# VISUALIZING DATA STRUCTURES IN AN E-LEARNING SYSTEM

Michael Striewe and Michael Goedicke
*Specification of Software Systems, University of Duisburg-Essen, Essen, Germany*

Keywords: Computer-aided assessment, Self-training, Data visualization, Data structures.

Abstract: In introductory courses on programming it is important to discuss algorithms at the syntactic level in terms of program code as well as at the semantic level in terms of affected data structures. While single visualizations for examples during a lecture are easy to create for a teacher, students have to create visualizations of their exercises on their own, which is time consuming and may lead to wrong results. To avoid this, we present an extension for an e-learning and e-assessment framework that allows mass visualization of programming exercises. This way, learning basic skills in data structures is supported, because each student gets individual visualizations in an easier way.

## 1 INTRODUCTION

Many introductory courses to computer science combine teaching basic programming skills in object-oriented programming languages with basic knowledge about algorithms and data structures. This is reasonable, because all three topics are related very closely. Nevertheless they require thinking in entirely different scopes: program code is a static, precise description of behaviour; algorithms are the more abstract idea for this behaviour and objects forming a data structure are subjects to be changed by this behaviour. In a more focussed view we can state that program code gives us a syntactical view on algorithms, while data structures and their changes give us a semantical view. Both views are important and thus it is usual to support lectures not only by giving source code examples but also by showing visualizations of data structures.

Consequently exercises in computer science should also be supported not only by giving feedback to the program code, but also by visualizing objects and data structures produced by the program code created by students. These visualizations of each individual solution would be an important link between contents of the course and feedback for exercises. Moreover, they can help students in debugging their program code, because an image of the data structure may give them some useful information without any need for inserting debug statements into their program code.

In general, two different approaches exist for creating and using visualizations of data structures in the classroom: the use of visual debuggers, either as standalone tools (Gaylard and Zeller, 2009) or integrated into an development environment, or the use of development environments specialized on teaching and visualization (Klling et al., 2003; Zndorf, 2009). Both approaches are reasonable to some extent, but also come with major drawbacks.

First of all, all approaches require the students themselves to get active in order to generate visualizations for their exercise solutions. In an optimal case, students are motivated and talented in using those tools and consequently get some benefit from using them. In an avarage case one can expect at least some students having problems in using these tools, resulting in different and possibly misleading visualizations even for similar solutions. In the worst case, this can frustrate any learning success if the visualization does not show the expected result since students cannot distuingish whether they have written wrong program code or have used the tool in a wrong way.

The second problem is complexity. On the one hand, professional visual debugging tools offer many options that are not necessary in introductory courses, making these tools confusing to first year students. Readership skills in interpreting diagrams are known to be an important problem (Petre, 1995) and thus visualizations for exercises should be as close to the visualizations used in the respective lectures as possible. On the other hand, development environments specialized for teaching may be too



Striewe M. and Goedicke M. (2010).
VISUALIZING DATA STRUCTURES IN AN E-LEARNING SYSTEM.
In *Proceedings of the 2nd International Conference on Computer Supported Education*, pages 172-179
DOI: 10.5220/0002773801720179

limited for some kind of exercises, narrowing the teacher in designing exercises for the course. The same applies to students, who may be limitied in creating creative solutions.

As an alternative solution, we present in this paper an extension for an e-learning and e-assessment framework that is able to visualize data structures from programming assignments submitted by students. The extension is designed in that way that a teacher can define how data structures should be visualized and any exercise solution submitted to the system will be visualized according to this settings. The expected benefit is that complexity is handled by the teacher this way, while students can concentrate on solving their exercises and analyzing the visualizations that are generated automatically.

The remainder of this paper is organized as follows: section 2 elaborates on different visualization techniques for data structures and argues which of them are appropiate for our scenario. Section 3 presents our implementation and section 4 shows how it works in practice. Conclusions and future work are assembled in section 5.

# 2 VISUALIZATION OF DATA STRUCTURES

Three main aspects have to be taken into account when selecting techniques suitable for visualizing data structures in an e-learning system: First, general layout and design questions have to be answered. For example, data structures like trees or lists already hint towards layout contraints regarding the order of elements in an image. Second, the use case of an e-learning scenario requires special features like displaying objects that are missing in a data structure because they have been deleted unintentionally. Third, visualizations of data structures are used in the context of algorithms, producing sequences of changed structures over the time. Thus displaying these changes in an appropiate manner is an additional requirement.

## 2.1 General Layout

Data structures in object-oriented programming languages can in general be considered as attributed, directed graphs. Thus, each node of a graph represents an object, each arc between two nodes represents a reference from one object to another, i.e. a class attribute of a non-primitive type and each node attribute represents a class attribute of a primitive type.

According to the most common way of displaying objects in UML object diagrams (OMG, 2004), we decided to depict each node by a rectangular shape and show the name of its object type at its head. Different to UML object diagrams, we chose to display all attribute names inside the shape instead of displaying the names of non-primitive types as arc labels outside the shape. Note, that it is a heuristic decision to assume this solution as more appropiate for teaching purposes and not a crucial feature of the general approach.

More important is the impact of data structure semantics on the layout. Whereever possible, the layout should reflect properties of the data structure. Two main data structures have been considered at first: lists and trees. In both cases it is necessary to keep a direction of reading, which can be achieved by using algorithms for drawing layered graphs (Sugiyama et al., 1981; Kaufmann and Wagner, 2001). This way the entry point of a data structure, which is the “head” element in case of lists and the “root” element in case of trees, is displayed as the top-most object, while all subsequent objects follow below in the same order as in the data structure. Decisions have to be made if data structures are combined, e.g. a tree structure where each element is the head of a list. In this case, the direction of reading for the first data structure can be vertical, while the elements of the second data structure are arranged in horizontal direction.

Since the visualizations are intended to be used in an e-learning system running as an online service on the web, some additional requirements regarding the display environment have to be considered. First, space is limited by screen sizes and resolutions and bandwith may be limited when many students are accessing one server at the same time. So the layout algorithms should be able to create compact arrangements of objects without wasting much space, resulting in images of small file size that are readable on a screen without scrolling. Second, layouts using three dimensions instead of printing two dimensional images would require browser plugins or similar facilities to allow 3d navigation while viewing the visualization. Thus we chose to stick to 2d images in order to make using visualizations as simple as possible.

## 2.2 Special Features for Teaching

Beside the general layout constraints discussed in the previous section, some special requirements emerge from our e-learning scenario. The UML specification and thus also UML tools used by professionals do

in general not define or use any kind of coloring inside their object diagrams. However, at least three possibilities exist how coloring can be used to make object diagrams more comprehensible for first year students.

First, colors can be assigned to each object class, thus coloring all objects of the same type in the same color, making it easy to distungish them from objects of other types. This way of coloring is useful in situations, where many different classes are in action and using objects of the wrong type is expected to be a frequent fault made by students. This way of coloring is of little use if only one or two different classes are in action, e.g. in a list data structure with elements of the same type, where wrong references between objects are expected to be the most common fault. In these cases, the second method of coloring may be beneficial: the coloring of class attributes of non-primitve types. For example, in a binary tree, the left child of a node could always be colored yellow, while the right child is always colored blue.

A third way of using colors is to mark objects that have been removed from a data structure, e.g. by displaying them half-transparent or surrounded with a red border. This way, even a static visualization can give information about changes that happened to the data structure. Furthermore, it helps students to distuingish between a situation in which an object is missing because it has never been created and a situation in which an object is missing because it has been deleted. Being able to distuingish between these situations is assumed to be a crucial point in understanding behaviour of algorithms acting on data structures.

### 2.3 Sequences of Visualizations

An even better understanding of behaviour can be gained if not only one static visualization is used, but a sequence of static visualizations. This can simply be achieved by taking snapshots of the program state several times during run time. Despite this simplicity, this raises some additional requirements for the layout of each visualization.

The most important concept in this context is the “mental map” (Eades et al., 1991) of the student using this visualizations. According to this concept, changes between visualizations should be as minimal as possible. Thus it is not acceptable that objects displayed in one visualization are changing positions or colors in the subsequent one without changes in the relations between them. Even if changes in the data structure occur it is hard to identify where an object has gone if it has different positions in subsequent visualizations. Thus we require our layout algorithms to look ahead with respect to changes in the data structure, minimizing confusion for the students. In a simple solution, each object has only one fixed position in all visualizations generated for a program. This is well suited to perserve the mental map of the student, but may be confusing with respect to the behavior of the implemented algorithm. For example, an exercise may be to realize sorting of elemtens in a list. In this case, it would be much more helpful if objects in the visualization change place step by step until the algorithm terminates, instead of keeping each in a fixed position and just changing pointers between them.

Note that the requirements for sequences of visualizations may stand in contrast to our general requirements. For example, generating a compact arrangement of objects in order to reduce the size of the visualization would imply to move objects closely together and close gaps between them. However, being foresighted with regards to object positions would imply to reduce this movement and plan with gaps in early visualizations that are filled by objects that appear later on in subsequent visualizations.

## 3 IMPLEMENTATION

The flexible architecture of our e-learning and e-assessment framework JACK (Striewe et al., 2009) allowed to implement an independent module for creating visualizations. The implementation has been done in Java and is so far able to visualize data structures from Java programs.

### 3.1 Retrieving Data

The visualization module is based on the Java Platform Debugging Architecture (Sun Microsystems, Inc., b) and uses the JDI-API (Sun Microsystems, Inc., a) for retrieving objects and events from the Java virtual machine running the program submitted by the student.

Besides the source code it requires a file containing configuration information created by the teacher. This way the teacher can specify which classes of objects should be visualized and which classes are irrelevant for the visualization. Additionally, the teacher can specify breakpoints in terms of lines in the program code at which snapshots of the program state are taken. Note that this requires to have at least one source code file which is not modified by students so that line numbers are known to the teacher for sure.

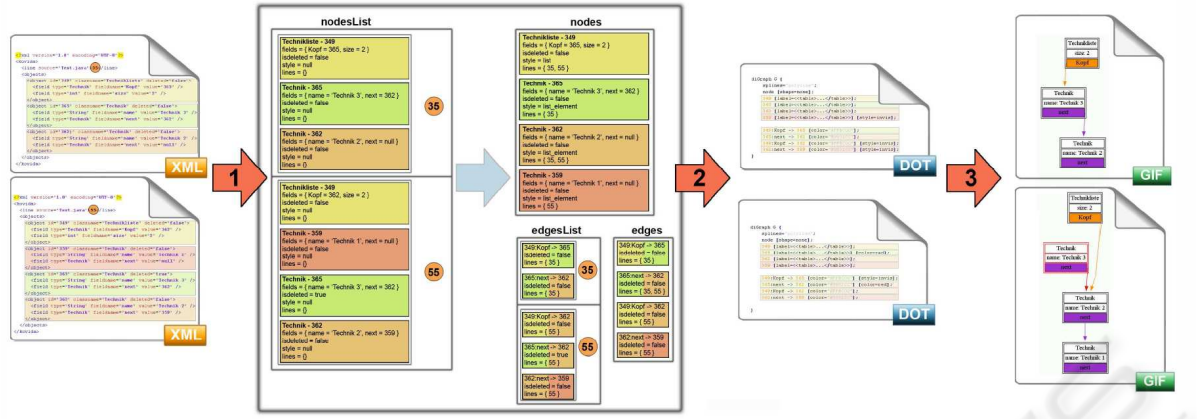



Figure 1: The process of generating a visualization. Step 1 compiles multiple lists of object information from different snapshots into one list of all objects and their relations. Step 2 calculates global settings based on this lists and generates configuration files for each snapshot layout. Step 3 runs the GraphViz routine generating the images based on the configuration.

According to this configuration, the Java Virtual Machine for running the students program is instructed to notify the visualization module every time an object of the given classes is created. Each time a breakpoint is reached, another notification is given and the module takes a snapshot of the currenct program state by collecting the current state for all recorded objects. Each snapshot is stored in a seperate file for further processing.

Also the current implementation is limited to programs written in Java, visualizing data structures from programs in other programming languages would be possible in almost the same way, provided similar debugging interfaces at run time are available or the same format of snapshot files can be produced.

### 3.2 Drawing Graphs

For creating the visualizations, the module makes use of the GraphViz library (GraphViz, 2009) for graph drawing. We use the “dot” routine of this library, because it offers exactly that kind of layered graphs that is suitable for our requirements explained in section 2. However, it is not able to handle sequences of visualizations, so additional effort had to be made to get the module foresighted with regards to optimal positions of objects. See figure 1 for an overview of the visualization process.

The visualization module first compiles a list of all objects relevant in at least one visualization and calculates positions for each of them. These positions are used in all visualizations, even if this would imply large gaps between objects. Thus our implementations preferes preservation of the mental map agains a compact arrangement. Nevertheless most visualizations where small enough to fit the screen without scrolling in our test cases, since GraphViz allows additional options for scaling.

After the list of all objects has been compiled, colors are assigned for coloring attributes of classes. Thus these colors are defined globally and do not change during sequences of visualizations. However, running the same program two times may result in different colorings, because internal mechanisms of the Java Virtual Machine may result in a different order of objects reported to the visualization module and thus a different order of colors assigned to them.

Having done all global settings, the module creates a layout configuration file for each snapshot and runs the GraphViz routine with this configuration in order to generate an image as the final step.

## 4 EXAMPLE

As an example we now consider a programming exercise in which students have to implement a phonebook. A phonebook consists of a data structure composed of objects of type `entry`, containing attributes for name and phone number of the contacts stored in the phonebook. In addition, each entry may have a reference to an object of type `profession`, which has an attribute for a title. Both a list of `entry` objects and a list of `profession` objects should be maintained in a phonebook by pointing from one element in a list to its successor, while an object of type `phonebook` points to the heads of both lists.

Java source code for the types `entry` and `profession` is provided to the students completely, so that they just have to implement algorithms for

inserting objects into a phonebook, sorting objects, removing objects and so on. The source code for the type `phonebook` is only provided as a stub that has to be filled by the students. See listing 1 for the given source code.

The exercise used in this example is taken from a course on programming for first year students in winter term 2008/09. It consists of several task that have to be solved step by step. After each task, a visualization is created, so we will explain and discuss benefits of single visualizations as well as influences of sequences of visualizations here.

```
public class Entry {
   private String name;
   private int number;
   private Entry next;
   private Profession profession;

   public Entry(String name, int number, Entry next){
      this.name = name;
      this.number = number;
      this.profession = null;
      this.next = next;
   }

   public Entry(String name, int number,
        Profession profession, Entry next){
      this.name = name;
      this.number = number;
      this.profession = profession;
      this.next = next;
   }
}

public class Profession {
   private String title;
   private Profession next;

   public Profession(String title, Profession next){
      this.title = title;
      this.next = next;
   }
}

public class Phonebook {
   public String city;
   public Entry headEntryList;
   public Profession headProfessionList;

   // Add here the constructores and methods
    // necessary to solve the exercise.
}
\vspace{1mm}
```

Listing 1: Given Java source code for the tree classes used in the phonebook data structure from the example. Predefined getter and setter methods are not shown here for brevity.

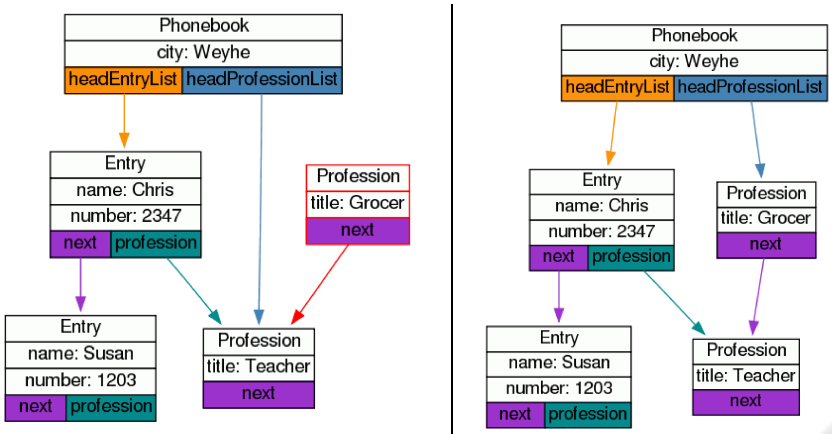
## 4.1 Task 1: Creating and Deleting Objects

The first task inside the exercise is to create a phonebook, insert one entry with a reference to a profession, insert another entry without a reference to a profession, insert an additional profession and afterwards remove all professions that are not references by any entry. In a correct implementation, a visualization of the resulting data structure looks as shown in figure 2(a). In this case, the visualization module was configured to show all objects relevant for the data structure and to consider objects of type `entry` and `profession` as elements of a list, thus ordering them one below each other. Most interesting in this visualization is probably the upper object of type `profession`, which is marked with a red border. This is exactly that extra object that had to be created and deleted afterwards according to the given description of the task. Thus, a single visualization is enough to show that this object has been created and destroyed correctly and there is no need to run extra test cases or insert debug statements into the program code. Moreover, we can see that the “headProfessionList” pointer has been moved correctly to the successor of the deleted element.

As a comparison, figure 2(b) shows a visualization based on the same configuration for a wrong solution of the same exercise. Obviously there is no deleted object. Since the visualization is configured to show all objects relevant for the data structure, it is easy to check whether the first object in the professions list is referenced by any object of type `entry`. Thus the student may conclude that something is wrong with the algorithm for deleting objects, but everything is right with the algorithm for inserting objects.

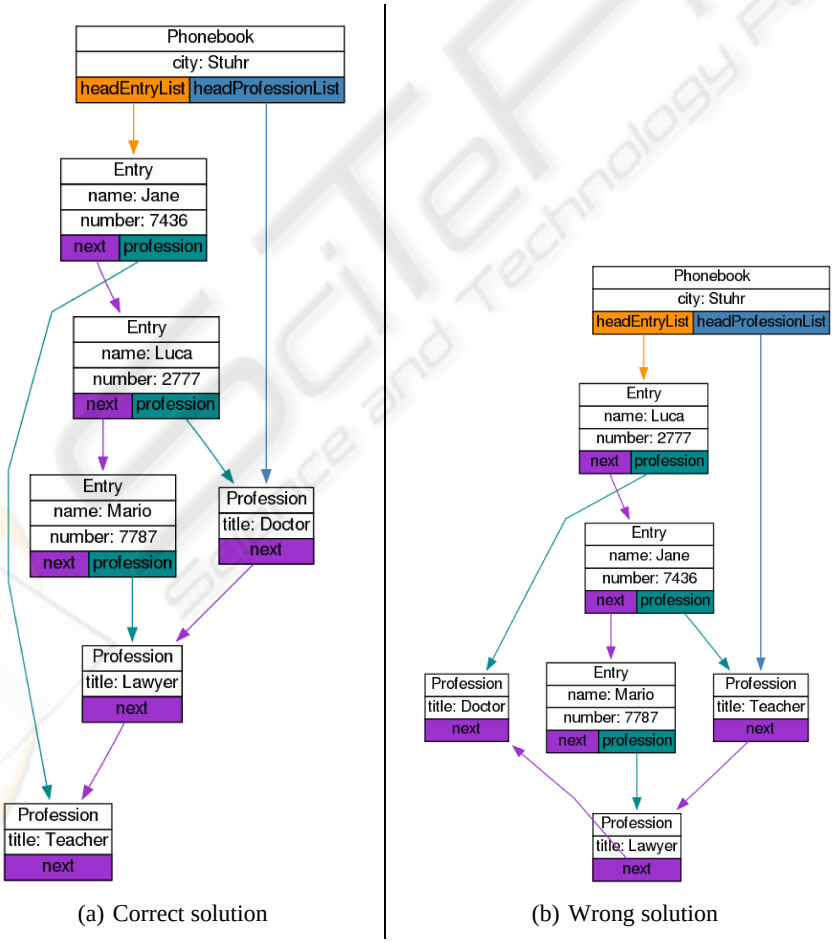
## 4.2 Task 2: Sorting Data Structures

The second task inside the exercise is to create another phonebook and insert several objects of the types `entry` and `profession` and some references between them. Afterwards both lists have to be sorted alphabetically. Figure 3(a) shows a visualization of a correct solution. The same configuration for the visualization module has been used again as for the visualizations of the first task. Obviously both lists are in alphabetical order as requested. In this simple case it may be sufficient to provide a textual output of both lists to check whether they are in the right order. However, it is easy to imagine, that these checks can be performed in an easier way with visualizations in more complex cases. This is already understandable from figure 3(b), which shows a visualization of a




(a) Correct solution (b) Wrong solution

Figure 2: Visualization of solutions for task 1 from the example. In the correct solution, the upper object of type `profession` marked with a red border has been created and removed correctly according to the given task and all pointers are set correctly, too. In the wrong solution, no object has been deleted, although it is clearly visible that the first object of type `profession` in the list is not referenced from any object of type `entry`.




(a) Correct solution (b) Wrong solution

Figure 3: Visualization of solutions for task 2 from the example. In the correct solution, it is easy to see that both lists are in the correct alphabetical order. In the wrong solution, the entries are obviously not sorted correctly. Note that the ugly layout is no bug. Compare to the subsequent visualization shown in figure 5 for explaination.

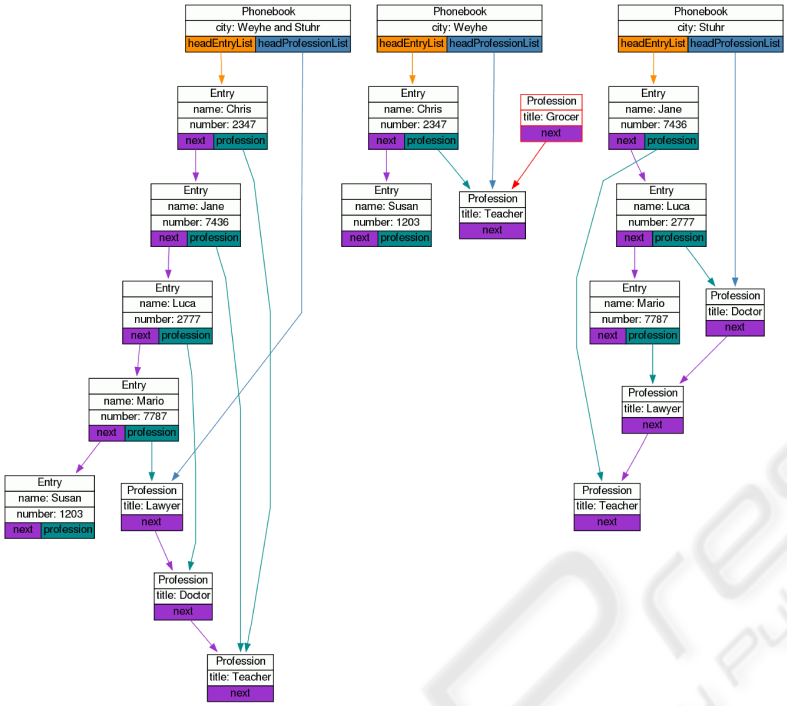



Figure 4: Visualization of a correct solution for task 3 from the example. The newly created phonebook is shown left, while the older phonebooks remain unchanged. Objects of type `entry` in the new phonebook are in the correct order as requested and the last object of type profession has two incoming references.

wrong solution. If we would use a plain textual representation of the list by printing out the names we would notice that they are in the wrong order, but we would not see that the student might just have misunderstood the task by sorting the entries by number.

## 4.3 Task 3: Merging Data Structures

As already stated above, most benefits from visualizations can be expected if data structures tend to get complex. Thus we now consider a third task in which students are asked to merge the both phonebooks created in the previous tasks. More detailed, they are asked to create a third object of type `phonebook`, containing copies of the complete content of the older phonebooks, i.e. copies of all objects of type `entry` in alphabetical order and copies of all objects of type `profession` in arbitrary order, but no duplicates. Thus pointers have to be corrected in case when two objects of type `profession` with the same title exist in both older phonebooks.

Figure 4 shows the visualization of a correct solution, again using the same visualization configuration as the previous ones. Obviously a third object of type `phonebook` has been created using copies of objects as requested. In textual output, students would be required to have much detailed knowledge about object identifiers to see whether they have used copies of objects or just added references, but the visualization makes this clear at once. All other requirements of the task can also be checked easily: The objects of type `entry` are in alphabetical order and the last object of type `profession` has two incoming references.

Figure 5 shows a visualization of a solution where something has gone completly wrong. A new object of type `phonebook` has been created, but no copies of older objects are used. In addition, only some pointers has been added or changed, so that the new phonebook does not even contain all old objects in correct order. However, it can be observed that the reference between the professions “doctor” and “lawyer” has been turned and hence even the data structure of the older phonebook has been destroyed. We would not have noticed this if we would just have used a textual output of the contents of the new phonebook. Additionally, it would have been much more difficult to find this change if the layout algorithm does not perserve the mental map in sequences of visualizations. Since we required our layout algorithm to be foresighted in section 2, this change in the reference is the reason why this objects has already been placed in a non-optimal position in the previous visualization.

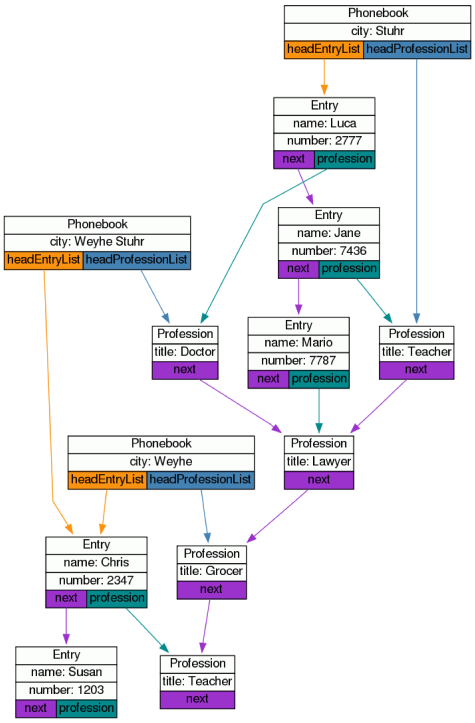



Figure 5: Visualization of a wrong solution for task 3 from the example. A new phonebook has been created, but objects from the older phonebooks are referenced instead of being copied. Moreover, only few pointers have been changed. The reuse of known objects in this task is the reason why the visualization module placed the object of type `profession` with title "doctor" at a position that did not seem to be reasonable in figure 3(b).

## 5 CONCLUSIONS

In this paper, we introduced a visualization module for an e-learning and e-assessment system for the subject of computer science. We argued how visualizations configured by a teacher can help students to analyze the data structures produced by their program code easier than with other devices. The concepts have been implemented in a prototype and shown by examples from a course on programming.

Besides adding more configuration options to the prototype in order to get even better results in layout, two main research tasks are considered at the moment: First, an empirical study among approx. 200 students using the e-assessment system during next winter term will be conducted for getting better feedback about benefits and limitations of the visualizations. This will be related to layout constraints as well as to theories of program comprehension. Second, techniques of search based software engineering and graph pattern matching may be applied to the data structure diagrams in order to comment them automatically. This may for example result in additional colorings, marking parts of a data structure as correct or incorrect.

## ACKNOWLEDGEMENTS

The authors would like to thank Mobasher Ullah for his enthusiasm in implementing the prototype and running many test cases.

## REFERENCES

Eades, P., Lai, W., Misue, K., and Sugiyama, K. (1991). Preserving the Mental Map of a Diagram. Research Report IIAS-RR-91-16E.

Gaylard, A. and Zeller, A. (2009). Data Displaying Debugger. www.gnu.org/software/ddd/.

GraphViz (2009). GraphViz. http://www.graphviz.org.

Kaufmann, M. and Wagner, D. (2001). *Drawing Graphs: Methods and Models*. Springer.

Klling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology*, 13(4).

OMG (2004). UML 2.0 superstructure specification. Object Management Group.

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44.

Striewe, M., Balz, M., and Goedicke, M. (2009). A flexible and modular software architecture for computer aided assessments and automated marking. In *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 2, pages 54–61. INSTICC.

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2).

Sun Microsystems, Inc. Java™Debug Interface API. http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html.

Sun Microsystems, Inc. Java™Platform Debugging Architecture API. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

Zndorf, A. (2009). eDOBS - Java Heap as UML Object Diagram. www.se.eecs.uni-kassel.de/se/index.php?edobs.