Software Engineering modules for Computer Science Talent Scout

Roman Fuchs

Master Project Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://pm.inf.ethz.ch/

August 2009

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

“To be playful and serious at the same time is possible, in fact, it defines the ideal mental condition.”
– John Dewey: How we Think[12]

Abstract

A majority of high school students has a very limited or wrong perception of a Computer Science course of study. A good mean to counteract this situation is to provide them with interactive learning environments that expose them to interesting and fundamental concepts of Computer Science. We found that, among the topics of existing environments, the field of Software Engineering is covered very briefly. In contrast, Software Engineering plays a key role in the work of most Computer Science graduates that work in the industry after their studies.

In this thesis we extend the *Computer Science Talent Scout* with modules about Software Engineering topics. This learning environment aims to both check key study skills and introduce relevant and interesting topics in order to attract talented students to the field of Computer Science. One topic we chose is *White-box Testing*, where the students learn the concept of code coverage and apply several testing techniques. The functions to be tested are represented by flow charts in order to facilitate the understanding of the control structures and to visualize the process of testing and debugging. Another topic we cover is *Tree Recursion*, where the students need to apply algorithmic thinking in the context of a recursive data structure.

We implemented two modules together with a suite of interesting problems. The module Tree Recursion comes with a graphical editor that allows creating recursive algorithms without typing code in order to simplify the task and to avoid syntax errors. Both modules provide a step-by-step animation of the flow chart evaluation and the algorithm execution respectively. We also developed a flow chart editor to easily create new flow charts using a graphical interface.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Studies have shown that a majority of high school students is unaware of the intellectual demands posed by a Computer Science course of study.[44] As a common misconception Computer Science is understood as the use of application programs, especially in the field of text editing, spreadsheet processing and image editing. This distorted perception is mainly based on the Computer Science classes being taught at high school. As a consequence students often don't know what to expect when choosing this discipline and high drop-out rates among Computer Science students are to be expected. There is also a growing concern about the reduction in students choosing to study Computer Science. Apparently, Computer Science fails to attract students of potentially high aptitude.[9]

Part of the students' disinterest in Computer Science is simply due to a lack of familiarity with the subject. High School students are not provided with the opportunities to find out what the field of computing encompasses. How are they to choose a study subject they know nothing about? To influence a student's choice of whether to pursue a major in the field of Computer Science, we mainly have to address two factors: *interest* and *self-efficacy*, i.e. a student's judgment of his or her capability to perform well as a CS major.[2] Therefore, students need to be exposed to intriguing topics, but also experience success. The tasks should be challenging, yet attainable. A good way to both check key study skills and to introduce relevant and interesting topics are interactive learning environments. With the Computer Science Talent Scout an appropriate tool exists, that covers a few areas of Computer Science. By extending its current scope with interesting and relevant concepts from Software Engineering we intend to arouse the interest of talented students to the field of Computer Science.

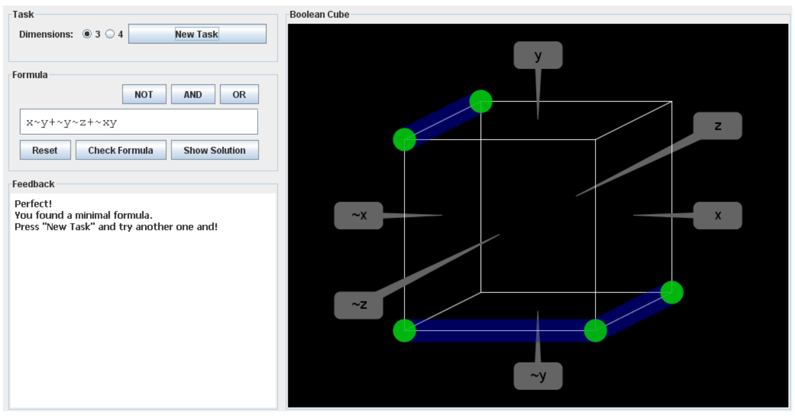
1.2 Talent Scout

The Computer Science Talent Scout[31] (CS-TS) is a collection of Java Applets (called "modules") with interactive tasks, puzzles and games that can serve as a self-assessment aptitude test for computer science. Mathematical maturity typical of high school will be expected to solve the problems posed in the modules. However, they do not depend on previous knowledge in Computer Science. All modules illustrate different concepts of relevance for a Computer Science course of study. CS-TS should help students to get an understanding of the type of thinking and problem-solving which is needed as a computer scientist. In this respect it also aims to change the perception of the computer from a mere tool to an object worthy of profound study and ultimately to attract bright students to the discipline of Computer Science.

There already exist five different modules for Talent Scout, which will be briefly described in the following:

Boolean Cube

A boolean cube is used as a visual aid to find a minimal formula for a boolean function. Each face of the cube is denoted by either x, $\sim x$, y, $\sim y$, z or $\sim z$. Each vertex of the cube is therefore equivalent to an input configuration of a boolean function. For a given selection of cube vertices the user has to find a minimal boolean formula representing this configuration. He can choose between functions with three or four variables.



x~y+~y~z+~xy

Perfect!
You found a minimal formula.
Press "New Task" and try another one and!

Figure 1.1: CS Talent Scout : The module *Boolean Cube*

Random Numbers

The goal of this module is to identify the imperfection of random numbers generated with a deterministic algorithm. The Linear Congruential Method is used to generate a sequence of pseudo-random numbers. The user can set the parameters of the generator formula

$$x = (a * x + c) \bmod m$$

and observe its effects. The sequence of generated numbers is visualized by a hyperplane. As one can see there is ultimately a cycle in the sequence of numbers. The distribution of the generated numbers can be studied with the chi-square test that estimates the probability that a real random sequence would accidentally have the observed properties.

Recursive Images

With this module the user can define a pattern of lines which will recursively replace all the lines it contains. Using a set of parameters he can generate and modify the recursive image, e.g. specify the iteration depth or change the color and opacity of the generated image. The user can also learn about recursive grammars and the Chomsky hierarchy.

Image Compression

Data compression is an important field of Computer Science. As an application this module presents two methods to efficiently reproduce the shape of a bitmap image. Using either rectangles or QuadTrees one has to cover the shape of a two dimensional bit pattern with minimal number of quadrilaterals. The compressed size and the compression ratio of the new representation will be shown and the efficiency of the different methods can be compared. A board game called “Land Grab” derived from the rectangle method lets the user compete against the computer that is using a greedy strategy. As can be shown with a simple example, always covering the largest possible rectangle is not optimal.

Image Steganography

This module allows the user to encode a secret message or information within an ordinary image using different steganography encode parameters (e.g. saturation and brightness). These can be saved for later recovery. Contrary, he can also open a steganography image and try to find the correct decode parameters to reveal the secret information. The module also contains a generator where the user can practice with random steganography images.

1.3 Contributions

The goal of this master thesis is to design and implement Software Engineering modules for the Computer Science Talent Scout in order to extend its current scope. The User Interface needs to have a simple design, so that users don't need much time to acquaint themselves. An important aspect is the combination of interactive entertainment and education to increase the motivation of the users. We assume that the students have an instructed tutorial about the underlying problem before using these modules.

The modules have to cover different topics in the broader field of Software Engineering. On the one hand that can include fundamental programming paradigms such as iteration or recursion, algorithmic strategies or programming languages. On the other hand this also includes specific subdisciplines of Software Engineering such as design (e.g. design patterns), testing or project management. After doing a survey on current educational software and some research on Computer Science education we eventually chose the topics *White-box testing* and *Tree Recursion*. The concepts of the modules are elaborated in chapter 3.3.

1.4 Overview

Chapter 1 states the problem this work addresses and introduces the Computer Science Talent Scout as the underlying framework of this project. Chapter 2 gives a survey of learning environments in Computer Science education. It presents the most popular fields and introduces some sample applications. Chapter 3 investigates various aspects of Computer Science education and how to teach Software Engineering. Chapter 4 gives an overview about the first module that is concerned with White-box testing. The different tasks and the animation are described in some detail as well as the Flow Chart Editor which we developed as an extension. Accordingly, chapter 5 presents the second module which is about Tree Recursion. The included editor, its language and the random tree generation are described in more detail. Chapter 6 introduces the basic system architecture of both modules and describes the visual framework. It further presents some implementation issues. Chapter 7 contains the conclusion and suggests some future work.

Chapter 2

Learning environments in Computer Science Education

Nowadays, computer-based learning environments can be found for a wide range of topics. They are mostly designed to provide an engaging environment where learning involves more than just receiving information. Interactive components promote student involvement and makes learning the theory to be more fun.

Especially in Computer Science, educational software can be very beneficial for students that have a hard time dealing with abstractions. The use of visualizations and multiple representations helps to get a better understanding of abstract concepts.

There is a broad selection of topics in Computer Science currently addressed by computer-based learning environments. However, while certain areas are very well covered, such as data structures, others have almost been neglected. Three very popular classes of educational software for Computer Science are visualization of algorithms and data structures, playful introduction into programming, and topics from Theory of Computation. In the following we survey current state of the art learning environments in these areas and conclude with an application covering a specific Software Engineering topic.

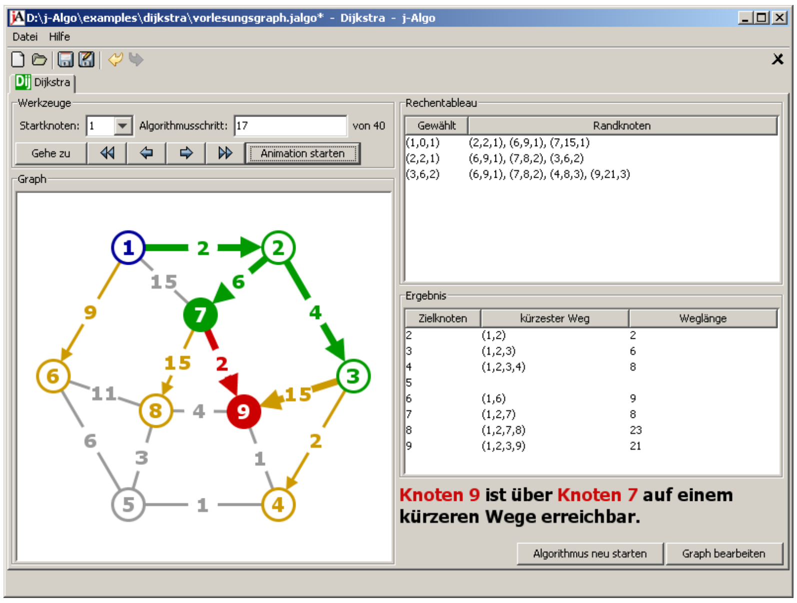
2.1 Visualization of algorithms

Very popular forms of teaching applications are visualizations of algorithms. Visualization is regarded as an effective mean to facilitate the understanding of a workflow. We will briefly introduce some current systems.

The *ANIMAL* system[37] is a multi-purpose animation tool which is so far focussing on algorithm animation. It currently offers about 60 animations of algorithms and data structures. Animations are created using a visual editor, by scripting or via API calls. Also existing animations can be edited visually on a drawing pane. It also supports embedding source code or pseudo code that will be highlighted during execution. Individual execution time can be assigned to each animation step.

JAWAA2[36] (the acronym stands for *Java and Web based Algorithm Animation*) was created at Duke University and uses a scripting language for defining algorithm animations. As a component it contains the JAWAA editor that allows users to create animations by arranging graphical objects, and then modifying them over time. A selection of sample animations for common data structures such as Queue, Stack, Array and List as well as search and sort algorithms is available on their website.

The animation system *j-Algo*[6] currently comprehends 7 modules that visualize classic algorithms and data structures such as AVL trees, Heapsort and Dijkstra's algorithm to solve the shortest path problem for a graph (Figure 2.1). The execution of an algorithm can be shown as an animation or step-by-step.



Gewählt	Randknoten
(1,0,1)	(2,2,1), (6,9,1), (7,15,1)
(2,2,1)	(6,9,1), (7,8,2), (3,6,2)
(3,6,2)	(6,9,1), (7,8,2), (4,8,3), (9,21,3)

Zielknoten	kürzester Weg	Weglänge
2	(1,2)	2
3	(1,2,3)	6
4	(1,2,3,4)	8
5		
6	(1,6)	9
7	(1,2,7)	8
8	(1,2,7,8)	23
9	(1,2,3,9)	21

Knoten 9 ist über Knoten 7 auf einem kürzeren Wege erreichbar.

Figure 2.1: j-Algo : Dijkstra's algorithm

While animated visualization tools are very useful to improve the understanding of dynamic systems such as algorithms and data structures, a common problem is the passive role of the user. Besides adjusting animation settings, they often involve very little input from students. We aim for an application where the user has to actively understand and apply concepts, rather than only seeing how it works. A type of such learning environments will be presented in the following section.

2.2 Playful programming

There are a number of programming environments designed for young people or programming beginners allowing them to get familiar with fundamental concepts of programming in a playful manner. In an effort to reduce the complexity barrier that modern programming languages represent for many beginners, various *mini-languages*[8] have been designed especially for an educational purpose. Following this approach programming can be learned using a small and simple language to facilitate the first steps. In most cases the student can control an actor such as a turtle or a robot acting in a microworld. The mini-language includes a small set of commands and queries of the actor and several control structures. Due to the small syntax and simple semantics students are able to familiarize themselves very quickly to these languages and can focus on the logic and correctness of their programs.

A major influence for the development of the mini-language approach was given by turtle graphics of *Logo*[25]. However, there were still several limitations when compared with newer systems such as the turtle of Logo being "blind", i.e. it can't check its surroundings in the microworld. The first real mini-language *Karel the Robot*[33] was designed by Richard Pattis in 1981. Karel contains all important control structures and teaches the basic programming concepts such as sequential and conditional execution, procedural abstraction, and repetition.

To this day, many systems have been created in the tradition of Karel, some of which improved considerably their visual interface. The aspect of visual illustration is not to be underestimated. Visual actions help understanding the semantics of introduced language constructs and principles of program structure and execution. Visualizations also make it easier to develop interesting problems.

Even a step further goes the approach of visual programming. This approach, also called *drag-n-drop programming*, removes the act of typing code and allows creating programs in a visual editor only using the mouse. The underlying principle is the same: to teach algorithmic thinking without the students being distracted by syntax.

We describe two representative systems for playful programming, Kara and Scratch, in more detail.

Kara

Kara[16] is a programming environment based on finite state machines. The students can program a virtual ladybug called Kara that lives in a simple, grid-like world (Figure 2.2). Kara can execute a few primitive actions such as *advance one square*, *turn right* or *pick up a cloverleaf* and has sensors that inform Kara about its immediate surroundings like *mushroom in front?* or *leaf on the ground?*. Using these commands and sensors, finite state machines are specified in the visual program editor. The use of finite state machines removes the complexity of a programming language and eliminates syntax errors. Students can choose from a collection of exercises with different levels of difficulty or they also create their own worlds and problem settings.

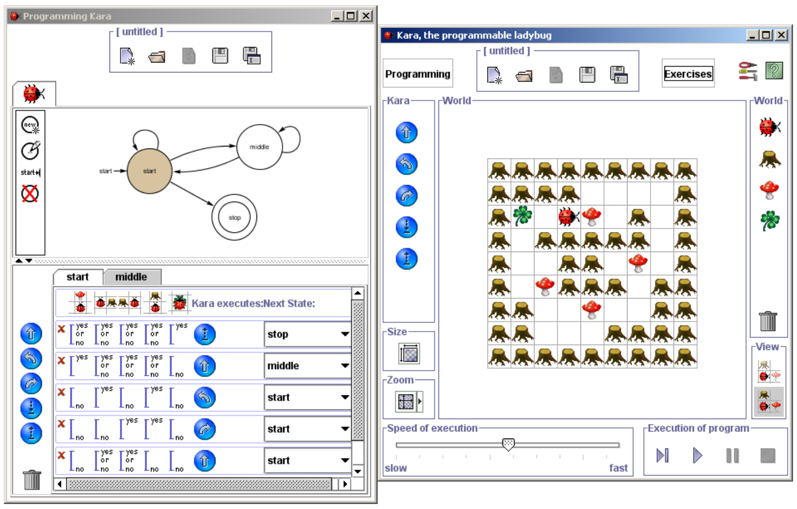


Figure 2.2: The Kara environment with the world (right) and the program editor (left)

Apart from the basic environment different ones have been implemented that offer playful introductions to fundamental concepts of programming at different levels: *MultiKara* introduces the basics of concurrent programming. Up to four ladybugs can be programmed and mechanisms for temporal synchronisation or mutual exclusion solve problems where they interfere with each other. *JavaKara* offers the possibility to write the programs in Java instead of finite state machines in order to decrease the gap between programming in the learning environment and in a real world environment. Other implementations with programming languages such as JavaScript, Python, and Ruby have also been published most recently. The *TuringKara* environment allows students to design and operate a two-dimensional Turing machine. Although a one-dimensional "tape" would be equally powerful, the use of a two-dimensional tape simplifies the solution for many problems. Finally, *LegoKara* is an implementation of Kara as a Lego Mindstorm robot[38] and enables users to test their programs on a physical robot.

Scratch

Scratch[26] is a simple visual programming language developed by the Lifelong Kindergarten[24] research group at MIT's Media Lab. It is mainly used as a teaching language for first-time programmers in introductory courses. The development environment provides code fragments (called "blocks") that can be dragged onto the script area and combined to create programs (Figure 2.3). The method is called building-block programming. Among these blocks are such fundamentals as statements, boolean expressions, conditions, loops, variables, threads, and events. The blocks have different shapes and only fit together in syntactically-correct ways. This approach eliminates syntax errors and type mismatches, allowing the students to focus on the problems they want to solve, not on the syntax. Using these blocks the students can program one or more "sprites" (i.e. characters) on a "stage" and therewith create animated stories, games or interactive art.

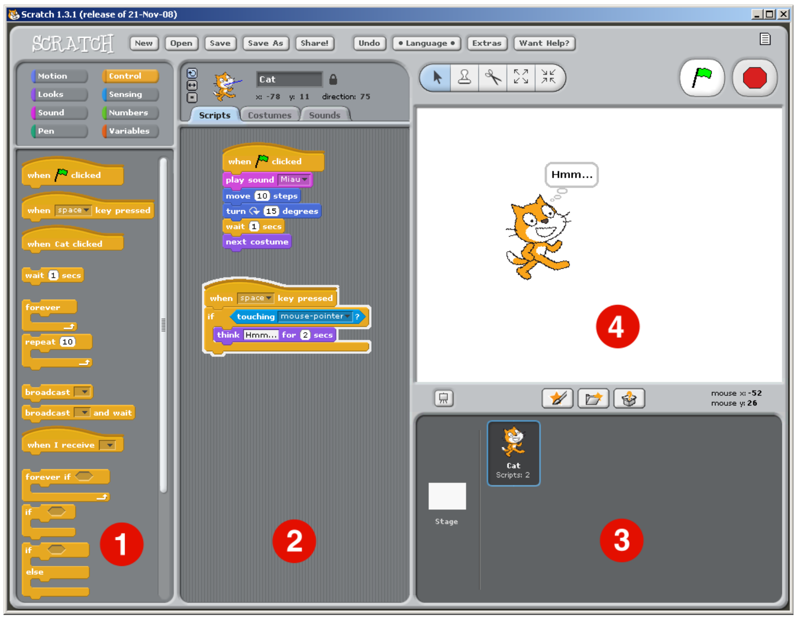


Figure 2.3: Scratch's interface consists of a *blocks palette* (1), a *scripts area* (2), a *selection area* (3), and a *stage* (4)

In contrast to traditional programming environments, where primary exercises usually involve manipulation of numbers, strings or simple graphics, Scratch allows manipulation of rich media, such as images, animations, movies, and sound. Another important aspect of Scratch is reusability. Programs developed in Scratch can easily be copied in another project and recombined. The only way to make a program available for use is by releasing the source for it.

In a similar way, also using the building-block approach, the 3D animation software *Alice*[11] lets the students animate 3D objects in a virtual world. As it has shown, it made the process of writing a program much more intriguing, especially for female students.[21]

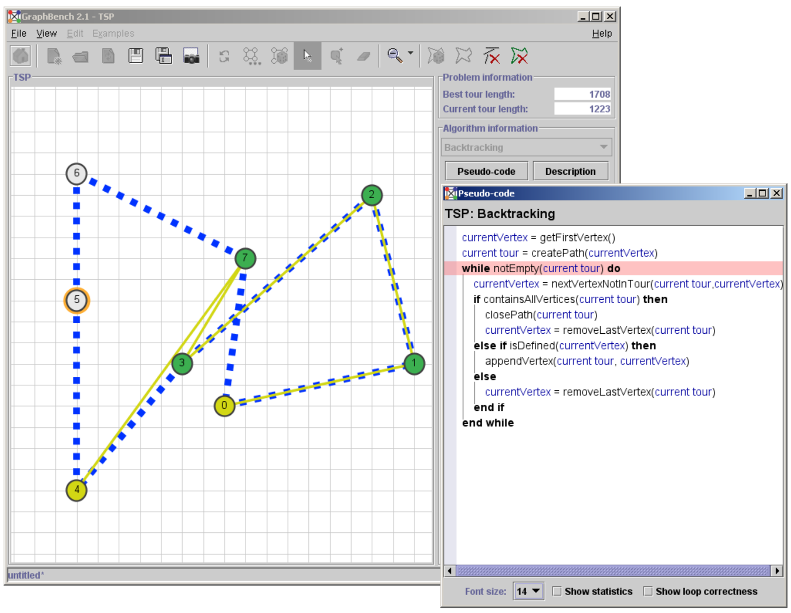
2.3 Theory of Computation

Despite the combination of abstract topics and formality, Theory of Computation is one of the best covered areas when it comes to interactive learning tools in the field of Computer Science. One of the reasons is certainly that especially problems from graph theory and automata theory are well suited to be visualized. In the following we introduce two popular learning environments covering the area of Theory of Computation. Both of them have been developed at ETH Zurich in the course of a doctorate.

GraphBench

GraphBench[7] is an learning environment for the theory of NP-completeness. It features eight different NP-complete problems from graph theory (e.g. Traveling salesman, Graph coloring or Independent Set) and nine different polynomial time reductions (e.g. reducing Satisfiability to Clique).

The learning environment allows students to solve arbitrary examples by hand or to use built-in solution algorithms. All featured NP-complete problems come along with various graphical representations of the provided solution algorithms (i.e. exhaustive search and heuristics). In order to support comprehension, the algorithms have a pseudo-code representation and an animated visualization during execution (Figure 2.4). GraphBench also includes a simple programming editor that allows students to implement their own graph algorithms in Java.



Best tour length: 1708
Current tour length: 1223

TSP: Backtracking

```
currentVertex = getFirstVertex()
current tour = createPath(currentVertex)
while notEmpty(current tour) do
  currentVertex = nextVertexNotInTour(current tour,currentVertex)
  if containsAllVertices(current tour) then
    closePath(current tour)
    currentVertex = removeLastVertex(current tour)
  else if isDefined(currentVertex) then
    appendVertex(current tour, currentVertex)
  else
    currentVertex = removeLastVertex(current tour)
  end if
end while
```

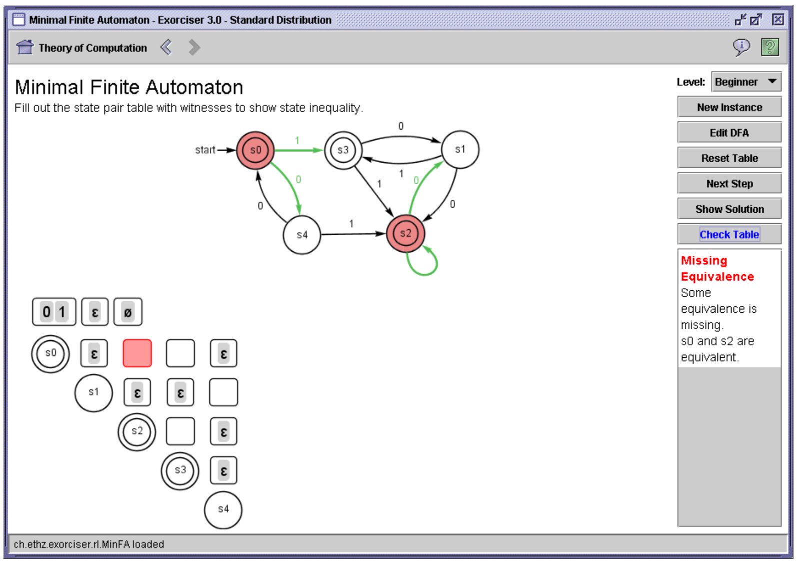
Figure 2.4: GraphBench : Visualization of the Backtracking algorithm for *Traveling Salesman*

To create further problem instances, GraphBench is capable to generate random examples and also allows students to manually create examples of their own or to modify existing ones as desired. Furthermore, GraphBench uses the concept of “selective level of detail” to hide unnecessary information and to help students to cope with problem instances of large scale.

Exorciser

Exorciser[43] is a learning environment for an introductory course on the Theory of Computation. It contains 25 interactive exercises and covers topics such as finite automata (Figure 2.5), context free grammars and Markov algorithms. The students can generate an unlimited number of distinct problem instances of various levels of complexity.

On request, an effective automated tutor provides meaningful feedback to every step of the solution process. The ability to provide individual step-by-step feedback is based on the concept of a solution space. The solution space for a given class of exercises is a structure that captures all consistent sequences of operations for solving problems of that type by following appropriate algorithms. The feedback provided to the student ranges from indicating the incorrectness of the proposed solution to a full correction of the solution. In case of mistakes the tutor is able to produce a counter example that proves the incorrectness of the solution.


Minimal Finite Automaton - Exorciser 3.0 - Standard Distribution

Minimal Finite Automaton

Fill out the state pair table with witnesses to show state inequality.

Level: Beginner

Missing Equivalence

Some equivalence is missing.

s0 and s2 are equivalent.

ch.ethz.exorciser.rl.MinFA loaded

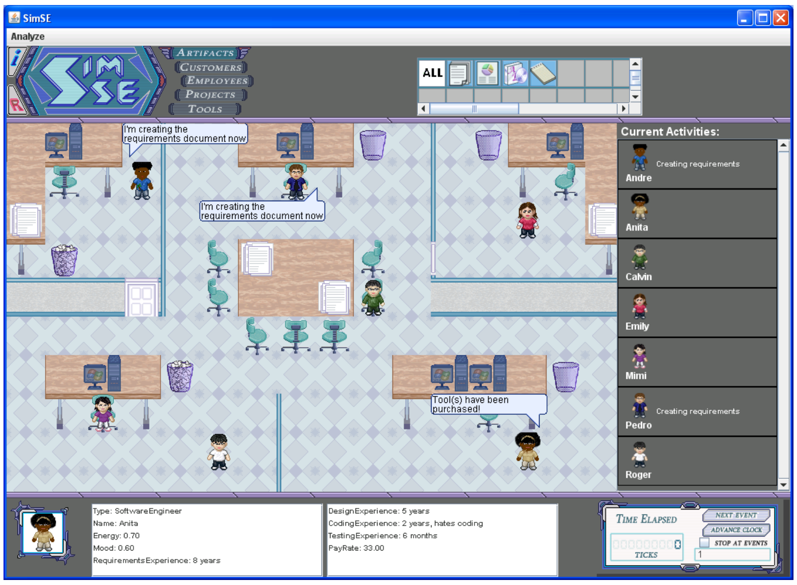
Figure 2.5: Exorciser : Interactive state minimization

2.4 Software Engineering

While we found a variety of teaching applications for different areas of Computer Science, the selection of educational software for specific Software Engineering topics was not as broad. However, there exist some Software Engineering simulation games and as an example we take a look at SimSE.

SimSE

SimSE[28] is a simulation game for the software engineering *process*. The player takes on the role of project manager of a team of developers that work on a software project. Management activities include, amongst other things, hiring and firing of employees, assigning them tasks, purchasing development tools, and monitoring the progress of the project. SimSE has a completely graphical user interface that displays the virtual office where the software engineering process takes place (Figure 2.6). It also shows relevant information about the employees (e.g. productivity, current task, and energy level), the customers (e.g. satisfaction level), the projects (e.g. time and budget) and tools (e.g. productivity increase factor). Using this information the player makes decisions and takes actions that effect project attributes, such as cycle time, cost, and quality and drive the simulation accordingly. While following good software engineering practices will lead to a successful completion of the project, ignoring them will lead to failure. At the end of the game the performance of the project is measured and the player receives a corresponding score. Furthermore, an explanatory tool is accessible both during and after the game that provides the player with further information about his game, including a trace of events and the state of various project attributes over time. This should help the player to draw his conclusions about the cause-and-effect relationships in the process (e.g. more employees leads to more parallel work, which leads to more integration problems). Currently, there are six different models of the software development process supported by SimSE: the waterfall model, an incremental model, a code inspection model, a rapid prototyping model, a Rational Unified Process model, and an Extreme Programming (XP) model.



I'm creating the requirements document now

I'm creating the requirements document now

Tool(s) have been purchased!

Andre — Creating requirements

Pedro — Creating requirements

Type: SoftwareEngineer
Name: Anita
Energy: 0.70
Mood: 0.60
RequirementsExperience: 8 years

DesignExperience: 5 years
CodingExperience: 2 years, hates coding
TestingExperience: 6 months
PayRate: 33.00

Figure 2.6: SimSE : Graphical User Interface

By providing a realistic high-level experience of the software engineering process, SimSE aims to address the lack of process education in the regular software engineering course. While this allows students to develop a better understanding of software process issues, it covers very broad topics and the approach doesn't seem suitable for high-school students. We also didn't want to cover the process activities of software engineering, but rather the underlying principles and concepts. In order to enable high-school students to learn concepts from software engineering, we therefore had to choose a specific subdiscipline. The didactic concept will be elaborated in the following chapter.

Chapter 3

Teaching Software Engineering

The classical educational paradigm stated that the primary means to gain knowledge are listening to lectures and reading books. This passive way of learning has been criticized since and the dominant theory of learning today is called *constructivism*. This theory claims that learning is more effective when students actively construct knowledge from their experience rather than simply receive and store knowledge communicated by the teacher. According to the constructivist paradigm, instructors have to take on the role of facilitators and not teachers. They should create an environment where students develop their own understanding of principles, concepts, and strategies. As a method of instruction constructivists often encourage the use of *discovery learning*, where students are expected to discover knowledge by themselves.

Constructivism has had a wide ranging impact on learning theories and teaching methods in education. One of the most widely known is *constructionism*, a learning theory developed by Seymour Papert.[15] It takes the constructivist view of learning as a reconstruction rather than as a transmission of knowledge and extends it with the idea that learning is most effective when the learner constructs a meaningful product. Papert proposed using a computer-based learning environment where students create programs in a microworld.[32] Based on these ideas the programming language Logo was created for educational use.

But constructivism also faces a strong opposition, especially when the learning theory is applied to teaching methods. Mayer (2004)[27], for example, criticizes the common view that equates active learning with active teaching and concludes that "the formula constructivism = hands-on activity is a formula for educational disaster". Rather than putting the focus on behavioral activity, learners should be "cognitively active" during learning. Furthermore, he stresses the importance of instructional guidance and criticizes the use of pure discovery learning, where students work with little or no guidance. He recommends the use of *guided discovery*, a mix of direct instruction and hands-on activities, where the teacher helps guide the students in productive directions.

Kirschner, et al. (2006)[23] argue that unguided methods of instruction are only effective when the learners' prior knowledge is sufficiently strong to provide "internal" guidance and are not useful for novices. Hereby they address a number of learning theories (discovery, experiential, problem-based, and inquiry-based learning) that all promote the constructivist concept "learning by doing". For learners with little or no prior knowledge they suggest using more structured methods instead.

Generally, one can say that learning is most effective when being done actively. But students should be assisted by guidance from the teacher or other support material. Also, the pre-existing knowledge should also be taken into consideration and one needs to be aware of misconceptions and needs to address them.

Another important prerequisite of successful learning is motivation. A place with extensive expertise about motivation is the game industry. Game players usually have the attitude which educators would like their students to have: interested, competitive, cooperative, results-oriented and actively seeking information and solutions.[34] Therefore, the educational paradigm *game-based learning* has been studied extensively in recent years in order to increase the motivation and the engagement of learners. Studies showed that for a successful implementation of this approach, the player needs to be provided with clear goals, immediate and appropriate feedback, and challenges that match his skill level.[22]

3.1 Computer Science Education

There is a significant amount of research in Computer Science education (CSE). The main motivation is to improve the quality of the teaching and learning of the subject in schools and universities. It is important to distinguish CSE from work being done on *computers in education*, which usually does not focus on the academic discipline of Computer Science.

A review of existing CSE literature shows a focus on a few topics such as course descriptions, development of tools, and computer aided learning. Although these topics are of importance, they are also relatively limited and the publications often do not include references to pedagogical theory. Also, research tends to be grounded in the technology, rather than in the didactics of Computer Science or educational theory.[17]

A substantial number of research publications are written by practitioners in CSE and are based on their own experiences of teaching a certain course. They are often concerned with problems such as high failure and drop-out rates in their introductory courses. However, the effectiveness of the proposed changes are rather difficult to evaluate. Other research projects focus on computer aided learning and developed intelligent tutoring systems as can be found e.g. in Exorciser. Some publications also refer to learning theories like constructivism[5], discovery learning[4] or game-based learning[22].

Another important aspect of CSE research appears by answering the traditional didactical questions of *why*, *what*, *how* and *for whom*. Why should Computer Science be taught as a subject in school, which topics should be covered and how can we build the desired knowledge? The question for the receiver hereby provides the basis for answering the other three. These questions are crucial for a successful coordination between teachers, policymakers and educational researchers. However, if we look around at schools we can see that there is no consensus, which topics should be at the core of each introduction into Computer Science. Let us look into some model curricula that have been developed in an effort to reach such a consensus.

3.1.1 Computer Science Model Curricula

Different model curricula have been developed by various institutions to set standards in Computer Science education. In the following we introduce two curricula that have been selected due to prevalence (IFIP/UNESCO) and long tradition (ACM, first recommendation in 1968).

The International Federation for Information Processing, usually known as IFIP, is an NGO established in 1960 under the patronage of UNESCO. In 2002 the IFIP published the latest curriculum for information and communication technology (ICT) in secondary schools.[10] The curriculum is structured into four modules:

- *ICT Literacy*
  The first module is designed for students to discover different ICT tools, their functions and how to use them. Therefore, basic ICT skills such as word processing, spreadsheets, and creating presentations are taught as separate subject.

- *Application of ICT in Subject Areas*
  The second module is designed for students to learn how to make use of ICT tools in the different subjects studied in secondary school including mathematics, natural and social sciences, languages, and art.

- *Infusing ICT across the Curriculum*
  The third module is designed to demonstrate the use of ICT tools across different subjects to work on real-world projects.

- *ICT Specialization*
  This module is primarily designed for students who plan to go into professions with an increased use of ICT such as engineering, business, and computer science. It covers topics such as introduction to programming, top-down program design and foundations of software development.

As one can see, in this arrangement "informatics" as a science is of less importance. The acquisition of ICT skills is predominant, which leads to an application-oriented view of Computer Science. This development can also be observed in many high schools and in political policies.[39]

The Computer Science Teachers Association (CSTA) published in 2003 the second edition of the ACM Model Curriculum for K-12 Computer Science.[3] This curriculum consists of four levels that can be teached from grade level K-8 to K-12 accordingly:

- *Foundations of Computer Science*
  Level I should enable students to acquire basic skills in technology and to incorporate algorithmic thinking as a general problem-solving strategy.

- *Computer Science in the Modern World*
  Students at Level II should develop a fundamental understanding of the principles of Computer Science and its applications and implement useful programs using simple algorithms. This course should also include an overview of career possibilities and a discussion of ethical issues related to computers. Since it will be the last encounter with Computer Science for most students, it is considered as an important preparation for the modern world.

- *Computer Science as Analysis and Design*
  Students who wish to study more Computer Science may elect the Level III course. It provides the students with an opportunity to explore their interest and aptitude for Computer Science as a profession. The main focus lies on the scientific and engineering aspects of Computer Science such as mathematical principles, algorithmic problem-solving and programming, software and hardware design, networks, and social impact.

- *Topics in Computer Science*
  Finally, the elective Level IV course aims to prepare the students for further studies at the tertiary level or for the worklplace. It also gives the students the opportunity to explore topics of personal interest in more detail. This course may either be an advanced placement course with focus on programming and data structures or a project-based course in multimedia design or a vendor-supplied course that leads to professional certification.

This curriculum is based on a "conservative" understanding of Computer Science and requires the educational content to be geared to scientific principles and concepts. It also makes clear that the occupation with technical details should be avoided and rather a concentration on basic scientific principles and concepts should take place. As a general goal students should be enabled to understand the nature of Computer Science and its place in the modern world.

The discrepancy of these two curricula shows the necessity of an international debate of computer science as part of general education. The discussion whether Computer Science education should be oriented more towards its applications or more towards its fundamentals or more towards its social effects is still ongoing. From our perspective it is clear that the reduction of Computer Science to mere computer handling skills has to be avoided. There should be a clear separation between Computer Science and general computer literacy in high school education. In order to do so, the value of Computer Science as a school subject has to be strengthened.

Another topic that is widely discussed, for example also in the ACM curriculum, is teacher education. What is the technical and nontechnical knowledge required for a successful Computer Science educator?[13] It starts with the formal education: While a doctoral degree in Computer

Science is a reasonably obvious prerequisite for college-level teachers, the requirement for high school-level teachers to be equipped with a Master's degree in Computer Science is not always recognized. In fact, this requirement is rarely met. Many Computer Science high school teachers have not passed a full university education in that field. As a result, some teachers lack a comprehensive technical understanding and are only able to teach programming and doing this as writing simple algorithms in a simple, fixed language. Beyond having a good technical knowledge, Computer Science educators should have a broad overview of the discipline. They should be exposed to a so-called bird's-eye view of the field to address, for instance, questions of the nature of the field and its relationships with other disciplines. Of course these skills need to be accompanied by didactic knowledge.

Nievergelt defines the task of a Computer Science teacher as follows: "A computer science teacher has to be capable to convey the fundamental ideas of Computer Science intelligent and understandable also for non-professionals."[30] But what are the fundamental ideas of Computer Science?

3.1.2 Fundamental Ideas of Computer Science

As a relatively young academic discipline Computer Science is still developing dynamically. Paradigm changes are constantly announced and each time much of the respective knowledge becomes obsolete. An educational principle to cope with these frequent changes is to base the education on so-called *fundamental ideas* (some authors also use the term *central concepts*[46]). Using this approach students are expected to be able to transfer earlier acquired knowledge to new situations. That means these ideas must be robust enough to meet the challenges of the latest developments in Computer Science.

A fundamental idea of Computer Science can be defined as a schema for thinking, acting, describing or explaining which satisfies the following four criteria: It has diverse applications and can be observed in different areas of Computer Science (*horizontal criterion*), it can be taught at any intellectual level (*vertical criterion*), it can be observed in the historical development of Computer Science and it will be relevant in the long term (*criterion of time*) and it also has a meaning in everyday life (*criterion of sense*).

In 1997, Schwill proposed three fundamental ideas that "dominate all stages of software development as well as all activities in computer science": *algorithmization*, *structured dissection* and *language.*[40] These ideas are explained in more detail in the following.

- *Algorithmization* covers the entire process of designing, implementing and running an algorithm. Further analysis of these activities reveals a number of other fundamental ideas that can be divided into four subdomains: design, programming, execution and evaluation (Figure 3.1).

- *Structured dissection* comprehends the process of subdividing an object into several parts in a structured way. It can be distinguished between a vertical aspect called *hierarchization*, where different levels of abstraction are created, and a horizontal aspect called *modularization* where an object is subdivided into parts of the same level of abstraction. By merging these aspects we obtain a *hierarchical modularization.*

- *Language* plays an important role in many areas of Computer Science, e.g. for programming, specification or verification, but also in data bases (query languages) and in operating systems (command languages). It is further associated with the two fundamental ideas *syntax* and *semantics.*

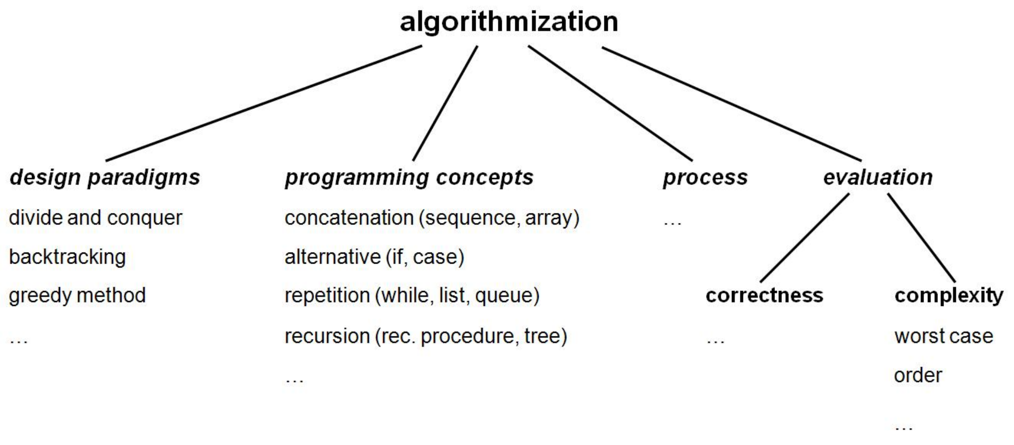


Figure 3.1: Algorithmization : a fundamental idea of Computer Science

3.2 Topics of Software Engineering

After having seen an overview of educational paradigms in general and specifically related to Computer Science, we also have to study the discipline of Software Engineering in regard to our goal of creating related modules. Software Engineering is widely considered as a subfield of Computer Science and can be defined as “application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software”.[19]

In 2004 the IEEE Computer Society (IEEE-CS) published a guide to the body of knowledge in the field of software engineering[1] in order to promote a consistent view of this field and to establish a foundation for curriculum development. In this guide, the subject areas of the Software Engineering discipline are organized into ten Knowledge Areas (KAs) as listed in Table 3.1.

Table 3.1: The SWEBOK Knowledge Areas (KAs)

Software requirements
Software design
Software construction
Software testing
Software maintenance
Software configuration management
Software engineering management
Software engineering process
Software engineering tools and methods
Software quality

Some of these KAs can in part also be assigned to related disciplines such as Project Management. For the development of our didactic modules we didn’t focus on the process-oriented activities such as requirements, quality assurance and management activities. More suitable topics are Software construction and Software testing.

3.3 Modules for Talent Scout

Based on the fundamental ideas of Computer Science and the topics of Software Engineering elaborated in the preceding sections, we developed a concept for the modules that we wrote for the CS Talent Scout. We mainly wanted to teach Software Engineering as a problem-solving discipline where also algorithmic thinking is needed. A problem is characterized by having a more or less well-defined goal, but no immediately apparent possibility to attain this goal.[18]

For the first module we chose the topic *White-box Testing*, where the students learn the concept of code coverage and apply several testing techniques. As we have seen in our survey, testing is a field which is not covered in this age group.

The second module is about the concept of *Tree Recursion*. Recursion is considered to be one of the universally most difficult concepts to teach.[13] As one of the fundamental programming paradigms, many applets exist that visualize the concept of recursion with concrete examples such as the Tower of Hanoi. While these give students a good idea of what it is, it doesn't involve much student interaction. Therefore, we intended not only to create a visualization of recursion but also providing the students with tasks, where they need to apply algorithmic thinking in the context of a recursive data structure.

The two modules will be elaborated in more detail in the following two chapters.

Chapter 4

Module White-box Testing

In June of 2002 the National Institute of Standards and Technology (NIST) released a study[29] stating that software bugs are costing the U.S. economy an estimated $59.5 billion per year. It further reported that software developers spend nearly 80% of development costs on identifying and fixing bugs. This shows the importance and cost factor of testing in the software engineering process and the improvements that still have to be achieved in this area. However, despite its significance, testing is often not sufficiently reflected in Computer Science curricula. This is also a reason why we chose the concept of testing as a topic for one of the modules.

In the module *White-box Testing* the students can familiarize themselves with some basic concepts of testing and debugging. They also learn about different metrics of code coverage (i.e. branch and path coverage) and the notion of an equivalence class and how to define one. These terms are introduced in the context of flow charts that represent simple functions with one or two input parameters. The functions consist of assignments and conditional statements that can either be in the form of an if-then-else branching or of a loop. They have either one or two input arguments (x and y) that the user needs to specify in order to test a specific path of the function. Furthermore, the local variables a and b are used sometimes in assignments. The module uses flow charts as a graphical representation of the functions in order to facilitate the understanding of the control structures and to visualize the process of testing and debugging.

4.1 Definition

Software testing is one activity of software engineering and can be defined as "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items".[19] One can distinguish between two kinds of Software testing techniques: *Black-box testing* (also known as specification-based testing or functional testing) that mainly validates the input-output behavior without any knowledge of internal implementation and *white-box testing* which makes use of the internal structure of the program to develop test cases.

White-box testing is synonymous with program-based testing or structural testing and aims to find bugs that can be discovered by source-code analysis. It can be further subdivided into the testing strategies *control-flow testing* and *data-flow testing*. The latter tries to detect data anomalies and studies the status of data objects, e.g. to assert that every variable has been initialized prior to its use. It is not included as a topic in this module. Control-flow testing on the other hand is a popular type of white-box testing whose goal it is to create test cases that satisfy a specific criterion of code coverage. Common test coverage criteria include *statement coverage*, *branch coverage* and *path coverage*. Coverage is reached when all statements, all decisions of control structures (such as an if statement) and accordingly every possible path in the function has been executed at least once. A path is defined as a unique sequence of branches from the function entry to the exit.

For this module we used branch and path coverage. Since loops introduce an unbounded number of paths, we need to restrict the analysis to a limited number of possibilities. We have chosen the simple definition of no or at least one iteration. Other sources distinguish three cases, i.e. zero, one or at least two iterations.

4.2 Tasks

The module White-box Testing is subdivided into four tabs and each of them contains a different task of testing. We present them in the following.

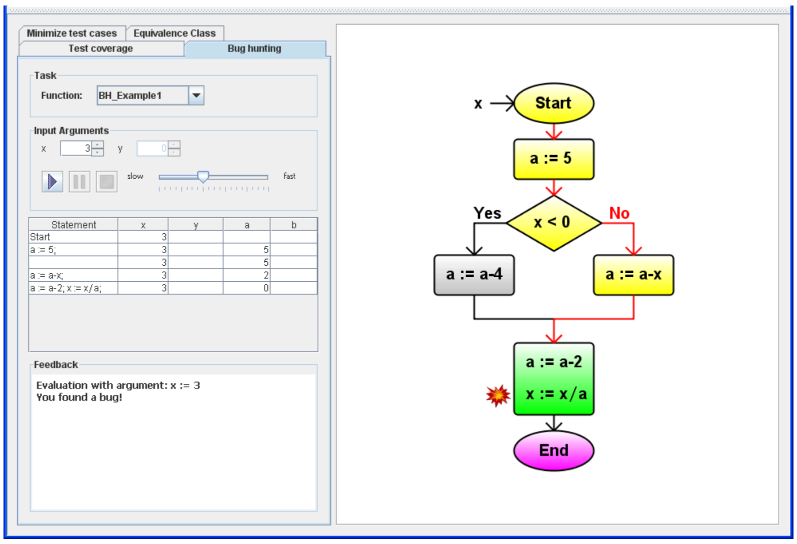
Test Coverage

In the first tab called *Test coverage*, the user's task is to specify different input arguments (x and/or y) to reach 100% code coverage. He can hereby select either branch coverage or path coverage. For any number of specified input arguments, the user is able to check the coverage that he has reached. Obviously, the needed number of input arguments depends on the selected coverage metric and the number of conditional statements that define the number of branches and paths. This task is very straightforward and allows students to familiarize themselves with the concept of the two coverage metrics and the user interface.

Bug Hunting

In the second tab called *Bug hunting*, the user has to find specific input arguments that reveal an arithmetic bug, such as a division-by-zero or a negative value under the square-root, in the function. Those simple illegal mathematical expressions have been chosen to keep things simple, as they are basic knowledge of every student in secondary school. The students basically need to consider two things. On the one hand, they need to select input arguments that will reach the branch with the critical statement. On the other hand, they need to make sure that the variables have the exact values that lead to an error.

While simple examples just involve ordinary conditional statements (i.e. *if statements*), more advanced ones also include loop statements where a certain number of iterations is necessary to exploit the bug. Figure 4.1 shows an example where we forced a division by zero.



Task
Function: BH_Example1

Input Arguments
x 3 y 0

Statement	x	y	a	b
Start	3			
a := 5;	3		5	
	3		5	
a := a-x;	3		2	
a := a-2; x := x/a;	3		0	

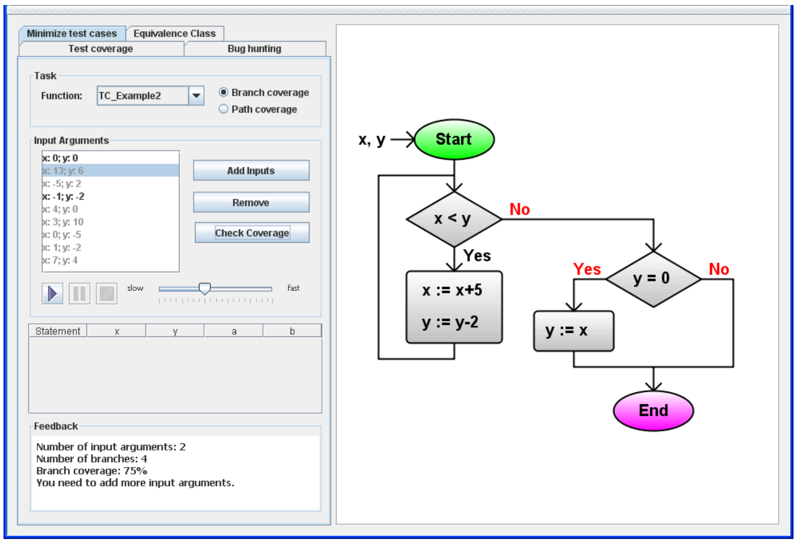
Feedback
Evaluation with argument: x := 3
You found a bug!

Figure 4.1: White-box Testing : Bug Hunting

Minimize test cases

In the third tab called *Minimize test cases*, the user is given a number of input arguments that cover all possible paths and therefore reach 100% code coverage for both branch and path coverage. The goal here is to minimize the number of test cases needed to reach 100% coverage. To do so the user has to identify and remove redundant test cases that do not increase coverage. They typically fall into the same equivalence class.

For the example shown in Figure 4.2 we selected two input arguments that reach a branch coverage of 75%. As there are four branches in the function, a careful selection would make it possible to get 100% branch coverage with just two test cases.



Function: TC_Example2
[✓ Branch coverage] Path coverage

Input Arguments
x: 0; y: 0
x: 13; y: 6
x: -5; y: 2
x: -1; y: -2
x: 4; y: 0
x: 3; y: 10
x: 0; y: -5
x: 1; y: -2
x: 7; y: 4

Number of input arguments: 2
Number of branches: 4
Branch coverage: 75%
You need to add more input arguments.

Figure 4.2: White-box Testing : Minimize test cases

Equivalence classes

The fourth tab further deals with the concept of *equivalence classes*. Randomly, one path of the function gets selected and highlighted. The task of the user is to define the equivalence class of this path, i.e. to define the range of values for the input parameters x and y that will all execute this path. The user can define any interval of integer values or positive and negative infinity.

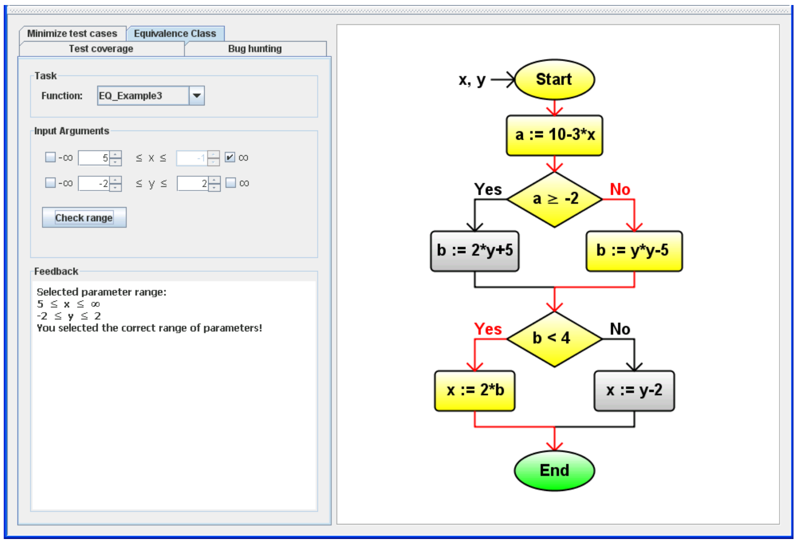
In the example shown in Figure 4.3 the user has to ensure that the condition $a \geq -2$ is evaluated to *false* and the condition $b < 4$ is evaluated to *true*. With the given assignments that leads to the equivalence class:

$$\begin{aligned} 5 &\leq x \leq \infty \\ -2 &\leq y \leq 2 \end{aligned}$$

4.3 Animation

Figure 4.4 shows the module White-box testing during the visualization of the function's step-by-step evaluation. Initially, every statement is colored in gray (with the exception of the start and end node that have the initial colors green and purple). During the evaluation, the background color of the currently executed statement changes to green. Statements that already have been executed are colored in yellow. The executed path of highlighted by red lines.

The control area on the left also contains a table listing an entry for each evaluation step. The table consists of five columns that show the assignments that have been executed in this step



Function: EQ_Example3

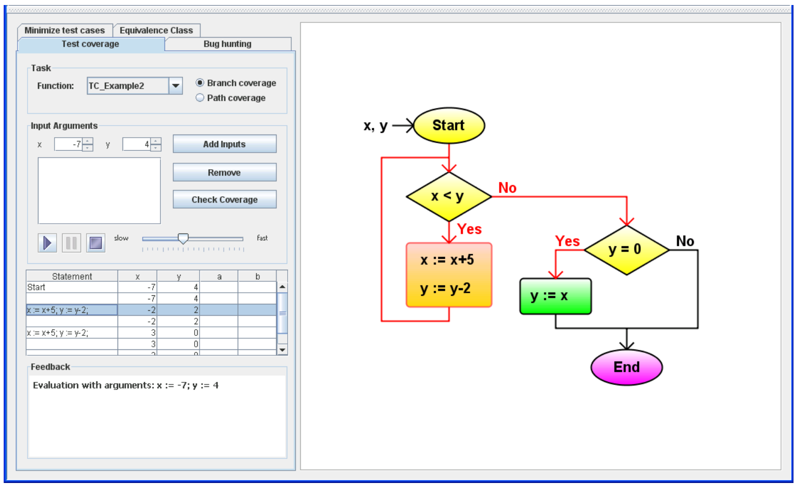
☐ -∞ 5 ≤ x ≤ -1 ☑ ∞

☐ -∞ -2 ≤ y ≤ 2 ☐ ∞

Selected parameter range:
5 ≤ x ≤ ∞
-2 ≤ y ≤ 2
You selected the correct range of parameters!

Figure 4.3: White-box Testing : Equivalence classes

together with the current values of the input parameters x and y and the local variables a and b. When the evaluation is paused or after it has finished, the user can scroll through the list in order to reconstruct the evaluation and to see why a certain path has been chosen, how many iterations of a loop have been executed or why we did or did not get a bug in the execution. The selected step is also highlighted in the draw panel. The corresponding statement box has an additional transparent red coloring on top of the yellow or green background color.



Function: TC_Example2 [✓ Branch coverage] Path coverage

x -7 y 4

Statement	x	y	a	b
Start	-7	4		
	-7	4		
x := x+5; y := y-2;	-2	2		
	-2	2		
x := x+5; y := y-2;	3	0		
	3	0		

Evaluation with arguments: x := -7; y := 4

Figure 4.4: White-box Testing : Step-by-step evaluation

4.4 Flow Chart Editor

As an extension to this module we wrote an editor to facilitate the creation of new flow charts. The editor offers a fully graphical user interface where new flow charts can either be created from scratch or existing flow charts can be opened and modified. New flow charts initially only have the start and end node and one can add statement blocks (represented by rectangle shapes) and conditional statements (represented by diamond shapes) as needed. Using the right-click context menu of the different shapes, the user can specify the successor of an element and therewith define the program sequence. Once the successor of an element has been specified, the connection lines between the shapes are placed automatically. For conditional statements, the user can select either an if-then-else, an if-then or a loop type with according placement of the connection lines to the loop body or the then branch (Figure 4.5). This type is also of importance when the flow chart is saved and the XML representation is being generated.

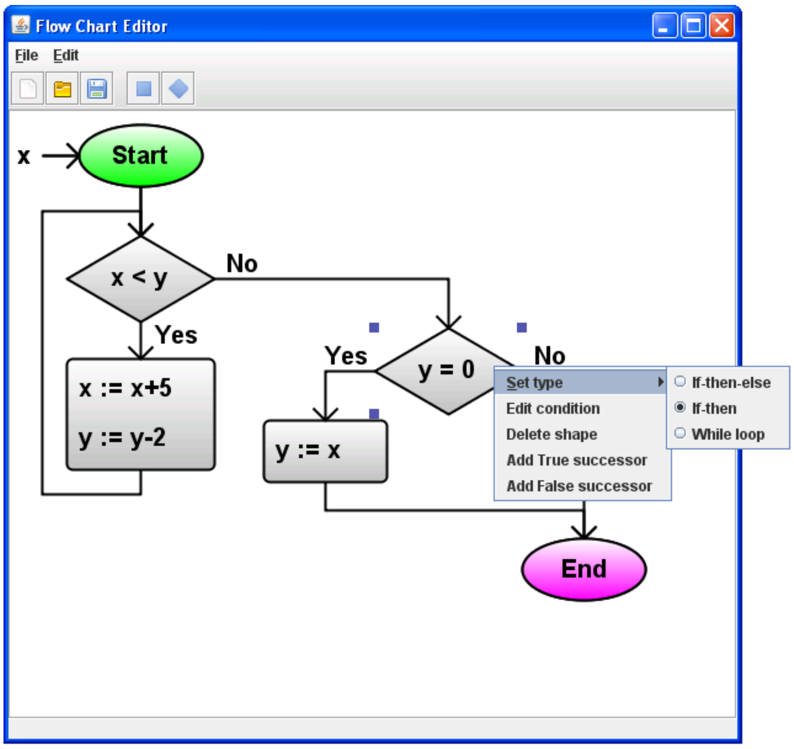


Figure 4.5: Flow Chart Editor : Context menu for conditions

Statement blocks contain an arbitrary number of assignments that the user can specify in a popup window (Figure 4.6). For editing these assignments as well as the boolean condition of conditional statements, we adapted and reused the statement editor of the second module (Section 5.4). While the height of a statement block is set according to the number of assignments contained, the width can be resized by the user by dragging the red colored handle on the right edge of the shape. The editor supports multiple shape selection which allows the simultaneous movement of several elements that keep the relative placement as long as no element touches the top or left border of the editor window.

When the user wants to save a flow chart, the flow chart model is checked for completeness (i.e. conditions and assignments) and strict sequential structure of the statements. Multiple entry points of a loop or conditional branches for instance are not supported. Before using the generated XML document, minor additions are still necessary. For the task *Equivalence classes* for example the user needs to add the definition of the equivalence class for some selected test paths. The structure of the XML documents is further elaborated in section 6.3.1 and an example XML file

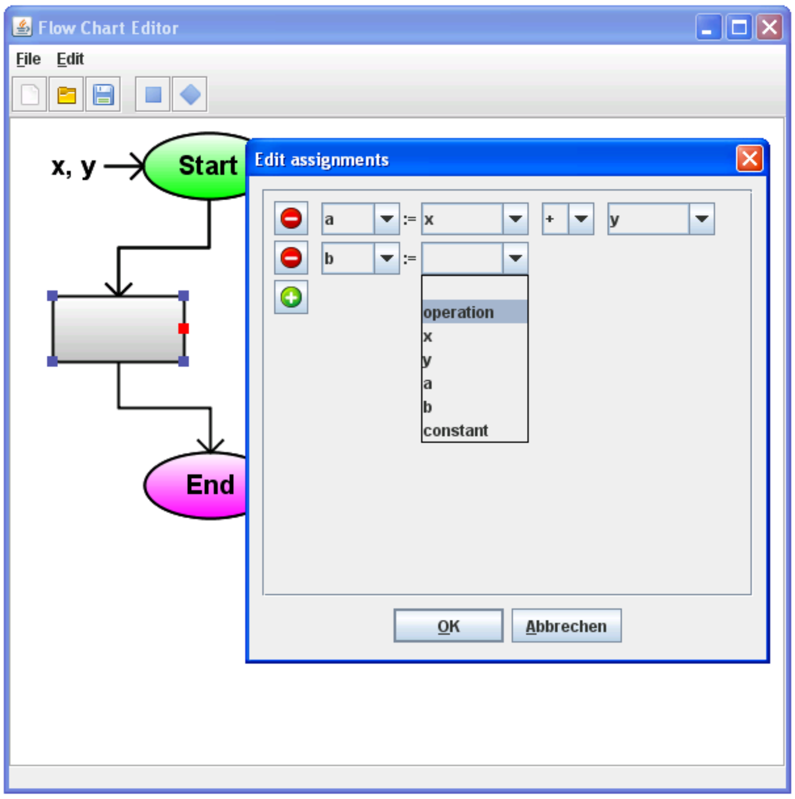


Figure 4.6: Flow Chart Editor : Edit assignments of statement block

is shown in Appendix A. The technical aspects of the editor are discussed in some more depth in section 6.3.2.

Chapter 5

Module Tree Recursion

Recursion is a very important concept used in many applications of Computer Science and Mathematics, in fact it is one of the fundamental ideas of Computer Science (Section 3.1.2). It is therefore not a new approach to facilitate the understanding of this concept by using interactive tools. But in contrast to many existing educational applications that cover the topic of recursion solely by visualization of the process, we would also like the students to create a recursive algorithm on their own in order to solve a given task.

With the module Tree Recursion we would like to teach the basic concepts of recursion in the context of the binary tree data structure. The underlying idea we would like to give the students is that we are able to solve the problem locally, independent of the overall structure. In the context of binary trees, such an algorithm needs to handle four different cases: a node with two children, a node with only one child on the left, a node with only one child on the right and a leaf node without any children. When we define a procedure for these four cases that solves the problem locally, we can apply this algorithm to the whole tree and get a solution independent of the tree structure.

Using a graphical editor (Section 5.4), students can create algorithms using three types of statements: assignments, conditional statements and recursive calls (Section 5.3). After executing a recursive call, the result of the corresponding child node can be used for further calculations.

5.1 Definition

Recursion is an important programming concept that is used in many problem-solving strategies to divide a problem into smaller sub-problems of the same type. A recursive function invokes itself several times with each recursive call reducing the problem until it is small enough to be solved directly. The smallest problem instance is called the base case and is needed as a stopping criterion to avoid infinite recursion.

Not any problem can solved using recursion. Instead, recursive functions require a specific type of problem. Niklaus Wirth described this as follows: “Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms.”[45]

The most simple type of recursion is *linear recursion* where each function invokes itself exactly once. A typical example of a function that can be expressed using linear recursion is the *factorial* function:

$$fac(n) = \begin{cases} n * fac(n-1) & \forall \text{n} > 0 \\ 1 & \text{if n} = 0 \end{cases}$$

Linear recursion can be extended to arbitrary dimensions, these patterns are known as *tree recursion* where each function invokes itself twice or even more times. A simple example for this recursive structure is the calculation of the *Fibonacci numbers*:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \forall \text{n} > 1 \\ 1 & \text{if n} = 1 \\ 0 & \text{if n} = 0 \end{cases}$$

However, some recursive algorithms are very inefficient. The Fibonacci example using the scheme above leads to many redundant computations and has exponential time complexity. Some problems can be solved more efficiently using an iterative approach. These alternative solutions as well as computational inefficiency have to be discussed, when specific recursive solutions are introduced.[41]

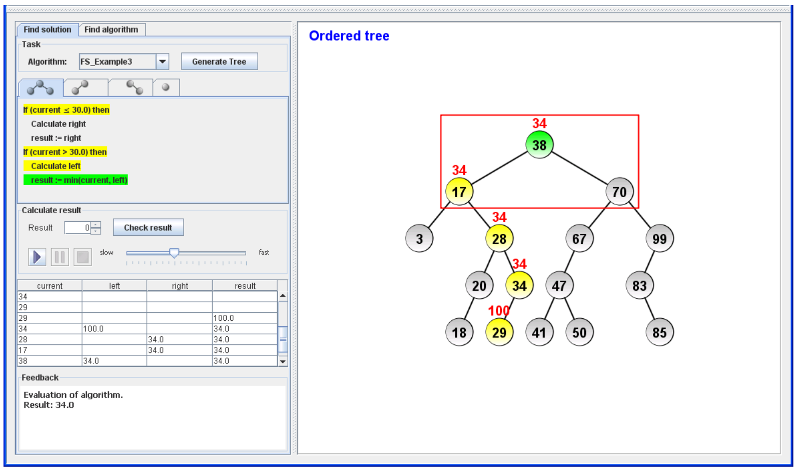
In real-world programming however, recursive algorithms are mostly used when working with recursive data structures. The most popular applications are the list data structure for linear recursion and binary trees for tree recursion. Recursive algorithms on binary trees require two recursive calls for the left and the right subtree. The base case that stops the recursion is the leaf node that has no further children.

5.2 Tasks

The module Tree Recursion is subdivided into two tabs where the user's task is either to apply or to develop a recursive algorithm. We present them in the following.

Find solution

In the first tab called *Find solution*, the goal is to understand and to apply a recursive algorithm. The user can select one of various example algorithms that are given with the textual description using assignments, recursive calls and conditional statements. When the task of the algorithm is understood correctly, the user can apply the algorithm on his own and derive the end result for the given binary tree. He is then able to check his result and to run the algorithm which is animated (Section 5.5). Figure 5.1 shows an algorithm that is looking for the smallest number greater than 30, and returns 34 for the given binary tree.



```
If (current ≤ 30.0) then
  Calculate right
  result := right
If (current > 30.0) then
  Calculate left
  result := min(current, left)
```

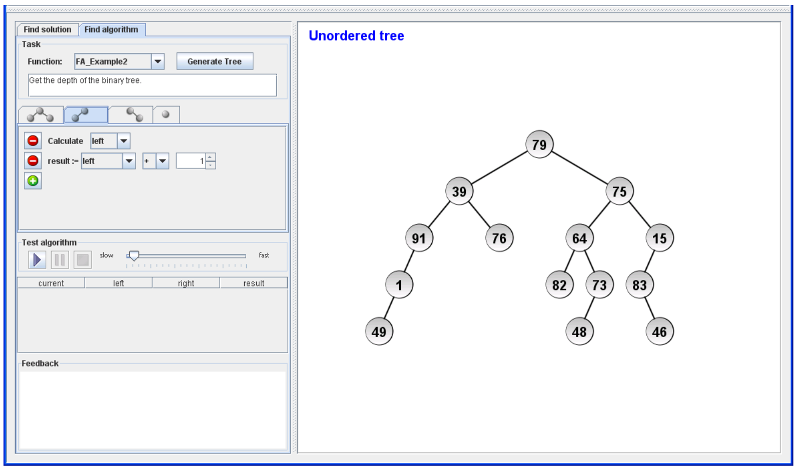
current	left	right	result
34			
29			
29			100.0
34	100.0		34.0
28		34.0	34.0
17		34.0	34.0
38	34.0		34.0

Evaluation of algorithm.
Result: 34.0

Figure 5.1: Tree Recursion : Find solution (The local case with two child nodes is selected.)

Find algorithm

In the second tab, the user is given a task and has to develop a recursive algorithm accordingly. It is essentially the inverse task of the first tab. In order to specify the algorithm, the user doesn't need to write code but rather has a convenient graphical editor that allows specifying the algorithm using only the mouse. A detailed description of the editor's functionality is given in section 5.4. Figure 5.2 shows a sample task where the user needs to calculate the depth of the binary tree.



Task

Function: FA_Example2

Get the depth of the binary tree.

Calculate left

result := left + 1

current	left	right	result

Figure 5.2: Tree Recursion : Find algorithm

5.3 Language

The recursive algorithms used in this module use a simple language that contains three types of statements (i.e. assignments, conditional statements and recursive calls) and four different variables (i.e. current, left, right and result). At the core of each algorithm are the recursive calls that allow each node to use the results of the evaluated subtree on the left or right.

The variables have a local scope corresponding to the current node in the binary tree. The variables *current*, *left* and *right* are read-only and hold the values of the current node or the result that comes from a recursive call to the subtree of either side respectively. The variable *result* on the other side is only used as target of assignments and after all statements for the current node have been executed, its value is returned to the parent node.

Figure 5.3 shows the correlation of the variables and nodes. The variable *current* holds the value 53 and the variable *left* holds the value 2. The value *right* however has not been initialized and cannot be used yet as the right subtree has not been evaluated by a recursive call yet. When a value is assigned to the variable *result*, it will become available to the root node as its local variable *right*.

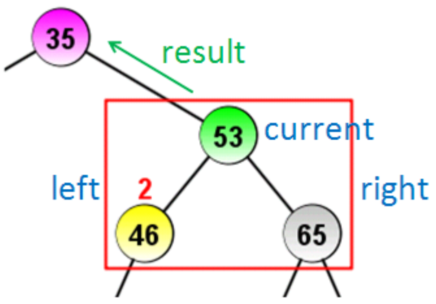


Figure 5.3: Tree Recursion : The local variables for the node with value 53

An assignment contains an arithmetic expression that uses variables, constant values, arithmetic operators and the binary functions *min* and *max* in any sequence (Listing 5.1). The conditional statement is made up of a simple condition that uses a boolean expression together with variables and constant values. The body of the conditional statement can contain again all elements of the language; therefore the algorithm can have an arbitrary nested structure. The complete language used in this module is described in EBNF in Appendix B.

```
AssignmentStmt = "result := " Expression ;
Expression = Operation | Function | Variable | Constant ;
Operation = Expression ArithmeticOperator Expression ;
Function = ( "min" | "max" ) " (" Expression ", " Expression ")" ;
Variable = "current" | "left" | "right" ;
Constant = [ "-" ] Digit { Digit } ;
```

Listing 5.1: Structure of an assignment statement in EBNF

5.4 Graphical programming editor

As we don't want to depend on previous programming knowledge of the students, we followed the approach of visual programming (Section 2.2) and created a graphical programming editor. To develop the recursive algorithms the students don't need to type code and follow a specific syntax which itself can be an error-prone activity and source of confusion. Instead, this module provides a graphical editor that simplifies creating an algorithm and avoids syntax errors. Subdivided into the four cases the algorithm needs to handle, students can specify a sequence of statements using buttons, drop down lists and spinners. We used the concept of *edit-in-place* which makes editing very quick and simple.[35]

Statements can be added and removed arbitrarily using buttons with either a plus or minus icon. A new statement starts with a first drop down list that asks the user to select the type of statement and offers assignments, recursive calls and conditional statements.

When selecting *Assignment*, a new drop down list appears to specify the expression. The user has the choice between arithmetic binary operations, simple functions (i.e. the functions *min* and *max*), the variables *current*, *left* or *right*, and a constant. The first two expressions (operations and functions) recursively ask for two subexpressions that can be specified analogical. The users can build arbitrarily nested expressions. While the variable *current* returns the value that the current node in the binary tree contains, the variables *left* and *right* contain the result of the according recursive call which therefore needs to be executed first, before using these variables. When the user decides to assign a constant, an integer value can be set using a spinner.

A *Recursive Call* allows in a second step to specify the direction, i.e. the left or the right subtree. However, this choice is only given in the first case, where the node has two children. In case we have only either a left or a right child the choice is limited accordingly, and of course for leaf nodes the recursive call is missing in the list of statement types. The result of the recursive call is stored in the according variable *left* or *right* that can be further used in arithmetic or boolean expressions.

When selecting *Condition* the user can specify a boolean expression using boolean operators and the variables *current*, *left* and *right* as well as constant values. For the body of the conditional statement, a new button appears on the next line with a certain indentation that allows to add new statements on this level.

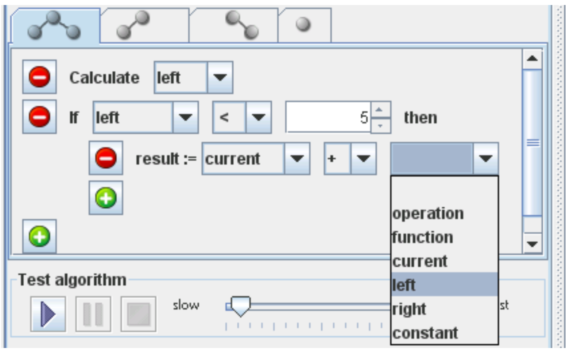
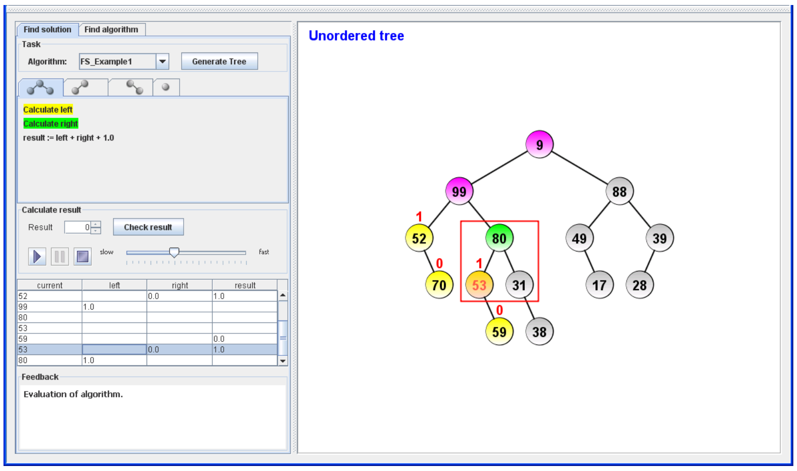


Figure 5.4: Tree Recursion : Graphical programming editor

5.5 Animation

The visualization of the algorithm execution uses a very similar concept like the first module and is shown in figure 5.5. Each node can be in one of four states that are shown by the use of different colors. Initially all nodes are *unvisited* which is represented by the gray color. *Visited* nodes where the algorithm hasn't been completed are colored purple and the *current* node is green. After the algorithm has been completely *evaluated* for a node, the background color changes to yellow. Additionally, the returned value is written in red digits on top of the node. A red rectangle around the current node and its child nodes highlights the case of the local structure (e.g. only left child).

In the control area on the left the editor always shows the current case of the algorithm and highlights the current and executed statements also with green and yellow background colors. Below we have again a table listing an entry for each execution step. This table consists of four columns that show the values of the variables *current*, *left*, *right*, and *result* for the associated tree node at the time. Again, after the execution or when it is paused, the user can scroll through the list and the corresponding nodes get highlighted in the draw panel accordingly.



Calculate left
Calculate right
result := left + right + 1.0

current	left	right	result
52		0.0	1.0
99	1.0		
80			
53			
59			0.0
53		0.0	1.0
80	1.0		

Evaluation of algorithm.

Figure 5.5: Tree Recursion : Animated algorithm execution

5.6 Random tree generation

In order to test the algorithms on binary trees with various shapes, the module Tree Recursion contains a tree generator that is able to generate random binary trees. The chosen algorithm hereby specified whether the tree needs to have an ordered structure or not which is also visible in the draw panel. For some algorithms where we're looking for a specific value, the tree needs to be ordered. The generator uses a simple random insertion algorithm with certain constraints that ensure variable depths and numbers of nodes.

When testing an algorithm that we manually created in the editor, the algorithm is tested on the given tree and additionally on five randomly generated trees in the background. Doing this, we heuristically check that the algorithm does not only return the correct result for the given tree but rather works for any random tree.

Chapter 6

Design and Architecture

The Computer Science Talent Scout library consists of a set of modules in the form of Java applets. Although each module was developed and can be run independently of the others, they are expected to follow certain guidelines to provide a consistent user interface across all modules.

In this chapter we first introduce the basic system architecture of both modules that we developed and subsequently describe the visual framework. Finally, we discuss several implementation issues.

6.1 System Architecture

Both modules have the same basic system architecture. They consist of five packages: the base package *ch.ethz.inf.csts.<moduleName>* (where *<moduleName>* is either *testing* or *treeRecursion*) and the four sub packages *.gui*, *.manual*, *.examples* and *.images*. Additionally, there is a main class in the package *ch.ethz.inf.csts.modules* with the name of the module (i.e. *Testing* and *TreeRecursion*).

- *ch.ethz.inf.csts.<moduleName>* contains the main logic of the modules. Common parts of both modules are a parser class for handling the XML files (example file in Appendix A), a sequencer class that controls the animated step-by-step evaluation, an evaluator class that evaluates a given flow chart or binary tree, and the class *Main* that is described below. Furthermore, the package includes some special data structures (e.g. *Function/Algorithm* and related classes) and some helper classes (e.g. *TreeGenerator*).

- *ch.ethz.inf.csts.<moduleName>.gui* contains as the name suggests the classes that define the graphical user interface. The class with the main JPanel is called *GUI* and gets extended by a class *Main* in the base package of the module, which combines all the components and adds the functionality to the GUI elements.

- *ch.ethz.inf.csts.<moduleName>.manual* contains classes and HTML files that are needed for the integrated manual of the modules. The manual is integrated to the module user interface with a *JSplitPane* in the main class of the module.

- *ch.ethz.inf.csts.<moduleName>.examples* contains the XML files that define the used functions or algorithms together with an XML schema (*function.xsd* and *algorithm.xsd*).

- *ch.ethz.inf.csts.<moduleName>.images* contains some icons and other images that are used in the graphical user interface.

The modules were designed according to the Model-View-Controller (MVC) design pattern in order to separate the user interface from the underlying data model.

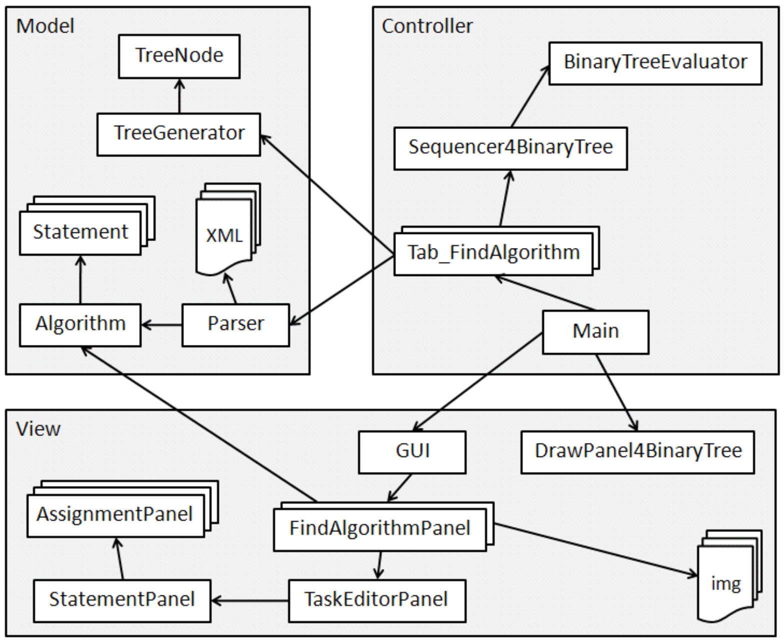


Figure 6.1: An overview of the module Tree Recursion implementing the Model-View-Controller design pattern. Rectangles represent classes and arrows stand for a direct access relationship. The underlying gray shaded areas show the division by the MVC pattern.

6.2 Visual Framework

A main design goal of our work was to create a user interface that is easy to use. Therefore we used a *Visual Framework*[42] to specify a common basic design for the user interface which also follows the overall guidelines for the CS Talent Scout. The two modules have been deliberately designed to use a consistent layout, colors, and positioning of important elements to provide an overall look-and-feel. For this reason, users don't need much time to get familiar with the user interface when switching from one exercise to another or when using a different module.

Figure 6.2 shows the visual framework used by the modules that we developed for the CS Talent Scout library. We describe the individual parts in the following.

1. The *instructions area* on top provides on the full window width a manual that contains a short introduction to the problem and instructions for its use. This manual is divided by a split pane from the actual interface of the module. Even though the manual introduces the relevant concepts and the functionality, it is still expected that a short instruction by a teacher is given in advance. The modules are not supposed to be used as an auto didactic learning tool.

2. The *control area* on the left is the location where most of the user interaction takes place. It uses a tabbed interface to divide between the different tasks of the module. The panel for each tab is again structured in a similar way: On top the user can typically select or generate the environment for the current task (i.e. the flow chart or an algorithm and a binary tree) and directly underneath we have a panel to specify the input arguments or to construct the algorithm. Next element is the navigation control for the animation that also includes a tempo slider to adjust the animation speed. The undermost elements are the variable status

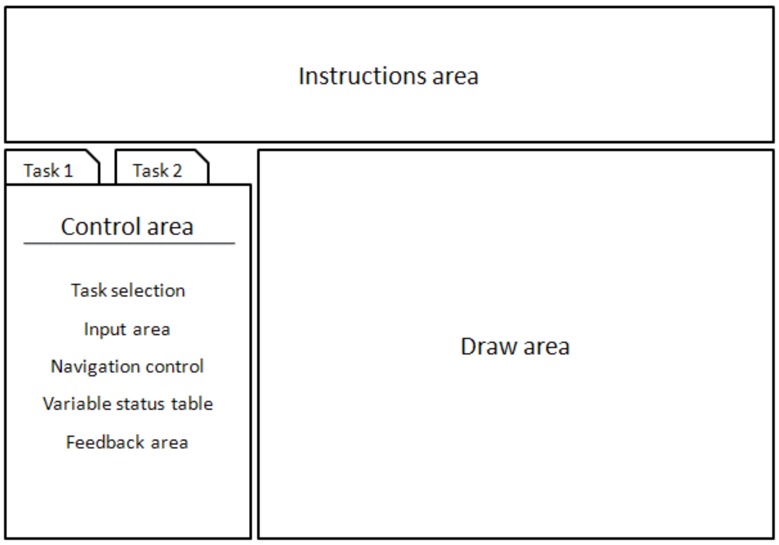


Figure 6.2: The Visual Framework

table showing the current values for each animation step and, at last, a text area to provide feedback and notifications.

3. The largest part of the visual framework is dedicated to the *draw area*. This area displays the visual representation of the function as a flow chart or the binary tree that is used for the recursive algorithms. It is animated during the execution (Section 4.3 and 5.5).

6.3 Implementation issues

In this section we discuss some selected implementation issues.

6.3.1 Flow chart representation

Initially, we were considering specifying a simple, pseudo-code like language to specify the functions that we used for the module White-box testing. This would have implied that we need to write our own text file parser together with a lexer and semantic analyzer and finally create an internal data structure for the function. Furthermore, this would also require a smart drawing algorithm that translates the function into a flow chart that we can draw in the module. This however seemed like too much effort simply to end up with a graphical representation of a function. For this reason we decided to include the drawing information into the file that describes a flow chart. Essentially, we just need to know the position and size of the elements.

Due to the wide availability of XML parsers and the ease of use we shortly chose to represent the flow charts by XML files. Java offers two popular interfaces to process XML files: The Document Object Model (DOM) interface and the Simple API for XML (SAX) interface. However the latter does not create an in-memory representation of the XML file. Therefore, we used the DOM parser called *DocumentBuilder* to create a DOM Document that we used as an internal structure to draw and evaluate the flow charts (Figure 6.3). To check the correct structure of an XML we created an XML schema file (*functions.xsd*).

The XML files are structured as follows: The root element is named *function* and has an attribute that references to the schema file. The first child elements specify the number and names of the input parameters. We consistently named them x and y. After the parameters an element named *dataflow* follows which contains the main information of the file. The first child element of dataflow is always the *start* node which is followed by a sequence of assignment blocks,

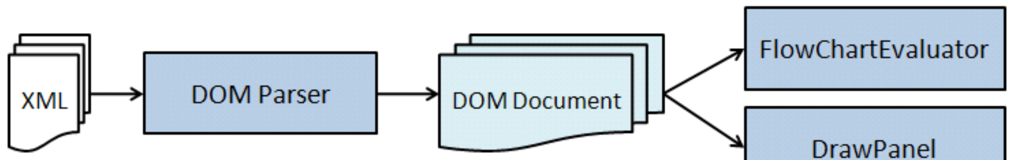


Figure 6.3: The flow chart representation

if and while statements and finally the *end* node. All statements (including start and end node) have the three attributes *id*, *position* and *size*. The identifiers are later used as a reference for the connection lines.

An assignment block can contain an arbitrary number of *assignment* elements as children. The assignment element also has a position attribute and an attribute denominating the target variable. It has one child element which can be a nested arithmetic expression consisting of variables, constants and arithmetic operators.

```
<block id="102" position="5,225" size="100,50">
    <assign name="b" position="20,256">
        <expr type="times">
            <const value ="2" />
            <variable name="x" />
        </expr>
    </assign>
</block>
```

Listing 6.1: XML representation of an assignment block

The if statement has the three child elements *condition*, *then* and *else*, where the last one is optional. The condition element again has a position attribute and an attribute specifying the boolean operator. It has two arithmetic expressions as child elements. The then and else elements can recursively contain a sequence of statements. Very similarly structured is the while statement. It has two child elements: *condition* and *do* that also contains a sequence of statements.

After the dataflow element, which is typically the largest part of the XML file, we have the element *graphsize* which is used to center the flow chart in the draw panel. At the end of the XML file we have the element *lines* which contains all connection lines of the flow chart. Both the element *line* and *arrow* have the attributes *from* and *to* with the position information as well as *refFrom* and *ref* with the reference to a statement node of the flow chart. The references are used for highlighting the executed path of the flow chart.

```
<lines>
    <arrow from="155,60" to="155,80" ref="101" refFrom="100" />
    <arrow from="155,130" to="155,150" ref="1" refFrom="101" />
    <line from="95,185" to="55,185" ref="102" refFrom="1" />
    ...
</lines>
```

Listing 6.2: XML representation of lines

Depending on the usage of the function, the XML file further contains an element *testpaths* which specifies the equivalence classes of all paths that can be defined linearly. Other XML files contain an element *testcases* which lists some sample input arguments where the user needs to minimize the number of arguments needed for full testing coverage.

The detailed XML structure is shown in Appendix A which contains the sample XML file *EQ_Example1.xml*. We also used XML files to specify the recursive algorithms for the module Tree Recursion. They have a similar, yet simpler structure.

6.3.2 Flow Chart Editor

The XML structure of the flow charts is very convenient for file I/O and to use it as an internal data structure to operate on. However, the creation of new examples is not. Writing the XML file by hand is a tedious and repetitive activity. Therefore, we decided to write a tool to generate these files for us. The two options we considered were a graphical editor and as an alternative an automated formatter that takes as an input the textual representation of a function. The automated formatter however would have to consider many special cases and seemed too unflexible. We rejected this idea basically for the same reasons we preferred the XML files over pseudo-code text files in the first place.

In the following we describe the architecture of the Flow Chart Editor by package and hereby mention the most important classes.

- *ch.ethz.inf.csts.flowchartEditor* contains the main class *FlowchartEditor* that sets up a JFrame and coordinates actions across the editor. The package furthermore includes the class *FlowchartModel* that keeps track of all flow chart elements and distributes the unique identifiers to them. It has two constructors that either create an empty model or create a model based on a DOM document. It also offers the method *generateModelDocument()* to create a DOM document from the model.

- *ch.ethz.inf.csts.flowchartEditor.images* contains some icons that are used in the toolbar and the assignment editor.

- *ch.ethz.inf.csts.flowchartEditor.ui* contains the classes *DrawPanel* and *Toolbar* that are part of the editors user interface. It also includes various shape classes that represent elements of the flow chart model, i.e. *ShapeRectangle* for assignment blocks, *ShapeDiamond* for if and while statements and *ShapeStart* and *ShapeEnd* with common super class *ShapeEllipse* for the end and start node of the flow chart. The abstract super class *Shape* provides a number of methods related to the position and size of the element and also keeps track of its selection state and the successor shape. Each extension of Shape overrides the methods *paint(Graphics2D g2)* to draw itself and *contains(int x, int y)* that checks if a given point is inside the specific shape (Listing 6.3). Some shapes implement the *PopupGenerator* interface that provides a context menu on a mouse right click.

```java
@Override
public boolean contains(int x, int y) {
    if (!super.contains(x, y)) {
        return false;
    }

    // move point of origin to center of ellipse
    double e1 = (double) w / 2;
    double e2 = (double) h / 2;
    double dx = x - (this.x + e1);
    double dy = y - (this.y + e2);

    double distSquared = dx * dx / (e1 * e1) + dy * dy / (e2 * e2);
    return distSquared <= 1.01;
}
```

Listing 6.3: Method *contains* in class *ShapeEllipse*

Depending on the shape properties two different selections are used: the *SelectionMove* which can only be moved and the *SelectionResize* which additionally provides a handle to resize the selected shape. Most of the UI functionality is implemented in the class *SelectionManager* which implements the interfaces *MouseListener*, *MouseMotionListener* and *KeyListener*. It handles all selections and generally most user interactions. It distinguishes three special modes: *dragMode* for dragging all selected shapes, *resizeMode* when a shape is being resized and *createLineMode* when we are selecting a successor shape to add a new connection line. When the mouse button is pressed on an empty spot and then dragged, a selection rectangle is drawn to select shapes. The implementation of method *mouseDragged* is shown in listing 6.4.

```
@Override
public void mouseDragged(MouseEvent e) {
    if (resizeMode) {
        // we resize the corresponding shape
        resizeSelection.resize(e);
    } else if (dragMode) {
        // we move all selected shapes
        for (Shape s : selectedShapes) {
            s.getSelection().mouseDragged(e);
        }
    } else {
        // we draw the selection rectangle
        int x = e.getX();
        int y = e.getY();

        showSelectRect = true;

        int boundX = Math.min(selectAnchor.x, x);
        int boundY = Math.min(selectAnchor.y, y);
        int boundW = Math.max(selectAnchor.x, x) - boundX;
        int boundH = Math.max(selectAnchor.y, y) - boundY;

        selectRect.setBounds(boundX, boundY, boundW, boundH);
    }
    editor.repaint();
}
```

Listing 6.4: Method *mouseDragged* in class *SelectionManager*

- *ch.ethz.inf.csts.flowchartEditor.ui.actions* contains various extensions of *AbstractAction* that are mainly used in the editor menu and toolbar but also in the context menu of graphical elements. Sample actions include *OpenAction* that opens a JFileChooser dialog to create a flow chart from an XML file, *EditAssignmentsAction* that opens an window to edit the list of assignments of a statement block or *DeleteSelectionAction* that deletes all selected shapes in the editor.

- *ch.ethz.inf.csts.flowchartEditor.ui.editor* contains the editor panels for conditions which are used for conditional statements as well as for a list of assignments to specify the content of statement blocks. For these editor panels we adapted and reused the editor from module Tree Recursion (Section 5.4).

- *ch.ethz.inf.csts.flowchartEditor.xml* contains the utility classes *Parser* and *XMLGenerator* to create a DOM document from an XML file and vice versa.

Because we only need one instance of the the class *FlowchartEditor* and we use it to coordinate actions across the editor (e.g. in most AbstractActions that we defined), we used the *Singleton* pattern[14] to remove the necessity to provide it as an argument to many objects. Listing 6.5 shows the implementation of the Singleton pattern. We didn't use *lazy creation* as we always need an instance of the FlowchartEditor class.

```
public final class FlowchartEditor extends JFrame {

   /** A handle to the unique Singleton instance */
   private static final FlowchartEditor instance =
                              new FlowchartEditor("Flow Chart Editor");

   /**
    * Private constructor prevents external instantiation.
    */
   private FlowchartEditor() {}

   private FlowchartEditor(String title) {
      [..]
   }

   /**
    * @return The unique instance of this class.
    */
   public static FlowchartEditor getInstance() {
      return instance;
   }

}
```

Listing 6.5: Class *FlowchartEditor* implementing the Singleton pattern

6.3.3 Graphical programming editor

Programming is a difficult skill to acquire. It requires strong analytical and abstract thinking and is best learned by practice. When students first encounter programming, their motivation is very diverse.[20] But if students are to learn effectively, they must to be motivated so that they will engage appropriately. The approach of visual programming is very appealing especially to programming novices. Furthermore, we wanted to keep the complexity of the programming task low, as the concept of recursion was the main focus of our module Tree Recursion.

The visual programming editor we wrote has a tabbed interface with a *TaskEditorPanel* for each of the four cases the recursive algorithm needs to handle. The constructor of this class takes the type as an input parameter in order to restrict the available statements (e.g. *TYPE_LEAF* removes the recursive calls from the list of statements). The class contains a list of StatementPanels that are vertically aligned. A *StatementPanel* consists of a JButton (to add another statement and on the second click to remove itself) and a selectionPanel that contains a combo box to select a statement type. When a type is selected the selectionPanel is replaced by the panel of the specified statement, i.e. *RecursiveCallPanel*, *ConditionPanel* or an *ExpressionPanel* to specify the arithmetic expression of an assignment (Listing 6.6).

```
public StatementPanel(int type, TaskEditorPanel taskEditorPanel) {
   [..]

   if (type == TaskEditorPanel.TYPE_LEAF) {
      statementList = new String[] { SELECT_STMT, ASSIGNMENT, CONDITION };
   } else {
      statementList = new String[] { SELECT_STMT, ASSIGNMENT, RECURSIVE_CALL, CONDITION };
   }

   statementComboBox = new javax.swing.JComboBox();
   statementComboBox.setModel(new javax.swing.DefaultComboBoxModel(statementList));
   statementComboBox.setMaximumSize(new java.awt.Dimension(125, 25));
   statementComboBox.addItemListener(new ItemListener() {
      @Override
      public void itemStateChanged(ItemEvent e) {
         statement = (String) e.getItem();
```

```
            if (ASSIGNMENT.equals(statement)) {
                addAssignment();
            } else if (RECURSIVE_CALL.equals(statement)) {
                addRecursiveCall();
            } else if (CONDITION.equals(statement)) {
                addCondition();
            }
        }
    });
}

/**
 * Adds a recursive call statement.
 */
private void addRecursiveCall() {
    statementComboBox.setVisible(false);
    recursiveCallPanel = new RecursiveCallPanel(type);
    selectionPanel.add(recursiveCallPanel);
}
```

Listing 6.6: Constructor and method *addRecursiveCall* in class *StatementPanel*

The class StatementPanel as well as each panel of a specific statement has a method *getStatement()* that returns the specified statement. While the RecursiveCallPanel only contains one combo box to specify the direction of the recursive call, the ExpressionPanel can recursively add other ExpressionPanels to create nested expressions. Similarly, the ConditionPanel contains three combo boxes to specify the boolean expression and additionally a *ThenPanel* that can create an arbitrary number of statements like the TaskEditorPanel. These statements are also added to the TaskEditorPanel that provides each statement with a unique ID and keeps track of them in a hash table. This is needed to highlight the executed and current statements for the current tree node during the algorithm execution, which is done by the method *setStep(TreeNode node)*. Figure 6.4 shows how the various panels are created.

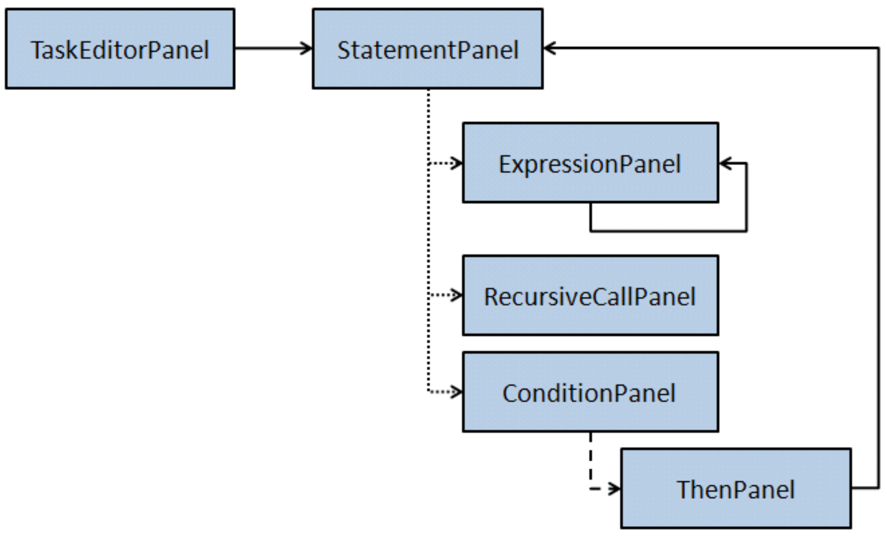


Figure 6.4: The structure of the visual editor. A solid line means that an arbitrary number of elements can be created, a dashed line means exactly one element can be created and a dotted line means one element of a selection can be created.

Chapter 7

Conclusion

We have implemented two Software Engineering modules for the Computer Science Talent Scout together with a suite of interesting problems. The module White-box testing visualizes the functions to be tested as flow charts and lets students become familiar with the process of testing and debugging by revealing arithmetic exceptions in the given functions. It further introduces the concept of code coverage and equivalence classes. The module provides a step-by-step animation of the flow chart evaluation that enables students to fully comprehend the work flow.

We also developed a flow chart editor to quickly and easily create new flow charts using a graphical interface. Assignment blocks and conditional statements can be added, positioned and connected by mouse actions in order to define the program sequence. The user can specify the arithmetic and boolean expressions in popup windows by selecting entries from combo boxes. When a flow chart is completed, the editor is able to generate the XML file that can be used in the module as a new problem instance.

The module Tree Recursion is concerned with recursive algorithms in the context of the binary tree data structure. It breaks down the algorithms into four local cases and allows students to solve the problem locally and independent of the overall tree structure. These local procedures are then recursively applied to the randomly generated tree in order to get a global solution. The module comes with a graphical programming editor that allows creating algorithms without typing code in order to simplify the task and to avoid syntax errors. The step-by-step algorithm evaluation on the binary tree is animated likewise to the other module. We have used a consistent visual framework across both modules that is easy to use and provides appealing visual representations of the tasks.

As a future work it would be useful to do an evaluation of the developed modules with some high school classes. Besides evaluating the understanding of the involved concepts, the evaluation also has to record the motivation of the participants, e.g. if they find the interface easy to use and visually appealing and the tasks interesting and challenging. It would also be interesting to include an assessment of the students' knowledge of a Computer Science course of study and to find out whether they consider pursuing it and for what reasons.

We now extended the collection of modules for the Computer Science Talent Scout by Software Engineering topics. However, there are still several fields of Computer Science uncovered and one can think of various applications, e.g. in the area of Visual Computing or Distributed Systems, that remain as possible future extensions.

Appendix A

Sample XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<function
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation='function.xsd' >
    <parameter name="x" />
    <dataflow>
        <start id="100" position="105,10" size="100,50" />
        <block id="101" position="95,80" size="120,50">
            <assign name="a" position="110,111">
                <expr type="minus">
                    <const value ="4" />
                    <expr type="times">
                        <const value ="2" />
                        <variable name="x" />
                    </expr>
                </expr>
            </assign>
        </block>
        <if id="1" position="155,150" size="120,70">
            <condition type="less" position="130,191">
                <variable name="a" />
                <const value ="0" />
            </condition>
            <then>
                <block id="102" position="5,225" size="100,50">
                    <assign name="b" position="20,256">
                        <const value ="10" />
                    </assign>
                </block>
            </then>
            <else>
                <block id="103" position="205,225" size="100,50">
                    <assign name="b" position="220,256">
                        <expr type="times">
                            <const value ="2" />
                            <variable name="a" />
                        </expr>
                    </assign>
                </block>
            </else>
        </if>
        <end id="104" position="105,335" size="100,50" />
    </dataflow>
    <testpaths>
        <path x="inf,2" />
        <path x="3,inf" />
    </testpaths>
    <graphsize size="320,390" />
    <lines>
        <arrow from="155,60" to="155,80" ref="101" refFrom="100" />
        <arrow from="155,130" to="155,150" ref="1" refFrom="101" />
        <line from="95,185" to="55,185" ref="102" refFrom="1" />
        <arrow              to="55,225" ref="102" refFrom="1" />
        <line from="55,275" to="55,305" ref="104" refFrom="102" />
        <line               to="155,305" ref="104" refFrom="102" />
        <line from="215,185" to="255,185" ref="103" refFrom="1" />
        <arrow              to="255,225" ref="103" refFrom="1" />
        <line from="255,275" to="255,305" ref="104" refFrom="103" />
        <line               to="155,305" ref="104" refFrom="103" />
        <arrow from="155,305" to="155,335" ref="104" />
    </lines>
</function>
```

Listing A.1: EQ_Example1.xml

Appendix B

EBNF of recursive algorithm language

```
Algorithm = Case Case Case Case ;
Case = Statement { Statement } ;
Statement = AssignmentStmt | RecursiveCallStmt | CondStmt ;
AssignmentStmt = "result := " Expression ;
RecursiveCallStmt = "call " Side ;
CondStmt = "if " Condition " then " { Statement } ;
Expression = Operation | Function | Variable | Constant ;
Side = "left" | "right" ;
Condition = Variable BoolOperator ( Variable | Constant ) ;
Operation = Expression ArithmeticOperator Expression ;
Function = ( "min" | "max" ) " (" Expression ", " Expression ")" ;
Variable = "current" | "left" | "right" ;
Constant = [ "-" ] Digit { Digit } ;
BoolOperator = "<" | "≤" | "=" | "≠" | "≥" | ">" ;
ArithmeticOperator = "+" | "-" | "*" | "/" ;
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Listing B.1: Extended Backus Naur Form of recursive algorithm language

Bibliography

[1] Alain Abran, Pierre Bourque, Robert Dupuis, and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Computer Society, Los Alamitos, CA, USA, 2004.

[2] Asli Yagmur Akbulut and Clayton Arlen Looney. Inspiring students to pursue computing degrees. *Commun. ACM*, 50(10):67–71, 2007.

[3] Jonathan Anderson and Tom van Weert. Information and Communication Technology in Education: A Curriculum for Schools and Programme of Teacher Development, 2002. UNESCO.

[4] Doug Baldwin. Discovery learning in computer science. In *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 222–226, New York, NY, USA, 1996. ACM.

[5] Mordechai Ben-Ari. Constructivism in computer science education. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 257–261, New York, NY, USA, 1998. ACM.

[6] Malte Blumberg et al. j-Algo, The Algorithm Visualisation Tool. http://j-algo.binaervarianz.de.

[7] Markus Brändle. *GraphBench: Exploring the Limits of Complexity with Educational Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2006.

[8] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. Mini-languages: a way to learn programming principles. *Education and Information Technologies*, 2(1):65–83, 1998.

[9] Lori Carter. Why students with an apparent aptitute for Computer Science don't choose to major in Computer Science. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer Science Education*, pages 27–31, New York, NY, USA, 2006. ACM.

[10] Computer Science Teachers Association. A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force, October 2002.

[11] Matthew J. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, University of Virginia, 1997.

[12] John Dewey. *How We Think*. D.C. Heath & Co., Boston, MA, USA, 1910.

[13] Judith Gal-Ezer and David Harel. What (else) should CS educators know? *Commun. ACM*, 41(9):77–84, 1998.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1994.

[15] Idit Harel and Seymour Papert. *Constructionism*. Ablex Publishing, Norwood, NJ, USA, 1991.

[16] W. Hartmann, J. Nievergelt, and R. Reichert. Kara, finite state machines, and the case for programming as part of general education. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 135, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Christian Holmboe, Linda McIver, and Carlisle George. Research Agenda for Computer Science Education. In *Proceeding of the 13th Annual Workshop of the Psychology of Programming Interest Group*, pages 207–223, Bournemouth, UK, 2001.

[18] Ludger Humbert. *Didaktik der Informatik*. Teubner Verlag, Wiesbaden, 2005.

[19] IEEE. *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std.610.12-1990*. IEEE Computer Society, 1990.

[20] Tony Jenkins. The motivation of students of programming. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 53–56, New York, NY, USA, 2001. ACM.

[21] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. Storytelling Alice motivates middle school girls to learn Computer Programming. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1455–1464, New York, NY, USA, 2007. ACM.

[22] Kristian Kiili. Digital game-based learning: Towards an experiential gaming model. *The Internet and Higher Education*, 8(1):13–24, 2005.

[23] Paul A. Kirschner, John Sweller, and Richard E. Clark. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist*, 41(2):75–86, 2006.

[24] Lifelong Kindergarten, MIT Media Lab. http://llk.media.mit.edu/.

[25] Logo Foundation. Logo. http://el.media.mit.edu/logo-foundation/.

[26] David J. Malan and Henry H. Leitner. Scratch for budding computer scientists. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 223–227, New York, NY, USA, 2007. ACM.

[27] Richard E. Mayer. Should there be a three-strikes rule against pure discovery learning? the case for guided methods of instruction. *American Psychologist*, 59(1):14–19, 2004.

[28] Emily Oh Navarro and André van der Hoek. SimSE: An interactive simulation game for software engineering education. In *Proceedings of the 7th IASTED International Conference on Computers and Advanced Technology in Education*, pages 12–17, Kauai, HI, USA, 2004. ACTA Press.

[29] Michael Newman. Software Errors Cost U.S. Economy $59.5 Billion Annually. NIST Assesses Technical Needs of Industry to Improve Software-Testing. http://www.nist.gov/public_affairs/releases/n02-10.htm, June 28, 2002.

[30] Jürg Nievergelt. Was ist Informatik-Didaktik? Gedanken über die Fachkenntnisse des Informatiklehrers. *Informatik Spektrum*, 16(1):3–10, 1993.

[31] Jürg Nievergelt and Aurea Perez. CS Talent Scout – a self-assessment aptitude test for Computer Science. In *Proceedings of the IADIS International Conference e-Learning 2007*, 2007.

[32] Seymour Papert. *Mindstorms: children, computers, and powerful ideas.* Basic Books, Inc., New York, NY, USA, 1980.

[33] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.

[34] Marc Prensky. Digital game-based learning. *Comput. Entertain.*, 1(1):21–21, 2003.

[35] James Rice, Adam Farquhar, Philippe Piernot, and Thomas Gruber. Using the Web instead of a window system. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 103–110, New York, NY, USA, 1996. ACM.

[36] Susan H. Rodger. Introducing computer science through animation and virtual worlds. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 186–190, New York, NY, USA, 2002. ACM.

[37] Guido Rössling, Markus Schüer, and Bernd Freisleben. The ANIMAL algorithm animation tool. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pages 37–40, New York, NY, USA, 2000. ACM.

[38] Madeleine Schep and Nieves McNulty. Use of lego mindstorm kits in introductory programming classes: a tutorial. *J. Comput. Small Coll.*, 18(2):323–327, 2002.

[39] Schweizerische Konferenz der kantonalen Erziehungsdirektoren. Strategie der EDK im Bereich Informations- und Kommunikationstechnologien (ICT) und Medien, March 2007.

[40] Andreas Schwill. Computer science education based on fundamental ideas. In *Proceedings of the IFIP TC3 WG3.1/3.5 joint working conference on Information technology : supporting change through teacher education*, pages 285–291, London, UK, UK, 1997. Chapman & Hall, Ltd.

[41] Ivan Stojmenovic. Recursive Algorithms in Computer Science Courses: Fibonacci Numbers and Binomial Coefficients. *IEEE Transactions on Education*, 43(3):273–276, August 2000.

[42] Jenifer Tidwell. *Designing Interfaces.* O'Reilly Media, Inc., 2005.

[43] Vincent Tscherter. *Exorciser: Automatic generation and interactive grading of exercises in the theory of computation.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2004.

[44] Anja Umbach-Daniel and Armida Wegmann. Das Image der Informatik in der Schweiz, April 2008. Study by order of the advancement program FIT in IT of the Hasler foundation.

[45] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

[46] Andreas Zendler and Christian Spannagel. Empirical foundation of central concepts for computer science education. *J. Educ. Resour. Comput.*, 8(2):1–15, 2008.