An Interactive Source Code Visualisation Plug-in for the MindXpres Presentation Platform

Reinout Roels, Paul Meştereagă and Beat Signer

Web & Information Systems Engineering Lab
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussels, Belgium
{rroels,bsigner}@vub.ac.be

Abstract. Nowadays, the teaching of programming concepts and algorithms is often conducted via slideware such as PowerPoint or Keynote, with the instructor going through a sequential series of slides showing static pieces of program code. As outlined in this paper, such a slideware-based approach has its limitations in terms of the authoring as well as the delivery of content for a programming course. Nevertheless, there is a rich body of research on how to best teach programming concepts and algorithms where it has been shown that this process very much depends on the mental models developed by scholars when learning how to program. Based on this existing body of research, we derived a number of requirements for an improved source code visualisation and presentation in slideware tools. We present an interactive source code visualisation plug-in for the MindXpres presentation platform, which addresses these requirements and introduces a number of innovative concepts for an interactive visualisation of source code. Based on two concrete examples showing how our solution can be used for the teaching of recursion by means of a recursion tree or to explain sorting algorithms by using animation, we illustrate the extensibility and flexibility of the presented interactive source code visualisation approach. Ultimately, the presented solution should help in reinforcing a student's mental model about a presented algorithm and improve the knowledge transfer of presentations delivered in programming courses.

Keywords: slideware, presentation-based teaching, programming

1 Introduction

The teaching of programming concepts and algorithms forms a fundamental part 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 [16,2,11,17,20,23]. Research from the early 1980s highlights the importance of mental models when learning how to program [22]. As defined by Mayer [22], 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 [21]. This is further confirmed by Milne and Rowe [24] 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 [22]. 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 [5].

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 [21,32]. 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 [22] states that a possible solution for providing an effective mental model is to use visuals and to show the user the changes in state—such as variable changes—while the program is executed. In terms of teaching methods, Jenkins [15] argues that the main role of a teacher in programming courses should be the one of a motivator. In many other areas of computing the teacher is mainly a communicator of information. However, the teaching of programming based on only presenting information such as syntax and structure in a lecture is not sufficient as it is not immediately clear how states change and 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 earlier. 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 [35], slide decks have evolved from their physical counterparts including 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, it 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 since programming concepts such

as methods, conditional statements or loops cause the program to be executed in an order that differs 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 thereby 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 Sect. 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 Sect. 3. We then detail our proposed solution in Sect. 4 and present the technical details of our prototype which has been implemented as a plug-in for the MindXpres presentation platform [28,30] in Sect. 5. In Sect. 6, we describe two different use cases of the plug-in. Finally, some concluding remarks are provided in Sect. 7.

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 effort 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 like 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 pasting (e.g. HTML or RTF). Another popular workaround is to simply take a screenshot of the source code in the development environment and to include it in the presentation. However, the inclusion of a screenshot often results in blurry or pixelated text. While these approaches address the issue of manual formatting, they remain tedious and often suboptimal workarounds.

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 [32] 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 [33]. Another solution is the VIP tool [37] 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 [26] is a tool for visualising the object-oriented Java language. Given the

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

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, making the evaluation inconclusive [25]. Another notable feature of Jeliot is the extensible visualisation mechanism, allowing potential new third-party visualisations to be added at a later stage.

The notional machine [3] is another recent tool for visualising Java programs which bases itself on the work of Boulay [4]. 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. Similarly, UUhistle [34] also offers an interactive coding mode for the Python programming language, adapting the visualisations in real-time as the user makes changes to the source code. Also jGRASP [6] defines itself as an application for understanding a program through visualisation while the user is writing the code. Finally, there is a category of tools that focusses on a specific aspect of program execution. For instance, RGraph [31] is a solution that generates a static visualisation of recursion graphs for Java programs in order to help students with the understanding of 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.

In conclusion, projects such as CodeWitz [19] show that there is a clear need for better teaching tools in computer science. However, in the context of presentations, presenters are often limited to two choices. Either they have to work around the limitations of existing slideware tools or they have to use one of the standalone applications mentioned above, switching between their presentation and the external application on demand. Besides not being beneficial for the flow of the presentation, most of the standalone tools only focus on a very specific aspect of programming (e.g. recursion or memory management) and are made for a specific programming language. This implies that a programming course may require more than one of these tools to illustrate all relevant concepts or that a tool for the programming language used in the course might not even exist.

3 Requirements

Based on a detailed analysis of the related work presented in the previous section, we derived a number of 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 since 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 [13].

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 [12,8]. While each of the stand-alone tools presented in Sect. 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 [10,36]. 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 [7]. 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 Towards 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.

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 programming 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 programming 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 programming 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 Sect. 2, different 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 a 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 in Fig. 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

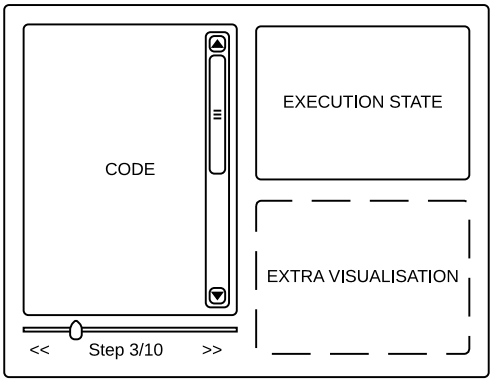

Fig. 1. Source code visualisation mock-up

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 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

We now present our implementation of the concepts discussed in the previous section. Before describing the technical details of our prototype implementation, we outline the overall architecture of our interactive source code visualisation solution for the MindXpres presentation platform.

5.1 Architecture

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

The architecture chosen to support this extensibility for new programming languages is shown in Fig. 2. When source code is given to our tool by the presenter, the language can automatically be detected. In our implementation we used a Naive Bayes classifier to infer the language. This technique works particularly well for this purpose since each programming language's syntax consists

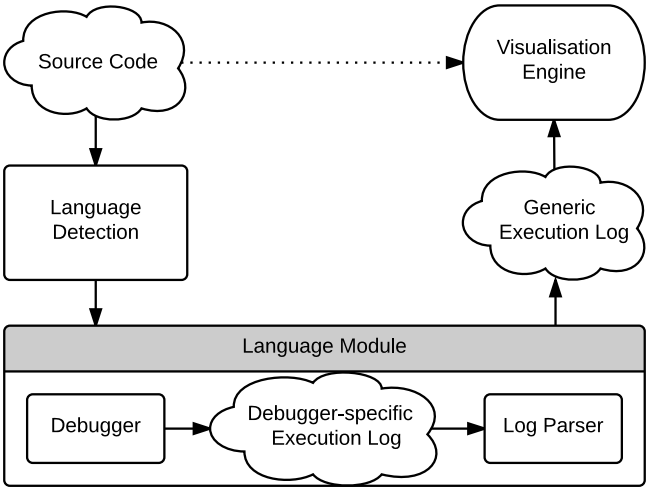

Fig. 2. Architecture for extensible language support

of some reserved keywords (e.g. `new` or `import`) that are either unique to the language or drastically reduce the amount of potential candidates. When the code has been analysed 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 additional 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 for 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 [32], 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 one specific language only and different debuggers produce output in different formats. Therefore, 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 original source code in order that the visualisation engine 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. Since the visualisation engine works with

the generic execution log it does not need to have knowledge about the original language which implies that visualisations can be reused for different languages.

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 1.1. 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 1.2 shows the JSON output resulting from the execution of the small C program shown in Listing 1.1.

```
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 1.2. Generic execution log (in JSON) for Listing 1.1

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 to 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. Listing 1.4 shows the JSON execution log generated for the Java program illustrated in Listing 1.3.

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

Listing 1.3. 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 1.4. Generic execution log (in JSON) for Listing 1.3

5.3 MindXpres Source Code Plug-in

Our interactive source code visualisation prototype has been implemented as a plug-in for the MindXpres presentation platform [30,27]. 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 application programming interface (API) 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 they can use a plug-in that visualises the same content in a classic linear fashion as known from existing slideware. The core MindXpres engine provides various abstractions, which allows plug-in creators to focus on their innovative ideas instead of having to reimplement the basic functionality every time. For instance, the graphics engine provides functionality related

to the visualisation of content with 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 and enables plug-ins for various audience-driven functionality such as polls, quizzes or screen mirroring [29]. Furthermore, the communication engine allows the easy integration of hardware such as clickers, digital pens or gesture capturing devices (e.g. Leap Motion[3]) and is also able to direct the stream of captured events to other relevant MindXpres instances.

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 easily possible 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 also takes responsibility for visualising content placed within these tags.

```
<presentation>
  <slide title="Fibonacci Numbers">
    <bulletlist>
      <item>Fn = Fn-1 + Fn-2</item>
      ...
    </bulletlist>
    <image source="fib.jpg"/>
  </slide>
  <slide title="Fibonacci Implemented Recursively">
     <code>
        int fibonacci(int n)
        {
          if (n == 0)
            return 0;
          else if (n == 1)
            return 1;
          else {
            return fibonacci(n-1) + fibonacci(n-2);
          }
        }
     </code>
  </slide>
  <slide title="Fibonacci - Iterative">
     <code source="fib_it.c"></code>
  </slide>
</presentation>
```

Listing 1.5. MindXpres presentation in XML

We have realised our proposed interactive source code visualisation approach by creating a code visualisation plug-in for MindXpres that introduces the `code` tag to the vocabulary. The plug-in provides two ways to include source code in a presentation document. Either the presenter uses an attribute of the `code`

[3] https://www.leapmotion.com

tag to refer to an external file containing source code, or the presenter just pastes the code between `code` tags. Listing 1.5 shows a shortened snippet of a MindXpres presentation that uses both ways to include source code and the resulting presentation can be seen in Fig. 3.

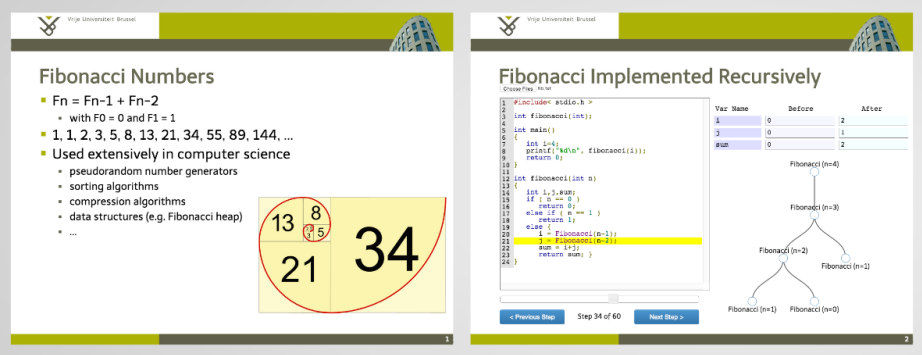

Fig. 3. A MindXpres presentation with the embedded source code plug-in

When the MindXpres document format is compiled into a portable presentation, a MindXpres plug-in can be invoked if it contains compile-time triggers for the content that it is responsible for. In this case, the code plug-in will use a compile-time trigger to be notified when source code is encountered in the presentation. The plug-in will then detect the language and let the correct language module generate the generic execution log. This means that the corresponding language module is only invoked once, namely at compile time. The resulting log is then bundled in the presentation together with the code plug-in that is going to perform the run-time visualisation 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 relevant functionality. For code formatting and syntax highlighting we use Google's prettify[4] library. Furthermore, the plug-in uses the D3[5] 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[6] debugger while the Java version uses JDB, a debugger included with the Java Development Kit[7].

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.

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

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 source code plug-in detects that the presentation contains source code that was written in the programming language mentioned in the manifest file. Because the MindXpres compiler is based on Node.js, the JavaScript code can make use of existing libraries and even 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. However, note that there are alternative ways how a language module might obtain an execution log. For instance, a language module could also directly implement a basic interpreter in JavaScript and generate the generic execution log without the use of external tools.

6 Technical Evaluation

After describing the implementation in the previous section, we now detail two different use cases of the source code visualisation plug-in. As part of the initial prototype, we implemented two extra visualisations. A first visualisation is used to display the recursion tree when executing a recursive algorithm. The other visualisation uses animation to show how a list of numbers is processed during the execution of a sorting algorithm. Note that both visualisations can be used for any of the supported languages.

6.1 Teaching Recursion by Means of a Recursion Tree

Recursion is an important but far from trivial programming concept. However, it has previously been shown that visualisations can be beneficial when teaching recursion [7]. For this reason, we have chosen to develop a recursion visualisation as part of our technical evaluation. A common application of recursion in programming is to make a method or function call itself, possibly multiple times, to compute a smaller part of the task that it was given. One of the standard examples to illustrate this is the recursive implementation for calculating the Fibonacci sequence. In the Fibonacci sequence each number is the sum of the previous two numbers ($Fib(n) = Fib(n-1) + Fib(n-2)$). In other words, to calculate the n^{th} number in the sequence we need to know the numbers at position $n-1$ and $n-2$, with the base case $Fib(0) = 0$ and $Fib(1) = 1$. This translates particularly well to most programming languages as it can be implemented as a function that calls itself to calculate the previous two numbers, just like the mathematical definition.

Figure 4 shows the MindXpres source code plug-in in action for an implementation of the Fibonacci function in the C programming language. 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 `n`, `i`, `j` and

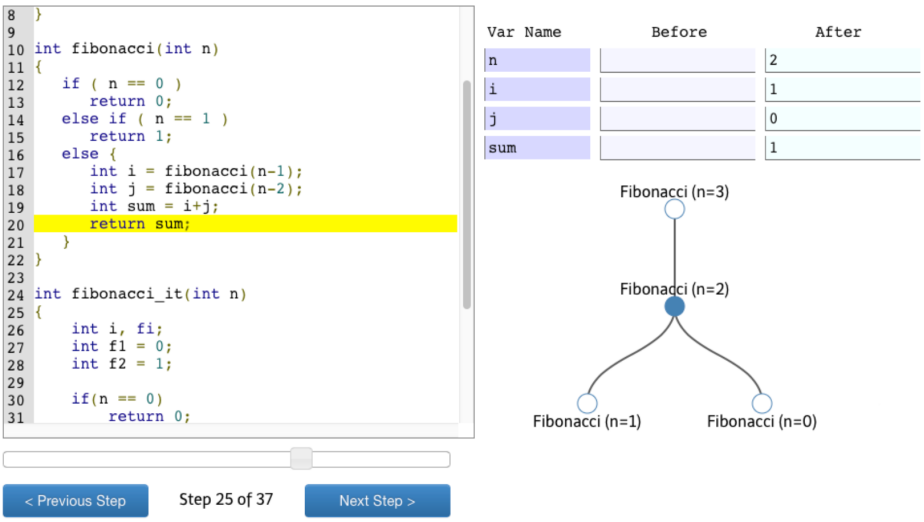

Fig. 4. MindXpres source code plug-in with recursion tree

`sum` are shown. This includes their old value (`Before`) and the new value (`After`) that was assigned to them at that point of execution. The recursion tree shows a history of recursive function calls up to that point making it clearer what has happened in the previous steps. The blue dot indicates which Fibonacci number we are currently calculating. In this case, the reader can see that the program started with $Fib(3)$, but to get the result it had to calculate $Fib(2)$ and $Fib(1)$. The blue dot shows that it is currently finishing the calculation of $Fib(2)$. The tree also makes it clear that in order to calculate $Fib(2)$ it first had to calculate $Fib(1)$ and $Fib(0)$ and the results were stored in the variables `i` and `j` respectively. Hence the variable `sum`, in this case the result of $Fib(2)$, equals 1. As the recursion is performed in depth-first order and is currently backtracking, a new branch is about to be added under the top node for the calculation of $Fib(1)$. Its result will be added to the result of the left branch to form the third Fibonacci number.

While the example in Fig. 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 1.3 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. Similarly, a recursion tree could be built based on method calls instead of function calls.

6.2 Teaching Sorting Algorithms by Using Animation

In addition to recursion, the visualisation of sorting algorithms also has been proven to be beneficial when teaching [1,9]. While changes in arrays can be shown in the upper right section dedicated to state changes, we implemented a

second extra visualisation to show array manipulations more clearly by making use of colours, arrows and animation.

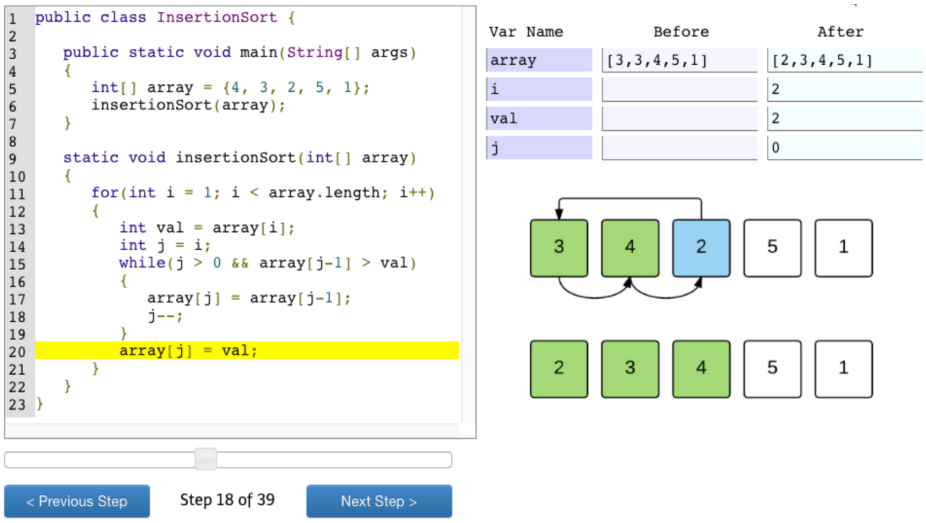

Fig. 5. MindXpres source code plug-in with array visualisation

In Fig. 5, an implementation of the well-known insertion sort [18] sorting algorithm is shown. In short, the insertion sort works by maintaining a sorted subarray on the left side and adds new elements one by one while keeping the subarray sorted. As the presenter steps through the code, the manipulation of the array is visualised. The visualisation shows both the current state of the array as well as the changes that lead to that current state (shown above the current state). The green elements represent the part of the array that is already sorted. To give an example, Fig. 5 shows us the step where the number 2 has been added to the sorted part. In order to have space to insert the number 2, the numbers 3 and 4 had to be moved to the right as indicated by the arrows. Note that the arrows visualise multiple steps leading up to the insertion of the number 2. In this case, our extra visualisation was configured to accumulate steps until the variable `i` changes its state.

7 Discussion and Conclusion

We have introduced an interactive source code visualisation plug-in for the MindXpres presentation platform. Apart from the two discussed examples for the teaching of recursion by means of a recursion tree and the explanation of sorting algorithms by using animation, the presented solution serves as a framework for additional future source code visualisations. It further enables the experimentation with alternative innovative forms for teaching programming concepts and

algorithms based on more interactive presentations. While we currently realised two language modules for the C and Java programming languages, with minimal effort it is possible to add support for additional programming languages. The presented interactive source code visualisation plug-in currently focuses on some major imperative and object-oriented programming languages. However, with further investigation of new visualisations, also non-imperative programming languages can be supported in a future version of the interactive MindXpres source code visualisation plug-in. For example, most of the presented execution state and extra visualisation would not be useful for functional programming languages such as Haskell which avoid state changes and mutable data. The presented approach for navigating source code is also not appropriate for source code that has been written in a declarative programming language (e.g. Prolog), since these programs consists of rules that are queried and triggered rather than a number of sequentially executed instructions. While the presented interactive source code visualisation does currently not support these alternative programming paradigms, in the future we might investigate how alternative visualisations can help students in enhancing their mental model for functional or declarative programming languages.

The presented source code visualisation is already an improvement when discussing larger pieces of source code since an instructor can scroll through the source code and no longer has to spread source code over multiple slides. Nevertheless, the navigation could be further improved by analysing the source code in order to add some enhancements. When the presenter, for example, clicks on a function or method call, an enhanced version of the interactive source code visualisation plug-in might jump to the function or method definition as known from most existing integrated development environments (IDEs). A limitation of the presented solution is that a presenter can only step through the source code as it has been included in the presentation with no possibility to modify the source code while delivering a presentation. In the future, we would therefore like to make the execution of programs more interactive and, for instance, allow the presenter to execute the same algorithm multiple times but with different start parameters in order to illustrate the effect of varying parameters. Furthermore, the functionality of the presented solution could be further enhanced by offering the presenter the possibility to modify values at any point during the execution and visualisation of an algorithm.

While we presented our solution mainly from the perspective of the presenter, the MindXpres presentation platform provides a number of abstractions for implementing features that are commonly found in audience response and classroom systems [29]. As discussed by Hundhausen et al. [14], the effectiveness of the visualisation of algorithms can be further increased by involving the students more closely through active learning. While a MindXpres presentation with our source code visualisation can already be used as interactive study material after the lecture, the next step would be to also include the audience during the lecture. Students might be given the chance to interactively navigate through the source code and it would no longer be the sole responsibility of the teacher

to control the navigation. Of course such a collaborative source code navigation tool might also be beneficial in exercise session where students would have to reason over a presented program in some form of group work.

The technical side of the proposed extensible architecture has been evaluated by implementing two different language modules for the C as well as Java programming languages. While parts of the presented functionality of our interactive source code visualisation solution is based on earlier research in the domain of how to best teach programming concepts, in the future we also plan to do a user evaluation of the discussed interactive source code visualisation plug-in for MindXpres. When having a look at the teaching material from various universities from all around the world, one can identify that many teachers of programming courses still use traditional slideware solutions with source code that is often spread over multiple slides. Our solution can be seen 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. Finally, we hope that our new way of presenting source code in a more interactive manner might inspire other researchers to also investigate new forms of presentation-based teaching solutions for programming concepts and algorithms that go beyond simply showing a set of slides with pieces of static source code.

References

1. Baecker, R.M.: Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science, chap. 24, p. 369–381. MIT Press (1998)
2. Bennedsen, J., Caspersen, M.E.: Failure Rates in Introductory Programming. ACM SIGCSE Bulletin 39(2), 32–36 (2007)
3. Berry, M., Kölling, M.: The Design and Implementation of a Notional Machine for Teaching Introductory Programming. In: WiPSE 2013, 8th Workshop in Primary and Secondary Computing Education. pp. 25–28. ACM (2013)
4. Boulay, B.D.: Some Difficulties of Learning to Program. Journal of Educational Computing Research 2(1), 57–73 (1986)
5. Cañas, J.J., Bajo, M.T., Gonzalvo, P.: Mental Models and Computer Programming. International Journal of Human-Computer Studies 40(5), 795–811 (1994)
6. Cross, II, J.H., Hendrix, T.D.: jGRASP: An Integrated Development Environment with Visualizations for Teaching Java in CS1, CS2, and Beyond. Journal of Computing Sciences in Colleges 23(1), 5–7 (2007)
7. Dann, W., Cooper, S., Pausch, R.: Using Visualization to Teach Novices Recursion. In: ACM SIGCSE Bulletin. vol. 33, pp. 109–112. ACM (2001)
8. Dewar, R.B., Schonberg, E.: Computer Science Education: Where Are the Software Engineers of Tomorrow? Crosstalk: the Journal of Defense Software Engineering 21(1), 28–30 (2008)
9. Furcy, D., Naps, T., Wentworth, J.: Sorting out Sorting: The Sequel. In: ITiCSE 2008, 13th Annual Conference on Innovation and Technology in Computer Science Education. pp. 174–178. ACM (2008)
10. George, C.E.: 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. pp. 33–44 (2000)

11. Gomes, A., Mendes, A.J.: Learning to Program - Difficulties and Solutions. In: ICEE 2007, International Conference on Engineering Education. pp. 53–58 (2007)
12. Guo, P.: Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. BLOG@CACM, http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext (July 7, 2014)
13. Holzinger, A., Kickmeier-Rust, M.D., Albert, D.: Dynamic Media in Computer Science Education; Content Complexity and Learning Performance: Is Less More? Educational Technology & Society 11(1), 279–290 (2008)
14. Hundhausen, C.D., Douglas, S.A., Stasko, J.T.: A Meta-Study of Algorithm Visualization Effectiveness. Journal of Visual Languages & Computing 13(3), 259–290 (2002)
15. Jenkins, T.: Teaching Programming - A Journey From Teacher to Motivator. In: LTSN-ICS 2001, 2nd Annual Conference of the LSTN Center for Information and Computer Science (2001)
16. Jenkins, T.: The Motivation of Students of Programming. ACM SIGCSE Bulletin 33(3), 53–56 (2001)
17. Jenkins, T.: On the Difficulty of Learning to Program. In: LTSN-ICS 2002, 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences. vol. 4, pp. 53–58 (2002)
18. Knuth, D.E.: The Art of Computer Programming, Volume 3: (2nd edition) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
19. Kujansuu, E., Tapio, T.: Codewitz - An International Project for Better Programming Skills. In: EdMedia 2004, World Conference on Educational Media and Technology. pp. 2237–2239. AACE (2004)
20. Lahtinen, E., Ala-Mutka, K., Järvinen, H.M.: A Study of the Difficulties of Novice Programmers. ACM SIGCSE Bulletin 37(3), 14–18 (2005)
21. Ma, L., Ferguson, J., Roper, M., Wood, M.: Improving the Viability of Mental Models Held by Novice Programmers. In: ECOOP 2007, 11th Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts. Springer (2007)
22. Mayer, R.E.: The Psychology of How Novices Learn Computer Programming. ACM Computing Surveys (CSUR) 13(1), 121–141 (1981)
23. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: 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. pp. 125–180. ACM (2001)
24. Milne, I., Rowe, G.: Difficulties in Learning and Teaching Programming - Views of Students and Tutors. Education and Information Technologies 7(1), 55–66 (2002)
25. Moreno, A., Joy, M.S.: Jeliot 3 in a Demanding Educational Setting. Electronic Notes in Theoretical Computer Science 178, 51–59 (2007)
26. Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M.: Visualizing Programs With Jeliot 3. In: AVI 2014, Working Conference on Advanced Visual Interfaces. pp. 373–376. ACM (2004)
27. Roels, R., Meşthereagă, P., Signer, B.: Towards Enhanced Presentation-based Teaching of Programming: An Interactive Source Code Visualisation Approach. In: CSEDU 2015, 7th International Conference on Computer Supported Education. pp. 98–107. SCITEPRESS (2015)

28. Roels, R., Signer, B.: An Extensible Presentation Tool for Flexible Human-Information Interaction. In: Demo Proceedings of BCS HCI 2013, 27th BCS Conference on Human Computer Interaction. p. 59. British Computer Society (2013)
29. Roels, R., Signer, B.: A Unified Communication Platform for Enriching and Enhancing Presentations with Active Learning Components. In: ICALT 2014, 14th IEEE International Conference on Advanced Learning Technologies. pp. 131–135. IEEE (2014)
30. Roels, R., Signer, B.: MindXpres: An Extensible Content-driven Cross-Media Presentation Platform. In: WISE 2014, 15th International Conference on Web Information System Engineering. pp. 215–230. Springer (2014)
31. Sa, L., Hsin, W.J.: Traceable Recursion with Graphical Illustration for Novice Programmers. InSight: A Journal of Scholarly Teaching 5, 54–62 (2010)
32. Smith, P.A., Webb, G.I.: Reinforcing a Generic Computer Model for Novice Programmers. In: ASCILITE 1995, 7th Australian Society for Computer in Learning in Tertiary Education (1995)
33. Smith, P.A., Webb, G.I.: 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 (2000)
34. Sorva, J., Sirkiä, T.: UUhistle: a Software Tool for Visual Program Simulation. In: Koli Calling 2010, 10th Koli Calling International Conference on Computing Education Research. pp. 49–54. ACM (2010)
35. Tufte, E.R.: The Cognitive Style of PowerPoint: Pitching Out Corrupts Within. Graphics Press (July 2003)
36. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A.: InfoVis Interaction Techniques in Animation of Recursive Programs. Algorithms 3(1), 76–91 (2010)
37. Virtanen, A.T., Lahtinen, E., Järvinen, H.M.: VIP, a Visual Interpreter for Learning Introductory Programming with C++. In: 5th Koli Calling Conference on Computer Science Education. pp. 125–130 (2005)