, this typing rate can either be positive or negative. We will therefore look at the absolute typing rate value.

For comparison, we extract the absolute values of all non-zero typing rates within true positive intervals. Given a sorted sequence of all those extracted values, we calculate the rank of the larger textual change. As the total number of extracted values per screencast varies, we will compare the ratio of the calculated rank to the total amount of extracted values. Our analysis shows, that the majority of relative ranks are at 0. In contrast, we also identified three outliers with values over 0.99.

relative ranks of 0	relative ranks > 0.99
34	3

Table 5.3.: Relative rank of larger textual changes.

Those extreme results can be explained by our implementation. As described, the proposed method sets the change value to zero if the additive and the subtractive change each lie over a certain threshold. When a user copies text over a preexisting text, we are therefore not able to detect the correct change value.

### 5.2.5. Exemplary Study for Night Mode

Within this section, we perform an exemplary study in order to evaluate the behavior of our method for unknown UI settings. For this evaluation, we analyze the three screencasts, which use the Eclipse-IDE in night mode, and for which the exercises are not yet known.

The table below shows the results of the performance indicators for all binary classifiers.

Exercise	Accuracy	Precision	Recall	$F_2$
Exercise 2	91.3%	82.8%	85.7%	85.1%
Exercise 3	93.3%	100%	72.7%	76.9%
Exercise 4	68.3%	47.4%	100%	81.8%

Table 5.4.: Results of exemplary study.

All results lie within the expected values for the respective exercises. When compared with the results of other Eclipse screencasts, the accuracy of exercises 2 and 3 exceed the expected values. Consequently, we can assume, that the proposed method has a similar behavior for IDEs in night mode.

## 6. Conclusion

### 6.1. Summary

Within this work, we aimed to distinguish semantically relevant sections of screencasts from irrelevant sections. In particular, this work proposes a method to identify typing sections within programming screencasts. Through the evaluation of unknown recordings, we conclude that the proposed method is able to distinguish between relevant and irrelevant sections. The chosen approach is able to correctly classify sections, the quantification of change is imprecise, though.

Furthermore, we establish that the performance of our method does not depend on the specific programming task, which is solved within the screencast. In contrast, the performance of the method varies for different IDEs. We also performed an exemplary study of three screencasts with radically different UI settings (night mode). This study showed that the results do not differ from default settings.

Previous work focused on global extraction methods. In contrast, our research question required the extraction of changes in between frames. In order to obtain reliable results, we focused this approach on detecting features of ROIs as well as features of the content within. It turned out that this method was well suited for the task of finding relevant sections in screencasts.

### 6.2. Further Research

Based on our evaluation of larger textual changes, further research is needed in order to correctly quantify the amount of textual changes. The main challenge for this part of the research question was the distinction between actual textual changes and other activities. For example, the proposed method can not distinguish between copy-paste activities over preexisting text and tab changes within an IDE.

As described in Chapter 2, similar research questions were tackled using deep learning

approaches and yielded comparable results. We would expect that similar approaches would also work well in our context. In order to implement a deep learning approach, it would be required to put significantly more effort into manually labeling screencasts. An interesting research question would be the comparison between our results and those obtained through other approaches.

As our data for development and evaluation originate from similar programming exercises, we expect the workflow within those recordings to be similar. To better understand the behavior of the proposed approach for unknown setups, further research could evaluate the method for a broader variety of programming screencasts.

# Bibliography

- [1] Mohammad Alahmadi et al. “Accurately Predicting the Location of Code Fragments in Programming Video Tutorials Using Deep Learning”. In: *PROMISE '18*. Ed. by Burak Turhan, Ayse Tosun, and Shane McIntosh. ICPS: ACM international conference proceeding series. New York, New York: The Association for Computing Machinery, 2018, pp. 2–11. ISBN: 9781450365932. DOI: 10.1145/3273934.3273935
- [2] Mohammad Alahmadi et al. “Code Localization in Programming Screencasts”. In: *Empirical Software Engineering* 25.2 (2020), pp. 1536–1572. ISSN: 1382-3256. DOI: 10.1007/s10664-019-09759-w
- [3] Lingfeng Bao et al. “Enhancing developer interactions with programming screencasts through accurate code extraction”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Ed. by Prem Devanbu, Myra Cohen, and Thomas Zimmermann. New York, NY, USA: ACM, 11082020, pp. 1581–1585. ISBN: 9781450370431. DOI: 10.1145/3368089.3417925
- [4] Lingfeng Bao et al. “psc2code”. In: *ACM Transactions on Software Engineering and Methodology* 29.3 (2020), pp. 1–38. ISSN: 1049-331X. DOI: 10.1145/3392093
- [5] George E. P. Box et al. *Time series analysis: Forecasting and control / George E.P. Box, Gwilym M. Jenkins, Gregory C. Reinsel, Greta M. Ljung*. Fifth edition. Wiley series in probability and statistics. Hoboken, New Jersey: Wiley, 2016. ISBN: 978-1-118-67502-1
- [6] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb's Journal of Software Tools* (2000)
- [7] Kenneth Dawson-Howe. *A practical introduction to computer vision with OpenCV*. Chichester: Wiley, 2014. ISBN: 9781118848456

- [8] Eclipse Foundation. *Eclipse IDE Working Group | The Eclipse Foundation*. 16/05/2022.  
<https://eclipseide.org/>
- [9] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. [New ed.] Upper Saddle River, N.J: Prentice Hall, 2002. ISBN: 0-201-18075-8
- [10] Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins. *Digital Image processing using MATLAB*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2004. ISBN: 0-13-008519-7
- [11] *Imgcodecs (OpenCV 3.4.17 Java documentation)*. 21/05/2022.  
[https://docs.opencv.org/3.4/javadoc/org/opencv/imgcodecs/Imgcodecs.html#imread\(java.lang.String,int\)](https://docs.opencv.org/3.4/javadoc/org/opencv/imgcodecs/Imgcodecs.html#imread(java.lang.String,int))
- [12] Kandarp Khandwala and Philip J. Guo. “Codemotion: Expanding the Design Space of Learner Interactions with Computer Programming Tutorial Videos”. In: *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. Ed. by Scott Klemmer. ACM Other conferences. New York, NY: ACM, 2018, pp. 1–10. ISBN: 9781450358866. DOI: 10.1145/3231644.3231652
- [13] Michael Kölling and John Rosenberg. *BlueJ*. 28/03/2022.  
<https://www.bluej.org/>
- [14] J. Matas, C. Galambos, and J. Kittler. “Robust Detection of Lines Using the Progressive Probabilistic Hough Transform”. In: *Computer Vision and Image Understanding* 78.1 (2000), pp. 119–137. ISSN: 10773142. DOI: 10.1006/cviu.1999.0831
- [15] *OpenCV: Image Filtering*. 05/12/2021.  
[https://docs.opencv.org/4.x/d4/d86/group\\_\\_imgproc\\_\\_filter.html#gaa13106761eedf14798f37aa2d60404c9](https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#gaa13106761eedf14798f37aa2d60404c9)

- [16] J. Ott et al. “Learning Lexical Features of Programming Languages from Imagery Using Convolutional Neural Networks”. In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 2018, pp. 336–3363
- [17] Luca Ponzanelli et al. “Automatic Identification and Classification of Software Development Video Tutorial Fragments”. In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 464–488. ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2779479
- [18] Luca Ponzanelli et al. “Too long; didn’t watch!” In: *ICSE’16*. Ed. by Laura Dillon, Willem Visser, and Laurie Williams. [New York]: ACM, Association for Computing Machinery, 2016, pp. 261–272. ISBN: 9781450339001. DOI: 10.1145/2884781.2884824
- [19] Thomas Risse. “Hough transform for line recognition: Complexity of evidence accumulation and cluster detection”. In: *Computer Vision, Graphics, and Image Processing* 46.3 (1989), pp. 327–345. ISSN: 0734189X. DOI: 10.1016/0734-189X(89)90036-4
- [20] Hanno Scharr. “Optimale Operatoren in der Digitalen Bildverarbeitung”. PhD thesis. Heidelberg University Library, 2000. DOI: 10.11588/heidok.00000962. <http://archiv.ub.uni-heidelberg.de/volltextserver/962/>
- [21] Alaa Tharwat. “Classification assessment methods”. In: *Applied Computing and Informatics* 17.1 (2021), pp. 168–192. ISSN: 2210-8327. DOI: 10.1016/j.aci.2018.08.003
- [22] Shir Yadid and Eran Yahav. “Extracting code from programming tutorial videos”. In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Ed. by Eelco Visser. New York, NY: ACM, 2016, pp. 98–111. ISBN: 9781450340762. DOI: 10.1145/2986012.2986021
- [23] Dehai Zhao et al. “ActionNet: Vision-Based Workflow Action Recognition From Programming Screencasts”. In: *2019 IEEE*. Piscataway, N.J.: IEEE, 2019, pp. 350–361. ISBN: 978-1-7281-0869-8. DOI: 10.1109/ICSE.2019.00049



# Appendix

## I. Input Data Analysis

Student ID	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	$\Sigma$
1	BlueJ	BlueJ	BlueJ	BlueJ			4
2		BlueJ	BlueJ	BlueJ			3
3		BlueJ	BlueJ	BlueJ			3
4	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
5		BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	5
6		BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	5
7	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ		5
8	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
9	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse		5
10	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
11	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
12	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ		5
13	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ		5
14		BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	5
15		BlueJ	BlueJ	BlueJ	BlueJ		4
16		BlueJ	BlueJ	BlueJ	BlueJ		4
17	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
18	BlueJ	BlueJ	BlueJ		BlueJ		4
19	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
20		x					1
21		BlueJ	BlueJ	BlueJ	BlueJ		4
22	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse		5
23			BlueJ	BlueJ			2
24	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
25		BlueJ	BlueJ	BlueJ	BlueJ		4
26	BlueJ	BlueJ		BlueJ			3
27	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
28	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
29		BlueJ		BlueJ			2
30	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
31	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
32	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
33	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
34	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
35	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
36	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
37	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	6
38	BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	Eclipse	6
39	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
40	x	x	x	x	x	x	6
41	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	Eclipse	6
42	BlueJ	BlueJ	Eclipse	Eclipse	Eclipse	Eclipse	6
43	BlueJ	x		BlueJ	BlueJ	BlueJ	5
44		BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	5
45		BlueJ	BlueJ	BlueJ	BlueJ	BlueJ	5
	<b>31</b>	<b>44</b>	<b>41</b>	<b>43</b>	<b>38</b>	<b>28</b>	

evaluation
testing & development
night mode
not used

x - unusable due to technical difficulties

## II. UML class diagramm

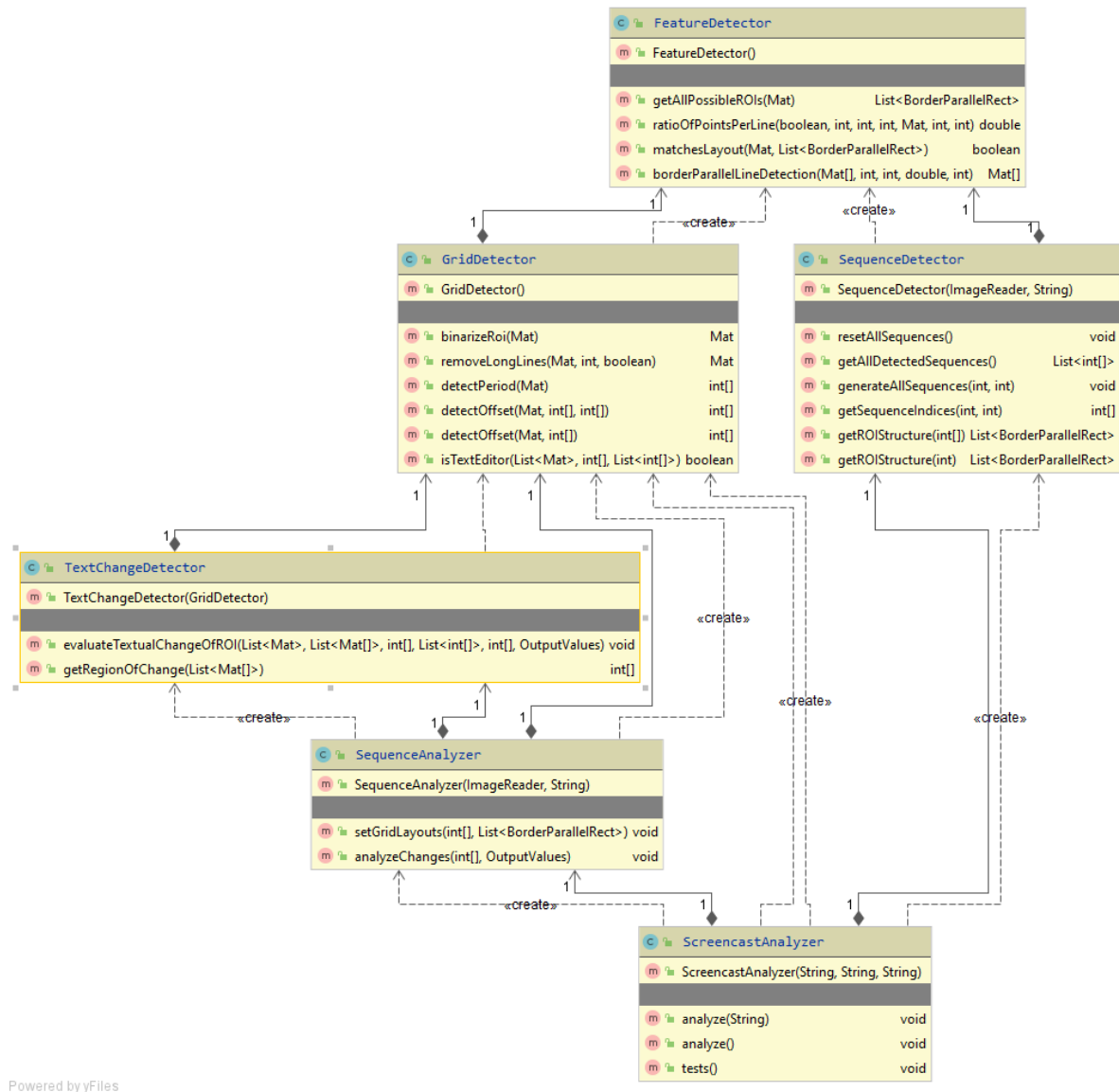


Figure I.: Package videoanalysis: classes and their dependencies

## III. Code Documentation

Package com.unileoben

## Class Main

java.lang.Object  
com.unileoben.Main

```
public class Main
extends java.lang.Object
```

This class contains the main entry point for this application.

### Constructor Summary

#### Constructors

Constructor	Description
Main()	

### Method Summary

#### All Methods Static Methods Concrete Methods

Modifier and Type	Method	Description
static void	main(java.lang.String[] args)	This method is the main entry point for this application.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### Main

```
public Main()
```

### Method Details

#### main

```
public static void main(java.lang.String[] args)
```

This method is the main entry point for this application.

#### Parameters:

args - Three input arguments are required for this application. The arguments must be stated in the order presented below: - An input directory in which the input frames are stored - A directory path, in which the output is stored - A descriptive title under which the output is stored

Package com.unileoben.videoanalysis

## Class Config

java.lang.Object  
com.unileoben.videoanalysis.Config

```
public class Config
extends java.lang.Object
```

This class contains all constant values and parameters, which are used within this project.

### Field Summary

#### Fields

Modifier and Type	Field	Description
static int[]	<code>dimensions</code>	General: resolution of display
static int	<code>FD_GEOMETRIC_DISTANCE</code>	Line Clustering: neighborhood in pixel for detection of line cluster
static double	<code>FD_LAYOUT_MATCH_SIDE_TH</code>	Layout Matching: ratio of white pixel to side length for layout matching
static int	<code>FD_MAX_LINE_GAP</code>	Hough Transform Line: maximum pixel gap for Hough Transform
static double	<code>FD_MAX_ROI_SIZE</code>	Rectangle Detection: maximum ratio of rectangle area to total screen for possible ROIs
static double	<code>FD_MIN_ROI_SIZE</code>	Rectangle Detection: minimum ratio of rectangle area to total screen for possible ROIs
static double	<code>FD_MINIMUM_OVERLAP</code>	Line Clustering: Minimum overlap of two lines for detection of line cluster
static double	<code>FD_RATIO_OF_LINE_LENGTH</code>	Hough Transform line: ratio of rows for minimum line length for Hough Transform
static int	<code>FD_TH_LINE_DETECTION</code>	Hough Transform Line: TH for detecting lines in the Hough Space
static double	<code>FD_TH_SIDES</code>	Rectangle Detection: ratio of white pixel to pixel per side to support a rectangle
static int	<code>GD_MAX_CHAR_WIDTH</code>	Grid Detector: maximum char width
static int	<code>GD_MAX_LINE_HEIGHT</code>	Grid Detection: maximum line height
static int	<code>GD_MIN_CHAR_WIDTH</code>	Grid Detector: minimum char height
static int	<code>GD_MIN_LINE_HEIGHT</code>	Grid Detector: minimum line height
static double	<code>GD_RATIO_PADDING_HOR</code>	Grid Detector: ratio of horizontal grid field padding
static double	<code>GD_RATIO_PADDING_VER</code>	Grid Detector: ratio of vertical grid field padding
static double[]	<code>GD_ROI_PADDING</code>	Grid Detector: padding for a ROI - {top, left, bottom, right}
static int	<code>OV_CHANGE_NEIGHBOURHOOD</code>	Output: the change value is set to zero, in case no other change could be detected within this frame neighbourhood
static double	<code>SD_MIN_ROI_OVERLAP</code>	Layout Matching: minimum overlap for two ROIs to be considered similar
static int	<code>SD_SAMPLING_FREQ</code>	Layout Matching: sampling frequency for routine checks for new layout
static java.lang.String[]	<code>SUPPORTED_IMG_FORMATS</code>	General: all supported image formats
static java.util.List<int[]>	<code>taskbarPerDisplay</code>	General: List of all taskbar locations per display
static int	<code>TC_CHAR_LARGER_CHANGE</code>	Text Change: minimum amount of altered grid fields to be considered a larger textual change
static int	<code>TC_TEXTBOX_REAPPEARANCE</code>	Text Change: minimum number of frames after which a text box must be disappeared

### Constructor Summary

**Constructors**

Constructor	Description
<code>Config()</code>	

**Method Summary**

- All Methods
- Static Methods
- Concrete Methods

Modifier and Type	Method	Description
static void	<code>setDimensions(int[] dimensions)</code>	This method sets all lengths relative to the display resolution.
static void	<code>setTaskbarLocation(int indexFrame)</code>	Sets the taskbar location for a specific frame.

**Methods inherited from class java.lang.Object**

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

**Field Details**

**FD\_RATIO\_OF\_LINE\_LENGTH**

`public static final double FD_RATIO_OF_LINE_LENGTH`  
 Hough Transform line: ratio of rows for minimum line length for Hough Transform  
**See Also:**  
[Constant Field Values](#)

**FD\_MAX\_LINE\_GAP**

`public static final int FD_MAX_LINE_GAP`  
 Hough Transform Line: maximum pixel gap for Hough Transform  
**See Also:**  
[Constant Field Values](#)

**FD\_TH\_LINE\_DETECTION**

`public static final int FD_TH_LINE_DETECTION`  
 Hough Transform Line: TH for detecting lines in the Hough Space  
**See Also:**  
[Constant Field Values](#)

**FD\_GEOMETRIC\_DISTANCE**

`public static final int FD_GEOMETRIC_DISTANCE`  
 Line Clustering: neighborhood in pixel for detection of line cluster  
**See Also:**  
[Constant Field Values](#)

**FD\_MINIMUM\_OVERLAP**

```
public static final double FD_MINIMUM_OVERLAP
```

Line Clustering: Minimum overlap of two lines for detection of line cluster

**See Also:**

Constant Field Values

**FD\_TH\_SIDES**

```
public static final double FD_TH_SIDES
```

Rectangle Detection: ratio of white pixel to pixel per side to support a rectangle

**See Also:**

Constant Field Values

**FD\_MIN\_ROI\_SIZE**

```
public static double FD_MIN_ROI_SIZE
```

Rectangle Detection: minimum ratio of rectangle area to total screen for possible ROIs

**FD\_MAX\_ROI\_SIZE**

```
public static double FD_MAX_ROI_SIZE
```

Rectangle Detection: maximum ratio of rectangle area to total screen for possible ROIs

**FD\_LAYOUT\_MATCH\_SIDE\_TH**

```
public static final double FD_LAYOUT_MATCH_SIDE_TH
```

Layout Matching: ratio of white pixel to side length for layout matching

**See Also:**

Constant Field Values

**SD\_MIN\_ROI\_OVERLAP**

```
public static final double SD_MIN_ROI_OVERLAP
```

Layout Matching: minimum overlap for two ROIs to be considered similar

**See Also:**

Constant Field Values

**SD\_SAMPLING\_FREQ**

```
public static final int SD_SAMPLING_FREQ
```

Layout Matching: sampling frequency for routine checks for new layout

**See Also:**

Constant Field Values

**GD\_MAX\_LINE\_HEIGHT**

```
public static final int GD_MAX_LINE_HEIGHT
```

Grid Detection: maximum line height

**See Also:**

Constant Field Values

**GD\_MIN\_LINE\_HEIGHT**

```
public static final int GD_MIN_LINE_HEIGHT
```

Grid Detector: minimum line height

**See Also:**

[Constant Field Values](#)

**GD\_MAX\_CHAR\_WIDTH**

```
public static final int GD_MAX_CHAR_WIDTH
```

Grid Detector: maximum char width

**See Also:**

[Constant Field Values](#)

**GD\_MIN\_CHAR\_WIDTH**

```
public static final int GD_MIN_CHAR_WIDTH
```

Grid Detector: minimum char height

**See Also:**

[Constant Field Values](#)

**GD\_RATIO\_PADDING\_HOR**

```
public static final double GD_RATIO_PADDING_HOR
```

Grid Detector: ratio of horizontal grid field padding

**See Also:**

[Constant Field Values](#)

**GD\_RATIO\_PADDING\_VER**

```
public static final double GD_RATIO_PADDING_VER
```

Grid Detector: ratio of vertical grid field padding

**See Also:**

[Constant Field Values](#)

**GD\_ROI\_PADDING**

```
public static final double[] GD_ROI_PADDING
```

Grid Detector: padding for a ROI - {top, left, bottom, right}

**TC\_CHAR\_LARGER\_CHANGE**

```
public static final int TC_CHAR_LARGER_CHANGE
```

Text Change: minimum amount of altered grid fields to be considered a larger textual change

**See Also:**

[Constant Field Values](#)

**TC\_TEXTBOX\_REAPPEARANCE**

```
public static final int TC_TEXTBOX_REAPPEARANCE
```

Text Change: minimum number of frames after which a text box must be disappeared



**See Also:**

Constant Field Values

**OV\_CHANGE\_NEIGHBOURHOOD**

```
public static final int OV_CHANGE_NEIGHBOURHOOD
```

Output: the change value is set to zero, in case no other change could be detected within this frame neighbourhood

**See Also:**

Constant Field Values

**dimensions**

```
public static int[] dimensions
```

General: resolution of display

**taskbarPerDisplay**

```
public static java.util.List<int[]> taskbarPerDisplay
```

General: List of all taskbar locations per display

**SUPPORTED\_IMG\_FORMATS**

```
public static final java.lang.String[] SUPPORTED_IMG_FORMATS
```

General: all supported image formats

**Constructor Details****Config**

```
public Config()
```

**Method Details****setDimensions**

```
public static void setDimensions(int[] dimensions)
```

This method sets all lengths relative to the display resolution.

**Parameters:**

dimensions - resolution of current display

**setTaskbarLocation**

```
public static void setTaskbarLocation(int indexFrame)
```

Sets the taskbar location for a specific frame.

**Parameters:**

indexFrame - frame index

Package com.unileoben.videoanalysis

## Class DisplayTBDetector

java.lang.Object  
com.unileoben.videoanalysis.DisplayTBDetector

```
public class DisplayTBDetector
extends java.lang.Object
```

This class contains all methods which are used to evaluate the resolutions of the images within the input directory. Furthermore, it contains methods for detecting regions at the bottom of each individual screen, in which no or few changes occur over the duration of the recording.

### Constructor Summary

#### Constructors

Constructor	Description
<code>DisplayTBDetector(ImageReader imageReader)</code>	Constructs a DisplayTBDetector.

### Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
java.util.List<int[]>	<code>getDisplaySequences()</code>	Getter for a list of all display sequences.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### DisplayTBDetector

```
public DisplayTBDetector(ImageReader imageReader)
```

Constructs a DisplayTBDetector. When an instance of this class is created, all images within a specified input directory are inspected. They will be grouped according to their resolution and analyzed with respect to low variance regions at the bottom of the screen.

#### Parameters:

imageReader - ImageReader for this project

### Method Details

#### getDisplaySequences

```
public java.util.List<int[]> getDisplaySequences()
```

Getter for a list of all display sequences.

#### Returns:

list of start and end index of all detected display sequences

Package com.unileoben.videoanalysis

## Class FeatureDetector

java.lang.Object  
com.unileoben.videoanalysis.FeatureDetector

```
public class FeatureDetector
extends java.lang.Object
```

This class contains all methods of feature detection methods used within this project.

### Constructor Summary

#### Constructors

Constructor	Description
FeatureDetector()	Constructs a FeatureDetector.

### Method Summary

All Methods	Instance Methods	Concrete Methods	
Modifier and Type	Method		Description
org.opencv.core.Mat[]	<code>borderParallellineDetection(org.opencv.core.Mat[] inputImages, int minLength, int th, double theta, int maxGap)</code>		Performs a probabilistic Hough Transform to get all border parallel lines.
java.util.List<BorderParallelRect>	<code>getAllPossibleROIs(org.opencv.core.Mat frameGrayscale)</code>		Results in a list of possible regions of interest for a single frame.
boolean	<code>matchesLayout(org.opencv.core.Mat frameGrayscale, java.util.List&lt;BorderParallelRect&gt; topmostROIs)</code>		Decides, if a given layout of topmost ROIs fits the current frame.
double	<code>ratioOfPointsPerLine(boolean horizontal, int bound1, int bound2, int fixedPos, org.opencv.core.Mat img, int minLength, int pixelValueTh)</code>		Calculates the ratio of white pixels line length of border parallel line.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### FeatureDetector

```
public FeatureDetector()
Constructs a FeatureDetector.
```

### Method Details

#### getAllPossibleROIs

```
public java.util.List<BorderParallelRect> getAllPossibleROIs(org.opencv.core.Mat frameGrayscale)
```

Results in a list of possible regions of interest for a single frame.

Parameters:

frameGrayscale - frame

**Returns:**

list of all possible ROIs

### ratioOfPointsPerLine

```
public double ratioOfPointsPerLine(boolean horizontal,
                                   int bound1,
                                   int bound2,
                                   int fixedPos,
                                   org.opencv.core.Mat img,
                                   int minLength,
                                   int pixelValueTh)
```

Calculates the ratio of white pixels line length of border parallel line.

**Parameters:**

horizontal - true if the line is horizontal

bound1 - start value for line

bound2 - end value for line

fixedPos - constant value

img - input image

minLength - minimum length for line

pixelValueTh - threshold for counted pixel

**Returns:**

ratio of pixel values above th to the total length

### matchesLayout

```
public boolean matchesLayout(org.opencv.core.Mat frameGrayscale,
                             java.util.List<BorderParallelRect> topmostROIS)
```

Decides, if a given layout of topmost ROIs fits the current frame.

**Parameters:**

frameGrayscale - frame

topmostROIS - list of the topmost ROIs

**Returns:**

true, if layout can be detected in the frame

### borderParallelLineDetection

```
public org.opencv.core.Mat[] borderParallelLineDetection(org.opencv.core.Mat[] inputImages,
                                                         int minLength,
                                                         int th,
                                                         double theta,
                                                         int maxGap)
```

Performs a probabilistic Hough Transform to get all border parallel lines.

**Parameters:**

inputImages - image where the method should be applied

minLength - minimal line length of detected lines

th - accumulator threshold parameter. Only those lines are returned that get enough votes

theta - angle resolution of the accumulator in radians.

maxGap - maximal allowed gap for a detected line

**Returns:**

output vector of lines

Package com.unileoben.videoanalysis

## Class GridDetector

java.lang.Object  
com.unileoben.videoanalysis.GridDetector

```
public class GridDetector
extends java.lang.Object
```

This class contains all methods for constructing and evaluating a character grid for a specific ROI.

### Constructor Summary

#### Constructors

Constructor	Description
GridDetector()	Creates a GridDetector.

### Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
org.opencv.core.Mat	binarizeRoi(org.opencv.core.Mat roi)	Binarizes a ROI using an adaptive threshold.
int[]	detectOffset(org.opencv.core.Mat roi, int[] periods)	Detect the offset values for a specific ROI and period values.
int[]	detectOffset(org.opencv.core.Mat roi, int[] periods, int[] compareOffset)	Detect the offset values for a specific ROI and period values.
int[]	detectPeriod(org.opencv.core.Mat roi)	Detect a period for horizontal and vertical direction for a single ROI.
boolean	isTextEditor(java.util.List<org.opencv.core.Mat> rois, int[] period, java.util.List<int[]> offsets)	Decides for a given sequence of ROIs and the respective grid values whether is a text editor.
org.opencv.core.Mat	removeLongLines(org.opencv.core.Mat binarizedRoi, int maxLength, boolean blackLines)	Removes long lines by applying a Probabilistic Hough Transform.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### GridDetector

```
public GridDetector()
Creates a GridDetector.
```

### Method Details

#### binarizeRoi

```
public org.opencv.core.Mat binarizeRoi(org.opencv.core.Mat roi)
Binarizes a ROI using an adaptive threshold.
```

**Parameters:**

roi - ROI

**Returns:**

binarized ROI

**removeLongLines**

```
public org.opencv.core.Mat removeLongLines(org.opencv.core.Mat binarizedRoi,
                                           int maxLength,
                                           boolean blackLines)
```

Removes long lines by applying a Probabilistic Hough Transform.

**Parameters:**

binarizedRoi - binarized ROI

maxLength - maximum length of lines

blackLines - specifies the color of the foreground (true, black - false, white)

**Returns:**

binarized ROI with long lines removed

**detectPeriod**

```
public int[] detectPeriod(org.opencv.core.Mat roi)
```

Detect a period for horizontal and vertical direction for a single ROI. In case no values could be found, the respective value will be -1.

**Parameters:**

roi - ROI

**Returns:**

period values {vertical, horizontal}

**detectOffset**

```
public int[] detectOffset(org.opencv.core.Mat roi,
                          int[] periods,
                          int[] compareOffset)
```

Detect the offset values for a specific ROI and period values. The results are compared with a compare offset. The compare offset will be accepted, in case it results in a similar value.

**Parameters:**

roi - ROI

periods - period values

compareOffset - compare offset

**Returns:**

grid offset {vertical, horizontal}

**detectOffset**

```
public int[] detectOffset(org.opencv.core.Mat roi,
                          int[] periods)
```

Detect the offset values for a specific ROI and period values.

**Parameters:**

roi - ROI

periods - period values

**Returns:**

grid offset {vertical, horizontal}

**isTextEditor**

```
public boolean isTextEditor(java.util.List<org.opencv.core.Mat> rois,  
                           int[] period,  
                           java.util.List<int[]> offsets)
```

Decides for a given sequence of ROIs and the respective grid values whether is a text editor.

**Parameters:**

rois - list of binarized ROIs from that sequence

period - period values for that ROI

offsets - list of offset values for that sequence

**Returns:**

decision whether the ROI is a text editor

Package com.unileoben.videoanalysis

## Class ScreencastAnalyzer

java.lang.Object  
com.unileoben.videoanalysis.ScreencastAnalyzer

```
public class ScreencastAnalyzer
extends java.lang.Object
```

This class is the central component of this project. Given a path to the directory in which the frames of a screencast are stored, this class coordinates the workflow of this application.

### Constructor Summary

#### Constructors

Constructor	Description
<code>ScreencastAnalyzer(java.lang.String dirInput, java.lang.String dirResults, java.lang.String screencastTitle)</code>	Creates a new ScreencastAnalyzer.

### Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description
void	<code>analyze()</code>	Analyzes the specified screencast.
void	<code>analyze(java.lang.String evalFile)</code>	Analyzes the specified screencast and evaluates it with respect to precision and recall using the specified evaluation data.
void	<code>tests()</code>	

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### ScreencastAnalyzer

```
public ScreencastAnalyzer(java.lang.String dirInput,
                          java.lang.String dirResults,
                          java.lang.String screencastTitle)
```

Creates a new ScreencastAnalyzer. All instances of required classes are created within the constructor.

#### Parameters:

`dirInput` - path to directory in which input frames are stored  
`dirResults` - path to directors in which output will be saved  
`screencastTitle` - title of the screencast

### Method Details

#### analyze

```
public void analyze(java.lang.String evalFile)
```

Analyzes the specified screencast and evaluates it with respect to precision and recall using the specified evaluation data. The results are saved within the specified directory.



**Parameters:**

evalFile - file path to evaluation file

**analyze**

```
public void analyze()
```

Analyzes the specified screencast. The results are saved within the specified directory.

**tests**

```
public void tests()
```

Package com.unileoben.videoanalysis

## Class SequenceAnalyzer

java.lang.Object  
com.unileoben.videoanalysis.SequenceAnalyzer

```
public class SequenceAnalyzer
extends java.lang.Object
```

This class contains all methods for analyzing the contents within a sequence.

### Constructor Summary

#### Constructors

Constructor	Description
<code>SequenceAnalyzer(ImageReader imageReader, java.lang.String dirResults)</code>	Constructs a SequenceAnalyzer.

### Method Summary

All Methods			Instance Methods			Concrete Methods		
Modifier and Type	Method		Modifier and Type	Method		Modifier and Type	Method	
void	<code>analyzeChanges(int[] sequence, OutputValues values)</code>							Calculates the typing rates for all relevant ROIs within a specified sequence and hands them over to an instance of OutputValues.
void	<code>setGridLayouts(int[] sequence, java.util.List&lt;BorderParallelRect&gt; structure)</code>							Defines the ROIs and the corresponding grid values for a list of possible ROIs.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### SequenceAnalyzer

```
public SequenceAnalyzer(ImageReader imageReader,
                        java.lang.String dirResults)
```

Constructs a SequenceAnalyzer.

#### Parameters:

imageReader - ImageReader for obtaining the frames within a screencast

dirResults - directory in which the results are stored

### Method Details

#### setGridLayouts

```
public void setGridLayouts(int[] sequence,
                          java.util.List<BorderParallelRect> structure)
```

Defines the ROIs and the corresponding grid values for a list of possible ROIs.

#### Parameters:

sequence - frame indices of the respective sequence

structure - list of all possible ROIs

### analyzeChanges

```
public void analyzeChanges(int[] sequence,  
                           OutputValues values)
```

Calculates the typing rates for all relevant ROIs within a specified sequence and hands them over to an instance of OutputValues.

**Parameters:**

`sequence` - frame indices of the respective sequence

`values` - instance of OutputValues, which consolidates all typing rates within this screencast

Package com.unileoben.videoanalysis

## Class SequenceDetector

java.lang.Object  
com.unileoben.videoanalysis.SequenceDetector

```
public class SequenceDetector
extends java.lang.Object
```

This class contains all methods for detecting sequences within a screencast with a similar layout of ROIs.

### Constructor Summary

#### Constructors

Constructor	Description
<code>SequenceDetector(ImageReader imageReader, java.lang.String dirResults)</code>	Creates an instance of SequenceDetector for a specified screencast.

### Method Summary

All Methods	Instance Methods	Concrete Methods	
Modifier and Type	Method		Description
void	<code>generateAllSequences(int startIndex, int endIndex)</code>		Generates all sequences between two frame indices.
java.util.List<int[]>	<code>getAllDetectedSequences()</code>		Getter for all detected Sequences.
java.util.List<BorderParallelRect>	<code>getROIStructure(int index)</code>		Returns the ROI structure of a given index.
java.util.List<BorderParallelRect>	<code>getROIStructure(int[] sequence)</code>		Returns the ROI structure of a given sequence.
int[]	<code>getSequenceIndices(int startIndex, int maxEndIndex)</code>		Generates the next sequence for a specified start frame index.
void	<code>resetAllSequences()</code>		Resets all detected sequences.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### SequenceDetector

```
public SequenceDetector(ImageReader imageReader,
                        java.lang.String dirResults)
```

Creates an instance of SequenceDetector for a specified screencast.

#### Parameters:

imageReader - ImageReader for obtaining the frames within a screencast

dirResults - directory in which the results are stored

### Method Details

#### resetAllSequences

```
public void resetAllSequences()
```

Resets all detected sequences.

#### getAllDetectedSequences

```
public java.util.List<int[]> getAllDetectedSequences()
```

Getter for all detected Sequences.

**Returns:**

list of all sequences as frame indices

#### generateAllSequences

```
public void generateAllSequences(int startIndex,  
                                int endIndex)
```

Generates all sequences between two frame indices.

**Parameters:**

startIndex - start frame index

endIndex - end frame index

#### getSequenceIndices

```
public int[] getSequenceIndices(int startIndex,  
                                int maxEndIndex)
```

Generates the next sequence for a specified start frame index.

**Parameters:**

startIndex - start frame index

maxEndIndex - max end frame index

**Returns:**

sequence {start index, end index}

#### getROIStructure

```
public java.util.List<BorderParallelRect> getROIStructure(int[] sequence)
```

Returns the ROI structure of a given sequence.

**Parameters:**

sequence - frame indices of the respective sequence

**Returns:**

list of possible ROIs

#### getROIStructure

```
public java.util.List<BorderParallelRect> getROIStructure(int index)
```

Returns the ROI structure of a given index.

**Parameters:**

index - frame index

**Returns:**

list of possible ROIs

Package com.unileoben.videoanalysis

## Class TextChangeDetector

java.lang.Object  
 com.unileoben.videoanalysis.TextChangeDetector

```
public class TextChangeDetector
extends java.lang.Object
```

This class contains all methods for detecting the typing rate for a specified ROI.

### Constructor Summary

#### Constructors

Constructor	Description
<code>TextChangeDetector(GridDetector gridDetector)</code>	Constructs a TextChangeDetector.

### Method Summary

- All Methods
- Instance Methods
- Concrete Methods

Modifier and Type	Method	Description
void	<code>evaluateTextualChangeOfROI(java.util.List&lt;org.opencv.core.Mat&gt; binaryFrames, java.util.List&lt;org.opencv.core.Mat[]&gt; changeImgs, int[] period, java.util.List&lt;int[]&gt; offset, int[] sequence, OutputValues values)</code>	Evaluates the textual changes of a specified ROI within the entire sequence.
int[]	<code>getRegionOfChange(java.util.List&lt;org.opencv.core.Mat[]&gt; changeImgs)</code>	

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### TextChangeDetector

```
public TextChangeDetector(GridDetector gridDetector)
```

Constructs a TextChangeDetector.

#### Parameters:

gridDetector - GridDetector for supporting methods

### Method Details

#### evaluateTextualChangeOfROI

```
public void evaluateTextualChangeOfROI(java.util.List<org.opencv.core.Mat> binaryFrames,
                                       java.util.List<org.opencv.core.Mat[]> changeImgs,
                                       int[] period,
                                       java.util.List<int[]> offset,
                                       int[] sequence,
                                       OutputValues values)
```

Evaluates the textual changes of a specified ROI within the entire sequence.

#### Parameters:

binaryFrames - list of ROI as binarized images

`changeImgs` - list of ROI as change images

`period` - period values for this ROI

`offset` - list of offset values for this ROI

`sequence` - frame indices of the respective sequence

`values` - OutputValues for the respective screencast

#### **getRegionOfChange**

```
public int[] getRegionOfChange(java.util.List<org.opencv.core.Mat[]> changeImgs)
```

Package com.unileoben.videoanalysis.content

## Class BorderParallelRect

java.lang.Object  
 com.unileoben.videoanalysis.content.BorderParallelRect

### All Implemented Interfaces:

java.lang.Comparable

```
public class BorderParallelRect
extends java.lang.Object
implements java.lang.Comparable
```

This class represents a rectangle with border-parallel lines.

### Constructor Summary

#### Constructors

##### Constructor

**BorderParallelRect**(int upperHorValue, int lowerHorValue, int leftVerValue, int rightVerValue)

##### Description

Constructs a BorderParallelRect using the constant values per line.

### Method Summary

All Methods	Instance Methods	Concrete Methods	
Modifier and Type	Method	Description	
void	<code>addNestedWindow(BorderParallelRect r)</code>	Adds a new BorderParallelRect to the set of all directly nested BorderParallelRects.	
int	<code>compareTo(java.lang.Object o)</code>	Compares two BorderParallelRect for ordering.	
boolean	<code>equals(java.lang.Object o)</code>	Compares two BorderParallelRect for equality.	
int	<code>getArea()</code>	Getter of area.	
org.opencv.core.Point	<code>getLeftBottomCorner()</code>	Getter of left bottom corner point.	
VerticalLine	<code>getLeftLine()</code>	Getter of left line.	
org.opencv.core.Point	<code>getLeftUpperCorner()</code>	Getter of left upper corner point.	
java.util.List<BorderParallelRect>	<code>getNestedRects()</code>	Getter of a list of all directly nested BorderParallelRect.	
org.opencv.core.Point	<code>getRightBottomCorner()</code>	Getter of roght bottom corner point.	
org.opencv.core.Point	<code>getRightUpperCorner()</code>	Getter of right upper corner point.	
HorizontalLine	<code>getUpperLine()</code>	Getter of upper horizontal line.	
int	<code>hashCode()</code>	Returns a hash code value for this object.	
double	<code>overlapWith(BorderParallelRect compareBorderParallelRect)</code>	Determines the percentage of a second BorderParallelRectangle which overlaps with this BorderParallelRectangle.	
void	<code>resetNestedRects()</code>	Sets the directly nested BorderParallelRects to an empty list.	
java.lang.String	<code>toString()</code>	Converts this BorderParallelRect object to a String.	



**Methods inherited from class java.lang.Object**

clone, finalize, getClass, notify, notifyAll, wait, wait, wait

**Constructor Details****BorderParallelRect**

```
public BorderParallelRect(int upperHorValue,  
                          int lowerHorValue,  
                          int leftVerValue,  
                          int rightVerValue)
```

Constructs a BorderParallelRect using the constant values per line.

**Parameters:**

upperHorValue - value of upper horizontal border

lowerHorValue - value of lower horizontal border

leftVerValue - value of left vertical border

rightVerValue - value of right vertical border

**Method Details****getLeftUpperCorner**

```
public org.opencv.core.Point getLeftUpperCorner()
```

Getter of left upper corner point.

**Returns:**

left upper corner Point

**getRightUpperCorner**

```
public org.opencv.core.Point getRightUpperCorner()
```

Getter of right upper corner point.

**Returns:**

right upper corner Point

**getLeftBottomCorner**

```
public org.opencv.core.Point getLeftBottomCorner()
```

Getter of left bottom corner point.

**Returns:**

left bottom corner Point

**getRightBottomCorner**

```
public org.opencv.core.Point getRightBottomCorner()
```

Getter of roght bottom corner point.

**Returns:**

right bottom corner Point

**getUpperLine**

```
public HorizontalLine getUpperLine()
```

Getter of upper horizontal line.

**Returns:**  
upper line

### getLeftLine

```
public VerticalLine getLeftLine()
```

Getter of left line.

**Returns:**  
left line

### getArea

```
public int getArea()
```

Getter of area.

**Returns:**  
area

### getNestedRects

```
public java.util.List<BorderParallelRect> getNestedRects()
```

Getter of a list of all directly nested BorderParallelRect.

**Returns:**  
list of all directly nested BorderParallelRect.

### resetNestedRects

```
public void resetNestedRects()
```

Sets the directly nested BorderParallelRects to an empty list.

### addNestedWindow

```
public void addNestedWindow(BorderParallelRect r)
```

Adds a new BorderParallelRect to the set of all directly nested BorderParallelRects.

**Parameters:**  
r - new directly nested BorderParallelRect

### overlapWith

```
public double overlapWith(BorderParallelRect compareBorderParallelRect)
```

Determines the percentage of a second BorderParallelRectangle which overlaps with this BorderParallelRectangle.

**Parameters:**  
compareBorderParallelRect - compare ROI

**Returns:**  
overlap

### compareTo

```
public int compareTo(java.lang.Object o)
```

Compares two BorderParallelRect for ordering.

**Specified by:**

compareTo in interface `java.lang.Comparable`

**Parameters:**

o - `BorderParallelRect` for comparison

**Returns:**

the value 0 if the argument `BorderParallelRect` has the same area as this `BorderParallelRect`; a value less than 0 if the area of this `BorderParallelRect` is smaller than the area of the `BorderParallelRect` argument; and a value greater than 0 if the area of this `BorderParallelRect` is larger than the area of the `BorderParallelRect` argument.

**equals**

```
public boolean equals(java.lang.Object o)
```

Compares two `BorderParallelRect` for equality. The result is true if and only if the argument is not null and is a `BorderParallelRect` has the same geometric position, as this object.

**Overrides:**

equals in class `java.lang.Object`

**Parameters:**

o - the object to compare with

**Returns:**

true if the objects are the same; false otherwise

**hashCode**

```
public int hashCode()
```

Returns a hash code value for this object.

**Overrides:**

hashCode in class `java.lang.Object`

**Returns:**

a hash code value for this object

**toString**

```
public java.lang.String toString()
```

Converts this `BorderParallelRect` object to a `String`.

**Overrides:**

toString in class `java.lang.Object`

**Returns:**

a string representation of this `BorderParallelRect`

**Package** com.unileoben.videoanalysis.content

## Class HorizontalLine

java.lang.Object  
 com.unileoben.videoanalysis.content.Line  
 com.unileoben.videoanalysis.content.HorizontalLine

**All Implemented Interfaces:**

java.lang.Comparable

---

```
public class HorizontalLine
extends Line
```

This class represents a horizontal line.

### Constructor Summary

#### Constructors

Constructor	Description
HorizontalLine(int pxPos1, int pxPos2, int pxVerticalPos)	Constructs a horizontal line.

### Method Summary

- All Methods
- Instance Methods
- Concrete Methods

Modifier and Type	Method	Description
int	distanceBetween(Line line)	Calculates the distance between two HorizontalLines.
int	getPxLeft()	Getter for left restrictive pixel value.
int	getPxRight()	Getter for right restrictive pixel value.
int	getPxVerticalPos()	Getter for vertical pixel position.
int	overlapPX(HorizontalLine line)	Determines the number of pixel which overlap between this and a second line.
double	overlapWith(Line line)	Determines the percentage of a second line which overlaps with this.
void	setPxVerticalPos(int pxVerticalPos)	Setter for vertical pixel position.

#### Methods inherited from class com.unileoben.videoanalysis.content.Line

compareTo, equals, getLength, getPoint1, getPoint2, hashCode, setPoint1, setPoint2

#### Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

**HorizontalLine**

```
public HorizontalLine(int pxPos1,
                    int pxPos2,
                    int pxVerticalPos)
```

Constructs a horizontal line.

**Parameters:**

pxPos1 - one restrictive pixel value (e.g. the left pixel value)

pxPos2 - one restrictive pixel value (e.g. the right pixel value)

pxVerticalPos - the vertical position for this line

**Method Details****getPxVerticalPos**

```
public int getPxVerticalPos()
```

Getter for vertical pixel position.

**Returns:**

vertical pixel position

**setPxVerticalPos**

```
public void setPxVerticalPos(int pxVerticalPos)
```

Setter for vertical pixel position.

**Parameters:**

pxVerticalPos - enw vertical pixel position

**getPxLeft**

```
public int getPxLeft()
```

Getter for left restrictive pixel value.

**Returns:**

left restrictive pixel value

**getPxRight**

```
public int getPxRight()
```

Getter for right restrictive pixel value.

**Returns:**

right restrictive pixel value

**overlapWith**

```
public double overlapWith(Line line)
```

Determines the percentage of a second line which overlaps with this.

**Overrides:**

overlapWith in class Line

**Parameters:**

line - compare line

**Returns:**

the percentage of the second line which overlaps with this

**overlapPX**

```
public int overlapPX(HorizontalLine line)
```

Determines the number of pixel which overlap between this and a second line.

**Parameters:**

line - compare line

**Returns:**

the percentage of the second line which overlaps with this

**distanceBetween**

```
public int distanceBetween(Line line)
```

Calculates the distance between two HorizontalLines.

**Overrides:**

distanceBetween in class Line

**Parameters:**

line - compare line

**Returns:**

pixel between the compare line and this

**Package** com.unileoben.videoanalysis.content

## Class Line

java.lang.Object  
 com.unileoben.videoanalysis.content.Line

**All Implemented Interfaces:**

java.lang.Comparable

**Direct Known Subclasses:**

HorizontalLine, VerticalLine

```
public class Line
extends java.lang.Object
implements java.lang.Comparable
```

This class represents a two-dimensional line.

### Constructor Summary

**Constructors**

Constructor	Description
Line(org.opencv.core.Point point1, org.opencv.core.Point point2)	Constructs a line.

### Method Summary

All Methods	Instance Methods	Concrete Methods
-------------	------------------	------------------

Modifier and Type	Method	Description
int	compareTo (java.lang.Object o)	Compares two Line for ordering.
int	distanceBetween (Line otherLine)	Returns the distance between this Line and another.
boolean	equals(java.lang.Object o)	Compares two Line for equality.
int	getLength()	Getter of line length
org.opencv.core.Point	getPoint1()	Getter of start point of the line.
org.opencv.core.Point	getPoint2()	Getter of end point of the line.
int	hashCode()	Returns a hash code value for this object.
double	overlapWith(Line line)	Determines the percentage of a second BorderParallelRectangle which overlaps with this BorderParallelRectangle.
void	setPoint1(int x, int y)	Setter of start point of the line.
void	setPoint2(int x, int y)	Setter of end point of the line.

#### Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

**Line**

```
public Line(org.opencv.core.Point point1,
            org.opencv.core.Point point2)
```

Constructs a line.

**Parameters:**

point1 - start point of the line

point2 - end point of the line

### Method Details

#### getPoint1

```
public org.opencv.core.Point getPoint1()
```

Getter of start point of the line.

**Returns:**

start point

#### setPoint1

```
public void setPoint1(int x,  
                     int y)
```

Setter of start point of the line.

**Parameters:**

x - new x coordinate

y - new y coordinate

#### getPoint2

```
public org.opencv.core.Point getPoint2()
```

Getter of end point of the line.

**Returns:**

end point

#### setPoint2

```
public void setPoint2(int x,  
                     int y)
```

Setter of end point of the line.

**Parameters:**

x - new x coordinate

y - new y coordinate

#### getLength

```
public int getLength()
```

Getter of line length

**Returns:**

line length

#### distanceBetween

```
public int distanceBetween(Line otherLine)
```

Returns the distance between this Line and another.

**Parameters:**



otherLine - compare line

**Returns:**

the value 0, if the argument Line or this Line are both of type Line.

**overlapWith**

```
public double overlapWith(Line line)
```

Determines the percentage of a second BorderParallelRectangle which overlaps with this BorderParallelRectangle.

**Parameters:**

line - compare line

**Returns:**

the value 0, if the argument Line or this Line are both of type Line.

**compareTo**

```
public int compareTo(java.lang.Object o)
```

Compares two Line for ordering.

**Specified by:**

compareTo in interface java.lang.Comparable

**Parameters:**

o - Line for comparison

**Returns:**

the value 0 if the argument Line has the same length as this Line; a value less than 0 if the length of this Line is smaller than the length of the Line argument; and a value greater than 0 if the length of this Line is larger than the length of the Line argument.

**equals**

```
public boolean equals(java.lang.Object o)
```

Compares two Line for equality. The result is true if and only if the argument is not null and is a Line has the same geometric position, as this object.

**Overrides:**

equals in class java.lang.Object

**Parameters:**

o - the object to compare with

**Returns:**

true if the objects are the same; false otherwise

**hashCode**

```
public int hashCode()
```

Returns a hash code value for this object.

**Overrides:**

hashCode in class java.lang.Object

**Returns:**

a hash code value for this object

**Package** com.unileoben.videoanalysis.content

## Class VerticalLine

java.lang.Object  
 com.unileoben.videoanalysis.content.Line  
 com.unileoben.videoanalysis.content.VerticalLine

**All Implemented Interfaces:**

java.lang.Comparable

---

```
public class VerticalLine
extends Line
```

This class represents a vertical line.

### Constructor Summary

#### Constructors

Constructor	Description
VerticalLine(int pxPos1, int pxPos2, int pxHorizontalPos)	Constructs a vertical line.

### Method Summary

**All Methods**   **Instance Methods**   **Concrete Methods**

Modifier and Type	Method	Description
int	distanceBetween(Line line)	Calculates the distance between two lines.
int	getPxHorizontalPos()	Getter for horizontal pixel position.
int	getPxLower()	Getter for lower restrictive pixel value.
int	getPxUpper()	Getter for upper restrictive pixel value.
int	overlapPX(VerticalLine line)	Determines the number of pixel which overlap between this and a second line.
double	overlapWith(Line line)	Determines the percentage of a second line which overlaps with this.
void	setPxHorizontalPos(int pxHorizontalPos)	Setter for horizontal pixel position

#### Methods inherited from class com.unileoben.videoanalysis.content.Line

compareTo, equals, getLength, getPoint1, getPoint2, hashCode, setPoint1, setPoint2

#### Methods inherited from class java.lang.Object

clone, finalize, getClass, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

**VerticalLine**

```
public VerticalLine(int pxPos1,
                   int pxPos2,
                   int pxHorizontalPos)
```

Constructs a vertical line.

**Parameters:**

pxPos1 - one restrictive pixel value (e.g. the upper pixel value)

pxPos2 - one restrictive pixel value (e.g. the lower pixel value)

pxHorizontalPos - the horizontal position for this line

**Method Details****getPxHorizontalPos**

```
public int getPxHorizontalPos()
```

Getter for horizontal pixel position.

**Returns:**

horizontal pixel position

**setPxHorizontalPos**

```
public void setPxHorizontalPos(int pxHorizontalPos)
```

Setter for horizontal pixel position

**Parameters:**

pxHorizontalPos - new horizontal pixel position

**getPxUpper**

```
public int getPxUpper()
```

Getter for upper restrictive pixel value.

**Returns:**

upper restrictive pixel value

**getPxLower**

```
public int getPxLower()
```

Getter for lower restrictive pixel value.

**Returns:**

lower restrictive pixel value

**overlapWith**

```
public double overlapWith(Line line)
```

Determines the percentage of a second line which overlaps with this.

**Overrides:**

overlapWith in class Line

**Parameters:**

line - compare line

**Returns:**

the percentage of the second line which overlaps with this

**overlapPX**

```
public int overlapPX(VerticalLine line)
```

Determines the number of pixel which overlap between this and a second line.

**Parameters:**

line - compare line

**Returns:**

the percentage of the second line which overlaps with this

**distanceBetween**

```
public int distanceBetween(Line line)
```

Calculates the distance between two lines.

**Overrides:**

distanceBetween in class Line

**Parameters:**

line - compare line

**Returns:**

pixel between the compare line and this

Package com.unileoben.videoanalysis.input

## Class ImageReader

java.lang.Object  
com.unileoben.videoanalysis.input.ImageReader

```
public class ImageReader
extends java.lang.Object
```

This class provides the image data throughout this application.

### Constructor Summary

#### Constructors

Constructor	Description
<code>ImageReader(java.lang.String frameDirectory)</code>	Constructs an ImageReader.

### Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
org.opencv.core.Mat	<code>getGrayscaleFrame(int indexFrame)</code>	Provides a specified grayscale image.
java.lang.String[]	<code>getImageNames()</code>	Getter for an Array of image names.
java.lang.String	<code>getNameOf(int indexFrame)</code>	Provides the file name of a specified frame number.
int	<code>numberFrames()</code>	Provides the number of images within the specified directory.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### ImageReader

```
public ImageReader(java.lang.String frameDirectory)
```

Constructs an ImageReader.

#### Parameters:

frameDirectory - directory to where the frames are stored

### Method Details

#### getImageNames

```
public java.lang.String[] getImageNames()
```

Getter for an Array of image names.

#### Returns:

Array of all image names

#### getNameOf

```
public java.lang.String getNameOf(int indexFrame)
```

Provides the file name of a specified frame number.

**Parameters:**

indexFrame - frame index

**Returns:**

corresponding file name

### getGrayscaleFrame

```
public org.opencv.core.Mat getGrayscaleFrame(int indexFrame)
```

Provides a specified grayscale image.

**Parameters:**

indexFrame - frame number of image

**Returns:**

Mat as grayscale

### numberFrames

```
public int numberFrames()
```

Provides the number of images within the specified directory.

**Returns:**

number of images

Package com.unileoben.videoanalysis.output

## Class OutputValues

java.lang.Object  
com.unileoben.videoanalysis.output.OutputValues

```
public class OutputValues
extends java.lang.Object
```

This class contains of all methods for handling typing rates.

### Constructor Summary

#### Constructors

Constructor	Description
<code>OutputValues(int frameNo, java.lang.String resultDir)</code>	Constructs OutputValues, where all typing rates are set to 0.

### Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description
void	<code>addShiftAtBoundsValues(int boundChangeValue, int frameNo)</code>	Adds a textual change, which was detected at the border of a ROI and had been caused by a shift, for the specified frame index in case the absolute new value is higher the the current value.
void	<code>addTextualChangeValues(int[] changeValues, int startFrameNo)</code>	Adds a consecutive textual changes for specified frame indices in case the absolute new value is higher the the current value.
void	<code>addTextualChangeValues(int changeValue, int frameNo)</code>	Adds a textual change for specified frame index in case the absolute new value is higher the the current value.
void	<code>calculatePrecisionAndRecall(java.lang.String evalFile)</code>	Calculates precision and recall values for the current change values and a specified evaluation file and prints them as a console log.
void	<code>generateOutput(java.lang.String name)</code>	Generates a CSV file for the current textual changes.
void	<code>generateOutputInclShiftAtBounds(java.lang.String name)</code>	Generates a CSV file for the current textual changes including textual changes for shifts at the border of a ROI.
int[]	<code>getChangesWithBounds()</code>	Getter of the current textual changes including textual changes for shifts at the border of a ROI.
int[]	<code>getTextualChanges()</code>	Getter of the current textual changes.

#### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

### Constructor Details

#### OutputValues

```
public OutputValues(int frameNo,  
                   java.lang.String resultDir)
```

Constructs OutputValues, where all typing rates are set to 0.

**Parameters:**

frameNo - number of frames within the screencast

resultDir - path to the directory for which the resulting CSV file should be stored

### Method Details

#### getTextualChanges

```
public int[] getTextualChanges()
```

Getter of the current textual changes.

**Returns:**

textual changes

#### getChangesWithBounds

```
public int[] getChangesWithBounds()
```

Getter of the current textual changes including textual changes for shifts at the border of a ROI.

**Returns:**

textual changes

#### generateOutput

```
public void generateOutput(java.lang.String name)
```

Generates a CSV file for the current textual changes.

**Parameters:**

name - file name for the output file

#### generateOutputInclShiftAtBounds

```
public void generateOutputInclShiftAtBounds(java.lang.String name)
```

Generates a CSV file for the current textual changes including textual changes for shifts at the border of a ROI.

**Parameters:**

name - file name for the output file

#### addTextualChangeValues

```
public void addTextualChangeValues(int changeValue,  
                                   int frameNo)
```

Adds a textual change for specified frame index in case the absolute new value is higher than the current value.

**Parameters:**

changeValue - new change value

frameNo - frame index



**addShiftAtBoundsValues**

```
public void addShiftAtBoundsValues(int boundChangeValue,  
                                  int frameNo)
```

Adds a textual change, which was detected at the border of a ROI and had been caused by a shift, for the specified frame index in case the absolute new value is higher than the current value.

**Parameters:**

boundChangeValue - new change value

frameNo - frame index

**addTextualChangeValues**

```
public void addTextualChangeValues(int[] changeValues,  
                                   int startFrameNo)
```

Adds a consecutive textual changes for specified frame indices in case the absolute new value is higher than the current value.

**Parameters:**

changeValues - Array of change values

startFrameNo - start frame index

**calculatePrecisionAndRecall**

```
public void calculatePrecisionAndRecall(java.lang.String evalFile)
```

Calculates precision and recall values for the current change values and a specified evaluation file and prints them as a console log.

**Parameters:**

evalFile - file path to an evaluation file