Novis: A notional machine implementation for teaching introductory 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 notional machine implementation, *Novis*, 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. *Novis* provides a real-time visualisation of this notional machine, and is integrated into BlueJ.

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.

1.1 Notional Machines

The difficulties of learning to program are well known; 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 Du Boulay [4] 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. Du 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".

A meaningful discussion about notional machines therefore 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 [5].

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.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

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.

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 [9], 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 [14] 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 [12], 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.

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. 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 [11]. 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" [13].

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 in an object-oriented language 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. A usability study related to the tool will be presented separately in a future 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. These books were methodically chosen by taking the top ten selling relevant, introductory Java programming textbooks from Amazon. The notional machine should be able to model and visualise all examples from these books.

5. DESIGN

The notional machine notation has been partly described in a previous paper [3]. It also introduced Novis, an implemented form of the notional machine that has been integrated into BlueJ's main interface. Novis can present static notional machine diagrams at selected stages of program execution, or animate ongoing execution in real time. An earlier prototype of Novis was also presented in a previous paper [3], the version described here is more complete and the description more detailed.

5.1 BlueJ

The notional machine and visualisation that this work has produced is integrated as part of the BlueJ IDE, an existing IDE designed for educational use [9]. The design of BlueJ is based around Blue [8], an earlier system that incorporated a similar environment with a separate language. BlueJ is a popular tool; it has been translated into 17 languages and "is used in introductory programming courses at secondary schools and Universities worldwide" [15].

The majority of BlueJ's main interface contains a class diagram which is useful for visualising the static relationships between classes and interfaces. Any classes created are added to the diagram, and both their "uses" and inheritance relationships are shown. The layout is partially automated; the user positions the classes by hand, but the inheritance and use arrows are drawn automatically. An object bench is also provided. Constructors of the classes in the class diagram can be interactively invoked, creating an instance of the class on the object bench. These current elements provide limited functionality with respect to dynamic visualisation; in particular the class diagram only shows compile time relationships.

Novis provides a new view that utilises the real estate previously occupied by the class diagram and object bench.

5.2 Class diagram

Novis depicts the static view of classes and their references between them in a very similar way to BlueJ (Figure 1). As with BlueJ, the user is responsible for the layout of the classes but Novis draws the arrows automatically. When objects are created in this view, they are displayed "collapsed" towards the right hand side.

Classes in the diagram are placed automatically when they are created but can be moved by the user; clicking on the class toggles between its simplified and detailed state. The inheritance and use arrows are placed and updated automatically by the software.

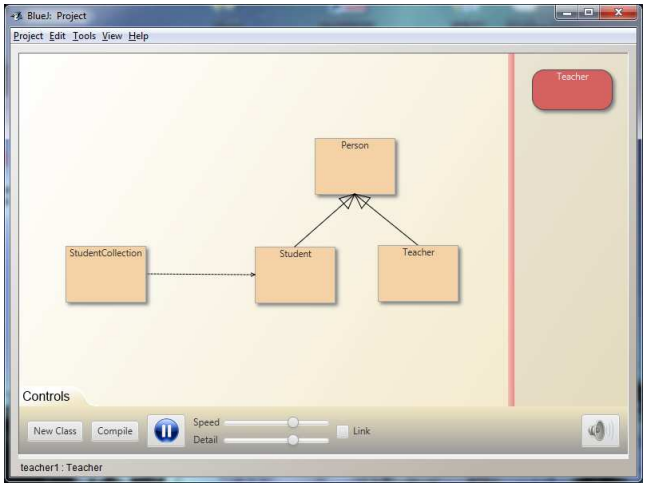

Figure 1: The class diagram shown in Novis.

5.3 Object creation and destruction

The class diagram may be collapsed towards the left hand side of the window at any time so that the dynamic object visualisation fills the majority of the window (Figure 2). Objects may be either *user created* or *implicitly created*. User created objects are created as a direct result of the user executing an object's constructor; implicitly created objects are instead created as a result of other code executing. When an object is created, Novis searches for an appropriate space in the diagram and places the object into that space (with a short animation) before executing its constructor.

Initially, the object is displayed grey in colour to indicate that it has not been fully instantiated. It then switches to its default red colour on the constructor's completion. If an exception occurs during the object's constructor, it may remain in its default grey colour to indicate that a problem has occurred with its instantiation.

User created objects may only be removed manually, by right clicking on the object and selecting "Remove object". Implicitly created objects are always removed from the visualisation when they become eligible for garbage collection (not when they are actually collected, which may of course be some time later.) They may not be removed manually, since forcibly removing an object from the graph is not normal behaviour, and would likely have undesirable side effects! In both cases, the objects disappear with a brief "puff of smoke" animation, drawing the user's attention to the disappearance of the object. As opposed to the traditional BlueJ object bench, Novis visualises all objects created as part of a program execution, not just the top level objects explicitly created by the user.

5.4 Viewing object state

Objects are initially created in their "collapsed" state, that is we do not see details about the object's underlying state. If a user wishes to view the contents of an objects fields however, this can be done by clicking on the object. The name of the field is displayed as a label on the left hand side, and the value of the field is displayed on a white box on the right hand side (Figure 3). The field value may be an inline value or a reference. In either case, the values of these

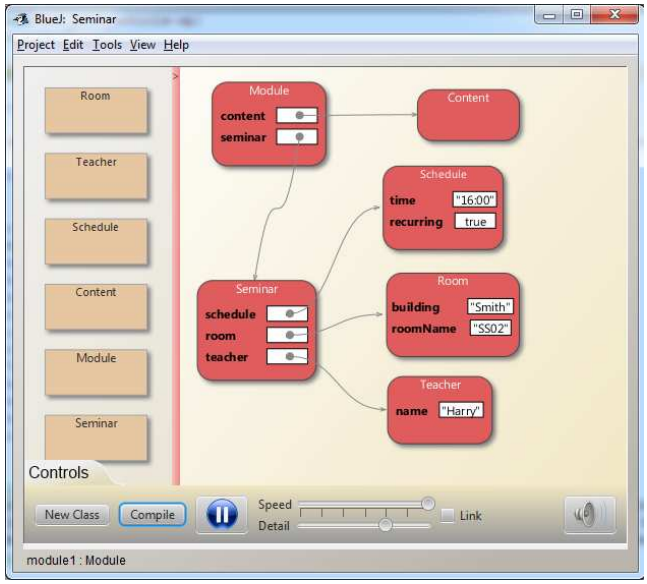

Figure 2: The Novis notional machine visualiser with a simple example.

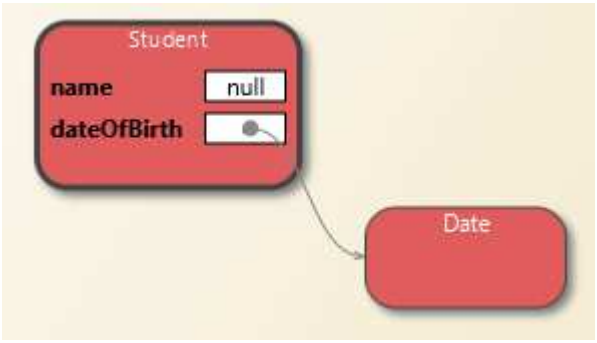

Figure 3: The expanded view of an object.

fields are updated in real-time as the object graph changes.

5.5 Interactive method calls

Methods may be called interactively on any object by right-clicking it, showing a context menu. Any methods may be executed by clicking on their name in this menu; if a method requires a parameter then a dialog will be shown (similar to that in BlueJ) for the user to enter the parameter value.

5.6 Method execution

Active method executions are visualised on the bottom right hand side of objects or classes, depending on whether they are instance or static methods. The methods are only visualised when they are part of the currently active call stack. If more than one method is executed on the same object (even if this is the same method such as when using recursion) then the method is displayed underneath the first (as on the `Student` object in Figure 4).

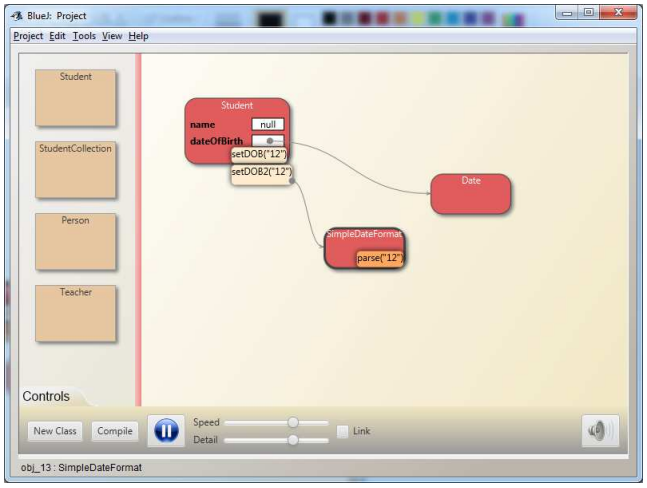

Figure 4: Novis visualising the execution of method calls.

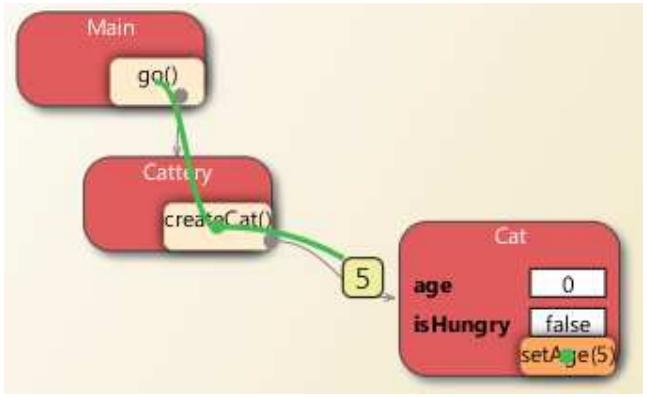

Figure 5: Novis displaying a method call chain, passing a parameter.

5.7 Call chain

The stack is often represented as a distinct visualisation from the object heap, violating Mayer's synchronization principle [10]. This states that information should not be presented to the user in two distinct diagrams; the information should be unified in a single diagram. If not, as in the traditional case, the user is then responsible for mentally synchronizing two distinct diagrams (and this places increased cognitive load on the user). Novis unifies the traditionally separate stack diagram with the object heap view, displaying a trace over the methods in currently active stack frames (Figure 5). This trace dynamically updates as the state of the stack changes. The topmost method on the stack is additionally shaded in a slightly different colour so the user can easily visualise the topmost element on the stack (this is the area where code is currently executing, so is arguably the most important method.)

5.8 Parameter passing and return values

Figure 5 also shows a visualisation of a parameter being passed. Parameters are shown in boxes that first appear

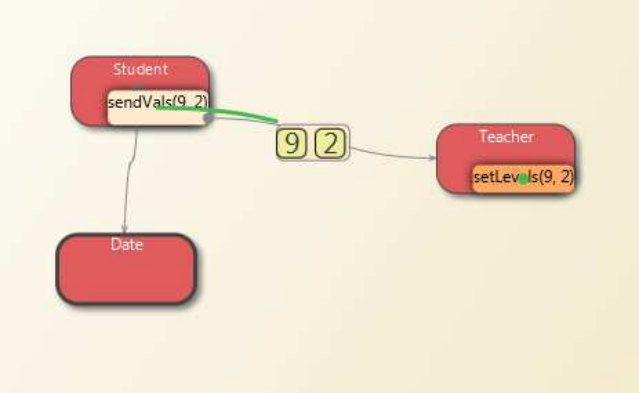

Figure 6: Novis visualising the passing of two parameters.

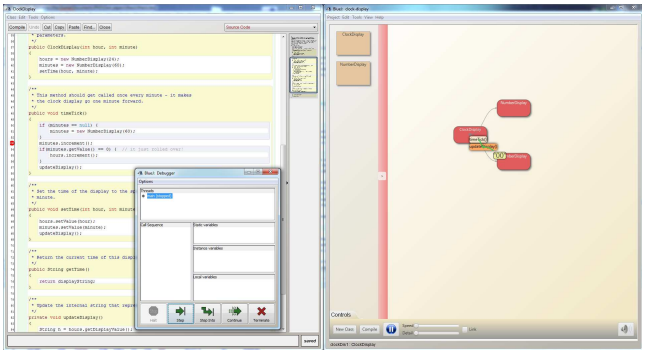

Figure 7: Novis working with the BlueJ debugger.

on the calling method, and then travel at the front of an animation of the call chain arrow until they reach the destination method. In the case of more than one parameter, the values are enclosed in a separate box and visualised in the same way (Figure 6). Return values are also visualised using this technique – the return value is displayed in a box and retracts from the method that returned the value as the call chain retracts.

5.9 Speed and stepping granularity

The speed of the visualisation should not be fixed – in some parts of a program, the user may not find the visualisation useful, in other parts they may wish to slowly step through it in great detail. It is therefore necessary to allow the user to control the speed of the visualisation, and ensure they can easily change this setting while the visualisation is running.

Novis therefore 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. 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.

There are situations where user-controlled, line by line execution is useful. The obvious scenario where this type of control may be desirable is with highly novice programmers, those who are still working with programs of only a few lines. While this high level of detail would present an overload of information for all reasonably sized programs, in this scenario this is not the case. This use case of the *comprehension of programming* remains prevalent for only a short period of time; only while the novice is solely working with very small programs. However, this can also be useful when attempting to understand the particular workings of a small section of code – that is the *comprehension of a particular program*. This use case remains viable for much longer, well beyond the first year of a computer science degree – even seasoned developers will occasionally need to step, line by line, through a particular section of code to understand the full extent of its behaviour.

Novis therefore provides a mechanism for user-controlled, line by line execution in that it integrates well with the debugger already present in BlueJ (Figure 7). A breakpoint can be set at any point in the editor, and line by line execution then visualised by pressing the "step" button. This enables the user to view objects and references being created while they step through the source, allowing them to view the visualisation on a more controlled, line by line level.

Through this mechanism a user can opt to visualise the majority of a program in rather low detail, only switching to a slow, high level of detail for a particular section of interest (whether this is for debugging or comprehension purposes.) They can of course also use it to visualise small programs in very high detail if they wish, which may be ideal for lecturers explaining very small programs at the beginning of CS1. It is important to note however that the usefulness of this functionality is not limited to this small scale, novice approach, it is useful for visualising small sections of much larger programs in detail.

5.10 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.

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, destruction and activity levels. The two sliders – speed and detail – can be linked in the user interface to allow both to be adjusted in a single interface gesture. User control over speed and animation detail ensures that our

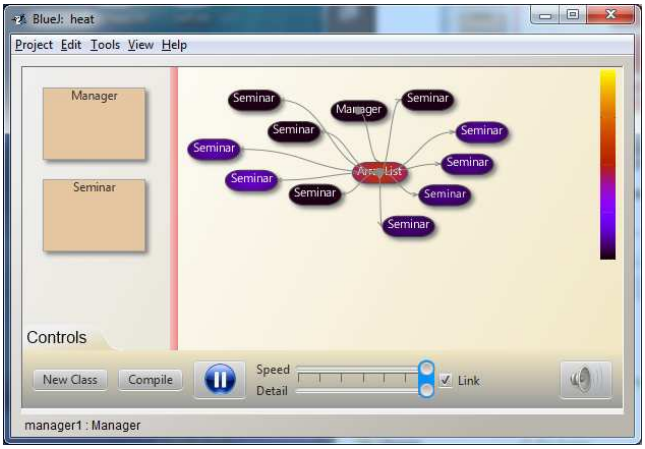

Figure 8: Heatmap view illustrating program activity.

notional machine visualisation addresses a broad range of use cases and remains relevant after the first few weeks of programming instruction.

At the lowest level of detail Novis's display turns into a heatmap (Figure 8). Here, colour is used to indicate object 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 view provides a very useful level of detail for examining an entire program, or a large section of a program. While it is deliberately vague, it allows many problems to be detected on a broad scale. Performance problems, for instance, can be highlighted at the object level – if a particular operation on an object takes longer than expected to complete it will stay "warmer" for a longer period of time. The relevant sub-section of the program can then be run using a display that provides a much greater level of detail, homing in on the exact method where the problem occurs.

6. STATUS AND FUTURE WORK

Novis is currently available for testing and evaluation purposes; it can be downloaded from http://bluej.org/novis.zip/. The system has been tested with a small number of users with promising results, but no formal study on learning impact has yet been completed.

Work in the near future will concentrate on further testing of usability and effectiveness 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.

7. 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.

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] Michael Berry and Michael Kölling. The state of play: A notional machine for learning programming. ITiCSE '14, New York, NY, USA, 2014. ACM.

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

[5] Michael Edelgaard Caspersen. *Educating Novices in The Skills of Programming*. DAIMI PhD Dissertation. Department of Computer Science, 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. *The Design of an Object-Oriented Environment and Language for Teaching*. PhD thesis, Basser Department of Computer Science, University of Sydney, 1999.

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

[10] Richard E. Mayer. Multimedia learning. volume Volume 41, pages 85–139. Academic Press, 2002.

[11] 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.

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

[13] 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.

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

[15] Ian Utting, Neil Brown, Michael Kölling, Davin McCall, and Philip Stevens. Web-scale data gathering with bluej. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 1–4, New York, NY, USA, 2012. ACM.