The State Of Play: A Notional Machine for Learning Programming*

Michael Berry
School of Computing
University of Kent
Canterbury, Kent, UK
mjrb5@kent.ac.uk

Michael Kölling
School of Computing
University of Kent
Canterbury, Kent, UK
mik@kent.ac.uk

ABSTRACT
Comprehension of programming and programs is known to be a difficult task for many beginning students, with many computing courses showing significant drop out and failure rates. In this paper, we present a new notional machine design and implementation to help with understanding of programming and its dynamics for beginning learners. The notional machine offers an abstraction of the physical machine designed for comprehension and learning purposes. We introduce the notional machine and a graphical notation for its representation. We also present *Novis*, an implementation of a dynamic real-time visualiser of this notional machine, integrated into BlueJ.

Categories and Subject Descriptors
H.3.2 [**Computers and Education**]: Computer and Information Science Education; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Animations*; I.6.8 [**Simulation and Modelling**]: Types of Simulation—*Animation, Visual*

General Terms
Human Factors

Keywords
Program visualization, novice programming, Novis

1. INTRODUCTION
It is well understood that programming is a fundamental activity in computer science; it is the process by which conceptual ideas are mapped to instructions that can be understood and interpreted by a machine. The teaching of introductory programming within computer science is essential, and mastery of this skill necessary for students to progress. To be successful in programming, students have to be able to form a valid and consistent mental model of the machine executing their instructions. Forming such a model is not easy, and the computing education community has no agreed, shared abstract model in widespread use. Often, ad-hoc models are formed by instructors or students, but these are not guaranteed to be consistent or correct. A shared, accepted and valid mental model – a notional machine – would benefit both instructors and students in their attempts to teach and learn programming.

*This paper expands on a previous short paper, presented at WiPSCE 2013, Aarhus, Denmark.

1.1 Notional Machines
The difficulties of learning to program are well documented; Kim & Lerch, for example, provide a summary[7]. Many students fail or drop out of introductory courses, with a failure rate of 33% reported by Bennedsen and Caspersen not out of line with many courses around the world[2]. A popular hypothesis presented by Boulay[3] states that students find the concepts of programming too hard to grasp, do not understand the key properties of their program, and do not know how to control them by writing code. Boulay took this as a starting point and motivation to formalise the concept of a *notional machine*. A notional machine is an abstraction designed to provide a model to aid in understanding of a particular language construct or program execution. The notional machine does not need to accurately reflect the exact properties of the real machine; it presents a higher conceptual level by providing a metaphorical layer above the real machine (or indeed several such layers) that are hoped to be easier to comprehend than the real machine.

Some teachers, when presented with the idea of a notional machine, are initially skeptical, holding the view that students need to understand what "really happens" to become expert programmers. It should be noted that all models held by almost all programmers are notional, in that they represent simplifications of the real machine. Even discussions about assembly language or machine code are almost necessarily abstractions, since hardware optimisations of modern processors are so complex that they cannot fully be taken into account when reasoning about program execution (other than by a small group of highly trained specialists working on processor design). In addition, details of processor designs are often trade secrets of the manufacturer – we cannot actually know what "really happens".

Therefore a meaningful discussion about notional machines does not centre around the question whether or not to use one, but around the most useful level of abstraction to aim

for. Whatever the preferred abstraction level, it is important that the notional machine is complete and consistent: it must be able to explain all observable behaviour of the real machine, and reasoning about the notional machine must allow accurate predictions to be made about behaviour of the real machine[4].

The design of the notional machine will typically be heavily influenced by the programming paradigm of the language used for implementation. In this paper, we discuss a notional machine for programs written in Java, therefore representing an object-oriented model.

A notional machine's metaphorical layer can be presented in many forms. Visual metaphors are most commonly used to present state and events that unfold in the actual machine. Visual representations can be replaced or augmented by other media, such as sound.

1.2 The status quo

At present, one of the most common techniques for teachers to explain dynamic elements of object orientation, and the execution of object-oriented programs, is through the drawing of diagrams of objects and classes, often by hand on a whiteboard. No consistent, complete and widely accepted shared notation exists across classrooms, and it is left to the student to form a mental model based on often ad-hoc diagrams the teacher may use. UML provides a variety of standardised notations, but its dynamic diagrams are often perceived as too complex and therefore are not widely used in introductory teaching. Students are often confronted with differing notations to describe the state of a program when switching between teachers, textbooks or tutorials.

One contribution of this work is to provide a shared model and notation that can be used by teachers and lecturers, in textbooks and in discussions. It provides learners with a consistent, correct and useful representation to support the formation of a mental model that transfers to a number of contexts and environments.

The second contribution is *Novis*, an implementation of this notional machine in a software system. Novis is integrated as a new main interface in an experimental version of the BlueJ environment[8], where it replaces the traditional object bench. It uses the notional machine notation to visualise the execution of a Java program in real time. It can show the state of a program at selected points in time of the execution, or it can animate the execution over a period of time.

2. RELATED WORK

Several educational software systems are in use in classrooms that offer presentations and animations of notional machines. UUhistle[12] is a software tool that provides animated, live visualisations of the execution of Python programs. The model employed operates at a fairly low level, animating single statements to illustrate the functionality of single constructs, such as assignment or parameter passing. A related tool, Jeliot[10], operates at a similar conceptual level to UUhistle using the Java language. Both of these tools lose their usefulness once the functionality of the basic programming language constructs is understood by the learner. The level of abstraction is too low to usefully visualise larger examples or more complex data structures, and therefore these tools are often employed for only a few weeks at the beginning of a learning experience. In contrast, a goal for our notional machine design is to be able to visualise somewhat larger examples and to be useful to illustrate or investigate program behaviour even after basic constructs have been mastered.

Also of interest is JGrasp[5]; an integrated environment providing several separate visualisations of parts of the system. Its visualisations operate at a higher level than UUhistle and Jeliot, with a data structure viewer being of specific interest. This viewer shows objects and their relations in a layout and representation purpose-built for several known data structures.

The use and effectiveness of these systems for learning is still under debate. Although literature regarding algorithm visualisation effectiveness is readily available, literature on program visualisation is more scarce. For algorithm visualisations, one meta-study[6] found a high correlation of effectiveness in those studies that actively involved the students. Similar results have not yet been shown for program visualisations. Where literature does exist, it is far from conclusive, with different studies even on the same tool claiming different results. In one study evaluating Jeliot's effectiveness, Moreno and Joy found that on average, the transfer of knowledge from the tool to the student was not successful[9]. However, a different study (also using Jeliot) claims "a significant percentage of students had achieved better results when they were using a software visualisation tool"[11].

For our own work this means that demonstrating the effectiveness of the tool has yet to be demonstrated in future work. No convincing prior work exists that allows reliable conclusions to be drawn about the efficacy of such systems.

3. RESEARCH QUESTIONS

This work supports two distinct and separate use cases: the *comprehension of programming* and the *comprehension of programs*. The first is most relevant for beginning programmers: the goal here is to understand how a computing system executes program code, the mechanics and details of a programming language and the concepts of the underlying paradigm. Typical questions that the system helps to answer in this case are *What does an assignment statement do?* or *How does a method call work?* For experts who have mastered the language this aspect is no longer relevant.

The second use case is to understand and investigate a given program. The goal is to become familiar with a given software system, or to debug a program. Typical questions in this case are *Why does my program behave like this?* or *How many objects are being created when I invoke this method?* This part of the functionality remains relevant even for seasoned programmers.

These use cases lead us to the main aims of the model:

Aim 1 : To provide a shared notation for representing the execution of an object-oriented program within the proposed model.

Aim 2 : To provide a valid mental model for learning and reasoning about object-oriented programming.

Aim 3 : To provide a basis for an implementation in software that can be used to provide a visualisation of the model alongside a running object-oriented program.

These aims further lead us to the two principle research questions:

Research question 1 : What should the notation for a high level, consistent model of a notional machine, developed to aid novices in learning to program look like?

Research question 2 : Can a software tool be created that dynamically visualises the execution of typical beginners' programs using this notional machine notation in a way that is manageable and useful?

For the purpose of RQ1, we define *consistent* to mean that valid reasoning within that model must correctly predict the behaviour of the underlying system. Our targeted problem space covers Java programs of a complexity up to first year university programming problems. Thus, we can explicitly exclude some constructs from our model, if we postulate that they are outside our targeted problem space. This is discussed in more detail below (section 4).

This paper presents the design of the notional machine and does not include an evaluation of its effectiveness for learning or teaching. This will be presented separately in a later paper.

4. PROBLEM SPACE

Our notional machine is aimed at first year programming students and therefore focuses on material typically covered within that year. While the model and software system may well remain useful for tasks in later years, where a conflict between scope and simplicity emerges, simplicity will take precedence for constructs not typically discussed in introductory programming courses.

For a more systematic definition of the problem space, we look at programming examples and projects covered in some popular introductory textbooks. A small number of the most popular introductory Java textbooks (including *Objects First With Java*[1], a book frequently used when teaching with BlueJ) are used to set the scope against which completeness is defined. The notional machine should be able to model and visualise all examples from these books.

4.1 Abstractions

Tying the scope of the notional machine to first year programming examples places implicit bounds on the abstractions that should be shown in the notional machine. Many advanced concepts, such as explicit concurrency and synchronisation, packages, class loading and annotation processing can be excluded from the scope – these are rarely covered in first year programming courses. In addition, we do not aim to illustrate abstractions at the very low level. Simple statements, such as assignments or if-statements, are not represented in the model – the most atomic "step" in the model is defined as a single method execution, and the model focusses on object interaction and method calls. Students typically do not have long term trouble in understanding basic syntax and the behaviour of these simple statements, therefore a visualisation of these would not remain helpful in the longer term.

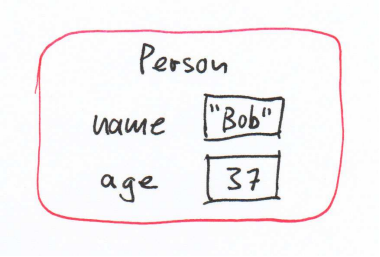

Figure 1: Representation of an object with two fields.

The abstractions used for presentation of the notional machine centre around objects and classes, and their interactions. Objects and classes are represented with their current state. The state visualised includes both the state of objects and classes at a given point in time, including primitive fields and object references, and the state of any execution currently in progress, visualising the current locus of execution as well as the path of invocation (traditionally shown as a stack trace).

5. NOTIONAL MACHINE NOTATION

The notation for the notional machine is designed to be usable in various different media and formats, including line drawings (possibly produced using drawing software), hand drawings on paper (see Figure 1) or a whiteboard, or generated automatically with a software tool (discussed in more detail below).

Some elements of the notation, such as fill colour, are defined but optional and mostly relevant to automatically generated versions of the notional machine diagrams. Hand drawn versions of the diagram are still be readable without the optional elements.

5.1 Notation details

5.1.1 Objects

Objects are represented by rectangles with rounded corners. The object is labelled with its type. The type displayed is the dynamic type, not the type of any declared variable. If fill colours are used, objects are red (Figure 2). Objects do not have unique identifiers, as in some other object visualisation systems. Assigning unique identifiers may sometimes help to talk about the objects, but leads to misconceptions that should be avoided.

A field of an object is presented as a box with a label to its left. The box is white (if colours are used), and it contains the field's value. An object representation contains a list of all its fields. If the value of the field is a primitive or a string, it is displayed in its textual form within the representation of

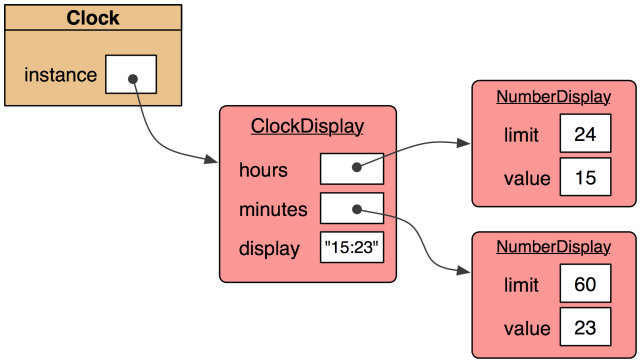

Figure 2: Representation of object references.

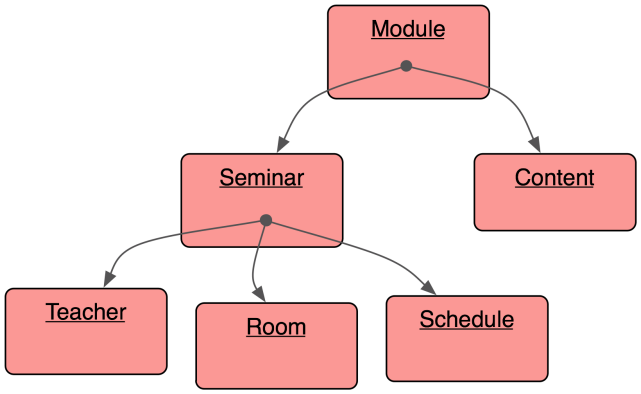

Figure 3: This shows the simplified view – where fields are not of direct interest, they can be omitted as in this example.

the field. Only instance fields are shown on the object; static fields are displayed on the class instead (see section 5.1.4).

5.1.2 *References*

If the value of a field is a reference to another object then it is displayed as an arrow originating from the field and pointing to the object that it references (Figure 2).

Strings are treated as a special case. While strings are objects in Java, they are displayed using a literal representation (the characters of the string in double quotes, written directly into the field). The immutability of string objects ensures that this representation does not lead to inconsistent or invalid conclusions, and this representation significantly simplifies many examples. This exception only applies to strings; all other objects must be represented in their full form via an arrow.

5.1.3 *Simplified view*

Often the relationships between objects, the object graph, is the sole point of interest. For this case a simplified notation of an object may be used. In this notation fields are omitted and references are drawn originating from the centre of the referring object (Figure 3). Primitive field values are not shown in the simplified view.

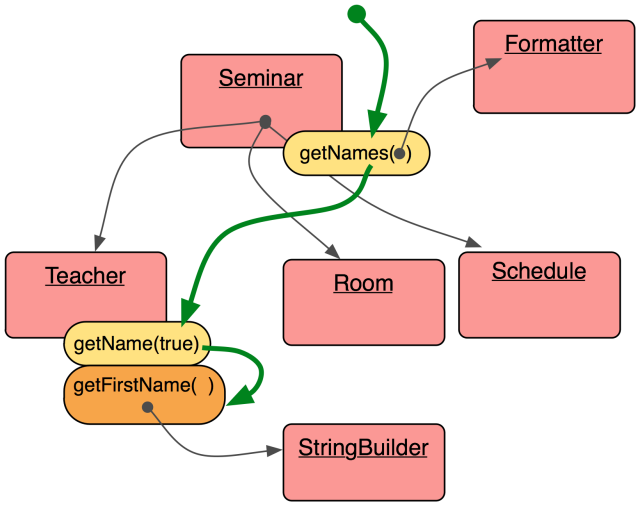

Figure 4: An active method call chain with object parameters.

5.1.4 *Classes*

Classes are represented as rectangles with hard (not rounded) corners (Figure 2). The class is labelled with its name and static members are displayed in the class icon. If colours are used classes have a light brown sand colour.

The display of static fields in the class is similar to that of instance fields in objects – primitives and strings are displayed inline, object references are displayed with arrows.

5.1.5 *Method calls*

In addition to the state of objects and classes the notional machine diagram can also show the state of a currently active execution. While the notation described so far represents a classic heap diagram, the execution state at any given point in time is most often depicted as a separate diagram, showing a call stack. In our notation, the execution state is overlayed over the same diagram.

An active method call is depicted as an orange (if coloured), oval-shaped label attached to the bottom right of the object that it is called on (Figure 4). The label remains visible as long as the method is active. If this method calls further methods, those are displayed as well, linked by a call chain arrow. In colour representations, the call chain arrow is green. Further method labels may appear attached to the same object or other objects, depending on the receiver of the method call.

The call chain arrow (or, in talking about a notional machine diagram, usually just ”call chain”) depicts the complete current sequence of open method calls, their dependencies and order. When using this notation on a whiteboard, the call chain is often extended to show nested method calls, and wiped out again to illustrate the completion of a method invocation.

Method labels include the list of actual parameters. For primitives and strings, the parameter value is shown as a

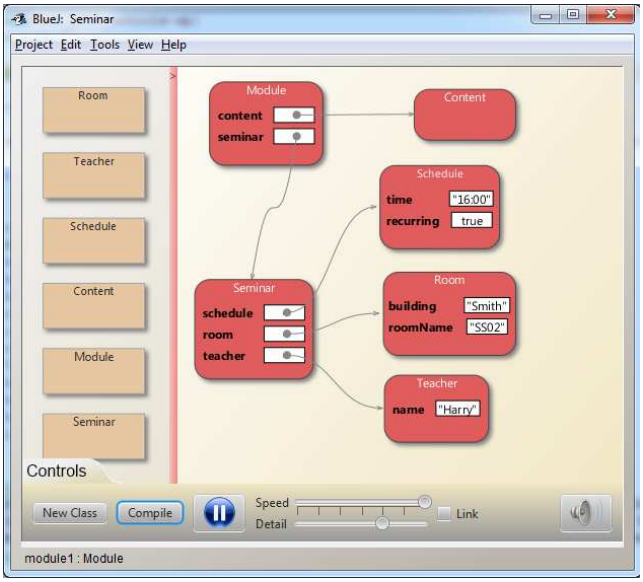

Figure 5: The Novis notional machine visualiser with a simple example.

literal; for object parameters, a reference to the object is shown originating from the parameter list (see *getNames* method, Figure 4). References from local variables may be shown originating from the method label, below the method name (see *getFirstName* method, Figure 4).

6. SOFTWARE VISUALISATION

Novis, a visualisation creating automatic and animated versions of notional machine diagrams as described here, has been created and integrated into BlueJ's main interface. This implementation can present static notional machine diagrams at selected stages of program execution, or animate ongoing execution in real time.

6.1 Static display

Novis depicts the static view of classes and their references between them as described in section 5.1 (Figure 5). Objects in the diagram are placed automatically when they are created but can be moved by the user; clicking on the object toggles between its simplified and detailed state. The reference arrows are placed and updated automatically by the software.

6.2 Object creation and destruction

When an object is created the software searches for an appropriate space in the diagram and "pops" the object into that space (with a short animation) before executing its constructor. The object is initially grey in colour to indicate that it has not been fully instantiated, and then switches to its default red colour on the constructor's completion. User generated objects (created interactively from the class, as was always possible in BlueJ) are removed manually by the user; objects created in code are removed from the diagram whenever they become eligible for garbage collection (not when they are actually collected). In both cases, the objects disappear with a brief "puff of smoke" animation.

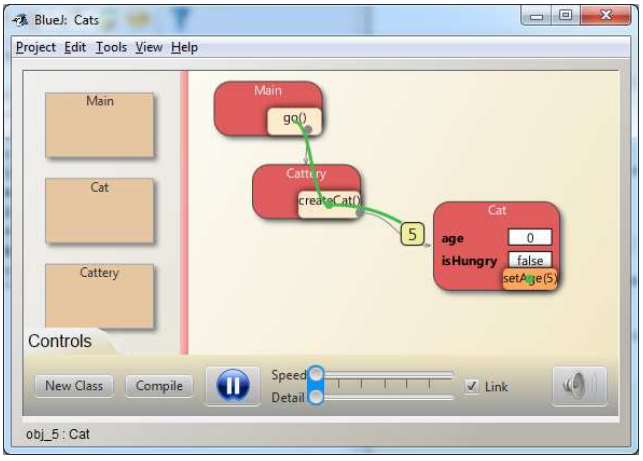

Figure 6: Novis displaying a method call chain.

6.3 Execution

Methods can be interactively invoked on any object in the traditional BlueJ-style, by right clicking the object and selecting the method from the resulting context menu. A following chain of method calls is then depicted as described in section 5.1.5. As methods are called, the method call labels appear in the animation, drawing the attention of the viewer. The call chain arrow is animated – it extends from its origin to its destination. Parameter values are depicted in a moving animation, following the call chain arrow from the caller to the invoked method (Figure 6).

6.4 Speed and stepping granularity

The notional machine viewer includes a slider to control the speed of the animation. At its maximum setting, no delay is added and the program executes at the maximum speed possible with the current choice of animation detail (see section 6.5, below). At the slowest, a two-second pause is added between each step of the program. The interim levels have pauses that scale linearly between these two values. A "step" of the program in our context is a method call or a method return – single statement executions are not visualised.

6.5 Level of detail

A second slider in the interface controls the level of detail displayed in the diagram. With full detail visible, the animation performs as described above: objects are shown in detailed view with their fields visible, object creation and destruction are animated, and method calls are dynamically visualised with call chain arrows slowly extending, parameter values passed visually from one method to another, and return values moving the other way at the end of a method execution as the call chain arrow retracts.

This level of detail is useful in early stages of learning, when the focus of the learner is on understanding basics of object interaction and method calls, when examples are small and execution chains short. In later examples, this level of detail becomes a hindrance, illustrating concepts that have already been understood and obscuring information about the program under investigation.

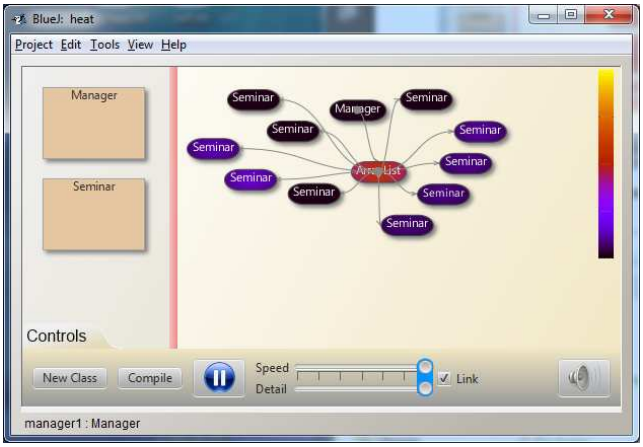

Figure 7: Heatmap view illustrating program activity.

At that stage, the level of detail displayed can be reduced. The visualisation offers seven levels of detail display, gradually reducing or omitting various animations and display elements as the setting is decreased. The lower-detail settings show objects in their simplified view by default, resulting at the extreme end in a "heatmap" view that focuses on object creation, desctruction and activity levels (see section 6.6).

The two sliders – speed and detail – can be linked in the user interface to allow both to be adjusted in a single interface gesture. When linked, they are inversely related: the higher the speed, the less detail is displayed. The linked states represent commonly useful settings when viewing typical examples.

User control over speed and animation detail ensures that our notional machine visualisation addresses a broad range of use cases and remains relevant after the first few weeks of programming instruction. While some settings support the understanding of basic constructs (such as object references and method calls), others allow the investigation of specific data structures and their associated algorithms, the study of specific programs, or specialised debugging tasks.

6.6 Heatmap

At the lowest level of detail Novis's display turns into a heatmap (Figure 7). Objects are shown in a compact notation using just enough space to display their type, and colour is used to indicate activity. Method calls are not textually displayed; instead, objects "warm up" as methods are invoked, first turning a lighter purple, then red, then yellow with increased activity. All objects cool down gradually when not being active, so that the most recent active objects are easily recognisable. This notation – depicting object creation and destruction, as well as hotspots of activity – provides a quick high level overview of programs with ongoing activity.

7. STATUS AND FUTURE WORK

A prototype implementation of BlueJ with Novis integrated to replace the object bench has been completed, and is currently available for testing and evaluation purposes. The system has been tested with a small number of users with promising results, but no formal user studies have yet been completed.

Work in the near future will concentrate on further testing of usability and effeciveness with first year students, including studies to evaluate effects on program comprehension. The results from these studies will then be used to refine the interface and functionality of the model and corresponding implementation.

8. ACKNOWLEDGEMENTS

We wish to thank Michael Caspersen for many discussions about notional machines and their potential uses in programming education. His ideas were instrumental in starting and shaping this project.

9. REFERENCES

[1] David J Barnes and Michael Kölling. *Objects first with Java: a practical introduction using BlueJ*. Pearson, Boston, 2012.

[2] Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *SIGCSE Bull.*, 39(2):32–36, June 2007.

[3] Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2:57–73, 1986.

[4] Michael Edelgaard Caspersen. *Educating Novices in The Skills of Programming*. DAIMI PhD Dissertation. Department of Computer Science, 2007.

[5] T. Dean Hendrix and James H. Cross II. jGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. *J. Comput. Sci. Coll.*, 23(2):170–172, December 2007.

[6] Christopher Hundhausen, Sarah A. Douglas, and John T Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

[7] Jinwoo Kim and F. Javier Lerch. Why is programming (sometimes) so difficult? programming as scientific discovery in multiple problem spaces. *Information Systems Research*, 8(1):25 –50, March 1997.

[8] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13:249–268, December 2003.

[9] Andrés Moreno and Mike S. Joy. Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science*, 178(0):51–59, July 2007.

[10] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. page 373. ACM Press, 2004.

[11] Sanja Maravic Cisar, Dragica Radosav, Robert Pinter, and Petar Cisar. Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3. *International Journal of Computers Communications & Control*, 6, 2011.

[12] Juha Sorva and Teemu Sirkiä. UUhistle. pages 49–54. ACM Press, 2010.