FORESTRY AND NATURAL SCIENCES

Andrés Moreno

Re-designing Program Animation

From tools' roles to new learning activities

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences No 149

UNIVERSITY OF
EASTERN FINLAND

ANDRÉS MORENO

Re-designing Program Animation

From tools' roles to new learning activities

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences
No 149

Academic Dissertation
To be presented by permission of the Faculty of Science and Forestry for public examination in the Auditorium AU100 in Aurora Building at the University of Eastern Finland, Joensuu, on September, 5, 2014,
at 12 o'clock noon.

School of Computing

Grano Oy
Joensuu, 2014
Editors: Profs. Pertti Pasanen, Pekka Kilpeläinen,
Kai Peiponen, and Matti Vornanen

Distribution:
University of Eastern Finland Library / Sales of publications
julkaisumyynti@uef.fi
http://www.uef.fi/kirjasto

ISBN: 978-952-61-1542-9 (printed)
ISSNL: 1798-5668
ISSN: 1798-5668
ISBN: 978-952-61-1543-6 (pdf)
ISSNL: 1798-5668
ISSN: 1798-5676

Author's address:
University of Eastern Finland
School of Computing
P.O.Box 111
FI-80101 JOENSUU
FINLAND
email: amoreno@cs.uef.fi

Supervisors:
Professor Erkki Sutinen, Ph.D.
University of Eastern Finland
School of Computing
P.O.Box 111
FI-80101 JOENSUU
FINLAND
email: erkki.sutinen@uef.fi

Associate Professor Mike Joy, Ph.D.
University of Warwick
Department of Computer Science
CV4 7AL COVENTRY
UNITED KINGDOM
email: M.S.Joy@warwick.ac.uk

Reviewers:
Associate Professor Michael E. Caspersen, Ph.D.
Aarhus University
Faculty of Science and Technology
C.F. Møllers Allé 8, Building 1110
DK-8000 AARHUS C
DENMARK
email: mec@cse.au.dk

Professor Carsten Schulte, Ph.D.
Freie Universität
Department of Mathematics and Computer Science
Königin-Luise-Str. 24-26
14195 BERLIN
GERMANY
email: schulte@inf.fu-berlin.de

Opponent:
Professor Steven P. Reiss, Ph.D.
Brown University
Department of Computer Science
P.O.Box 1910
RI 02912-1910 RHODE ISLAND
USA
email: spr@cs.brown.edu

ABSTRACT

Learning programming successfully requires not only the acquisition of knowledge, but also skills and attitudes. Program animation tools have been developed for students to help gaining the required skills and knowledge by means of visualizing the various aspects of program execution. Despite developers' and researchers' intentions, the learning impact of program animation tools does not correspond to long-term efforts in developing and researching them. While learning impact has mostly been quantitatively evaluated, the process in which students acquire the knowledge and skills using program animation tools is not documented in detail. In particular, the learning impact of Jeliot 3, a program animation tool, has been assessed quantitatively and Jeliot 3 has proved its effectiveness for certain cohorts of students –those not very strong nor very weak. However, reasons and processes of why Jeliot 3 may improve students' learning are only suggested in previous research.

In this thesis, the challenges faced by students when using Jeliot 3 to learn programming have been looked deeply at. The conducted observations have directed the development of new prototypes derived from Jeliot 3, and of new learning activities based on the tool. The research follows a systems development methodology as it is centered in the creation of an efficient artifact to teach programming. Two qualitative studies of English and Tanzanian students using Jeliot 3 prompted the development of several prototypes: Jeliot Adaptive, Jeliot with Explanations, and Jeliot Conflictive Animations. Two of the prototypes, Jeliot with Explanations and Jeliot Conflictive Animations, were empirically evaluated using a between-subject design with pre-tests and post-tests to assess the impact the prototypes had on the students' learning.

The qualitative studies revealed the roles the animation tool could take when used by a student learning a new programming concept. Five roles were identified: empty, exploring, confusing, teaching, and evaluating. The roles of the tool were adopted by the students at different stages in the student's learning path. To

promote the importance of the animations and to avoid the empty role of the tool, conflictive animations were devised. Conflictive animations require the student to find the errors contained in the animation, rather than in the code. However, the empirical evaluation only showed a modest gain in effectiveness when learning about class inheritance using Jeliot Conflictive Animation compared to normal Jeliot 3. In another prototype, Jeliot with Explanations, explanations were added to the animations produced by Jeliot 3. It was found that the explanations were more effective when placed after the animation, rather than before the animation.

The field of program animation has been advancing gradually with iterations of the same idea: improving the interaction and information of animation tools that is presented to the students. These incremental innovations improve the learning impact of the tools, as in the case of Jeliot with Explanations. However, new approaches, like conflictive animations, are needed to advance the field in ways no imagined before. In this thesis, the foundation for future activities and animations based on conflictive animations are laid, and one of the possibilities is explored further: Jeliot ConAn. Conflictive animations, and errors in general, need to be explored further as a way to improve knowledge, skills and attitudes of programming students.

Universal Decimal Classification: 004.42, 37.091.3, 378.147
INSPEC Thesaurus: computer science education; educational technology; educational computing; computer aided instruction; teaching; training; programming; program visualisation; computer animation; error detection
Yleinen suomalainen asiasanasto: opetusteknologia; tietokoneavusteinen opetus; oppimisympäristö; oppiminen; tietokoneavusteinen oppiminen; tietojenkäsittely; ohjelmointi; vasta-alkajat; animaatio; visualisointi

Acknowledgements

These long years of research could have not ended if it were not for the support and encouragement of the people I have close to me, both in my head and in my heart.

This whole trip started with Erkki Sutinen's suggesting me to take risks in life. I took the risks back then without knowing how hard he will try to ease them for me. The time we have spent together sharing ideas, results and conflicts on two continents have shaped me as a researcher and as a person, and for that I am grateful of the opportunity to take the risks. I want to thank Mike Joy because he was pushing me to finish this work. All this time I knew I could rely on his prompt and helpful feedback when I needed it most: whenever I lost track.

I am thankful to my colleagues, brilliant and friendly minds, from whom I have learnt a lot: Niko Myller started the trip with me and held my hand as I delved into the academic world; Roman Bednarik's passion for science and attention to detail improved my work in many ways, long days at the department felt shorter after a game or two with him; Michael Yudelson let me peek in his adaptive world and adapted it to my purposes when I needed it; Peng Wang helped me to think of Jeliot 3 in new ways that I had not thought of before, his help was important to finish my research puzzle; and Carolina Islas-Sedano planted the gaming seed in me long time ago to only be harvested recently in our collaboration. Many others at the department of Computer Science have made this a rewarding trip: Ilkka, Marcus, Teemu to name a few.

I thank Carsten Schulte and Michael Caspersen for agreeing being the reviewers of my work, and providing helpful critique and comments. I am honored and thankful to have Steven Reiss as an opponent.

This research has been supported in parts by the Department of Computer Science Department, University of Joensuu, Finland, by

the East Finland Graduate School in Computer Science and Engineering (ECSE), Finland, and by the North-South-South Program, CIMO Finland. Research reported here has taken place in Joensuu, Finland, University of Warwick, UK, and Iringa University College, Tanzania. Preliminary work has been done at the University of Helsinki, Finland. In all these places I am indebted to the local administrative teaching staff and, specially, their students, without whom there would not have been anything to report. In Joensuu, endless chats with my friends about life, research, and friendship filled my evenings and lifted some of the worries from my shoulders, *majos*, you know who you are. Elsewhere, the Jeliot family, specially Mordechai Ben-Ari and Ronit Ben-Bassat Levy, provided great feedback about my work that reflected positively in my motivation.

I want to dedicate this work to my family, back in Spain, who during all these years away from home have nurtured myself and my desire to research and find answers on how people learn to program. Finally, I want to thank my loved one, Paula, for making me smile after sleepless nights, working against deadlines and my own fears. She did me good because she knows best.

Helsinki August 13, 2014 *Andrés Moreno García*

LIST OF PUBLICATIONS

This thesis consists of the present review of the author's work in the field of optical interconnects and the following selection of the author's publications:

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

Paper **II** A. Moreno, M. Joy, and E. Sutinen, "Roles of animation tools in understanding programming concepts," *Journal of Educational Multimedia and Hypermedia* **22,** 165–184 (2013).

Paper **III** A. Moreno, R. Bednarik, and M. Yudelson, "How to Adapt the Visualization of Programs?," in *Proceedings of Workshop on Personalisation in E-Learning Environments at Individual and Group Level, 11th International Conference on User Modeling* (2007), pp. 65–70.

Paper **IV** A. Moreno, M. Joy, N. Myller, and E. Sutinen, "Layered Architecture for Automatic Generation of Conflictive Animations in Programming Education," *IEEE Transactions on Learning Technologies* **3,** 139–151 (2010).

Paper **V** A. Moreno, E. Sutinen, and M. Joy, "Defining and Evaluating Conflictive Animations for Programming Education: The Case of Jeliot ConAn," *in Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14* (2014), pp. 629–634.

Paper **VI** P. Wang, R. Bednarik, and A. Moreno, "During automatic program animation, explanations after animations have greater impact than before animations," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12 (2012), pp. 100–109.

Paper **VII** A. Moreno, E. Sutinen, and C. Islas Sedano, "A game concept using conflictive animations for learning programming,"

(2013), *in International Games Innovation Conference (IGIC)*, (2013), IEEE, pp. 175–178.

Throughout the overview, these papers will be referred to by Roman numerals. Publications I - VII have been included at the end of this thesis with their copyright holders' permission.

AUTHOR'S CONTRIBUTION

The publications selected in this dissertation are original research papers on program animation. The author was the main contributor to all the manuscripts but of Paper I. For Papers I-V,VII, the author has implemented the presented educational software and collected the research data. Paper VI is based on Peng Wang's Master's thesis [127] that the author guided in the implementation of the explanations within the program evaluation tool and the empirical evaluation of the modified tool.

LIST OF FIGURES

LIST OF TABLES

Contents

1 Introduction

This thesis explores the role of program animation tools in knowledge creation by students and the opportunities to re-design an existing program animation tool, Jeliot 3. With the lessons learnt from the research, modifications to Jeliot 3 were developed and evaluated. Finally, conflictive animations, a new direction for programming education, are proposed to promote conceptual learning, programming skills, and attitudinal changes in students learning programming.

Programming education has been an issue since the first programmable computer was built. Somebody needed to teach other people how to make the machine compute as intended. Soon universities created the first computer science degrees, a mix of mathematics, formal logic, physics, and engineering [118]. From that moment, student enrolment on computer science degrees grew in western universities till just recently, when the tech bubble exploded. Recent reports reveal the problems that teachers have been aware of since the beginning: college students face great challenges when understanding the basic concepts in programming [66]. In parallel with the increased number of computer scientists, programmers started to create tools to teach programming, as if it were natural for programmers to solve their educational problems by programming.

In 1998 Dijkstra was presented with one of those learning tools, which used visualizations [26], and his reaction was not positive at all. Already in 1975, Mayer [64], had devised a visual model of computers that he used to teach programming concept to total novices. His evaluation resulted in mixed results. On the one hand, the students who used the visual model based learning material were better in some aspects (solving problems that required interpretation). On the other hand, students in the control group were better in other aspects (solving problems that required programming similar problems to the one showed in the learning materials). His re-

sults showed promise for the idea of complementing programming education with visual models that students may relate to.

Since then, researchers and developers have implemented *software visualization* tools, which follow the path marked by Mayer. They have tried to fulfil the promises and expectations of visual models of computing in programming education and practice, aiming at better educated students and better software produced. This thesis focuses on the educational side of programming, where software visualization is divided between the tools that animate general programming paradigms, *program visualization*, and the tools that focus on algorithms, *algorithm visualization* [46].

Program visualization tools bring to the surface the experts' view on the machine that executes the program. This view consists of visual representations and metaphors that explain how the code is structured, in the case of UML tools, or how the code is executed, in the case of visual debuggers. Program visualization tools, and specially the subset of *program animation* tools, add the possibility to replicate the dynamic nature of program execution, which students need to understand to create or debug programs.

Expert programmers, who either design the visualizations or teach using them, find the visualizations self-explanatory. They can follow them and present them to students. What about the students, who later will have the opportunity to engage with the visualizations? Do they understand the visualizations? Do they follow them? Evaluations usually report if learning has happened after using the visualization, but one cannot imply causation unless the process of learning with visualization tools is better explained.

Sajaniemi *et al*. [111] studied the impact of visualization tools in the graphical representations of program executions that students drew. In three separate drawing tasks, students' representations of program state were found to be most frequently independent of the previous visualizations demonstrated to them. While many reasons could explain this behaviour, it reinforces the idea that students have a different understanding, or mental model, of the computer than the experts. Thus, should bridges between the two under-

standings be built via visualizations?

To check on the expectations and promises of educational visualization tools, Hundhausen *et al.* [42] did a *meta-evaluation* of empirical evaluations of visualization tools. As in Mayer's early results, the evidence was certainly positive, albeit a bit mixed. The success of visualisations depended, according to Hundhausen *et al.*, on how the students engage with the tool, rather than what the students watched. The importance of the activities done with the tools resulted in a taxonomy of engagement modes with visualization tools [86], which categorized engagement from *no-viewing* to *teaching*, *i.e.*, from students not using visualization tools at all to students using the tools to teach other students.

It is implicit that, as the students use more engaging visualization tools, their grades after using the tools will improve. Urquiza-Fuentes and Velázquez-Iturbide [122] and Sorva [117] have reviewed and evaluated tools in different categories of the taxonomy. Again, their findings are mixed. Partly due to only collecting successful evaluations of visualization tools, Urquiza-Fuentes and Velázquez-Iturbide could not point to a higher pedagogical effectiveness of tools in higher levels of the engagement taxonomy. However, they recommended that certain features, not necessarily graphical, were required for the tools to make visualizations effective, for example explanations.

In this thesis the research and development has been based on the program animation tool Jeliot 3, for which the author was part of the development team. Designing and developing new tools is in the nature of computer scientists, and when researching the issue of understanding visualizations it was natural for the author to develop new solutions based on Jeliot 3.

Jeliot 3 [75], a program visualization tool, is one of the long-standing program visualization tools that is meant for novices. In its decade-long history [6] several iterations and evaluations have been made. Evaluations have pointed at the benefits of using the tool, but also that the tool does not work the same for everyone.

One can think that mixed evaluation outcomes are expected in

any educational evaluation, small changes in instruction can have high impact, and conversely, large changes can lead to no significant impact. In any case, research proposes explanations and leads to new developments.

Developing tools for software visualization is a task akin to compiler development, even *meta-programming*. While the programming language is fixed, the development of the tools forces to think about the language in ways that it may not have been considered before, thus opening new ways to visualize programs. The result of this line of thinking leads to the nature of conflictive animation, which is based on fine modifications to the programming language interpretation so that it is different from the language standard.

1.1 FOCUS OF RESEARCH AND RESEARCH QUESTIONS

The focus of research reported in this thesis is divided into two main parts. First, the thesis aims to understand the current impact of program animation tools in students learning to program, both on how the students use the tool and how the students understand new concepts. Secondly, Jeliot 3's modular architecture is used to design, develop and experiment with new animation concepts. Three different software solutions are researched to increase students' engagement and understanding of programming concepts when using program animation tools: adaptation, explanations, and conflictive animations.

Benefits of program animation have been reported [8] but the literature rarely establishes the role of animation tools in forming students' programming knowledge. The thesis picks up from previous reports of students failing to use program animation tools, such as Kannüsmaki *et al*. [44], and presents several roles that programming tools have in different scenarios. Two main questions lead this part.

QUESTION 1. How do novice students engage in using Jeliot 3 when learning new programming concepts?

Students learning to program face a complete new set of challenges that they probably have seldom if ever seen before: abstract and logical reasoning, complex problem solving, debugging and tracing. Program animation tools are meant to facilitate the learning process by presenting a visual model of the program execution. However, the tools and their visualization are also novel to the students and are used in different ways to solve the learning challenges. In the process the tool takes different roles to support learning.

QUESTION 2. How do novice students understand the visualizations provided by Jeliot 3 when learning new programming concepts?

Teachers use program animation tools in lectures and students use it at home to work on their assignments.

If the student engages with the program animation tool, it is expected that the student improves their understanding of the demonstrated concepts after repeated visualizations. Research has shown that this is not the case for every student [111]. Program animation tools use visual metaphors to represent the execution. The metaphor introduces new elements that the student has to comprehend at the same time that the new concept being learnt. Thus, when the conceptual knowledge is fragile, the student's understanding of both the animation and the concept behind varies.

Derived from the first part, the second part introduces new visualization concepts and pedagogical improvements that are designed and implemented using Jeliot 3's modular architecture. The effect of the new versions of Jeliot in student engagement and understanding are evaluated in two cases.

QUESTION 3. How can new features be implemented in Jeliot 3 using its modular architecture in a way that facilitates its usage in diverse learning scenarios?

Jeliot 3 was designed to be modular, so new features could be added easily, unlike it predecessor Jeliot 2000. This ease of adding new features to Jeliot, allows for quick prototyping of new solutions to overcome difficulties and problems that teachers and students find when using Jeliot 3. Through the addition of several features, a refined process of ideation, conceptualization, implementation and evaluation is crystallized in the development of Jeliot ConAn, a new version of Jeliot 3 that creates erroneous animations.

The thesis answers a final question that combines both parts of the research into a single track.

QUESTION 4. What effect has the newly implemented features on students' engagement and understanding of new programming concepts?

1.2 RESEARCH METHODS

As presented in the research questions, the thesis aims to answer questions related to two main topics: design and implementation of visualization tools and evaluation of visualization tools. Each of the two topics feeds each other with the results of the other topic.

The focus of the research is in the development of visualization tools. As such, systems development research methodology was chosen for the research [91]. Four main components of the methodology can be defined, see Figure 1.1: *system development* itself, *theory building*, *observation* and *experimentation*. Observation aims to get an understanding of the field, while experimentation aims to validate the theories built or the developed systems. Theory building includes the development of new ideas, concepts or models. In

contrast to classical epistemological questions, the task of this thesis is not to enquire into what is true, but rather *what is effective*, as proposed by Hevner [38].

Systems development defends the use of other methodologies to answer the questions in each of the components. In this thesis, then, **Question 3** corresponds to the system development component and new visualization tools have been prototyped, so that they could be fed to the other three components.

The observation (**Questions 1 and 2**) and experimentation (**Question 4**) components link with classical education research, in which one uses either qualitative or quantitative methods according to the research question posed. On the one hand, quantitative methods reflect a positivist view of reality: a unique reality exists and can be evaluated and measured. On the other hand, qualitative researchers aim to explore the meaning of the observed reality. These particular observations depend on the observer: qualitative researchers acknowledge the subjective interpretation of reality.

To answer these three questions (**Questions 1,2 and 4**) I have taken a mixed methods approach, in which the method is chosen depending on the question being asked, similar to the pragmatic approach taken by Sorva [117]. A mixed methods approach recognises the fact that different phenomena may be elicited by different means. As well, the power of combining qualitative and quantitative methods is increased by the triangulation of the results [8, 58].

The questions related to the observation component (**Questions 1 and 2**), are better suited to be answered with qualitative research methods, as they provide a deeper understanding of the field. They were used to explore the students' understanding of programming constructs and the previously undefined roles a programming visualization tool takes when learning them. Two experiments were carried out with different students. In both of the experiments students were asked to explain a certain programming concept just taught. In turn, the analysis of their verbal answers revealed the role the tool had in their learning. Verbal data have been used before to investigate problem solving and mental models [20, 29]. Ver-

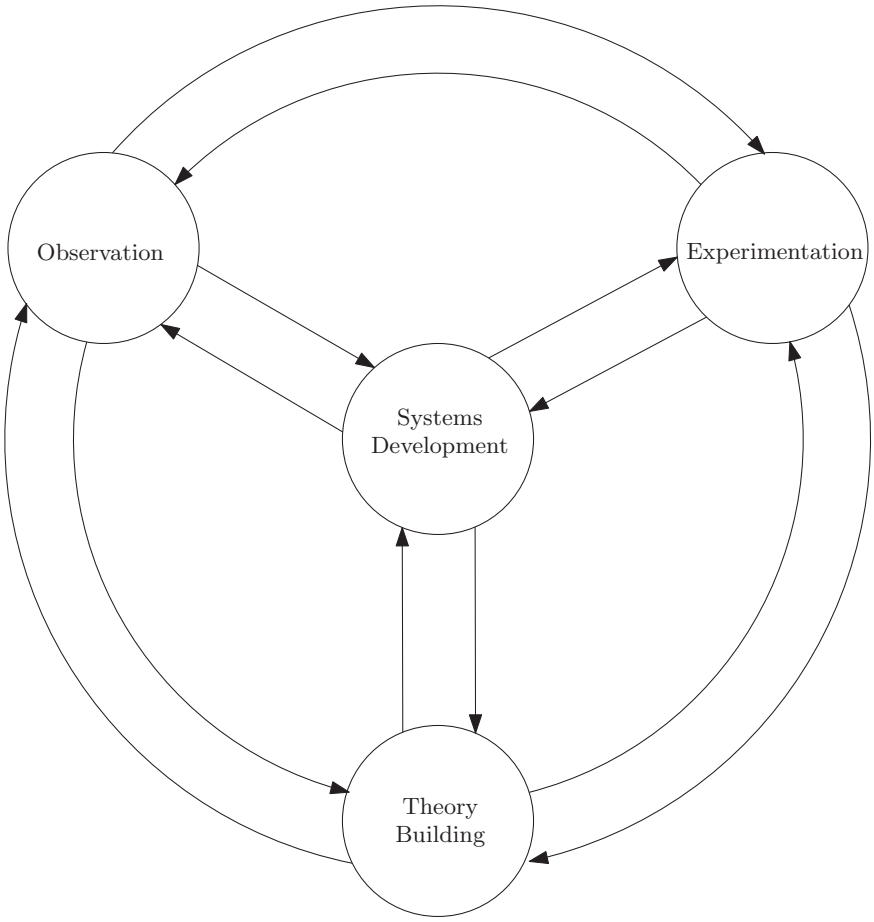

Figure 1.1: The main components of Systems Design Research method, modified from Nunamaker et al. *[91]*

bal protocols provide more data from the interventions than written feedback and can reflect the thinking process subjects go through when coming with an explanation, answer, or description.

In the first experiment, two possible uses of Jeliot 3 were suggested, as a debugger and as a learning aid, and their importance evaluated by interviewing students after using the tool. The interviews were transcribed and analysed looking for patterns of the usefulness of Jeliot 3 and how students had used it.

To understand better the roles of Jeliot 3 as a learning aid, in the following evaluation, six students from a programming course were randomly sampled. From the transcriptions of their learning sessions, roles were inferred from transcriptions using an inductive category generation process, a similar technique also used by Eckerdal [28]. Qualitative data were obtained by capturing the ongoing visualization and, in parallel, students' descriptions of that visualization. This technique is a combination of the interview methods that Fleury [32] and Holmboe [40] used to elicit programming knowledge from students. Fleury asked students in the interviews whether a given program will work and and why they think so [32]. Holmboe led the interviews with a simpler set of questions phrased like "What is ..." followed by a programming concept [40]. The result of the combination is adapted to the dynamic nature of animations with the question "What happens next?", which was used as a prompt in Paper II.

For the experimentation component, **Question 4**, quantitative methods have been preferred as they can better answer the *what's effective* question. I have investigated the learning effects of two experiments that used modified versions of Jeliot 3. Specifically, the design chosen has been single-factor, between-subjects experiments. Data were gathered by means of pre-tests and post-tests, and feedback questionnaires. Pre-tests and post-tests have been previously used in research on the educational impact of visualizations, *e.g.* Hansen *et al*. [37]. Moreover, Hundhausen *et al*. suggest in their meta-study that this kind of evaluation is more likely to find an impact if it exists [42] than a single post-test measurement. How-

ever, they warned that the pre-test may be considered as part of the learning and impact the results. In the studies presented here, the impact of the pre-test can be partly disregarded as the pre-test is the same for all the subjects, and the pre-test is presented shortly before the treatment.

Data gathered for the qualitative and quantitative methods were augmented with students' descriptions of animations. These descriptions were written by the students next to printed screenshots of Jeliot, and collected after watching the animations and answering the post-test or verbally describing a similar animation. To my knowledge, this kind of data has not been analysed before to study students' understanding of programming constructs, nor the animation itself. Vainio and Sajaniemi used the opposite approach — analysis of students' drawings of programming concepts — with interesting results [123]. In the experiments presented here, written descriptions were considered necessary to complement the qualitative verbal descriptions as students could have trouble watching the animation and talking at the same time, two cognitively demanding tasks. Quantitive test scores benefited from the insights provided by students' descriptions. Descriptions forced students to use their programming knowledge to describe the visualizations.

To evaluate the engagement levels provided by the new program visualization concepts, and to answer **Question 4**, heuristic analysis has been used with conflictive animations concept. Nielsen proposed heuristic evaluation as a low-cost and fast method for finding usability problems [89]. Here, it has been adapted to evaluate the fun-factor of animations and to increase the students' engagement with the animations.

In all empirical evaluations, ecological validity of the results has been of utmost importance, resulting in sacrificing statistical significance when not enough experiment subjects could be found. Evaluations were done in situations in which the author could have control, *e.g.* being the teacher, so that the experimental sessions were positioned at the right time in the students' process of learning programming. This way, the results are ecologically valid in the sense

Table 1.1: The relationship between research questions, systems development research component, and paper including the research method used

Research Question	Systems development component	Method	Paper(s)
1	Observation	Qualitative evaluation	I, II
2	Observation	Qualitative evaluation	I, II
3	Systems development	Prototyping	III,IV,V,VII
4	Experimentation	Quantitative evaluation, Heuristic Analysis	IV, VI, VII

that they are obtained from a learning scenario as close as possible to a real one. Thus, they can be generalized to other similar learning scenarios. The empirical evaluation described in Paper 6 was carried out by Peng Wang and ecological validity could not be ensured.

Regarding the theory building component of systems development research, it is embedded in all the papers included in this dissertation. My research builds new theories, models and artifacts either as a precursor for empirical results or as a by-product of the research, thus theory building is intertwined in the research process.

Tables 1.1 and 1.2 summarize and link the research papers, research methods and papers presented in this thesis.

1.3 RESEARCH PROCESS AND MAIN RESULTS

The research process has been mostly linear, with some branching and pruning, rather than iterative. Figure 1.2 shows a schema of the process that I explain herein.

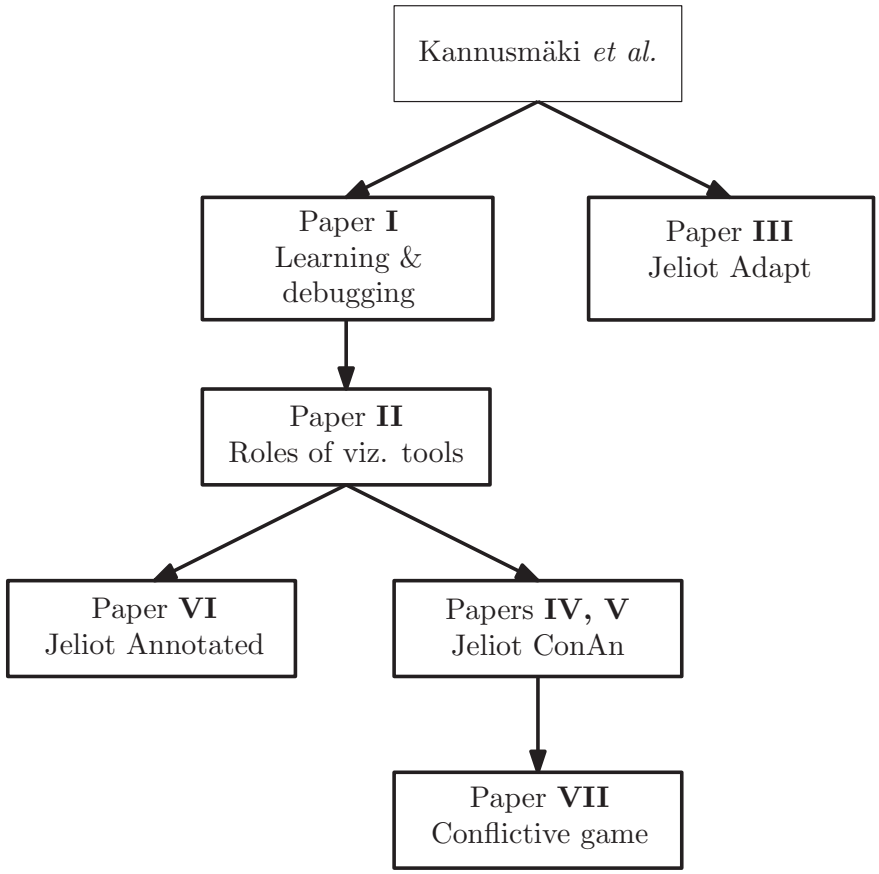

Figure 1.2: Research process graph

Table 1.2: Summary of research methods and research questions by paper

Paper	Method	Research Questions
I	Qualitative evaluation	1, 2
II	Qualitative evaluation	1, 2
III	Constructive research	3
IV	Constructive research and quantitative evaluation	3, 4
V	Constructive research	3
VI	Quantitative research	4
VII	Constructive research, Heuristic Analysis	3

The work started from the realization that some students refused to use program animation tools and that some of those who used them found them too complex to benefit from [44]. The first idea considered for investigation was to consider a smart version of Jeliot 3 [3]. The first experiment, reported in Paper I and carried out at the University of Warwick (UK), aimed to find ways on how to adapt Jeliot 3 to cater for different kinds of students and situations. The experiment revealed that some students indeed have problems understanding and viewing the animations. Specially, they were not able to explain the meaning of animations they had seen several times before. While adaptation was thought to be a possible solution it was not clear from the study how to apply it.

Anyway, collaborating with the Personalized Adaptive Web Systems Lab from the University of Pittsburgh (USA), the first adaptation features for program animation were designed and implemented in Jeliot 3 (Paper III). Under closer inspection the adaptive solution was discarded and not evaluated due to the minimal increase in engagement it would provide compared to the work needed to provide adaptation features to a large set of Java con-

structs.

The first experiment served as a base for a second one, carried out at Tumaini University (TZ) and described in Paper II. This time, the role of Jeliot 3 was explored more deeply to understand the issues discovered in Paper I. To overcome the difficulties of students using Jeliot as a learning aid, two solutions were implemented and evaluated. One solution, adding explanations, was natural and already done in other animation systems. The other, creating conflictive animations, was innovative and with lots of potential to change the computer science education field.

One Master's student, Peng Wang, implemented and evaluated the solution with explanations under my guidance [127]. Given that explanations are already proven to be effective for multimedia learning [65], the evaluation focused on the point of time that the explanation would be shown to the student: before or after the explained step of the animation.

The idea of conflictive animation was introduced first in 2007 [77] and developed further in Papers IV (system design) and V (empirical evaluation carried out at the University of Joensuu (FI), with a pilot at the University of Helsinki (FI)).

Finally, aiming to take the idea of conflictive animations further and to overcome students' criticisms, a new activity merging conflictive animations and games was conceived and presented in Paper VII.

All the steps taken in the process have led to concrete results that are summarized in the following section and, in more detail, in Chapter 4, which includes the limitations inherent to the results (Section 4.5).

1.3.1 Summary of the results

1. Engagement and understanding

    (a) Understading of new programming concepts with Jeliot 3

Description Students face problems when using Jeliot 3 to understand new concepts. While students may finally comprehend the topics explained by Jeliot 3, they have difficulties to link the visualization with the concept being explained, partly due to their previous fragile knowledge.

Validity Two experiments in completely different settings have informed this qualitative result.

(b) Roles of program visualization tools

Description Program visualization tools can take five changing roles as students use them to understand new concepts: empty role, explanatory, confusing, teaching, and evaluating.

Validity One qualititave experiment was used to infer the roles. Results depended in the previous knowledge and understanding of the author at that time. However, the analysis process was guided and reviewed by the supervisors of this thesis at every step to ensure that the roles inferred followed logically from students' protocols.

2. Jeliot Adapt

(a) Adapting visualizations of a whole programming language is hard

Description The development work carried out in Jeliot Adapt was abandon due the difficulty of automatically adapting the visualization of every step depending on the students' changing knowledge.

Validity The result is only pertinent to Jeliot 3, but the fact that there are not adaptive visualization tools that cover a whole programming language seem to support the result.

3. Jeliot ConAn

(a) Extended Engagement Taxonomy with Conflicts

Description The engagement taxonomy from Naps *et al.* [86] has been duplicated to consider the engagement levels that conflicts allow in visualization tools.

Validity This is a theoretical result that can guide future developments of tools using conflicts or errors.

(b) Jeliot ConAn

Description An architecture for creating conflicts in program visualization tools and prototype of Jeliot ConAn were built to automatically create conflicts on a set of programming concepts.

Validity The prototype was tested on existing book examples and found to automatically animate most of them without changing them.

(c) Jeliot ConAn does not improve students' understanding

Description When compared to a normal Jeliot 3 activity, debugging a small program using the animation, Jeliot ConAn students did not significally improve their understanding of the programming concept. Anecdotal evidence show that Jeliot ConAn helped with students' meta-cognitive skills.

Validity A small size experiment did not provide statistical strength to the result. As well, experiment group had not used Jeliot ConAn before, nor introduced to the error based teaching.

4. Jeliot 3 and Explanations

(a) Explanations in Jeliot 3 help students when placed after the animation

Description A version of Jeliot 3 that adds explanations to the naimation steps was developed together with a Master's student. An experiment found that students apprehended more of the explained concept

and its animation when the explanation is placed just after the step being explained.

Validity A small size experiment that yielded significant results. A larger size experiment should be done to validate the results.

1.4 ORGANIZATION OF THE THESIS

The rest of the thesis is divided as follows: Chapter 2 introduces the field of programming education and summarizes the theories that I have used. Chapter 3 describes the field of program animation, though program and algorithm visualization tools are also discussed. The research and implementation methods of these tools divide that chapter. A special section in that chapter is devoted to Jeliot 3, the tool used in this research process. Chapter 4 summarizes the results that are presented in the papers of the thesis. The thesis concludes with Chapter 5, in which new openings for programming animation, and programming education in general, are presented. As well, recommendations are given for the developers of visualization tools and programming teachers.

2 Learning and Teaching Programming

University teachers have struggled for the last decades trying to find the right combination of teaching methods, materials and tools to improve the grades of their programming students. These students enroll in programming courses at their beginning of the computer science or engineering degrees. Their knowledge, skills and attitudes clash with those that teachers expect from them. The teachers' teaching performance and students' learning is evaluated using students' grades as a proxy. The results have not been good: programming courses face high failure and dropout rates. Two studies concisely reflect the problem. A study on students' programming skills by McCracken *et al.* [66] assessed the skills of students from several institutions and countries. Their findings pointed to students not having the expected skills after their programming course, according to them the lack of problem solving skills were to blame. Later on, Lister *et al.* [59] explored the issue of the lack of program comprehension and tracing skills of students, fundamental to debugging [96], and, according to Lister *et al.*, a prerequisite for problem solving. They found that students lacked program comprehension and tracing skills. Because of these results, apart from the fact that researchers happened to find what they were looking for, computer science educators and researchers have not yet found the elements for successful teaching and learning of programming at universities.

The rest of this chapter introduces the theoretical elements and the previous research that sustain the theory building, observation, and experimentation components of this research; these components are presented in Chapter 4. Interested readers can refer to Sorva's dissertation [117] for a deeper presentation of the theoreti-

cal discussion.

2.1 THEORETICAL COMPONENTS OF LEARNING AND TEACHING PROGRAMMING

Different learning theories emphasize different components of the learning process and, at the same time, they promote different kinds of activities to support the development of those aspects. *Behaviourism* postulates that knowledge, or learnt behaviour, can be transferred from an instructor to the learner by means of repeating the information or stimuli. This implies the assumption that there is only one possible understanding and that the instructor possesses it. Learning programming conventions, syntax rules, and other basic programming knowledge is well suited to this theory.

With multiple-choice questions, a teachers and evaluators get an authoritative answer to whether the student has learnt the topic. Moreover, automatic grading and fairness have also been mentioned as a benefit [51]. These properties have led researchers to add multiple-choice questions to several tools for teaching computer science [85]. As Kuechler and Simkin mention, multiple-choice question only capture one dimension of learning [51]. Other methods and theories are needed.

Cognitive theories try to open the black box that the brain represents for behaviourists. Cognitive psychologists aim to formulate (or invent) the structures and processes that support and make people learn. Specially for programming, their idea of *mental models* [90] is quite relevant as it also captures the dynamic nature of programs.

Resources that support the creation and modification of mental models have been guided by *cognitive constructivism* theory. The focus of *social constructivism* is on students constructing their own knowledge and advancing through their individual *Zone of Proximal Development* with the help of teachers and peers. The teacher's mission is to guide and scaffold students' learning. Students construct their knowledge mainly by means of exploration, and they

are encouraged to express what they have found. In this setting, errors and misunderstandings are accepted, and even encouraged, as solving them will help the students create solid knowledge [113].

Constructivist acceptance of diverging understandings or mental models presents a problem for programming education. Neither the computer nor the compiler is flexible as to how the program runs the student's code. It is still necessary that students should reliably acquire the foundational knowledge of computers and programming, and put aside their misunderstandings and misconceptions [5]. As a consequence, computer science educators and researchers have been trying to improve students' learning by creating new ways to interact and engage with the material they produce.

Researchers in constructivism have promoted cognitive conflict as a way to promote conceptual changes [112] and to correct students' misconceptions. This approach has been successfully used in physics education [57] and programming [62]. Students are asked to explain an empirical observation, and their incorrect explanations are challenged with further empirical observations that they cannot explain or refute. Students are more open to changing their conceptions when an alternative and correct explanation is given. In their learning programming experiment, Ma *et al*. used visualizations to present the correct model of object assignment [62].

Three main taxonomies have been developed to assess students' understanding or to organize *learning outcomes*: Bloom's [13], Gagne's learning outcomes [33], and the *structured of observed learning outcome* (SOLO) taxonomies [12]. Bloom's and Gagne's taxonomies start with the basic knowledge or information acquisition, and progress towards more demanding skills. SOLO is a more recent development and delves into the ability of students to manipulate knowledge. The higher levels of the taxonomy are when students can relate different concepts and when they can produce new concepts based on the ones they have learnt. Bloom's and the SOLO taxonomies have been used in programming education to compare novice and experts understanding of code [60] and design assess-

ments [119].

The next sections will describe three components or aspects that are important for both education and programming learning: knowledge, mental models, and skills. Skills are sometimes referred as being a component of knowledge but here they are considered as an independent component.

2.1.1 Knowledge

In this section we define knowledge as the facts and information acquired through learning.

One classification useful for programming is the one that distinguishes between *procedural* and *declarative* knowledge. Declarative knowledge is the most simple one and refers to the retention of a single statement or item of knowledge, and it is usually taught by behaviourist means. Procedural knowledge is the knowledge required to do something and usually involves remembering a number of steps or acquiring a mental model. For example, in programming, declarative knowledge would be knowing what a keyword stands for, whereas procedural knowledge will be needed to program a while loop.

From a constructivist point of view, Holmboe [39] developed a "cognitive framework for knowledge in informatics" that combines both types of knowledge. He developed his framework by analyzing people's understanding of object orientation. For him, knowledge is structured in four categories: from *hunches*, or first attempts to understanding a concept, to *holistic knowledge*. In between, he describes two other categories which are at the same level, *practical knowledge* and *theoretical knowledge*. Practical knowledge is achieved when students can relate the program to reality and explain it using object oriented concepts. Theoretical knowledge is akin to declarative knowledge.

Holistic knowledge is the result of interconnecting practical and theoretical knowledge. At this stage, students have accommodated the knowledge and can operate with it; or in constructivism terms,

the learnt concept is now part of their first order language. Holmboe's proposal for programming education is simple: combine theory with practice in an authentic scenario. However, he acknowledges that it is unrealistic to expect students to have holistic knowledge after just passing a programming course.

Holmboe's classification aligns well with the SOLO taxonomy levels — pre-structural, uni-structural, multi-structural, relational and extended abstract [11]. However Holmboe's classification acknowledges better the practical aspect of programming because its holistic knowledge demand authentic experiences. In the SOLO taxonomy, the outcomes relate to a task, normally set up by the teacher.

In Paper II, Holmboe's [39] categories are used to characterize students' understanding of programming concepts, especially his hunches category. In Paper II the categories from Perkins and Martin are also used to trace the evolving knowledge of the students and how the tool modifies it [96]. Perkins and Martin decomposed the fragile knowledge that recent students make use of when programming. They worked with several students as the students solve a programming exercise. Along with the intervention the researchers gave strategic prompts to strengthen students' fragile knowledge. These prompts included more information relative to the animation than normal prompts.

Perkins and Martin described four kinds of fragile knowledge in practice — *partial*, *inert*, *misplaced* and *conglomerated*. Partial knowledge is self-explanatory and reveal incomplete knowledge. Inert knowledge is the one that students have but do not use when needed. Misplaced knowledge represents knowledge that the student applies but is not relevant in the current context, whereas conglomerated knowledge represents those cases when students' code contains "disparate elements" that are not supposed to be together.

Fragile knowledge and troublesome knowledge [95] have sparked the idea of *threshold concepts* [67]. By investigating the learning process of students in several fields, a set of concepts appears to be common to them in the sense that they are: *"transformative (oc-*

casioning a significant shift in the perception of a subject), irreversible (unlikely to be forgotten, or unlearned only through considerable effort), and integrative (exposing the previously hidden interrelatedness of something)." [67]. In programming, Sorva [117] proposes three main threshold concepts: program dynamics, information hiding, object interaction, "and possibly addressable memory".

2.1.2 Mental Models

Mental models are mental representations or beliefs of real things [43], be it a computer or some physics phenomenon. By having mental models, people comprehend the world and knowledge around them and operate with them. While similar to Holmboe's idea of holistic knowledge [39], mental models can be incomplete, unrealistic and evolving [90]. The goal of students is to create a *viable* mental model [61], that is, a mental model that can sufficiently explain the concept and succesfully apply it to solve new problems.

Importantly for programming education, mental models can also be *run* by their owners. Students and programmers execute their code in their head according to their own mental model. A student's mental model relates to a *notional machine* that the programming language or the teacher conveys. The notional machine is a set of rules that the computer virtually operates under [27]. It is the job of the teacher to make sure that an adequate notional machine is presented to the student.

The lack of a viable mental model is at the source of plenty of students' misconceptions.

2.1.3 Skills

After students have acquired some minimal knowledge of programming, and they need to put that knowledge to use, two main skills are necessary: problem solving [105] and debugging [35]. First, students use their problem solving skills to devise a solution that uses their knowledge. Later, when running their program, the debugging skill becomes important as students try to find their own errors

in the code. Naturally, the errors can be due to several causes, such as fragile knowledge, misunderstanding of the problem and others. Unfortunately, and as mentioned before, two working group reports [59, 66] point to the lack of these two skills as the reason why so many students fail to learn programming.

Indeed, teaching problem solving skills has been neglected in introductory programming courses, or at least not explicitly taught, even if students are expected to learn it [66]. However, the importance of the skill was already noted by Mayer [64] and Riley [105]. To teach the skill, Riley [105] asked students to write a problem definition and to continue the process by writing the algorithm in pseudo-language. Thompson [120] presented a way to teach the skill (*How to program it*) following Polya's *How to solve it* method, which added the reflection part once the problem is solved [99]. More recently, new approaches [10, 30] show the instructors solving the problems step by step, revealing the process and the missteps they take when programming. In their report, McCracken *et al*. suggest that successful students are mastering the programming knowledge required to pass the course, but not learning the skills of programming [66].

Debugging incorporates several important skills such as program tracing and program comprehension. In Lister *et al.*'s working group report [59], the authors claimed that students' lack of problem solving skills was not a cause but a symptom. The required viable mental model for programming, which tracing and proper program comprehension would reinforce, was missing. In summary, students could barely start programming, or answering multiple choice questions, if they could not understand the program they were working with.

The importance of tracing [96] as a pre-requisite for programming [124] is seen in computer tests all around. Teachers ask all the time what will be the output of certain program snippet if executed. The ability to correctly trace a program by a student reveals the students's grasp of a notional machine, and that their mental model of it is a viable one for the task at hand.

Program comprehension skills do depend on students's previous knowledge. Pennington [94] specified two kinds of knowledge needed for program comprehension: *text-structure* knowledge and *plan* knowledge [115], which loosely translates to declarative and procedural knowledge.

2.2 OBJECT-ORIENTED CONCEPTS AND JAVA

Object-oriented programming (OOP) made its major appearance in higher education in the last decade: industry demanded it and it appealed to educators and researchers. Java was usually the language of choice due to its purer object orientation, when compared to C++, wide support, and extensive libraries, among others.

Proponents of the object oriented paradigm aimed to improve program comprehension and design by relating programming objects to real-world objects that can be decomposed into its parts and properties. Several authors [25, 93] argue that what makes object orientation better is also what complicates learning for novices: being able to abstract from the real world the behaviour of objects.

Additionally, Sajaniemi and Kuittinen [110] summarized the research carried out in programming and object-orientation on education. They noted the differences in *notation* (or syntax), and notional machine between imperative and object oriented programming added to the extra difficulty of teaching and learning OOP [110, 130]. The notation in OOP, especially Java, includes an extensive new vocabulary that the student has not seen before (see Listings 2.1 and 2.2). The notional machine focuses on message passing rather than step-wise execution. These factors make novices focus on extraneous programming elements that worsen their program comprehension and their mental model *execution*.

Listing 2.1: Hello world program in Java

```
public class HelloWorld {
   public static void main(String[] args) {
       System.out.println("Hello, world!");
   }
```

```
}
```

Listing 2.2: Hello world program in Python

```
print("Hello, world!")
```

Ragonis and Ben-Ari [102] elicited the new kind of misconceptions that object oriented programming produce. The importance of class design in OOP courses means that program execution flow is not sufficiently explained, or is only explained later in the course. They advocate for teachers tools to reveal the link between class design and program execution flow, the static and the dynamic. Benaya and Zur [9] mentioned other problems of students' understanding the finer details of the Java notional machine, *e.g.* the process of object creation involving the invocation of superclass.

Be it the demand of tools that try to overcome these difficulties, be it the fact that educational programs support better graphics and can be easily distributed across the world, many of visualization tools have appeared [6, 22, 50, 92, 117]. Teachers have a wide range of tools and research to choose their toolkit of choice. Still, the caution teachers apply when using the tools is unwarranted [7], and results in the perpetuation of the problems found.

2.3 SUMMARY

The three components mentioned before are not isolated and their boundaries are not clear. Figure 2.1 describes my interpretation of the relationship between the components.

The chapter has given a small recapitulation of the theories and the components of learning to program. Researchers have tried to devise models, or to describe knowledge representations, of students learning programming. Their goal is certainly a worthwhile pursuit to refine or contradict previous results. In my research I have selected from different sources those theories and results that allow me to describe the purported benefits of program animation and Jeliot 3, see the next Chapter. This theories are also the base for the evaluation and re-design of Jeliot 3, summarized in Chapter 4.

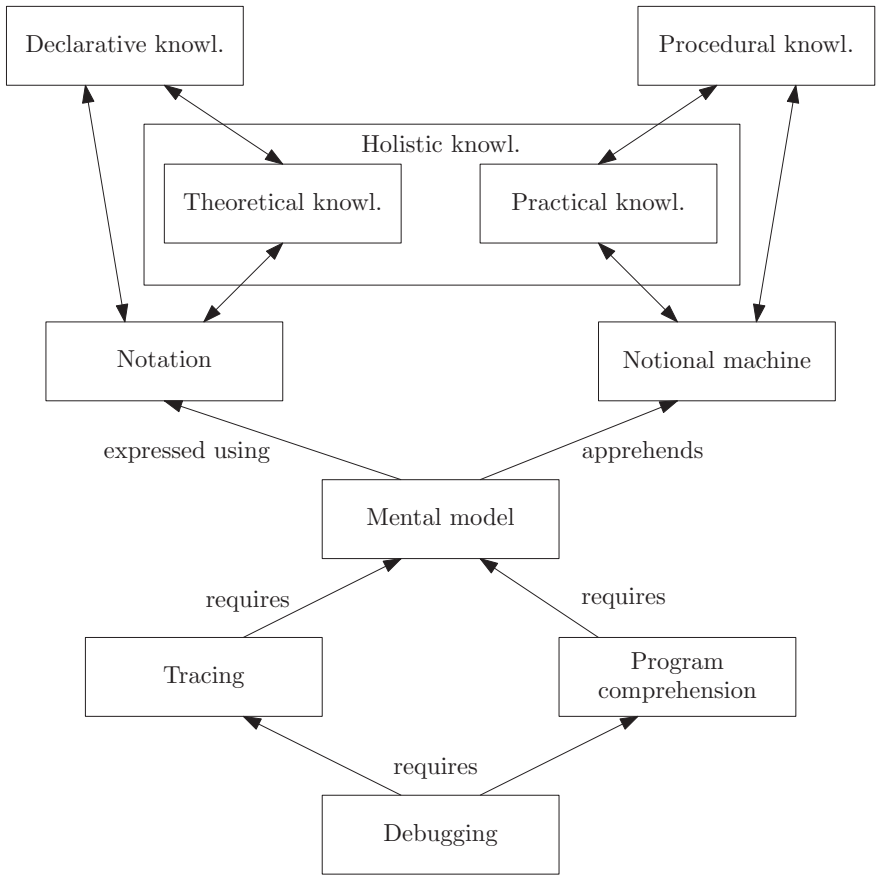

Figure 2.1: Theoretical components and their relationships. Bi-directional linkages mean that the components are closely related.

3 Program Animation and its Evaluation

Program visualization (PV) is one of the tools proposed to help students in learning programming. By visualizing a notional machine, PV tools should help students in creating a viable mental model. The graphical features and step by step execution support the development of their tracing and debugging skills.

Program visualization is one of the fields of software visualization related to education, the other field closely related is algorithm visualization. While program visualization usually focuses in conveying the details of programming constructs and program execution, algorithm visualization focuses on algorithms and data structures. The visualization term allows for both dynamic and static representations. This work focuses on program animation, which is equivalent to dynamic program visualization. Examples of each field are the Transparent Prologue Machine for static program visualization [15] and JHAVÉ for algorithm visualization [85], MatrixPro for algorithm animation [45], and, of course, Jeliot 3 for program animation [6].

As part of a workshop, I reviewed a set of visualization tools with a focus on human computer interaction that had been evaluated at that time (2007) [69]. The review pointed to the need of more user studies to develop a set of guidelines for future visualization tools. A more recent review of tools can be found in Sorva's dissertation [117]. On one hand, his review makes apparent the increasing number of tools developed, and on the other hand the lack of proper studies and the abundance of anecdotal-based recommendations.

Despite the growing numbers of tools and their moderately positive educational outcomes, the use of visualization tools in programming education is not yet widespread. Ben-Bassat Levy and

Ben-Ari found out that the main drivers behind the acceptance, or rejection, of tools by the *teachers* are the "integrating [of] the tools with other learning materials and on addressing the role of the teacher in the use of software by the students" [7]. Later they found out that the teachers' lack of control when using the tools was also hindering the usage of the tools. Knobeldorsf *et al*. studied the same problem but from the students' point of view: students do not use the available visualization tools [48]. They suggest that the issue is a complicated one: some students do not use a tool because the tool has not yet been *internalized*, and others because the tools and the concepts have already been internalized [48]. Two ways to get out of this conundrum are proposed and evaluated in Papers III, IV and VII.

3.1 IMPLEMENTATION OF PROGRAM ANIMATION TOOLS

Roman and Cox [106] extracted three roles of people involved in program visualization, the *programmer* who writes the program to be visualized, the *animator* who defines and constructs the mappings, and the *viewer* who actually uses the visualization. In this thesis I will focus on automatic program animation where the student takes the roles of programmer and viewer, while the animator is automated. In other approaches the student is also the animator by coding the animation, *e.g*. the Leonardo development environment [23]. In other approaches, it is the teacher who tailors the visualization of each program by annotating the source code fed to the animation engine, *e.g*. VIP [125], Ville [103] and PlanAni [108]. Annotations are not shown to the students and thus the animation may appear to the students as automated.

Plenty of tools make use of the debugger metaphor and are augmented with graphical representations to create *visual debuggers* of programs [68, 107]. While not animated, visual debuggers are usually automatic and developers of visual debuggers face similar challenges than for program animation tools.

Two styles guide the implementation of program animation tools

that automatically generate the animation. One is based on the generation of an intermediate code that traces the execution of the program, and the other is based on event-listener architecture, where messages are passed as execution advances [70].

The main characteristics of the event-listener approach are that it provides a clean separation between the program being executed and its animation. JAVAVIS is an example of an event-listener based tool for Java programs [92], where the program is compiled and the Java Virtual Machine produces the execution events as the program is executed through the Java Debugging Interface. The use of compiled code limits the detail of the animation as the Java Virtual Machine only passes information regarding the method calls and their results.

A refinement of the event-listener approach is the *observer architecture* for program visualization [52]. In the observer architecture, the program to be animated is decomposed into components, and they become *observable* objects by being linked to a visualization object. The visualization object observes the components and creates the visualization according to the changes in those components. This solution implements elegantly the history of changes and allows for a re-wind feature. The same drawback of the event-listener approach applies here as the reason of the changes are not visualized.

To gather more information from the execution, other programming animation tools run the code through a dynamic interpreter of the abstract syntax tree of a program. The interpreter does not compile the program to machine code and run it, but instead it parses the source code and traverses the tree it creates. For program animation purposes the interpreter is modified to produce intermediate code as it traverses the tree. Jeliot 3 is an example of an interpreter based program animation tool.

Price *et al.* [101] noted that intelligence in automatic systems was low. As the system designer takes all the visualization decisions when the tool was programmed, the tool can adapt little to the program being visualized. For Java-oriented tools, that usually means

that every data structure (*e.g*. a list) except for arrays is displayed as an object. Also, the user cannot change the parameters of the visualization, what is shown and how it is shown in the screen. In Paper III we propose an architecture to automatically adapt the contents of the visualization according to the student's current knowledge.

If the system is not automatic, then it is implied that the teacher, or student, tailors the visualization [23, 103, 125]. To tailor the visualization, special libraries allow for annotations to be included in the visualized programs. This non-automatic solution results in personalized visualizations, but often they are too time-consuming and not preferred by the teachers [98]

3.2 EVALUATION OF PROGRAM ANIMATION TOOLS

Hundhausen *et al*. conducted a meta-study of visualization tools that collected 24 studies of controlled experiments [42]. In their study, both program and algorithm visualization tools were considered. They wanted to answer two questions: which theories and factors better predict the success of visualization tools, and which measurement is most sensitive to the learning benefits of visualization tools. They noted that certainly the results were mixed, with the same number of significant and non-significant results on the positive impact of the tools in learning. From the analysis, cognitive constructivism is the most robust theory, as the experiments based on that theory yield significant results. The caveat is that for the experiments in cognitive constructivism the effort and time of students in the experiment group is more important than the content of the visualization [42]. This finding has led to researchers focusing on how the students interact with the animation, rather than what the animation shows.

Naps *et al*. devised the engagement taxonomy [86] which proposed seven categories in which students could engage with visualization tools in learning. The taxonomy included a category for when no visualization was used: *non-viewing*. The others represent the active visualization engagement categories:

1. *Viewing*. The student views the visualization. According to Lauer [56], viewer can be:
   (a) **Passive**: teachers take the conducting role and explains the graphical components in the visualization, and they will also explain the steps if the visualization consists of an animation;
   (b) **Active**: students watch the visualization by themselves.
2. *Responding*. Students are asked to answer questions as the visualization is demonstrated. In the case of animations, they may be asked to guess what happens next in the animation.
3. *Changing*. Students have to change the appearance of an existing visualization.
4. *Constructing*. Students create the visualization from scratch.
5. *Presenting*. Students present their own visualizations to their peers.

The engagement taxonomy has recently been used as yardstick for evaluations of visualization tools. Urquiza-Fuentes and Velázquez-Iturbide report in a long-term study that the increasing engagement of the student does not always lead to increased grades [121]. However, the drop-out rates of the students were lower after using the more engaging tool, also there were learning improvements when using engaging visualizations for learning complex topics.

Knobelsdorf *et al*. suggest that most of the tools and studies are expert and teacher centered [48]. Almost none of them has been developed by having the student in center. Novice students only participate in the final evaluation of the tool, and thus the tools may not reflect their needs and understanding. They propose "to investigate how students use a visualization tool regularly for their programming assignments and how they interact with the tool in the process of internalization" [48]. A similar line of research is presented in Paper II, where the roles the animation tool can take for the student are presented.

The experimental studies on automatic program animation and visual debuggers have yielded usually positive results, if sometimes due to the extra time spent learning by the subjects — as Hundhausen *et al*. detected [42]. VINCE [107] and OGRE [68] are two visual debuggers that, when students use them, they learn better than those who do not use them.

The animation tool PlanAni bases its animation on the roles of variables [108] and studies have focused both on the benefits of the roles of the variables and on the impact of the animation on the students' learning. One of the benefits of the tool is that the use of roles is good for communication between teacher and student [109]. Nevalainen and Sajaniemi [87, 88] compared different versions of the tool to find the most beneficial factor for learning. An animated version, a static version of PlanAni, and a textual version with explanations about roles were given to the students to interact with. The final finding was that the students' knowledge about the roles of variables did not depend on the animation nor on the graphical representation.

3.3 JELIOT 3

Jeliot 3 belongs to a family of visualization tools that include algorithm animation (Eliot [53] and Jeliot I [36]) and program animation tools (Jeliot 2000 [8] and Jeliot 3 [76]). In this section we summarize the history of Jeliot, but a more complete review of the history of Jeliot has been published by Ben-Ari *et al.* [6].

Jeliot 2000 was the first to focus on program animation and the predecessor to Jeliot 3, and they share many features. After Jeliot I, many of the design choices of Jeliot 2000 were meant to support novices in their learning. Common to the family is the use of a theater metaphor [53] to explain the visualization components (*actors*) and the execution steps (*script*). In the program animation branch of the family, the program (script) is only self animated (the director is the tool). In the algorithm animation branch, the user can also take the role of the *director* and choose what to animate and how.

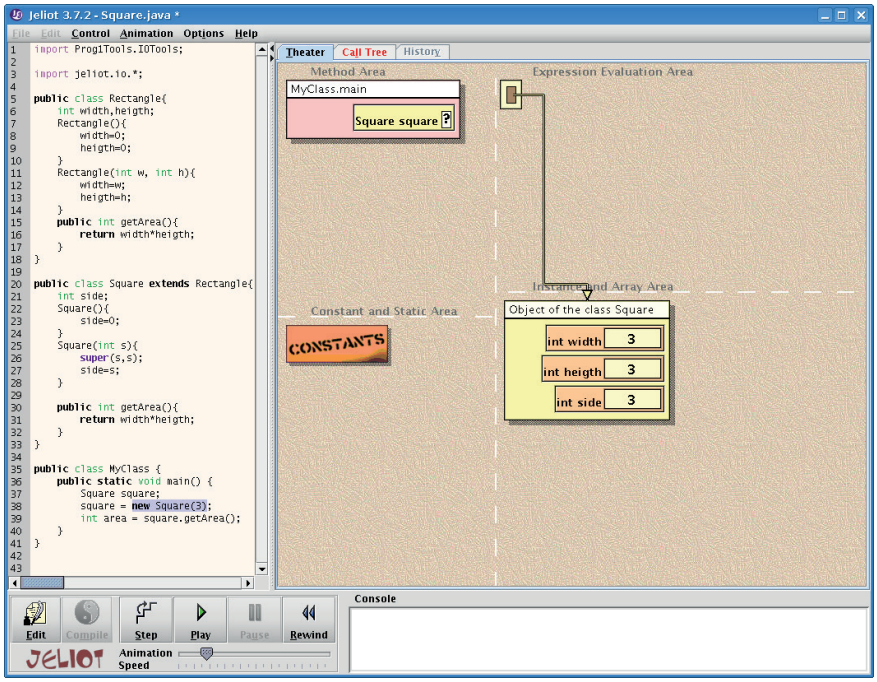

Figure 3.1: Jeliot 3 window

Jeliot 3 development was guided by technical necessities. Jeliot 2000's implementation of the animation was tightly coupled to the source code interpreter. As the need to animate more object-oriented concepts arose, the limitation was apparent, and a new modular architecture for Jeliot 3 was implemented [70, 82]. The next section describes the architecture, and Papers III, IV, and VI present the modifications proposed and implemented using the current architecture.

Jeliot 3's interface is divided into two main components, see Figure 3.3. The left-hand side panel is the editing panel and consists of the editor menu bar, containing buttons to access text-edit functions, and the editor itself. The right-hand side panel is the animation area, the theater, where visualization occurs. On the bottom, the control buttons are represented using a VCR remote controller

metaphor (DVR would be a more current term). They allow the user to compile the code, and to run the animation. Next to the control buttons, the console is displayed, which prints out the messages and system output. When the animation starts, the theater drapes open and the editor is blocked during the animation. The Java notional machine defined by Jeliot 3 is visible once the animation starts.

The theater, see Figure 3.2, is divided into several areas that represent parts of the memory and the processing unit. Program execution will update the theater with new actors (variables, methods, constants, ...) which will be displayed as boxes and moved around according to the source code.

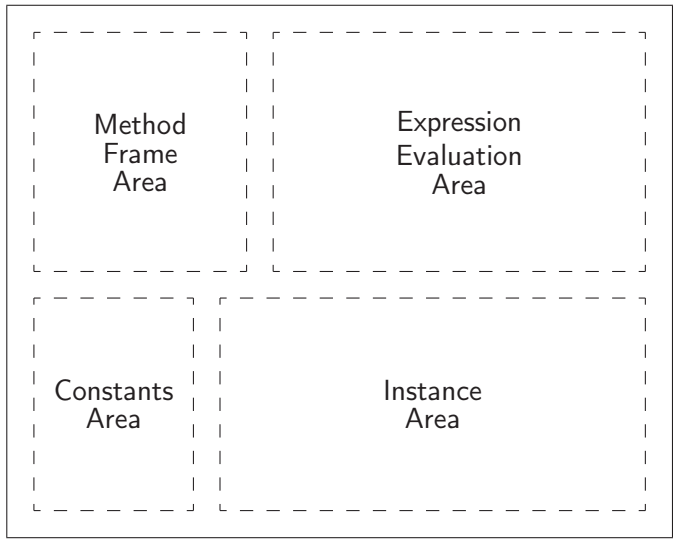

Figure 3.2: The structure of the animation frame (theatre) in Jeliot 3.

3.3.1 Implementation

The modularity requirement of Jeliot 3 was one of the main reasons of developing Jeliot 3 to improve the system structure and design. The modular design, see Figure 3.3, was described in Myller's Mas-

ter's thesis [82]:

> *"The structure of Jeliot 3 is shown in 3.3. The user interacts with the user interface and creates the source code of the program (1). The source code is sent to the interpreter and the intermediate code is extracted during the execution of the code (2 and 3). The intermediate code is interpreted and the directions are given to the visualization engine (4 and 5). The user can control the animation by playing, pausing, rewinding or playing step-by-step the animation (6). Furthermore, the user can give input data, for example, an integer or a string, to the program executed by the interpreter (6, 7 and 8). "*

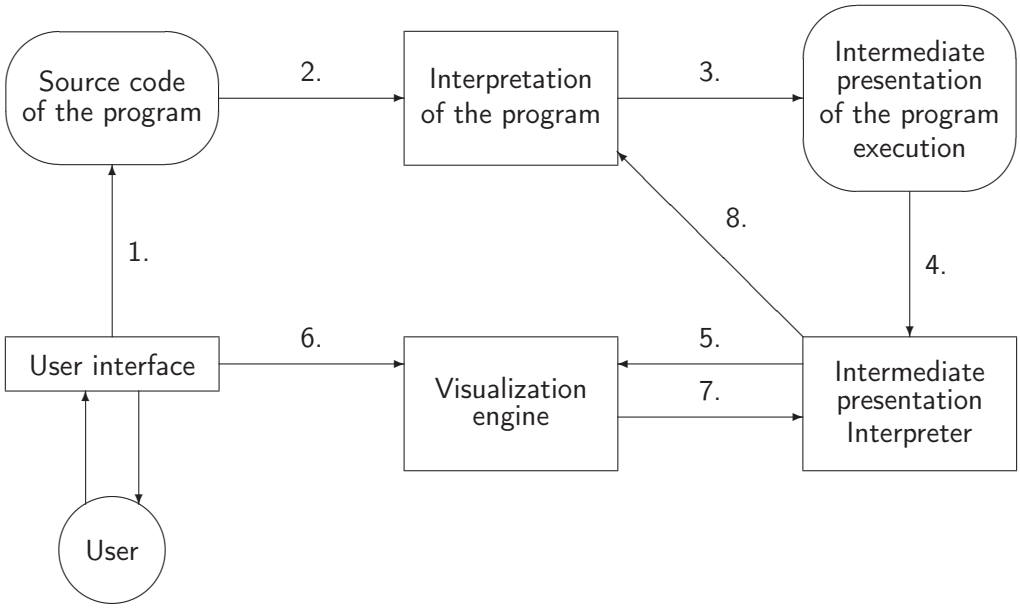

Figure 3.3: The functional structure of Jeliot 3. From Myller's Master's Thesis [82].

The intermediate presentation of the program execution, between steps 3 and 4, was defined with an intermediate code called

MCode [70]. The MCode represented the steps taken by the Java interpreter and was meant to be general enough to represent the execution of other programming languages. Proving the modularity design, independent researchers replaced the Java interpreter and created versions of Jeliot for C++ [47] and Python [31]. The modular design allowed for the creation of different interpreters that will produce different visualizations (step 4). An approach to this feature was used by Myller to automatically add questions to the visualization in Jeliot 3 [83].

3.3.2 Evaluation

As said before, it was the research done in the algorithm animation branch of the Jeliot family that started the development of the program animation. Lattu *et al.* [55] did preliminary research on Jeliot I with two groups of students learning to program, one from a university and another from a high school. On one hand, Lattu *et al*. found that novices struggled with the complexity of Jeliot I interface and that teachers could not use the tool to teach basic programming concepts like control flow [55]. On the other hand, students appreciated the animations of values moving according to the algorithm. Further research by Lattu *et al.* [54] mentions flexibility as a key property of visualization tools so that they can be used as demonstration programs.

In the program animation branch, Ben-Bassat Levy *et al*. studied the usefulness of Jeliot 2000 [8]. They concluded that the support provided by Jeliot 2000 was valuable to teach *mediocre* students or those who had difficulties with understanding programming abstract concepts. They found out that Jeliot 2000, with its predefined visualization behavior, provided a common language between students and teachers. In their experiment, the impact of the tools was limited to students with better understanding of programming.

Kannusmäki *et al*. carried out the first experiment with Jeliot 3 [44]. In that experiment, Jeliot 3 was used by online education students. In the qualitative analysis of the students' weekly feedback, weak,

as in weaker than mediocre, students self-assessed themselves as having learnt programming concepts using Jeliot 3. Not trusting students' self-assessment, Papers I and II have compared students' self-assessed knowledge with their actual knowledge as elicited from the program animation.

In the study of Kannusmäki *et al.*, students could be divided into two groups, one that used Jeliot 3 at some point while working on the exercise and the other one that preferred a standard Java toolchain [44]. Strong and mediocre students mostly used another development environment and did not like Jeliot 3. Complaints from the students were varied: too slow, too detailed, not standard Java. It was proposed to develop Jeliot 3 further to allow students to tailor the animation according to their knowledge. In Paper III, I explore the idea of automatically tailoring the visualization, in other words, adapting it to the students' knowledge.

Bednarik used eye-tracking recording to compare the strategies used by novices and experts when using Jeliot 3 for program comprehension and debugging [2]. In a study focusing on program comprehension, he found out that as novices repeteadly used Jeliot 3 to understand what a program did, they changed the focus of their attention from the animation to the source code. Experts, on the contrary, did not need repeated visualizations, and could reason the program's purpose by mostly looking at the animation [4]. In contrast to the novices, experts had read the code before the animation, and used the animation to fine-tune their answers.

Myller *et al*. investigated the effect of visualizations in collaborative learning settings [84]. In the process they extended the engagement taxonomy adding four more categories and modifying two of them: controlled viewing (students choose what to view), entering input (students enter input that modifies the execution and visualization of the program), modifying (students change the visualization by modifying the program or data structure), and reviewing (feedback on the animation is expected from the students) [84].

After finding the status of students' mental models on assignment (only 17% held a *viable* one) [61], Ma *et al*. investigated the

use of Jeliot 3 and cognitive conflict to improve students' mental models [62]. When compared to a bespoke tool for teaching assignment, Jeliot 3 did not improve students' mental models. The lack of textual explanation was considered as an important factor, explanations were included in the bespoke tool. The idea of adding explanations to Jeliot 3 had been around for a while, and in Paper VI we investigate the temporal placement of explanations: after or before the concept being explained.

Čisar *et al*. carried out the largest study on the effectiveness of Jeliot 3 in programming education with 400 students [63]. They could assert that there was a significant difference between the control group, no visualization, and the experiment group, Jeliot 3, in the test at the end of the course. A similar study, albeit smaller, also found Jeliot being beneficial to students of programming [41]. Hongwarittorrn and Krairit also checked students' attitude towards OOP after the course. Jeliot did not have an effect on the students' attitudes [41].

3.4 SUMMARY

This chapter has summarized the relevant research and development literature in program visualization software, in particular automatic program animation tools. It has also briefly introduced Jeliot 3, the tool on which my design research process has focused.

Papers III, VI, and specially Paper IV, focus on stretching the modular design of Jeliot 3 and finding out if its architecture can adapt to implement features suggested by the above literature and by the findings presented in Papers I, II and V.

4 Summary of the Publications

In this chapter, the main findings are presented and discussed. The chapter comprises two main parts. First, results from the observation component are presented in Section 4.1, which summarizes the findings related to the students' engagement with Jeliot 3 and their understanding of OOP concepts and the animations of Jeliot 3. The following sections, 4.2–4.4, present the siblings of Jeliot 3 that have been developed and evaluated during my research. Those sections correspond to the systems development and experimentation components of the research.

The limitations of the findings and results presented in the first four sections of this chapter are considered in Section 4.5. Finally, the research questions are revisited and answered, Section 4.6.

4.1 ENGAGEMENT AND UNDERSTANDING

This qualitative research explores the impact that Jeliot 3 and its animations have on students' understanding of programming, and in more detail when they are learning object-oriented concepts. This section summarizes Papers I and II, which report two different studies.

From my point of view, using examination or test results as a measurement of understanding was not enough, and interviews and verbal protocols were used to bring forth students' understanding. The same verbal protocols were used to recognize the roles Jeliot 3 took when being used by the students.

4.1.1 Students' Knowledge and Understanding

In the first study, a group of six students volunteered to intensive progamming tutoring using Jeliot 3 at University of Warwick (UK), the *tutored* group, and their understanding of Jeliot 3 was compared to those students that used Jeliot 3 without guidance, the *normal* group. The original purpose of this set up was to have a first hand account of how the tutored students used Jeliot 3. However, the interesting results were only when their understanding was compared with the other group.

According to one of the research hypothesis, the two students from the *tutored* group should have shown a greater vocabulary and understanding than the four from the *normal* group. After the basic object oriented concept was explained and practiced in the course all of them where asked to verbally explain what goes on in the Jeliot screen when an object is created. The description of the two groups showed a similar level of detail when describing the animation, and they also showed several misconceptions about the object creation animation. Only two of the normal group students were correct in their explanation of Jeliot 3 animation. All of the students were positive of Jeliot 3 and claimed it had helped them to understand arrays and objects. Sorva would say that arrays and objects have been *transliminal* concepts [116], or concepts used to overcome a threshold concept [67], in this case addressable memory. Jeliot 3, then, is a valuable tool for students when grounding their knowledge.

In the second study, the issue of *understanding* during one whole programming course was explored at Iringa University College, Tumaini University (Tanzania). Six Tanzanian students were selected to participate in three individual sessions at different points of the 12-week long course. Each individual session covered one topic. Array creation, method calling, and object creation were the topics selected as their execution is complex and Jeliot 3 animation details all the steps. Students had to watch the animation at least twice per topic. Each time, students were asked to describe what was

happening in the screen, similarly to what was asked to the English students at the end of the experiment. The goal of the intervention was to observe the evolution of their knowledge of programming concepts and of the Jeliot 3 animations.

Students did suffer when explaining the animations, and their descriptions were not very verbose. As well, the animations were confusing the students, and when students tried to narrate the animation they showed a mix of conglomerated and misplaced knowledge [96]. For example one student confused the array concept with its depiction in Jeliot 3 that also shows the length of the array. Thus, this student confused the array with one of its properties. In Paper I, it is reasoned that this confusion could be explained by Holmboe's interpretation of Vygotsky's ideas on language and learning [40].

> *"One possible reason is that for simpler concepts, a student's previous vocabulary is enough to describe what happens on the screen and in the program. For more complex concepts, students are still assimilating the concept, and the terms required to describe the concept do not form part of the students' first order language (composed of the words that are self-explanatory). "*

Together, these studies highlight the positive and negative aspects of Jeliot 3. Some of the outcomes are due to the philosophy behind the tool: the notional machine is useful to understand the concepts. However, even with the tool, the fine details of the notional machine are lost on the students, even with considerable tutor support.

Other outcomes are related to the practical implementation of the tool: its features and their usability. The animations of Jeliot 3 and its easy user interface are good to attract novice students without programming experience. The animation is a drawback when the same concept is visualized several times. Without added explanations students will either ignore the animation, or will be confused with it. As it is said in Paper I:

> *"The number of repetitions of the same animation may desensitize students to the importance of the animation itself, and reduce it to a "movie of moving boxes". They were able to follow the boxes, and discover when they have been misplaced, but whilst this is useful to identify bugs, the underlying meaning of the animation may not have been assimilated."*

For these reasons solutions were sought, modifying both the philosophy behind Jeliot 3 and some of its features: Jeliot Adapt, Paper III, Jeliot ConAn, Papers IV and V, and Jeliot with Explanations, Paper VI.

4.1.2 Roles of Program Animation Tools

In Paper I, it was postulated that *learning aid* and *debugger* were the roles Jeliot 3 could take and classified students' answers in the interview according to these two roles. Students answers fit in those categories.

As stated before, for several students and for several threshold concepts, Jeliot 3 helped them understand the concepts. Jeliot 3 animations were used as *scaffolding* to progress through the *zone of proximal development* [126], *i.e.* as a learning aid. For example one student said:

> *"It [Jeliot 3] is really useful because you can follow it, go through the code [...]. It is helping me to understand how I create objects. [...] It clarifies things, rather than just being a white screen."*

In this study I also sought to find out whether the students used the tool differently if they had had some extra tutoring. Unfortunately, the experiment set up did not led to any insights.

In the second study, Paper II, I analyzed the roles the tool had during students learning. From the recordings, the notes, and the tests from the sessions I induced four different roles Jeliot 3 took while being used to understand the animation and program a small task. The roles were described in Paper II as:

Empty role We say that the tool has an empty role when the student does not actively visualize the animation, or when the visualization changes neither the student's knowledge nor their attitude in an appreciable way.

Exploring role Animation tools have an exploratory role when they prompt the student to explore or to discover the meaning of the animation and the animated concept through an active visualization.

Confusing role The confusing role of an animation tool is often an unintended one. Animations are made to clarify or to serve as a learning tool. It occurs when the student cannot answer the questions that the active visualization provokes.

Teaching role An animation tool will have a teaching role when it has successfully been used for learning by the student. This role is expected for any animation tool, as learning through animations is the final goal.

Evaluating role The animation tools have an evaluating role when students use them to evaluate or assert their own knowledge.

In the analysis, it is claimed that the tool could take several roles at the same time, and that with time the predominant role is evaluating. Figure 4.1 depicts the transitions between roles found in the data. The loops in the transitions represent when a student gets stuck in one phase of the learning. The goal of the tool is to make the student progress, so that he or she uses the tool in the evaluating role. From the students' descriptions, it was inferred that the tool was evaluating their knowledge when students tried to guess what will happen next. It is my interpretation that if students were describing the animation step after it had happened, then the tool was in a teaching role, *i.e.* students were trying to understand the notional machine and forming their mental model.

I consider the confusing role as crucial, as it can indicate that the student is in the zone of proximal development [126]. With

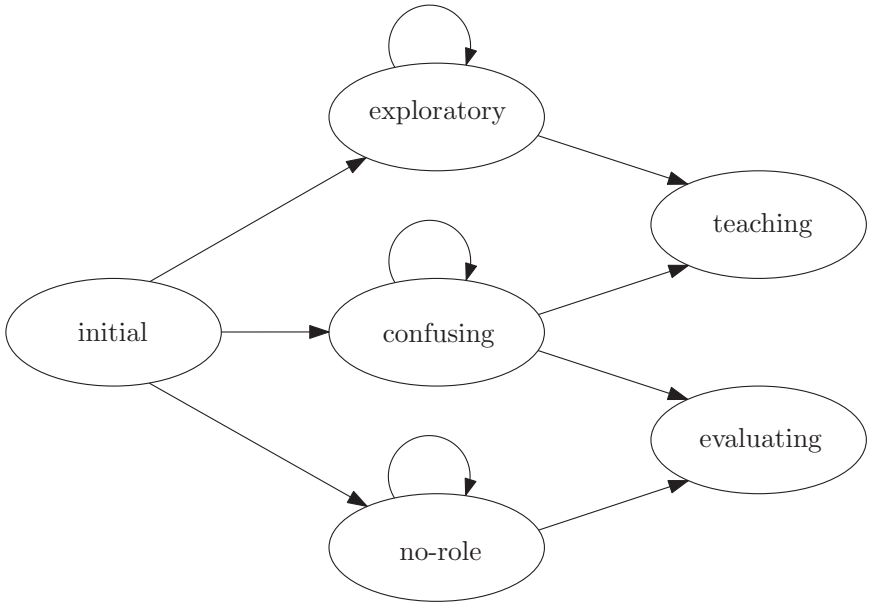

Figure 4.1: Transitions of the roles the tool takes as students use it to learn a new concept, modified from Paper II

appropriate support at that stage the tool will reveal its teaching and evaluating role.

Finally, the activity used to elicit students' descriptions, view an animation and comprehend a program may have increased the presence of the exploring and evaluating roles. If the student had been using the animation tool for debugging purposes, those roles would have been secondary.

While the roles described here could be the basis of adaptation of animation tools, I did not pursue this approach in Paper III, Jeliot Adapt. The work and analysis resulting in the roles happened after the planning and development of Jeliot Adapt, which followed a more classical adaptation, *e.g*. Brusilovsky *et al*. [18].

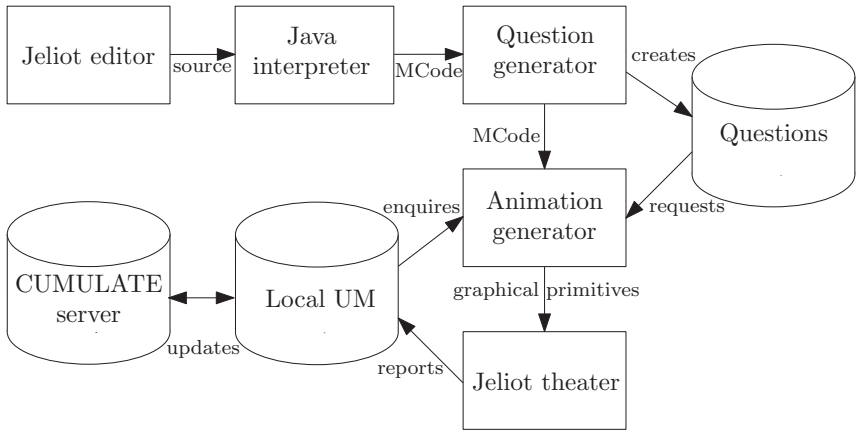

Figure 4.2: Structure of Jeliot Adapt from Paper III

4.2 JELIOT ADAPT

The Jeliot Adapt prototype aimed to tailor the animation steps to the student's own goals, knowledge and skills. The work was presented in Paper III. The animation steps would be sped up or hidden according to the student's current status of knowledge. The status of knowledge would be stored in a so called user model (UM). The user model would be remote and shared between different learning tools that will feed it with data regarding student's progress [131].

4.2.1 Implementation

Jeliot 3 had been previously improved by the addition of stop-and-think questions [83], which could be generated automatically. The prototype included that new module and added the components to communicate with the remote online user model, see Figure 4.2. In Paper III, the proposed adaptation was limited to the display of the automatically generated question. If the concept of the question had not been mastered by the student, as estimated by the user model, the question would be displayed. Depending on the correctness of the answer the user model would be updated locally

and propagated to the central user model, called CUMULATE [131]. Further work on the adaptation prototype was carried out to implement more advanced adaptive features, but not reported in Paper III.

In a following prototype, the user model would also be updated when the different concepts were animated by Jeliot 3, indicating that the student has watched that animation. Together with the questions and the visualizations, a more accurate model could be built, to decide whether to omit the steps of the animation.

This line of development was stopped due to several reasons: the complexity of the implementation grew exponentially as more concepts were added to the adaptation, and the previous studies had failed to determine that a certain number of repeated visualizations implied that the student created a viable mental model of the execution.

4.2.2 Evaluation

Due to the early end of the development of the tool, it was not evaluated. However, in Paper III we laid the ground for future empirical evaluations following Brusilovsky et at.'s layered approach [18], and suggesting a mixed methods approach.

Interestingly, we proposed the use of eye-tracking technology to evaluate the adaptive Jeliot 3, in a similar way to that which Bednarik proposed [1]. If evaluation was successful, eye-movement data could also serve as a source for the user model. That way the user model could make the distinction between what Jeliot 3 displays and what the student looks at, and have a more accurate picture of the student's understanding.

4.2.3 Discussion

Brusilovsky *et al.* [17] criticized the detailed approach of Jeliot 3 to animations as "shallow" when compared to single concept visualization tools, such as the adaptive explanatory visualization tool

WADEIn II [19]. Their reason was that single concept tools can expand the visualization with explanations and detailed assessments related to that concept. I concur with them in that adapting for a single concept it is easier that making it general, it would have been hard to adapt all the animations in Jeliot 3. However, further work is needed to make Jeliot 3 beneficial for all kinds of students, even those who may be taken aback due to its "simplicity". As well, Myller's automatic questions for Jeliot 3 [83], and the automatic explanations presented here require fresh approaches to be relevant to all students.

4.3 JELIOT 3 AND EXPLANATIONS

From my research, and others people's research, the need and benefits of extending Jeliot 3 with explanations was clear. Brusilovksy *et al*. already presented a program visualization system that included them [16, 129] Thus, in our case we focused on investigating how to properly design the feature for better learning impact. I supervised Wang's implementation work on the feature and his Master's thesis [127], which included an empirical evaluation. Paper VI contains a report on the evaluation.

Our prototype, see Figure 4.3, included a concise explanation and a more detailed explanation. The concise explanation was always displayed and it explained the animation at each step. The extended explanation included a theoretical description of the concept and it was displayed on the students' demand. The explanation window was a separate one and it was open on the top-right corner. Explanations used in the prototype were taken from Raposa's Java textbook [104].

4.3.1 Implementation

The implementation of the explanations used the modular design of Jeliot 3. The theater interpreter processed the intermediate code to produce the animation and display the explanation. Explanations

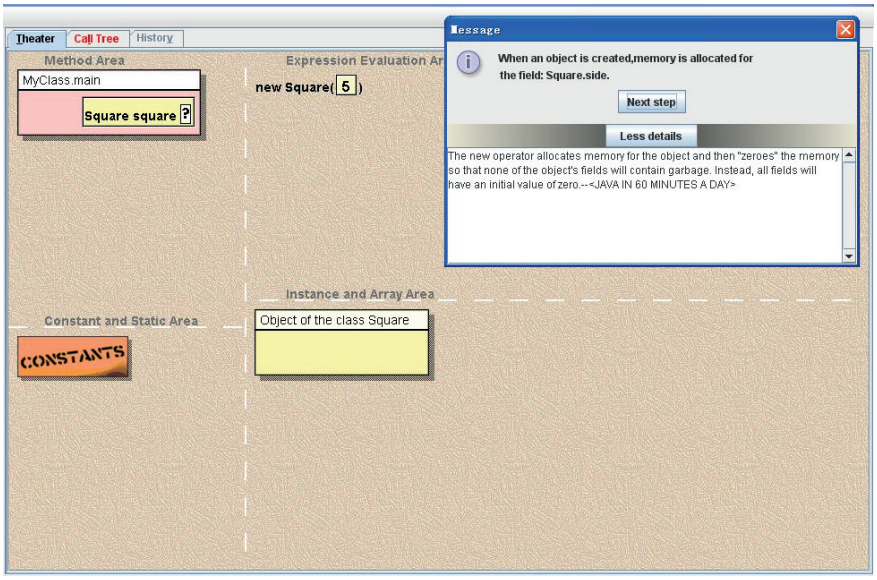

Figure 4.3: Screenshot of Jeliot augmented with explanations from Paper VI

were linked with the animation and they were only shown at the corresponding animation step [127]. Several libraries and designs for the explanation windows were tested. The final result can be seen in Figure 4.3.

4.3.2 Evaluation

The evaluation focused on the effect of the temporal placement of the explanation regarding the animated concept. In the between subject study, thus, there were two levels in the primary factor. In one level the explanation was first and in the other the animation was first. We did not consider the possibility of explanation and animation at the same time.

The study was carried out at the University of Eastern Finland in Joensuu, and 18 students volunteered to participate, 15 male and 3 female. The experiment consisted of students attending one individual session. The students watched the animations corresponding to three programming concepts: 1) object initialization and

Table 4.1: Mean learning gains, standard deviations, t value, and 2-tailed p value

	Q 1	Q 2	Q 3	Mean gain
Animation-first (N=10)	0.18 (0.23)	0.06 (0.13)	0.19 (0.20)	0.15
Explanation-first (N=8)	0.00 (0.00)	0.07 (0.12)	0.00 (0.00)	0.02
t value	2.250	-0.139	2.732	2.413
p value (2-tailed)	0.039	0.891	0.015	0.028

"this" keyword, 2) reference return and assignment, and 3) garbage collection. They watched the animation twice: the first time without explanations, and the second time with the explanations shown according to the condition. A pre-test and post-test were handed at the beginning and at the end of the session respectively to measure students understanding and record their vocabulary. The tests were identical and included one question for each concept (Q1, Q2, and Q3). The scores of the pre-test ($score_1$) and post-test ($score_2$) were graded from 0 to 5, and we measured the learning gain in the study as $Learning\ gain = \frac{score_2 - score_1}{max - score_1}$, where max is the maximum score possible.

Table 4.1 contains the main result of the experiment. The statistical tests show that there are significant "differences in the learning contingent with the temporal arrangement of animations and explanations" [128]. The animation-first students significantly improved their knowledge of the concepts on two concepts: object initialization and "this" keyword, and garbage collection.

The analysis of the written answers indicated that students from the animation-first group did improve their vocabulary by acquiring the vocabulary from the explanations. Explanation-first students rarely improved their answers or they vocabulary.

4.3.3 Discussion

Classical studies of multimedia learning as those of Moreno and Mayer [81] have demonstrated the impact of what they call the contiguity principle. That is, information given using, for example,

verbal and visual means should be as spatially and temporally contiguous as possible. Moreno and Mayer did not find significant differences in the temporal arrangement of the textual explanations. Moreover, Clark and Mayer [21] suggested adding the explanation at the beginning of the animation. Other program visualization tools have included explanations but have had them in a corner and simultaneous to the animation [103, 117]. Here, the results contradict those of Moreno and Mayer [81] and also discourage the use of simultaneous explanations in favor of explanations shown after the animation. This study does not investigate the viability of the students' mental model, but the acquisition of vocabulary is a step in the right direction towards being able to understand complex concepts which students cannot manipulate [40].

4.4 JELIOT CONAN

MatrixPro, an algorithm simulation tool [45], was the first to include the possibilty of errors in a visualization. In an interesting exercise, the tool asks students to construct an incorrect data structure. However, that kind of exercise was not further researched, and the possibilities have not been explored until now. The idea of conflictive animations for programming starts with the assumption of the importance of errors in programming, and in education in general. In papers IV and V the idea of conflictive animations is presented, implemented and evaluated.

According to Paper V, errors have been used in education for improving three key aspects of learning: *conceptual knowledge* [34, 114], *student skills* [34] and *student attitudes* [14, 100]. In programming, Ma *et al*. have used students' errors as a way to sparkling cognitive conflicts [62]. Bennedsen and Caspersen proposed that students should be exposed to the errors that occur during the programming process [10]. Ma *et al*. used animations to correct students' understandings and Bennedsen and Caspersen used pre-recorded videos of a programmer working in a development task.

In Moreno *et al.* [78], we defined conflictive animations as:

> *"those [animations] that deliberately do not reproduce what the animated code or algorithm is programmed to perform."*

In that sense, animations are wrong and students cannot trust them. This contrasts with the use of errors by Ma *et al.* [62]. Here it is the animation creating the conflict, albeit the student does not know when the conflict happens.

The original engagement taxonomy did not account for conflictive animations [86]. Thus, to list the possible kinds of activities that conflictive animations prompt, a new conflictive dimension can be added to the taxonomy with the same categories but with different interpretations.

1. *Viewing*. With conflictive animations, viewing the visualization can be done in two ways.
   (a) **Passive**. The teacher takes the conducting role and explains the steps in the animation, highlighting the conflictive steps and explaining the reasons.
   (b) **Active**. Students watch the animation by themselves, looking for an error that has already been explained.
2. *Responding*. Students are asked to spot the error or errors in the animation. Identifying the error does not necessarily mean that the student has wholly grasped the concept, but at least that they have a functional mental model of the program execution or algorithm.
3. *Changing*. From a given conflictive animation students should correct the error in the animation, usually by modifying the animation. This error may have been spotted by them or marked by the teacher in advance. Correcting the conflict requires a good understanding of the concept behind it and of the graphical representation.
4. *Constructing*. In this case, students purposely create several conflictive animations of a given concept. These animations can reflect how their understanding of the concept has evolved.

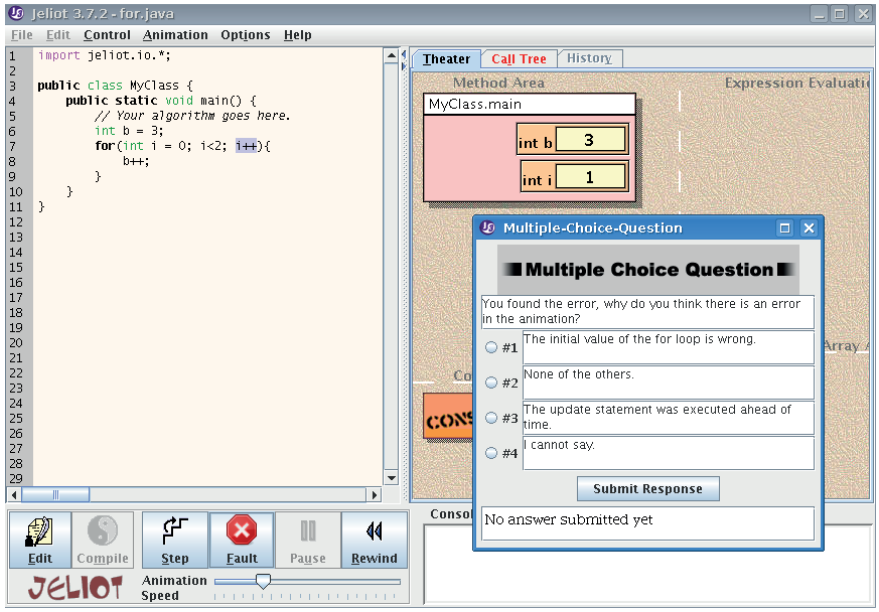

Figure 4.4: Screenshot of Jeliot ConAn from Paper IV

5. *Presenting*. At this level, students are asked to present their own conflictive animations and try to deceive peers into thinking it is a correct animation. This activity introduces added motivation in the form of competition among peers. As a side effect, students may delve into the more obscure features of the algorithm or programming concept.

Jeliot ConAn (*Con*flictive *An*imation) is a tool to automatically produce conflictive animations from teachers' or students' source code. The otherwise not very active Viewing category becomes more engaging when conflicts are added. Students using Jeliot ConAn advance the animation step by step and when following the animation they have to identify the step that produces the conflict and signal it pressing the "Error" button. Figure 4.4 shows Jeliot ConAn after the user has correctly identified the error. The student has to answer a question to avoid random guesses.

4.4.1 Implementation

In Paper IV, I proposed a layered implementation of automatic conflictive animations based on the modular design of Jeliot 3. Before the animation is shown to the students several system components, or layers, of an automatic animation tool are capable of creating the following conflicts.

Source code layer The tool can transparently modify the source code to be animated.

Parser layer The parser can change the meaning or order of execution of operators.

Tree interpreter layer The interpretation of the abstract syntax tree by the language interpreter can be modified to change the choice of methods to be executed. This layer holds most of the information in an easy way to be manipulated, and, thus, it is a good source for the automatic generation of conflicts.

Intermediate code layer A new intermediate code interpreter can be added to the tool to produce a completely different animation. However, this layer lacks most of the execution context and it is not a good source of conflicts: the execution context would have to be reproduced in this layer to create conflicts that are consistent with the ongoing animation.

Visualization layer The visualization layer has even less information about the program running, but an easy way to create conflicts is by changing the icons of operations, or names of methods being displayed. In any case, the execution and other values would be consistent and correct.

The Jeliot ConAn prototype creates conflicts at the tree interpreter layer, and the intermediate code layer is modified to keep track of the generated conflicts. For demonstration purposes another prototype has been created that creates the conflicts in the parser layer.

To develop Jeliot ConAn, the Jeliot 3 system structure, see Figure 3.3, was modified and new modules were incorporated, see

Figure 4.5. From Paper V, the generation of conflictive animations is as follows.

> *"The generation of conflicts starts when the user enters the source code for a program (1) that is sent for interpretation by the conflicting version of DynamicJava. The interpreter produces the intermediate code for the execution. At some point, the interpreter will misinterpret a statement in the program according to preprogrammed behavior—for example, an overridden method may be called instead of the overriding method. This misinterpretation will produce an alternative execution that is reflected in the MCode. The resulting intermediate code is surrounded by specific MCode instructions marking the beginning and the end of the conflictive part (3). At this time, the conflict object will have been created (4) containing all the relevant information of that conflict (location, method called, class information, etc.). This MCode is sent to the intermediate code interpreter (5), which will interpret the intermediate code line by line as the animation progresses step by step."*

4.4.2 Evaluation

Jeliot ConAn was tested in two different ways. First the implementation was tested for coverage, and, secondly, the educational impact was studied with a between subject study.

The implementation of Jeliot ConAn made it easy to create new conflictive animations. In the prototype, three conflictive concepts were developed: conflictive method overriding, conflictive implicit super call, and conflictive `for` update statements. The automatic ability to produce conflicts was tested by systematically collecting example programs from 5 Java textbooks and 1 website, and running them in Jeliot ConAn. In total 27 programs were found, 12 demonstrating inheritance concepts and 15 `for` loops, see Table 4.2. Of the 27 programs, 3 were not able to automatically produce conflictive animations due to the implementation. 16 could be animated straight away to produce conflictive animations, and an extra

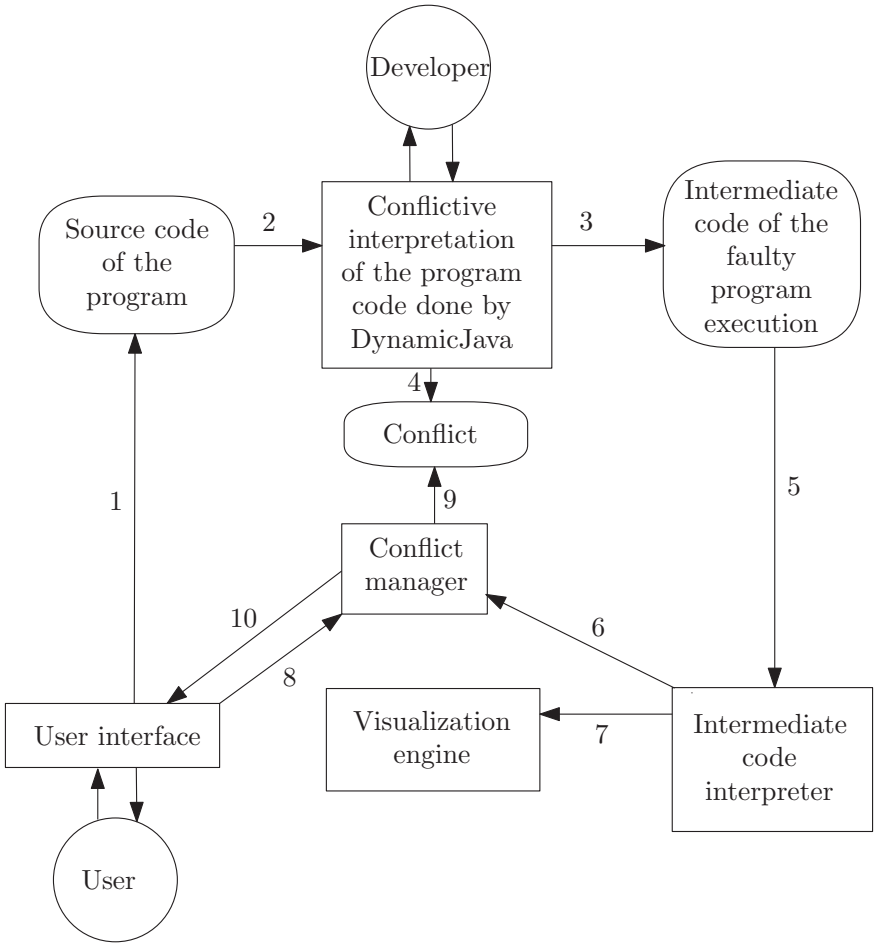

Figure 4.5: System structure of Jeliot ConAn from Paper V

Table 4.2: Results from testing of textbook program examples against Jeliot ConAn

Result	Inheritance	For Loop
Automatic generation	3	13
Automatic generation with minor changes	2	0
Automatic generation with major changes	1	0
No generation due to example program	5	0
No generation due to conflictive animation implementation	1	2
Total	12	15

5 required modifications. The remaining 5 concepts did not contain code that executed any of the conflictive concepts developed.

Those results indicated the little effort required by teachers to incorporate conflictive animations to their toolbox using readily available examples.

Given students' difficulties in understanding Jeliot 3's animations of object oriented concepts, I evaluated the impact of one learning session on inheritance concepts with students using Jeliot 3. Eighteen students from the University of Eastern Finland (Joensuu) took part in a between subject study (11 male, 7 female). The control group used Jeliot 3 to find the bugs in two computer programs. The experiment group used Jeliot ConAn to find the conflicts in conflictive animations. The programs from both groups were similar.

Students completed two identical knowledge tests at the beginning and at the end of session with identical multiple-choice questions. As well, they wrote down the description of an object creation. Tables 4.3 and 4.4 show the main results. Normality test for the group distribution failed and non-parametric statistical tests were chosen, Wilcoxon Signed-rank and Mann-Whitney tests.

According to these results the impact of conflictive animations regarding students' conceptual knowledge is not statistically sig-

Table 4.3: Average and standard deviation of previous programming experience, pre-test, post-test, and the difference between those two (gain)

Group	Prog. experience	Pre-test	Post-test	Gain
Jeliot 3 (N=9)	2.73 (σ 1.56)	3.00 (σ 2.52)	2.89 (σ 2.31)	-0.11 (σ 1.34)
Jeliot ConAn (N=9)	3.11 (σ 1.36)	1.38 (σ 2.91)	2.16 (σ 2.54)	0.78 (σ 1.28)

Table 4.4: Results from the Jeliot ConAn group are analyzed regarding their improvement in the concepts demonstrated with conflictive animations

Concept	Number of questions	Maximum gained points possible	Accumulated gained points	Number of students finding the conflict	p-value
Method Overloading	2	18	0	2	NA
Constructor	3	22	4	3	0.25
General object-oriented	5	20	1	NA	1

nificant, but it exists. Students from the experimental group had improved their knowledge, if only by half a point, after interacting with the conflictive animations for 40 minutes.

Regarding the graphical questionnaire, which consisted of 11 questions, the difference was not statistically significant (Mann-Whitney Rank Sum test, p-value=0.62). However, the average score result for this questionnaire was higher in the control group, 6.82 points, than in the experimental group, 4.82. As a side note, previous programming experience showed a correlation with the graphical questionnaire results in both groups (Spearman's test: Jeliot 3 group p-value= 0.03; Jeliot ConAn group p-value=0.12). Expert students described better the animation than non-expert students. There was no correlation between the gain and the experience in any of the groups.

4.4.3 Discussion

The engagement taxonomy, which has been augmented as part of this dissertation, partly reflects the roles that were identified in the study presented in Section 4.1. The engagement taxonomy presents what students are expected or asked to do with the tool, while the

roles of the tools expands on what the tool actually sparks in students' learning process. Jeliot 3, as has been used in the studies here, can be considered as belonging to the active viewing level in the engagement taxonomy. However, as we have seen, the tool takes different roles as students try to understand a new topic when viewing the animation, and modifying the code that produces the animation. The fact that one of the roles the tool can take is *confusing* has resulted in exploring that idea further by developing a *confusing responding* tool, Jeliot ConAn, as per the augmented engagement taxonomy. This design aimed to lead students to use Jeliot 3 in the *teaching* and *evaluating* roles, as consequences of the now forced *confusing* role.

The implementation and the results presented here are only a first step towards a better understanding of the roles of errors for learning in animations and other visualizations. The implementation only covers automatic program visualization, which is quite general and flexible for several learning scenarios. However, not all the activities presented in the extended engagement taxonomy are covered by Jeliot 3. Creating more conflictive exercises in MatrixPro, or supporting conflictive simulations with UUhistle, would enable new activities, one of them presented in Paper VII.

Of the three key aspects of learning that errors can impact (conceptual knowledge, student skills, and student attitudes), the study presented here shows some improvements in students's conceptual knowledge and skills after using conflictive animations in Jeliot 3 when compared to using Jeliot 3 as a debugging tool. Some evidence points that conflictive animations made students more aware of their gaps in understanding, and maybe in a painful way: "conflictive animation students" were less confident in their knowledge, and less comfortable using the tool. The experienced discomfort by the students can be a reflection of the confusing role of the tool, and thus positive towards learning as it is theorized in Figure 4.1.

4.5 LIMITATION OF THE RESULTS

The results presented here are the product of the research carried out around Jeliot 3. While being one popular program animation tool, it is not the only one. Observing students' reaction and description of animations produced by other tools, and even other programming paradigms, would have led to more generalizable results. However, the theater metaphor in Jeliot 3 is present in newer program visualization tools like UUhislte [117], which already includes explanations and the option for the student to simulate the program execution.

Regarding the experiments, the low number of subjects may threaten the statistical reliability of the quantitative experiments. To improve ecological validity and in order to evaluate the effect of Jeliot 3 in students' learning, it is noted that teachers and students should have used it before [8]. Due to the difficulties to arrange programming courses which used Jeliot 3 as a tool, I resorted to evaluate those courses that I was either teaching or assisting at, except for Paper VI. In those courses, I used Jeliot to explain the main concepts to small groups. This improved ecological validity but not so powerful statistical test could be used.

4.6 RESEARCH QUESTIONS REVISITED

In light of the summary of the publications above, it is time to revisit the research questions that have guided this research. The answers are based on the already presented results and take into consideration their limitations.

QUESTION 1. How do novice students engage in using Jeliot 3 when learning new programming concepts?

QUESTION 2. How do novice students understand the visualizations provided by Jeliot 3 when learning new programming concepts?

These two questions aimed to understand better visualizations, especially the animations produced by Jeliot 3, while they are used by the students. In this research, first we postulated that Jeliot 3 main uses were as a learning tool and as debugger. Later, the roles were investigated and four were identified, exploring, confusing, teaching, and evaluating — plus an extra empty one for when the tool have no role.

The defined roles explain what what the students went through while visualizing an animation they did not fully understand. A linear path from exploring to evaluating roles would be desired, in other words, from initially trying to understand the animation to using the animation to check on their own knowledge. However, students followed several paths, as shown in Figure 4.1, that not always ended in evaluating, or even teaching. In my understanding, from students' descriptions, Jeliot 3 failed to untangle students' fragile knowledge, specially the misplaced an conglomerated kinds of fragile knowledge (see Section 2.1.1). That is, for some of the weaker students the confusing role of the tool was neither remedied by special tutoring, nor repeated visualizations. Students with a more solid understanding of Java followed a more direct path towards using the tool for evaluating their knowledge.

QUESTION 3. How can new features be implemented in Jeliot 3 using its modular architecture in a way that facilitates its usage in diverse learning scenarios?

With the modular design of Jeliot 3, a new system has been developed from Jeliot 3, Jeliot ConAn, and the current one has added two versions, Jeliot with Explanations and Jeliot Adapt. The modular architecture has resulted in a system that can be modified to add new features without removing current ones. Explanations, a needed feature, were added by Wang [127]. However, the flexibility of Jeliot 3 to automatically animate programs complicates the efforts to adapt the animation in a meaningful and complete way.

The layered implementation of conflictive animations in Jeliot ConAn is an example of how the architecture of Jeliot 3 allows for

new features that require the interaction of several modules.

Jeliot with Explanations is meant to be used when students are on their own, maybe after the teachers have explained the animations or the concepts, in other words, in an active viewing role as per the engagement taxonomy [56, 86]. Jeliot ConAn can be used in more diverse learning scenarios, viewing and responding, though not all of the ones listed in the conflict-extended engagement taxonomy.

QUESTION 4. What effect have the newly implemented features on students' engagement and understanding of new programming concepts?

The effect of explanations and conflictive animations were overall positive for students' understanding. The explanations, together with the animation, improved students understanding after a short session. However, there was a difference in the effectiveness due to the temporal arrangement of the explanations. Explanations before animation resulted in better written descriptions of the programming concepts by the students.

Conflictive animations, as implemented in Jeliot ConAn, did not have a significant effect in understanding. As well, students using Jeliot ConAn reported discomfort, which may have lead to lower engagement. In any case, Jeliot ConAn users were resorting to learning material when trying to find the errors, which did not happen when students tried to debug a program with Jeliot 3. This result indicates that students improved their meta-cognition and evaluated their knowledge while using Jeliot ConAn.

5 Conclusions

"I was recently exposed to a demonstration
of what was pretended to be an educational
software for an introductory programming course.
With its "visualizations" on the screen it was
such an obvious case of curriculum infantilization..."
– Edsger W. Dijkstra

In this thesis, Jeliot 3 has been evaluated in real contexts to observe how students use it to learn new concepts using the animations. Following a systems development research approach, these observations have resulted in the development of several systems. Experimental evaluation of these systems have not been conclusive and more evaluations are needed to measure the impact of the new developments. In particular, this thesis has opened the interesting path of conflictive animations, and highlighted the possibilities of errors in programming education.

In this chapter, the previous results and discussions are transformed into recommendations and opportunities for future research.

5.1 NEW WAYS FOR PROGRAMMING EDUCATION

As Postman says, anytime we are teaching, or developing educational software, we need to asks ourselves what is the end of education. Programming education at university level have focused very much on formal learning. In this case, the end has been creating a solid foundation for the rest of the computer science studies. Formal learning usually involves a teacher and a set of declarative knowledge that teachers believe students have to learn. This approach can be behind the troubles in current programming courses. In parallel, younger children in schools are learning to program by creating games and robots with friendly tools, *e.g.*, Scratch, Alice and AgentSheets. Attempts have been made to bring these tools to

what is called "Introductory Computer Science", or CS0, and the results are encouraging, lowering drop-out levels and broadening the interest of students in computer science. Eventually, the transition to formal learning is done and the struggles reappear. That change of paradigm can be confusing, and it would be better to have a cross-cutting paradigm that will be used during the whole computer science degree.

In this thesis, I have explored the idea of errors as a useful resource for programming in the form of conflictive animations. Eventually, a whole computer science curricula could be designed around them. In the research presented, students learnt by finding the errors in conflictive animations. However, students complained of not being comfortable using the tool. In order to develop further the conflictive animation idea, a conflictive game is proposed and described in Paper VII and summarized next.

5.1.1 Conflictive Animations Game

The conflictive animation game presented in Paper VII aims to increase the engagement of the student by having them constructing the conflictive animation. The game part is when they have to find the errors in other students' animations. An environment akin to Peerwise [24] can be used to promote the best animations. In Peerwise, students answer and rate other students' multiple choice questions. In our game, the animations will be rated and errors found by the students.

5.1.2 Future work

The conflictive animations game is a concept that still needs to be developed and researched, but it should be seen as the first step to combine errors and games for computing education. Further work should clarify how to expand the combination of errors and games to the curricula. On one hand, having student deal with errors can improve their knowledge, skills and attitudes, as said in Paper V. On the other hand, games are great for motivation and engagement.

The eye-tracking evaluation suggested for Jeliot Adapt could be the method to further evaluate Jeliot ConAn and Jeliot with Explanations. Jeliot ConAn activities are meant for novice students paying more attention to the animation, eye-tracking data could easily confirm that. The empirical results of the evaluation of Jeliot with Explanations, students learn better when explanations are shown before the visualization, needs to be explored further. Eye-tracking data can serve to compare the different strategies students follow when being displayed the explanations.

The impact studies of Jeliot ConAn and Jeliot with Explanations would benefit from experiments that include a large sample of subjects and that last for an entire course of introduction to programming. However, in the case of Jeliot ConAn, given that it is a new concept and more needs to be understood, more qualitative studies are warranted to understand students' attitude towards conflictive animation and its role in their learning.

The definition of the roles that Jeliot 3 took with novices can be the basis for the design of a smart user interface. Jeliot 3 options and visualizations could change depending on the current understanding of the student. Students' verbal descriptions, after being analyzed via voice or text recognition, could be the basis of the adaptation.

In the experiment with Jeliot with Explanations, the data collected showed that students who watched explanations before the visualization gave better descriptions of the animations. The vocabulary they used in their answers was also closer to the one used in the explanations. These two phenomena are an indication of better learning; however, further research is needed to compare the viability of the mental models of the two groups after the intervention. The experiment as it was may be only testing for students' recall rather than understanding.

Recent theoretical developments present TPACK as a framework to study the interaction of technological knowledge with pedagogical and content knowledge [49]. TPACK makes explicit the linkages between the different types of knowledge that teachers require

to teach in a certain context. TPACK surveys assess the teachers' self-awareness and confidence on using technology in their teaching practice. This framework and associated tools can complement the work carried out here and the work of Ben-Bassat Levy [7] in studying the use, or lack of it, of visualization tools by programming teachers. In particular, it would be interesting to find out how explicitly teaching the roles of visualization tools to teachers can increase the teachers' *Technological Pedagogical Content Knowledge* as measured by TPACK instruments, and how it will reflect in the adoption of tools by teachers to teach programming.

5.2 IMPLICATIONS

The research presented here translates to practical implications for developers and teachers involved in program visualization.

5.2.1 Implications for Program Visualization Developers and Researchers

Developers and researchers of program visualization tools, and even any educational visualization tools, should accommodate for the different roles the tool takes with the students using it. Different roles require different support to smoothen student's path towards the evaluating role. Concept explanations given at the confusing role is the most obvious one as implemented in Paper VI. The exploring role of the tool could be supported with contextual help that displays when students move the cursor to the graphical blocks of the visualization.

Tackling adaptation in program visualization is better done in small conceptual steps, and not aiming to cover the language constructs and concepts. How to combine students' own programs with adaptation is still an open question.

Finally, researchers should not be afraid of exploring innovative ways to move the program visualization field further. Here, conflictive animations and a game were presented, but still many other

unconventional solutions are around that are fun for students and fun for the developers and researchers.

5.2.2 Implications for Teachers Using Program Visualization Tools

Jeliot 3, and other program visualization tools, usually represent faithfully the complexity of program execution and lots of fine details are not fully grasped by the students. In this case, that ability is very powerful and complete, but risks in students not having the knowledge, skills or attitudes to grasp the information [97]. Teachers should make sure that the graphical blocks of the visualization are understood before student can proceed with the animations.

Jeliot ConAn re-enforces the roles of errors in education and brings confusion. Teachers should be free to explore how to use errors and confusions in their lectures to keep students' attention. Errors and confusion can make students discomfortable, but learning should prepare students to accept discomfort for a greater good.

5.3 CONCLUDING REMARKS

The research carried out here has opened more doors than it has closed. Through the active research and development of Jeliot 3, the roles that Jeliot 3 takes have been defined, and several siblings have come to light and have been empirically evaluated. The final outcome is that more research and development in program animation and in Jeliot is necessary.

References

[1] R. Bednarik, "Potentials of eye-movement tracking in adaptive systems," in *Proceedings of the Fourth Workshop on the Evaluation of Adaptive Systems* (2005), pp. 1—8.

[2] R. Bednarik, *Methods to analyze visual attention strategies: Applications in the studies of programming*, PhD thesis (University of Joensuu, Department of Computer Science and Statistics, 2007), Available at `ftp://cs.joensuu.fi/pub/Dissertations/bednarik.pdf`.

[3] R. Bednarik, A. Moreno, N. Myller, and E. Sutinen, "Smart program visualization technologies: Planning a next step," in *Advanced Learning Technologies, 2005. ICALT 2005. Fifth IEEE International Conference on* (IEEE, 2005), pp. 717–721.

[4] R. Bednarik and M. Tukiainen, "An eye-tracking methodology for characterizing program comprehension processes," in *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06 (2006), pp. 125–132.

[5] M. Ben-Ari, "Constructivism in Computer Science Education," *Journal of Computers in Mathematics and Science Teaching* **20,** 45–73 (2001).

[6] M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen, "A decade of research and development on program animation: The Jeliot experience," *Journal of Visual Languages & Computing* **22,** 375–384 (2011).

[7] R. Ben-Bassat Levy and M. Ben-Ari, "We work so hard and they don't use it: acceptance of software tools by teachers," *SIGCSE Bulletin* **39,** 246–250 (2007).

[8] R. Ben-Bassat Levy, M. Ben-Ari, and P. A. Uronen, "The Jeliot 2000 program animation system," *Computers & Education* **40,** 1–15 (2003).

[9] T. Benaya and E. Zur, "Understanding Object Oriented Programming Concepts in an Advanced Programming Course," in *Informatics Education - Supporting Computational Thinking*, Vol. 5090, R. T. Mittermeir and M. M. Systo, eds. (Springer Berlin Heidelberg, 2008), pp. 161–170.

[10] J. Bennedsen and M. E. Caspersen, "Exposing the Programming Process," in *Reflection on the Teaching of Programming*, J. Bennedsen, M. E. Caspersen, and M. Kölling, eds. (Springer, 2008), pp. 6–16.

[11] J. Biggs and C. Tang, *Teaching for Quality Learning at University: what the student does*, 3 ed. (Open University Press, 2007).

[12] J. B. Biggs and K. F. Collis, *Evaluating the quality of learning: The SOLO taxonomy (structure of the observed learning outcome)* (Academic Press (New York), 1982).

[13] B. Bloom and D. Krathwohl, *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain* (Longman, 1984).

[14] R. Borasi, "Capitalizing on Errors as "Springboards for Inquiry": A Teaching Experiment," *Journal for Research in Mathematics Education* **25,** 166–208 (1994).

[15] M. Brayshaw and M. Eisenstadt, "A Practical Graphical Tracer for Prolog," *International Journal of Man-Machine Studies* **35,** 597–631 (1991).

[16] P. Brusilovsky, "Explanatory visualization in an educational programming environment: Connecting examples with general knowledge," in *Human-Computer Interaction*, Vol. 876, B. Blumenthal, J. Gornostaev, and C. Unger, eds. (Springer Berlin Heidelberg, 1994), pp. 202–212.

[17] P. Brusilovsky, J. Grady, M. Spring, and C.-H. Lee, "What should be visualized?: faculty perception of priority topics for program visualization," *SIGCSE Bulletin* **38,** 44–48 (2006).

[18] P. Brusilovsky, C. Karagiannidis, and D. Sampson, "The benefits of layered evaluation of adaptive applications and services," in *Workshop on Empirical Evaluation of Adaptive Systems* (2001), pp. 1–8.

[19] P. Brusilovsky and T. D. Loboda, "WADEIn II: a case for adaptive explanatory visualization," *SIGCSE Bulletin* **38,** 48–52 (2006).

[20] M. T. H. Chi, "Quantifying qualitative analyses of verbal data: A practical guide," *Journal of Learning Sciences* **6,** 271–315 (1997).

[21] R. C. Clark and R. E. Mayer, *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*, 3rd edition ed. (Wiley. com, 2011).

[22] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-D tool for introductory programming concepts," *Journal of Computing Sciences in Colleges* **15,** 107–116 (2000).

[23] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi, "Reversible Execution and Visualization of Programs with LEONARDO," *Journal of Visual Languages & Computing* **11,** 125 – 150 (2000).

[24] P. Denny, A. Luxton-Reilly, and J. Hamer, "The PeerWise System of Student Contributed Assessment Questions," in *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78*, ACE '08 (2008), pp. 69–74.

[25] F. Détienne, "Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams," *Interacting with Computers* **9,** 47–72 (1997).

[26] E. W. Dijkstra, "On the cruelty of really teaching computing science," (1988), circulated privately.

[27] B. du Boulay, "Some Difficulties of Learning to Program," *Journal of Educational Computing Research* **1,** 57–73 (1986).

[28] A. Eckerdal, *Novice Programming Students' Learning of Concepts and Practise*, PhD thesis (Uppsala University, Division of Scientific Computing, Numerical Analysis, 2009), Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-9551`.

[29] K. Ericsson and H. Simon, *Protocol Analysis: Verbal Reports As Data* (MIT Press, 1984).

[30] K. Falkner and E. Palmer, "Developing authentic problem solving skills in introductory computing classes," *SIGCSE Bulletin* **41,** 4–8 (2009).

[31] O. Feder, "Adding Multiple Language Support to Jeliot 3," (2008), Available at `https://code.google.com/p/jeliot3/source/browse/branches/PEliot/doc/PEliotDocumentation.doc`.

[32] A. E. Fleury, "Programming in Java: student-constructed rules," *SIGCSE Bulletin* **32,** 197–201 (2000).

[33] R. M. Gagne, "Learning outcomes and their effects: Useful categories of human performance.," *American Psychologist* **39,** 377 (1984).

[34] C. Große and A. Renkl, "Finding and Fixing Errors in Worked Examples: Can this Foster Learning Outcomes?," *Learning and Instruction* **17,** 612–634 (2007).

[35] L. Gugerty and G. M. Olson, "Comprehension differences in debugging by skilled and novice programmers," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers* (1986), pp. 13–27.

[36] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Terasvirta, and P. Vanninen, "Animation of user algorithms on the Web," in *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on* (IEEE, 1997), pp. 356–363.

[37] S. R. Hansen, N. H. Narayanan, and D. Schrimpsher, "Helping learners visualize and comprehend algorithms," *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning* **2,** 10 (2000).

[38] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quaterly* **28,** 75–105 (2004).

[39] C. Holmboe, "A cognitive framework for knowledge in informatics: the case of object-orientation," in *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education* (1999), pp. 17–20.

[40] C. Holmboe, "A framework for knowledge: Analysing high school students' understanding of data modelling," in *12th Workshop of the Psychology of Programming Interest Group* (2000), pp. 267–279.

[41] N. Hongwarittorrn and D. Krairit, "Effects of program visualization (Jeliot 3) on students' performance and attitudes towards java programming," in *The SPRING 8th International conference on Computing, Communication and Control Technologies* (2010), Available at `http://www.iiis.org/CDs2010/CD2010IMC/CCCT\_2010/Abstract.asp?myurl=TA750PM.pdf`.

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

[43] P. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference and consciousness* (Cambridge University Press, Cambridge, 1983).

[44] O. Kannusmäki, A. Moreno, N. Myller, and E. Sutinen, "What a novice wants: students using program visualization in distance programming course," in *Proceedings of the Third Program Visualization Workshop (PVW'04)* (2004), pp. 126–133.

[45] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke, "MatrixPro - a tool for demonstrating data structures and algorithms ex tempore," in *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on* (2004), pp. 892–893.

[46] A. Kerren and J. T. Stasko, "Algorithm Animation - Introduction," in *Revised Lectures on Software Visualization, International Seminar* (2002), pp. 1–15.

[47] S. Kirby, B. Toland, and C. Deegan, "Program Visualisation tool for teaching programming in C," in *Proceedings of the International Conference on Education, Training and Informatics, ICETI* (2010), Available at `http://www.iiis.org/CDs2010/CD2010IMC/ICETI\_2010/PapersPdf/EB134TP.pdf`.

[48] M. Knobelsdorf, E. Isohanni, and J. Tenenberg, "The reasons might be different: why students and teachers do not use visualization tools," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12 (2012), pp. 1–10.

[49] M. Koehler and P. Mishra, "What is technological pedagogical content knowledge (TPACK)?," *Contemporary Issues in Technology and Teacher Education* **9,** 60–70 (2009).

[50] M. Kölling, "The Greenfoot Programming Environment," *Transactions on Computing Education* **10,** 14:1–14:21 (2010).

[51] W. L. Kuechler and M. G. Simkin, "How well do multiple choice tests evaluate student understanding in computer programming classes?," *Journal of Information Systems Education* **14,** 389–400 (2003).

[52] A. Kumar and S. Kasabov, "Observer Architecture of Program Visualization," *Electronic Notes in Theoretical Computer Science* **178,** 153–160 (2007).

[53] S.-P. Lahtinen, E. Sutinen, and J. Tarhio, "Automated Animation of Algorithms with Eliot," *Journal of Visual Languages & Computing* **9,** 337–349 (1998).

[54] M. Lattu, V. Meisalo, and J. Tarhio, "A visualisation tool as a demonstration aid," *Computers & Education* **41,** 133 – 148 (2003).

[55] M. Lattu, J. Tarhio, and V. Meisalo, "How a Visualization Tool Can Be Used - Evaluating a Tool in a Research & Development Project," in *Proceedings of the 12th workshop of the Psychology of Programming Interest Group* (2000).

[56] T. Lauer, "Reevaluating and refining the engagement taxonomy," *SIGCSE Bulletin* **40,** 355–355 (2008).

[57] M. Limón, "On the cognitive conflict as an instructional strategy for conceptual change: a critical appraisal," *Learning and Instruction* **11,** 357–380 (2001).

[58] R. Lister, "Mixed methods: positivists are from Mars, constructivists are from Venus," *SIGCSE Bulletin* **37,** 18–19 (2005).

[59] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," *SIGCSE Bulletin* **36,** 119–150 (2004).

[60] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: novice programmers and the SOLO taxonomy," *SIGCSE Bulletin* **38,** 118–122 (2006).

[61] L. Ma, J. Ferguson, M. Roper, and M. Wood, "Investigating the viability of mental models held by novice programmers," *SIGCSE Bulletin* **39,** 499–503 (2007).

[62] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood, "Using cognitive conflict and visualisation to improve mental models held by novice programmers," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education* (2008), pp. 342–346.

[63] S. Maravić Čisar, D. Radosav, R. Pinter, and P. Čisar, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *International Journal of Computers Communications Control* **6,** 669–682 (2011).

[64] R. E. Mayer, "Different problem-solving competencies established in learning computer programming with and without meaningful models," *Journal of Educational Psychology* **67,** 725–734 (1975).

[65] R. E. Mayer, *Multimedia Learning* (Cambridge University Press, 2001).

[66] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," *SIGCSE Bulletin* **33,** 125–180 (2001).

[67] J. H. Meyer and R. Land, "Threshold concepts and troublesome knowledge (2): Epistemological considerations and a conceptual framework for teaching and learning," *Higher Education* **49,** 373–388 (2005).

[68] I. Milne and G. Rowe, "OGRE: Three-Dimensional Program Visualization for Novice Programmers," *Education and Information Technologies* **9,** 219–237 (2004).

[69] A. Moreno, "Algorithm Animation," in *Human-Centered Visualization Environments*, Vol. 4417, A. Kerren, A. Ebert, and J. Meyer, eds. (Springer Berlin Heidelberg, 2007), pp. 295–309.

[70] A. Moreno, "Intermediate Code in Program Animation Software," MSc thesis (Department of Computer Science, 2012).

[71] A. Moreno, R. Bednarik, and M. Yudelson, "How to Adapt the Visualization of Programs?," in *Proceedings of Workshop on Personalisation in E-Learning Environments at Individual and Group Level, 11th International Conference on User Modeling* (2007), pp. 65–70.

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

[73] A. Moreno, M. Joy, N. Myller, and E. Sutinen, "Layered Architecture for Automatic Generation of Conflictive Animations in Programming Education," *Learning Technologies, IEEE Transactions on* **3,** 139–151 (2010).

[74] A. Moreno, M. Joy, and E. Sutinen, "Roles of animation tools in understanding programming concepts," *Journal of Educational Multimedia and Hypermedia* **22,** 165–184 (2013).

[75] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing Program with Jeliot 3," in *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004* (2004), pp. 373–380.

[76] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing Program with Jeliot 3," in *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004* (2004), pp. 373–380.

[77] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller, "Conflictive animations as engaging learning tools," in *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Vol. 88, CRPIT (2007), pp. 203–206.

[78] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller, "Conflictive animations as engaging learning tools," in *Seventh Baltic*

Sea Conference on Computing Education Research (Koli Calling 2007), Vol. 88, CRPIT (2007), pp. 203–206.

[79] A. Moreno, E. Sutinen, and C. Islas Sedano, "A game concept using conflictive animations for learning programming," in *Games Innovation Conference (IGIC), 2013 IEEE International* (2013), pp. 175–178.

[80] A. Moreno, E. Sutinen, and M. Joy, "Defining and Evaluating Conflictive Animations for Programming Education: The Case of Jeliot ConAn," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14 (2014), pp. 629–634.

[81] R. Moreno and R. E. Mayer, "Cognitive principles of multimedia learning: The role of modality and contiguity.," *Journal of educational psychology* **91,** 358 (1999).

[82] N. Myller, "The Fundamental Design Issues of Jeliot 3," MSc thesis (Department of Computer Science, 2004), Available at `http://cs.uef.fi/jeliot/pub/theses.php`.

[83] N. Myller, "Automatic Generation of Prediction Questions during Program Visualization," *Electronic Notes in Theoretical Computer Science* **178,** 43 – 49 (2007), Proceedings of the Fourth Program Visualization Workshop (PVW 2006).

[84] N. Myller, R. Bednarik, E. Sutinen, and M. Ben-Ari, "Extending the Engagement Taxonomy: Software Visualization and Collaborative Learning," *Transactions on Computing Education* **9,** 7:1–7:27 (2009).

[85] T. L. Naps, J. R. Eagan, and L. L. Norton, "JHAVÉ - an environment to actively engage students in Web-based algorithm visualizations," *SIGCSE Bulletin* **32,** 109–113 (2000).

[86] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide, "Exploring the role

of visualization and engagement in computer science education," in *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education* (2002), pp. 131–152.

[87] S. Nevalainen and J. Sajaniemi, "An experiment on short-term effects of animated versus static visualization of operations on program perception," in *Proceedings of the second international workshop on Computing education research*, ICER '06 (2006), pp. 7–16.

[88] S. Nevalainen and J. Sajaniemi, "An experiment on the short-term effects of engagement and representation in program animation," *Journal of Educational Computing Research* **39,** 395–430 (2008).

[89] J. Nielsen, "Finding Usability Problems Through Heuristic Evaluation," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92 (1992), pp. 373–380.

[90] D. Norman, Chap 1 in *Some Observations on Mental Models* (Lawrence Erlbaum Associates, Inc., New Jersey, 1983).

[91] J. F. Nunamaker, Jr., M. Chen, and T. D. M. Purdin, "Systems development in information systems research," *Journal of Management Information Systems* **7,** 89–106 (1990).

[92] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in *Revised Lectures on Software Visualization, International Seminar* (2001), pp. 176–190.

[93] R. Or-Bach and I. Lavy, "Cognitive activities of abstraction in object orientation: an empirical study," *SIGCSE Bulletin* **36,** 82–86 (2004).

[94] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology* **19,** 295–341 (1987).

[95] D. Perkins, "The many faces of constructivism.," *Educational leadership* **57,** 6–11 (1999).

[96] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, , and R. Simmons, "Conditions of Learning in Novice Programmers," in *Studying the Novice Programmer*, J. C. Spohrer and E. I. Soloway, eds. (Ablex Publishing Company, 1989), pp. 261–279.

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

[98] S. Pollack and M. Ben-Ari, "Selecting a visualization system," in *Proceedings of the third program visualization workshop* (2004), pp. 134–140.

[99] G. Polya, *How to Solve It* (Princeton University Press, 1971).

[100] N. Postman, Chap The Fallen Angel in *The End of Education* (Vintage, 1996).

[101] B. A. Price, R. M. Baecker, and I. S. Small, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages & Computing* **4,** 211–266 (1993).

[102] N. Ragonis and M. Ben-Ari, "On understanding the statics and dynamics of object-oriented programs," *SIGCSE Bulletin* **37,** 226–230 (2005).

[103] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski, "Effectiveness of Program Visualization: A Case Study with the ViLLE Tool.," *Journal of Information Technology Education* **7,** 15–32 (2008).

[104] R. F. Raposa, *Java in 60 Minutes A Day* (Wiley, 2003).

[105] D. D. Riley, "Teaching problem solving in an introductory computer science class," *SIGCSE Bulletin* **13,** 244–251 (1981).

[106] G. Roman and K. Cox, "A taxonomy of program visualization systems," *Computer* **26,** 11–24 (1993).

[107] G. Rowe and G. Thorburn, "VINCE—an on-line tutorial tool for teaching introductory programming," *British Journal of Educational Technology* **31,** 359–369 (2000).

[108] J. Sajaniemi and M. Kuittinen, "Program animation based on the roles of variables," in *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03 (2003), pp. 7–ff.

[109] J. Sajaniemi and M. Kuittinen, "An Experiment on Using Roles of Variables in Teaching Introductory Programming," *Computer Science Education* **15,** 59–82 (2005).

[110] J. Sajaniemi and M. Kuittinen, "From procedures to objects: A research agenda for the psychology of object-oriented programming education," *Human Technology* **4,** 75–91 (2008).

[111] J. Sajaniemi, M. Kuittinen, and T. Tikansalo, "A study of the development of students' visualizations of program state during an elementary object-oriented programming course," *Journal on Educational Resources in Computing* **7,** 3:1–3:31 (2008).

[112] P. H. Scott, H. M. Asoko, and R. H. Driver, "Teaching for conceptual change: A review of strategies," in *Research in Physics Learning: Theoretical Issues and Empirical Studies* (1991), pp. 71–78.

[113] J. P. Smith, III, A. A. diSessa, and J. Roschelle, "Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition," *The Journal of the Learning Sciences* **3,** 115–163 (1993-1994).

[114] J. P. Smith III, A. A. Disessa, and J. Roschelle, "Misconceptions reconceived: A constructivist analysis of knowledge in transition," *The journal of the learning sciences* **3,** 115–163 (1994).

[115] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *Software Engineering, IEEE Transactions on* 595–609 (1984).

[116] J. Sorva, "Reflections on threshold concepts in computer programming and beyond," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10 (2010), pp. 21–30.

[117] J. Sorva, *Visual Program Simulation in Introductory Programming Education*, PhD thesis (Aalto University, Department of Computer Science and Engineering, 2012), Available at `http://lib.tkk.fi/Diss/2012/isbn9789526046266/`.

[118] M. Tedre, *The Development of Computer Science: A Sociocultural Perspective*, PhD thesis (University of Joensuu, Department of Computer Science, 2006), Available at `http://epublications.uef.fi/pub/urn\_isbn\_952-458-867-6/`.

[119] E. Thompson, A. Luxton-Reilly, J. L. Whalley, M. Hu, and P. Robbins, "Bloom's taxonomy for CS assessment," in *Proceedings of the tenth conference on Australasian computing education - Volume 78*, ACE '08 (2008), pp. 155–161.

[120] S. Thompson, "Where do I begin? A problem solving approach in teaching functional programming," in *Programming Languages: Implementations, Logics, and Programs*, Vol. 1292, H. Glaser, P. Hartel, and H. Kuchen, eds. (Springer Berlin Heidelberg, 1997), pp. 323–334.

[121] J. Urquiza-Fuentes and J. Velázquez-Iturbide, "Toward the effective use of educational program animations: The roles of student's engagement and topic complexity," *Computers & Education* **67,** 178–192 (2013).

[122] J. Urquiza-Fuentes and J. A. Velázquez-Iturbide, "A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems," *Transactions on Computing Education* **9,** 9:1–9:21 (2009).

[123] V. Vainio and J. Sajaniemi, "Factors in novice programmers' poor tracing skills," *SIGCSE Bulletin* **39,** 236–240 (2007).

[124] A. Venables, G. Tan, and R. Lister, "A closer look at tracing, explaining and code writing skills in the novice programmer," in *Proceedings of the fifth international workshop on Computing education research workshop*, ICER '09 (2009), pp. 117–128.

[125] A. T. Virtanen, E. Lahtinen, and H.-M. Järvinen, "VIP, a visual interpreter for learning introductory programming with C++," in *Proceedings of The Fifth Koli Calling Conference on Computer Science Education* (2005), pp. 125–130.

[126] L. S. Vygotsky, "Mind in society," (1978).

[127] P. Wang, "Exploring impact of the order of explanations and animations in Jeliot 3," MSc thesis (University of Eastern Finland, School of Computing, 2012), Available at `http://cs.uef.fi/jeliot/pub/theses.php`.

[128] P. Wang, R. Bednarik, and A. Moreno, "During automatic program animation, explanations after animations have greater impact than before animations," in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, Koli Calling '12 (2012), pp. 100–109.

[129] G. Weber and P. Brusilovsky, "ELM-ART: An Adaptive Versatile System for Web-based Instruction," *International Journal of Artificial Intelligence in Education* **12,** 351–384 (2001).

[130] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. Corritore, "A comparison of the comprehension of object-oriented and procedural programs by novice programmers," *Interacting with Computers* **11,** 255–282 (1999).

[131] M. Yudelson, P. Brusilovsky, and V. Zadorozhny, "A User Modeling Server for Contemporary Adaptive Hypermedia:

An Evaluation of the Push Approach to Evidence Propagation," in *User Modeling 2007*, Vol. 4511, C. Conati, K. McCoy, and G. Paliouras, eds. (Springer Berlin Heidelberg, 2007), pp. 27-36.

Andrés Moreno

Re-designing Program Animation

From tools' roles to new learning activities

Programming animation tools aim to lower the cognitive barriers to learning programming by graphically representing the expert's view on programming. However, students who use them face the problem of not understanding the animations. This work presents the roles a programming animation tool, Jeliot 3, takes when students use the tool to understand new concepts. These roles have led to the development of conflictive animations, a novel way to engage students in learning with animations.

UNIVERSITY OF
EASTERN FINLAND

Publications of the University of Eastern Finland
Dissertations in Forestry and Natural Sciences

ISBN 978-952-61-1542-9
ISSNL 1798-5668
ISSN 1798-5668
ISBN 978-952-61-1543-6 (PDF)
ISSNL 1798-5668
ISSN 1798-5676