Towards Enhanced Presentation-based Teaching of Programming
An Interactive Source Code Visualisation Approach

Reinout Roels, Paul Meştereagă and Beat Signer
Web & Information Systems Engineering Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050, Brussels, Belgium

Keywords: Slideware, Presentation-based Teaching, Programming.

Abstract: The teaching of programming concepts and algorithms very much depends on the mental models developed by scholars when learning how to program. There is a rich body of research on how to best teach programming. Nevertheless, many instructors follow a presentation-based approach where existing slideware such as PowerPoint or Keynote is used to show a sequential series of slides with static pieces of source code. Such a presentation-based approach based on existing slideware tools might not be optimal for the authoring as well as the delivery of programming courses. We outline how presentation-based eduction in programming can be improved by paying attention to existing research on how to best teach programming. We derive a number of requirements for more efficient source code visualisation in presentation tools and present an architecture as well as an extensible prototype for enhanced presentation-based teaching of programming. The presented interactive source code visualisation plug-in for the MindXpres presentation tool can be seen as a step towards enhancing existing slideware in order to achieve a more efficient and interactive teaching of programming concepts and algorithms. The ultimate goal of the presented approach is to present source code in a way that reinforces a user's mental model and thereby increases the knowledge transfer of presentations delivered in programming courses.

1 INTRODUCTION

The teaching of programming concepts and algorithms is a fundamental aspect of any Computer Science and Engineering degree. However, grasping the concepts taught in programming courses is far from trivial and has been proven to be a challenge for both students as well as teachers (Jenkins, 2001b; Bennedsen and Caspersen, 2007; Gomes and Mendes, 2007; Jenkins, 2002; Lahtinen et al., 2005; McCracken et al., 2001). Research from the early 1980s highlights the importance of mental models when learning how to program (Mayer, 1981). As defined by Mayer (Mayer, 1981), a mental model is *"a mental representation of the components and the operating rules of the system"* and the completeness of this representation may vary. An incomplete representation that differs from the actual characteristics of the system results in an incomplete understanding of how the computer works and will cause the novice programmer to have difficulties in writing correct programs (Ma et al., 2007). This is further confirmed by Milne and Rowe (Milne and Rowe, 2002) who state that students who are not able to create a mental model of the program execution do not have the ability to comprehend what is happening to the program in memory. Hence the importance of students being able to retell the learned concepts in their own words was also first brought up by Mayer (Mayer, 1981). It is widely accepted that by having access to a more complete mental model of the system, the learning and practising of programming can be achieved in a more effective way (Cañas et al., 1994).

Given the importance of such a mental model, it is not surprising that researchers aim to develop tools and methods in the form of visual aids for reinforcing the mental model of students (Ma et al., 2007; Smith and Webb, 1995). In a procedural programming language, the program becomes a sequential process. This process is represented by various changes of states after an expression has been executed. Therefore, Mayer (Mayer, 1981) states that a possible solution for providing an effective mental model is to use visuals and show to the user the changes in state—such as variable changes—while the program is executed. In terms of teaching methods, Jenkins (Jenkins, 2001a) argues that the main role of a teacher in programming courses should be the one of a motiva-

Roels R., Meştereagă P. and Signer B..
Towards Enhanced Presentation-based Teaching of Programming - An Interactive Source Code Visualisation Approach.
DOI: 10.5220/0005445300980107
In *Proceedings of the 7th International Conference on Computer Supported Education* (CSEDU-2015), pages 98-107
ISBN: 978-989-758-107-6

tor. In many other areas of computing the teacher is mainly a communicator of information. However, the teaching of programming based on just presenting information such as syntax and structure in a lecture is not enough as it is not immediately clear how states change and thus there is a lack of contribution to a student's mental model.

Nevertheless, the majority of programming courses are at least partially taught via lectures accompanied by slide decks which is not in line with the research in the domain of teaching how to program that has been mentioned before. In fact, these slide decks often form a major part of the study material. To make matters worse, slides that have been created by slideware such as PowerPoint or Keynote are a particularly unsuitable medium for presenting source code. As argued by Tufte (Tufte, 2003), slide decks have evolved from physical counterparts such as photographic slides or transparencies for overhead projectors and therefore also share the limitations of these physical media types. Content is presented in a strictly linear way, is fairly static and spatially restricted by the boundaries of the physical slides. In addition to the consequences on knowledge transfer during a lecture, these tools also impose a number of issues during the authoring phase by a presenter who would like to show some source code. In programming environments, source code is usually indented and colour coded via syntax highlighting in order to improve the readability. However, when source code is copy and pasted into a presentation, this formatting is often lost and presenters are required to manually format the code. Furthermore, even simple examples of algorithms result in lengthy blocks of source code and spatial restrictions make code less understandable. Due to these spatial restrictions, the presenter is forced to spread their code examples over multiple slides. In addition, they have to jump back and forth as programming concepts such as methods, conditional statements or loops cause the program to be executed in an order that is different from how it is written down.

We introduce an approach to present source code in a way that reinforces a user's mental model and thus helps to increase the knowledge transfer of presentations delivered in programming courses. In addition, our solution allows presenters to include source code in their presentations without the hassle usually associated with existing slideware tools. We start by discussing some related work in Section 2. Based on the existing body of work and some of the shortcomings of current solutions, we derive and formulate a number of requirements for a more efficient source code visualisation in presentation tools in Section 3. We then detail our proposed solution in Section 4 and present the technical details of our prototype which has been implemented as a plug-in for the MindXpres presentation tool (Roels and Signer, 2013; Roels and Signer, 2014b) in Section 5. Some concluding remarks are provided in Section 6.

2 RELATED WORK

In the context of presentation tools, there exists little to no academic work trying to improve upon the issues associated with the presentation and visualisation of source code. At authoring time, one can see that state-of-the-art presentation tools do not make any attempt to support the authoring and integration of source code. As mentioned before, the indentation and syntax highlighting of source code is lost when copy and pasting from the programming environment to a presentation tool such as PowerPoint. Common workarounds include the use of command line tools such as pygmentize[1] or web-based tools like ToHTML[2] in order to convert source code to a representation that preserves the formatting when copy and pasted (e.g. HTML or RTF). While this approach addresses the issue of manual formatting, it remains a tedious workaround.

When broadening our view beyond the domain of presentations, we can find research that builds on the principles outlined in the previous section. In all cases these are stand-alone desktop applications that use visualisations to help users build a mental model of a program. One of the earliest tools is the Bradman tool (Smith and Webb, 1995) for the C programming language. It mainly relies on showing state changes after the execution of each line of code and an evaluation of the tool revealed an improved understanding of the code by its users (Smith and Webb, 2000). Another solution is the VIP tool (Virtanen et al., 2005) for a subset of the object-oriented C++ programming language. While the VIP tool also focusses on visualising state changes, it distinguishes itself by making the concept of pointers and references—which is considered to be a difficult concept to grasp for students—more understandable. Jeliot 3 (Moreno et al., 2004) is a tool for visualising the object-oriented Java language. Given the nature of Java, the tool also visualises the objects and their relationships in an UML-like notation in addition to state changes while the program is executed. However, an evaluation of Jeliot highlighted that the animations were hard to interpret and apply for students,

[1] http://pygments.org/docs/cmdline/
[2] http://tohtml.com

making the evaluation inconclusive (Moreno and Joy, 2007). Another notable feature of Jeliot is the extensible visualisations, allowing potential new third-party visualisations to be added at a later stage.

The notional machine (Berry and Kölling, 2013) is another recent tool for visualising Java programs which bases itself on the work of Boulay (Boulay, 1986). While being similar to Jeliot, the notional machine intentionally limits the stepping granularity for state changes to the level of method invocations and method returns rather than to single statements. Additionally, the notional machine also allows methods to be invoked interactively (on demand) in contrast to the other tools where the execution is only possible in the order of the logical execution flow.

Finally, there is a category of tools that focusses on a specific aspect of program execution. For instance, RGraph (Sa and Hsin, 2010) is a solution that generates a static visualisation of recursion graphs for Java programs in order to help students with understanding recursion.

Note that we intentionally limited ourselves to the tools built for supporting students during the learning process. There are plenty of commercial products that visualise code characteristics such as the amount of lines or dependencies, performance metrics or editing history, but these solutions serve an entirely different purpose than the aforementioned tools.

3 REQUIREMENTS

Based on a detailed analysis of the related work presented in the previous section, we derived a number a requirements for more efficient source code visualisation in presentation tools. While these requirements overlap with the requirements for stand-alone desktop tools, the use in the context of a lecture requires some further thought. For instance, the typical traditional lecture mainly consists of a unidirectional flow of information as students are not as involved as, for example, in lab sessions.

R1: Automatic Indentation and Syntax Highlighting. A first step towards making source code understandable is to make it more readable. Indentation and syntax highlighting are well-established methods to improve the readability of source code as they help to interpret scope and syntactic structures. Additionally, the indentation and syntax highlighting of source code in a presentation makes the code similar to what students are used to in their programming environments. However, in contrast to existing practices in presentation tools, the formatting of code should not be a burden to the presenter and should be done automatically by the presentation tool.

R2: Efficient Navigation of Source Code. When explaining the working of a piece of source code, it is necessary to display the code as part of a presentation. However, even simple programs consist of more lines of code than would fit on a single slide. Additionally, programs rarely execute sequentially and may jump back and forth between different pieces of code, not necessarily in the order in which they were written. This forces presenters to jump back and forth between slides making it difficult for both the audience as well as the presenter to follow the program flow.

R3: Visualise the Working of the Code. From related work we learn that the mental model of a program can be built much easier when accompanied by visual aids. In addition to showing the code of the program, the tool might for instance display state changes or illustrate concepts such as pointers or recursion in order to make it clearer what is actually happening when the program executes. The idea of visualising source code in a dynamic way is supported by recent studies showing that the use of dynamic media brings measurable improvements in knowledge transfer over the use of static media (Holzinger et al., 2008).

R4: Integration in Presentation Tools. As slide decks are often used during lectures, it makes sense to integrate the interactive visualisations directly into our presentation rather than relying on a stand-alone application. If the interactive source code visualisation is not integrated into the presentation tool, the presenter is forced to switch between applications which takes time and breaks the flow of the presentation.

R5: Extensible Support for Multiple Languages. Even though there are a few more commonly used programming languages in the list of all existing languages, there is no consensus on what language to teach in introductory programming courses. While each of the stand-alone tools presented in Section 2 only focusses on a single language, we believe that a tool for use in presentations should be able to deal with more than one language. By supporting only a single language, the tool would automatically be excluded from being used in the larger share of lectures that use other languages. Additionally, we claim that the set of supported programming languages should be extensible by third parties instead of limiting

ourselves to a fixed set of languages.

R6: Extensible Visualisation Choices. It has been shown that graphical representations of programming concepts have an important role in the construction of a mental model (George, 2000; Velázquez-Iturbide and Pérez-Carrasco, 2010). However, different visualisations are needed for different scenarios. Not only do programming languages have different characteristics (e.g. object-oriented versus procedural) but also the topic may influence the type of visualisations required for the program. Related work shows that customised visualisations help with the teaching of specific concepts such as memory pointers or recursion (Dann et al., 2001). It therefore makes sense to allow the presenter to select the desired visualisation apart from just showing some source code. Additionally, it should be possible that the pool of visualisations can be extended, especially when considering the previous requirement R5.

R7: Interactive Program Execution. Next to visualising a program, the execution might also be made more interactive and controllable by the presenter. For instance, the presenter might want to show how the same program reacts to different kinds of input, or they may wish to execute different parts of the program based on different scenarios or feedback from the students.

4 INTERACTIVE SOURCE CODE VISUALISATION

Based on the requirements presented in the previous section, we now introduce our approach towards presenting source code in a more accessible and efficient way. In order to fulfil requirement R4, it is obvious that our solution should integrate into an existing presentation tool. As the most basic feature the presenter should be able to include any piece of source code in a presentation and the tool should present it in a readable manner. From a technological standpoint there is no reason why a presentation tool could not automatically handle formatting issues such as indentation and syntax highlighting. A user can either explicitly specify the language or simple techniques such as Bayesian filters can be applied to automatically detect the programming language of a piece of source code. Since a programming language's syntax is formally defined, the parsing and syntax highlighting is hardly a challenge. Nevertheless, to the frustration of many presenters this simple feature is not present in current presentation tools and should therefore be provided by our tool as demanded by requirement R1. In order to break free of spatial restrictions, we suggest that source code should be scrollable if it does not fit on a single slide rather than presenting isolated chunks of code spread across slides. This allows the presenter to illustrate source code more coherently as it is easier for audience members to grasp the bigger picture. Making the code scrollable contributes to the navigation of the source code and therefore also addresses requirement R2.

While these aesthetic improvements enhance knowledge transfer, related work indicates that the visualisation of program execution may be one of the most important techniques to help students in building a mental model of a program. Therefore our solution should not only display the static source code but one should also be able to step through the execution of the program in forward or backward order. This does not only help in illustrating the program flow, but also state changes can be shown simultaneously to highlight how each line of code influences the state of the program as described in requirement R3. Since the presenter does not always want to step through the program execution from the very beginning to the end, it also makes sense to provide a means of manually jumping to the relevant parts of the source code.

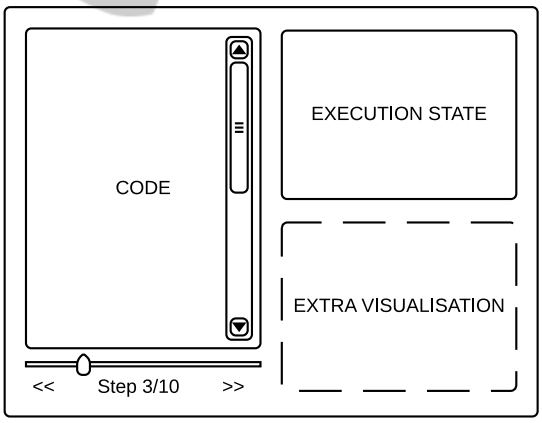

Figure 1: Source code visualisation mock-up.

In order to fulfil requirement R5, an interactive source code visualisation solution should not limit itself to a single programming language. The tool should be able to handle different languages and the resulting code visualisation and execution should work in the same way, regardless of the language. However, the execution and interpretation of programs is language dependant and therefore it is impossible to offer a generic implementation that is guaranteed to work for all languages. We address this challenge by offering modular language support.

Language-specific functionality should be bundled together as a module with a predefined interface so that the tool can select the corresponding module based on the detected language. The language-specific module is then responsible for interpreting the execution of the program and translating it to a generic representation that is understood by the visualiser. This way, the tool can support a wide variety of languages regardless of technical differences and even allows the set of supported languages to be extended by third parties.

Finally, in addition to the visualisation of the source code and state changes we deem it meaningful to provide some further optional graphical visualisation. As discussed in Section 2, visualisations have been developed to provide a better understanding of concepts such as memory pointers or recursion. These extra visualisations should also be implemented as interchangeable modules in order that they can be extended. They can make use of the generic execution representation provided by the language-specific modules and are thus language independent. The visualisation module gets the execution data such as state changes and method invocations in the common format and does not have to deal with language-specific details. Note that this also means that visualisation modules can be reused for different languages since, for example, recursion is a concept available in many languages.

A mock-up of our proposed solution is shown Figure 1. Based on the real estate available on a slide, the left half of the slide is dedicated to the visualisation of the source code. Note that the source code is properly indented, syntax highlighted and scrollable if necessary. The slider below the code allows the presenter to quickly move to a particular point in the execution and the buttons underneath allow them to go through the code one step at a time, either forwards or backwards. The right-hand side is dedicated to visualisations that adapt as the presenter steps through the code. The upper right part shows the state of relevant variables whereas the lower right part can optionally be used for context-specific visualisations.

While the goal is that our tool should do as much as possible automatically, there are also cases where a presenter might want to configure the tool beforehand. For instance, in larger programs it makes no sense to visualise the changes of every single variable. In these cases, the presenter may choose to select those variables that contribute to the understanding of a program and the rest will not be displayed. The presenter might also want the execution to start at a specific point instead of having to manually find the right spot they want to discuss. Furthermore, the presenter may also choose which additional visualisation to use or decide to not show any additional visualisation at all and use the full width of the slide for displaying the source code.

5 IMPLEMENTATION

In this section, we present our implementation of the concepts discussed in Section 4. Before describing the implementation of the prototype, we outline the overall architecture of our interactive source code visualisation for the MindXpres presentation tool.

5.1 Architecture

As briefly mentioned in Section 4, special measures need to be taken in order to support multiple languages. The main reason is that in order to fulfil requirement R3 we need to execute the provided source code to extract events, such as state changes or method invocations, from its execution. Unfortunately, this process is different for each programming language which makes it impossible to provide an all-in-one solution. As detailed before, we bundle language-specific functionality in interchangeable modules making it possible to add new languages.

The architecture chosen to support this functionality is shown in Figure 2. When source code is given to our tool by the presenter, the language can automatically be detected. The tool then searches its collection of language modules for the detected language. In the case that no matching module is found, the tool limits itself to just displaying the source code without any interactive features. However, if a matching module is found, it is passed the source code. The language module is then responsible for extracting the relevant information from the running program and translating it to a generic representation that is understood by the visualisation engine. One of the benefits of isolating language-specific features is that the modules can make use of existing applications and libraries instead of having to implement everything from scratch. For example, we found existing debuggers to be particularly useful. A debugger is a tool that examines a running application and offers functionality to provide insights about the program flow and for finding unwanted behaviour in the form of bugs. Debuggers are hard to use and they make no real effort to reinforce a mental model (Smith and Webb, 1995), but their output, a so-called execution trace, can be turned into something more meaningful by our tool. Nevertheless, debuggers are standalone applications dedicated to a specific language only and different debuggers produce output in different formats. Therefore,

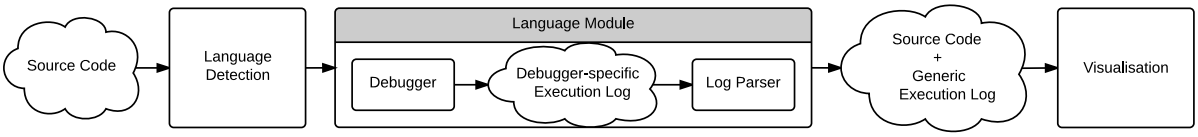

Figure 2: Architecture for extensible language support.

when the language module invokes the debugger and gets an execution trace, it also translates the resulting trace into a generic format ensuring that the output of each language module has the same format. This generic execution log is then transferred to the visualisation engine together with the source code so it can display the source code and provide additional interactive functionality such as the visualisation of state changes while the presenter steps through a piece of source code.

5.2 Generic Execution Log

As explained before, the generic execution log is the key to supporting multiple languages in an extensible manner. We have chosen the JavaScript Object Notation (JSON), a lightweight data interchange format for representing the execution log. For each programming language, a language module translates the language-specific execution log into this JSON-based representation. This means that the visualisation tool only needs to be able to process the generated JSON format and does not need to be aware about the specifics of a particular programming language.

```
1 int sum = 0;
2 for(int i = 0; i < 2; i++){
3   sum = add(sum, i);
4 }
```

Listing 1: A small C program.

From the language-specific execution logs the language module needs to extract events such as variable definitions, variable state changes as well as function invocations. For example, Listing 2 shows the JSON output resulting from the execution of the C program shown in Listing 1.

The C programming language is a purely procedural programming language but in order to support object-oriented languages the details specific to objects can also be expressed in the generic log format. This includes, for instance, method invocations and changes in object fields. While we could have used the same representation as for function invocations and variable changes, there are languages such as C++ that can have both functions and methods and therefore a separate representation is needed.

```
1  [ {"line": 1, "type": "VarDefinition",
2     "details": {"name": "sum",
3                 "value": "0"}},
4    {"line": 2, "type": "VarDefinition",
5     "details": {"name": "i",
6                 "value": "0"}},
7    {"line": 3, "type": "FunctionCall",
8     "details": {"name": "add"}},
9    {"line": 3, "type": "StateChange",
10    "details": {"name": "sum", "old": "0"
11                , "new": "0"}},
12   {"line": 2, "type": "StateChange",
13    "details": {"name": "i", "old": "0"
14                , "new": "1"}},
15   {"line": 3, "type": "FunctionCall",
16    "details": {"name": "add"}},
17    ...
```

Listing 2: JSON execution log for Listing 1.

Listing 4 shows the JSON execution log generated for the Java program illustrated in Listing 3.

```
1 Person person = new Person("John");
2 person.setAge(19);
```

Listing 3: A small Java program.

Note that a specific line of code may need multiple entries in the execution log. For instance, a line of code may define a new variable, invoke a method and use the returned value to set its state. Even though they all occur on the same line of code, the execution log should contain separate entries for each of these events. The visualisation tool may combine them into a single step for the visualisation, but at least it has access to the finer details in case certain visualisation plug-ins should need them.

```
1  [
2    {"line": 1, "type": "VarDefinition",
3     "details": {"name": "person",
4     "initialValue": "null"}},
5    {"line": 1, "type": "Constructor",
6     "details": {"class": "Person"}},
7    {"line": 1, "type": "StateChange",
8     "details": {"name": "person",
9     "old": "null", "new": "Person"}},
10   {"line": 2, "type": "MethodCall",
11    "details": {"name": "setAge",
12                "object": "person"}},
13   {"line": 2, "type": "ObjStateChange",
14    "details": {"name": "person.age",
15    "old": "0", "new": "19"}}
16 ];
```

Listing 4: JSON execution log for Listing 3.

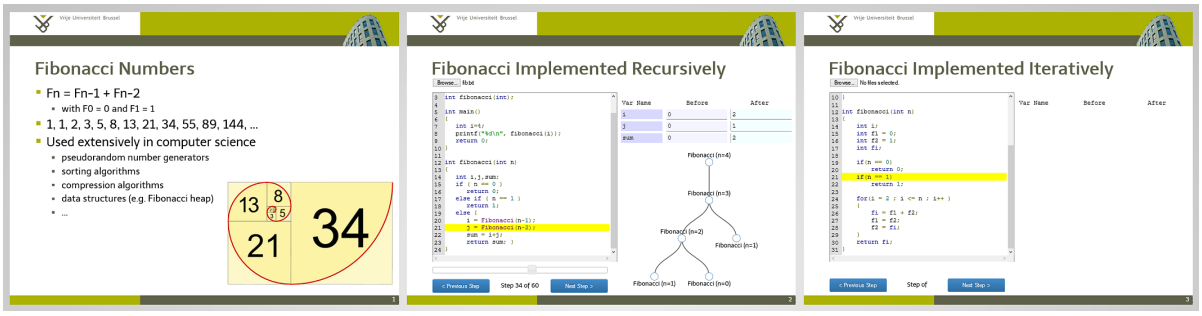

Figure 3: A MindXpres presentation with the source code plug-in.

5.3 MindXpres Source Code Plug-in

Our interactive source code visualisation prototype has been implemented as a plug-in for the MindXpres presentation tool (Roels and Signer, 2014b). MindXpres has been developed to overcome the limited extensibility of well-known slideware tools such as PowerPoint or Keynote and to offer a rapid prototyping platform for novel presentation ideas. The motivation behind this is that although PowerPoint offers an API (application programming interface) for creating extensions, it still enforces the usage of linear sequences of slides with relatively static content. Therefore it is often not possible to extend PowerPoint with radically new functionality. The highly modular MindXpres architecture allows any component to be replaced and new components and functionality can easily be added. For instance, users may choose to use a plug-in that visualises content using a zoomable user interface (ZUI) or can use a plug-in that visualises the same content in a classic linear fashion like in existing slideware. The core MindXpres engine provides various abstractions that allows plug-in creators to focus on their ideas instead of having to reimplement the basics every time. For instance, the graphics engine provides functionality related to the visualisation of content which drive features such as the ZUI and interactive rich media visualisation plug-ins. The communication engine allows instances of a MindXpres presentation to form networks which allows plug-ins to communicate across devices, enabling plug-ins for various audience-driven functionality such as polls, quizzes or screen mirroring (Roels and Signer, 2014a). MindXpres uses HTML5 and related technologies for enhanced portability and plug-ins are written entirely in JavaScript. Although a graphical editor is under development, MindXpres presentations are currently defined in a XML-like declarative language similar to the LaTeX language used for text documents. The reasoning behind this is also similar; let the user focus on content and let the tool worry about the layout and styling. While MindXpres comes with a default set of plug-ins for basic components such as images, bullet lists, videos or slides, it is easy to add new plug-ins for new content types. Plug-ins also extend the vocabulary used in the MindXpres document format. More specifically, a plug-in can add new XML tags for usage in the document format. A plug-in that introduces new tags then takes responsibility for visualising content placed within these tags.

```
1  <presentation>
2    <slide title="Fibonacci Numbers">...</slide>
3    <slide title="Fibonacci - Recursive">
4      <code>
5        int fibonacci(int n)
6        {...}
7      </code>
8    </slide>
9    <slide title="Fibonacci - Iterative">
10     <code source="fib_it.c"></code>
11   </slide>
12 </presentation>
```

Listing 5: MindXpres presentation in XML.

We have realised our approach by creating a code visualisation plug-in for MindXpres that introduces the code tag to the vocabulary. Source code can be included in a presentation in two ways. Either the presenter refers to an external file containing source code via an attribute of the code tag, or the presenter just pastes the code between code tags. Listing 5 shows a shortened snippet of a MindXpres presentation that uses both ways to include source code. The resulting presentation can be seen in Figure 3.

When the MindXpres document format is compiled into a portable presentation, a MindXpres plug-in can be invoked if it contains compile-time triggers. In this case, the code plug-in will detect the language and let the correct language module generate the generic execution log. The log is then bundled in the presentation together with the code plug-in that

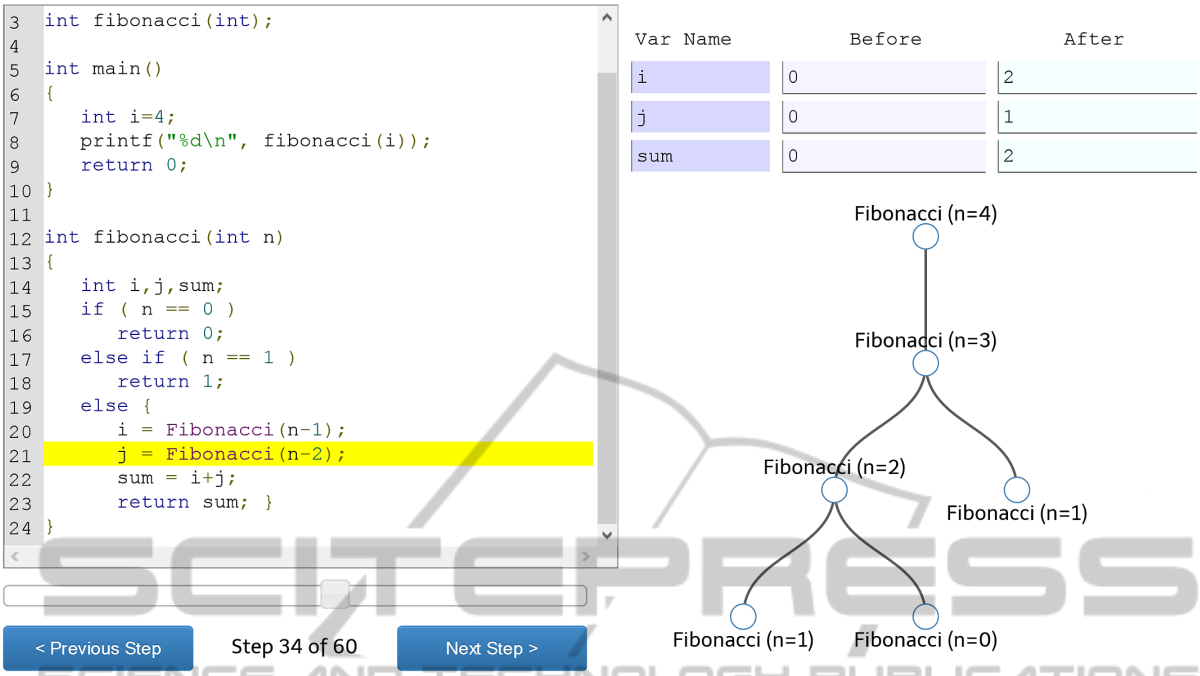

Figure 4: MindXpres source code plug-in example.

will visualise it at run-time (when the final presentation is opened for viewing). Because MindXpres plug-ins are written in JavaScript, we are free to use some of the powerful existing libraries offering the corresponding functionality. For code formatting and syntax highlighting we use Google's prettify[3] library. Furthermore, the plug-in uses the D3[4] visualisation library for some of its optional visualisations. For this prototype implementation we have implemented two language modules, namely one for C and one for Java. For the creation of the execution traces, the C module uses the GDB[5] debugger while the Java version uses JDB, a debugger included with the Java Development Kit[6]. Creating a language module requires some programming but the provided abstractions make the process fairly straightforward. The language module itself is implemented as a folder containing at least two files. A manifest file provides some metadata and specifies the programming language the module can process. The second file contains JavaScript code that implements a single method which accepts source code as input and returns a generic execution log. This code is executed if the compiler detects that a given piece of source code was written in the programming language mentioned in the manifest file. Because the compiler is based on Node.js, the JavaScript code can make use of libraries and binaries placed alongside the two required files. In most cases, it is sufficient for a language module to include an existing debugger, have it create an execution log and translate this log into the generic execution format. Note that there are alternative ways how a language module might obtain an execution log, such as the direct interpretation of the given code via the JavaScript code.

An example of the MindXpres source code plug-in in use is shown in Figure 4. In this case a recursive implementation of the Fibonacci function is illustrated. The highlighted line indicates what line of code is being executed in the current step. As mentioned before, the buttons can be used to go forwards or backwards in the execution and the presenter may also use the slider to jump to a particular point of interest. On the right-hand side the state changes for the variables `i`, `j` and `sum` are shown. This includes their old value (Before) and the new value (After) that was assigned to them at that point of execution. As the Fibonacci function is a classic example for demonstrating recursion, we included the recursion tree as the optional extra visualisation. The tree shows a history of recursive function calls up to that point making it clearer what has happened in the previous steps. In this case the viewer can see that the recursion is performed depth-first and has just finished the backtrack-

[3]https://code.google.com/p/google-code-prettify/
[4]http://d3js.org
[5]http://www.gnu.org/software/gdb/
[6]https://www.oracle.com/java/

ing phase for the calculation of the third Fibonacci number. A new branch is about to be added under the top node for the calculation of the second Fibonacci number, which will be added to the result of the existing branch to form the fourth Fibonacci number.

While the example in Figure 4 shows purely procedural code, the same visualisation can be used for object-oriented code. For instance, the execution of the code presented in Listing 3 would first show a state change in the variable person, from null (not initialised) to a new instance of a Person. The second line would then result in a state change in person.age from 0 to 19. If more details are required, the recursion tree could be replaced with another optional visualisation that shows the object and relevant changes in more detail. Similarly, state changes in arrays can also be shown. In the current implementation the complete array is shown in the `Before` and `After` columns. However, if the presented code is centered around arrays—for example in a sorting algorithm—it makes sense to replace the recursion tree visualisation with another more efficient array visualisation making use of colours and animations to show how array elements are moved in each step.

6 DISCUSSION & CONCLUSIONS

The presented interactive source code visualisation prototype provides a framework for future extension and experimentation with innovative forms of presentation-based teaching of programming. So far we have implemented language modules for the C and Java programming language but the architecture allows for the addition of new languages with minimal effort. In its current form, our tool is suitable for presenting code written in the major programming languages but there are some languages that will need further investigation. For instance, functional programming languages such as Haskell avoid state changes and mutable data which implies that most of our additional visualisations will be useless for these types of languages. Furthermore, there exist declarative programming languages based on logic (e.g. Prolog) where programs are not constructed out of sequentially executed instructions but rather consist of rules that are queried and triggered in a different manner. Programs in these two categories of programming languages benefit less from the proposed way of navigating and visualising source code. Further investigation is therefore needed in order to also help students in enhancing their mental model for these types of languages.

We currently allow presenters to navigate longer pieces of source code by scrolling. However, by letting the visualisation plug-in reason over the presented source code, further enhancements might be made to navigate through larger pieces of source code in a more efficient way. For instance, the presenter could click on a function or method call to jump to the definition of the function or method if the tool would be aware of these concepts. The current version of our source code visualisation plug-in further only allows the presenter to step through the code as it has been included in the presentation but no modifications are possible. As a future improvement, we would also like to make the execution of programs more interactive. For instance, the presenter might want to execute the same algorithm multiple times with different start parameters to illustrate the resulting effect. The possibility to modify values at any point during the execution of an algorithm is also something to be investigated in future work.

The MindXpres presentation tool also provides abstractions to implement features commonly found in audience response and classroom systems (Roels and Signer, 2014a). By involving the students through active learning we can increase the effectiveness of the visualisation of algorithms even more as discussed by Hundhausen et al. (Hundhausen et al., 2002). A first step could be to allow the students to interactively navigate through the source code which might later be extended to small exercises that force the students to reason over the presented program.

So far we have only evaluated the technical side of the proposed architecture by implementing language modules for the C as well as Java programming language. While parts of the presented functionality is based on earlier research in the domain of how to teach programming concepts that we have applied in the domain of presentation tools, we plan to do a user evaluation with the presented interactive source code visualisation MindXpres plug-in. Based on available teaching material from universities around the world, we can infer that many teachers of programming courses still use static source code examples spread over multiple slides managed by traditional slideware solutions. We see our approach as a step towards enhancing the omnipresent presentation-based teaching of programming by providing better tools for the authoring of source code slides as well as for the interactive presentation of code examples. Last but not least, we hope that other researchers might be inspired to investigate new forms of presentation-based teaching of programming concepts and algorithms that go beyond the presentation of a series of slides containing static source code snippets.

REFERENCES

Bennedsen, J. and Caspersen, M. E. (2007). Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin*, 39(2):32–36.

Berry, M. and Kölling, M. (2013). The Design and Implementation of a Notional Machine for Teaching Introductory Programming. In *WiPSE 2013, 8th Workshop in Primary and Secondary Computing Education*, pages 25–28. ACM.

Boulay, B. D. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73.

Cañas, J. J., Bajo, M. T., and Gonzalvo, P. (1994). Mental Models and Computer Programming. *International Journal of Human-Computer Studies*, 40(5):795–811.

Dann, W., Cooper, S., and Pausch, R. (2001). Using Visualization to Teach Novices Recursion. 33(3):109–112.

George, C. E. (2000). Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models. In *PPIG 2012, 12th Annual Workshop of the Psychology of Programming Interest Group*, pages 33–44.

Gomes, A. and Mendes, A. J. (2007). Learning to Program - Difficulties and Solutions. In *ICEE 2007, International Conference on Engineering Education*, pages 53–58.

Holzinger, A., Kickmeier-Rust, M. D., and Albert, D. (2008). Dynamic Media in Computer Science Education; Content Complexity and Learning Performance: Is Less More? *Educational Technology & Society*, 11(1):279–290.

Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290.

Jenkins, T. (2001a). Teaching Programming - A Journey From Teacher to Motivator. In *2nd Annual Conference of the LSTN Center for Information and Computer Science*.

Jenkins, T. (2001b). The Motivation of Students of Programming. 33(3):53–56.

Jenkins, T. (2002). On the Difficulty of Learning to Program. In *3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A Study of the Difficulties of Novice Programmers. 37(3):14–18.

Ma, L., Ferguson, J., Roper, M., and Wood, M. (2007). Improving the Viability of Mental Models Held by Novice Programmers. In *11th Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts*. Springer.

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys (CSUR)*, 13(1):121–141.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *ITiCSE-WGR 2001, Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, pages 125–180. ACM.

Milne, I. and Rowe, G. (2002). Difficulties in Learning and Teaching Programming - Views of Students and Tutors. *Education and Information Technologies*, 7(1):55–66.

Moreno, A. and Joy, M. S. (2007). Jeliot 3 in a Demanding Educational Setting. *Electronic Notes in Theoretical Computer Science*, 178:51–59.

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing Programs With Jeliot 3. In *AVI 2014, Working Conference on Advanced Visual Interfaces*, pages 373–376. ACM.

Roels, R. and Signer, B. (2013). An Extensible Presentation Tool for Flexible Human-Information Interaction. In *Demo Proceedings of BCS HCI 2013, 27th BCS Conference on Human Computer Interaction*, page 59. British Computer Society.

Roels, R. and Signer, B. (2014a). A Unified Communication Platform for Enriching and Enhancing Presentations with Active Learning Components. In *ICALT 2014, 14th IEEE International Conference on Advanced Learning Technologies*, pages 131–135. IEEE.

Roels, R. and Signer, B. (2014b). MindXpres: An Extensible Content-driven Cross-Media Presentation Platform. In *WISE 2014, 15th International Conference on Web Information System Engineering*, pages 215–230. Springer.

Sa, L. and Hsin, W.-J. (2010). Traceable Recursion with Graphical Illustration for Novice Programmers. *InSight: A Journal of Scholarly Teaching*, 5:54–62.

Smith, P. A. and Webb, G. I. (1995). Reinforcing a Generic Computer Model for Novice Programmers. In *ASCILITE 1995, 7th Australian Society for Computer in Learning in Tertiary Education*.

Smith, P. A. and Webb, G. I. (2000). The Efficacy of a Low-level Program Visualization Tool for Teaching Programming Concepts to Novice C Programmers. *Journal of Educational Computing Research*, 22(2):187–216.

Tufte, E. R. (2003). *The Cognitive Style of PowerPoint: Pitching Out Corrupts Within*. Graphics Press.

Velázquez-Iturbide, J. Á. and Pérez-Carrasco, A. (2010). InfoVis Interaction Techniques in Animation of Recursive Programs. *Algorithms*, 3(1):76–91.

Virtanen, A. T., Lahtinen, E., and Järvinen, H.-M. (2005). VIP, a Visual Interpreter for Learning Introductory Programming with C++. In *5th Koli Calling Conference on Computer Science Education*, pages 125–130.