Explorable Code Slides

Michael Fritz, Benjamin Biegel, Stephan Diehl
University of Trier, Germany
{*biegel,diehl*}*@uni-trier.de*

Abstract

Presenting source code to others is not only a typical task of computer science teachers, but also when practitioners as well as researchers in software engineering are faced with this task. Usually, classical presentation tools or source code editors are used for such presentations. However, while the former are too inflexible for the presenter to deviate from a preplanned scheme, the latter show too many irrelevant details to the audience. In this paper, we introduce Explorable Code Slides, which address both issues. We not only describe its concept and features but also report on our practical experience from using Explorable Code Slides in class and feedback gathered from our students.

1. Introduction

There are many situations where teachers, developers, or computer science researchers have to present source code. In order to do so, they most often either use the program editor of their favorite IDE or a presentation tool such as Microsoft PowerPoint. However, it should be noted that both approaches come with certain problems:

Program editor: The source code is shown in the editor view of an IDE, ideally using an increased font size. During the presentation many details of the editor and the interaction with the editor are visible to the audience. For example, file open dialogs may be shown and screen space is wasted for displaying menu bars and other views. Furthermore, presentations can only be prepared to a very limited extent in advance.

Presentation tool: Code is cut&pasted from the IDE into the slides of a presentation tool and formatted using the features of the tool. The presenter has to decide on the typically linear order and format of the presentation in advance. This premature commitment further restricts the interaction with the audience, because the presenter cannot react or adapt to the question by deviating from the "script". Preparing slides on which source code is unfolded or folded during the presentation is a tedious task. Finally, if the underlying source code changes, the slides have to be revised by hand.

Both approaches are sometimes combined in programming courses: first, the main parts of a program are presented on slides, then the program execution is demonstrated using the IDE before the program is then improved or extended by using the IDE. While this approach combines the

978-1-4673-5140-9/13 © 2013 IEEE CSEE&T 2013, San Francisco, CA, USA

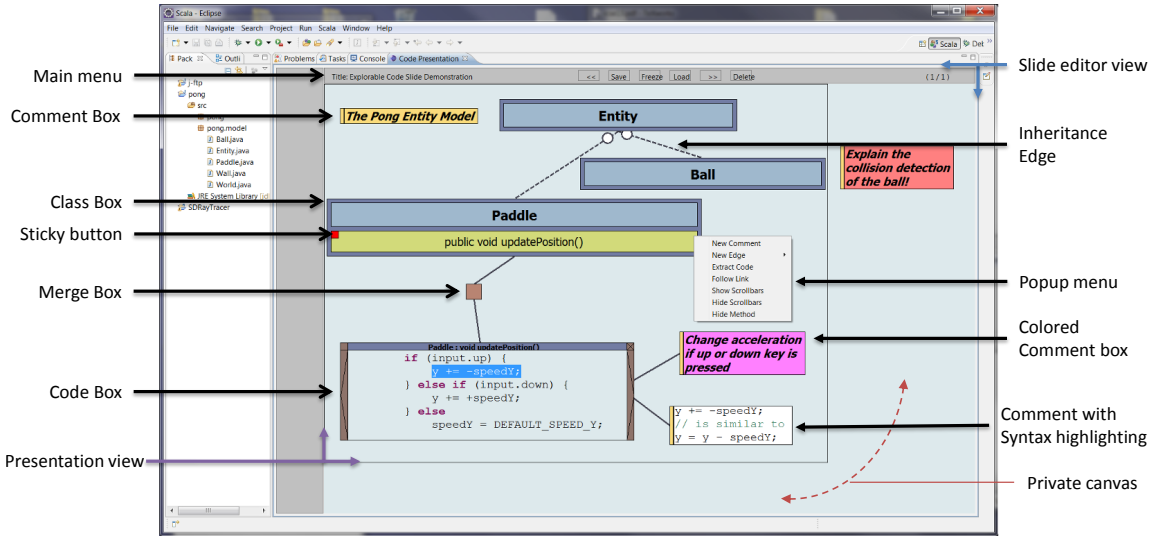


Figure 1. An overview of most elements and features in Explorable Code Slides presented in the slide editor view.

flexibility of the IDE with the possibility to prepare slides in advance, it also inherits most of the disadvantages of both approaches. Moreover, switching between the presentation tool and the editor introduces another level of complexity both for the presenter as well as for the audience.

Taking the analysis above as a starting point, we came up with a set of requirements for a presentation tool designed for source code and developed an approach called **Explorable Code Slides**, which enables the presenter

- to prepare slides in advance by using familiar presentation features;
- to interactively explore the source code at any time during the presentation;
- to hide the presenter's code search and navigation activities from the audience;
- to have full control over which elements should or should not be shown;
- to easily perform a dynamic or static walkthrough through the source code by following method calls or data types;

Finally, as Explorable Code Slides is integrated into the IDE, the presenter has full access to all features of the IDE and can thus work in a familiar environment.

2. Concepts behind Explorable Code Slides

In order to meet the requirements described above, we developed the following key concepts. Figure 1 provides an overview of most elements and features in Explorable Code Slides.

2.1. Dual View

By using a separate presentation view, which is either displayed in a separate window or on a second monitor or projector, the interactions of the presenter with the presentation tool and, in

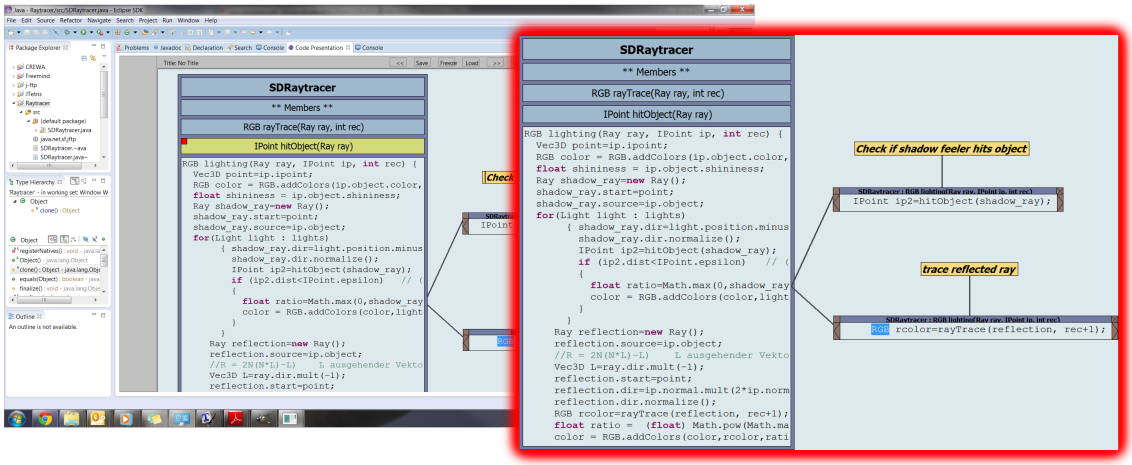


Figure 2. Dual view: The Eclipse window as seen by the presenter (left) and the presentation view as seen by the audience (right).

particular, the IDE are not visible to the audience. In other words, the presenter can switch between synchronous and asynchronous presentation modes. In the former case, all changes to the slides become immediately visible to the audience, while in the latter the presenter has to initiate the redrawing of the presentation view once he has completed his changes. Figure 2 shows both the view of the presenter as well as the presentation view.

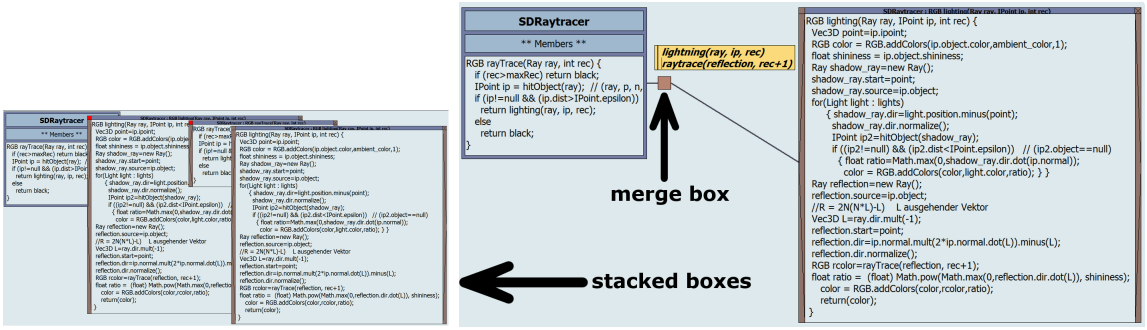
2.2. Private Canvas Area

An important and controversially discussed design decision is the spatial limitation of the canvas instead of an unlimited canvas. By following code dependencies, like method calls or class definitions, it is often important to keep the origin of such a dependency path. By using an unlimited canvas, it could be possible to pan the origin out of the visible area of the presentation. In order to gain some of the benefits of an unlimited canvas, we introduced border lines, which indicate the borders of the slide in the slide editor view. The extended canvas enables the lecturer to place elements outside of the slide prior to the session, which can then be interactively dragged into the view of the audience during the presentation. In Section 3.2 we will see, how this private canvas area can be utilized to achieve a new fluent and interactive presentation style.

2.3. The Box Model

Classes including their attributes and methods, as well as code fragments are all displayed in boxes. Lines connecting such boxes indicate different kinds of relations. These boxes can be freely arranged on the slide. During the presentation new boxes can be created. In particular, this can be done by clicking on a method or a class name in a certain code fragment. The source code of the method or class will then be shown in a new box which is connected by a line to the original box. The presenter can follow different links at the same time, which results in an expansion tree.

Explorable Code Slides provides four different types of boxes with different properties: *class box, code box, merge box,* and *comment box*.



(a) Expanded recursive call tree of a method. To fit all boxes on the slide, the class boxes were stacked on top of each other.

(b) Here the code boxes of the intermediate method calls are subsumed by a single merge box. A comment lists the method calls within the merge box.

Figure 3. Use of merge boxes to hide intermediate code boxes in a branch.

Class and Code Boxes: Classes can be dragged directly out of the IDE into the slide editor and are represented as *class boxes* on the slide. A class box is divided into multiple blocks. The first block includes the class name, while the second block contains the fields, and the subsequent blocks the methods. Thus, the structure of a class box is very similar to the representation of a class in an UML class diagram. A priority button at the upper left corner of a class box helps to remove irrelevant inner blocks. Only sticky inner blocks remain visible. A code block can be marked as sticky by clicking the sticky button which is available in each code box. Noteworthy, these buttons are only visible in the editor view and not in the presentation view. A class box can be minimized to the extent that it only shows the class name. It is also possible to manually draw inheritance edges between class boxes which is similar to those in UML class diagrams. Furthermore, in a pull down menu the presenter can select the set of methods which should be shown in the class box.

A *code box* represents a particular source code fragment which includes both methods or custom selections of source code. A code box can either be used as a freely movable box on the slide or an inner block of a class box. A code box can be expanded, minimized, or hidden. By selecting a part of the source code and dragging it out of the current code box, a new code box will be created showing the selected source code fragment. This feature can be useful when explaining long methods. Hence, it helps by splitting them into parts and putting the code boxes containing these parts side by side for further comparison.

Folding and Unfolding (Merge Boxes): By following links and by extracting source code, the presenter can quickly build large trees of class and code boxes. Unfortunately, the space of the presentation view is limited and gets quickly cluttered. Typically, the presenter is interested in explaining the leaves of the tree, while the tree itself provides the context. In order to keep the presentation view clear and the focus on the most important code fragments, it is possible to subsume multiple boxes of a branch in a *merge box*, see Figure 3. Thereby, a merge box is a place holder for multiple boxes within a branch. In general all boxes between the source (origin) box and the last box of the branch will be automatically subsumed in a merge box. By using the the sticky button, the presenter can mark boxes which should not be subsumed in a merge box. The two branch buttons are shown as triangles to the left and right of a code box and make it possible to navigate through the boxes subsumed by the merge boxes to left or right of the code box.

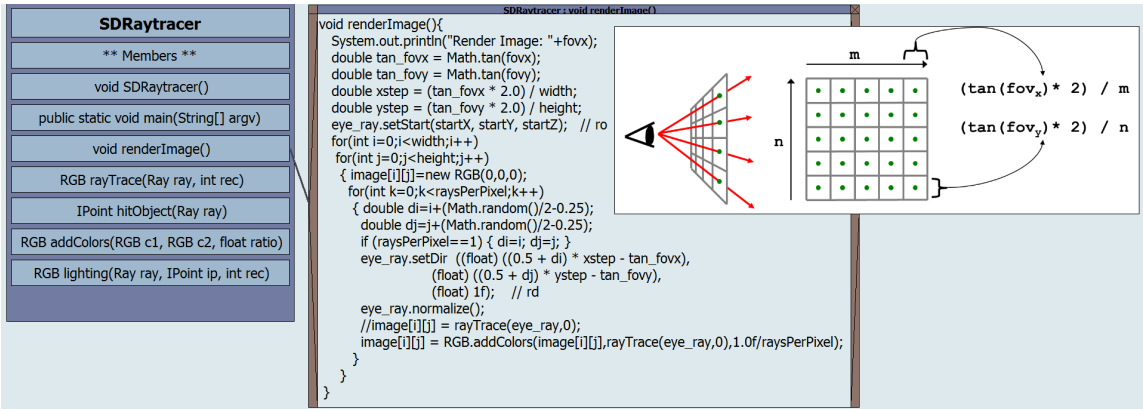


Figure 4. Combining code with illustrations.

Additional Text and Illustrations (Comment Boxes): Textual annotations or images can be added to the slides as part of *comment boxes*. A comment box can be linked to another box or to the slide itself. If boxes are moved all their related comments will also be moved relative to their parent boxes. All those comment boxes which are related to boxes in a merge box are not shown on the slide. In order to support adding alternative source code to the presentation, a comment box can also display custom source code in a monospace font by highlighting its syntax. The combination of source code and illustrations on a slide is demonstrated in Figure 4.

3. Practical Experiences

In order to get first insights into the practical benefits of our approach, we implemented a plugin for Eclipse and then applied it in different user studies. Thereby, we followed the *Rapid Iterative Testing and Evaluation method* [6] to achieve a stepwise refinement of the plugin. That is, we gathered the participants' feedback, which is taken as a basis for continuous improvements, after each application.

3.1. Usability Review

In the first application, we intended to review both the implemented concept as well as the usability of an early prototype. Hence, in order to achieve this, we asked a graduate student and three PhD students with experience in usability and human computer interaction to participate in a formative user study. Together with the authors, the participants presented source code they were already familiar with by using Explorable Code Slides. In an open discussion we then determined problems and improvements, which should be fixed and implemented before using our presentation prototype in the lectures. Besides technical suggestions like changing the color scale, font size, or text adjustment, we also came up with some conceptual improvements (e.g. private canvas area, highlighting of boxes in the focus, linking of comments, option to pre-select methods in a list, etc.).

3.2. Educational Application

After improving our prototype based on the results of the usability review, we applied Explorable Code Slides in two computer science courses—an undergraduate Java programming course and a graduate course on computer graphics. The first was an introduction to the selection sort algorithm,

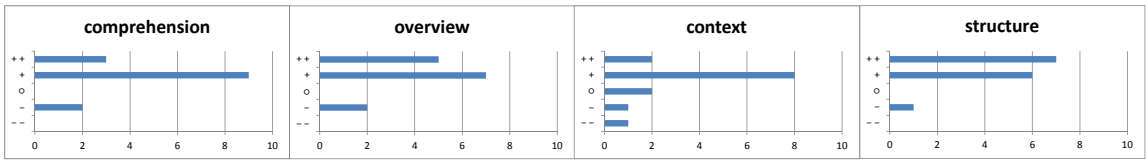


Figure 5. Survey Question: Do you agree with the fact that Explorable Code Slides helps leading to a better . . . ?

which was followed by a functional decomposition of this sorting algorithm using refactorings. While the latter introduced the theoretical principles and the implementation of a ray tracer. Each of both courses was held by one of the authors.

In order to get an impression of the advantages of Explorable Code Slides, we asked the students to participate in a study. In total, 14 students (9 undergraduates and 5 graduates) volunteered to fill in a questionnaire after the lectures. The results of the survey are summarized in the following.

Evaluation of the Concept: In the first part of the survey, we wanted to find out how Explorable Code Slides influenced the learning process. As for that, the students rated if there were any advantages in the presentation for each of the following categories: *comprehension*, *overview*, *context*, and *structure*. For the rating, we used a Likert scale consisting of five grades from *"I totally disagree" (−−)* to *"I totally agree" (++)*. As can be seen in Figure 5, in each category there were at least 10 of 14 students who thought that using Explorable Code Slides for source code presentation was beneficial. In particular, they gave high ratings for the structure and the overview of the presented slides. Obviously, there seemed to be some problems with our approach when it came to keeping track of the context.

In addition to the Likert scale, we asked the students to write down pros and cons of Explorable Code Slides using a free text form. Thus, we received specific answers to the context problems as mentioned before. Some students expressed their concerns about large source code fragments, which cannot be completely displayed on the presentation view. According to them, the context might get lost, hence, Explorable Code Slides could possibly not scale for presenting large source code fragments. Apart from that, the students wrote throughout positive comments on Explorable Code Slides. They not only considered our approach as being reasonable for source code presentation in education because it helps creating clear and structured slides but also mentioned that this concept forces them to focus on the relevant code fragments, instead of distracting them by irrelevant information during the presentation. Further, the free spatial placement of the boxes helps to emphasize the relations of code fragments. Hence, Explorable Code Slides could be well suited to introduce modular source code. For the students, however, the most important fact seems to be the explorative presentation style. In particular, they liked the interactive rearrangement of elements during the presentation, which resulted in an open and spontaneous lecture.

A further question asked the students to tick general topics, which can be presented by Explorable Code Slides in teaching. The answers are summarized in Table 1. Although we presented simple and complex algorithms during the user study, and the suitability of our approach was further rated fairly high, it is interesting that there is such a spread of opinions. Partially, this contradicts the previously discussed results. However, as there is a large approval on the topic *software design*, it seems that the students highly value the representation of relations between source code fragments.

Table 1. Survey Question: Which of the following topics can be presented by Explorable Code Slides in teaching?

Topic	Yes	No
Simple Algorithms: Control structures without outgoing dependencies.	8	6
Complex Algorithms: Code fragments with outgoing dependencies.	8	6
Software Design: Relations and interplay of classes and their methods.	10	4

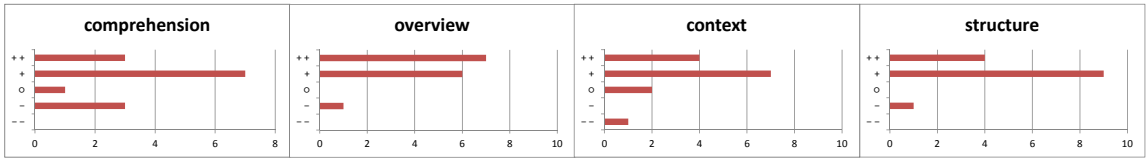


Figure 6. Survey Question: In comparison to other presentation techniques you are already familiar with, do you agree with the fact that Explorable Code Slides performs better in . . . ?

Comparison to other Presentation Techniques: Next, we aimed for an idea of the benefits of Explorable Code Slides in comparison to other source code presentation techniques. Again, for each of the categories mentioned above, the students rated if Explorable Code Slides is more appropriate than other presentation techniques they know. The results in Figure 6 show that our technique to present source code is preferred by most of the students.

The free-text answers reveal that some students prefer representations they are familiar with. Taking the comprehension ratings into account, we can assume that Explorable Code Slides was to a certain extent confusing and distracting at times. A reason for this might be boxes which are not completely created within the visible area of the presentation. If this happened the lecturer had to manually move boxes like these back into the view of the audience. Of course, effects like these only occur rarely in offline-preprared and static presentations. One student recommended to restrict the automatic creation of boxes to the visible area in order to avoid this.

Besides this, the students again gave positive feedback on the open and explorative lecture style which allows the lecturer to better respond actively to the students during the presentation. Further, the students considered the visualized relations between code fragments, represented by lines, helpful to keep track of corresponding fragments. Some students even noted that Explorable Code Slides would be an outstanding alternative to conventional static presentations to introduce source code.

Further Suggestions from the Students: The students provided also a lot of constructive criticism on how to improve our approach. First, it might be of benefit to emphasize the different kinds of source code fragments. For example, at times it is important to know whether a box represents a method or any other kind of code fragment. In order to achieve this, a student suggested to introduce different representations of the boxes depending on their corresponding code fragment.

Since the box representation in Explorable Code Slides is a new presentation style, many students requested some representations that they are familiar with. This could be done by making use of both syntax highlighting as well as monospace font for source code. This was a surprise to us, because in an earlier state we marked these features on our todo list as non-essential. However,

the user study taught us that details like these are very important in order to make a new approach successful and accepted by potential users.

Finally, the students made some general suggestions on how to improve the lecture itself. First, it might be helpful to execute every program which was introduced and to show its output. Second, after explaining several parts of a program, it might be of benefit to show the whole program during the lecture, which is necessary in order to keep track of the context.

First Experiences as Presenters: Putting ourselves in the role as lecturers, we also gained some interesting insights. Each presenter summarized his personal impressions immediately after the lecture:

> *"I liked the fact that I did not have to work out every single slide to the smallest detail, as now the source code can be explored online during the presentation. For example, I introduced how to perform a functional decomposition of a code fragment. The corresponding method was opened on the current slide and I interactively extracted some fragments into new code boxes. In order to label those new boxes, I simply dragged prepared comment boxes from the invisible area of the canvas of the presentation. At the same time, I used these comment boxes as an aide-mémoire during the presentation."* —Benjamin Biegel, Java programming course

Another helpful feature is the integration of Explorable Code Slides into Eclipse:

> *"I like the fact that once a class is dragged into the slide editor, the class box offers sufficient means to navigate through its code, extract code, and expand method calls. I mostly used the functionality of Eclipse to find the initial class for the presentation, and to compile and run the program."* —Stephan Diehl, computer graphics course

All in all, as it is possible to influence the shape of the presentation at any time, we noticed that our presentation style has changed in a positive way. Several times we extended the slides with new contents, all without breaking the flow of the presentation. Further, it is helpful to be able to import images into the slides. This makes it possible to create traditional slides, which are helpful in order to explain different theoretical facts. In both lectures, this technique was used to introduce the theoretical approach, which was explained in detail by source code examples afterwards.

3.3. Application Scenarios

So far, we have used Explorable Code Slides for presenting code in lectures only, nevertheless, we think that there are other promising applications not only in education, but also in software development in general.

Education: As has been shown in this paper, Explorable Code Slides can be used to present code in lectures without preparing all slides in advance. In addition, the slides can be made available to students, thus, they can use them to do both revise the lecture at home as well as a starting point for exploring the source code. Finally, students can also produce slides as part of their homework, e.g. to document their own code or to annotate third party code, e.g. the roles of classes and methods in an instance of a design pattern.

Table 2. Survey Question: Which of the following application scenarios are reasonable for Explorable Code Slides?

Area	Yes	No
Introducing and explaining of source code examples in a lecture.	13	1
Independent Exploration of a software system.	5	9
Interactive recapitulation at home after a lecture.	7	7
Online presentation of a self-developed software system.	4	10
Finding bugs or reviewing source code.	5	9

Software development: The most obvious application of Explorable Code Slides in software development is the ad-hoc presentation of a software system by its developer(s). In addition, it might also be used for code review and thus for finding bugs. Its source code exploration capabilities might also be useful for software comprehension tasks in group meetings, e.g. for understanding legacy or third party code. Finally, since the slides are stored within the Eclipse project and thus also in the software repository, e.g. via Subversion, they might be used as additional documentation of a software system

After the lectures, we asked our students whether they consider it necessary to apply Explorable Code Slides to some of the scenarios mentioned above, see Table 2. While 93% of them considered Explorable Code Slides a good tool for source code presentation in lectures, for the other scenarios the results were by far not as positive. We think that this is also due to the fact that they have only seen the presentation view and not the slide editor and how well it integrates into the Eclipse IDE. Furthermore, the graduate students see a greater potential for alternative application scenarios than the novices in the undergraduate course.

3.4. Threats to Validity

The feedback of the students, experts, and the lecturers makes us believe that a tool like Explorable Code Slides is in demand to enhance source code presentations, especially in teaching. Nevertheless, our evaluation also contains some threats to validity, that is, we cannot exclude that the participants of the survey consciously gave good ratings on Explorable Code Slides because they are actually our students. For example, some of them might have feared that their ratings, although anonymously gathered, might influence their mark. Further, a good presentation also strongly depends on the quality of the lecturer. Also, the difficulty of the topics could play a role in the rating of the tool. However, the students know the lecturers and attended other courses of them. Since the students really liked the ability to focus on small and relevant snippets, they nevertheless fear that there could be problems to keep track of the context by presenting large source code fragments. Further studies need to clarify whether large source code documents could be a problem in teaching presentations.

4. Related Work

The concept of placing source code in boxes and connecting these boxes by lines is not new. Approaches like Code Canvas [3], Code Bubbles [1], and Debugger Canvas [2] are meant to provide novel interfaces for programming editors, and they place code in connected boxes where the code within the boxes can be edited. In contrast to our approach, they are not meant as presentation

tools and hence, the boxes are placed on an possibly endless canvas, whereas in our approach fixed sized slides were used. With Fluid Source Code Views, Desmond and Exton [4] present an inline source code exploration technique that embeds related code from a different file into the current context on demand. Similar to our approach, the idea is to keep track of the context by placing related elements close together. However, in their approach the related elements are inlined linearly into the corresponding source code, whereas our box model allows to arrange those elements freely on the screen. The integrated teaching environment BlueJ [5] offers both a textual and a UML-like representation of the underlying source code. In contrast to our approach, each of these representations is placed in a separate view.

5. Conclusion

In this paper we introduced Explorable Code Slides—a new concept for interactive presentation of source code. We also performed some initial user studies to evaluate our new approach and to gather experience for further development. In general, the students not only accepted our new approach, but also thought that it is superior to the traditional ways of presenting source code. We also briefly discussed some future application scenarios both in education as well as in software development in general.

References

[1] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proc. of the ACM/IEEE International Conference on Software Engineering ICSE'10*, pages 455–464 vol. 1. ACM, 2010.

[2] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *ICSE*, pages 1064–1073, 2012.

[3] R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *Proc. of the ACM/IEEE International Conference on Software Engineering ICSE'10*, pages 207–210 vol. 2. ACM, 2010.

[4] M. Desmond, M.-A. D. Storey, and C. Exton. Fluid source code views. In *Proc. of the IEEE International Conference on Program Comprehension ICPC'06*, pages 260–263. IEEE, 2006.

[5] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[6] M. Medlock, D. Wixon, M. Terrano, R. Romero, and B. Fulton. Using the rite method to improve products: A definition and a case study. *Usability Professionals Association*, 2002.