



^b
**UNIVERSITÄT
BERN**

EggShell

A workbench for modeling scientific communities

Bachelor Thesis

Dominik Seliner

from

Biel BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

26. August 2016

Prof. Dr. Oscar Nierstrasz

Leonel Merino

Software Composition Group

Institut für Informatik

University of Bern, Switzerland

Abstract

The collaboration in a scientific community can be analysed through the publication record of its members. The analysis of the metadata (*e.g.*, title and authors) of those publications can help researchers to identify groups of collaboration, their evolution, and key authors. However, the criteria for collecting the papers of some communities might exceed the expressiveness offered by public databases and search engines available. Hence, the data has to be retrieved from the papers' files themselves.

Usually, scientific papers are available in unstructured file formats for which automatic extraction of data poses a challenge. To model the metadata of a community users have to define a pipeline. In it, each step contributes to the accuracy of the extracted data. The main challenge is to identify to which type of field of the document a piece of text corresponds.

Previous research proposed heuristics to identify certain fields like the title and authors from papers' files by analyzing their layout. The performance of such heuristics might vary across papers that use different layouts. Hence, ensuring the accuracy of a given heuristic is a challenging problem. Small improvements in a heuristic that tackles a popular layout can make a high impact on its overall performance. However, identifying popular layouts and evaluating the impact of improvements can be a laborious task.

Visualization offers techniques that fit the analysis of such multivariate data. Through visualization, a developer who is implementing a heuristic for data extraction can obtain an overview of how it performs and find hotspots that can lead to improvements that impact the overall efficacy.

In this thesis, we propose EggShell, a workbench that incorporates visualization to assess the performance of modeling pipelines for scientific papers in PDF format. We elaborate on examples of how EggShell allows users to define multiple pipelines. Pipelines can then be improved by assessing their output using visualization. We collected a corpus of 300 papers published by SOFTVIS/VISSOFT venues. We used a subset of 100 papers as a learning set to develop the pipelines, and then used the remaining 200 papers to evaluate their performance for modeling collaboration in the community. We observed that our best performing pipeline exhibits an accuracy of 70%.

Contents

1	Introduction	1
2	EggShell	3
2.1	A Modeling Pipeline	3
2.1.1	Data Transformation	3
2.1.2	Semantic Structure Recovery	4
2.1.3	Modeling the Extracted Data	6
2.2	Analysis Examples	7
2.2.1	Text-based Data Extraction	7
2.2.2	XML-based Data Extraction	10
2.2.3	Mutual Modeling Step	15
2.3	Assessment Visualization	16
2.3.1	The Assessment Grid	17
2.3.2	Popup	19
2.3.2.1	Upper Part of the Popup	20
2.3.2.2	Lower Part of the Popup	22
2.3.3	Details-on-Demand	23
2.3.4	Analysis Example	24
2.4	Technical Details	26
2.4.1	Installing EggShell	27
2.4.2	Usage Example	27
3	Conclusion and Future Work	31
4	Anleitung zu wissenschaftlichen Arbeiten	33
4.1	Creating a new pipeline for EggShell	33
4.1.1	Importer	33
4.1.2	Modeler	35

1

Introduction

Proceedings published by an event such as a conference are interesting data for understanding how a community of scientists is composed and evolves over time. Authors collaborate among such communities in publications and these collaborations form clusters which differ in shape and size. To analyze such collaborations, we need a model of the community which contains the metadata of its publications. Although in some cases the metadata like the title and the authors that contributed to a paper are available digitally, extracting the data directly from papers' files give users the flexibility to model communities based on specific criteria (for which databases and search engines are limited). In order to automatically create such models, users have to define a pipeline. Such a pipeline consists of multiple steps that we show in Figure 1.1.

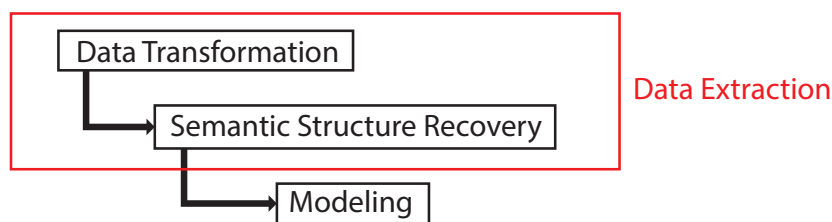


Figure 1.1: A modeling pipeline for the analysis of collaboration in a scientific community.

The first step, *data transformation*, consists of transforming the binary content of a PDF file into a text version. Then the second step, *semantic structure recovery*, aims at

recovering the meaning of each part of that text. These two steps combined are known as *data extraction*. Finally, users can interact with a *model* of the extracted data to analyze collaborations. In Chapter 2 we provide a detailed description of the steps.

Previous research [5] proposed algorithms for recovering the semantic structure of papers in PDF format by analyzing their layout. Such algorithms need upfront knowledge of the location of each type of data (e.g. title, authors) specified by the layout used in a paper. Because these layouts can sometimes be unexpected, the algorithm cannot assure an correct result in all cases. Such an algorithm with an unsure outcome is referred to as a *heuristic*. In effect, the papers of a research community can be published in various venues and in multiple years exposing vastly different layout conventions. Hence, the performance of a modeling pipeline can vary across papers. Improving the performance of a pipeline for a popular layout can make a high impact on its overall performance. However, the process of identifying such popular layouts and assessing the effect of changes to the pipeline on the overall outcome can be a laborious task. Through visualization, users can obtain an overview of how heuristics perform and find hotspots that can lead to improvements that impact the performance for many papers.

We propose EggShell, a workbench for assessing modeling pipelines for scientific communities. In it, users define modeling pipelines and assess their performance using visualization. We elaborate on examples of how users define such a pipeline. Then, we explain how EggShell is used to assess its performance on a data set and how EggShell helps users to improve its overall accuracy.

The contributions of the thesis are (1) a design and implementation of a system that allows users to define and customize modeling pipelines, and (2) a study of the performance of two pipelines that model publications of the VISSOFT/SOFTVIS community.

The remainder of the thesis is structured as follows: Chapter 2 describes the EggShell workbench and is split into section 2.1 to explain how a pipeline is built, section 2.2 to describe the performance of two pipelines that we created, section 2.3 to explain how users interact with EggShell to assess and improve the performance of these pipelines, and section 2.4 to introduce technical aspects of how to install EggShell. Conclusions and future work are presented in Chapter 3. Finally, Chapter 4 describes the steps to create a pipeline such as the ones presented in the example analysis of this thesis.

2

EggShell

In this chapter we describe the steps involved in a modeling pipeline. Then, we present analysis examples of two pipelines. Lastly, we show how visualization offered by EggShell is used to assess and improve the performance of such pipelines.

2.1 A Modeling Pipeline

Modeling pipelines include three main steps as shown in Figure 1.1, they: 1) transform the PDF files into a text representation, 2) recover the semantic structure of each paper from the transformed text (*e.g.*, title and authors), and 3) model the community based on the extracted data. In the following we explain these steps in more detail.

2.1.1 Data Transformation

The Portable Document Format (PDF) is a widely established file format, used for digital documents. In contrast to file formats for data exchange (*e.g.*, CSV), PDF is focused on presenting documents to a human reader. The binary file has the text encoded, then a program that wants to manipulate that text has first to transform it. However, the extracted text from PDF files of scientific papers may contain unexpected special characters that are only intended for presentation to a human reader known as *ligatures*. Ligatures are joined letters like æ. Multiple PDF files contain such ligatures for various pairs of letters (*e.g.* *fi* is a single character for the letters f and i). These ligatures can lead to unexpected behaviors. For example, classifying two occurrences of the name “Winfield” as two

different names, because one of them contains the ligature *fi* and the other does not. These ligatures originate from the behavior of *TeX*¹, which is a typesetting system that has a prominent role in the creation of scientific papers. *TeX* automatically transforms certain pairs of letters into ligatures [1]. Ultimately, the ligatures are not present in the source file from which the PDF was created, but only in the PDF file itself.

2.1.2 Semantic Structure Recovery

The PDF format includes metadata such as title and authors. However, in our experience documents rarely occupy those fields with reliable data (as we describe later in Section 2.2). The text of a scientific document is structured into multiple fields (*e.g.*, title, authors, abstract, introduction). Although each of them has a specific semantic, the semantic is not explicit in the extracted text. For example, paragraphs, words or even characters are often stored as multiple separate objects (*e.g.* ö is often split into o and ¨). Visually identifying these fields is not a problem for a human reader. Each field has a well-known location in the documents' layout. However, it is still a challenge to extract them programmatically.

To understand the basic mechanism of recovering the semantic structure of a paper, we first have to introduce the concept of a *block*. Multiple lines of text are called a block if they can be confined in a box that does not overlap with the boxes of other blocks such as a paragraph. We recover the semantic structure of the paper by identifying the blocks in a PDF document and analyzing their position.

In order to identify what blocks correspond to the title and contributors, based on our experience, we make three assumptions about the layout of the paper:

1. We assume that the title and the contributors of the paper are mentioned in the header. We define the header as the section on the first page between the top of the page and the body of the paper (*e.g.* everything above the abstract or introduction) as can be seen in Figure 2.2. With this assumption we can narrow down the scope of the search when identifying the blocks that contain the title and the authors.

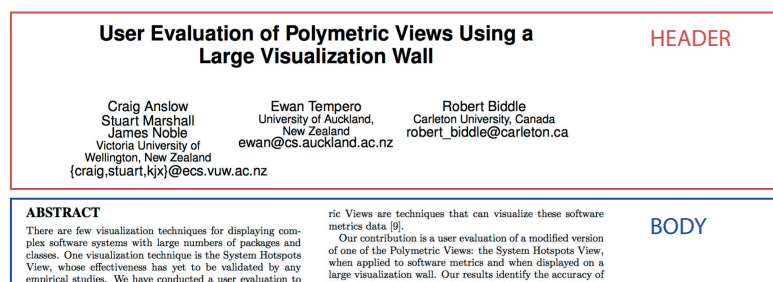


Figure 2.2: Example of a header

¹<http://tug.org/>

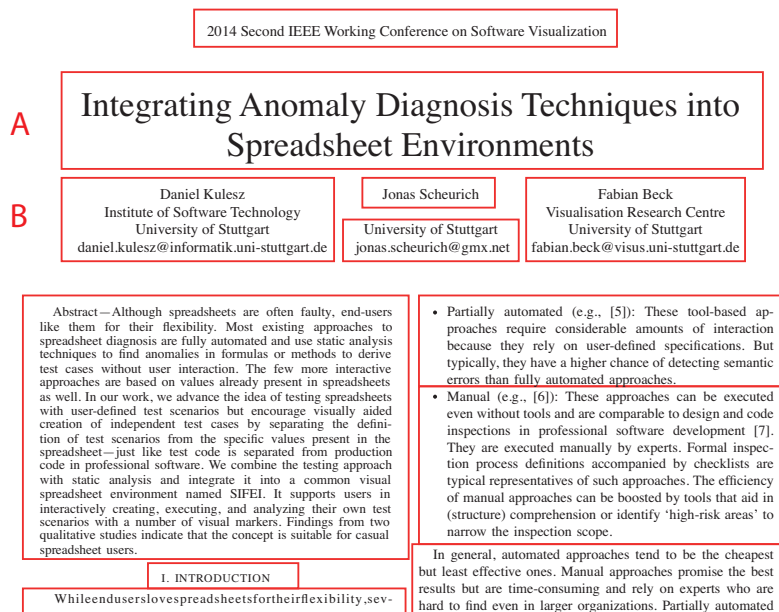


Figure 2.1: Page with highlighted blocks

- The title of the paper is the text with the *largest* font at the *top* of the first page (as seen in Figure 2.1-A).
- The names of the contributors are written *below* the title and are followed by one or more lines of text (e.g. e-mail or affiliation) as shown in Figure 2.1-B. However, we notice that papers use multiples layouts in the contributors’ block, such as:
 - One block below the title that mentions all contributors in the first line as shown in Figure 2.3.

Ala Abuthawabeh and Dirk Zeckzer
TU Kaiserslautern, Germany
Email: {abuthawa, zeckzer}@informatik.uni-kl.de

Figure 2.3: *Single block*

- Multiple blocks aligned horizontally as shown in Figure 2.4.

Andrea Adamoli Faculty of Informatics University of Lugano Lugano, Switzerland andrea.adamoli@usi.ch	Matthias Hauswirth Faculty of Informatics University of Lugano Lugano, Switzerland matthias.hauswirth@unisi.ch
--	--

Figure 2.4: *Single-row layout*

- c) Three author blocks horizontally aligned followed by additional blocks below. In the example shown in Figure 2.5, we highlight with a red-border a block that does not mention any contributor names but the affiliation of the aforementioned contributors.

Edward E. Aftandilian eaftan@cs.tufts.edu	Sean Kelley sean.kelley@tufts.edu	Connor Gramazio connor.gramazio@tufts.edu
Nathan Ricci nricci01@cs.tufts.edu	Sara L. Su sarasu@cs.tufts.edu	Samuel Z. Guyer sguyer@cs.tufts.edu
<div> Department of Computer Science Tufts University http://www.cs.tufts.edu/r/redline </div>		

Figure 2.5: *Grid layout with shared affiliation block*

- d) A layout of author blocks that do not follow rows and columns (shown in Figure 2.6).

Michael Balzer University of Konstanz, Germany	Oliver Deussen University of Konstanz, Germany
Claus Lewerentz Brandenburg University of Technology Cottbus, Germany	

Figure 2.6: *Non-grid layout*

- e) Author blocks that share a block with additional information (e.g., affiliation and e-mail) is shown in Figure 2.7.

Christian Collberg ^{1*}	Stephen Kobourov ^{1†}	Jasvir Nagra ^{2‡}	Jacob Pitts ¹	Kevin Wampler ^{1†}
¹ Department of Computer Science, University of Arizona, Tucson, AZ 85721. {collberg, kobourov, jpitts, wamplerk}@cs.arizona.edu ² Department of Computer Science, University of Auckland, Auckland, New Zealand. jas@cs.auckland.ac.nz				

Figure 2.7: *Non-grid layout with shared contact information block*

2.1.3 Modeling the Extracted Data

We model the extracted data as first-class objects to encapsulate the data extracted from papers. Each modeled paper has a title and references to author objects.

We observe that authors who contributed to multiple papers may have varying representations of their name. Figure 2.8 illustrates an example of this circumstance. The task of identifying unique authors in a community is known as *author disambiguation*.

D. Zeckzer*
Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern Germany

Dirk Zeckzer
University of Kaiserslautern
Kaiserslautern, Germany
zeckzer@informatik.uni-kl.de

Figure 2.8: Different representations of the same name in two papers

2.2 Analysis Examples

In this section we elaborate on the two concrete pipelines that we created. In these examples we look at the problems that arise during pipeline development and show how EggShell can help users to solve them. The pipelines use different implementations for the *data extraction* steps while they share the same implementation for the *modeling* step.

2.2.1 Text-based Data Extraction

For the *data transformation* step, this heuristic transforms the PDF files into plain text. It relies on the *pdftotext* binary from the tool XPDF² to obtain the plain text version. The tool mimics the visual appearance of the paper by inserting white spaces and empty lines. The resulting document allows us to identify blocks by analysing the position of the text that we use when recovering its semantic structure. However, we observe that some aspects are lost such as the size of the text.



Figure 2.9: Transforming a PDF file into plain text

During the *semantic structure recovery* step the heuristic searches the plain text version of the paper for an author block. It does so by going through several tests which look at different properties of the text, like the position of empty lines and blocks, and

²<http://www.foolabs.com/xpdf/>

stops on the first positive test. In the following we explain the tests included in the activity diagram shown in Figure 2.10.

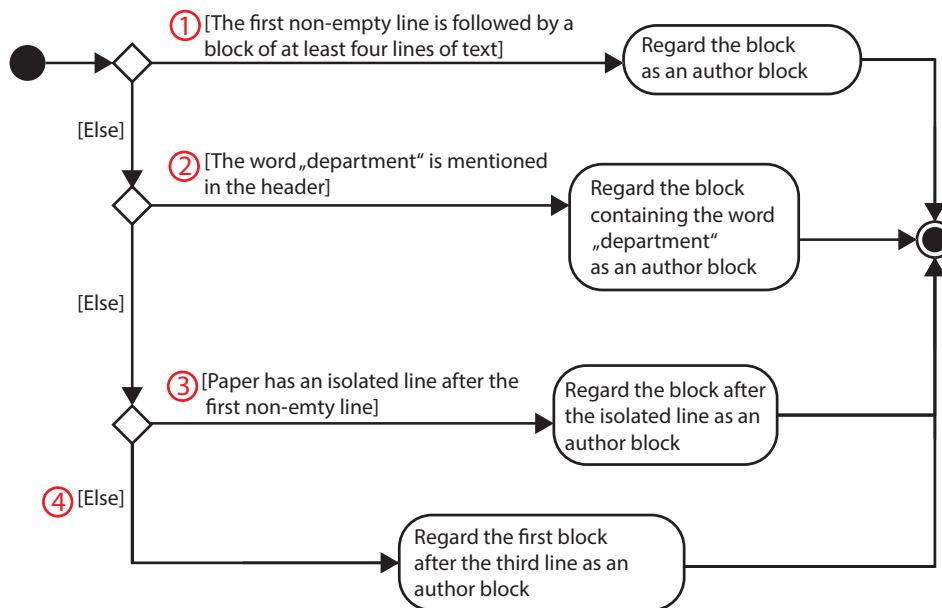


Figure 2.10: Extracting author names from the text version of a paper

1. If after the first non-empty line comes an empty line followed by a block of at least four lines, we assume that it is an author block.
2. We check if there are lines that contain the word *Department* in the first 20 lines of the page. We also make sure that we only check for such a line above the abstract. If such a line is present, we can assume that it comes right below the first line of an author block.

1	Two-dimensional C++
2	
3	Johannes Reichardt
4	Department of Computer Science
5	
6	Hochschule Darmstadt
7	j.reichardt@fbi.h-da.de

Figure 2.11: Contributor name above a line that states the department.

3. We check if the paper contains a single *isolated line* after the first non-empty line. An *isolated line* is a non-empty line that is surrounded by empty lines. If such an *isolated line* exists, we view the first non-empty line after it as the start of an author block.

4. If all tests above fail, we view the second non-empty line of the document as the start of an author block

In this heuristic we identify the first line of only one author block. For papers using the *single-row layout* presented in Section 2.1.2, this is enough to retrieve the authors from all blocks because they are all mentioned on the same line of the plain text version as shown in Figure 2.12.

Web Software Visualization Using Extensible 3D (X3D) Graphics

Craig Anslow, James Noble, Stuart Marshall*
Victoria University of Wellington, New Zealand

Robert Biddle†
Carleton University, Canada



1	Web Software Visualization Using Extensible3D (X3D) Graphics	
2		
3		
4	CraigAnslow, JamesNoble, Stuart Marshall*	Robert Biddle†
5	Victoria University of Wellington, New Zealand	Carleton University, Canada
6		
7		

Figure 2.12: A paper with a *Single-row layout* with three author blocks is transformed to plain text where all contributors are present on one line.

The heuristic extracts individual authors from the identified line. We notice that authors are usually separated by: 1) multiple white spaces 2) a comma, 3) the word “and”, and 4) a comma followed by the word “and”.

Once we have extracted the names of the contributors, we use their position to detect the block that contain the title. In this heuristic, we assume that the title block starts on the first non-empty line of the plain text version of the paper and continues until right before the line with the contributor names. Notice that this approach will return a wrong title if the paper contains text above the title or if it contains a subtitle.

We executed the described heuristic on a learning set of 100 papers and iteratively improved it. For each iteration we compared the results to a ground truth reference model that allowed us to spot failures.

Then we analyzed the accuracy of the heuristic on several sets of disjoint papers of various sizes (up to 140). The results showed that titles and contributors were detected correctly in 55% of the papers. The analysis of the the remaining (incorrect) cases, showed that in 52% of them the heuristic was not able to detect any of the contributors of the paper.

One limitation of this heuristic is that it cannot handle papers that mention contributors on multiple lines. However, in our learning data set this represents less than 10% of the cases.

2.2.2 XML-based Data Extraction

In the following we describe the data extraction steps of the second pipeline that is based on a XML representation of papers.

For the *data transformation* step, we transform the first page of each PDF to the XML format as shown in Figure 2.13. We created the tool *pdftoxml* which is a modified version of the *pdftohtml* binary of XPDF², to transform the PDF files. While *pdftohtml* transforms the PDF files to the HTML format, *pdftoxml* creates an XML version with the following elements:

1. A *page* element encapsulates all the data from a page of a PDF file. The *page* element does not have any attributes.
2. A *line* element contains a line of text up to a carriage return. In a multi-column paper, lines of text that have the same vertical position will be part of different line elements. All immediate children of a *page* element are *line* elements. A *line* element has the attributes *left*, *right*, *top*, *bottom* to describe its respective position.
3. A *span* element is always a child of a *line* element. Its content is the actual text from the paper. It has the following attributes:
 - (a) *id* (assigns each distinct combination of values for the following attributes an id. This way we can easily determine if two *span* elements have the same font attributes),
 - (b) *font-size*,
 - (c) *color*,
 - (d) *font-family*,
 - (e) *font-weight*, and
 - (f) *font-style*.

```
<page>
  <line left="82" right="527" top="71" bottom="88">
    <span id="f1" font-size="17" vertical-align="baseline" color="#000000" font-family="san...
  </line>
  <line left="126" right="483" top="116" bottom="127">
    <span id="f2" font-size="11" vertical-align="baseline" color="#000000" font-family="san...
    <span id="f3" font-size="11" vertical-align="baseline" color="#990000" font-family="san...
  </line>
  ...
```

Figure 2.13: XML version of a paper

For the *structure recovery* step we first identify all blocks from the first page of the paper and then determine which of these blocks represent the title block and author blocks. Figure 2.14 summarizes the process of determining the blocks of the XML version of the paper.

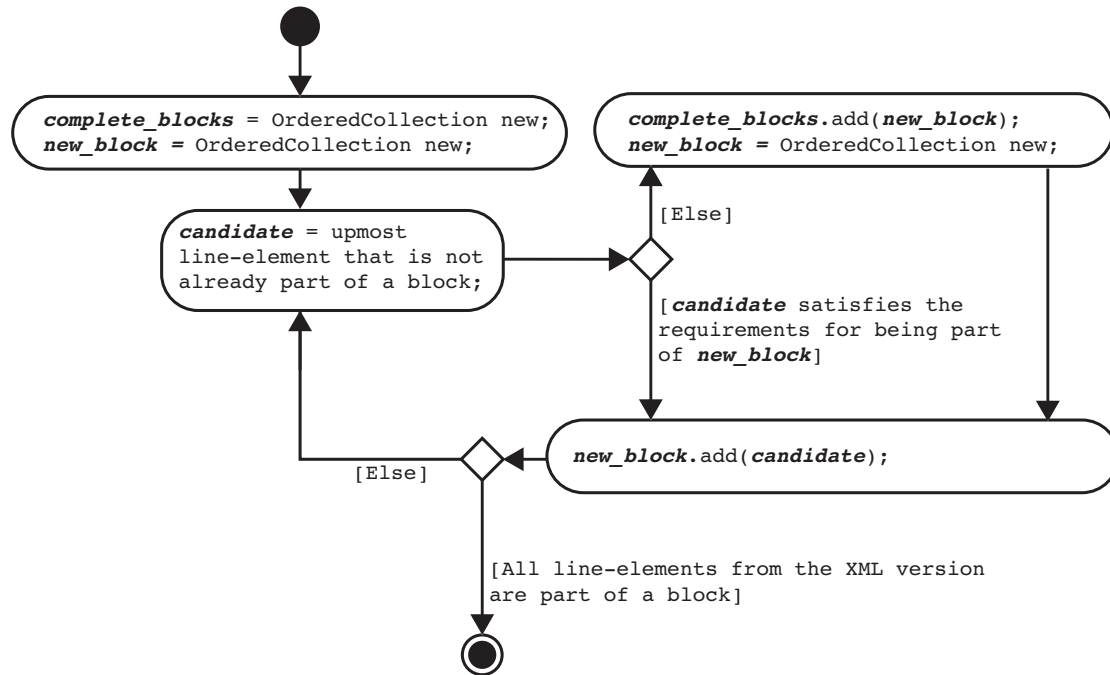
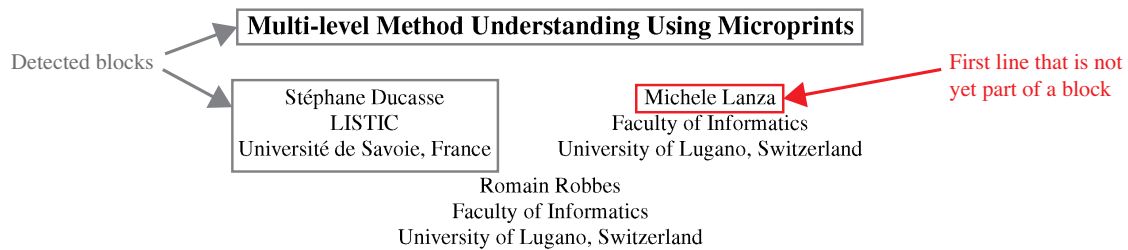


Figure 2.14: Activity Diagram for determining the blocks of the header

To detect each block, we first determine the upmost line-element in the XML version of a paper that is not already part of a block as shown in Figure 2.15. If multiple such line-elements exist, we choose the one with the leftmost position.



Abstract

Understanding classes and methods is a key activity in object-oriented programming, since classes represent the

structs. SeeSoft [4] can visualize large amount of code but it associates a color to a complete line and does not introduce a specific visualization for method semantics. Moreover, it does not provide object-oriented specific informa-

Figure 2.15: Determining first line of a new block

We then collect other lines in the same block. To this end, we first check if there is another line that is close and is horizontally centered to be considered part of the same block. Figure 2.16 shows examples of the possible cases. The distance between the lines must be under a certain threshold which we determined on the basis of the findings from our learning set.

Michele Lanza Faculty of Informatics	- Not centered at the same horizontal position + The lines are close to one another
Michele Lanza Faculty of Informatics	+ Both lines are centered at the same position - The second line is too far away from the first line
Michele Lanza Faculty of Informatics	+ Both lines are centered at the same horizontal position and close to one another

Figure 2.16: Multiple cases how lines are positioned in relation to each other

The system records the vertical distance between these first two lines and looks for a centered line below at that same distance as shown in Figure 2.17.

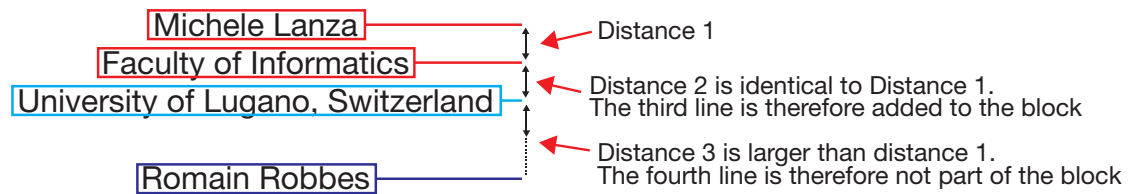


Figure 2.17: Allowed distance when adding lines to a block

An overview of the process of checking if a candidate line-element satisfies the requirements to be part of a block is shown in figure 2.18.

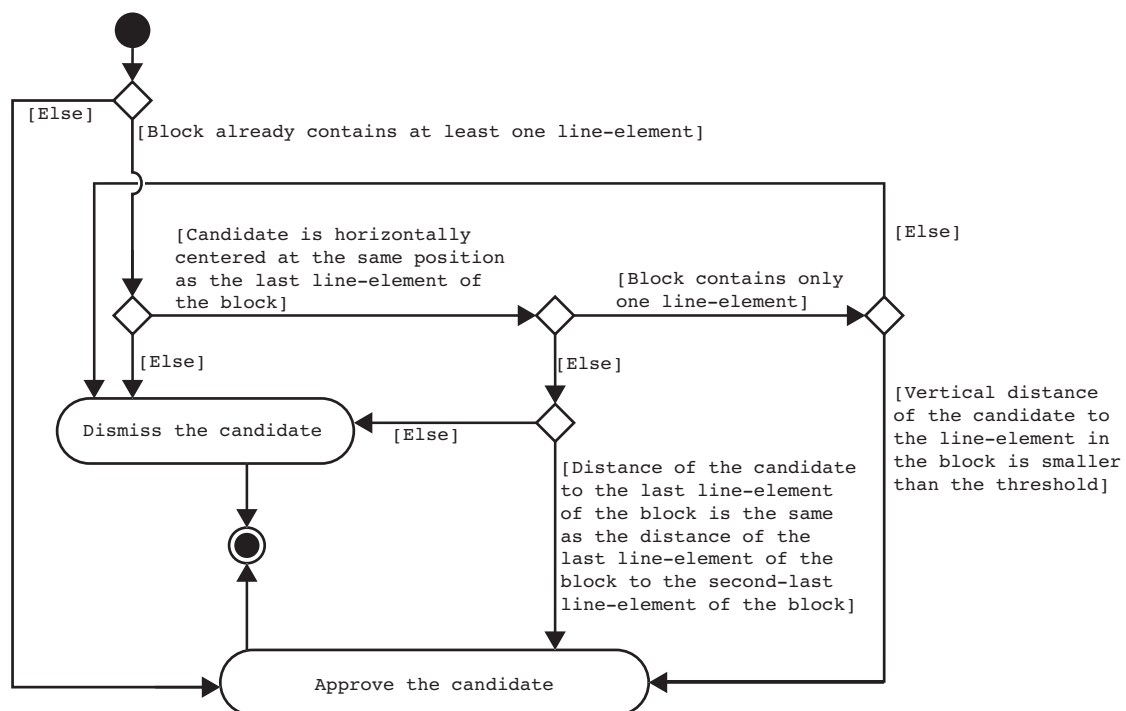


Figure 2.18: Activity Diagram for determining if a line-element satisfies the requirements for being part of a block

Once we have identified all blocks in the XML version, we select the blocks that are part of the header. We do this by first identifying the *body* of a paper because we assume that all blocks that are positioned at a higher vertical position than the body are part of the header. We identify the beginning of the *body* by looking for a title named *abstract* or *introduction*. If a paper contains neither of those, we look for the first block that contains at least two lines, that has the text fully justified, and on which the distance from the

left edge of the page does not surpass a certain threshold. Such a block is highlighted in Figure 2.19

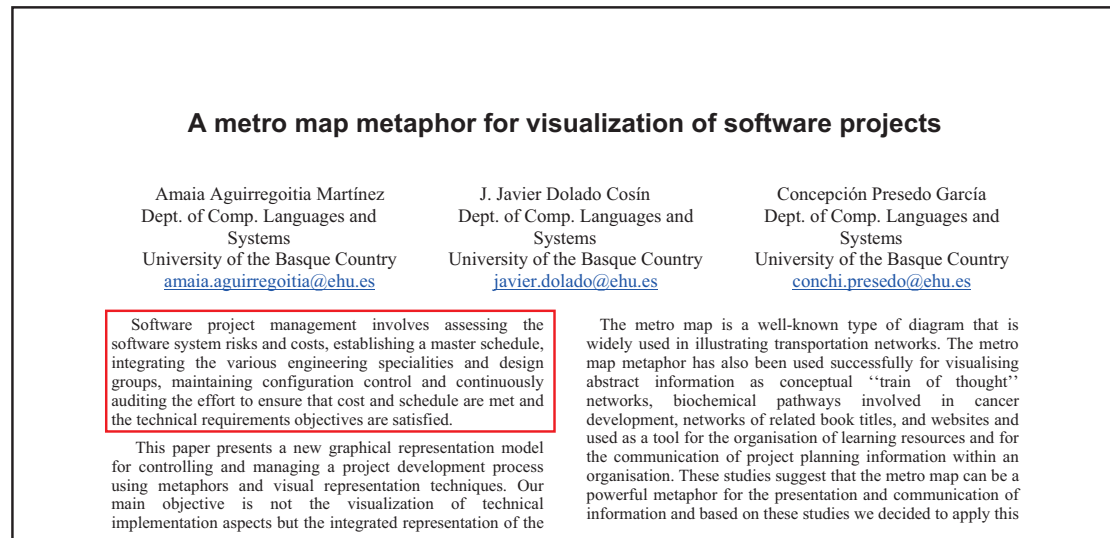


Figure 2.19: The first block of the body of a paper

Next, we identify the title block which is always the block of the header that contains the text with the biggest font size. The author blocks are then identified by looking at each block from the header that is positioned below the title. We filter out blocks where the first line does not describe a contributor by detecting certain words such as names of universities and country names. We regard the remaining blocks as author blocks and proceed to extract the names of the authors from the first line of each of them.

We assessed the accuracy of this heuristic using the same sets of papers as the ones we have used to assess the accuracy of the text-based data extraction described in Section 2.2.1. The titles and contributors were detected correctly in 70% of the papers. In the remaining 30% of papers only some of the contributors were extracted correctly.

In conclusion, while we were not happy with the accuracy achieved by the *Text-based* data extraction, the flexibility of our framework allowed us to improve the accuracy by exchanging the technology used. Hence, the *XML-based* data extraction correctly detected the title and the contributors of approximately 15% more papers than the *Text-based* approach. Additionally, the *XML-based* data extraction was able to create a better approximated model, in 75% of the cases, where it had partial data. Neither of the heuristics rely on metadata stored in fields of the PDF files, because it was only correct in 12% of the papers. In all the other cases it was either incorrect, incomplete or not available at all.

2.2.3 Mutual Modeling Step

Once we have extracted the desired data from papers, we move to the next step in the pipeline which is to model the data. It consists of two steps: (1) author disambiguation, and (2) name normalization.

For the *author disambiguation* part, we use an algorithm that is inspired by the research in the field of Strotmann *et al.* [4].

The algorithm starts by associating all identical names to unique authors. It then groups the resulting authors into *compatibility groups*. Each *compatibility group* contains candidate authors that could refer to the same individual but have different name representations. The algorithm identifies unique authors from the analysis of co-authors within a compatibility group. Figure 2.20 shows an example of this process.

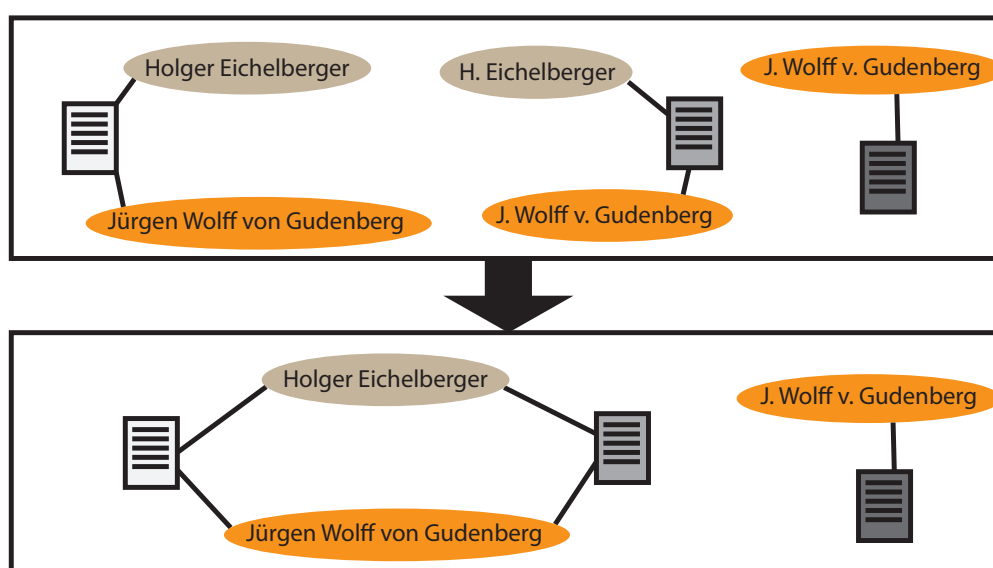


Figure 2.20: Identifying unique authors based on their co-authors. Circles with the same color indicate authors of the same compatibility group.

After the *author disambiguation* we normalize the authors' names. A normalized name is written with the family name in the beginning, followed by a comma and the initials of all given names in order as shown in the following example:

Concepción Presedo García → García, C. P.

We observe that, although usually the family name and each given name are each made up of one word, there are exceptions. We only consider two of such exceptions that were the most common cases found in our learning set:

- If a given name contains a hyphen, we turn it into two initials as follows:

Anne-Marie \rightarrow A.-M.

- In some languages, family names that consist of multiple separate words are common (e.g. family names that consist of two or more words like "von de Guzman"). For this implementation we created a list that contains some of these special cases. If such a case is detected, we normalize the name accordingly.

When making changes to the steps of a pipeline, it is hard to assess the impact on its performance. In order to facilitate such assessments, EggShell provides a visualization that helps in the process of improving pipelines. We explain this visualization in the next section.

2.3 Assessment Visualization

We observe that given the complexity of the process of extracting and modeling data from papers, we expect that some modeled papers might have an incorrect title, and/or contributors. Consequently, we want to assess the accuracy of each implemented pipeline. For each pipeline we want to understand what failed. We formulate the following questions as requirements that the assessment tool should satisfy.

RQ1) How different is the performance between multiple pipelines?

RQ2) How do pipelines perform for each paper?

RQ3) What data was modeled wrongly (if any) for each paper (e.g., title, authors)?

RQ4) What does the data of failed models look like for each pipeline and compared to a ground truth data set?

RQ5) What led to failures for poorly modeled papers?

To satisfy these requirements we designed a visualization that consists of three parts: 1) *The Assessment Grid* provides an overview also allows users to spot failures, users can investigate a failure through a 2) *Pop-up* that appears when one of the papers is focused, and users can also require 3) *Details-on-demand* of each paper towards detecting the cause of a failure.

2.3.1 The Assessment Grid

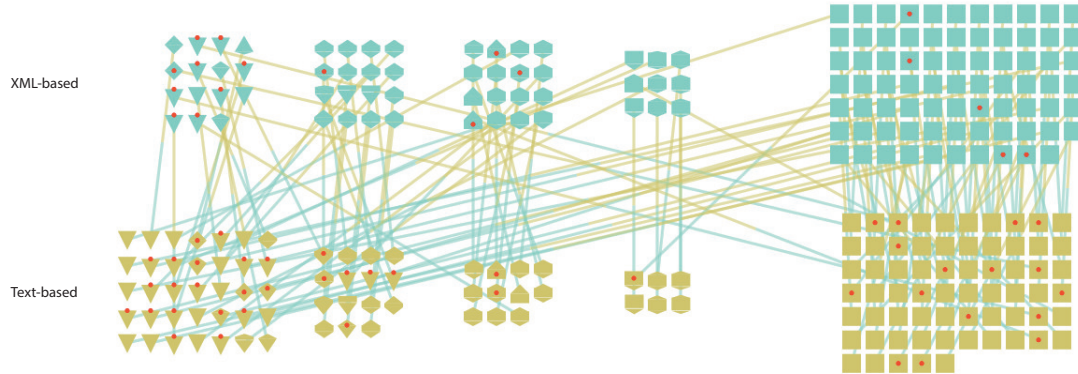


Figure 2.21: Assessment grid of the assessment visualization

In the Assessment Grid each modeled paper is represented by a glyph as shown in Figure 2.21. The shape of the glyph encodes an overview of the accuracy of the modeled data. The visualization can be used to assess the result of several pipelines simultaneously. Glyphs representing paper models from the same pipeline are grouped together by vertical position and color. Edges connect glyphs that represent different models of the same paper. The horizontal alignment of glyphs encodes how accurately the contributors of a paper were modeled. The further right the position, the more accurate the contributors of the respective model.

A paper can have several contributors. We observe that during the process of modeling the authors of papers two main issues arise: 1. The modeled paper has assigned an author who is not an actual contributor of the paper (*fake* author), and 2. The pipeline overlooks a contributor who is, therefore, missing in the model (*missing* author).

The accuracy of matched contributors for each paper is evaluated. The glyphs representing those papers are grouped into five groups according to that score. Groups are then aligned horizontally, and ordered by increasing score from left-to-right. We opted for this discrete sort, instead of encoding the score in the vertical position of glyphs, to avoid overlapping.

The edges in the assessment grid connect the glyphs that represent the same paper. The direction of the edges between the glyphs of two pipelines provides users insight into their overall performance. That is, outgoing edges that end in a glyph located more to the right, expose more accurately modeled papers. We use bi-color edges to denote the assigned color of the pipeline of the glyph on each side of the edge. This can be specially useful when the user wants to answer RQ1).

Initially, we considered multiple complex glyphs, but we observed that their readability decreased for a larger number of modeled papers. The chosen shape is therefore rather

simple. The design of the glyph is meant for coping with RQ2). It specifically answers: *RQ2.1*) Did the pipeline extract the title from the paper correctly?, and *RQ2.2*) How accurately did the pipeline extract the contributors of the paper?

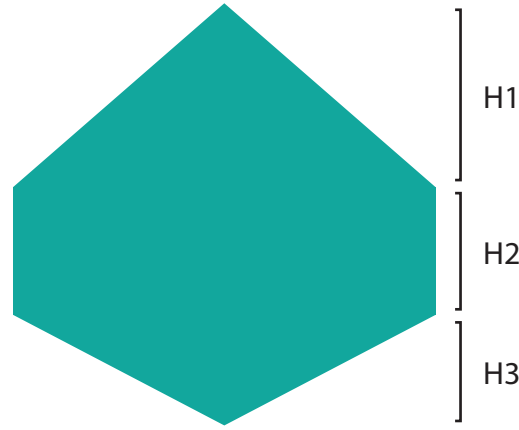


Figure 2.22: A glyph representing a paper used in the Assessment Grid

The glyph consists of the three vertically aligned parts shown in Figure 2.22:

1. An up-facing triangle at the top. Its height $H1$ indicates the proportion of extracted authors who are not present in the actual paper (*fake authors*)
2. A rectangle in the middle. Its height $H2$ indicates the proportion of correctly mapped authors.
3. A down-facing triangle at the bottom. Its height $H3$ indicates the proportion of authors who are missing in the modeled paper.

Each part has the size of the proportion it represents encoded in its height, which we calculate as follows:

$$height = \frac{\alpha}{\beta + \gamma + \delta} \quad (2.1)$$

Where:

- α = number of authors who are represented by the part
- β = number of authors correctly matched
- γ = number of missed authors
- δ = number of fake authors

If either β , γ or δ is zero, the height of that part of the glyph will have a height of zero and will therefore not be visible. Figure 2.23-A shows the case where γ and δ are zero and Figure 2.23-B shows the case where β and δ are zero. Figure 2.23-C shows the case where β and γ are zero which cannot occur because each paper has at least one matched or missed author as shown in Figure 2.23-D.

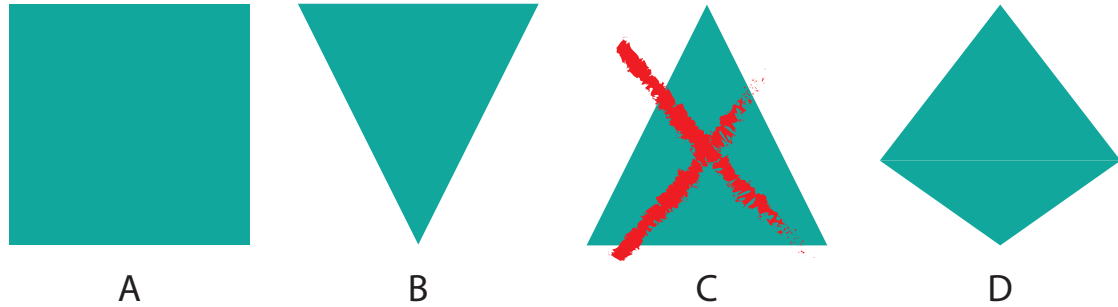


Figure 2.23: Extreme cases for the paper glyph

Additionally, a red dot over the glyph indicates that the title in the model is faulty. We observe that our pipelines are quite accurate in extracting the title, in consequence we decided on not highlighting these failures to ease the readability of glyphs. We rather disclose more detail in the popup of the visualization.

2.3.2 Popup

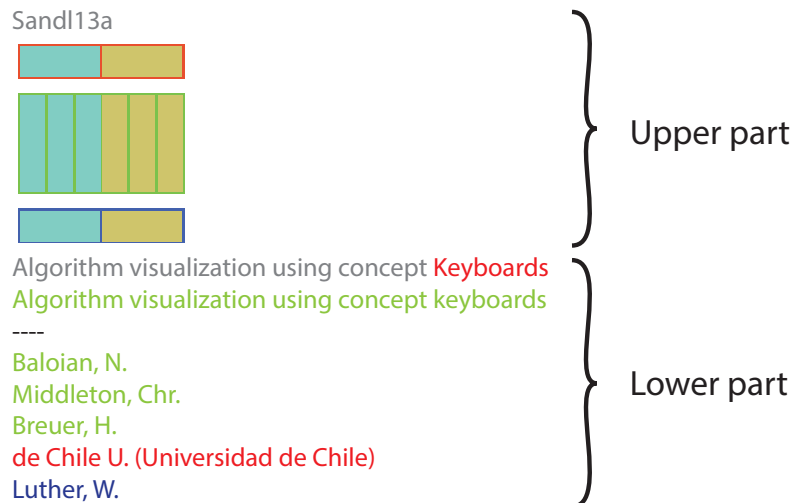


Figure 2.24: Popup view

After obtaining an overview of the accuracy of the various pipelines and probably spotting some issues, users get more details of a modeled paper by hovering over it and seeing a popup. The popup, which is shown in Figure 2.24, is designed to answer the questions RQ1), RQ2), RQ3) and RQ4). Consequently, the popup consists of two vertically aligned parts:

1. In the upper part is shown a glyph that compares the accuracy of all pipelines for the underlying paper.
2. In the lower part the modeled data is displayed and, eventually, hints to infer where the pipeline failed.

2.3.2.1 Upper Part of the Popup

Initially, we designed the pop up as a glyph to be used in the Assessment Grid but soon we realized that it hindered comprehension, and instead we developed a more simple glyph. However, we felt that the information provided by the design helps to answer important questions and therefore we decided to build upon the idea and use it as a pop up.

The pop up is composed of a glyph that contains three vertically aligned sections dedicated to analysing the possible outcomes of the modeling process of authors of a paper: *matched* authors, *missing* authors, and *fake* authors. The type of the outcome is encoded by a colored border. Matched authors are represented in the middle section with a green border, fake authors are at the top with a red border and missing authors are at the bottom with a blue border. Each section, in turn, is split into horizontally aligned cells which each represent a group of authors modeled by a pipeline that can be identified by its fill-color. The cells are themselves split into rectangles which each represent a single author. The glyph tackles the following specific questions which match with RQ1) and RQ2):

RQ1) How different is the performance between two pipelines?

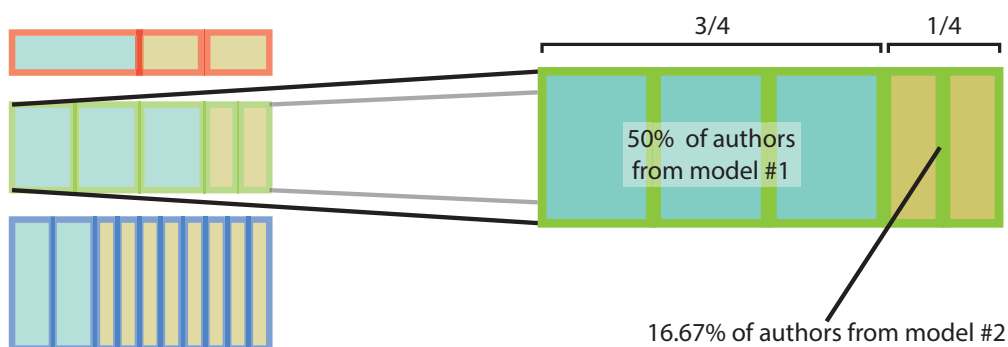


Figure 2.25: Comparing the size of the cells in a section

The user can compare the differences in amounts of fake, missed and matched authors of a paper between models by comparing the widths of cells in the corresponding section as shown in Figure 2.25 (*e.g.*, If a cell takes up most of the width of the section representing matched authors, we know that it modeled a bigger percentage of its authors correctly than the other models).

RQ2) How do pipelines perform for each paper?

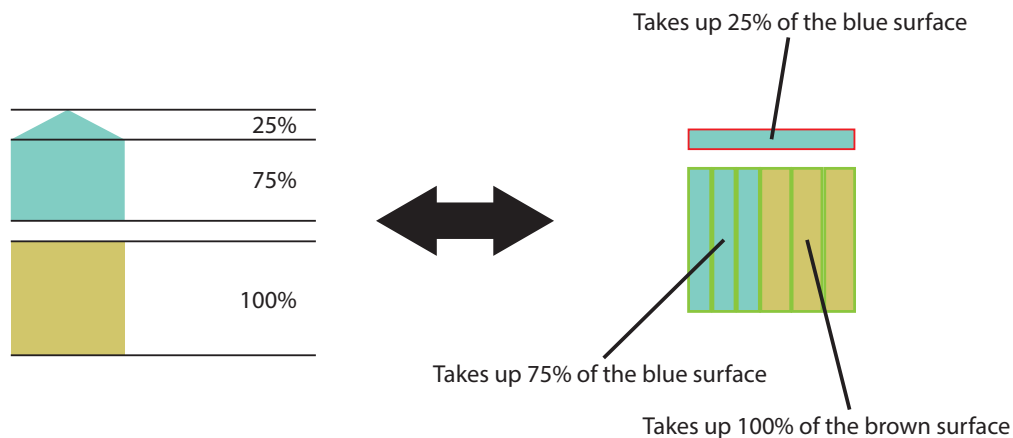


Figure 2.26: Comparing glyphs from the main view to the glyph from the pop up

The user can determine how accurately a pipeline modeled the authors by comparing the size of its cells as shown in Figure 2.26. The surface of one cell has the same relation to the surface of all cells of that model as the number of authors that are represented by the cell to the number of all authors from that model (*e.g.*, if the cell for matched authors of a model takes up 75% of the volume of all cells from that model, this means that 75% of all authors of the model are matched correctly). The total area of the cells is the same for each pipeline.

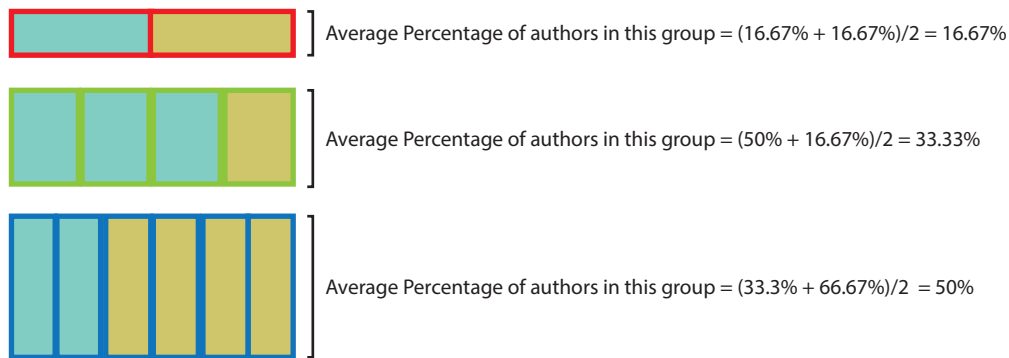


Figure 2.27: The height of each section represents the average percentage of the corresponding authors for each pipeline

Additionally, the user can determine the average performance across all pipelines by comparing the height of each section from the glyph as shown in Figure 2.27. The height of each section corresponds to the average percentage of represented authors (e.g. If the section for matched authors takes up 33.33% of the height of all sections, this means that across all models, on average 33.33% of the authors from each model are matched correctly).

2.3.2.2 Lower Part of the Popup

The lower part of the pop up shows an overview of the extracted data as well as information that is missing in the model.

On top, there is the extracted title as shown in Figure 2.28. If it matches with the title from the reference data, it is completely green. However, if the titles do not match, the title is colored grey on the left, and red from where the first error occurs. In this case, the pop up also shows the actual title on the next line.

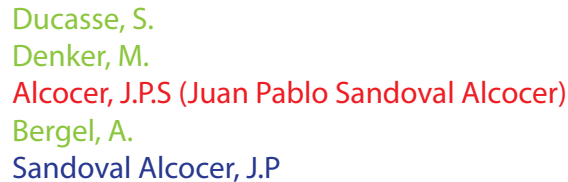
Performance Evolution Blueprint: Understanding the **impact of Software Evolution on Performance**
 Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance

Figure 2.28: Title of a paper as shown in the Popup

Below the title, the pop up lists the various authors of the paper. First it lists all the author names that were extracted correctly by the pipeline in green. Next, it lists, in red, all the author names that are present in the model, but not in the reference data. The text between the parentheses shows the names before they were normalized. Lastly, all author names that are present in the reference model but not in the model, are listed in blue.

Users can use this list to answer the questions RQ3) and RQ4). The example in Figure 2.29 shows a fake author and a missing author. When we look at the name in the

parentheses we can see that the pipeline was able to detect the position of the name in the paper, but was not able to normalize it correctly. Therefore, the paper in our model contained the fake author *Alcocer, J.P.S.* which should be replaced with the missed author Sandoval *Alcocer, J.P.*.



Ducasse, S.
Denker, M.
Alcocer, J.P.S (Juan Pablo Sandoval Alcocer)
Bergel, A.
Sandoval Alcocer, J.P

Figure 2.29: Author names from the Popup

2.3.3 Details-on-Demand

When a user has analyzed the outcome of some pipelines through the Assessment Grid, she may want to get a better understanding of what led to failures and therefore answering RQ5), so she can eventually improve the pipeline. Although viewing the paper with an external tool can provide some clues, we see at least two drawbacks:

1. Leaving EggShell to search for the PDF file is not convenient.
2. The information of the outcome of the internal steps of the pipeline is missing.

To overcome this, when a user selects a paper in the assessment grid she gets details-on-demand. In it, a view of the extracted version (before it is normalized and modeled) of the first page of the paper is displayed. Thus, users can investigate the cause that triggered a failure.



Figure 2.30: Details-on-demand representing the extracted first page of a paper from two pipelines: Text-based and XML-based.

Figure 2.30 shows the view in which a reconstruction of the extracted data of each pipeline is presented side-by-side. Each one is surrounded by a border with a distinct color. In the presence of ground truth data set, correctly determined contributor names are colored green and those of faulty contributor names are colored red.

Through this view users can answer the following three questions which help us answer RQ5): *RQ5.1*) determine where in the paper a fake author has been found. *RQ5.2*) get an idea of the layout of the paper. *RQ5.3*) verify if the first step of the pipeline performed well by comparing the representation in the view to the PDF file.

The view is created with the extracted data (first step of the pipeline). Therefore, if the representation in the view differs from the PDF file, we can conclude that the first step of the pipeline failed, which answers RQ5.3).

2.3.4 Analysis Example

In this section we give an example of how the assessment visualization was used while developing the presented pipelines. In the example, we show how we used the visualization

for implementing blacklisted words to prevent false positives when identifying author blocks in the *semantic structure recovery* step of the XML-based pipeline. Supposed author blocks are dismissed if they contain any of the blacklisted words on the first line as shown in Figure 2.31.



Figure 2.31: If the word *University* is blacklisted, the *shared affiliation block* is dismissed from being an author block because it contains a blacklisted word on the first line.

Whenever we wanted to assess the impact of a candidate for a blacklisted word, we created two versions of the pipeline:

1. *old version* of the pipeline where the word is not blacklisted
2. *new version* of the pipeline where the word is blacklisted

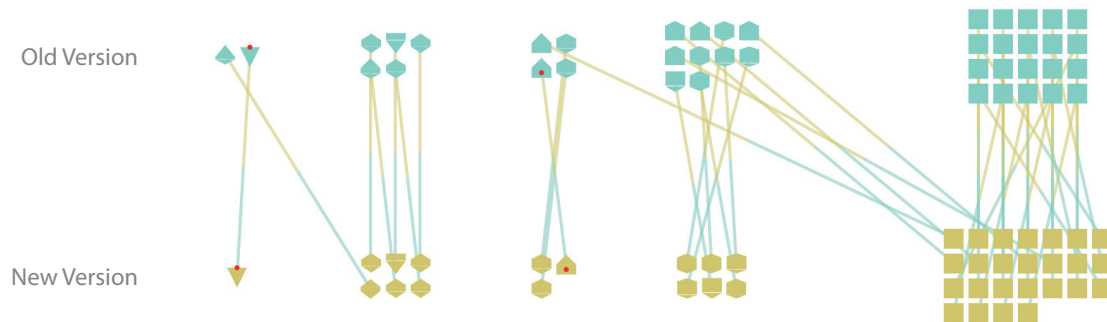


Figure 2.32: Assessment grid for two pipelines from the example analysis

Using the assessment grid of our visualization, we were able to assess the difference in performance between the two versions of the pipeline as shown in Figure 2.32 where we assessed the impact of blacklisting the word *Software*. We found that for all papers where the two versions of the pipeline did not perform the same, the *new version* always performed better.

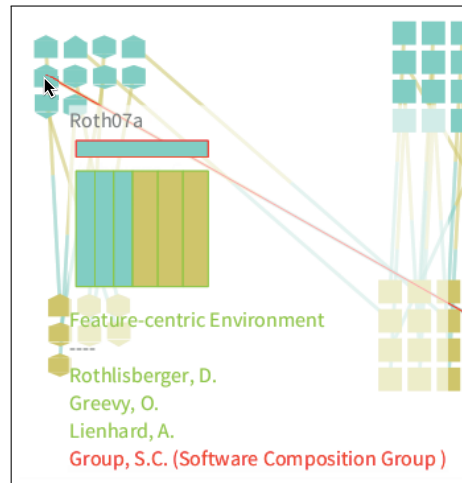


Figure 2.33: Popup showing that *Software Composition Group* was mistaken as an author name

The assessment grid also helped us identify potential words for the blacklist by exposing poorly performing papers with *fake authors* through the glyphs and their position. We then used the Popup to identify what the authors of the failed papers look like and how they compare to the reference model as seen in Figure 2.33, in order to identify words that could be blacklisted.

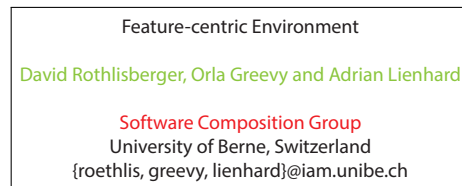


Figure 2.34: An excerpt of the *Details-on-demand* where we verify that the term *Software Composition Group* is in fact mentioned on the first line of a block

For each failed paper where we identified candidates for blacklisted words, we also checked if the problems are in fact falsely identified author blocks. We did this by checking in the *Details-on-Demand* if the fake authors are listed on the first line of a non-author block of the transformed version of the paper as shown in Figure 2.34.

2.4 Technical Details

This section describes installation instructions and usage examples for EggShell.

2.4.1 Installing EggShell

1. *Getting Moose.* In order to run EggShell, we will need the *Moose* 5.1 image and the *Pharo Virtual Machine* to run it. Both can be downloaded from the official Moose[3] website³.
2. *Eggshell Repository.* We can load EggShell from the following SmalltalkHub⁴ repository:

```
MCSmalltalkhubRepository
  owner: 'DominikSeline'
  project: 'Eggshell'
  user: ''
  password: nil
```

3. In order to run EggShell we also need to download the precompiled binaries of the *EggShell PDF Data Extractor*⁵. The folder *tools* which contains the binaries has to be placed in the same folder as the Moose image.

2.4.2 Usage Example

To create a model of a scientific event from a set of PDF files, we need to group those files together into a folder which we refer to as the *PDF folder*. EggShell provides an example *PDF Folder* which can be used for this usage example. In order to get this folder and run the usage example, the user has to open the *Playground* in *Moose* and go through the following steps:

1. In order to download the example *PDF Folder* the user has to execute the following code:

```
EggShell loadExamplePDFs.
```

This will download the folder *examplePDFs* into the same folder as the *Moose* image.

2. To model the community, the user has to call the method *modelPapersInFolder:* *using:* of the class *EggShell* with a String of the relative path to the *PDF folder* and a Symbol which indicates the pipeline he wants to use as the arguments. Currently, the user can choose between the two example pipelines we introduced in this thesis with the symbols *#ESTextPipeline* and *#ESXMLPipeline*. We recommend *#ESXMLPipeline* for a more accurate result. The user therefore has to execute the following code:

³<http://www.moosetechnology.org>

⁴<http://smalltalkhub.com>

⁵<https://github.com/selineDominik/EggShell-PDF-Data-Extractor>

```
model := EggShell modelPapersInFolder: 'examplePDFs'
      using: #ESXMLPipeline.
```

After we have created a model, we can now assess its accuracy with the visualization that EggShell provides. To do this, we execute the following steps:

1. We use objects of the class *ESAccuracyAnalyzer* to assess how accurately a model depicts a community. First, let's create the analyzer object:

```
analyzer := ESAccuracyAnalyzer new.
```

2. This object can be used to assess one or more pipelines. Each resulting model from the pipelines can be added by sending the *addVersion: named:* message with the model and a name as the arguments. The name helps in identifying the source of the model within the visualization.

```
analyzer addVersion: model named: 'XML Pipeline'.
```

In order to assess the accuracy of the models, we need to compare it against a reference model that contains the ground truth of the same data set.

3. In Eggshell we provide a reference model of a community by placing a *Comma Separated Value* (CSV) file with the ground truth data into the *PDF folder*. The folder *examplePDFs* already includes such a file. We create a model from CSV by sending the message *model* with the path to the folder that contains the CSV and PDF files:

```
reference_model := ESReferenceModelLoader new
               model: 'examplePDFs'.
```

The CSV file has to be named *original_model.csv* by default, however the user can choose his own name by sending the message *setCSVFileName* to the *EReference-ModelLoader* with the desired name as an argument. The header (first row) of the CVS file must have the following names: “*filename*”, “*title*”, and one or more fields with the name “*contributor*”.

4. After we have created the reference model, we can add it to the analyzer by sending the *setOriginal:* message.

```
analyzer setOriginal: reference_model.
```

5. Lastly we execute the view by sending the message *open* to the analyzer.

```
analyzer open.
```


6. All in all we executed the following code inside the *Playground* for the usage example:

```
EggShell loadExamplePDFs.  
model := EggShell modelPapersInFolder: 'examplePDFs'  
        using: #ESXMLPipeline.  
analyzer := ESAccuracyAnalyzer new.  
analyzer addVersion: model named: 'XML Pipeline'.  
reference_model := ESReferenceModelLoader new  
        model: 'examplePDFs'.  
analyzer setOriginal: reference_model.  
analyzer open.
```

7. We can now assess the performance of pipelines through the view described in the section 2.3.

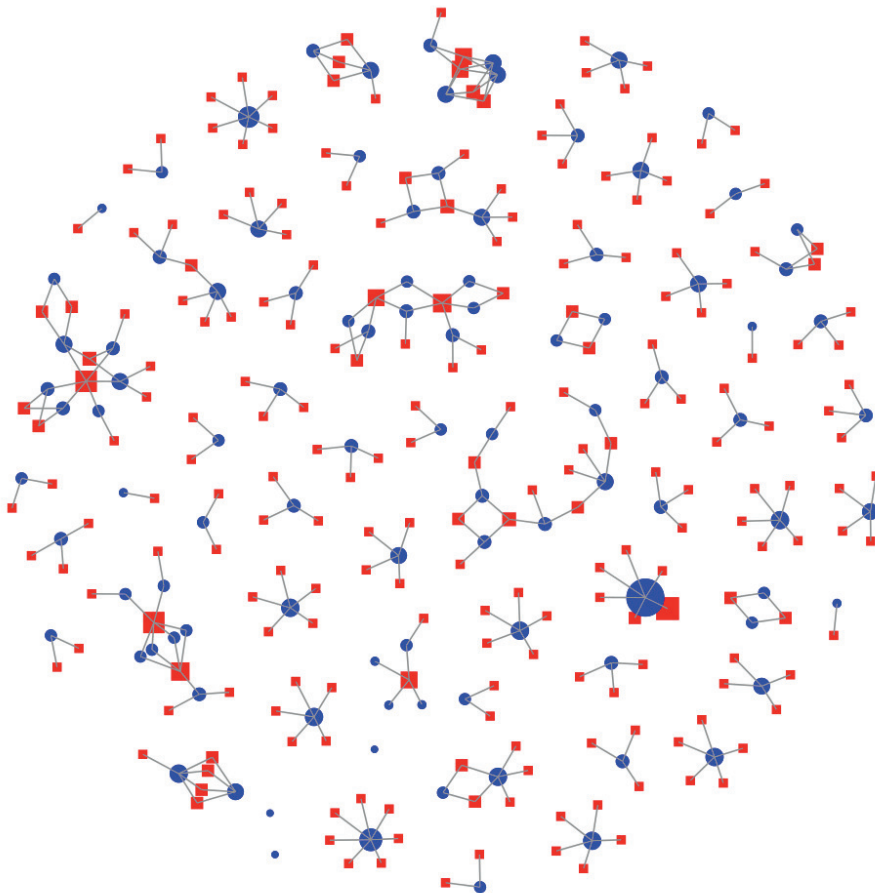


Figure 2.35: Visualization of model

Figure 2.35 shows a simple visualization of models which we have included for demonstration purposes with the class *ESEventVisualizationExample*. Creating and visualizing a model for such a scenario therefore looks as follows:

```
model := EggShell modelPapersInFolder: 'examplePDFs'
      using: #ESXMLPipeline.
visualization := ESEventVisualizationExample new.
visualization setModel: model.
visualization open.
```

Recently we successfully used EggShell to produce a modeling pipeline for the visualization of collaboration in scientific communities [2].

3

Conclusion and Future Work

The analysis of the collaborations among authors of a scientific community requires to model it. Researchers who analyse such collaborations spend great effort to create such a model.

When creating the model, researchers do not always have the papers' data, such as title and authors, available in a formatted source. In such cases they have to extract the data directly from the published files, which are often provided in PDF. However, the PDF format does not contain information about the semantic structure of a document. Therefore, extracting the needed data is a laborious task. We call the collection of steps that need to be executed to create a model a pipeline. We propose EggShell, a workbench for defining pipelines that create models of scientific communities from the PDF files of their publications. It offers users dedicated visualizations to assess the performance of these pipelines and investigate the reason of a failure that helps them to improve pipelines' accuracy. We elaborate on an analysis example on which we stress the benefits of EggShell creating two modelling pipelines. We collected the 300 papers published in SOFTVIS/VISSOFT community. We split the collection into a learning set of 100 papers that we used to tailor a modeling pipeline. We then used the remaining 200 papers of the collection to test the accuracy of the modeling pipeline which achieved an accuracy of 70%.

For future work we see multiple opportunities for improvement in both the assessment visualization and the example pipelines. Especially the example pipeline that first transforms the PDF files into the XML format provides a good basis for improvement because it is able to reconstruct a good portion of the semantic structure of the paper which is lost in the PDF file. Additionally, it would be possible to expand the pipelines

to extract additional information such as the authors' email addresses, the country and citations. For the assessment visualization we see two points where future work could be done. The assessment grid theoretically allows the user to compare any number of pipelines against each other. However, in practice it does not scale very well and becomes clustered for more than two pipelines. Future work could therefore explore ways to improve the scalability of the visualization.

4

Anleitung zu wissenschaftlichen Arbeiten

4.1 Creating a new pipeline for EggShell

In this tutorial, we will create a new pipeline for EggShell. The pipeline will have two parts: One for importing information from files into Pharo, and one for creating a model from said information. The pipeline will be very simple, and therefore not accurate. Also, it will not contain the information that is needed to use it with the *ESAccuracyAnalyzer*. It, however, is a good starting point to create a more accurate pipeline.

4.1.1 Importer

First, we create a class for importing data from PDF files into Pharo. It should extend the class *ESImporter* and we will name it *ESSimpleTextImporter*.

```
ESImporter subclass: #ESSimpleTextImporter
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'EggShell'
```

A class that extends *ESImporter* needs to implement the method *importFolder*. This method should extract the papers from the PDF files in a folder and return it as an *OrderedCollection*.

```
importFolder: fileReference
  | files import |
  files := fileReference allChildren
```

```

    select: [ :each | each basename endsWith: '.pdf' ].
    import := files
    collect: [ :pdfreference | self importPdf: pdfreference ].
^import

```

The method *importFolder:* takes a reference to the folder which contains the PDF files. First, it creates an *OrderedCollection* which contains references to all PDF files at the root of the folder. This *OrderedCollection* is assigned to the variable *files*. Next, it imports each of the PDF files into Pharo and saves the data into the variable *import* which it then returns. The method *importPDF* takes care of the import process. We now need to implement this method:

```

importPdf: pdfreference
| string pdfData |
string := self importPdfAsText: pdfreference path.
pdfData := self extractAuthorsAndTitleFromText: string.
^ pdfData

```

In the method *importPdf*, we do two things: First, we create a string from a given PDF file and assign it to the variable *string*. We use the method *importPDFAsText*, which we need to implement, to create this string. The method *extractAuthorsAndTitleFromText*, which we also need to implement, then searches *string* for the title and the contributor names and returns them. We save the title and the contributor names into the variable *pdfData* which we then return.

First, let us implement the method *importPdfAsText*:

```

importPdfAsText: path
| pdftotextPath string filePath cmd |
pdftotextPath := FileSystem disk
    stringFromPath: FileSystem disk workingDirectoryPath
    / 'tools' / 'mac' / 'pdftotext'.
filePath := FileSystem disk stringFromPath: path.
cmd := pdftotextPath , ' -f 1 -l 1 -layout ' , filePath , ' -'.
string := (PipeableOSProcess command: cmd) output.
^ string

```

The method *importPdfAsText* executes the following steps:

1. The method assigns the path to an external binary to the variable *pdftotextPath*.
2. The method assigns the path to the PDF file to the variable *filePath*.
3. The method uses the variables from the preceding steps to form a command for unix terminals and assigns it to the variable *cmd*. This command will use the binary that is located at *pdftotextPath* to extract data from the PDF file that is located at *filePath*.

4. The method executes the command *cmd*, and stores the extracted data into the variable *string*, which it then returns.

After the PDF file is present as a string inside Pharo, we can determine the title and the contributor names from the paper. This is done by the method *extractAuthorsAndTitleFromText* which we need to implement:

```
extractAuthorsAndTitleFromText: text
| lines newData |
newData := Dictionary new.
lines := text findTokens: String lf ,
    String cr , String crlf escapedBy: nil.
newData at: 'contributors'
    put: (((lines at: 3) findTokens: ',')
        collect: [ :name | name trim ]).
newData at: 'title'
    put: (lines at: 1).
^ newData
```

In the method *extractAuthorsAndTitleFromText*, we first create an *OrderedCollection* which holds all lines from the variable *text*. Next, we set the first line of the variable *lines* as the title of the paper. That way, we can determine the title of many papers correctly, because the title is often present at the first line of the variable *lines*. We then try to extract author names from the third line of *lines*. We assume that the names of the contributors are listed on the third line of *lines* and that multiple names are separated by a comma. This assumption, however, will not always hold true and is, therefore, one of the reasons why this pipeline is not accurate.

We can now create an *OrderedCollection* with the imported papers. The following code first defines the location of the PDF files and then assigns an *OrderedCollection* with the imported papers to the variable *import*:

```
fileReference := FileSystem disk workingDirectory / 'pdfFiles'.
import := (ESSimpleTextImporter new) importFolder: fileReference.
```

4.1.2 Modeler

After we have imported the PDF files into Pharo, we now want to create a model from the imported data. To do this, we implement the class *ESSimpleScienceEventModeler* which extends the class *ESScienceEventModeler*.

```
ESScienceEventModeler subclass: #ESSimpleScienceEventModeler
instanceVariableNames: ''
classVariableNames: ''
category: 'EggShell'
```

This class needs to implement the method *model:*. In this tutorial we create a very simple version of this method:

```
model: anOrderedCollection
  | model |
  model := Dictionary newFrom:
    {'contributors' -> (OrderedCollection new).
     'papers' -> (OrderedCollection new)}.
  anOrderedCollection do: [ :importedPaper |
    | paperModel |
    paperModel := Dictionary newFrom:
      {'title' -> (importedPaper at: 'title').
       'contributors' -> (OrderedCollection new)}.
    (model at: 'papers') add: paperModel.
    (importedPaper at: 'contributors') do: [ :importedContributor |
      | contributorModel |
      contributorModel := Dictionary newFrom:
        { 'name' -> (importedContributor at: 'originalName').
          'papers' -> (OrderedCollection with: paperModel) }.
      (paperModel at: 'contributors') add: contributorModel .
      (model at: 'contributors') add: contributorModel .
    ].
  ].
^model
```

This method returns a model of all papers and authors who are present in the imported papers. Each paper is represented by a *Dictionary* that contains the title of the paper and an *OrderedCollection* which contains all authors who contributed to it. Meanwhile, each author is represented by a *Dictionary* that contains the name of the author and an *OrderedCollection* with all papers that said author contributed to.

The model that the method *model:* returns is represented by a *Dictionary* with the keys *papers* and *contributors* that hold all the modeled papers and contributors.

We can now create a model from a folder of PDF files with the following code:

```
fileReference := FileSystem disk workingDirectory / 'pdfFolders'.
import := (ESTextImporter new) importFolder: fileReference.
model := (ESSimpleScienceEventManager new) model: import.
```

The first line needs to assign the reference to the folder that contains the PDF files. The second line then imports the papers into Pharo. Lastly, the third line creates a model from the imported papers.

Bibliography

- [1] D.E. Knuth. *The TeXBook*. Addison Wesley, 1986.
- [2] Leonel Merino, Dominik Seliner, Mohammad Ghafari, and Oscar Nierstrasz. Communityexplorer: A framework for visualizing collaboration networks. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, 2016.
- [3] Oscar Nierstrasz and Stéphane Ducasse. Moose—a language-independent reengineering environment. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 58:24–25, July 2004.
- [4] Andreas Strotmann, Dangzhi Zhao, and Tania Bubela. Author name disambiguation for collaboration network analysis and visualization. *Proceedings of the American Society for Information Science and Technology*, 46(1):1–20, 2009.
- [5] Dominika Tkaczyk and Łukasz Bolikowski. Extracting contextual information from scientific literature using cermine system. In *Semantic Web Evaluation Challenge*, pages 93–104. Springer, 2015.