THE DESIGN AND IMPLEMENTATION OF A NOTIONAL MACHINE FOR TEACHING INTRODUCTORY PROGRAMMING

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF PHD.

By
Michael Berry
June 2015

Contents

List of Figures

Abstract

Comprehension of both programming and programs is a difficult task for novices to master, with many university courses that feature a programming component demonstrating significant failure and drop out rates. Many theories exist that attempt to explain why this is the case. One such theory, originally postulated by du Boulay, is that students do not understand the properties of the machine; they do not understand what they are or how they are controlling them by writing code. This idea formed the development of the notional machine, which exists solely as an abstraction of the physical machine to aid with its understanding and comprehension.

This work contributes a design for a new notional machine and a graphical notation for its representation. The notional machine is designed to work with object-oriented languages (in particular Java). It provides several novel contributions over pre-existing models – while existing similar models are generally constrained to line by line operation, the notional machine presented here can scale effectively across many program sizes, from few objects and lines to many. In addition, it is able to be used in a variety of formats (in both electronic and "unplugged" form). It also melds together three traditionally separate diagrams that had to be understood simultaneously (the stack trace, class diagram and object heap.)

"Novis", an implemented version of the notional machine, is also presented and evaluated. It is able to create automatic and animated versions of notional machine diagrams, and 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.

The evaluation of Novis is presented in two parts. It is first tested alongside a selection of methodically chosen textbook examples to ensure it can visualise a range of useful programs, and it then undergoes usability testing with a group of first year computer science students.

Acknowledgements

I would like to express my utmost appreciation and thanks to my supervisor, Michael Kölling. Your expert knowledge, advice and assistance have all been paramount in the completion of this work, and your friendliness, understanding and encouragement throughout the process is something for which I am extremely grateful. I would also like to thank my panel: Sally Fincher, Ian Utting and Simon Thompson for their advice and encouragement. You have always been on hand and open for any questions that I have had; I consider myself very fortunate to have completed my studies with such knowledgable and approachable academics.

I am fortunate to have too many friends to list here (you know who you are!), all of whom have provided a great amount of friendship and support over the last few years.. Thanks to you all for helping and encouraging me through this process, each in your own ways! I could also not have ended up where I am now without the support of my loving, caring, amazing and hard working girlfriend – Vicki, thank you so much for sticking by me, even when I have been rather irritable! You have always been there to give me the encouragement I need to keep going.

My family have always been unconditionally loving and supportive in many ways: financially, practically and emotionally. I cannot express how appreciative I am of the support you have given me, not just in my PhD but throughout my entire upbringing – something for which I will forever be grateful.

Chapter 1

Introduction

It is well understood that programming is a fundamental concept of computer science; it is the process by which conceptual ideas are mapped to instructions that can be understood and interpreted by the machine. The concept of programming, in one form or another, has existed since the first computers, and has evolved in many ways since. By implication this shows that the teaching of introductory programming within computer science is essential, whatever language may be used for this teaching, since it is through this that those new to the field will acquire these skills.

Many studies in the literature have shown that learning to program is hard (Kim and Lerch, 1997), with many students failing or dropping out of courses that teach introductory programming (Bennedsen and Caspersen, 2007). There have been many reasons proposed as to why this is the case. One popular theory is that students find the concepts of programming too hard to grasp; they do not understand the key properties of their program and how they are controlling them by writing code (du Boulay, 1986). This idea was developed by du Boulay into the concept of a notional machine, a machine that exists to provide an easy way of understanding how a particular language or program is executing (this is discussed in more detail in section 2.2). The notional machine does not necessarily accurately reflect the exact properties of the real machine; rather it works on a higher conceptual level by providing a metaphorical layer (or indeed several such layers) on top of the real machine that are easier to understand. The metaphorical layer that the notional machine provides can take on many forms. Visual metaphors are some of the most commonly used, and provide cues representative of the various events that unfold in the underlying machine. This is not the only way the metaphor can work however, it can be replaced or augmented by other forms, such as sound.

At present, it is common for educators to explain elements of object orientation, and the flow of object-oriented programs by drawing diagrams of objects and classes on a white board. However, between institutions (or even within a given faculty) there is no standard notation in use; rather it is left to the student to form a mental model based on the ad-hoc diagrams that may be used. The potential therefore exists for confusion amongst students when they are presented with inconsistent representation from a variety of sources.

BlueJ(Kölling, 1999b) is a successful IDE designed to aid novice programmers in learning Java. It is the successor to Blue(Kölling, 1999a), an earlier work which incorporated a separate language and environment. BlueJ makes some attempt to provide a mental model to students – it visualises the static relationship between classes in the main portion of its window, showing the uses and inheritance relationships between them (Figure 1). It also provides a very simplistic visualisation of objects on its "object bench" (shown at the bottom of the window).

While this makes some attempt to visualise objects, it is a very simplistic and incomplete view of the object graph. A better, more accurate version of the object bench has been a commonly requested feature of BlueJ since early in its history. Currently, it only shows objects created by the user, does not show their state and makes no attempt at a dynamic visualisation. Methods may be called on objects in the bench, but the implication these methods have on the object hierarchy is not shown, including any other objects that are created or destroyed as a result of the method call.

This work provides two key contributions that addresses the above issues. First, a universal framework in the form of a notional machine is presented. It is designed to be used by educators and students, in textbooks and in implemented form alike.

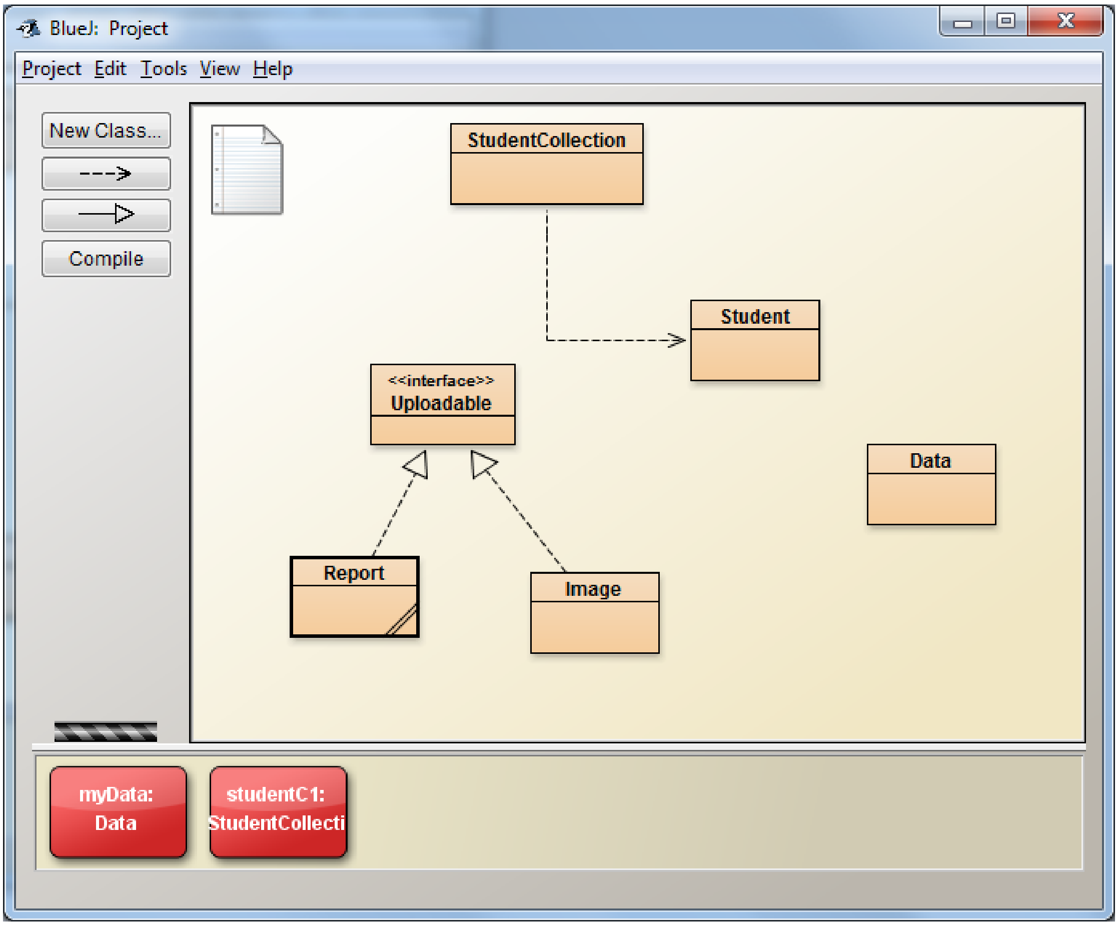


Figure 1: The BlueJ IDE.

This provides students with a consistent, correct and helpful mental model that they can refer to and use in a number of contexts and environments; replacing or augmenting any potentially invalid model which they may automatically create and subscribe to otherwise.

Second, "Novis", an implemented version of the notional machine, is also presented and evaluated. It is able to create automatic and animated versions of notional machine diagrams, and 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. In addition to providing a visualisation of the class hierarchy, Novis' visualisation also provides a much more complete version of the object graph, and allows users to examine the state of objects.

1.1 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. The diagram notation also aids in discussion of particular programs; it can be used as a common base for educators and students to discuss and reason about specific sections of code (as well as the specific states of the object graph that can arise from the code) by providing a language to express the relevant phenomenon. An implemented form of the notional machine can also provide insight and help into the workings of a particular program without the presence of an educator.

1.1.1 Contributions

From these two use cases, we present three main contributions for the notional machine.

Level of abstraction

Designing the notional machine for the above two use cases necessitates a discussion on the appropriate level of abstraction to use. This should be chosen such that the observed behaviour can be explained consistently; this then lends greater affordance to the creation of a valid mental model. This will always be the case if the notional machine is based on very low level machine code or byte code. However, this then adds a great level of complexity to even the simplest operations. Simply guaranteeing consistency is not enough; the notional machine should be on a high enough level to be useful – it should handle object-oriented concepts directly, and not necessarily show detailed information about each atomic operation that occurs in the underlying machine. Its task is to explain the observed behaviour, and to do this in a way that does not provide an overload of information to the user. The challenge is providing a higher level abstraction of these concepts that still forms a valid and consistent model.

Universal notation

The notation for the notional machine should be designed so it is useful both on-screen and off-screen. Many materials and mediums are used for teaching, and as such the notional machine should be able to be used effectively in a variety of off-screen settings. Lecturers should be able to discuss and reason about programs on a whiteboard for instance, allowing some limited form of animation (that which is reasonably possible by quickly drawing and erasing.) It should also be useful where even this limited “animation” is not possible – it should be able to be used in textbooks as a standardised notation, again for discussing and reasoning about object-oriented programs.

The notation should also be standardised as far as possible, allowing for consistency and interoperability across publications and institutions. Ad-hoc diagrams are often used by educators and authors, but these are in no way standardised across institutions. Indeed, such diagrams are often based on the personal choice of the educator and thus not even standardised within the same institution, adding a further source of confusion. The notation is presented in detail in Chapter 4.

Animated implementation

As mentioned above, users should be able to use the notional machine in a multitude of settings – this should include the aid of a software tool where available. The tool should allow the user to interact with a running program in real time, examining the object graph (or part of it) in detail should they wish. It should be capable of displaying the changing state of the object graph itself as well as the state of the individual objects it contains. However, it should also not be obtrusive – the user should not be forced to step through an executing program if they feel the visualisation does not provide a benefit, and they should have some form of control over the speed of the visualisation as well as the elements that are displayed.

The user should be capable of executing a program they have written (rather than just preset examples) and must be able to observe the development of the object graph: this should include the creation and destruction of objects, method calls, references being passed and fields updating. The tool should be capable of providing this visualisation both incrementally, as well as in a smooth animation, depending on the user’s preference.

Novis, an animated visualisation of the notional machine, has been designed and built as part of this work. It is built on the BlueJ IDE and discussed in detail in

Chapter 5.

From the above colloquial introduction of the requirements, we now formulate three specific aims of the notional machine.

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

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 Research Question 1, 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.

For the purpose of Research Question 2, we outline our expectation of a *typical beginner's program* in Chapter 3. In brief, we aim to cover programs that would be expected to fall within the first year of an undergraduate computer science course. Research Question 2 also necessitates a more concrete definition of the terms "manageable" and "useful". We define "manageable" as meaning the tool can be used effectively to visualise a program by its target audience (first year computer science students), and “useful” to mean that the tool can effectively represent a variety of first year textbook examples and provide information to a user that is not obvious from the source code.

Determining whether the software tool is manageable and useful will be covered in later sections, by evaluating it firstly against a corpus of textbook examples (Chapter 6), and secondly with first year undergraduate students (Chapter 7.)

The first research question can further be divided into three sub-questions:

Research question 1a : Can the object heap and stack trace be effectively represented in the same, unified diagram? Can this diagram provide a snapshot of the state of a program, as well as depicting continuous execution?

Research question 1b : Is the notation used for the notional machine complete? Can every observable behaviour within the target problem space be explained in the notional machine? Following on from this, can the notional machine also be used to predict the observed behaviour?

Research question 1c : Is the notional machine consistent with observable behaviour?

The preceding sub-questions necessitate some further definitions. Here, by *effectively*, we mean that it should be usable by the target audience (first year computer science students). The usability of the tool is later examined in Chapter 7. By *unified*, we mean that both the stack and the heap should be represented in the same semantic diagram – the elements from both should interact, as opposed to a separate depiction of these views.

We also make some further assumptions:

- We assume that the target audience are using Java as their primary programming language throughout CS1. The notional machine is therefore designed to visually represent object-oriented concepts that would often be taught on an objects-early CS1 course.

- We assume that students will use it to investigate Java programs that they might typically encounter throughout CS1. The notional machine should therefore display information relevant in this regard – it should represent what is traditionally the heap and the stack, illustrate object interactions in a way that is useful, and offer explanation about any observable relevant behaviour of the underlying program.

In relation to research question 1a, the *static* aspects of the program are defined as the current objects in scope and the references between them – the information typically used to form a standard object graph. In contrast, the *dynamic* aspects are defined as the call chain of events, the causal state of the program.

From these research questions, an overall hypothesis is formed for the expected outcome of this work:

Hypothesis 1 : A notional machine as described above can be effectively used by first year computer science students.

This hypothesis is explored in depth throughout this thesis. After a thorough review of the relevant literature in Chapter 2, Chapter 3 explores the problem space for this work – that is, the subset of programming concepts within Java that the notation is expected to clearly and consistently display. Chapter 4 explores and details the notation used in the notional machine. This makes a number of key contributions, introducing a scalable, diagrammatic visualisation that unifies the traditionally separate views of the stack trace and object heap.

Chapter 5 describes Novis, the implemented form of the notional machine. This is based on BlueJ, replacing the traditional class view and object bench with a single pane that represents both; it automatically forms, places and animates a visualisation that consistently reflects the state of a program. Some design decisions specific to an on-screen, animated format of the notation are also presented in this chapter. The following two chapters serve to provide an evaluation of this work; Chapter 6 uses a methodically chosen set of textbook examples to evaluate Novis, examining the usefulness of the generated diagrams and the quality of the layout. Finally, Chapter 7 presents Novis to a number of first year computer science students in order to ascertain its usability.

A second hypothesis is also defined for future work:

Hypothesis 2 : The formation and use of a notional machine as described above helps students to learn about object-oriented programming more effectively.

By "more effectively" in this sense, the hope is a tangible improvement of students' understanding of programming should be observed over traditional techniques by the incorporation of the proposed model into the teaching of object-oriented programming. However, it should be noted that this hypothesis is not one that can be tested as part of this work. Proving a long term, longitudinal learning effect would require an extensive study over a long period of time. This is beyond the scope of this project.

Chapter 2

Literature Review

2.1 Introduction

In this chapter we will discuss two major categories of research. The first is in relation to conceptual models, both in terms of existing mental models and the theory behind them. The second covers existing tools that visualise the execution of programs – implementations of conceptual models.

2.2 The concept of a Notional Machine

du Boulay outlines five key areas of difficulty in learning to program (du Boulay, 1986). Although a significant amount of research has been published in this area since this paper (Ford, 1993; Sajaniemi et al., 2008; Sorva, 2013), these fundamental difficulties are still relevant in the teaching of programming today.

Orientation This involves finding out why programming is useful, the sort of tasks it can solve and the eventual overall advantage in taking time to learn how to program.

Pragmatics This area is the practical aspect of programming – learning how to implement the development cycle (specification, development, testing, debugging) using the available tool chain for instance. This area also covers the concept of *structures*, which in this case means acquiring a knowledge of small segments of code that can be used as the common building blocks of larger applications.

Structures & Idioms These are small scale plans or cliches that "can be used to achieve small-scale goals, such as computing a sum using a loop" (du Boulay, 1986).

Notation One key area that students often struggle with in the early stage of learning to program is mastering the syntax and the semantics of the language being taught; the *notation* of the language.

Notional Machine The notional machine is defined as "general properties of the machine one is learning to control", and a key difficulty is mapping the properties of the real machine onto this notional machine. It has been shown much more recently that many students in the early stage of learning to program find this challenging (Lister et al., 2004), so it does not appear the concept's difficulty has faded with time. It is this concept that is most relevant to this work.

While the notional machine is the prominent area of research in this thesis, the other concepts outlined in the same paper are also relevant in part, and du Boulay clearly mentions there is a definite, deliberate overlap between these five areas. A notional machine for instance should also help somewhat with structures, if a student becomes accustomed to a particular structure within the notional machine, they may find it easier to grasp the structure in code. Similarly, a notional machine may shed some light on the underlying semantics of the language.

2.3 Conceptual models

A study by Ma et al (Ma et al., 2007) investigated how different conceptual models were used and held by students in an attempt to understand the notional machine that they were presented with. They found that novice students often think in terms of a given conceptual model (or mental model), one they create to explain the behaviour that they observe. This conceptual model may be a good fit for the actual behaviour, but more likely it is a rough approximation that will break down at some point, and be replaced or augmented with another conceptual model. The same study investigated the viabilities of such models held by novice programmers. Specifically, they examined in detail the mental models held by students of value and reference assignment. The study showed that by the end of the first year programming course, only 17% of the users being studied held a viable mental model for reference assignment, and in addition to this there was a strong correlation between those users who held such a model and passed the exam successfully.

While this study mainly focused on the mental models around value and reference assignment, the results also highlighted other common misconceptions that are arguably more relevant for this work. Four of the participants in the study believed that "an object would be created automatically when a reference variable was declared", which the authors state as a potential cause of forgetting to provide a constructor call and thus receiving a `NullPointerException` later in execution. Correcting such a faulty mental model could help to alleviate this behaviour. Additionally, three of the participants believed that an object was actually a subset of a class, which could be a potential source of confusion when polymorphism is introduced as a concept.

The authors argue that traditional approaches do not do enough to counter these faulty models. They believe that new approaches, integrated with visualisation, should be used to challenge and repair the faulty models, and help students generate and understand more viable mental models. The importance of mental models is well established in literature, with many authors stating their efficacy (Ramalingam et al., 2004; Cañas et al., 1994; Wiedenbeck et al., 1999).

2.4 Metaphors

The metaphorical aspect is one that ties in closely with the idea of the notional machine and the conceptual model; it could sensibly be argued that a mental model is actually a collection of collaborative metaphors. du Boulay discusses many of these within the concept of a notional machine, including many common examples of metaphors that can reinforce unhelpful misconceptions – "misapplication of analogy." A variable is commonly referred to as a box for instance, holding a single value. However, this can be confusing to novice programmers because their existing idea of a box does not quite fit with this view. A box in a physical sense can hold more than one item for instance, and could sensibly have some mechanism to retrieve a set of previous values. A variable does not have these properties, which makes it an inappropriate analogy for this concept. du Boulay instead suggests the idea of a variable being a piece of slate, where a value can be written on it, and a new value can be written on it over the top of the old one, overwriting it.

However, while the idea of a piece of slate is arguably better than a box for the previously stated reasons, it does have problems of its own – it could highlight the concept of misapplication of analogy in other ways. You could argue, for instance, that a slate could hold more than one variable also, if the variable was small – a single digit number for instance. You could also assume that you need to clear the variable before writing over it otherwise you corrupt the data, as would be the case if you overlaid two numbers on a piece of slate.

Both the above concepts fit into the idea of familiarising metaphors, metaphors that create a notional mapping between ideas and concepts commonly encountered in the real world and a concept in the real machine (between a box or piece of slate and a variable for example.) These sorts of metaphors are commonly used because they provide an immediate real world concept which students can latch onto, and therefore hopefully gain some understanding of the concept underlying the metaphor. However, as already seen they can be confusing – it is very hard, if not impossible, to find a common real world element that maps well onto a completely separate concept. In most cases, while various desirable characteristics may be explained by the metaphor, if pushed too far the metaphor will almost always break down, reinforcing misconceptions such as those already discussed.

It has been widely suggested outside of Boulay's work that good[1] metaphors can be helpful for understanding programming concepts, but developing these types of metaphors is non-trivial and an ongoing research issue (Roman and Cunningham, 1990; Travers, 1996; Baldwin and Kuljis, 2000).

2.5 Cognitive Dimensions

Green and Petre define 14 cognitive dimensions of notations (Green and Petre, 1996). These are design principles that can be applied to user interfaces in general, including both visualisations and visual programming languages. The dimensions provide a mechanism for lightweight design analysis, as well as providing a common set of principles to aid discussion of similar systems. In the following section we examine each of these individual dimensions, stating their function and summarising their relevance to this work in section 2.5.15.

2.5.1 Abstraction gradient

The level of abstraction is examined here, with this dimension examining both the maximum and minimum levels of abstraction that can feasibly be used by the system.

2.5.2 Closeness of mapping

This dimension defines how closely the notation (or notional machine, in this work) corresponds to the underlying machine.

[1]In this context, by good metaphors, we mean those with potential for minimal misapplied analogies.

2.5.3 Consistency

After part of the notation has been learned, the consistency is defined by how easily the remainder can be guessed.

2.5.4 Diffuseness

This dimension examines both the physical space and the number of distinct symbols required to visualise a certain state.

2.5.5 Error proneness

This dimension examines the likelihood of the notation influencing the quantity of mistakes made by the user.

2.5.6 Hard mental operations

This dimension examines the balance between the mental processing at the notational level and the semantic level. Specifically, the user should not need to resort to addition artifacts (such as pointing with fingers or making notes) to keep track of operations occurring in the notional machine.

2.5.7 Hidden dependencies

This dimension examines the dependencies between separate entities within the notional machine – the dependencies should be visible, so that changes in the state that occur due to dependencies can be predicted by the user.

2.5.8 Juxtaposability

This dimension examines whether differing parts of the notation can be sensibly compared in a side by side fashion.

2.5.9 Premature commitment

This dimension examines any constraints within the notation on the order of tasks. It also examines the ability to correct any decisions which had to be made by the user before all necessary information was available.

2.5.10 Progressive evaluation

This dimension examines a program in an incomplete or incorrect state. The notation should allow a reasonable evaluation of this state to provide the user with useful feedback.

2.5.11 Role-expressiveness

This dimension describes the role of each component used in the notation; specifically, how obvious the role is of any individual component in the system.

2.5.12 Secondary notation

This dimension examines any extra information that may be depicted through a non-syntactical method, such as colour, layout or sound.

2.5.13 Viscosity

This area examines the difficulty in making a change to the notation. It is divided into three distinct types of viscosity:

Knock-on viscosity This is defined as a change that causes other internal constraints to be violated. Fixing these constraints may themselves cause further constraints to be violated in a recursive fashion.

Repetition viscosity This is defined as a single action within the mental model of the user that requires many repeated actions in the underlying notation.

Scope viscosity This is defined as a change in the size of data requiring a change to the semantic structure of the underlying program.

2.5.14 Visibility

The visibility dimension determines how parts of the notation can easily be identified and accessed when required.

2.5.15 Summary

The above cognitive dimensions are relevant to this work; primarily, they can be applied to the design of the notional machine (and the notional machine could be analysed in relation to these dimensions.) However, the dimensions are also relevant in an implementation based context; the implementation of the notional machine can therefore also be analysed in the same way.

2.6 Principles of Multimedia Learning

The concept of learning through visualisations is not one constrained to computing. While a limited amount of literature is available on multimedia learning specifically within this area, a much wider amount of material is available in a broader context. Mayer outlines several principles that are relevant in this general area (Mayer, 2002). Specifically, these principles are relevant to the implementation of the notional machine described in Chapter 5.

2.6.1 Interactivity principle

The first principle outlined is the interactivity principle, which states that to maximise learning outcome the implementation should be kept interactive (Mayer, 2002). This idea is supported by many others in the field, particularly within the scope of algorithmic visualisations. Hundhausen analysed 21 separate studies in one meta-study on the subject (Hundhausen et al., 2002). The meta study grouped the contained studies into two categories; those which involved the students interactively, and those which did not. Only 33% of studies in the non-interactive group claimed effective results, but in the interactive group this figure rose to 83%. Furthermore, one reference to this study (Naps, 2005) quotes two additional effective studies that were both interactive, if added to the previous meta-study, would boost the final figure to 86%. The use of interaction can take several forms – some tools or visualisations may choose to simply provide a button that has to be clicked to advance the visualisation, others, such as JHAVE (Naps, 2005) may require the user to answer a question correctly before they progress.

2.6.2 Apprehension principle

The apprehension principle in essence states that the animation should be kept simple – objects should be represented clearly in a way that is immediately understandable where possible, and anything that is not useful for the learning outcome should not be presented (Mayer, 2002). In particular, no feature that complicates the animation should be present purely for aesthetic purposes, since it distracts the user's attention from the message the visualisation is trying to convey. Jeliot's curtain animation is an example of something which breaks this rule; it takes time, is distracting and is purely aesthetic.

2.6.3 Congruence principle

The apprehension principle is augmented somewhat by the congruence principle which states that changes in the real machine do not need to correspond exactly to what is shown in the conceptual model (Mayer, 2002). In other words, the real machine does not need to be represented exactly if simplifying the inner workings can help understanding. This principle also states that realism is not strictly speaking necessary, which reflects the goal of a conceptual model – it is an abstraction layer above how things are actually executed and as such can distort reality in order to simplify understanding of the actual machine. Following that principle, a notional machine can and arguably should simplify steps that occur in the actual program so as not to confuse the student, and potentially hide some aspects of its behaviour altogether so that the student is not initially overwhelmed by all that is occurring.

2.6.4 Attention guiding principle

It has been shown that when faced with a visualisation, students focus on the more prominent parts, and potentially more relevant parts that do not feature so prominently tend to be ignored (Lowe, 2003). This leads to the attention guiding principle which states that the user should be guided, artificially if necessary, towards the more important parts (those that portray the key steps) with the use of animation, verbal commentaries or graphical elements such as arrows or highlighting (Mayer, 2002). These elements serve to direct the user's attention specifically towards the part of the visualisation that they should be viewing at the present time. The converse of this, and just as important, is that attention should not be misguided by poor use of these elements. For example, an ongoing animation in a "dormant" area of the visualisation would only serve to direct the user's attention to that point, potentially causing them to miss something else happening elsewhere that is essential for them to understand the concepts being conveyed.

2.7 Types of cognitive load

There are three distinct types of cognitive load defined in cognitive load theory (Sweller, 2010); they are discussed here.

2.7.1 Intrinsic

Intrinsic cognitive load is caused by the inherent difficulty of the task at hand, and this cannot be altered (Chandler and Sweller, 1991). While there will therefore, by definition, be an intrinsic cognitive load with any given task, this is not one that can be affected in any way by the design of the notional machine.

2.7.2 Extraneous

Extraneous cognitive load is caused by the vehicle used to present information to the learner; it is thus directly affected by any tool, notation or explanation that may be used (Chandler and Sweller, 1991). Where there is more than one possible method of explanation, these methods will likely require differing amounts of extraneous cognitive load. For example, explaining the parts to an engine could either be done through the use of diagrams, or purely verbally (as well as other means.) In this case, it might be possible for students to understand a verbal explanation, but a diagram would likely be much simpler and require much less extraneous cognitive load.

For this work, this is relevant since it involves choosing a method of notation that ideally requires minimum extraneous cognitive load. The notation chosen should be one that users can understand simply; they should in particular not have to learn or memorise additional concepts before understanding the notional machine notation. In this sense, extraneous cognitive load is closely related to the cognitive dimension of "Hard mental operations" (section 2.5.6).

2.7.3 Germane

Germane cognitive load is a type of cognitive load that is either augmented or diminished by the use of schemas – that is, the use of any organised, recurring pattern (Sweller et al., 1998). In contrast to intrinsic and extraneous cognitive load, it is recommended that this type of cognitive load is increased (at the expense of increasing extraneous load.)

In this work, increasing Germane cognitive load will involve the use of organised, recurring patterns – the notation should be predictable, where possible using the same (or similar) notation throughout the notional machine. This should allow users to construct and automate schemas, affording a positive learning effect.

2.8 Principles of reducing cognitive overload

In addition to the four general principles of multimedia learning, Mayer also outlines a further nine principles based on the overarching principle of eliminating cognitive overload (Mayer and Moreno, 2003). These are based on the learning theory that "humans possess separate systems for processing pictorial and verbal material ... each channel is limited in the amount of material that can be processed at one time" (Mayer and Moreno, 2003). The following are described as techniques that can be used to reduce the chance of cognitive overload, where too much information is presented (either in one particular channel or overall) for the learner to comprehend.

2.8.1 Off-loading

Off-loading states that some essential information should be presented as sound, rather than visuals. A typical example is that of narration, it should be presented as auditory data rather than captions on screen. This supports the notion of utilising sound with the visualisation rather than just presenting purely visual data to the user.

2.8.2 Segmenting

Segmenting is used to avoid the situation where both processing channels are overloaded with information. Instead of a continuous block of information, it should be split up into *learner-controlled* segments. The interesting point for this work is the "learner-controlled" part, stating that the learner should decide the length of these blocks, they should be able to stop when they feel they have been presented with enough information. This argument lends strong support to a greater degree of interactivity, allowing the user to control the pace and scale of the visualisation presented to them.

2.8.3 Pretraining

Pretraining states that the transfer of information occurs more effectively if the student is already familiar with the components used in the visualisation. This shows that for such a tool to be used most successfully the key principles should first explained to the student before they start using it. Such an explanation could also be incorporated into the tool itself, via an automated tutorial (or similar system.)

2.8.4 Weeding

Additional material not directly related to the learning task may be interesting, but is extraneous and should be removed. This will ensure neither cognitive channel is overloaded by extraneous information that could detract from the learning of essential material.

2.8.5 Signaling

This principle states that cues should be provided in some form to direct a learners attention to the most relevant parts of information. This could include highlighting an important part of a diagram, using animation to guide their attention or changing the tone of voice to emphasise a particular element in the narration. This complements the attention guiding principle discussed earlier, and for this work the use of animation and emphasis to draw a user's attention will likely be the most relevant aspect of this principle.

2.8.6 Aligning

Any printed words should be placed close to, or on the graphics in the visualisation which they are describing. This reduces the need for visual scanning and contributes to the "spacial contiguity effect", where better understanding occurs when textual information is displayed alongside the corresponding graphical representation.

2.8.7 Eliminating redundancy

The same exact information should not be needlessly duplicated across cognitive channels. The prime example of this is on screen captions and the equivalent narration, where all captions are narrated regardless of their importance. However, reinforcing *essential* information in multiple channels can be a useful technique, as long as this is not overused. This technique corresponds to the attention guiding principle.

2.8.8 Synchronizing

The related information presented across multiple channels should be presented concurrently. This avoids the need for the learner to hold the information from one channel in memory before rectifying it later with separate information conveying the same concept.

2.8.9 Individualizing

The concept of individualizing is mostly a fall-back when synchronizing is not possible; it states that in this case it should be ensured that the learners in question are skilled at remembering mental representations. Since this cannot (and arguably should not) be ensured in this work, it is mainly a principle of theory and the principle of synchronization should be followed in order to account for this.

2.9 Issues with static models

One existing object-oriented model is based around the metaphor of a filing cabinet (Gries, 2008). The classes are represented as the individual drawers in a filing cabinet, these drawers may be filled with folders. Each folder in the drawer then corresponds to a particular object, a particular instance of the "draw" in which it resides.

The folders contain features which represent the attributes of objects – a unique name, its type, its fields and methods. One such folder is represented in Figure 2 (Gries, 2008). At the top of the folder is a particular unique identifier; this serves as an index by which the chosen object can be retrieved (by searching through the relevant draw.) The type of the object, matching both the corresponding class and the draw in which it resides, is also drawn towards the top right of the folder. Both fields and methods are drawn on the body of the folder itself. The fields include the field name, type and value of that particular field – `name`, `String` and `null` respectively in this case. The methods include the method name and the types of any parameters the methods take, and are differentiated from fields by the use of parentheses after their name.

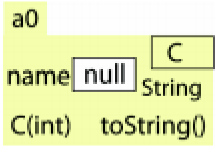


Figure 2: An example of a folder representing an object in the filing cabinet model.

A static class diagram is also available in BlueJ (Kölling et al., 2003) which represents the uses between classes as well as the inheritance hierarchy between them via blocks on screen connected by different styles of arrows (Figure 3). This can be a useful aid for students understanding how their program links together from a static perspective, but this model does not attempt to provide a dynamic view of objects and the relationships between them at runtime. Both of these models can be categorised as snapshot based, programmatic models (although it can be argued that the filing cabinet model has some continuous elements.)

Whilst static visualisations are helpful in understanding the structure of a program, and perhaps implying some form of loosely defined behaviour at runtime, they cannot help in visualising the operations a program performs as it runs dynamically. One study found that snapshot visualisations are helpful to novice programmers initially, but as they progress misconceptions can arise and students make incorrect assumptions about the conclusions they can draw from the visualisation. Specifically they can become confused with regards to the order in which things are run (Ragonis and Ben-Ari, 2005). This is an obvious limitation of a static diagram (by design you cannot show information as it occurs at runtime) but it alludes to the fact that some form of dynamic visualisation has the potential to be useful in conjunction with this approach. Indeed, the authors specifically recommend that “the BlueJ learning environment that we used should be augmented with dynamic visualisation so that students can coordinate the static and dynamic aspects of object-oriented programs.” While static models are therefore useful, they cannot provide a full and complete mental model for a particular program, and in particular do not provide a model for program execution.

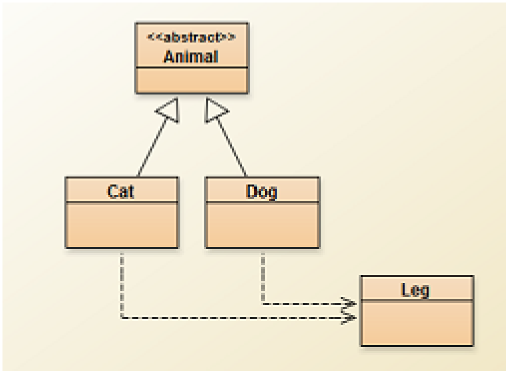


Figure 3: A static class diagram generated by the BlueJ IDE.

2.10 Algorithm visualisation

Since we have established that static models are often insufficient, it seems sensible to focus on dynamic models instead, and where applicable their corresponding implementations. The largest current area of such dynamic models within an educational context is that of algorithm visualisation, with many articles written on the subject (AlgoVis.org, 2012). They are often viewed very positively by both students (Stasko et al., 2001) and teachers (Naps et al., 2002). However, despite the popular use of these tools there is continuing debate in the literature as to whether such visualisations have any positive effect on the learning outcome of students, with many studies finding little or no improvement (Stasko et al., 1993a; Byrne et al., 1996; Gurka and Citrin, 1996). Algorithm visualisation were used as learning aids before these studies (Naps, 1990), but the first study that empirically evaluated the effectiveness of such visualisations was conducted by Stasko in 1993 (Stasko et al., 1993a). The study found that algorithmic visualisations did not aid students' learning. Later studies have supported this view, finding mixed results at best (Byrne et al., 1996; Gurka and Citrin, 1996).

These findings have prompted other studies to investigate elements that may help to increase the effectiveness of such visualisations. One study in particular found that providing the student with the ability to control the pace of the animation provided a much more effective learning outcome, as did a good data set and the ability to break down and show the steps in the algorithm logically (Saraiya et al., 2004). Another study showed that providing a level of interactivity also increased effectiveness (Stasko et al., 1993a).

2.11 Categorisations of existing systems

Conceptual models and visualisations within the context of programming are covered extensively in the literature, and can be grouped into a number of categories based on their form and function.

We explicitly exclude visual programming tools from this review. These are not tools that create a visualisation from a particular program or algorithm. Instead, they are programming environments in their own right, but where the program is created visually rather than compiled from text. This is often by means of dragging and dropping particular "blocks" of code that perform various functions. Scratch and the Lego Mindstorms environment are two particular examples of visual programming tools.

2.11.1 Granularity

The problem of scale and granularity arises because of the overwhelming potential difference in the size and complexity of different programs. A five line student program clearly needs a different approach than a 50,000 line enterprise level program for instance. An approach that works well with one is unlikely to be useful for the other. One simple, yet entirely valid approach is to restrict the domain to which a particular approach is suitable – this is the approach many tools take by only aiming to deal with student sized programs. Even within this domain, however, there exists a lot of variation. At the beginning of a programming course, students may be writing programs no more than 10 lines long, after a few months this could feasibly rise to hundreds of lines – and the same approach is not necessarily helpful across these two classes of examples.

2.11.2 Interactivity

While some visualisations are only viewable, other interactive systems allow some degree of interaction between the user and the implemented notional machine. Again, Uuhistle (Figure 4) (Sorva and Sirkiä, 2010) and Jeliot (Myller et al., 2007) are examples of interactive visualisations. The level of interaction can of course vary greatly; it can be as simple as controlling the pace of the visualisation or may be much more in depth. In a live visualisation that offers a greater level of interaction, the user may be permitted to manipulate the underlying machine's state.

2.11.3 Visualisation type

There are three main criteria which should be adhered to for a work to fall under the category of a visualisation (Kosara, 2007). Firstly, it must be based on a non-visual process or data; it must transform data that is not already represented in graphical form. Secondly, it must produce visuals that are the primary means of communication. It can be augmented with other elements such as sound, but this must act to supplement the visuals that are produced. Finally, the end result must be readable; it must provide a way to learn something about the data or process being visualised.

While the above defines the scope for any generic visualisation, it is important to note that a *good* visualisation in this context should fulfil other criteria. Firstly, it must be clear. This is a hard term to define precisely, and is somewhat subjective, but we will revisit this later. While a visualisation in general must be readable, a good one should strive to present the relevant data in a format that is easily understood by the target audience. It should also be correct – while it may be necessary or beneficial to simplify the underlying system, the fundamental, most important principles must be conveyed correctly. It should also be helpful, i.e. the information that the visualisation conveys should have a tangible benefit on the target audience.

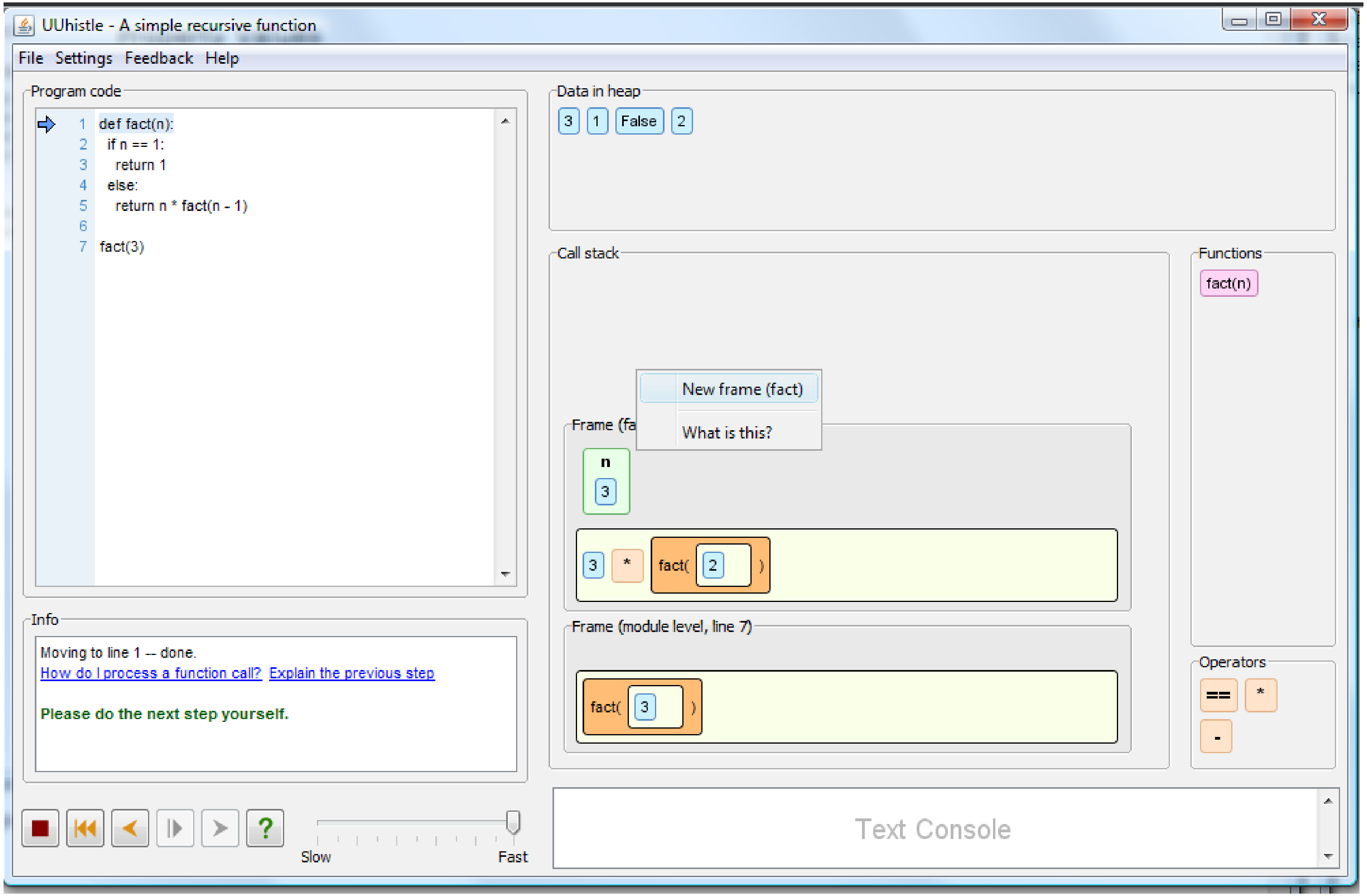


Figure 4: UUhistle's interface.

Algorithm visualisations / Program visualisations

Algorithmic visualisations are defined as those that visualise particular algorithms, with different sorting algorithms being the most common subcategory. They are often hard coded to visualise a particular type of algorithm in detail, and usually are not very adaptable beyond the algorithm (or set of algorithms) which they were initially designed to work with. In contrast, program visualisation systems are designed to work with arbitrary pieces of code, often written by the user, and represent the details of the underlying machine as this code runs. The user can then observe the visualisation as their code runs, perhaps using this information to form a conceptual model. In a particular case of a sorting algorithm for instance, a typical algorithmic visualisation might show each element in the list as a separate visual entity, it may adjust its position in the list many times over the course of the visualisation until the sorting is complete. A program visualisation however might show the representation of the classes and objects in the code and their changing relationships as the execution of the particular implementation of the underlying algorithm runs.

2.11.4 Snapshot / Continuous model

We define snapshot models as those that do not change - they are a single depiction that does not perform any animation. These are often seen as images on websites explaining a particular concept. They are arguably simpler to understand for simple examples since they can be studied without the aid of a separate tool. They can likewise be studied on paper as well as electronically. However, for more complex examples they may be harder to understand, as all the information for the entire execution must be presented on one static diagram.

In contrast, continuous models are defined as those that work with an algorithm or program as it runs and are thus capable of representing the changes and transitions in state, as well as potentially allowing the user to pause the animation and examine each state within the execution from a static perspective. They are therefore more flexible than snapshot models but also generally more complicated, both in their use and implementation.

2.11.5 Post-mortem / Live visualisations

Dynamic visualisations can be further split into two categories, post-mortem or live visualisations. As the name suggests, a post-mortem visualisation is defined as one produced from the history of a completed execution; the program being visualised has already finished executing. In contrast, a live visualisation unfolds alongside execution.

Post mortem visualisations allow certain functionality to be accomplished with greater ease – reverse execution is easy to implement for instance. The layout of the visualisation is also much easier to calculate, since all the details concerning the type quantity and lifespan of objects can be factored into any layout calculations. Post-mortem visualisations also posses a potential advantage from a performance perspective, since the visualisation and program are not being run simultaneously, and calculations needed for the visualisation can be computed before the visualisation starts.

In contrast, a live visualisation runs alongside the program it is modelling, observing changes in its state and mapping them onto itself. This gives more flexibility for changing the course of the program – it becomes technically feasible, although non-trivial, to change the state of the program as it is running by modifying the state of its objects. Tasks such as reverse execution however, that are comparatively easy to implement whilst taking a post-mortem approach, present a number of problems when using live visualisations. While it is possible to mimic the reversal of tasks such as assigning a field or creating a new object, others are impossible to treat in this way. If a page is sent to the printer for instance that action cannot be recalled and changed. Likewise, layout calculations with a live visualisation become challenging. On placing the first object, for example, the machine will not know whether that is the sole object that the program will create, or if there are millions more to follow in a short space of time.

2.11.6 3D visualisations

Within the educational domain, the vast majority of visualisation tools produce a two-dimensional visualisation. However, three-dimensional visualisations do exist and are used extensively in a number of other contexts (Greevy et al., 2006; Fronk et al., 2006; Brown and Najork, 1993). All the works covered in this review are two-dimensional unless otherwise stated.

The same article has suggested that the key to successful learning lies in organisation, representation and structuring of knowledge (Baldwin and Kuljis, 2000). The authors believe that a visualisation should tackle these issues head on with a novel approach rather than simply replacing existing text with visual components.

2.12 Implementation of existing systems

We will now discuss existing visualisation tools relevant to this work. The research in section 2.11 alludes to several important aspects that should be taken into consideration when constructing a mental model and subsequently providing an implementation. In this section, existing visualisation systems will be examined in the context of these aspects. The tools considered here are mostly program visualisation tools, not algorithm visualisation tools. It is mainly the former that are relevant for this work, since the implementation of the chosen mental model will likely fit into this category.

2.12.1 Granularity

Uuhistle

UUhistle allows students to step through programs in a great level of detail, on a sub-statement based level. When students are writing small programs this is useful, because the information presented is detailed without overloading the user. However, as the student moves on and understands the basic concepts that Uuhistle visualises, the same style of visualisation would not have as much use. The view provided is on too low a conceptual level to provide an overview of the structure of any larger program, and the number of steps required to execute it would quickly prove cumbersome. For example, Listing 2.1 shows a simple arithmetic example provided with UUhistle:

Listing 2.1: UUhistle's 5 line arithmetic example

```
a = 5
b = 2
result = a + b + 10
result = a + 10 * b + (a - 2) * a
print result
```

This particular 5 line example takes around 38 separate steps from within UUhistle to fully execute, since by default it breaks each line down into its component steps – multiple steps for each evaluation, then assignment, and so on. Each step takes time, it is animated in detail. When a student needs to understand the basics behind how assignments and evaluations are executed, this is potentially a benefit, since a lot of detail is available on each atomic step. However, while it is possible to skip to a particular point in the program if you do not want or need to visualise everything, it is clear from this that visualising larger programs of around a hundred lines in their entirety is impractical using this approach.

Jeliot

Similar behaviour can be observed with Jeliot – while it works with Java programs rather than Python programs, the underlying issues with scale are similar in that once programs move beyond those of a few lines it becomes meaningless to visualise them in their entirety. Breakpoints or code snippets have to be used to visualise a particular part in detail. While this level of detail is good for comprehending the details of small programs of a few lines, it potentially falls down when the programs become too large to feasibly animate every atomic operation. In correlation with previously mentioned studies (Hundhausen et al., 2002), the lack of interaction means students would likely become demotivated looking at the same form of visualisation time after time and would focus on the low level semantics of the language rather than a higher level view of how the program is running. This relates to the understanding of programming vs. the understanding of individual programs – such low level tools are good for understanding simple programming concepts, but not for understanding generic programs.

JIVE

There are few tools that seem to provide multiple levels of granularity, however a particularly interesting example in this regard is JIVE (Figure 5, Czyz 2008). JIVE is a dynamic, interactive and post-mortem visualisation system which works with Java programs (Gestwicki and Jayaraman, 2004), implemented as an Eclipse plugin. It supports multiple granularities of visualisation – different conceptual levels of the notional machine that the user can switch to. JIVE also supports simulated reverse execution so users can step backwards as well as forwards through their program (this is implemented by saving the visualisation's history so it can be examined.) The visualisation is interactive. Users can select objects in the visualisation to view more detail, and can pause, resume and control the speed of the animation.

JIVE provides three main levels of granularity over the objects it visualises – detailed, compact and minimised views. In a detailed view, the complete state of all the objects is shown, including any inherited members. In the compact view, many of these details are removed, leaving just a unique name formed from the class name of the object and an instance count (incremented each time an instance of that particular class is instantiated.) Finally, in the minimized view, objects are simply shown as points in a diagram, and the unique names (defined the same as in the compact view) are optionally shown as labels. JIVE also provides a hybrid between compact and minimized views, called a "Call-path view". Here, the minimized view is used for objects by default, but when an object has one of its methods called, it switches to the compact view. Such a view is useful when the user wants to focus on the state of the objects during a particular method

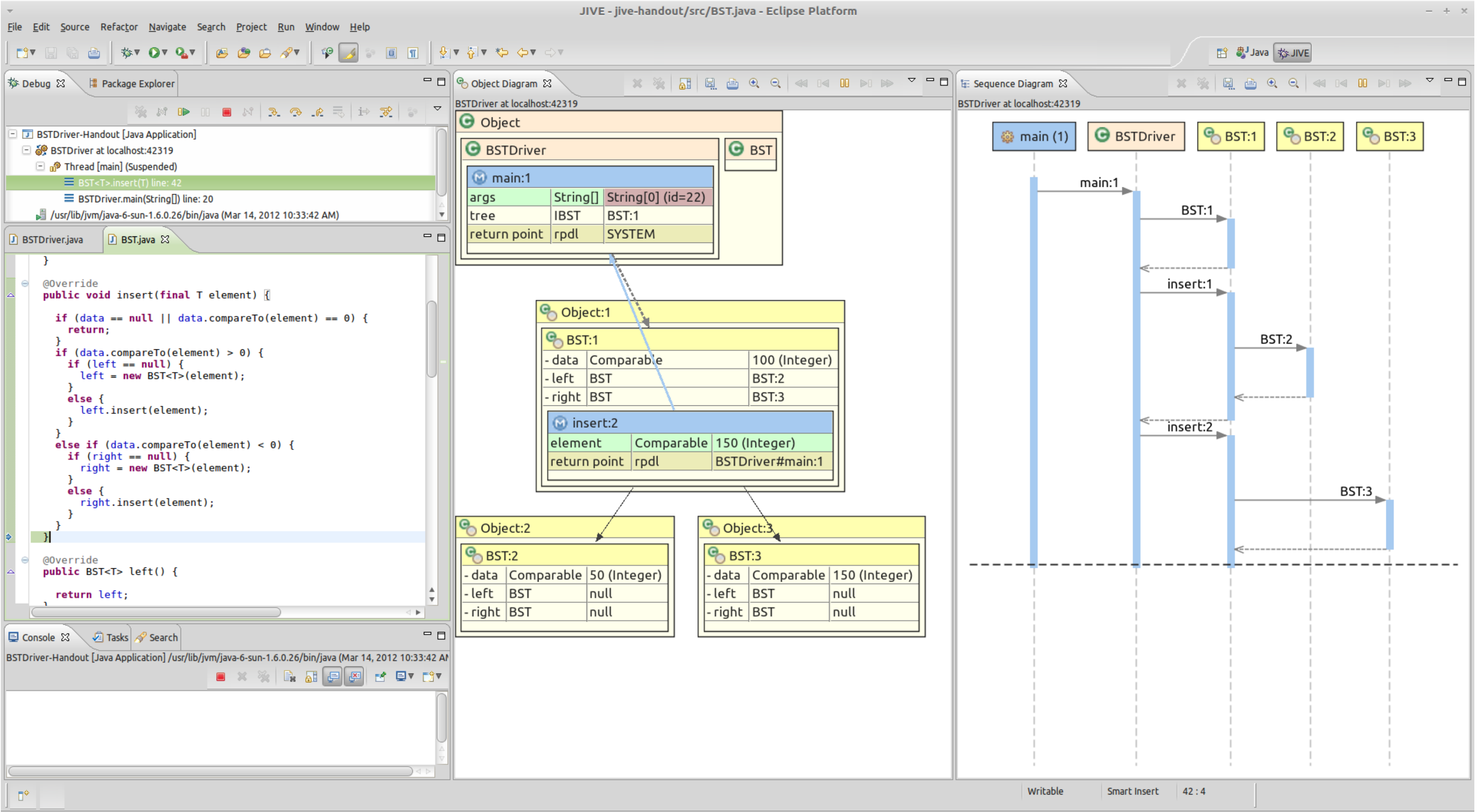


Figure 5: JIVE providing a visualisation within Eclipse.

invocation (or series of invocations.)

JOVE

JOVE is a further system developed by the same group who developed JIVE (Reiss and Renieris, 2005). It uses many of the same principles as JIVE and similarly it takes a dynamic and interactive approach, though differs in that it works alongside a live program rather than running post mortem. It also uses a different interface that aims to give a deeper, more detailed understanding of the code than JIVE. It does this using boxes of colour divided up into segments that represent the information about each object or thread (Figure 6).

JOVE first divides the image up using vertical dividers, with each section representative of each source file that is in use throughout the execution of the program. Within each file, it identifies “blocks” of code which will nearly always be executed in sequence – those without any branches or similar conditions. The amount of time spent in each block per file is then visualised on the lower half of the diagram, with a pie chart displayed towards the top of each vertical section – this visualises the amount of time each separate thread has spent executing blocks within the given file.

While this is the only view JOVE provides, it is still interesting from the perspective of granularity since it effectively visualises large applications from a high conceptual level. Individual objects, method calls and lines of code are not represented since JOVE is targeted at a size of application that would make those approaches ineffective. Instead, a completely different approach has been taken that functions meaningfully with large scale applications.

JOVE is also highly customizable, with a command line option provided that launches JOVE along with a normal Java application. The user can select exactly what classes they want to visualise, what they want to exclude and also the type of scale used for the graphs.

While this approach is very visual and provides a view that can be analysed after

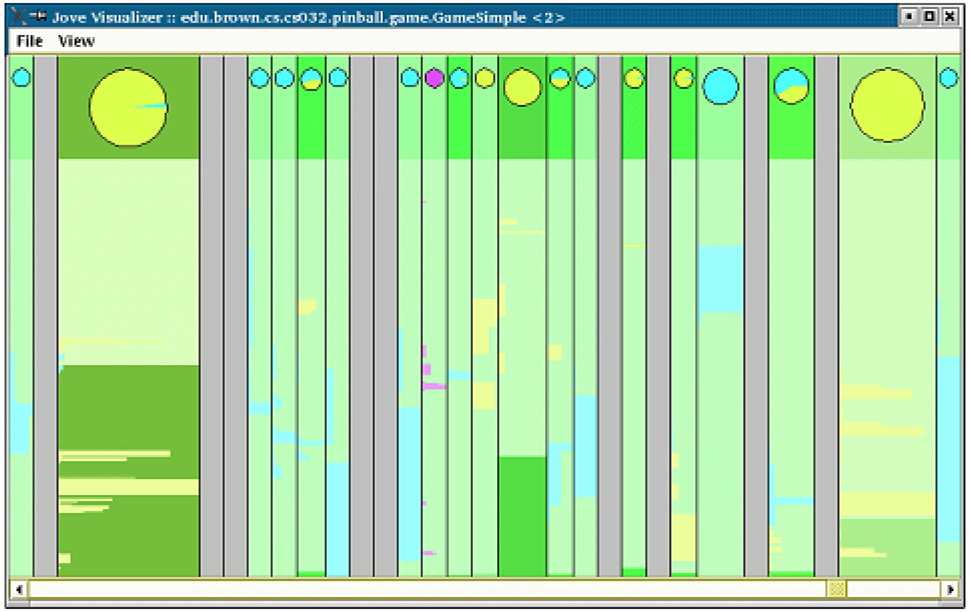


Figure 6: The colour box view provided by JOVE.

the user is aware of what each part entails, it is again not geared at beginners. It is good for visualising the details of large, complex enterprise level programs that may consist of many components. It is particularly suited for programs that are written with many threads simultaneously accessing the same or similar blocks of code due to a large part of the visualisation being reserved for the depiction of how various blocks interact with the threads. The concept of the code blocks are interesting however and may be a relevant aspect to study for this work, since they provide a mechanism for identifying mostly atomic operations that do not need to be written separately in their own methods. This could potentially be useful for providing a layer of abstraction to notionally group such code blocks.

2.12.2 Interactivity

It has already been established that interaction is an important aspect of learning through multimedia in general. A number of existing tools utilise varying degrees of interactivity in order to try to maximise the learning outcome; in fact most such educational tools seem to have at least some degree of interaction – from just allowing the user to control the pace of an animation to forcing them to answer questions in order for the visualisation to continue.

JHAVE

One example in the "interactivity" category is JHAVE. JHAVE is a dynamic, post-mortem and interactive algorithm visualisation system which allows a user to step through pseudo-code to visualise a number of algorithms. It forces the user to pay attention to the events that are occurring through the use of stop-and-think questions; Figure 7 shows a pop-up question being asked when stepping through a visualisation of a binary tree. The authors of JHAVE argue this is an extremely important feature (Naps, 2005) and that visualisations without interactive features do not necessarily translate to an initial understanding of the entity being visualised, despite being praised by those who already understand the algorithm

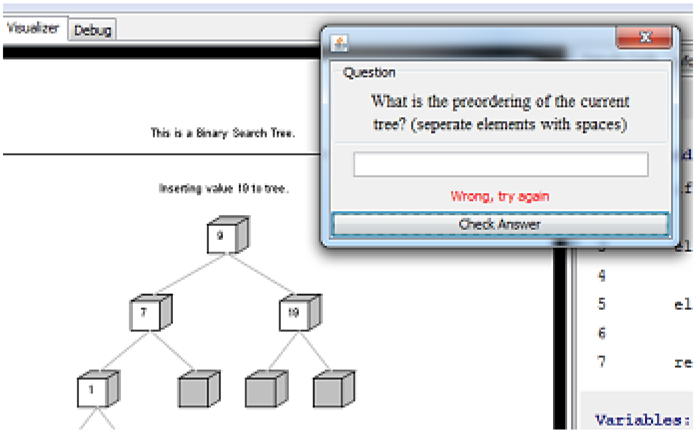


Figure 7: A stop-and-think question asked by JHAVE to ensure the student understands the visualisation.

in question (since to them the visualisation highlights the subtleties and nuances of the algorithm in a meaningful way.)

Trakla2

Trakla2 (Malmi et al., 2004) is another algorithm visualisation system which makes heavy use of interaction; the student is responsible for running steps in the chosen algorithms via a visual interface and this is automatically assessed by the server. The user is heavily involved in creating the visualisation rather than just watching it happen; they use a visual, online environment to answer questions on algorithms in a visual way. They are also given the option to resubmit their results if they initially submit an incorrect solution. It builds on the idea of stop and think questions in JHAVE to essentially pose a question to the user about the progression of the visualisation at each step. The authors have shown that this created a positive learning outcome for the participants in their study. The students were motivated enough to resubmit the solution multiple times until they devised the correct solution (Malmi and Korhonen, 2004).

UUhistle

Whilst it has a strong presence in algorithmic systems, interactivity has also been heavily used in programmatic systems. UUhistle (Sorva and Sirkiä, 2010) is a dynamic, live and programmatic (see sections 2.11.4, 2.11.5, 2.11.3) tool which visualises Python programs. Its interface contains a section for the code, the call stack, the current frame, the heap, and separate subsections for classes, functions and operators (Figure 4). As the program executes the components within the interface change to reflect this execution. The currently executing line number is shown highlighted in the editor, and as each line executes an animation step occurs.

When a new object is created from a class, the visualisation shows a space being reserved on the heap for this object, and its fields are populated as the constructor executes. The call stack is updated with a new stack frame when a method is entered and removes it when it exits. Mathematical evaluations are shown as they are evaluated on the stack by the use of the various operators (in the operators section of the visualisation.) Overall, this shows how the execution of a program unfolds on a line by line basis through the visualisation which UUhistle provides, and thus goes some way to helping the user formulate a notional machine based on this visualisation.

At a basic level, UUhistle provides a fine degree of interaction over how the user controls the running of the program. The controls are easy to use, and provide simple, clear buttons for executing the program, stepping through it line by line and controlling its speed. It also supports reverse execution by saving and recalling the state of the notional machine at each instruction.

UUhistle's key interactive element, however, is a mode where the user is responsible for executing the program by interacting with the notional machine to create objects, initialise them, call methods on them and so on. In Figure 8 for example, the student is creating an object of type "Person" on the heap, after having defined it in the "Classes" area. This ensures that the student understands what is happening – if the user attempts to perform an incorrect interaction, then the

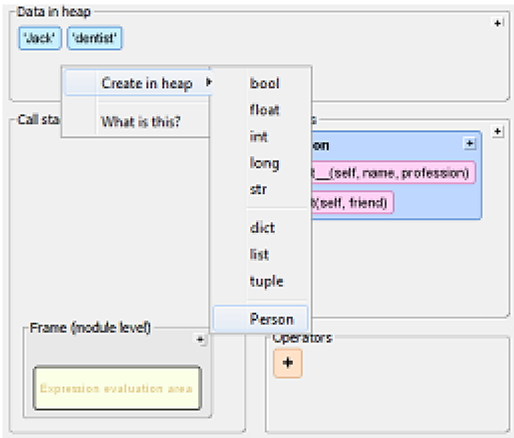


Figure 8: UUhistle in interactive mode, where the user is responsible for executing the program by interacting with the notional machine for each step.

code area turns red and the user is prompted to step backwards and correct the mistake. Programs can be executed in this way in their entirety, or can be set up so only certain parts need to be executed by the student. This means that not every little action needs to be manually executed, but a number of key checkpoints can be set up which ensures that program is being followed through carefully, not just being skipped over with no knowledge of the events within. Alternatively, it can help ensure that a particular commonly misunderstood section of the program is stepped through and understood thoroughly.

Jeliot

A related tool, Jeliot, works on a similar conceptual level to UUhistle, but the environment itself is slightly different and works with Java programs rather than Python (Moreno et al., 2004). It too fulfils the characteristics of being dynamic, live and programmatic. It has an instance area (corresponding to the heap area in UUhistle), a constant area, method area and expression evaluation area (Figure 9). Similarly to UUhistle it also has space for the program code on the left hand side and highlights the subsection of code currently being executed as the execution unfolds. The latest version of Jeliot (Jeliot 3) has also been developed into a BlueJ extension (Myller et al., 2007) which works alongside BlueJ's static class view to

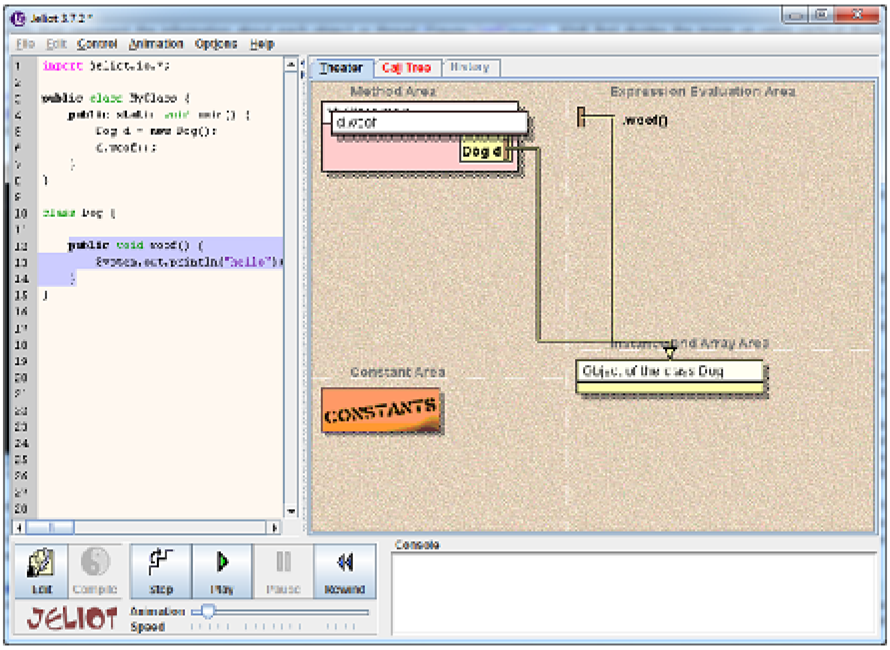


Figure 9: The Jeliot interface.

provide a dynamic view of the program as it runs. Jeliot can also visualise to some extent the object-oriented nature of programs, objects being created, references to those objects created and passed around and then destroyed.

Jeliot has relatively basic interactive elements – like UUhistle, it allows students to either step through the code or run it, and in the latter case control its speed. This however is the extent of Jeliot's interaction. It has no similar concept of stop and think questions, and while it allows limited debugging options such as setting breakpoints, it does not support reverse execution (simulated or otherwise) as UUhistle does.

One experiment found that while students found Jeliot useful for debugging programs and found the interface behind it easy to use, the animation did not make it any easier for novice students to learn – it was too hard to follow (Moreno and Joy, 2007). They recognised the benefits of Jeliot in some areas but overall stated "the transfer of knowledge from the tool to the student is not successful." They suggest some areas for improvement that may help to circumvent this issue. One of these involves the use of stop-and-think questions, reinforcing the idea that a greater degree of interactivity is essential for such a tool to work effectively.

Both Uuhistle and Jeliot work on relatively low levels of abstraction, at the highest it is a line by line level (and this can drop down to the statement level.) This may work well for small programs of a few lines, however for programs that are more complicated it presents an overload of information. Both JHAVE and Trakla2 work on a relatively high level of abstraction, as by design they are algorithmic visualisation tools. While they expose individual algorithmic steps, they do not expose individual lines of code in any particular language.

2.12.3 Visualisation type

Algorithm visualisations / Program visualisations

Most tools presented in this review are designed to visualise generic programs – Uuhistle, Jeliot, Jive and BlueJ all fall into this category. This does not preclude programs that utilise algorithms from being executed within the tools, but they are not specifically designed for the visualisation of algorithms.

However, both Trakla2 and JHAVE are algorithm visualisation tools, designed not to represent or visualise arbitrary programs but instead specific algorithms, such as a variety of searching and sorting algorithms. Interestingly, both these tools also require a high level of interactivity. In Trakla2, the user has to manually run the algorithm using the interface rather than just observing its behaviour. In JHAVE, the visualisation unfolds automatically but the user is required to answer questions correctly before certain steps of the algorithm will complete.

Stack Trace Visualisations

While not a categorisation in its own right, one interesting aspect to discuss is how different tools represent the state of a running program. In the object-oriented, classful domain we are covering here this falls into two main categories – the first is the objects that have been created and the references they contain to other

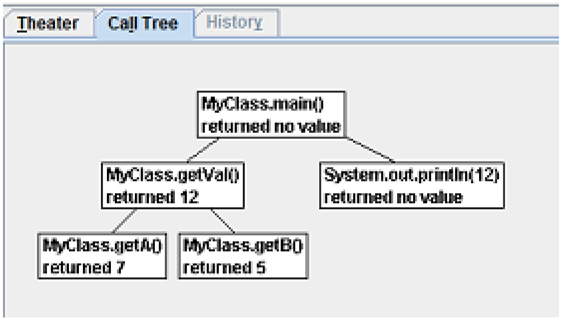


Figure 10: The call tree view in Jeliot.

objects. The second is the stack trace or its equivalent; uncovering the execution path that has led to the current method being called – it is this aspect we cover here.

The traditional method of representing this is some form of list or tabular structure of the various locations on the stack, including a source file, method name and line number for each location. While this simple approach has its advantages in certain situations, it does not lend itself much to a graphical context, implying that perhaps a different approach would be more useful for this domain.

Jeliot

Jeliot uses the approach of a call tree, which shows the call path throughout the method, not just the current stack frame (Figure 10). This has the advantage that it can show more information than a traditional stack trace, since a tree can be generated to show a full execution trace for any given thread. However, while this is comprehensive it has the potential to generate an overwhelmingly large amount of information which may not be helpful after a certain time. It can also make it more difficult to identify the path of the current stack trace in the given call tree. Jeliot's implementation of the call tree is also displayed in a separate diagram. It is not part of the visualisation, so the two cannot be viewed concurrently. This is a specific violation of the synchronizing principle discussed in Section 2.8.8; the user has to hold information about one diagram in their head whilst viewing another which leads to an unnecessary increase in congnitive load.

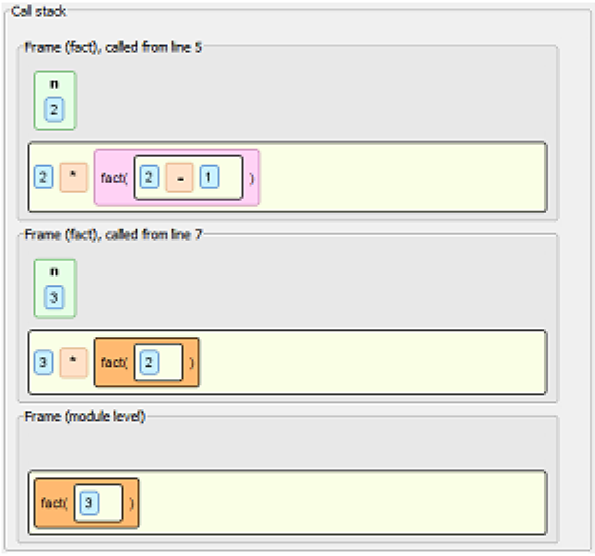


Figure 11: UUhistle's call stack panel.

Uuhistle

UUhistle uses a more traditional call stack rather than a tree based view, which shows the current stack frame (Figure 11). However, unlike Jeliot the view is integrated into the main visualisation window, without any need for other tabs. While this makes the stack view slightly clearer, it is still a violation of the synchronizing principle – despite being in close proximity, the stack is still a separate diagram. Whilst the stack frames are arranged in a list like structure, the items in the list are not just the locations in source files of each containing element, they are graphically represented. Figure 11 shows a snapshot from a recursive factorial method being called, initially with 3 as a parameter. Looking through the stack frames, it is easy to see the wider context of each frame, including how the results from methods that have been called will be integrated into the overall calculation.

Summary

While Jeliot and Uuhistle both provide stack trace visualisations, these are distinct diagrams from the heap. While this is the traditional approach, it requires the user

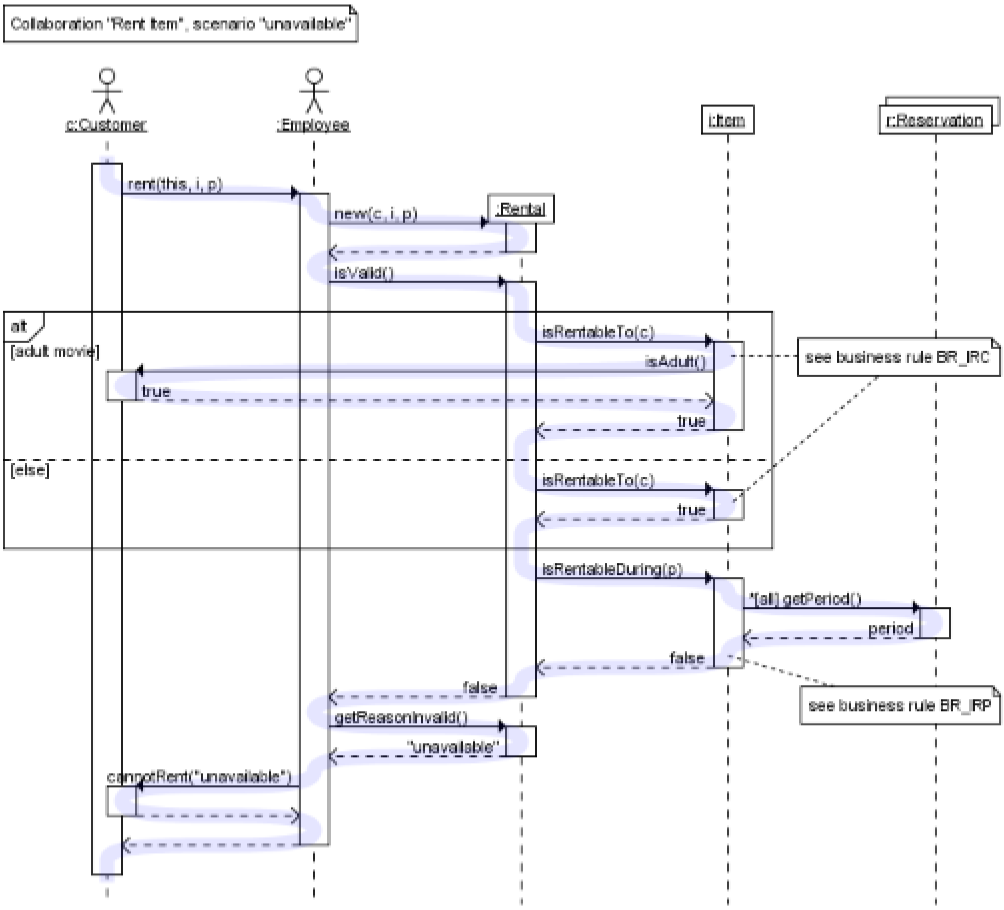


Figure 12: A UML sequence diagram (A quick introduction, 2016).

to mentally synchronize the two diagrams. This task places an increased cognitive load on the user, and can thus detract from the useful information being displayed by the visualisation.

2.12.4 Snapshot / Continuous model

The systems described thus far all represent continuous execution – they form animated visualisations based on the execution of a particular program or algorithm. In particular, Jeliot, Jive, Uuhistle and JHAVE are all examples of continuous, animated visualisations.

UML sequence diagrams

The most common form of UML diagram represents the static relationships between classes, representing the details of methods and fields available on each class (such as the number of parameters and the parameter types.) UML sequence diagrams are an extension of this common static notation, providing a popular "snapshot" notation of a program execution. These diagrams can represent the dynamic calls between objects, with this form of visualisation showing the object interactions from top to bottom. An example is shown in Figure 12. The objects, classes, parameters and return types involved in an execution can all be visualised in this manner. However, UML sequence diagrams can only feasibly represent short timelines with few objects before becoming cluttered and hard to comprehend. They are also not designed for novices, as the representation used is comprehensive, but often complex.

2.12.5 Post-mortem / Live visualisations

This category can only usefully be applied to tools that produce continuous visualisations, as visualisation "snapshots" (such as BlueJ's class diagram) do not run in any sense, either post mortem or live.

Jeliot, Jive and Uuhistle all produce live visualisations from running code. Not all visualisation systems take this approach however. ParaGraph (Heath et al., 1991) aims to provide the user with several different views for representing message-passing parallel programs. Instead of running alongside the program, its input is a trace file that represents the execution of the original program; the visualisation is produced from this trace file directly.

2.12.6 3D visualisations

The majority of educational tools use a 2D visualisation and corresponding mental model. In some cases there is an element of z ordering with overlapping components, but few work within a true 3D space. However, some approaches have been taken that aim to utilise 3D in displaying visualisations rather than the more traditional 2D approach, both in and out of an education environment.

ViSE3D

ViSE3D (Fronk et al., 2006) displays the relationships between packages and the classes they contain as a 3D association network, using geometric algorithms in an attempt to lay out the network in a clear and meaningful way. The diagram is a static representation but holds some level of interaction with the user – they can zoom, rotate and navigate the 3D space to view the model. This could potentially be a good solution for those looking to visualise the relationship between large number of packages and classes, since an inherent advantage of 3D is the ability to display more data in the available space, but the use in education especially is debateable. While the algorithms for laying out the elements in the model are impressive, the very nature of the 3D model means that it takes a number of interactions from the user simply to navigate the space, not just a single click and grab that is commonly associated with panning in a 2D area. This therefore means that if such an approach was used the student would have to spend time finding what they want to visualise and navigate to it, potentially becoming frustrating and diverting their attention away from the key elements used in the visualisation.

3D algorithm visualisation

3D visualisation has also been used in the area of algorithm visualisation a number of times previously (Carson et al., 2007; Stasko, 1990; Stasko et al., 1993b) though in many cases the advantages offered over traditional 2D models are not apparent.

In particular no study could be found that showed any advantage of using such a navigable 3D visualisation centred around programming in an educational setting. One particular study (Teyseyre and Campo, 2009) analyses 3D software visualisations in general, and within this analysis identifies a number of areas in which they believe they can be potentially helpful for software maintenance and design in industry. As well as the strengths of 3D visualisations such as greater information density they also identify weaknesses such that 3D metaphors are harder for users to understand and interact with. Many of the identified strengths fall within the bounds of being helpful for professional or more adept programmers, but do not translate to the requirements of novice programmers. The ability to display information more densely for instance is arguably a disadvantage according to Mayer's apprehension principle (Mayer, 2002). There is also a necessary interactivity penalty with a 3D interface – more complex controls for interacting with and navigating around the space are necessary.

The same study (Teyseyre and Campo, 2009) also outlines several important issues that are important to produce an effective 3D visualisation, though it is worth noting that these are in a generic visualisation context and not specifically geared to computing (let alone computing education). Some of the proposed issues could arguably also be applied to 2D in some contexts, so they are worth reviewing. Foremost is the choice of layout algorithm, which is crucial for producing a visualisation which is comprehensible and easy to understand. Whilst important in both 2D and 3D visualisations this is particularly the case in the latter. The potential is present for fewer intersections volumetrically via the use of 3D arcs (Parker et al., 1998), but the user will only ever see a 2D projection of this 3D visualisation. This therefore raises additional problems to be overcome, caused by occlusion and perspective (Schulz and Schumann, 2006). Representing nodes and the links between them is another issue, since memorable, representative aesthetics need to be chosen that are clear to the user – 3D offers more scope here but adds more potential for error as a result, and needs more complex algorithms in its representation. The final issue mentioned is that of clustering; the benefit of using 3D to be able to display more nodes and links does not solve the issue of scaling. Clustering algorithms group together similar entities, mitigating this issue somewhat and improving the clarity of the visualisation. This is also potentially an important topic for 2D visualisations, since while the problems of perspective are not as apparent, if large groups of nodes and links are involved then clustering related components may act as a visual cue to aid understanding.

3D visualisation summary

From the above it appears that a 3D context for the model presented in this research would not be the best approach; with little or no evidence to suggest that this could be a useful resource in an education setting, but a number of articles suggesting that the additional complication that 3D provides to the user might prove more of a burden than a help in practice (Mayer, 2002; Teyseyre and Campo, 2009).

2.13 Additional aspects of existing systems

In addition to the categorisations that have been defined above, there exist several other important points that are relevant to existing systems. These are discussed here.

2.13.1 Layout

The layout of the notional machine is a key aspect, and one that can significantly alter the mental model that is being portrayed. The most popular layout used by tools specifically geared to programming education appears to be one where the various "traditional" areas are being displayed – different labelled areas for the stack, heap, constant and method area for instance. Both UUhistle and Jeliot fall into this category, though no significant material in the literature could be found which showed this view was necessarily the best to use.

JGrasp

One particularly interesting tool in terms of layout is JGrasp (Figure 13). JGrasp is an IDE which has built-in support for the visualisation of data structures. It is particularly aimed at CS1 (introduction to programming) and CS2 (introduction to data structures and algorithms) so is targeted at novice programmers and aims to help them visualise and therefore better understand data structures and the algorithms which relate to them (Hendrix and Cross II, 2007).

The visualisation is tightly integrated with the IDE and auto-detects what form the visualisation should take based on the code that the user has written via a built in "structure identifier". If the IDE detects that a binary tree has been used for instance the visualisation will organise itself into a tree-like structure. If it detects a doubly linked list it will organise itself as such, and likewise for other detectable structures. The visualisations that it uses are correct, clear and helpful for simple programs without displaying too much cluttered information. One of its main features is the use of the visualisation layout to convey information about the type of data structure being visualised.

The ability to lay the visualisation out based on the type of structure in use is achieved through a structure identifier which statically analyses the code to determine the best layout. It applies its knowledge of common programming idioms to represent data structures to the user's code and from this makes an attempt to extrapolate the data structures in use. A linked list for instance is most commonly represented as an object containing a field of the same type, which may be set to the next object in the list or null if the object in question is the last in the list. The effectiveness of the JGrasp structure identifier was put to the test by using numerous examples from textbooks and then checking whether the correct visualisation was chosen or not (Montgomery et al., 2008). This was shown to be highly effective, with 80% of the examples rendering correctly straight away. Out of the remaining examples most could still be configured to display correctly with only 3% failing to display meaningfully at all.

The level of abstraction is also visible in the layout of the visualisation. Jeliot's

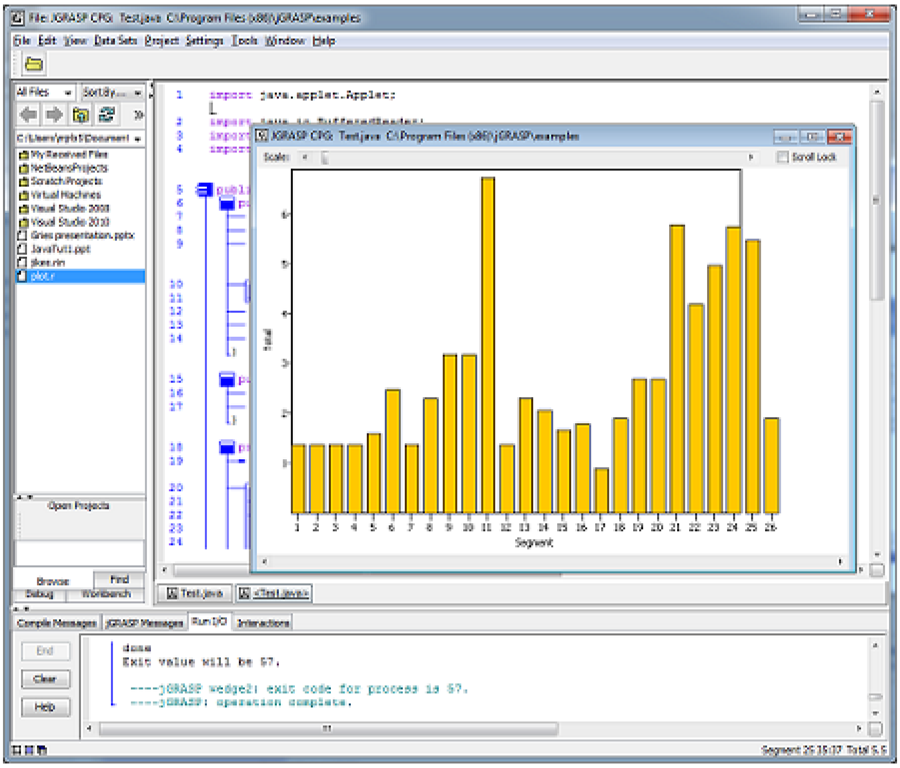


Figure 13: The main interface of the JGrasp IDE.

visualisation is closely based on the interpreter implementation; in this sense the notional machine matches the interpreter relatively closely. The underlying details of how the language is implemented are often exposed – synthetic references to objects in inner classes are visualised for instance. These are often considered implementation details in Java, rather than a component of object-oriented design.

An example of this behaviour can be seen in the visualisation of the following simple code listing[2]:

Listing 2.2: A simple Java program that can be visualised with Jeliot

```java
public class MyClass {
    /**
     * Main method.
     */
    public static void main() {
        new Dog().woof();
    }
}

public class Dog {
    public void woof() {
        System.out.println("woof");
    }
}
```

One of the later steps in the visualisation, when the `woof()` method is being called, is shown in Figure 14 on the next page. From a common object-oriented viewpoint, we might expect the *object* itself to contain the non static method `woof()`, or at least to be directly related to it somehow. In contrast, we would expect a static method to belong to the class. In Jeliot's visualisation however the

[2]The omission of "`String[] args`" as a parameter to the main method is deliberate, Jeliot makes use of a custom interpreter which does not use command line arguments (and therefore does not require this parameter.)

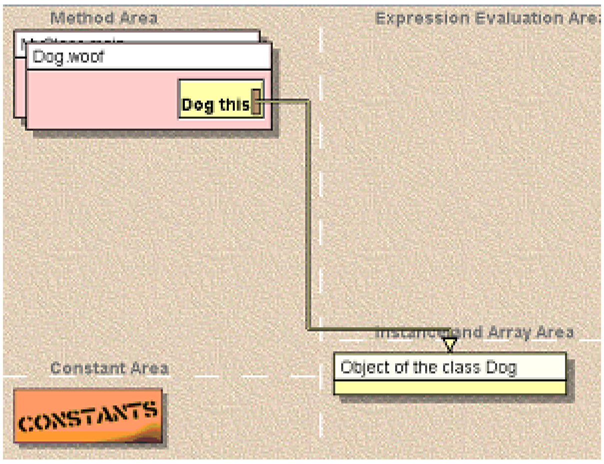


Figure 14: One visualisation step of Listing 2.2 shown in Jeliot.

non-static method is shown as belonging to the class, not the object – it simulates the expected behaviour by providing a reference to the underlying object on which it will be called.

This is a functionally valid visualisation, indeed it reflects how Jeliot's interpreter works internally. This does not necessarily mean it is the best way for a student to *understand* object orientation however, or that it is the most useful approach. This highlights a key point in that the mental model should be treated as an abstract layer above the implementation. As long as it is correct and helpful in explaining the behaviour, it does not need to, and arguably should not, reflect the implementation details where they complicate conceptual understanding. This is shown by the congruence principle (Section 2.6.3).

2.13.2 Aesthetics

While aesthetics may seem like a subjective point to examine, it has been shown that good visuals form a positive emotional reaction in the user, and thus are an important concept from a HCI perspective (Purchase, 2000). This is especially important when teaching, since a positive emotional reaction can cause students to learn more effectively (Pekrun et al., 2002).

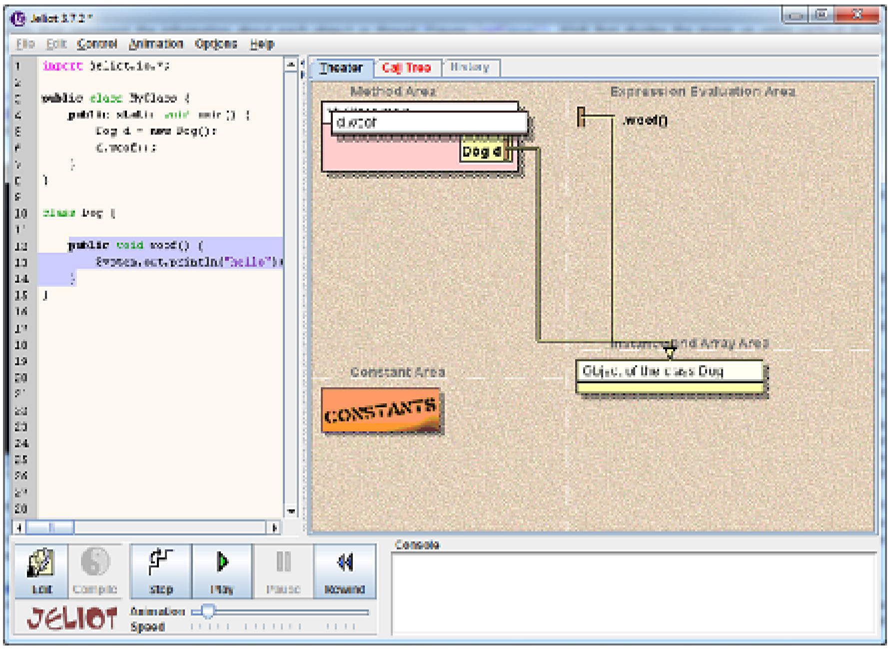


Figure 15: Jeliot's interface.

Jeliot

Jeliot is an example of a tool with rather antiquated visuals (Figure 15). It implements basic animations using plain Java2D and Swing – the overall appearance of the current version does not seem to have significantly changed since Jeliot 2000, the previous version of Jeliot. While it has been shown that learning can sometimes occur more effectively when animation is used to explain complex concepts (Weiss et al., 2002), a study whilst using the tool found that many students found the animations annoying after a while since they took too long to execute and drew their attention to aspects of the code they were already familiar with (Moreno and Joy, 2007). They were therefore prone to simply running the animations at full speed after a while, essentially rendering them futile.

Uuhistle

UUhistle (Figure 4) uses a more modern skin and significantly less time consuming animations than Jeliot. There are also a number of aesthetic aspects incorporated into the visualisation that Jeliot does not implement. Some are minor, such as rectangles having rounded corners, and others are more prominent, such as different colours being used for different blocks which makes them more easily identifiable. UUhistle also has the advantage that the visualisation is immediately available to run, whereas in Jeliot the code cannot be edited when the visualisation is executing – an edit / compile toggle button has to be clicked which switches between the two (in addition, the switch is not instantaneous, an animation of a curtain pulls back and forth which takes a couple of seconds.)

JIVE

JIVE uses simple rectangular boxes and arrows to represent classes, objects and references. It does not appear that a great deal of effort has been expended in the aesthetics, it is a functional visualisation with very little that is there purely for aesthetic purpose. This reflects its purpose as a tool for experienced programmers to use in the understanding of particular programs (as opposed to novices using it to understand the concepts of programming.)

2.13.3 Ease of use

Ease of use is one area particularly relevant to tools used in computer science education, since a tool that is difficult to use could well alienate beginners before they have attempted to use it, even if it provides a number of useful functions. There is also the argument that if a student has to spend too much time learning how to use a particular tool, this detracts from the time they could have spend learning to program. In any case, a tool that is hard to use increases the cognitive load of the student, forcing them to concentrate on the use of the tool rather than the useful material depicted int the visualisation.

Two such programs that score highly in this regard are UUhistle and Jeliot. Both are undoubtedly easy tools to use, with a simple interface that automatically populates and runs the visualisation alongside the code the user enters. In both cases, the IDE is split into distinct, clear sections, and it is immediately obvious how both systems are used. The controls for both environments represent the ISO 7000 tape player style controls (ISO, 2012) and thus are also familiar.

However, some visualisation tools aimed at experts are not easy to use. This is understandable since they are not aimed at novices, and experts can thus take longer to learn their functionality. Nonetheless, it is an area worth examining since it shows that a particular approach is needed within this specific context.

JIVE

JIVE is a comprehensive tool that supports many different kinds of visualisation – different data structures such as trees, method execution in a number of contexts and interaction between separate threads such as using the producer / consumer model. This is a good set of features and it has been shown to help in areas such as debugging programs and visualising errors within (Girgis et al., 2005). Taking the act of debugging concurrent programs as an example, JIVE can be queried to produce visualisation "snapshots" at the moment any particular thread exits via an exception. While this can be very useful for debugging such applications, it is really too complicated for a novice to use; they require something simpler and arguably would not require certain functionality such as the ability to debug multi-threaded programs. They would also most likely find it difficult to choose which view was most appropriate for the program being executed. While the different conceptual views provided by the tool are certainly interesting and useful, there is no automatic switching to the most appropriate view that a student with no prior programming experience may require. JIVE also provides a large amount of high level information in regards to the performance of the program. While comprehensive, this may well alienate beginners.

JInsight

Another system, JInsight, was developed by IBM (Pauw and Vlissides, 1998), and as far as the author can tell was the earliest example of a Java based visualisation system. It provided a very detailed trace data over a Java program. This was not designed for novices, instead designed for experts to gain an understanding of an existing application. It also required a modified JVM (Java Virtual Machine) and thus the program being inspected had to run alongside that particular modified JVM.

JInsight provides large amount of execution information through a number of separate views – pattern extraction, task-oriented tracing, database techniques and interactive navigation (Pauw et al., 2002). This type of information could be hugely useful for someone performing a rigorous analysis of an enterprise level program, but is undoubtedly not suited to anyone who does not understand the basics of object-oriented programming.

2.14 Table of existing systems

The features of the existing visualisation systems are summarised in Table 1.

	Granularity	Interactivity	Visualisation type	Snapshot/ Continuous	Post-mortem/ Live	2D / 3D
Jeliot	Fixed, line by line execution	Yes (but basic); the user can control the speed and step through code	Program	Continuous	Live	2D
Trakla2	Fixed, line by line execution	Yes; the user is responsible for running steps in the algorithm through a visual interface	Algorithm	Continuous	Live	2D
JHAVE	Fixed, line by line execution	Yes; stop and think questions test the user's understanding as the visualisation unfolds	Algorithm	Continuous	Post mortem	2D
JIVE	Multiple levels of granularity, objects are shown with little detail by default, but can be examined further if required	Yes, multiple levels of granularity, objects can be examined in detail, the speed of the execution can be controlled	Program	Continuous	Post mortem	2D
JOVE	Fixed, broad level of execution	None	Program	Continuous	Live	2D
Uuhistle	Fixed, line by line execution	Yes, the user can be responsible for executing a program by creating objects and calling methods	Program	Continuous	Live	2D
UML sequence	Fixed once created due to static nature of diagram, but different details can be included or omitted as required	N/A	Program	Snapshot	N/A	2D
ParaGraph	Fixed, but many types of views can be displayed depending on the information required	None	Program	Continuous	Post mortem	2D
Vise3D	Fixed, a broad overview using a 3D association network	Yes (but basic); the user can zoom, rotate and navigate the 3D space to view the model	Program	Snapshot	Post mortem	3D
JInsight	Fixed, but many types of views can be displayed depending on the information required	None	Program	Continuous	Live	2D
Jgrasp	Fixed, line by line execution	Yes (but basic); the user can control the speed and step through code	Program	Continuous	Live	2D

Table 1: A summary of existing visualisation systems.

2.15 Summary

There is sufficient material in the literature to suggest that an environment which models the object graph, either in live or post-mortem form, could be useful. This is the case both for the comprehension of programming, as well as the comprehension of individual programs. However, no conceptual model, nor associated tool, currently exists that provides an effective display of a program's object graph across the first year of a computer science undergraduate course. Some tools are able to depict the object graph of small programs, however the vast majority of models become ineffective when examining programs of any significant length.

The discussion presented here gives us a framework for some goals and guidelines for the design our own system. We shall refer back to them later when discussing the design of Novis (Chapter 5).

Chapter 3

Problem Space

3.1 Introduction

This chapter describes the scenarios where the notional machine should be useful, outlining the areas it covers as well as those areas which have been specifically declared as out of scope. Broadly speaking, the machine is aimed at first year computing students and therefore focuses on the material covered within that year. For this purpose, a generalisation about a "typical" first year course is made. It is hoped it may be useful for specific tasks in later years, but establishing this falls out of scope of this work.

There are a number of advantages of restricting the scope of the notional machine in this way. The biggest advantage is that first year course material is generally more consistent across establishments than material in later years. This is understandable since the basics of programming generally need to be taught before more advanced concepts can be covered, but after the basics are understood then teaching can (and often does) diverge rapidly between institutions (and in many cases within institutions, based on the choice of modules by the student.) The concept of a year is also a clear one, and a reasonable timescale – the student will not be trying to understand the notional machine for one week and then moving on, it is a tool that should be useful to them throughout their introductory programming course.

Within this first year, the notional machine should provide a good model to students with no programming experience at all; launching them straight into the concepts and ideas behind object-oriented programming. They should be able to use the notional machine to explain what is happening as their code unfolds. For this scenario, the notional machine notation should work at a relatively high level of detail.

The model should also be able to demonstrate object-oriented concepts to a student who has previous programming experience, understanding basic procedural concepts (functions, if statements, loops) but with no experience of object-oriented programming. In this scenario the notional machine should work on a higher conceptual level than for someone who has no programming experience since they will already be familiar with certain concepts which will not necessarily need to be exposed to them at a great level of detail. However, all the object object-oriented features will still need to be shown in detail in order for them to gain an understanding of this new domain.

3.2 Core requirements

Based on the above, the notional machine should support visualising the following:

Basic object-oriented concepts All students being taught object-oriented programming will have to understand several basic concepts before progressing onto others: classes and objects (and the difference between them), encapsulation, fields, methods and method calls. The model used for the notional machine should make these concepts clear and provide a framework for students to understand them.

Advanced object-oriented concepts As the student progresses through their first year they will have to understand more advanced concepts related to object orientation such as polymorphism, inheritance and method overriding. The notional machine should help the student to understand these concepts.

Understanding a program's structure The notional machine should work so that someone who has gained an understanding of object-oriented concepts should be able to run a program using an implementation of the notional machine, and gain some understanding about its structure and how it works internally without needing to examine the source code.

Debugging student sized programs Students should be able to use the notional machine to debug their programs. They should be able to run their program through the notional machine and identify how it is behaving. From this information they should then be able to obtain information to deduce why it is not behaving as they expect.

3.3 Out of scope

The following concepts have been defined as out of scope. This is mainly because they fall into the category of concepts not commonly taught in first year programming. It should be noted that the notional machine could be modified to include these features in some cases, and hence may be extensible for further specific use cases. For this work however, both in the design and the implementation of the notional machine, they are not included. Programs written that make use of these features may still be partly explainable using the notional machine, but this is not guaranteed.

Identifying issues with compilation The notional machine is limited to understanding "working" code, in this sense defined as code that compiles and executes (but not necessarily in the way the user expects.) It will not attempt to provide a framework for understanding why code does not compile.

Modeling concurrent applications Concurrency is growing in popularity and as more modern CPUs have more cores and are thus able to deal with more concurrent threads of execution, the benefits of concurrent programming are becoming clearer. However, while some languages such as Occam are clearly built around the idea of parallelism and concurrency, Java is not. As such, concurrency is not at present a common element in first year computing courses. There is one possible exception to this, however, and that is dealing with two threads of execution when GUI frameworks are concerned (the "user" thread and the event dispatch thread, or equivalent.) This is a common element of first year programming, although the threading concept is not necessarily explained in full as the student starts using a GUI framework. Since it falls under the remit of first year code, the notional machine should provide some explanation of this behaviour, though it will be limited to low numbers of threads.

Making use of native code The notional machine will be able to model object-oriented concepts. However, if a user uses JNA or JNI within their application this is not an area the notional machine will attempt to cover. If the native calls also happen to be designed in an object-oriented manner then the notional machine may be able to represent them effectively. However, this will be by coincidence rather than design and is not an area that will be considered for the implementation of the notional machine in software.

Advanced generics Generics in Java are a powerful mechanism for ensuring type safety in a number of scenarios, particularly collections. Whilst the use of simple generics (an `ArrayList<String>` for example) should be represented in the model, there will not be any special treatment by the notional machine of a user writing their own generic methods or classes.

Annotations Annotations will not be treated specially within the notional machine since they are generally an advanced feature.

Multiple reference types Hard references are not the only type of reference in Java – the concepts of soft, weak and phantom references also exist each with different behaviours in how they hold onto their corresponding object, and the concept could be transferred to any classful object-oriented language. These can be very useful in certain situations, but are not within the scope of a first year programming course.

Multiple virtual machines The notional machine will be restricted to explaining the activity within a single JVM (Java Virtual Machine). It will not attempt to represent any part of an application that is run remotely in another JVM.

Multiple class loaders Multiple class loaders are classed as an advanced feature and thus will not be given any special treatment in the notional machine.

Large scale enterprise applications The model will be geared mainly towards student sized programs. While this means for larger student sized programs of up to a few hundred lines it should be helpful, for enterprise scale applications reaching into the thousands of lines it may cease to be useful.

Chapter 4

Design

The design of the notional machine is presented in two parts. First, this chapter presents a general notation to be used in the notional machine in all formats. This covers both on-screen and off-screen (such as on paper, on whiteboards or in a textbook.) Chapter 5 will then present a dynamic extension to the notional machine notation, for specific use in implemented forms of the notional machine.

While it is envisaged that a visualisation based on this design will aid students in their understanding of the notional machine, the design has specifically been conceived with the aim that it should not just be useful and usable in this context. Practical decisions have been taken to ensure the design does not require an accompanying software implementation in order to be usable. It is, for example, also representable in static diagrams that could be used in textbooks, and as ad-hoc diagrams drawn on a whiteboard by an educator.

The design outlined here affords two distinct use cases, the *understanding of programming* and the *understanding of programs*. Initially it is the first of these categories that will be most relevant. The notional machine can be used by students with no or little programming experience, and through its visualisation it should help to impart knowledge about various programming constructs. A student may, for example, study a small program executing through the notional machine implementation on the most detailed level to help construct valid mental models on the subject of object creation and destruction, the differences between classes and objects, method invocation and parameter passing. Typical questions a student in this scenario may ask are *What does a method call do?*, or *How does a parameter get passed?*

This phase of learning will typically see small programs being used. These programs however will be discussed and examined in detail – students will be encouraged to examine each line individually to understand the relevant concepts of its execution. However, once a student gains an understanding of basic programming constructs, this particular use case is no longer a valid nor useful one. While existing visualisation software exists that caters for this phase of education, it will likely pass relatively quickly – it is hoped that within a few weeks or months, students will have a good understanding of (at least) the basic constructs.

The second use case, however, the *understanding of programs*, remains relevant for much longer. All programmers, even seasoned developers, will still require an understanding of the workings of different programs that are new to them. This particular use case is thus not one that disappears after gaining additional experience (though seasoned developers will have likely mastered this process to a much greater degree of efficacy and efficiency than a computer science student.) Students will certainly still be required to understand individual programs throughout their first year of university. Questions a student would ask in this scenario relate to *details of the specific program in hand* instead of *knowledge of programming constructs*. For example, typical questions might be *Why does value* **x** *not get updated when I call this method?* or *How does object* **y** *attain a reference to object* **x***?* Many notional machines and visualisations that exist do not cover this second use case well, or at all.

While the notional machine notation is deliberately designed around a single diagram, this design is presented in two main sections. Firstly we present the details of the object diagram, analogous to the traditional heap. Secondly the notation for execution is presented, analogous to the stack (Section 4.4).

This chapter focuses purely on the abstract design of the notional machine notation. The next chapter will describe the design of an animated implementation of this notional machine in software.

4.1 Design principles

The design of the notional machine is deliberately based on underlying BlueJ functionality, including both design principles and notation. This has advantages from two main standpoints, the first of which is user familiarity. BlueJ is used in introductory programming courses at secondary schools and Universities worldwide" (Utting et al., 2012), and thus its notation is already familiar to many students and educators. The notation used in BlueJ will thus form a base for this work, though existing notation will be changed as necessary and the system augmented with new notation. For many aspects there is no compelling motive to introduce new notation (the colour and shape of classes and objects for example), so this will be inherited.

While popular, BlueJ is by no means the only possible base for the design of the notional machine outlined here. Other existing systems, such as UML, also have distinct notation for classes, objects, fields and method executed – this could have also been used as a starting point. This approach would not be invalid, and would almost certainly lead to a very different design of the notional machine notation.

The second main advantage is that it allows the notional machine to be linked with existing BlueJ facilities. As well as removing the need to re-implement existing facilities (such as the editor), this also allows for the notional machine to leverage existing infrastructure as part of its operation. For example, the existing debugger and stepper can be used to afford a powerful composite view of program execution, particularly for procedural aspects (Figure 69).

Figure 16: Representation of an object.

4.2 Methodology

While no formal methodology was used in the design of the notional machine, several drafts of various aspects were conceived, presented to peers and students, then improved as a result of feedback. In particular, many depictions of objects and methods were presented before arriving at the final design presented in Section 4.3.1. While not the most robust or structured methodology, this allowed many more designs to be trialled in a limited time by using a lo-fi, sketch based approach.

Another approach could have involved trialling these designs with the target audience; asking what each particular depiction of objects, classes and methods represented. Such an approach would have undoubtedly taken longer to prepare and analyse, but the outcome would have provided more confidence in the notation at an earlier stage.

4.3 Object Diagram

4.3.1 Objects

Objects are represented by rectangles with rounded corners. This is geometrically distinctive to rectangles used to represent classes (discussed in Section 4.3.2) yet still retains a shape that can be used for writing text effectively (Figure 16). An early prototype of this work utilised a circle instead (Figure 17). This was found to be an awkward shape on which to display lists of references – in contrast, the proposed shape can easily be expanded vertically as required for this purpose.

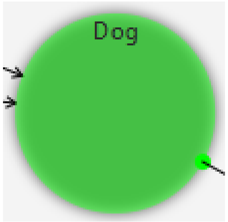


Figure 17: The “circle” representation used in the initial prototype.

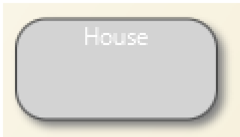


Figure 18: An object which has not yet been fully constructed.

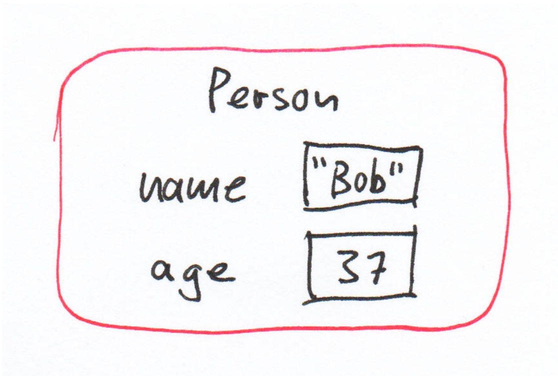


Figure 19: A hand drawn representation of an object with two fields.

A fill colour does not have to be used since this would be difficult in some instances – drawing the diagram on paper or on a whiteboard for example (Figure 19). However, if fill colours are used, “fully constructed” objects must be filled in red. The colour is chosen by convention, since this is the colour of objects on the object bench in BlueJ. Objects whose constructor has not yet finished executing may optionally be filled grey, as a visual affordance that the object is not yet fully constructed (Figure 18). The object is also drawn with its type at the top of the rectangle (Figure 16). This type must be the dynamic type of the object, not the static type of a variable declared as a reference to the object (of course, in many cases, these may be identical.)

Unique identifiers Many visualisation systems, such as UML, use unique identifiers to reference individual objects. This has some advantages, principally that it makes it easy to refer to the object in many settings – both for discussion, and for display when (for example) calling a method. However, this can also lead to misconceptions that we aim to avoid in this model. Specifically, it implies that all objects have a unique identifier by which they can be referenced. The natural assumption that can then be made is that references are themselves unique identifiers. This of course is not the case (objects can be referenced by many variables, all of which may have different names.)

One possible resolution might be to use an identifier that is, in fact, unique to any particular object in the underlying machine. There are two approaches here - to use a unique identifier assigned to the object purely in the notional machine, or use a detail of the underlying machine. The unique identifier assigned in the notional machine has the advantage of providing a more “friendly” name, but may well be confused with a reference just as easily – it may also be confusing as to why, in Java, a student could not simply “look up” any object on the heap by its unique name.

An alternative approach could be to use a detail that does indeed exist in the underlying machine. This could either be the memory address of the object, or the identity hashcode (as given by `System.identityHashCode()` in Java.) This

Figure 20: An object field.

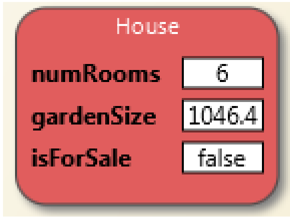


Figure 21: An object containing a number of fields.

would also allow an easier implementation, as one simply has to call an underlying method in the real machine to obtain a unique identifier. However, this would still strongly provide the assumption that an object should be able to be easily accessed via this address. While technically possible, this is not a common, nor recommended object-oriented approach.

For the above reasons, in the notional machine presented here objects do not have unique identifiers. This does not preclude the discussion of individual objects, but requires that students address them through the use of references rather than a memory address (or some other similar construct.)

Fields

A field of an object is represented as a box with a label to its left. The label must have exactly the same text as the variable name of the field in the source code. If the type information of the field is relevant, then this may also be displayed in an appropriate form (such as on a tooltip in implemented forms of the diagram.) As with objects, a fill colour is not required when this would be impractical. If fill colours are used, the box must be filled in white, and it contains the "value" of the field (Section 4.3.1). An example of a field can be seen in Figure 20.

The full object representation contains a vertical list of all its fields as described above (Figure 21). The order of the fields may be arbitrary, and does not necessarily have to match the order in the program. In some scenarios however, not all the

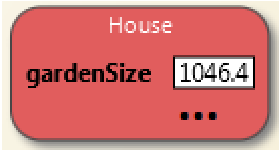


Figure 22: An object with some of its fields hidden.

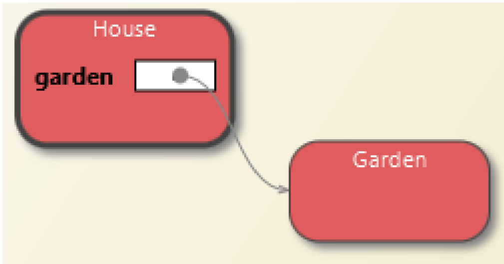


Figure 23: An object containing a field which references another object.

fields may be deemed relevant. If this is the case, the relevant fields can be shown as above and the others hidden, replaced with an ellipsis towards the bottom of the object (Figure 22). This ellipsis indicates that further fields are present, but not displayed in the diagram.

Only instance fields are shown on the object; static fields are displayed on the class instead (Section 4.3.2).

Field values

The value of the field can take two separate forms - an *inline value*, or a *reference value*. If the field is a primitive, it is displayed in its textual form in the box (the textual form should be the same as if the variable was printed) – this is the inline value. Figure 21 shows an object with a number of fields represented this way.

If the value is instead a reference to another object, then a reference value is displayed. A reference is depicted as an arrow, originating from a small circle in the field and pointing to the object that it references (Figure 23). If the object referenced by the arrow is not visible on the diagram because it is not specifically

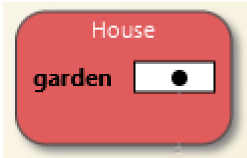


Figure 24: An object which references an object we do not want to display.

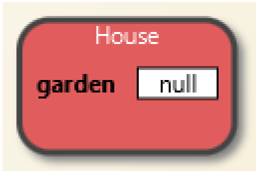


Figure 25: An object reference that currently contains the null value.

of interest, then the small circle may simply be used without the arrow, to indicate that a reference exists (Figure 24). In the case where the value of the field is null, then `null` is written in the box (Figure 25). In order to avoid confusion, there is no permissible state where the box can be empty.

String values

`String` objects present one exception to the field value notation; they may be displayed using either inline or reference notation. While a `String` is an object in Java, it sees extremely common usage, and is different from other objects in that it also has a literal representation. It is therefore envisaged that the majority of the time `String` objects will be displayed using the inline notation, since otherwise the prevalence of strings would mean many separate `String` objects would always need to be displayed, causing clutter and confusion in the diagram. However, in the case where an educator is specifically talking about `String` objects and

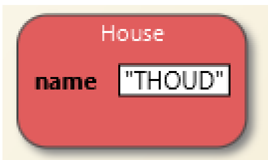


Figure 26: An object containing a string field (displayed as a literal.)

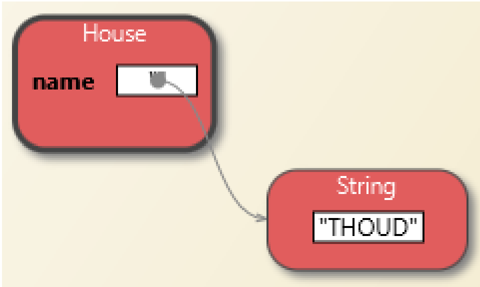


Figure 27: An object containing a string field (displayed as a separate object.)

their behaviour, the ability to draw string objects using the full notation will be desirable; this case is thus also permitted.

When `String` objects are displayed inline, the literal must be displayed within double quotes, as in Figure 26. Figure 27 shows an example of a `String` object represented in its full form.

Specifically, the above exception applies solely to `java.lang.String` objects, where their immutability and "baked-in" literal representation helps to guard against an inline representation leading to inconsistent or invalid conclusions. This must not be extended to any other objects.

Simplified view

The notation described previously is beneficial when specific fields are of interest (section 4.3.1). However, often the relationships between objects, the object graph, is the sole point of interest. In this use case, the details of *all* fields are irrelevant, and therefore would solely add clutter and noise to the object graph in question.

In this case, the fields are omitted entirely from the object diagram. Any fields with a reference value have their reference originating from the centre of the referring object; the specific field names are ignored. Fields containing the null value and fields containing an inline value are omitted entirely, since they do not contribute to the object graph. We define this as the *simplified view*. It is envisaged that

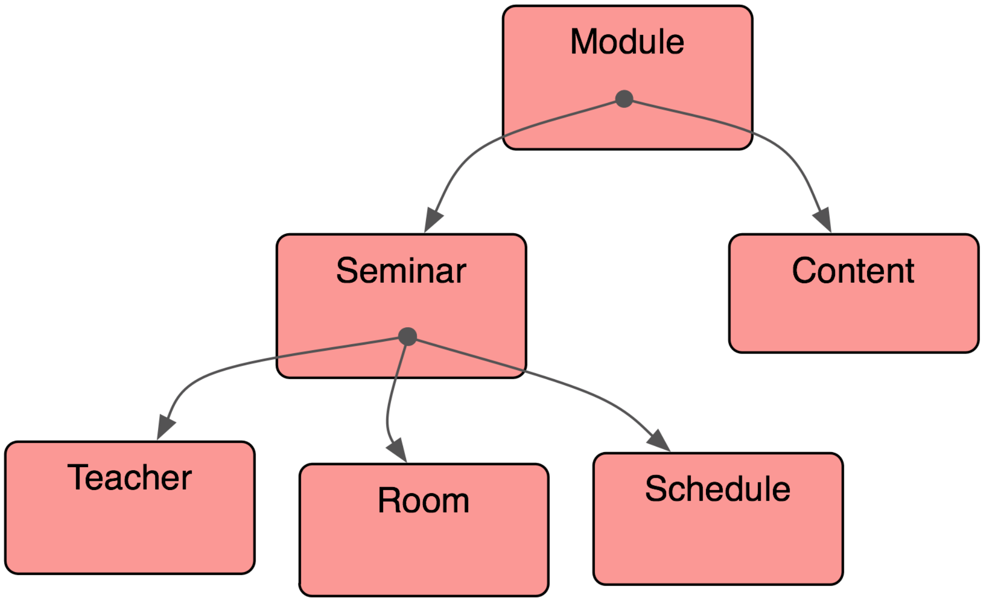


Figure 28: A simplified view of an object graph.

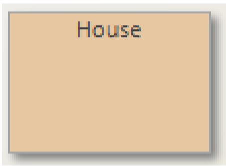


Figure 29: A class displayed using the notional machine.

the predominant use case of this specific notation will afford a state where all objects are displayed using the *simplified view* (Figure 28). However, situations may arise where the fields of a particular object, or particular set of objects are still relevant. In this scenario, it is permissible to display only a subset of objects in this state.

4.3.2 Classes

Classes are represented as rectangles with square (not rounded) corners (Figure 29), in order to make them geometrically distinctive from objects. If a fill colour is used, then they are filled with a peach colour, again following the convention from the BlueJ IDE. The name of the class is displayed as a label towards

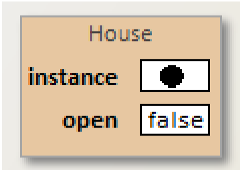


Figure 30: A class, with fields, displayed using the notional machine.

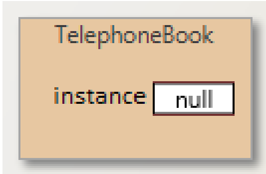


Figure 31: A class displaying a field with a null value.

the top of the rectangle, which must be identical to the source name of the class. In contrast to objects, classes *do* have unique names assigned to them, as ignoring package distinctions, the source name of the class must be unique to compile.

Fields

Fields on classes are displayed very similarly to those on objects, as described in (Section 4.3.1). The layout of the fields on objects, as well as the rules outlined previously also hold true for classes (Figure 30).

- The layout of the fields is identical (a label on the left, with a white-filled box on the right);
- Both inline and literal values are used identically to objects;
- If the value of a reference field is `null`, then the literal "null' must be used (Figure 31);
- The exception for `String` objects (which may use either notation) still applies;
- Fields not situationally relevant may be hidden from the class diagram and an ellipsis used to notify of this;

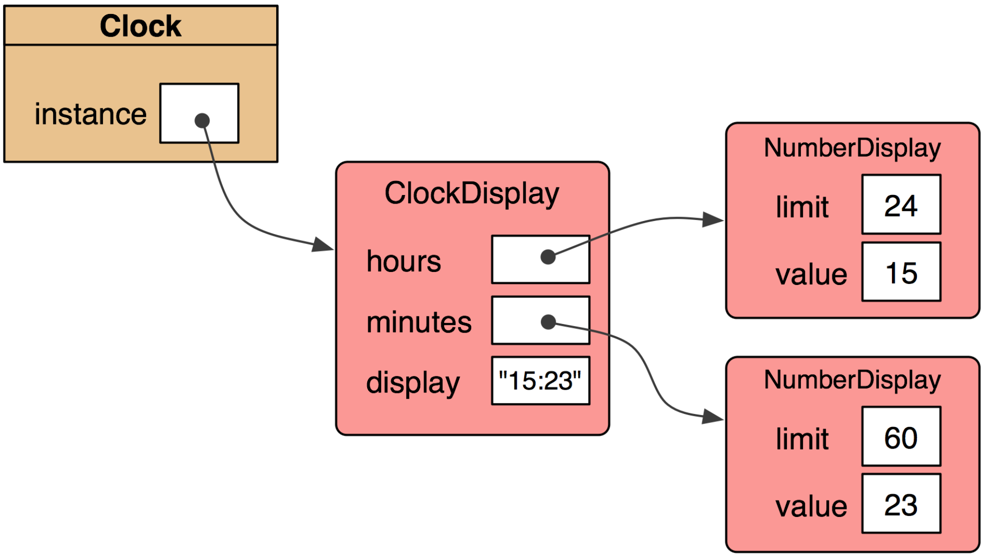


Figure 32: Representation of object references.

- Classes may also be displayed in a simplified view, using the same notation discussed previously.

The sole difference in notation is that no instance members may be displayed on a class – only static members are visualised. (Classes are the opposite of objects in this respect, which may only display instance members.)

A visualisation of a simple object graph using the notation discussed thus far may be seen in Figure 32. Note that this example represents a class, `Clock`, containing a singleton instance of `ClockDisplay`, which itself contains references to two `NumberDisplay` objects.

4.3.3 Custom views

For certain object types, the default expanded view (Section 4.3.1) does not necessarily provide a useful or appropriate level of abstraction for first year computer science students. For example, a first year student is typically not interested in the fields that reside within instances of the collection classes. These are in fact implementation details, and typically a distraction more than a help at the early level.

This is typically the case for many library classes; it is rare in first year computer science for the desired level of abstraction to be based on the implementation details of such a class. Instead, the logical behaviour of the library class commonly presents a much more important focus.

The notional machine notation therefore provides a solution for this issue by allowing the use of "custom" expanded views in place of the default field representation. In most cases we envisage them being used when discussing the object on a higher level of abstraction than that of its fields and their references. However, the views as specified here are not compulsory. If the internal implementation details of an object are indeed the primary focus, then the default expanded view showing the list of fields and their values can be shown.

Pre-defined custom views

As part of this work, we have designed custom views for a number of common cases, such as the collection classes. However, it should be noted that while these classes form an important and obvious use case for custom views as described here, they are not their sole use case. Custom views can be used for *any object* whose implementation details are not the appropriate point of abstraction for the situation at hand. Future work in this area should therefore define further custom views for other commonly used library classes.

The custom views already created as part of this work are outlined below.

4.3.4 Array

Arrays in Java present a special case, and a compelling argument in their own right for such a notation. In this scenario, it becomes impossible to outline the fields using the notation described previously – unlike other library classes, the array class is not written in the Java programming language, nor in an object-oriented

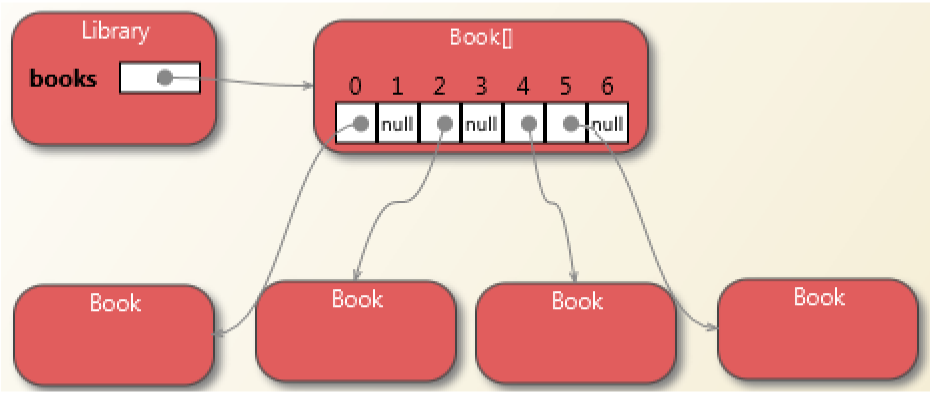


Figure 33: The custom view for arrays for reference types.

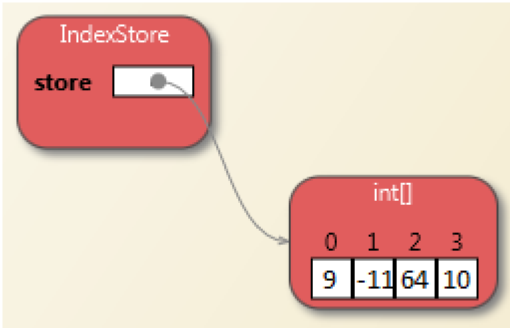


Figure 34: The custom view for arrays for literal values.

language. Showing implementation details in this case would involve exposing the student to a completely new, unfamiliar language, requiring a new set of constructs and mental models – there are no fields to show. This is clearly not the ideal case, and demonstrates a necessity for a different "expanded view" representation.

Arrays are represented with a number of "blocks", labelled from `0` to `length-1`, and these blocks can contain either reference or literal values, as described previously. When used as reference values, the blocks form potential anchor points for reference arrows to originate from. This is demonstrated in Figure 33, where an array holding a number of `Book` objects in some, but not all, of its slots is shown. The literal notation is shown in Figure 34.

In many cases, arrays will contain more elements than are feasible to display. In these cases, often a small, select range of the array is of interest. Here, a subset

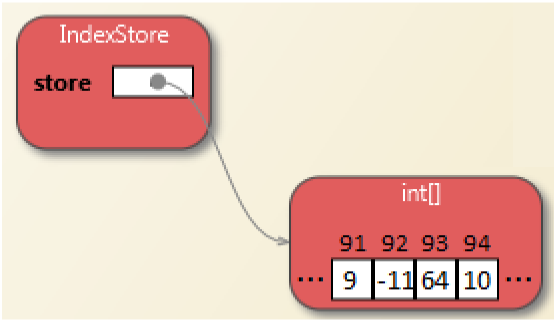


Figure 35: The custom view for only a subset of an array.

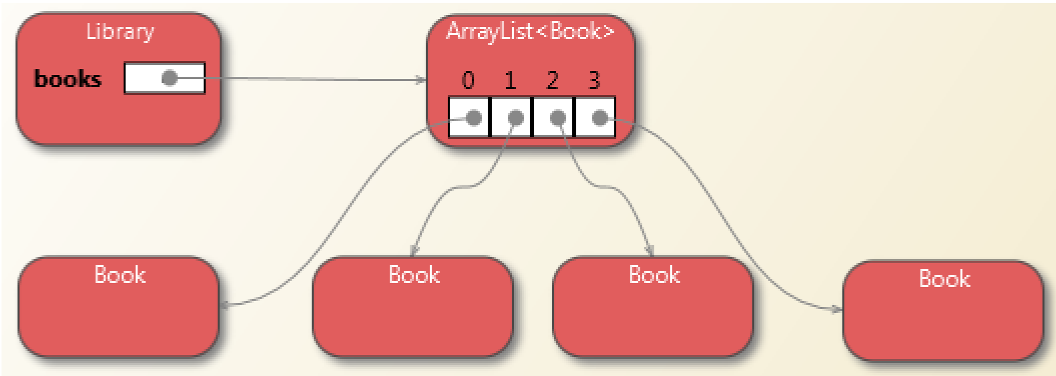


Figure 36: The custom view for `ArrayList` objects.

of the complete array may be displayed instead. In this case, ellipses should be added to the left and right of the blocks to illustrate further elements may exist (Figure 35).

As in the standard view of field values, if the array elements are primitives or strings, then the inline notation is used instead of the reference arrows.

4.3.5 ArrayList

The `ArrayList` is visualised very similarly to arrays (Section 4.3.4), with the same labelled block and reference notation described above. Literals may also be used with `ArrayList` objects; in this case (since collection classes in Java cannot be typed by primitives) the associated wrapper classes (`Integer`, `Boolean`, `Double`, etc.) can be treated the same way as their primitive forms.

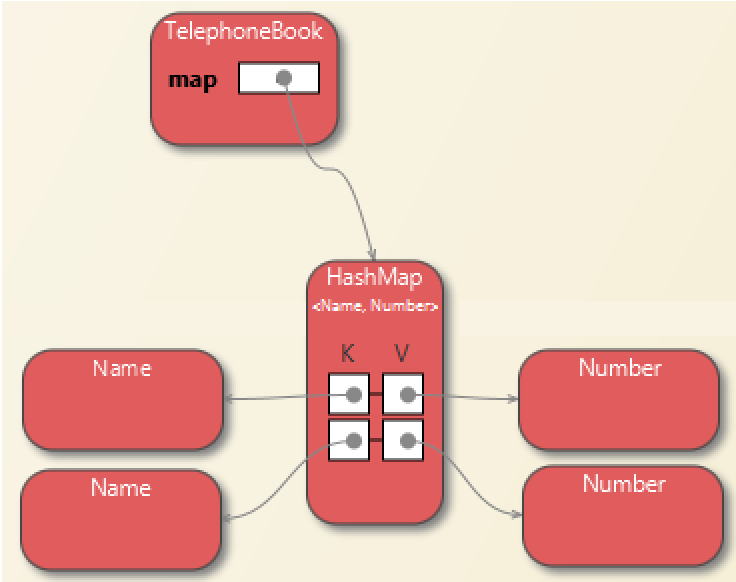


Figure 37: The custom view for `HashMap` objects.

The key differences between the notation of arrays and `ArrayList` objects lie in their dynamic nature - arrays may not expand in size, they are always fixed. `ArrayLists` however are dynamic, and so where this notional machine appears in animated form they may be visualised as such (an example implementation is described in Chapter 5).

`Vector`, a legacy collection class that existed before the introduction of `ArrayList`, should also use the exact same notation. The differences for `Vector` (and many of the other legacy collection classes) are purely to do with concurrency optimisations, and since this area is out of scope for this notional machine, they are treated as conceptually identical.

4.3.6 HashMap

`HashMap` objects are represented by a list of key-value pairs, again with the same labelled block and reference notation (Figure 37). The list of key-value pairs are deliberately not numbered to reinforce the fact that their order is arbitrary, and

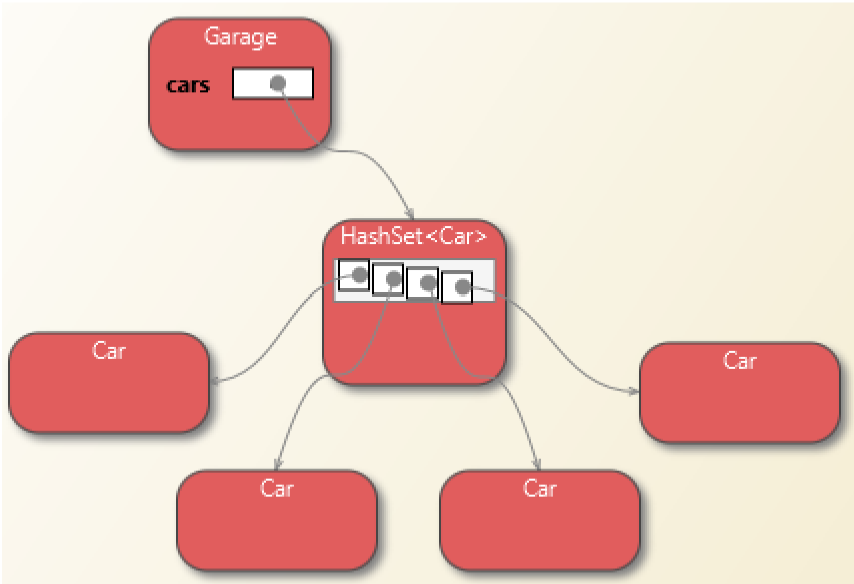


Figure 38: The custom view for `HashSet` objects.

not necessarily consistent.

`HashTable`, the equivalent legacy collection class, can also use this notation (for the same reasons discussed in Section 4.3.5.)

4.3.7 HashSet

`HashSet` objects are represented by a number of blocks arranged in an arbitrary fashion inside the expanded object (Figure 38). Again, they are deliberately left un-numbered and are not aligned to reinforce the conceptual lack of ordering within a `HashSet`.

4.3.8 String

The custom expanded view for `String` objects simply displays the equivalent literal, enclosed by double quotes, in a box in the middle of the rectangle (Figure 39.) While this is a very simple custom representation, it serves to simply show what the

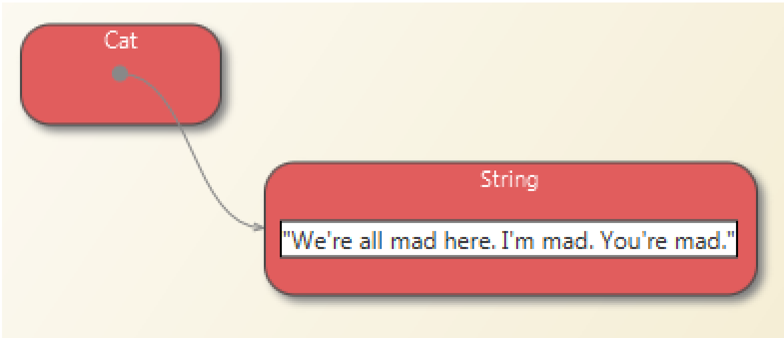


Figure 39: The custom view for `String` objects, just showing the literal value of the string rather than its internal fields.

string object currently holds, and hide the internal details of how it is stored.

4.3.9 Additional custom views

Wherever the fields of a library class are not the most helpful level of abstraction, educators are encouraged to devise their own custom views. This need not be simply for collection classes; `Date`, for example, is a non-collection class for which a custom view (perhaps showing just a textual representation of the date and time) would be useful. Arguably all but the most simple library classes should be represented using a custom view, as the fields are implementation details and not usually relevant to those studying CS1.

An API is provided in order to facilitate custom views - educators need just extend a `Drawer` class, then register it with a particular class type. This class contains a single method that is called with the Java object to be represented as a parameter, and returns a JavaFX `Node`. This `Node` will then be shown on the object in its expanded view. Utility methods are also provided to draw the white boxes and text commonly seen on other custom views above, in order to provide a consistent look and feel amongst custom views.

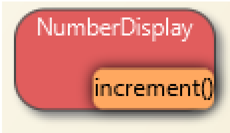


Figure 40: A method executing on an object.

Figure 41: A method executing on an class.

4.4 Execution

In addition to the object diagram, the notional machine has been designed to optionally represent the execution state of a program. Traditionally, this information is displayed in the form of a stack visualisation in many systems.

Many of the diagrams shown in this section are screenshots of Novis, the notional machine implementation. These may well be harder to grasp in static form than in implemented form, since the screenshots must necessarily abandon the animated elements of the implementation. Where static diagrams are drawn for use in an educational environment, we recommend that arrowheads in particular are clearly drawn (since these are used to show the order of execution.)

4.4.1 Method invocation

Active method calls are represented by smaller rectangles with rounded corners, which, in the case of instance methods, are displayed attached to the bottom right of the object that they are called on (Figure 40). In the case of static methods, they are displayed attached to the bottom right of the class that they are called on (Figure 41). As discussed previously, fill colour is optional. However, if colour is used the rectangle is filled orange. The label remains visible for as long as the method remains on the stack, even if it calls another method (and is thus no longer

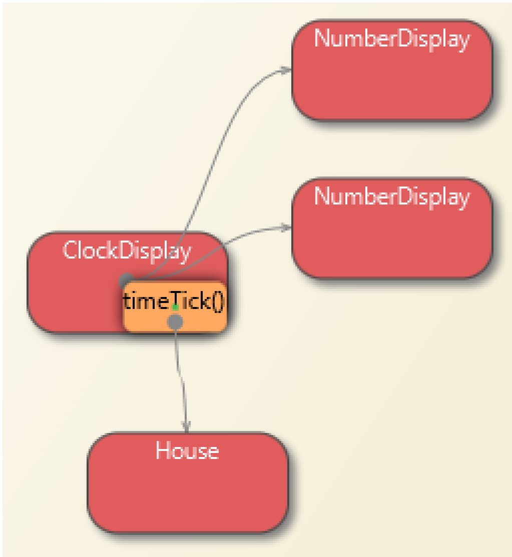


Figure 42: A reference to an object from a local variable.

the top most method in the stack.)

References from local variables may be shown originating from the method label, below the method name (Figure 42). As with objects, in the case where the actual reference is not of interest, the reference may simply be depicted with a dot (without the corresponding reference arrow.)

Nested method invocation

It is rare for execution to just be contained within a single method, the vast majority of examples will see nested method execution – that is, a method which calls another method. If a method in turn calls a further method on another object, then that method will be added to that object in the same way as described above – as a peach coloured rectangle attached to the bottom right of the object (Figure 43). If a method is called on the same object (or if a method is called on an object which already has a method on the stack) then the new method is simply added below the existing one (Figure 44). Further method labels may thus appear attached to the same object or other objects, depending on the receiver of the method call.

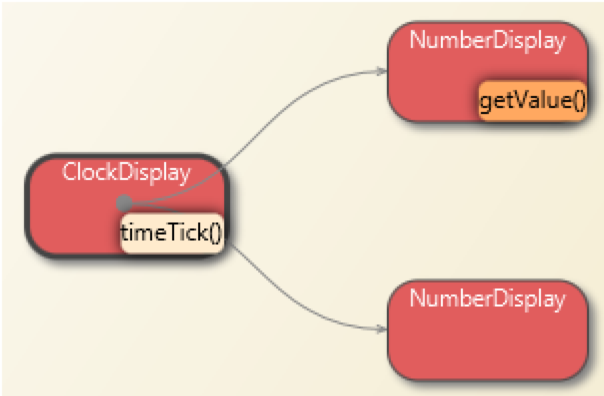


Figure 43: A method executing on an object, which has in turn called another method on a separate object.

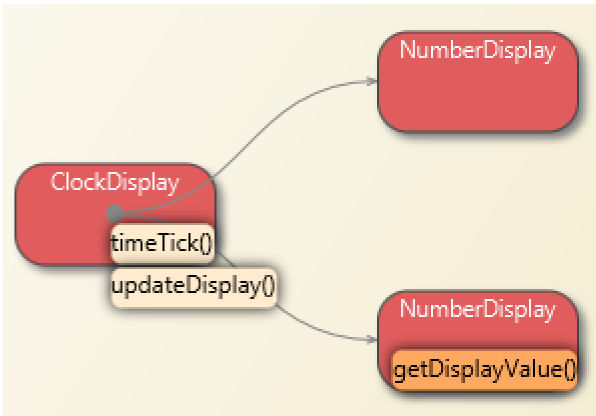


Figure 44: Multiple methods executing between objects.

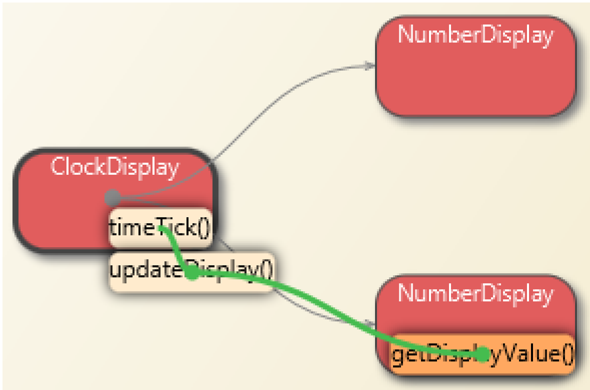


Figure 45: The call chain, showing the specific order of methods executed.

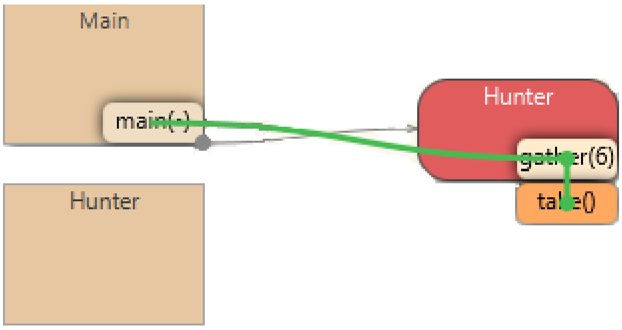


Figure 46: The call chain shown between methods on a class and object.

Optionally, the currently executing method (that is, the method at the top of the stack) may be coloured in a deeper shade of orange than other methods on the diagram, as a quick visual cue to indicate where the path of execution currently lies (Figure 43).

4.4.2 Call chain

In many simple examples, it may be possible to work out the execution order simply by studying the diagram. However, for most examples a more detailed display is needed to visualise the order of execution. This is drawn in the form of a call chain (Figure 45). The display of the call chain is optional. While the order

Figure 47: A method called with two parameters.

of execution of the diagram is often an interesting point of discussion, this is not necessarily always the case (and in trivial examples, the order of execution may be obvious.)

The call chain arrow (or, in talking about a notional machine diagram, usually just “call chain”) is an arrow which travels through all the methods currently on the diagram, from the first to the last method executed – traditionally, from the bottom of the stack to the top. If colour is used, then it must be filled in green. The call chain depicts the complete current sequence of open method calls, their dependencies and their order. The call chain can travel between static methods on classes and instance methods on objects (Figure 46).

The call chain updates every time the stack updates, however this does not restrict the use of a call chain to implemented forms of the notional machine. When using this notation in a textbook for example, several diagrams could be displayed in a “timeline” like format to show its progression between one state and the next. When used on a whiteboard, the call chain is often extended to show nested method calls, and wiped out again to illustrate the completion of a particular method invocation.

4.4.3 Parameters

When methods with parameters are called, the method labels must include the list of actual parameters (Figure 47). For primitives and strings, the parameter value is shown as a literal; for object parameters, a reference to the object is shown originating from the parameter list. As with similar notation described previously, if the object reference is not of use in the current scenario then the specific reference may be omitted. In this case, a “dot” would simply be shown, indicating that a

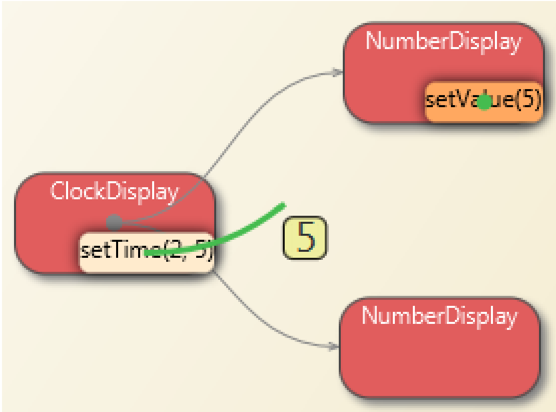


Figure 48: Parameter passing being visualised alongside the call chain.

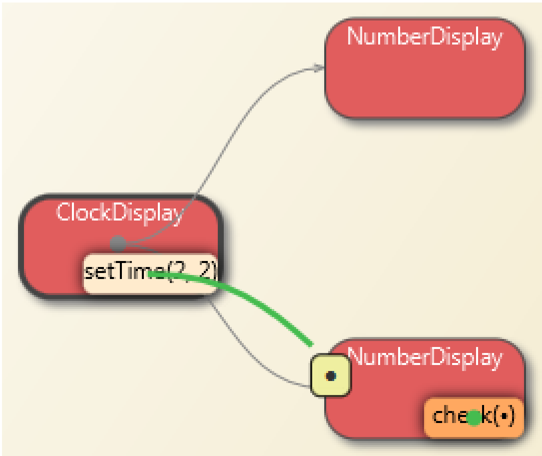


Figure 49: An object parameter being visualised alongside the call chain.

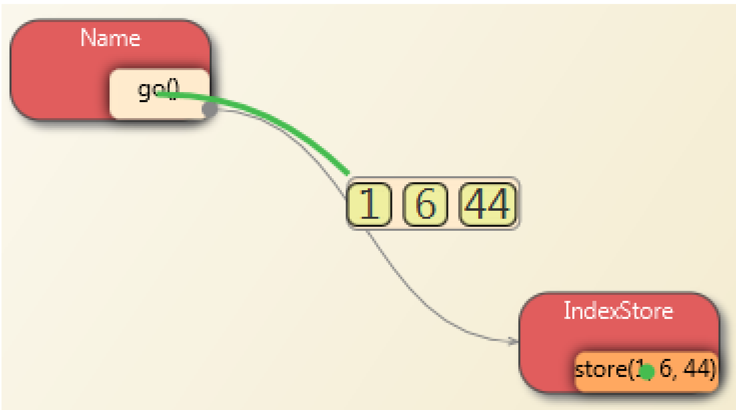


Figure 50: A method executing with multiple parameters.

reference is present (but not displayed in full.)

In an animated implementation, parameters may also be visualised passing “along” the call chain. In this scenario, the parameter should be displayed in a rectangle with rounded corners, and positioned somewhere along the line of the relevant part of the call chain (that is, the part between the method which is being called with parameters on itself, and the method which is calling it.) A fill colour is once again optional, however if used then the rectangle must be filled in yellow (Figure 48). It may be helpful to display an arrow on this specific part of the call chain, to indicate its direction (and thus avoid confusion with return values, discussed below.)

If multiple parameters are passed to a method, then the parameter rectangles should be placed alongside each other, and then within another rectangle with rounded corners (Figure 50). If fill colour is used, then a slightly darker shade of yellow than that used for the parameter boxes is recommended.

The notation used to display the specific parameters being passed is identical to that of fields – primitives should be written out as a literal notation, and references displayed as a “dot” with an arrow pointing to the object being referred to. Once again, if the object being referred to is not of use in the current scenario, then the specific reference may be omitted (Figure 49). To follow the convention set out for the display of fields, `String` objects may use either the inline or reference notation.

4.4.4 Return values

When a method with a return value is called, its return value may be optionally visualised being passed along the call chain, similarly to parameters (though obviously in this case, the value will be passed in the other direction.) In the case where the return value is displayed, this should be in a rectangle with rounded corners, positioned somewhere along the line of the relevant part of the call chain. A fill colour is optional; if used then the rectangle must again be filled in yellow. Once again, an arrow on this specific part of the call chain may be useful to avoid

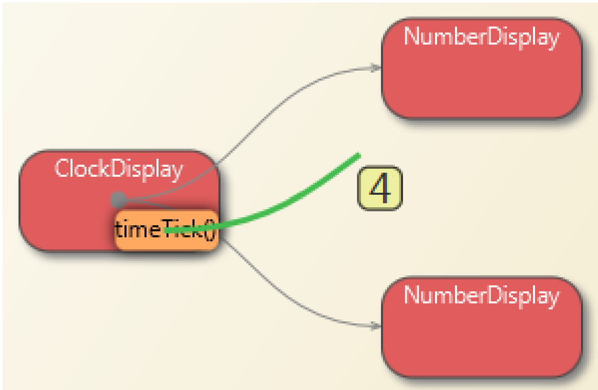


Figure 51: A visualisation of a return value from a method.

ambiguity with single parameter passing (as described above.)

4.4.5 Execution example

Figure 52 demonstrates a more complete example of the execution notation described above, showing a representation of a simple program. It demonstrates the display of the call chain, local variables (in the `getFirstName()` method) and inline and reference parameters (in the `getName()` and `getNames()` methods respectively).

4.4.6 Advantages over traditional methods

The traditional method of depicting the execution state is via a stack diagram. In this notation, the list of source code locations currently on the stack is shown to the user, usually via a textual representation. While this is widely used, it is far from ideal. Our notation, as described above, has several novel advantages over this former representation.

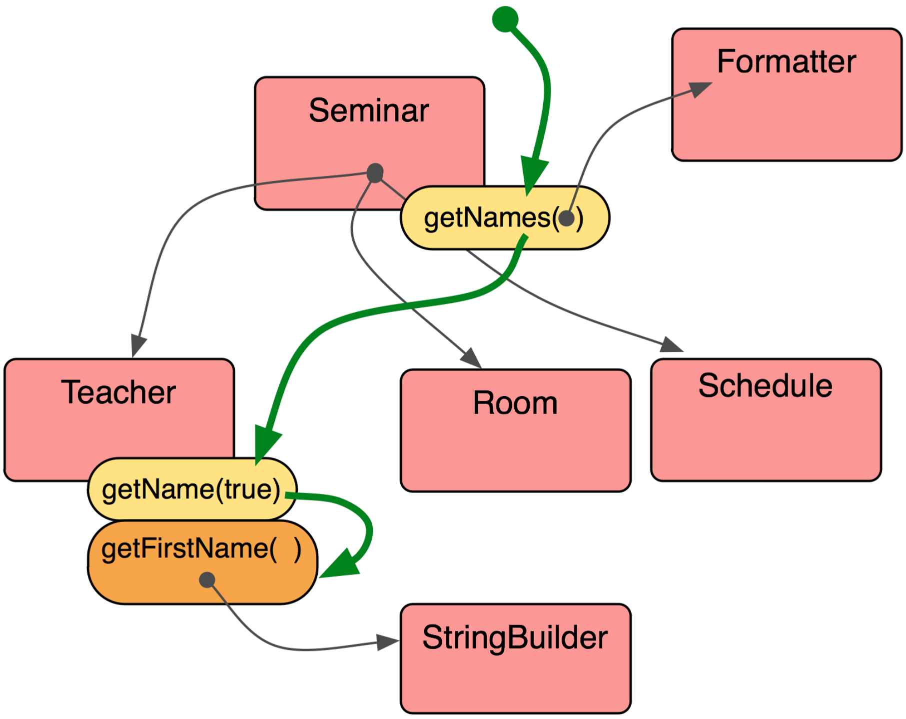


Figure 52: An active method call chain with object parameters.

Single diagram

The execution state at any given point in time has previously been most often depicted separately from the object diagram, either in textual representation, or a distinct diagrammatic form of a call stack. Regardless as to whether the representation is textual or diagrammatic, this requires the student to recall two distinct and separate diagrams, then mentally combine them, resulting in an increased cognitive load. This is also a best case scenario - when this process is executed by novice programmers, they may not mentally combine these entities correctly at all, causing yet further confusion and reinforcing incorrect mental models.

In our notation, we solve this issue by using just one diagram to represent both the execution state and the object diagram. This eliminates both the need to recall the diagrams, as well as the mental overhead of combining them.

Object level information

The current object is usually identified by including a "this" pointer (a local variable with a reference to the current object) on the stack. While this is a convention followed in many tools, it affords a mental model where the stack is a distinct entity that references objects. While this is not necessarily an incorrect model, it does place increased extraneous cognitive load on the user (since it requires the understanding of the stack as a separate data structure.) The use of the single diagram to unify the stack with the object graph (section 4.4.6) instead contextualises the call chain with object level information.

Affordance of animation

The stack trace does not easily lend itself to effectively representing changes in state. At any point in time it is simply a textual list, that must be re-written for each new state. By contrast, our notation affords a visualisation that can easily and effectively be updated in real time, both in implementation form, and on a white board with a pen and eraser. The changes in state also lend themselves well to animation, facilitating an electronic implementation in this regard.

Ease of hand drawing

While traditional methods are often easily represented electronically, they can take a long time to draw or write out by hand, effectively rendering them useless for quick, ad-hoc representation (such as an educator drawing on a whiteboard.) A traditional stack trace consisting of source files and line numbers would certainly be infeasible to draw on a whiteboard.

In contrast, the union of the object graph with the call chain used in our notation allows quick, effective ad-hoc diagrams to be drawn easily. This makes them much more feasible for educators to draw as part of a class, lecture or seminar.

4.5 Discussion

While visualisations and mental models already exist to represent the execution of object-oriented programs (Section 2.12), the design of the notional machine outlined above is, to the best knowledge of the author, unique in a number of areas.

4.5.1 Single diagram

Many notional machines already exist that visualise the execution of a program in a number of distinct visual areas, usually the heap, stack and call chain – UUhistle and Jeliot, arguably the two systems most similar to this work, both fall into this category. In the case of many object-oriented visualisations, classes and objects are also commonly visualised separately, further providing a visual distinction.

To the best of our knowledge, the notional machine defined in this work is unique in that it can provide the same level of information as all of these distinct views, but does so on the same diagram. There is no component of the notional machine that necessitates the recall of a separate diagram – everything can be drawn in the same place. This includes the addition of the call chain replacing the traditional "stack trace" (Section 4.4.)

4.5.2 Scalability

The notional machines that have been outlined in the literature review (Chapter 2) are all set to a certain "scale" – for the most part, this means that they are fixed to work on a line by line basis, highlighting each individual line as it is executed, then visualising the execution of this line. In some cases the level of detail shown is even finer grained; both UUhistle and Jeliot can display statement based visualisations. This can work well with very small programs of only a few lines each. However, when more complex programs are introduced, this model of execution becomes much too detailed to be useful. This means that the student would typically only spend a very short time making good use of the notional machine, and would tend to outgrow it rather quickly (as soon as they progress beyond programs of just a few lines.)

Some visualisations do exist that are targeted towards expert users in various capacities. A small selection of these have also been briefly explored in Chapter 2. However, these particular visualisations are also fixed in their scale, presenting the opposite problem to the target user group – they become much to complicated and specialist for first year computer science students.

The notional machine presented here aims to alleviate this problem by providing a large range of scale. For example, if the details of a particular object are not important then it can be shown in the simplified view, hiding its fields and providing a bare-bones, basic representation. If the object's details are relevant to the task at hand however, then it can be shown in its expanded view, where all its state is visible. Likewise, library objects that are not of concern need not be visualised at all, or if they do form points of interest then they can be displayed with a much more appropriate custom visualisation. The display of execution can also be customised; the call chain need only be displayed if it forms a relevant part of the example. In short, any part of the notional machine not of direct interest can simply be hidden or scaled down, ensuring the notional machine has enough scope to be used for a relatively long period of time (throughout the first year of a university computer science course in this case.)

4.5.3 Versatility

The notional machine as outlined above can be used in a number of formats.

The notation fits well into an implementation, since it could easily provide an animated form of the visualisation that updates in real time. This could either be implemented as a live or post-mortem visualisation (Section 2.11.5). An implementation could also provide an easy mechanism for allowing the user to view objects in the expanded or simplified format, and pick an appropriate level of abstraction to show to the user based on the program currently executing.

However, while the notation lends itself well to an implementation form, this is by no means a requirement. It can equally be represented in an "unplugged" format, such as by an educator drawing on a whiteboard or a static diagram in a textbook. While geometrically distinctive, the shapes are easy to draw by hand (rectangles, and rectangles with rounded corners) and the use of colour has deliberately been outlined as optional (though the colours chosen are readily available in whiteboard pens.) This "unplugged" format of the notional machine may even provide a limited form of animation with some examples. While this will not be possible to the same extent as an implementation, a textbook for example could show a series of diagrams throughout the execution of a short program, or short part of a program utilising a timeline style view. In addition, an educator drawing on a whiteboard could easily depict an overlay of a call chain, and then "execute" the program by rubbing off the old arrows, and drawing on new ones at each step.

4.6 Summary

Now the design of the notional machine has been outlined, a summary can be presented in line with the table shown in section 2.14.

4.6.1 Granularity

Multiple levels of granularity are possible, both in terms of object representation and execution. Objects may be represented with or without their fields, or with a subset of fields relevant in any particular circumstance. The level of detail represented in execution is also variable – where detail is required, methods may be shown in full, with a call chain, and with animated parameter passing and return values. However, they can also be shown without a call chain, or omitted entirely if execution detail is not relevant.

4.6.2 Interactivity

The system is highly interactive in the sense that it is highly customisable. The user can choose the speed of execution as well as the fields and methods that are shown on each particular object. However, it does not mandate user intervention as some tools do through the use of forced execution or stop and think questions (Naps, 2005).

4.6.3 Visualisation type

While algorithms could reasonably be displayed using the notional machine (as in other similar visualisation systems), it is a program visualisation rather than an algorithmic visualisation. This allows a user to execute any Java program within the problem space and view the corresponding visualisation. This does not prevent the notional machine from being used for any type of algorithm visualisation however – there is no reason why a Java program that demonstrated the operation of linked lists (for example) could not be written and executed using the notional machine.

4.6.4 Snapshot / Continuous

Either snapshot or continuous representations can be used. In an implementation the visualisation would be continuous, making use of animation to represent the sequence of states in a live running programm. The notional machine is not restricted to this however; it can also be represented in snapshot form in a textbook or on a whiteboard. In this format, individual states of execution can be shown, or multiple states can be shown using a sequence of images shown as a timeline.

4.6.5 Post-mortem / Live

The notional machine as defined above could be implemented in either post-mortem or live form.

However, the implementation performed as part of this work (section 5) is a live visualisation. While this makes some parts of the visualisation harder (such as the layout algorithm, since the objects that need to be laid out are not known in advance), it means that the output of the program (whether on a terminal, printer or some other means) is synchronised to the object graph.

4.6.6 2D / 3D

The notional machine is purely two dimensional, no capability is defined for three dimensional notation. Any benefit this could have (in terms of showing more information in the model at once) is quickly overshadowed by the additional complexity required to navigate such a space. This problem is compounded since the intended users are novices – first year computer science students.

Chapter 5

Implementation Design

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. This chapter will examine the design decisions necessary in creating an implemented form of the notional machine notation as described in the previous chapter.

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 (Kölling et al., 2003). The design of BlueJ is based around Blue (Kölling, 1999b), 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" (Utting et al., 2012).

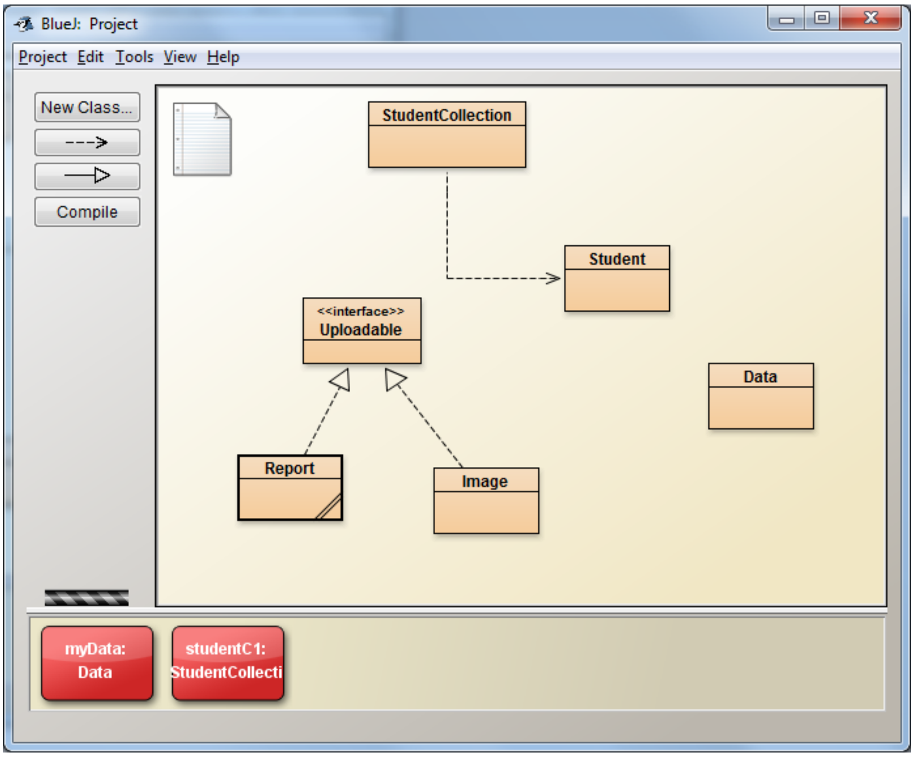


Figure 53: BlueJ's main interface.

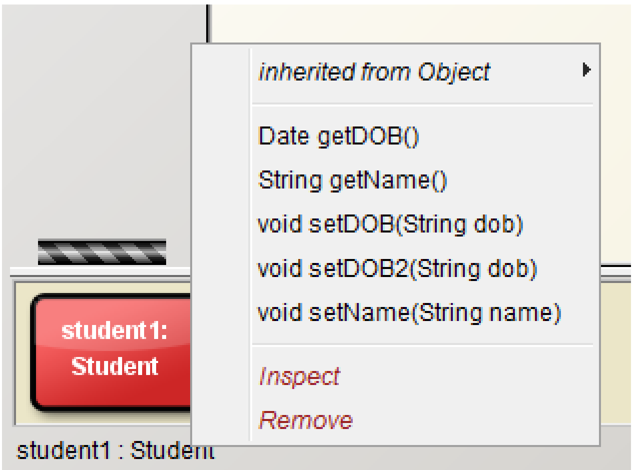


Figure 54: Calling a method on an object in BlueJ's object bench.

5.1.1 Class diagram

The majority of BlueJ's main interface (Figure 53) 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.

5.1.2 Object Bench

BlueJ also contains an object bench (Figure 53). Constructors of the classes in the class diagram can be interactively invoked, creating an instance of the class on the object bench. Methods can also be invoked on these objects by right-clicking on them. This displays a context menu which lists the available methods by name, including their parameters and return types (Figure 54). The object bench is useful for creating a few objects, executing their methods and examining the result.

5.1.3 Limitations of BlueJ

These current elements provide limited functionality with respect to dynamic visualisation. The class diagram only shows compile time relationships. This is arguably useful when designing the class structure of code, since it can provide a clear, visual layout of class relationships. However, it does not provide any additional information at runtime.

Some object level visualisation is available in terms of the object bench, however this simply shows instances of user created objects that can be interactively manipulated. It has a number of limitations when attempting to use it to explain a running program:

- It provides the user-created object with a visible unique name, potentially reinforcing misconceptions that objects in Java are discoverable and addressable via this method.

- It does not visualise the state of the object, it is not possible to see its fields or the value of those fields.
- The object display is incomplete. If any other objects are created as a result of instantiating the user-created object, they are not visualised.
- The state of the stack is not visualised – the only action visible to the user when executing a method call is the dialog displaying the result (if a method returns a value.)

5.1.4 Relationship to BlueJ

Examining the notional machine in relation to the conceptual model currently used in BlueJ would categorise it as a replacement of the object bench. It is, in effect, a much more comprehensive form of this that includes the pre-existing functionality (adding and removing objects and calling their methods) as well as providing much more visual information at runtime (visualising the creation of objects in user code, the changing references between objects, garbage collection of objects, and the chain of execution.) However, establishing the notional machine notation will require much more space than the object bench currently provides.

Novis thus provides a new view that utilises the real estate previously occupied by the class diagram and object bench. The old functionality is still present, but reimplemented for three major reasons:

- The visualisation can be easily switched between showing the static class relationships, or the dynamic object relationships – it can thus emulate the behaviour of the static class diagram if desired. Additionally, the static class diagram has very limited use at runtime since it does not show any updated state – while code is running it is generally far more desirable to view the current state and execution thread of the various *objects*, not just the static relationships between *classes*.
- The pre-existing object bench occupies just a small percentage of the window. While this is adequate for interactively creating a handful of objects,

automatically visualising every object and their interactions between each other requires much more real estate to show useful information.

- References may exist from classes to objects (in the case of static references), therefore the new diagram must include both of these components.

5.2 Class diagram

Novis depicts the static view of classes and their references between them in a very similar way to BlueJ (Figure 55). 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.

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 56). 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. The details of how Novis searches for an appropriate position in the diagram are covered later in Section 5.9.

Initially, the object is displayed grey in colour to indicate that it has not been

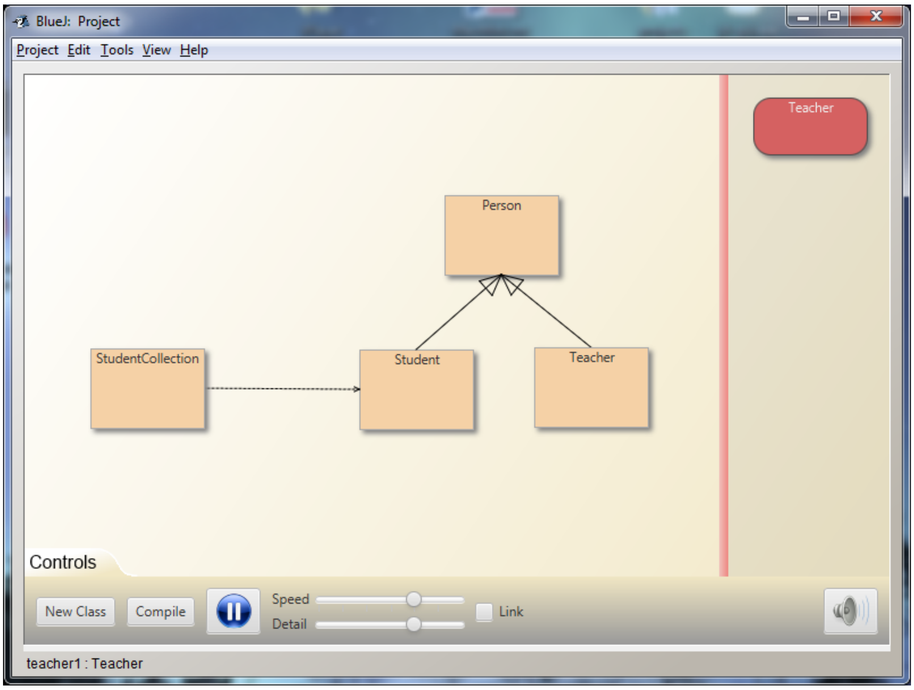


Figure 55: The class diagram shown in Novis.

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 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.

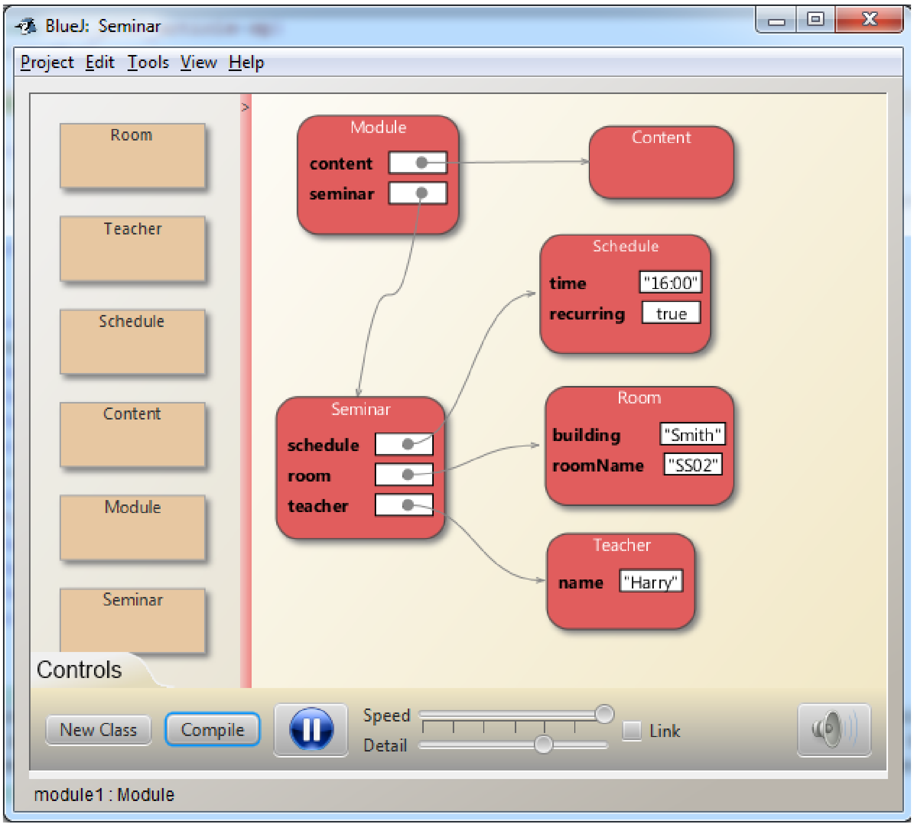


Figure 56: The Novis notional machine visualiser with a simple example.

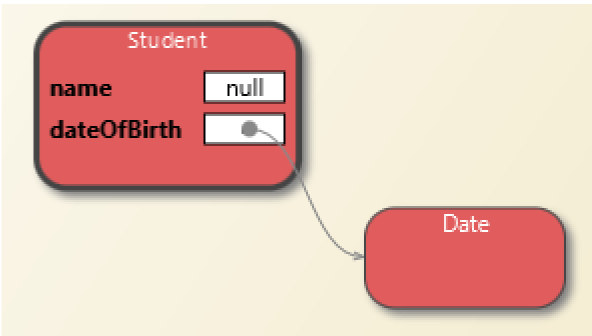


Figure 57: The expanded view of an object.

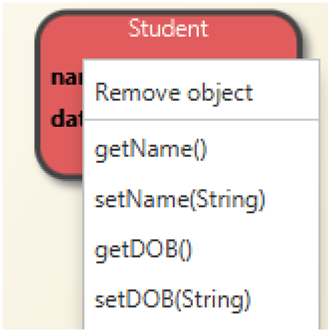


Figure 58: Calling a method.

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 object's fields however, this can be done by clicking on the object. The notation for displaying the fields is as described in Section 4.3.1 – 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 57). The field value may be an inline value or a reference. In either case, the values of these 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. Figure 58 shows a user created object, hence the addition of the "Remove object" option. Any of these methods may be executed by clicking on the method; 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 (Figure 59).

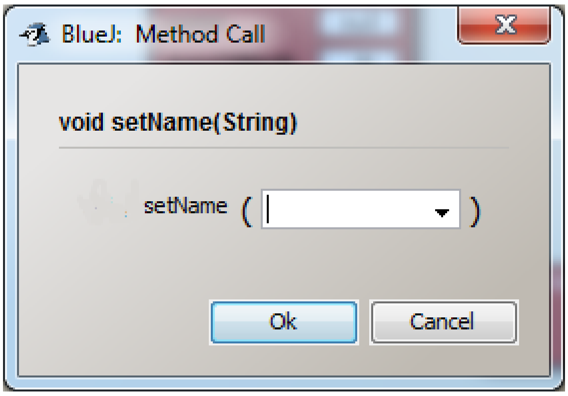


Figure 59: Providing a parameter to a method call.

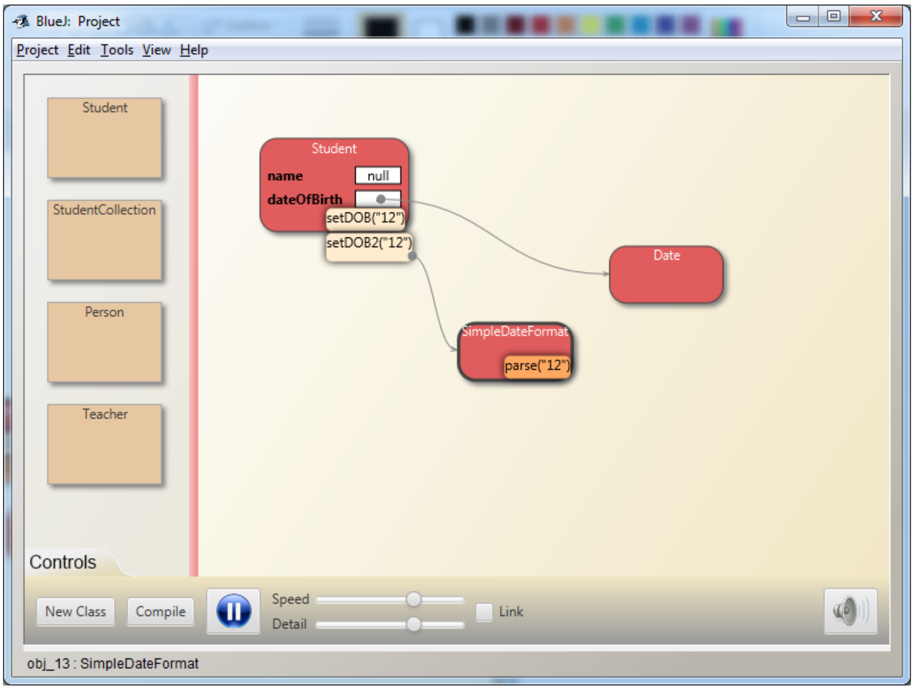


Figure 60: Novis visualising the execution of method calls.

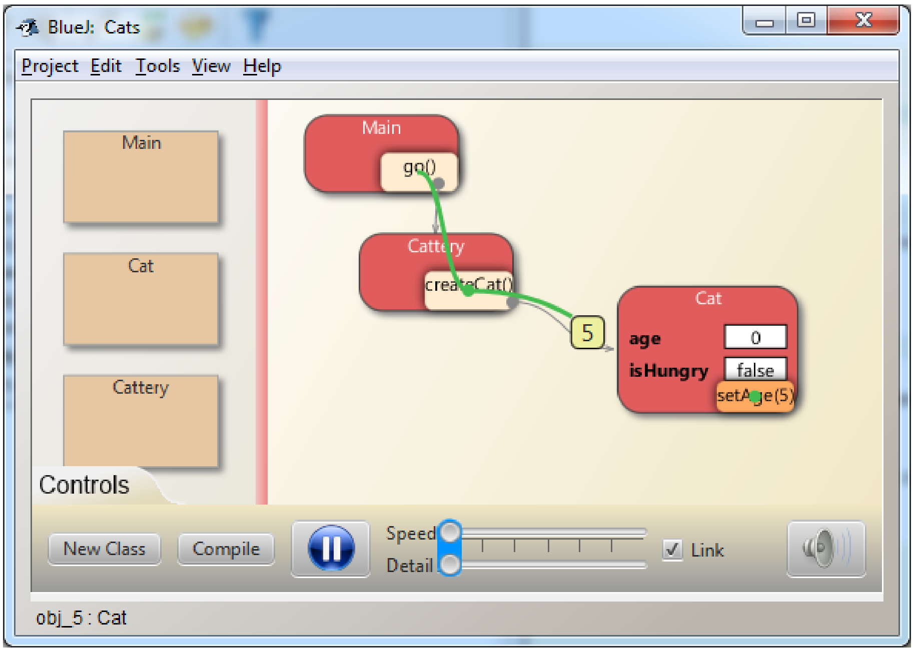


Figure 61: Novis displaying a method call chain, passing a parameter.

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 60).

5.7 Call chain

As discussed previously, the stack is often represented as a distinct visualisation from the object heap, violating the synchronization principle (Section 2.8.8). Novis unifies the traditionally separate stack diagram with the object heap view, displaying a trace over the methods in currently active stack frames (Figure 61). This trace dynamically updates as the state of the stack changes. The topmost

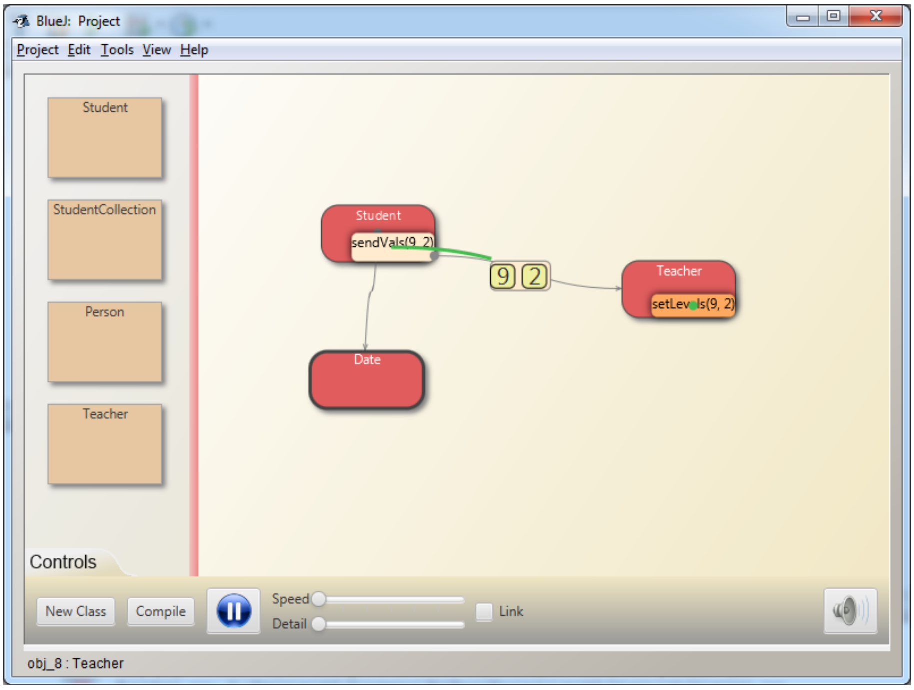


Figure 62: Novis visualising the passing of two parameters.

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 61 also shows a visualisation of a parameter being passed. Parameters are shown in boxes that first appear 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 62).

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 Layout

Layout is one of the known hard problems in systems of this kind for a number of reasons. Simply ensuring objects do not overlap would be trivial, but they must be displayed in a clear, consistent manner. With hand drawn diagrams, this is done instinctively and relatively easily, since it is anticipated the user will have a rough idea of the type of structure and quantity of objects being created. They can thus anticipate this and use their available space appropriately. In contrast, a machine based implementation does not posses this knowledge ahead of time – this is a disadvantage of live visualisations discussed previously (Section 2.11.5). Since Novis is a live visualisation (as oppose to a post-mortem visualisation), the layout presents a significant challenge.

5.9.1 Criteria

The layout algorithm used in Novis must fulfil the following set of criteria to be effective:

Minimal arrow crossings The node size is used when calculating the layout constraints, and it gracefully handles changes in object size without shifting their positions.

Fast The layout algorithm cannot unreasonably slow the execution of the notional machine.

Consistent The same objects placed in the same order should yield the same layout.

Scalable The algorithm should be able to perform effectively throughout the range of objects Novis is designed to handle (hundreds of objects.)

5.9.2 Existing layout algorithms

A number of existing algorithms were tested for their suitability for use in Novis by matching them to the above criteria.

Circular layout

Circular layout algorithms are simple and fast, simply having to find a sufficient space along a circular arc to place a new node. However, circular diagrams do not inherently lend themselves to large numbers of objects in a 2D space, and when there are hundreds of objects it becomes very difficult to reduce the number of arrow crossings.

Radial layout

Radial layouts are an extension of circular layouts; nodes are placed on several arcs that radiate from a centre point (Yee et al., 2001). While more computationally expensive than circular layouts, they are able to minimise arrow crossings to a greater degree. They are also able to scale to a greater number of objects for any given 2D plane.

Force based layout

A force based layout (Fruchterman and Reingold, 1991) models the layout of the graph by using a 2D physics engine; the nodes repel each other but graph edges tie them together. The emergent behaviour thus tends towards a graph with shorter edges, and fewer crossings. Initially this algorithm showed promise with small examples and was implemented in an early prototype. However, after extensive development and testing this could not be made to work well for the following reasons:

- The layout was not consistent. The individual, atomic movements in a force directed graph can be extremely small, meaning they are very easily influenced by the addition of new objects at slightly differing times. Since it is very unlikely that the placement of all objects will be at the same instant in time for each run of the same program, very different graphs were often produced.
- As the velocity of a force directed graph approaches zero, rounding errors are eventually introduced that can cause the graph nodes (in this case, the objects) to behave erratically, often jolting around a few times before fully coming to rest. This forces unwanted animation on objects, diverting the user's attention to typically uninteresting areas of the visualisation.
- The force directed algorithm (and indeed, most graph layout algorithms) is not aware of the size of their nodes, meaning that any adjustment of size would need to be done independently of the layout algorithm itself. This would, without careful alteration, result in the algorithm rapidly jumping from its stable state to a further unstable state.
- Force directed algorithms generally run continuously in the background, meaning they can easily become very resource intensive to run in real-time with graphs containing hundreds of nodes.
- The algorithm does not scale well from a performance perspective. While it performs adequately with a few objects, it starts to become increasingly intensive when moving into more than around ten objects. This is well below the number of objects that Novis is required to handle.

Wire routing algorithms

The layout of components on circuit boards requires detailed layout algorithms, and many such algorithms have been in development for many years (Frick et al., 1995; Goto, 1981; Yan and Wong, 2010). These algorithms must deal with many more interconnected nodes than the notional machine described here, as well as minimising crossings.

However, such algorithms are not designed to run quickly or in real time. On the contrary, they might often take many minutes to complete for a single graph – they are thus unsuitable for use in Novis. This is reflective of the tools for which they are designed - when designing a circuit board, a schematic is first drawn where the layout is determined entirely by the user. The algorithm only runs once the schematic is complete, thus it is of little consequence if it is computationally expensive.

Summary

Since no algorithm could be found that provided an adequate layout for the visualisation, a new, custom algorithm was developed. The algorithm fulfills the criteria set out in Section 5.9.1:

Minimal arrow crossings The algorithm exploits the fact objects are often linked in close proximity to minimise arrow crossings.

Fast The layout algorithm is fast, only needing to make a few calculations each time an object is added or a reference is changed.

Consistent The algorithm is deterministic - the same objects placed in the same order yield the exact same layout.

Scalable The algorithm is able to shrink and move existing objects where required, thus easily scaling into hundreds of objects with no noticeable issues.

5.9.3 Layout Algorithm description

Basic operation

The visualisation area is divided up into a uniform, virtual grid (these grid lines are not shown to the user.) Every cell in the grid always has the same width

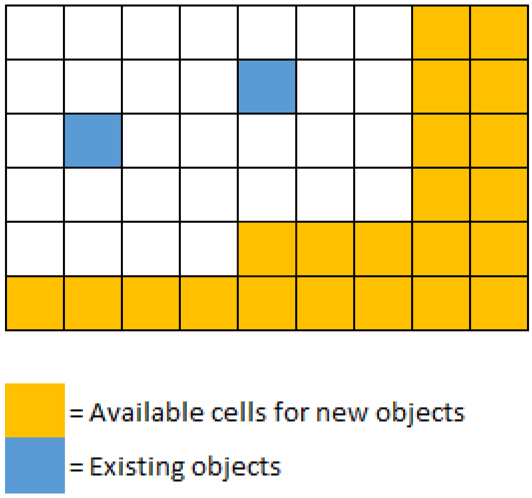


Figure 63: The set of cells in which orphan objects can be placed.

and height (that is, the cells must be of uniform size, they need not be square.) Initially, this is a default width and height that can comfortably house the object, in its collapsed state, with a border. The first object placed in the diagram is always placed in the centre cell.

The placement of further objects on the graph is dependant on whether, at the time of placement, a reference exists from any other object that is already placed on the graph. If this is not the case, then this object is denoted as an *orphan object*. The grid is searched in an arbitrary, but repeatable (non-random) order for a free cell at least three cells away from all other occupied cells in each direction (Figure 63). If found, the orphan object is placed in this cell.

In the case where an object is referred to by another object, a specific set of cells (defined as the *referent cell set*) around this *existing parent* are examined, in order, until a free cell is found (Figure 64.) An alternate example with space for real objects is shown in Figure 65. If this existing parent is itself referred to by another object (the *existing grandparent*), the search ordering may be changed to favour free cells that would not cause a reference to point back to the existing grandparent (Figure 66).

In addition to placement of new objects, the layout is also updated when a reference

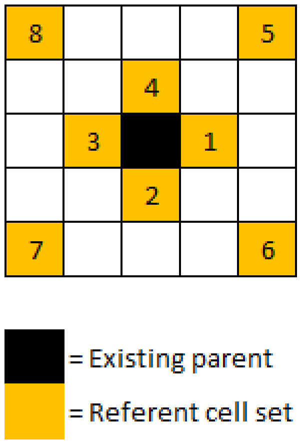


Figure 64: The initial referent cell set.

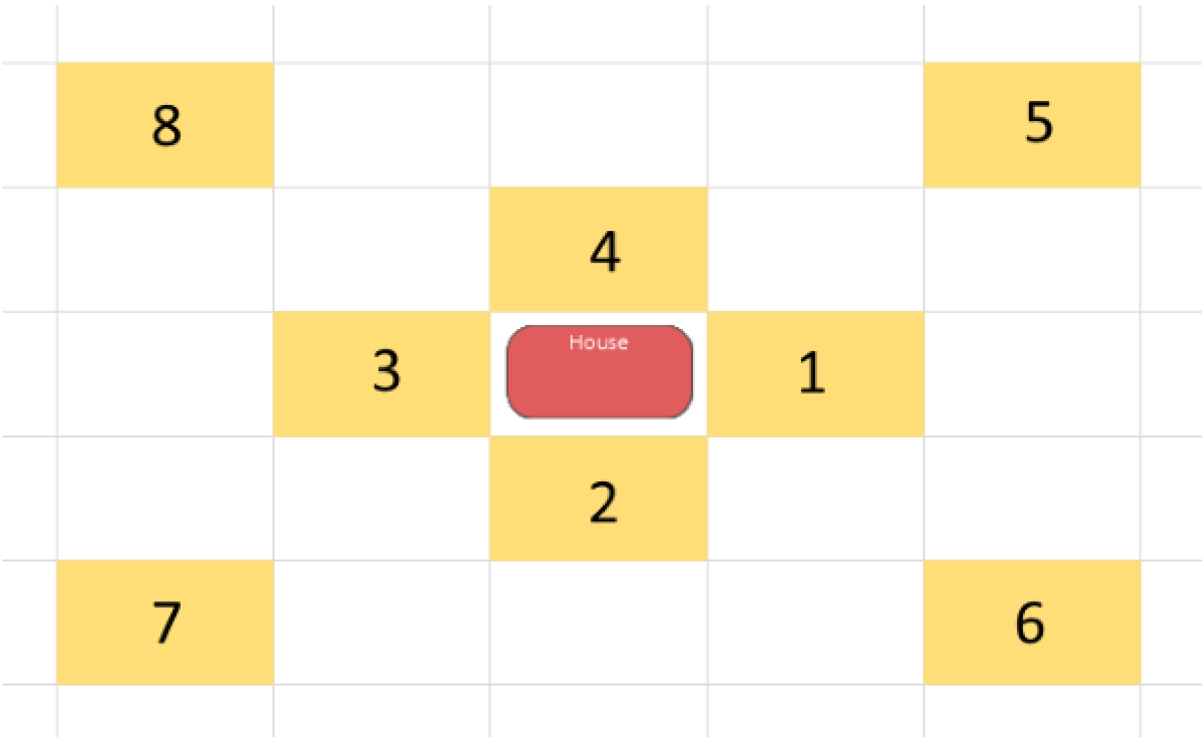


Figure 65: An alternate view of the initial referent cell set.

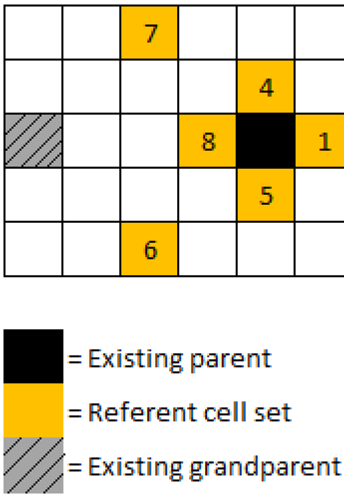


Figure 66: The initial referent cell set, with biased positioning because of an existing grandparent.

arrow has been removed. Providing that this removal still leaves the referent object with at least one remaining reference arrow (since otherwise it will be removed from the diagram), a new position for the object may be calculated if the object is no longer in an adjacent or surrounding grid square of at least one of its parents. In this case, the new position should be calculated exactly the same way as if the object were being newly placed on the diagram. If a specific existing parent needs to be chosen (in the case where more than one parent remains on the graph), then the parent with a free cell in its referent cell set should be chosen. If multiple parents satisfy this constraint, the parent that can supply the closest (spacial) cell on the diagram to its current position should be chosen.

Filling the graph

The layout algorithm, as described above, deliberately starts with a relatively large grid size in order to display objects clearly – if the running program only creates a few objects, then this is ideal. However, with programs that create more than a few objects the layout algorithm may run out of space, either reaching the edge of the visualisation area or having no remaining space to expand objects into (that

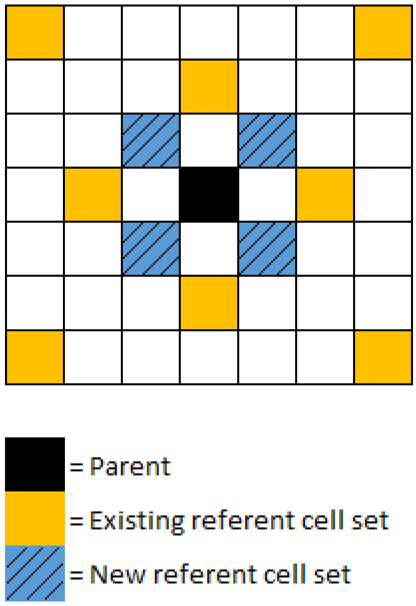


Figure 67: Existing reference cells pushed outwards to leave space for new reference cells.

is, if an object is added to the graph and no free cells remain in the referent cell set.)

We first consider the case where the referent cell set contains no free cells, which will occur if an object holds more than eight references. In this case, the existing referent cells will all be pushed outwards from their parent, creating space for four new reference cells (Figure 67.) If a cell tries to push into a cell that is already occupied, that cell is also pushed in the same direction (and this behaviour is recursed until a free space is found.) If these four new referent cells also become occupied, the process may be repeated.

While this process provides a mechanism for allowing the number of referenced objects to expand while still maintaining a readable layout, it does not deal with the case where the edge of the visualisation area is reached. This can occur through the process described above, as well as locating a free grid square to place a new object.

In the instance where the edge of the visualisation area has been reached, the algorithm will first attempt to shift the entire graph in the opposite direction – for example, if the top edge of the visualisation area has been contacted, the entire visualisation will be shifted one cell downwards. If the visualisation cannot be shifted (because an element is already against the bottom edge of the diagram) then the diagram will be "shrunk".

Shrinking

Each shrinking step of the diagram results in the cells being reduced to 80% of their former size, and consequently the contained objects being reduced in size in a similar way. Initially (to minimise unnecessary movement) the algorithm was implemented such that the objects were freed from their cells during this shrinking process, and then after the smaller cells had been allocated, they snapped to the closest cell to their current position. However, this proved ineffective in practice – animation still occurred, but it was less consistent. In addition, this made much less efficient use of the newly allocated cell space. The current implementation therefore fixes the centre cell (where the initial object is placed) and then moves all cells and objects towards the centre of the diagram, creating new cells on the outer edges. This shrinking step is repeated until enough free space is available to perform the requested operation. While this approach scales well, there is a point where the physical size of the objects presents a limit to the number that can reasonably be shown. On a typical 24 inch monitor, this limit is around 200 objects.

Once shrunk, the cells never grow again during the execution of the program, since we wish to avoid repeated growing and shrinking of the cells (if there have been many objects created during one part of the program's execution, it is deemed as highly likely this will happen again.)

Manual positioning

The user may, at any time, manually reposition any object, and the layout algorithm will adjust to take this position into consideration. Manual positioning of an object "locks" that object in place; any repositioning that might normally take place as a result of the algorithm described above will not occur for the locked object (with the exception of shrinking the graph when no free space remains.)

The user has full control over where they reposition the object - it will not snap to a particular grid square, since these are invisible to the user and therefore this would be confusing. However, for the purpose of the layout calculation, the object will be considered as occupying the grid square closest to its current spacial position. When the graph shrinks however, the object will be realigned with its chosen grid square (since in this case, the object must move anyway.)

There are two main cases where this functionality is useful. Firstly, if the layout algorithm is presented with a project where it does not perform well, it provides a "backup" case for the user to manually position the objects as they choose. Secondly, this means the user can create a layout meaningful to them – if they wish to mimic the exact layout drawn by a lecturer on a whiteboard, or drawn in a textbook for example.

Collection layouts

Some known classes afford a specific type of layout, and thus the placement of these objects is calculated differently. Such custom layout could be implemented for any known class (Section 5.9.4); in this case however we have implemented this for a subset of collection classes.

Array based collections Array based collections are those that contain an ordered set of elements. This includes array, `Vector` and `ArrayList` objects. The layout of this type of collection is commonly perceived as being linear; that is objects within an array have a predefined order. However, the basic layout, by design, "spreads" the objects around the collection – this is thus not the ideal layout in this scenario.

In contrast to the basic layout, elements of array based collections are not spread out around the central collection, but instead displayed in a linear fashion. In effect, the basic layout is followed, but with all blocks other than those "under" the existing parent declared invalid.

Maps Maps are collections that link a key to a value, with a value typically being quickly accessible via its key. This includes `Hashtable` and `HashMap` objects, and potentially all other objects that have a mapping (key-value) relationship. In this case, there is a clear relation between each key and each value, and the addition of a custom view for these objects reinforces this (Section 4.3.6.) In this custom design, the keys are displayed down the left hand side of the map, and the values down the right hand side.

The display algorithm thus works in a similar way to support this view, and in this case the closest free cell to the key or value slot *within* the object will be used. The only other restriction is that key objects must always be to the left of the object, and the value on the right. (This restriction is waived in the event that the same object is used as a key and value on the same map, however this is a rare occurrence.)

5.9.4 Improvements

The layout algorithm seems to work well for a number of examples (see Chapter 6), however there are a number of possible improvements.

Initial scale

At present, the algorithm is completely unaware of the potential scale of the program, and thus starts with assuming the creation of a modest number of objects.

Although it is capable of scaling to represent many, this can involve many shrinking steps, creating animation that may be unnecessary if the approximate number of objects created can be chosen ahead of time. This will always be a very difficult problem to solve since the program cannot be run ahead of time with a live visualisation. However, if an approximate number of created objects could be calculated, the algorithm could potentially choose a starting cell size that would result in much fewer shrinking steps. It could also store information about previous runs of examples, and persist the same scale between invocations.

Re-positioning cells due to manual positioning

When the user manually positions an object, the algorithm will allocate it to its spatially closest cell. However, it will in most cases not perfectly align with this cell, meaning any future objects created around it will also be out of alignment (and may in some cases slightly overlap.) This could be avoided in many situations if the grid around the user-positioned object were to be reallocated to align better with the user's positioning. Such allocation could be temporary, and would be reset whenever shrinking occurred. Alternatively, objects could be forced into existing cells, though this restricts the user's ability to finely position objects.

Further custom layouts

Custom layouts are implemented for relatively few collection classes at present (Section 5.9.3). An obvious improvement would be to analyse the common children and thus common layout requirements of further classes, and extend the algorithm with further layout exceptions that better represented these children. This could make the visualisation clearer, as well as potentially more compact.

5.9.5 Summary of Layout Algorithm

The layout algorithm as presented above has been designed to work well with relatively small programs, but will break down when used alongside programs that

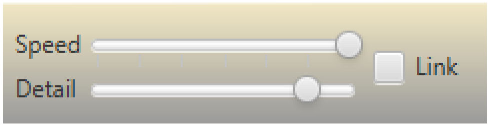


Figure 68: Novis' speed and detail sliders.

create many hundreds of of objects in user code. The reasonable limit is around 200 objects, on a normal 24 inch monitor. This is therefore not an algorithm that can be universally applied to visualise all Java programs, but should work well within its problem space (programs applicable to CS1).

5.10 Speed and stepping granularity

5.10.1 Speed slider

The speed of the visualisation cannot 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 (Figure 68), no delay is added and the program executes at the maximum speed possible with the current choice of animation detail (Section 5.11). 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.

5.10.2 Debugging

As mentioned in Section 4.5.2, 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 69). 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.

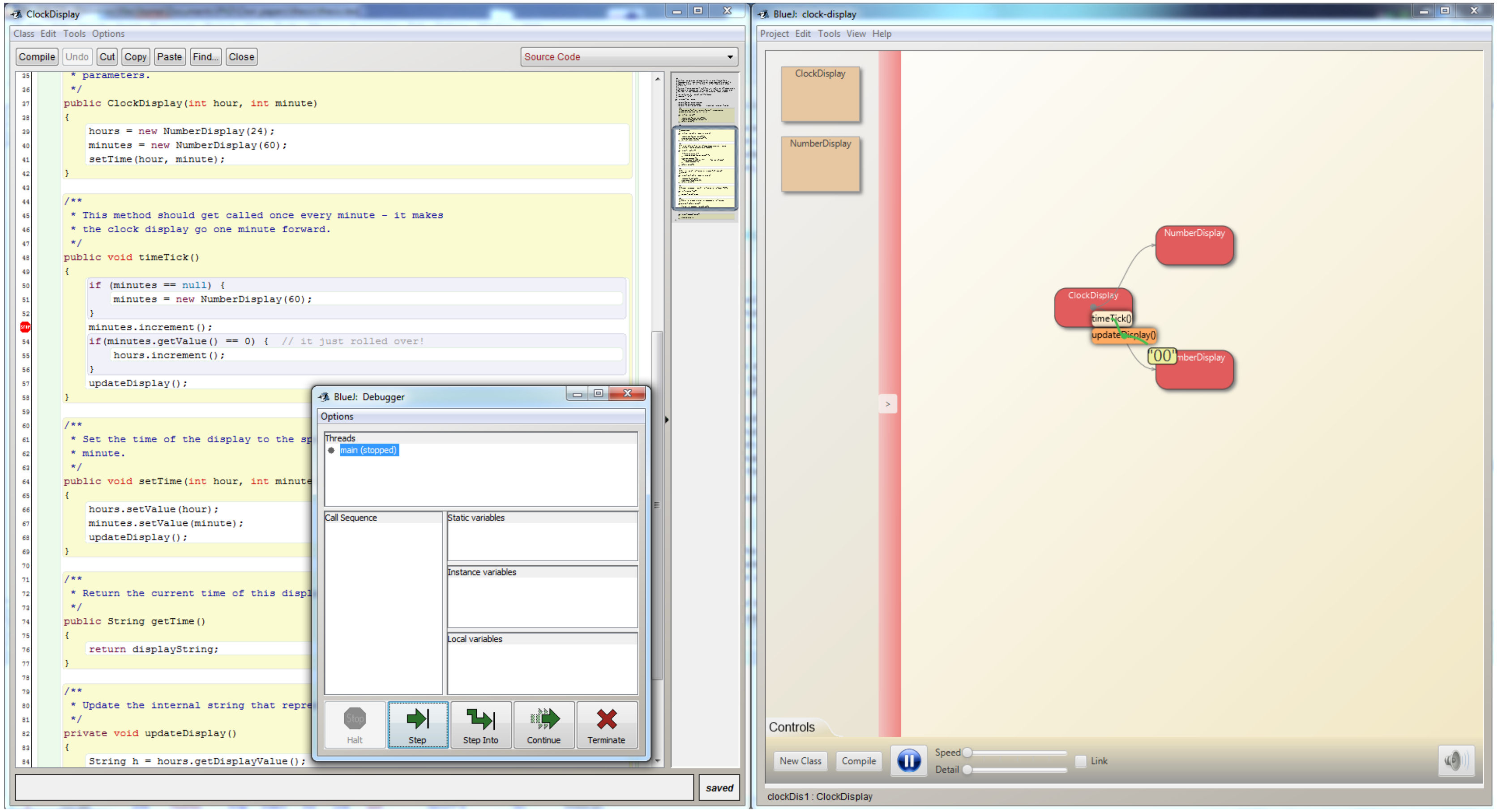


Figure 69: Novis working with the BlueJ debugger.

5.11 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.

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, destruction and activity levels (see Section 5.11.1).

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.

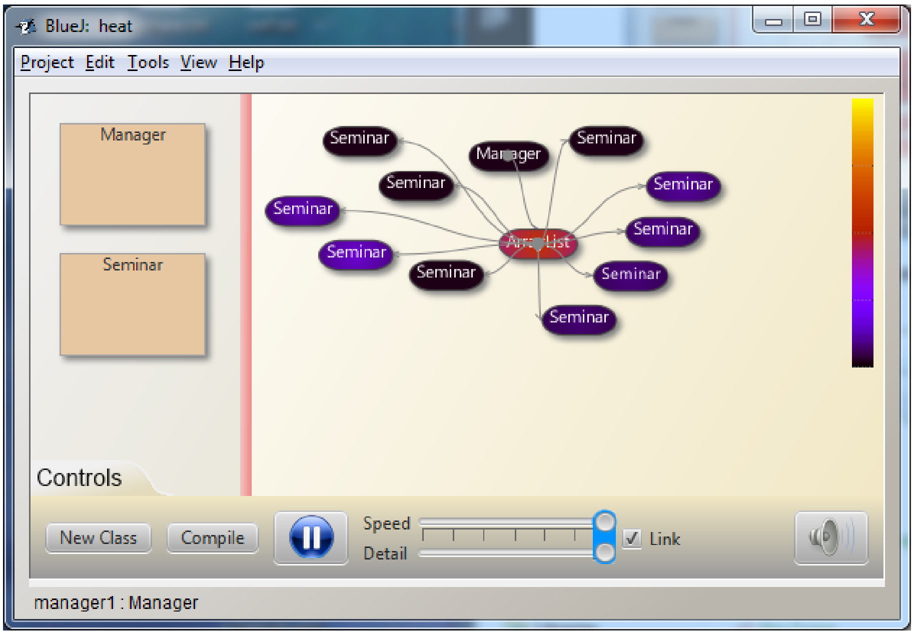


Figure 70: Heatmap view illustrating program activity.

5.11.1 Heatmap

In some programs many objects will be visible on screen, and any specific detail about fields, methods, or state on each one would become difficult or impossible to follow. Instead, a pattern of general activity is all that can reasonably be shown, and objects need to be shown in a compact notation using just enough space to display their type.

Initially, a "flashing" type approach was implemented - objects changed colour for a few milliseconds while when activity was present (a method was called or a field changed). This provided a good outline of all objects that were currently in use. However, distinguishing the objects that were being used the most was challenging with this method; any objects that saw any reasonable level of activity appeared very similar.

Instead of a simple flashing of active objects, Novis' display turns into a heatmap (Figure 70), where colour is used to indicate activity. 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 depicts object creation and destruction, as well as hotspots of activity, providing a quick high level overview of programs with ongoing activity.

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, retaining a colour that makes it distinct from surrounding objects. A higher level of detail can then be selected, and the rogue object, including its execution path, can be examined in more detail. Other problematic states such as infinite loops can be detected on the object level using this visualisation in a similar way. The opposite may also be easily spotted - if activity is not occurring as often (or at all) on a group of objects, this problem scenario will be easily distinguishable whilst in a heatmap view.

While this is not sufficient information for resolving the problem, the view provides enough contextual information so that many problems can be immediately spotted. 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.

5.12 Summary

The implementation as described above contains several novel features; these can work together to provide a usable visualisation for both understanding and debugging programs of a variety of sizes.

Chapter 6

Textbook Evaluation

The notional machine has been informally tested throughout its development. Many contrived and ad-hoc examples were used to refine the functionality discussed above, and Novis has been presented to peers at informal seminars in order to gain some form of user feedback. However, ad-hoc testing performed during development is not a guarantee of its effectiveness at visualising suitable examples – a more rigorous process is needed in order to assert that the system is useful. This will be presented in two main parts, first using a corpus of textbook examples, and second using students to perform a usability test (Chapter 7).

The textbook evaluation is performed on a methodically chosen set of textbook examples (Section 6.3). However, other approaches could have also been used in the selection of textbook examples. Instead of a systematic set of textbook examples chosen from the problem space, examples could have been chosen that proved deliberately challenging to Novis. They could have proved challenging for many reasons - a large number of objects, an unusual pattern of references (therefore presenting challenges for the layout algorithm), or many objects that had a very quick lifespan. This would have enabled a more detailed examination and evaluation on the limitations of the visualisation, but would have likely involved cherry picking examples, producing a less empirical evaluation.

Educators could have also been asked to choose examples that they commonly use in CS1, and this corpus of examples evaluated using the tool. This would have provided a greater confidence that the examples chosen were used in an educational context, but would arguably not be as representative as the chosen approach (choosing best selling introductory Java textbooks from Amazon.)

Other approaches could have also been used in the evaluation of the examples – principally, CS1 students and educators could have been asked to perform the analysis based on the devised ratings, and rated the tool for each example accordingly. While this approach would have removed any bias introduced by the author performing the analysis and rating of the tool, it would have been slower. It was thus deemed sufficient to mitigate any such bias by providing well-reasoned and detailed categorisation of the possible ratings before the evaluation began.

This chapter aims to evaluate how well Novis visualises a selection of different, appropriate textbook examples. Specifically, we analyse whether the examples that fall into the problem space are usefully visualised by the tool – we wish to use "typical relevant examples from the problem space". In order to formalise this, we define these terms as follows:

Problem Space: CS1 (See Chapter 3).

Relevant Examples that contain at least some form of object interaction.

Typical Examples chosen from popular textbooks.

We can thus translate the colloquial requirement above to "object interaction examples from popular CS1 textbooks".

6.1 Criteria

For the purposes of this evaluation, we place restrictions on the textbook examples that are deemed *relevant*. First, we restate the relevant criteria outlined in Chapter 3:

- The example must use vanilla Java - examples that cover other forms of

Java-like languages, Android for example, are excluded.

- The example must only use a single thread.
- The example must have no compile time errors.
- The example must not centre around annotations.
- The example must not make use of custom generics (however, library classes that use generics, such as the collection classes, are in scope.)

We also place two additional criteria on these examples:

- The example must make use of *objects* from at least one user-created (as oppose to a library) class.
- At least one interaction (such as a method call) must be performed on the aforementioned user-created object.

The above criteria deliberately exclude very simple examples only making use of static methods. Most notably "Hello World" style examples, often found at the start of many introductory programming textbooks, are deemed not interesting for the purposes of this evaluation. For the majority of textbooks, this means that relevant examples can be found from the point at which the book introduces objects, since it is at this point an object interaction (in any form) is usually presented to the student. The place at which this point occurs (and thus the availability of useful examples for selection) varies greatly depending on the textbook chosen - some books that take a more traditional procedural style approach to teaching Java introduce these concepts much later than those that take a deliberate objects-first approach. The point at which the textbook examples cease to fit into the problem space is often until either the topics of concurrency or graphical user interfaces are introduced.

In defining a "useful visualisation", the goal is to determine whether the important aspects of the example are visualised appropriately. In order to do this, we refer to Nielsen's heuristics (Nielsen, 1995). These are as follows:

H2-1, Visibility of system status : The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

H2-2, Match between system and the real world : The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

H2-3, User control and freedom : Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

H2-4, Consistency and standards : Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

H2-5, Error prevention : Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

H2-6, Recognition rather than recall : Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

H2-7, Flexibility and efficiency of use : Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

H2-8, Aesthetic and minimalist design : Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

H2-9, Help users recognize, diagnose, and recover from errors : Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

H2-10, Help and documentation : Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

However, many of the heuristics are not directly relevant to the individual examples, instead applying to the tool as a whole (for instance, help and documentation, user control and freedom). Since we are specifically analysing the tool's ability to usefully display relevant examples, rather than its user interface as a whole, they are not directly relevant for this evaluation.

The two particular relevant heuristics we will apply in this evaluation are therefore H2-1 and H2-8:

H2-1, Visibility of system status : "The system should always keep users informed about what is going on, through appropriate feedback within reasonable time." For our purposes, this equates to the tool providing timely information of all the educationally important events, and the objects being represented appropriately for each specific example.

H2-8, Aesthetic and minimalist design : "Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility." For our purposes, this does not relate to dialogues but instead the layout of the diagram. The layout should be as easy to understand as reasonably possible, without unnecessary arrow crossings and overlaps.

It could be argued that a third heuristic, H2-2 (Match between the system and the real world) should also be examined in the context of this evaluation. However, it would be very similar to H2-1 in this context – speaking the language of the user by providing relevant and timely feedback on the examples being represented. Since a clear distinction between the two is not obviously apparent here, it will not be used as a separate point for this evaluation.

6.2 Evaluation Criteria

The purpose of the evaluation is to determine how well Novis performs in visualising these examples; the criteria used are based upon the above heuristics. This has therefore been formalised into the following:

Layout : This criterion is in reference to heuristic H2-8; informally it asks the question "How well does Novis lay out the diagram?"

- **Good** : The diagram is clear, with few or no arrow crossings. Clear spatial grouping is applied to closely related objects, and there are no avoidable overlaps.
- **Moderate** : There may be a few arrow crossings or overlaps between objects, but there is a better possible layout.
- **Bad** : The diagram is confusing, unclear or has many avoidable arrow crossings and overlaps, there is clearly a better layout that the tool could easily have used.

Usefulness : This criterion is in reference to heuristics H2-1 and H2-2; informally it asks "How well are the important aspects of the example conveyed to the user?"

- **Good** : The visualisation clearly shows several important aspects of the example.
- **Moderate** : The visualisation shows some of the relevant aspects of the

example.

Bad : The visualisation does not effectively show any of the important aspects of the example.

6.3 Example selection

Six of the best selling, relevant textbooks were chosen as the basis for this evaluation.

Initially, the 20 best selling textbooks in the "Introductory Java" category on Amazon were selected for potential inclusion[1]. From these 20 textbooks, we excluded those that fell into the following categories:

- Duplicate entries (some textbooks were listed more than once on the list with slightly different names or different editions – these were not considered to be distinct entries.)
- Textbooks aimed at "non-standard" Java. These fell into two categories; those predominantly geared at Android development, and those geared towards web development. Both of these categories have been explicitly excluded from the problem space (Chapter 3) and thus textbooks purely based on this content were not deemed relevant.
- Textbooks that expected a background in programming. Despite being "introductory Java" textbooks, this was not a strict subcategory of "introductory programming", with a handful expecting a background in C++ or similar. Since this work is aimed at novice programmers, these are not relevant.
- Textbooks for which example code was difficult to source (not freely available on the book's website or in any other obvious location.)

This then left six textbooks which, in no particular order, are as follows:

[1] `http://www.amazon.co.uk/gp/bestsellers/books/269848/ref=pd_zg_hrsr_b_2_7_last` – Accessed 17th April 2014

- Rogers Cadenhead: Sams Teach Yourself Java in 21 Days – Sams (Cadenhead, 2012)
- Rogers Cadenhead: Sams Teach Yourself Java in 24 Hours – Sams (Rogers, 2011)
- Herbert Schildt: Java, A Beginner's Guide – McGraw-Hill Osborne (Schildt, 2011)
- David J. Barnes & Michael Kölling: Objects First with Java: A Practical Introduction Using BlueJ – Pearson (Barnes and Kölling, 2011)
- Y. Daniel Liang: Introduction to Java Programming, Comprehensive Version – Pearson (Liang, 2012)
- Mike McGrath: Java In Easy Steps – In Easy Steps (McGrath, 2011)

From these books, examples were then chosen as follows:

- The examples were examined in order until one was discovered that fit the problem space, as well as the additional requirements of this evaluation (Section 6.1). This was repeated for each textbook, giving a set of six systematically chosen "initial examples".
- The further examples were then examined in each textbook, up until the point they no longer fitted the defined problem space. Any that introduced different concepts to the examples already covered were added to a set of "later examples".

A consistent evaluation could then be performed on both the above sets of examples, making use of the evaluation criteria defined in Section 6.2. The full source code available for each example is included in Appendix A.

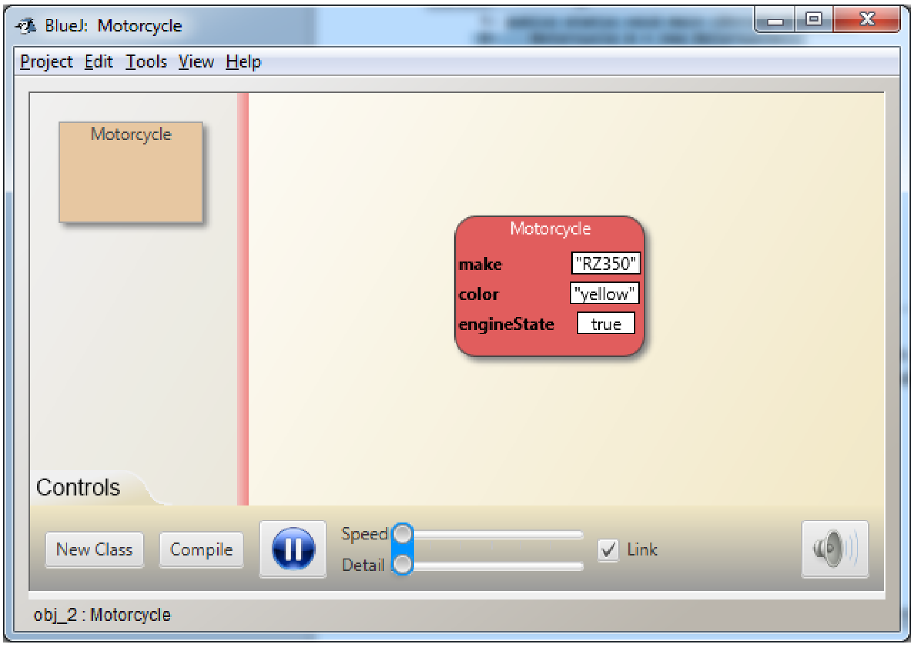


Figure 71: Cadenhead's "Motorcycle" example in Novis.

6.4 Initial examples

These are the first examples in each book involving object interaction. Here these six examples are be presented with a consistent template. The later examples will be evaluated separately in Section 6.5.

6.4.1 Example 1: Motorcycle

Source : Rogers Cadenhead: Sams Teach Yourself Java in 21 Days, chapter 10, page 127. (Appendix A.1.)

Description : The example creates a `MotorCycle` object, then changes its fields before calling a method to print out its fields (Figure 71.)

Educational Goal : Demonstrating basic object creation and method calling.

Evaluation : The layout creates just one single object, and Novis places it in the centre of the visualisation area – while trivial, it is difficult to imagine a better visualisation for it in this particular example. The project is run with both speed and detail sliders set to the slowest and most detailed level; this is clearly the appropriate setting for such a trivial example.

Novis clearly shows the creation of the object. The methods being called on the object are also clearly visualised by the rectangles that appear at its bottom right corner. The field changes are also visualised by the values of the fields changing in the object's expanded view. The educational goal of this example is thus well supported by visualising it in Novis.

Categorisation :

Layout : Good

Usefulness : Good

6.4.2 Example 2: Modem

Source : Rogers Cadenhead: Sams Teach Yourself Java in 24 hours, chapter 10, page 127. (Appendix A.2.)

Description : The example creates two different objects which are subclasses of `Modem`, and then calls a few methods on each (Figure 73).

Educational Goal : Demonstrating inheritance and polymorphism.

Evaluation : This example demonstrates inheritance and introduces polymorphism – it relies on a hierarchy of different modem classes, `CableModem`, `AcousticModem` and `DslModem`. All of these classes inherit from a common `Modem` superclass. Additionally, a `ModemTester` class makes use of a single type of modem, `DslModem` in the example. This is clearly visualised with Novis' class view (Figure 72).

In the object interaction of this example two objects are created, instances of `CableModem` and `DSLModem`. The constructor chaining is clearly apparent in Novis (the constructor of `DSLModem` is called first, then that of the

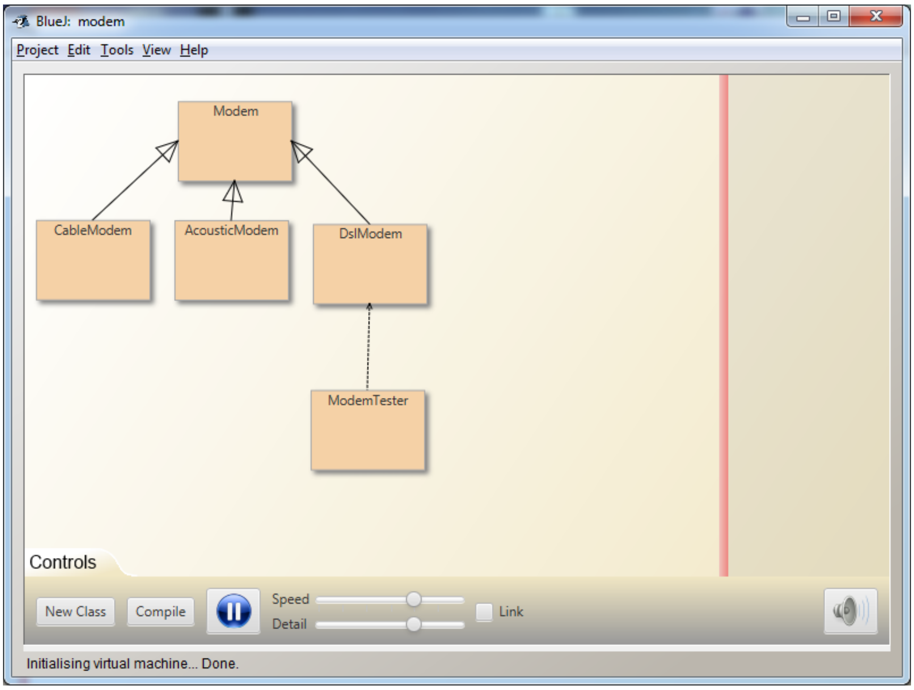


Figure 72: The class view of Cadenhead's "Modem" example in Novis.

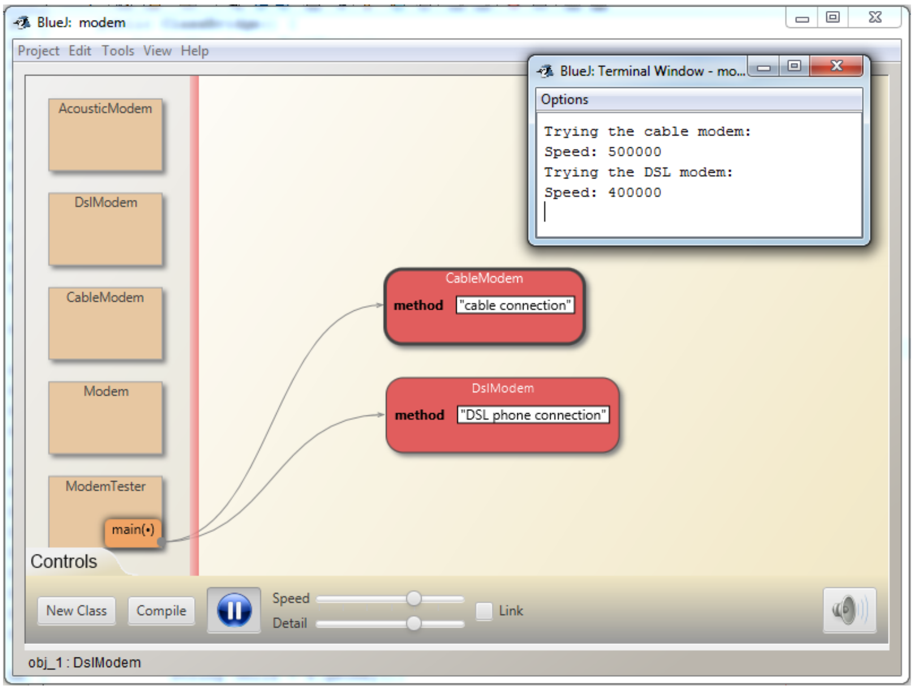


Figure 73: The object view of Cadenhead's "Modem" example in Novis.

superclass.)

The current visualisation shows a number of interesting aspects. It clearly depicts the number of objects created, and the fact these objects have a reference from the application's main method. The order of creation is also shown. The example could also be trivially modified to change the static types to `Modem` in the source (rather than keeping them identical to the runtime type) and thus demonstrate the difference between static and dynamic types. In this case, the objects will always show as their runtime type in the diagram regardless of the static type used to describe them in the source.

The layout is similar to the other previous examples of this size – it is clear and useful; Novis lays it out well (the objects are positioned one above the other.)

Categorisation :

Layout : Good

Usefulness : Good

6.4.3 Example 3: Vehicle

Author : Herbert Schildt: Java, A Beginner's Guide, chapter 4, page 118. (Appendix A.3.)

Description : This example features two classes, one (`VehicleDemo`) which simply has a main method. The other (`Vehicle`) is an "object" class that is created, and its fields are subsequently set to some custom values (not from within a method.) Conceptually, it is very similar to the Motorcycle example previously discussed in Section 6.4.1 (Figure 74).

Educational Goal : Introducing objects, methods, and fields.

Evaluation : The layout creates just one single object, and Novis places it in the centre of the visualisation area. The project is run with both speed and

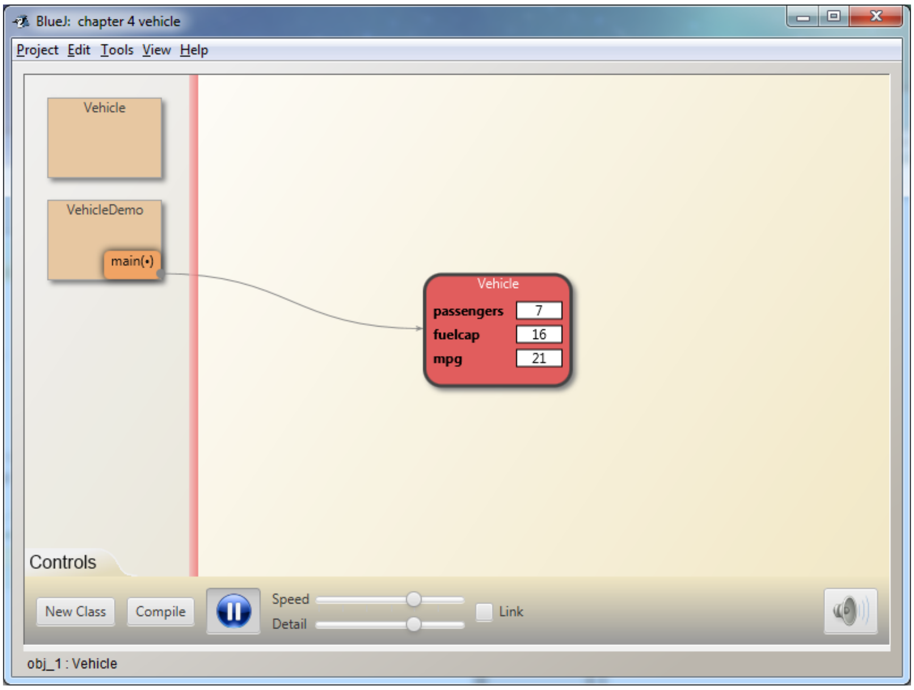


Figure 74: Schildt's "Vehicle" example in Novis.

detail sliders set to the slowest and most detailed level; this is clearly the appropriate setting for such a trivial example. Novis does clearly show the creation of the object and the methods being called on the object. The field changes are also visualised; the values are updated in the diagram as they are altered in the underlying machine.

Categorisation :

Layout : Good

Usefulness : Good

6.4.4 Example 4: Clock Display

Author : David Barnes and Michael Kölling: Objects First with Java: A Practical Introduction Using BlueJ, chapter 3, page 62. (Appendix A.4.)

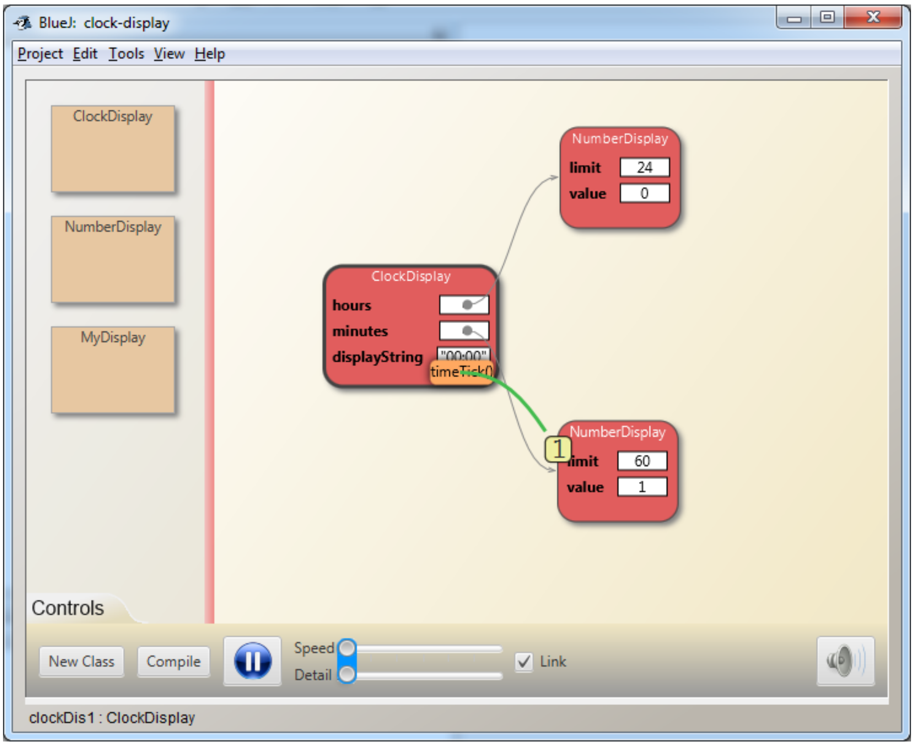


Figure 75: Kölling and Barnes' "Clock display" example in Novis

Description : This example creates a `ClockDisplay` object that references two `NumberDisplay` objects, one for the hours and one for the minutes of the clock. An `increment()` method is provided which increments the time by one unit (rolling over from the minutes to the hours display if necessary) (Figure 75).

Educational Goal : Demonstrating object interaction.

Evaluation : This example uses three objects. Novis shows the references between them, the method calls, call chain, parameter passing and return values. The example works well on the most detailed and slowest setting (since here the parameters are shown visually being passed from the `ClockDisplay` instance to each `NumberDisplay` instance.) In reference to H2-1 therefore, taking into account proximity of related objects and layout, Novis demonstrates object interaction well.

The layout shown in the screenshot is similar to the default layout, but was positioned manually. Novis initially laid out the objects with some minor overlap, so the `ClockDisplay` objects were dragged away from the `NumberDisplay` object in order to provide a clearer view. This highlights one use of the manual layout, since in this case it enabled the user to make a small change to create a good layout.

Categorisation :

Layout : Moderate

Usefulness : Good

6.4.5 Example 5: Circles

Source : Y. Daniel Liang: Introduction to Java Programming, Comprehensive Version, chapter 8, page 322. (Appendix A.5.)

Description : The `TestSimpleCircle` class provides a main method which, when called, creates three separate `SimpleCircle` objects. These simply contain one field, `radius`, which is set in the constructor (Figure 76).

Educational Goal : Demonstrating object creation.

Evaluation : The example is laid out well; the three objects are laid out in a column in the centre of the diagram with no overlap. The references that are present are static references, since the three objects are created from a static context. The main method call is also visualised, as are the constructors and the instantiation of the `radius` field in each object. Initially, this field is set to its default value (0 in this case); the constructor then executes to change its value to the one passed into its constructor (Figure 77).

The call chain is also visualised, showing that the execution begins with the `main` method which executes each constructor in turn. The parameters are shown on the call chain, reinforcing that each `SimpleCircle` object is created with a distinct argument (Figure 77).

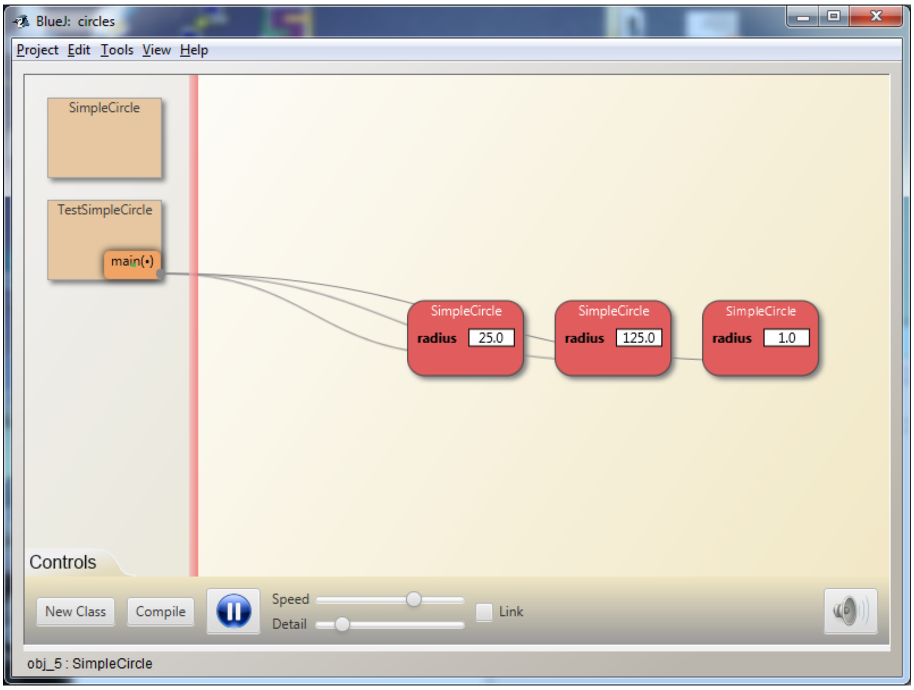


Figure 76: Liang's "Circles" example in Novis

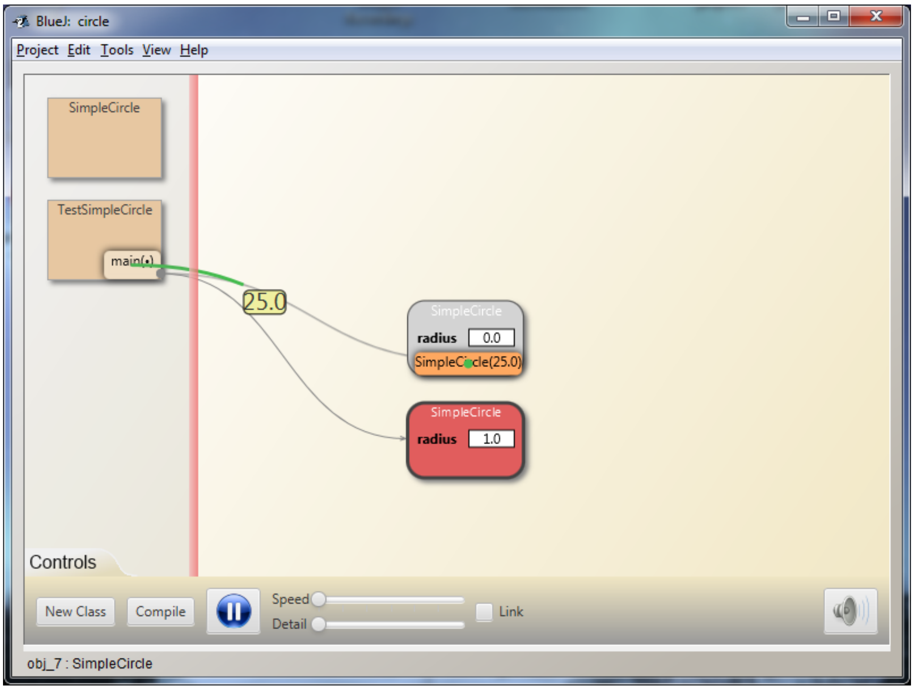


Figure 77: Liang's "Circles" example passing a parameter in Novis

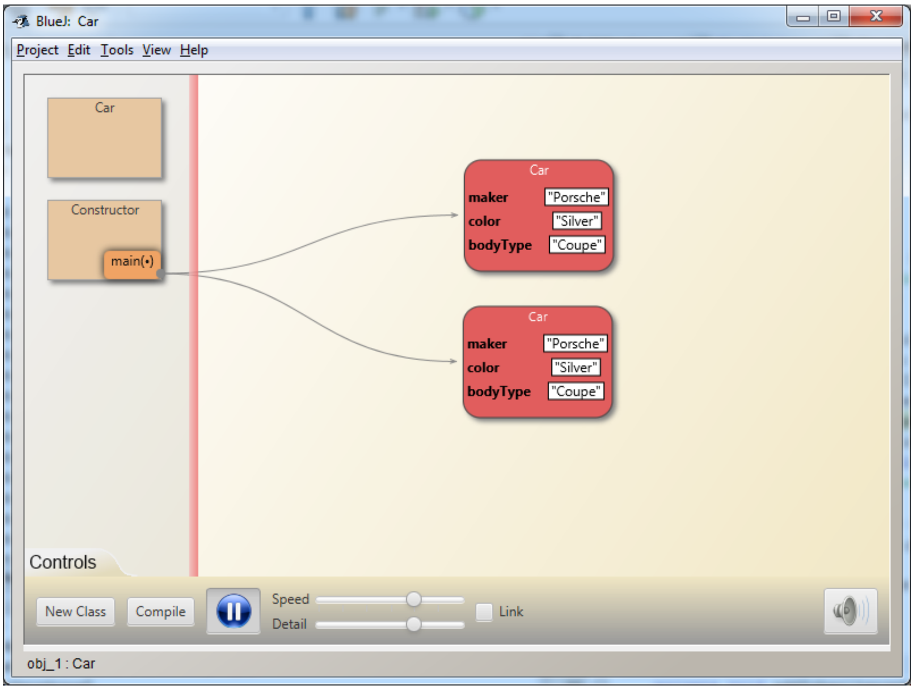


Figure 78: McGrath's "Car" example in Novis

Categorisation :

Layout : Good

Usefulness : Good

6.4.6 Example 6: Car

Author : Mike McGrath: Java In Easy Steps 4th Edition, chapter 6, page 112. (Appendix A.6.)

Description : A class named `Constructor` creates two instances from a `Car` class, using two different constructors via constructor overloading (Figure 78).

Educational Goal : Demonstrating object creation and initialisation.

Evaluation : The first relevant example in this textbook is contextually similar to the Liang's "Circles" example – - a class called `Constructor` creates two

`Car` objects. As in the previous example, there are no instance references between them; they are thus displayed in a column in the centre of the diagram. The references between the static `main` method and the objects are visualised clearly.

Both objects are created with default constructors (with no parameters); one object has its fields later set to different values by means of a setter method. The default constructors initialise the fields to some hard-coded values; before these values are set however the object is shown as grey (meaning it is not yet constructed) and the fields show their default value (`null` in this case.) In the setter method which is called later, the parameters are shown being passed along the call chain, then the values of fields are changed in the diagram.

One potentially confusing point does exist, that is the class that creates the objects is itself called `Constructor` -– so if trying to identify the constructor on the diagram, it is possible that students would select this class rather than the actual syntactical constructor. However, this is arguably an issue with the example rather than the tool or the notional machine itself, as it could just as easily become a confusing point with the use of any other environment.

Categorisation :

Layout : Good

Usefulness : Good

6.5 Later examples

While the methodology used to select the above set of initial examples was chosen to be objective and consistent across different textbooks, the nature of how they were chosen (simply choosing the first relevant example from each textbook) inevitably means that most of them are relatively simple. However, neither the notional machine notation nor Novis should be restricted to examples on a scale that simplistic.

We therefore performed evaluations on more complex examples. In addition to the examples presented above, a further set of interesting projects were chosen from the same textbooks. They were selected by examining later examples through each textbook, but eliminating any whose primary learning objective was one covered in the set of initial examples.

In summary, these were as follows:

- Demonstrating object creation,
- Demonstrating object interaction (such as calling a method on an object or changing its state),
- Introducing polymorphism.

The analysis of this further set of examples is presented below.

6.5.1 Example 7: Giftshop

Author : Rogers Cadenhead: Sams Teach Yourself Java in 21 Days, chapter 6, page 172. (Appendix A.7.)

Description : The example creates a `StoreFront` object that contains an `ArrayList` of `Item` objects (Figure 79.)

Educational Goal : Introducing collections, specifically the `ArrayList`.

Evaluation : Novis successfully visualises many features of this example. Similar to previous examples, it visualises object creation, field instantiation, the call chain, constructors and method execution / invocation. The `main` method on `GiftShop2` is first called, which instantiates a `StoreFront2` object. The `StoreFront2` object first creates the `ArrayList` object, then four `Item2` objects, adding each one to the `ArrayList` in turn.

This example highlights the Novis' ability to lay out `ArrayList` objects in a

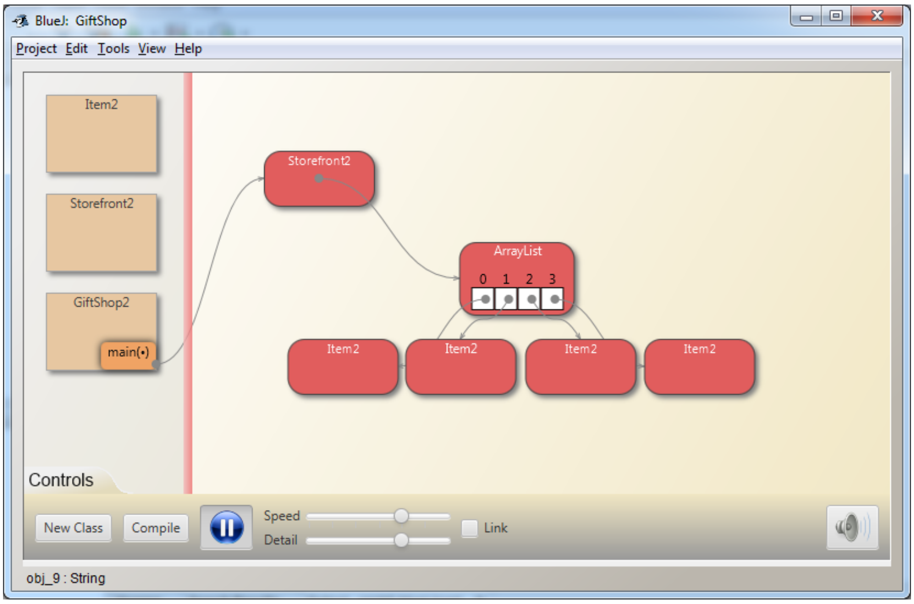


Figure 79: Cadenhead's "Storefront" example in Novis.

different way to normal objects - the elements, when added to the `ArrayList`, are displayed side by side underneath collection object. The layout in Figure 79 is fully automatic; the only manual addition was to click on the `ArrayList` to show its expanded view. This layout is much clearer for this type of collection than the default layout, since it also takes the order of the objects into consideration to ensure minimal crossovers.

Categorisation :

Layout : Good

Usefulness : Good

6.5.2 Example 8: BookDemo

Author : Herbert Schildt: Java, A Beginner's Guide, chapter 8, page 305. (Appendix A.8.)

Description : This example shows a `BookDemo` class referencing a `Book` array

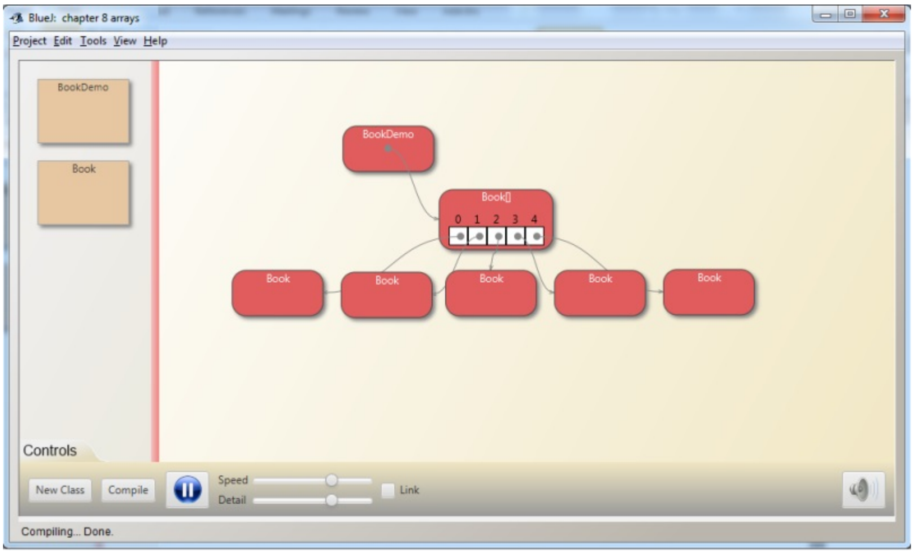


Figure 80: Schildt's "BookDemo" example in Novis.

object (which itself references a number of books.) The array is created, and then the code loops through each element within it, filling it with different `Book` objects (Figure 80.)

Educational Goal : Introducing collections, specifically arrays.

Evaluation : Novis uses its custom array layout for this example, which is the same layout used for the `ArrayList` in Section 6.5.1. The example first involves the user creating a `BookDemo` object, which then creates a `Book` array object. Unlike the previous example, this array is created with a fixed length, since primitive arrays are created with a fixed size and do not expand to allow a greater number of elements.

From this state, the `BookDemo` constructor then proceeds to create five separate `Book` objects to fill the array, then allocates them to sequential indices in the array. When the `BookDemo` object no longer holds a reference to the `Book` object (after it has been added to the array), the reference disappears and the object moves to its position in a row underneath the array object. The final layout looks good, with a close grouping of the objects and no arrow crossovers.

Categorisation :

Layout : Good

Usefulness : Good

6.5.3 Example 9: World of Zuul

Author : David Barnes and Michael Kölling: Objects First with Java: A Practical Introduction Using BlueJ, chapter 6, page 197. (Appendix A.9.)

Description : The world of Zuul is one of the more complex examples within the scope of Novis – it is a text based adventure game consisting of a number of `Room` objects, as well as a `HashMap` in each room, pointing to the other rooms that can be accessed from the one the player currently resides in (Figure 81.) This example has been chosen since there are a larger number of objects that pose a much greater layout challenge to the tool.

Educational Goal : Demonstrating class design and more complex object interaction.

Evaluation : Novis visualises the method execution behind the setup and creation of the objects well – the method and constructor calls are shown with the relevant parameters. The setup of the `Parser` object, in addition to the objects it directly refers to, is shown well and the objects retain their position. In contrast, the `Room` objects are first displayed referenced by the `Game` object (since this is where they are created). The location of the `Room` objects changes when the game object no longer holds a direct reference (since at this point, they are referenced by other rooms, dependant on how the map has been set up for the game.)

As the user is playing the game, the reference to the current room object changes; this takes place every time they navigate to another room. This approach clearly shows a specific change in the object graph as a result of the user's actions.

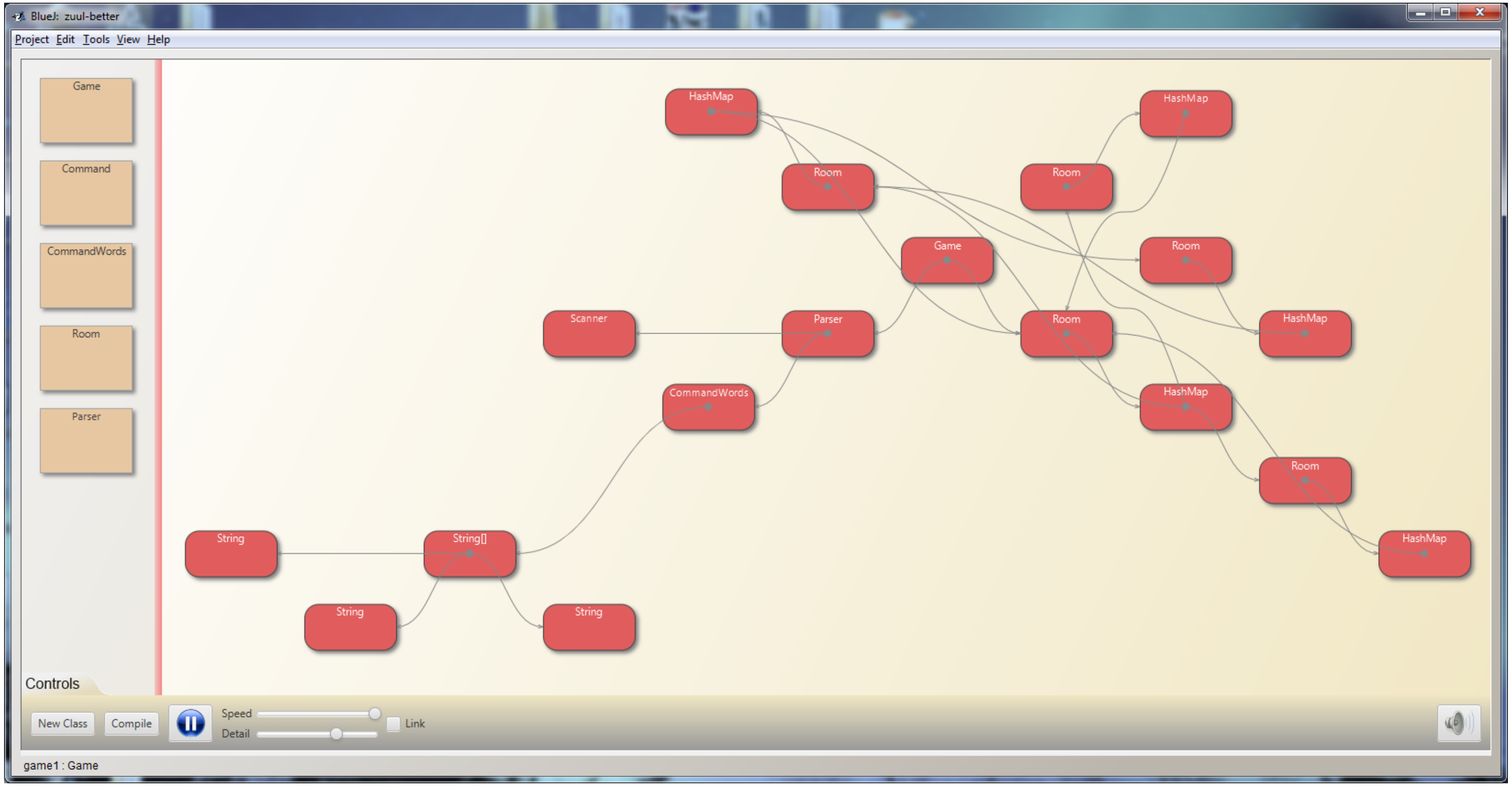


Figure 81: Kölling and Barnes' "World of Zuul" example in Novis with its default layout algorithm.

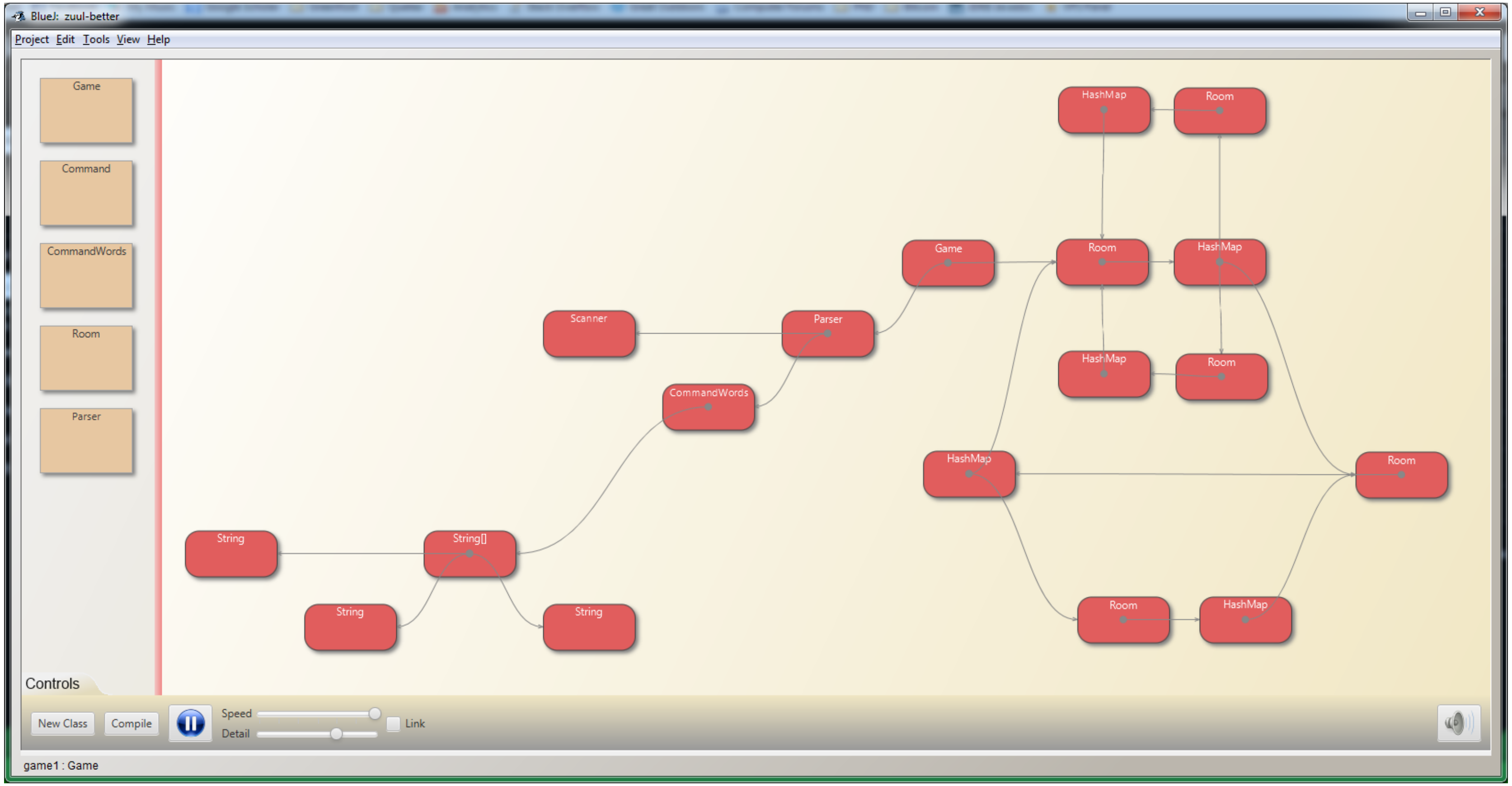


Figure 82: Kölling and Barnes' "World of Zuul" example in Novis, using manual positioning of the objects.

The layout algorithm does not work ideally for the entire diagram; this object graph presents a particularly hard use case. Figure 81 shows the objects as placed by the layout algorithm after the initial game setup has taken place and all the rooms have been created. While this does a reasonable job of laying out the classes it is not optimal. The section of the object graph containing the parser has been laid out well. However, the room layout contains many arrow crossings and is not easily fixed by hand. Figure 82 shows a much better layout using manual positioning, which certainly shows that the algorithm has scope for improvement.

Improving the layout algorithm for this particular case would be possible if it had more of a knowledge of the structure of the program. Most of the objects in this particular example are `Room` objects, each of which may reference four other `Room` objects (north, south, east and west respectively.) If the layout algorithm were able to analyse the fields and determine that this was the case, it could improve the final layout through some basic textual parsing. The north, south, east and west fields could be mapped to above, below, right and left of the room object respectively. This placement could then be used instead of placing the objects in the default order (as specified in section 5.9).

Categorisation :

Layout : Moderate

Usefulness : Good

6.5.4 Example 10: Student

Author : Y. Daniel Liang: Introduction to Java Programming, Comprehensive Version, chapter 10, page 396. (Appendix A.10.)

Description : The main method is called, and creates a `Student` object, which has a number of (private) fields, including a Date. The `getDate()` method is then called on the Student object, which clearly returns the date attached

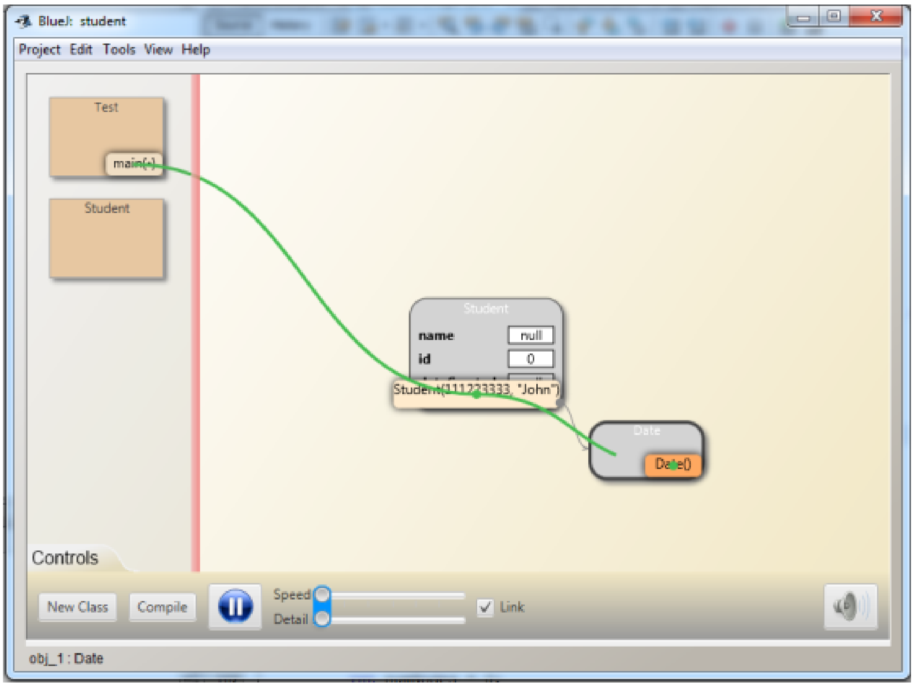


Figure 83: Liang's "Student" example in Novis.

to the student object – a mutator method (`setTime()`) is then called on that date object, changing its value (Figure 83.)

Educational Goal : Introducing immutability – specifically, that making all fields private in a particular object does not necessarily equate to making that object immutable.

Evaluation : The example highlights the educational goal well – since the same `Date` object is referenced by the `Student` object for the entire execution in the visualisation, it is always clear that only one `Date` object is present, and this is the one having its value altered in the example.

The example uses two objects, a `Student` and a `Date`. Both are laid out and visualised well. The diagram shows the two objects with the reference between them, in particular it clearly outlines the fact the only `Date` object shown is the one referred to by the `Student` object. Novis also displays the call chain, parameter passing, and the use of the `main()` method in the `Test` class.

Figure 83 shows the `main` method in the process of creating the `Student` object, which is in turn in the process of creating a `Date` object. Since neither of these objects are yet fully constructed, they are both shown in the grey colour (and change to the default red colour when their constructors have finished executing.)

Categorisation :

Layout : Good

Usefulness : Good

6.6 Summary

For all of the examples shown above, Novis provided a visualisation that was at least moderately useful, with at least a moderately good layout. No examples were found that were deemed within the scope of this work that could not effectively be displayed using the tool. Considering that usefulness is arguably the more important of the two criteria covered here, only three of the examples were not considered "good" in terms of their usefulness – this would seem to indicate that the majority of examples can indeed afford a useful visualisation when executed within Novis.

Specifically, this evaluation shows that Novis visualises a number of areas well:

- Object creation
- References / structure
- Method calls
- Parameters / return values
- Arrays
- Object identity

Evaluating the examples' layout, two were considered moderate (or moderate to good in one case) rather than good. While this shows that the layout algorithm forms a good basis for many examples, there is scope for it to be improved. However, while this presents an interesting problem, the layout algorithm is not the primary focus of this work.

The results above strongly suggest that Novis can usefully display a variety of examples deemed in scope for this work. Following on from this, further insight into Novis' usefulness would be gained from performing an evaluation involving first year students. This evaluation is presented in the following chapter.

6.7 Limitations

While the above evaluation appears positive, there are limitations to this study that are likely to have affected the results obtained. While the examples in this evaluation were systematically chosen, few test many of the more advanced areas of Novis. Aside from the World of Zuul (Section 6.5.3) all use few objects and method calls, presenting little challenge for the layout algorithm and little opportunity to use custom views. While this evaluation has therefore shown that some examples relevant to CS1 can be usefully displayed using Novis, it is not a reliable indication that CS1 examples in general can be displayed effectively.

There may also have been bias introduced, since the author performed the evaluation and rated the examples accordingly. While the examples and ratings were defined empirically, some amount of interpretation is involved in picking a particular rating. Allowing a CS1 educator or student to rate the tool would have removed this bias and potentially performed a more reliable evaluation.

It is also possible that a typical user of the system may not have rearranged the object graph manually when the layout algorithm did not provide good results by default (as in Section 6.5.3.) The inclusion of the custom layout is therefore not necessarily representative of the layout that may be achieved by a particular CS1 student.

Chapter 7

Usability Evaluation

7.1 Introduction

Chapter 6 investigated how well Novis can visualise relevant textbook examples. Another important area is its usability for its target user group (first year undergraduate computer science students.) The evaluation performed here will investigate this through the use of a usability study. The design of the study is outlined in the following section, before presenting the results and performing an analysis. Limitations of the study are examined in detail in section

7.2 Design

7.2.1 Method

Traditionally, five participants has been outlined as a good number for a usability study (Nielsen and Landauer, 1993). However, more recent literature has called this number into question for a thorough result (Faulkner, 2003) – we have therefore chosen twelve as a reasonable number that should provide a good level of confidence.

Twelve first year, undergraduate participants previously familiar with BlueJ were selected for the study. The selection was based on an opportunistic sample – students were chosen from one of the university's computing laboratories and asked if they would like to participate in the study. The study was completed in early March, around half way through the second term. While in the target user group, this sample is likely not representative of first year computer science students as a whole, since it is reasonable to assume that harder working students would generally be more likely to be found in this setting. However, this sample methodology was deemed acceptable since the study aims to test the usability of the tool rather than testing programming ability.

The use of an IDE such as Novis in an educational setting will almost always be preceded by a short introduction to the tool, where the educator will provide a brief demonstration and overview of the features. In order to mimic this, the participants were first given a brief 5-10 minute introduction to Novis, covering the following areas:

- The difference between the traditional class view and the new object view, and how to switch between them;
- How to create instances of classes;
- How to call methods on objects;
- The expanded view of an object (visible by clicking on it);
- The speed and detail sliders, and their operation.

This introduction was covered using a separate project from the one the participants would later use to complete the study.

After the introduction, each participant was provided with a set of pre-written classes and an associated question sheet (on paper.)[1] The question sheet provided the participants with a number of short tasks they had to complete and provide answers for[2]. After completing each task, the participants rated the task on two

[1]Full source code for the project is available in Appendix B.

[2]The question sheet is available in Appendix C

criteria – “Ease of use of Novis”, and “Usefulness of Novis”.

Both criteria were provided on a 5 point Likert scale, with 1 being useless or very difficult, and 5 being very useful and very easy.

7.2.2 Key areas

In order to test the usability of Novis, a number of key areas were identified. Each area was tested at least once within the tasks performed by the participants.

1. Opening a project
2. Object creation
3. Object removal
4. The object’s expanded view
5. The display of fields
6. The custom expanded view for collections
7. Method and constructor invocation; in particular the causal relationship between method invocation and manipulation of the object graph
8. The speed and detail sliders

7.2.3 Tasks

From these key areas, a list of tasks was devised for participants to complete. Some tasks involved interacting with Novis explicitly (to test usability); others involved answering questions about the program. This tested its usefulness in obtaining information about the program through Novis, as well as the usability of the commands required to obtain the answer.

Each of these tasks was constructed to ensure it involved at least one of they key areas as defined above:

- Open the project provided, and create an object of type "Student". How many objects are created in total as a result of instantiating this student? *(Areas 1 and 2)*
- How many fields does this student object have? *(Areas 4 and 5)*
- What is the current value of the "name" field? *(Area 5)*
- Set the student's name to a value of your choice by calling the `setName()` method. Which, if any, field updates as a result of this method call? *(Areas 5 and 7)*
- Set the detail to its highest level. Now set the date of birth of the student to "01/05/1986", first using the `setDOB()` method, then using the `setDOB2()` method. Is there a difference between how these methods work? If so, what is it? *(Areas 5, 7 and 8)*
- Remove the student object. Does any other object disappear also? If so, why do you think this is the case? *(Area 3)*
- Create a `StudentCollection` object. How many objects are created in total this time? *(Area 7)*
- What is the name of the student at position 1 in the `ArrayList`? *(Areas 5 and 6)*
- Set the detail to its highest level. Now call the `setName()` method on the `StudentCollection` object to change the name of the student at index 1 to "Hannah". Does the name change as expected? *(Areas 5, 6, 7 and 8)*
- Try calling the `setName()` method to set the names of the students at other indices. Do these change as expected? If not, can you work out what is wrong without looking at the source code? *(Areas 5, 6 and 7)*

Representation

The tasks as outlined above aim to cover a range of complexities, mirroring typical tasks that a student may be asked to complete in a first year computer science course. Questions 1-4 all cover very basic tasks; it is expected that these are typical tasks in the first few weeks of a CS1 course. Questions 8 and 9 are also rather basic, as they simply require calling a method and examining a result.

Question 5 is more in depth, as it requires a knowledge of how references work – specifically, the difference between setting the fields of an existing object, and replacing an existing object with a new object. Questions 6 and 7 provide a similar amount of depth, they both require a more detailed knowledge of the object lifecycle (when objects may be created by a constructor, and when objects become eligible for garbage collection.) We would expect this material to typically be covered towards the end of the first term of CS1.

Question 10 is much more challenging, as it involves some level of debugging (and presents a piece of buggy code that might be a mistake made by a typical CS1 student.) This might be a typical task set in the second term of a CS1 course, as it involves understanding the operation of a pre-written method in depth to work out where the problem might lie.

All the tasks presented above are reasonably small with respect to size; real CS1 coursework would likely involve writing a significant amount more code than the questions set out above.

7.2.4 Performance metrics

The performance of the tool was measured using three separate metrics – usefulness, difficulty and correctness.

Usefulness

Participants were asked to rate the usefulness of Novis for completing the task (the fact they were rating the usefulness of the *tool* in the context of completing the task was stressed to them, since the wording on the question sheet incorrectly referred to the usefulness of the *task*.)

Difficulty

Participants were also asked to rate the difficulty of using Novis to perform the task (this was correctly reference on the question sheet.) This is distinct from the usefulness of Novis to perform a particular task – Novis could be very easy to use, but ultimately not useful in helping to answer the set tasks.

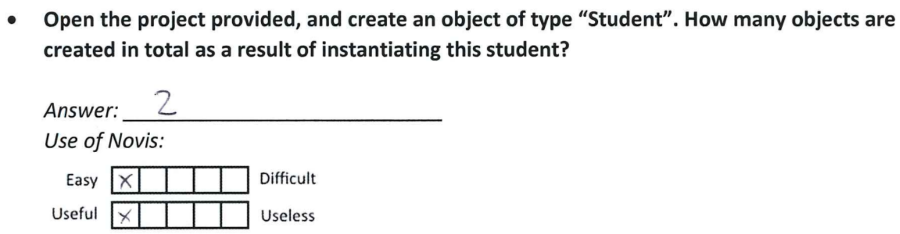
For example, a tool might have an intricate, wizard-like interface for creating new objects (rather than a simple two-click operation.) This may well not impact on the difficulty of using the tool to solve a task, but would certainly make it less useful, especially if the task involved the creation of many objects.

Correctness

Finally, the correctness of the tasks were recorded after the students had completed the study. They were rated either fully correct, partially correct, or incorrect.

7.3 Results

Twelve participants completed the study over a period of three weeks. One candidate was female, the remainder were male. All were first year computer science students studying in their second term at the same university. The participants took on average 20 minutes to complete all of the tasks. A sample question is shown in Figure 84 – a space is provided for an answer, as well as results for the usefulness and difficulty on the 5 point Likert scale.



- **Open the project provided, and create an object of type "Student". How many objects are created in total as a result of instantiating this student?**

Answer: 2

Use of Novis:

Easy [X] [ ] [ ] [ ] [ ] Difficult

Useful [X] [ ] [ ] [ ] [ ] Useless

Figure 84: A sample question from the usability study.

The results for the participants are presented below. Candidate 7 was unwilling to complete the final two questions of the study due to time constraints.

7.3.1 Usefulness

The usefulness is rated from 1-5, with 5 being the most useful.

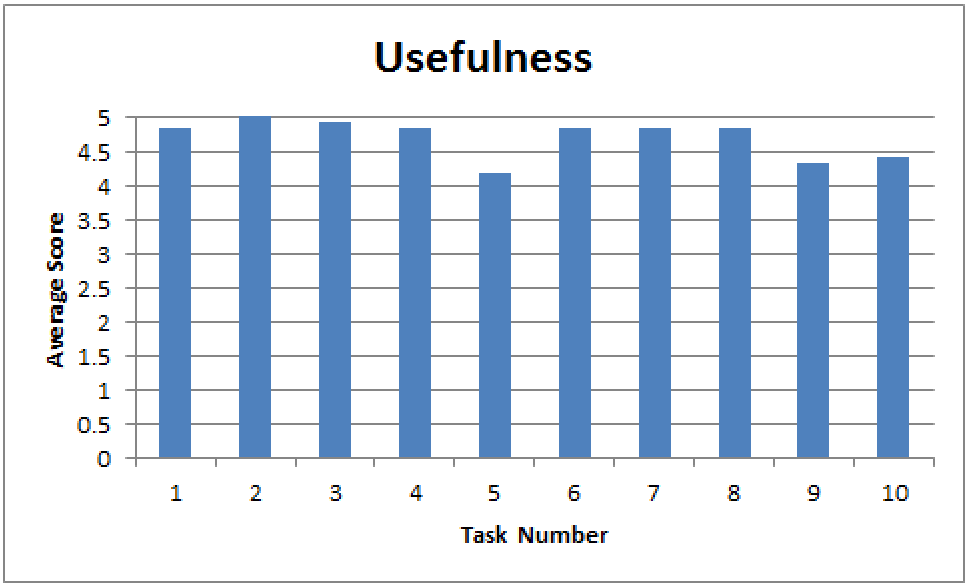


Figure 85: The "usefulness" results from the usability study.

7.3.2 Difficulty

The difficulty is rated similarly, with 1 being the most difficult and 5 being the least difficult.

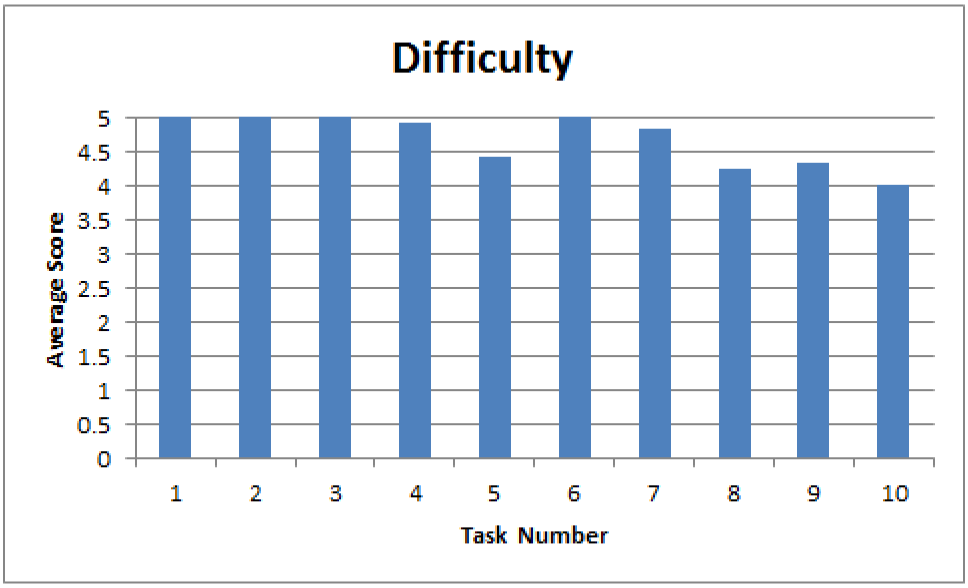


Figure 86: The “difficulty” results from the usability study.

7.3.3 Correctness

In addition to the self-assessment criteria, the usefulness can also be assessed by examining the correctness of the answers. The participants’ answers to the questions were rated as either correct, somewhat correct or incorrect.

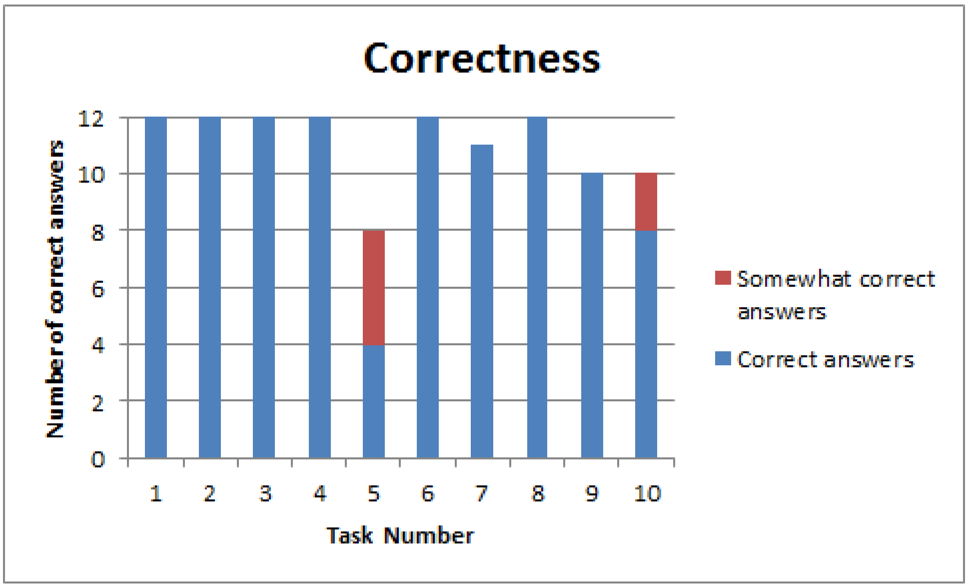


Figure 87: The "correctness" results from the usability study.

7.4 Analysis

7.4.1 Usefulness

96% of the usefulness ratings across all tasks were either very useful or useful, corresponding to a score of 4 or above. The remainder, bar one, received a score of three (indicating Novis was moderately useful for the task.) Questions 5, 9 and 10 had the lowest usability scores overall. However, questions 5 and 10 spanned a range of key areas (7.2.2), all of which were represented in other questions that had high usefulness ratings.

While this shows that there are some particular tasks for which participants certainly found Novis less useful, it does not point to any key area for which students found Novis less useful overall.

7.4.2 Difficulty

The responses show that participants found Novis more difficult to use for solving questions 5, 8, 9 and 10. Questions 5, 9 and 10 required more interaction steps than most other questions, however question 8 simply required showing the expanded view of the relevant object and examining the contents of the name field. The questions that Novis rated highest for on difficulty (with the exception of question 8) also saw Novis rated lowest on usefulness.

The participants indicated that Novis made the majority of the tasks easy to perform. Aside from questions 5, 8, 9 and 10, Novis was consistently rated as easy or very easy.

7.4.3 Correctness

In general participants confidently used Novis; few questions were asked regarding its use throughout the study and no areas of consistent problems were identified. In particular, tasks 1-4 and 6-9 posed little to no problems for the participants – only two cases of incorrect answers were identified (participant 8, question 7 and participant 12, question 9.) The difficulty rating also reflects this result, with only one other participant indicating that it was difficult to use Novis to perform the task (participant 7, question 8).

The majority of participants were also able to use Novis to obtain the correct answer (and reasoning) to question 10. Of particular note was one participant that very quickly (and correctly) noted that whatever parameter was provided, the object at the index 0 in the collection was always being changed. However, before reading the task fully and noting this behaviour was by design, he insisted this was a bug with Novis!

Only one participant indicated that the values changed exactly as expected (the wrong answer.) This particular participant performed the correct interaction steps which did indeed produce two distinct patterns of behaviour, but marked them as identical nonetheless.

All other students noted that something was different to the expected behaviour, though two were unable to explain exactly what the incorrect behaviour was in a coherent form. Again, both of these participants performed the correct environmental interaction steps – in both cases, they performed the same steps three times before reaching their (incorrect) conclusion. One of the aforementioned participants remarked after the third try “I just can’t remember what *[the visualisation looked like]* before”, indicating a possible issue with recall when comparing two states. In total, (73%) of participants were able to use Novis to determine the actual state of the machine and answer the question correctly.

Question 5 clearly posed much more of a challenge to participants, with only 33% obtaining the full, correct answer to the question. 33% were able to identify there was a difference, but could not use the visualisation to figure out any details. The same percentage found no difference between the two methods whatsoever, despite a distinct difference in both the animation and the working of the method (in one case the date object was replaced, in the other the existing object was set to a different value.)

An incorrect answer to a question does not necessarily imply a failure of the tool – it is unrealistic to assume that the use of Novis (or any tool) will automatically cause all students answer all questions correctly. However, repeated failures of the same task with different participants does imply that either the task is too challenging, or the tool is not helpful (too difficult to use or simply not useful) in solving that particular task. In the latter case, patterns in behaviour can help to identify limitations in the tool. In particular, the behaviour of participants in answering question 5 suggests a possible issue with recall. Specifically, it hints that the design of Novis might rely too heavily on recall to effectively aid students in solving these types of problems.

7.5 Comments

Participants were encouraged to leave general comments as to the usefulness of the tool on their question sheet. Most participants also provided verbal feedback, a small number also provided comments on the question sheet. The verbal feedback was generally very positive, with four participants asking for access to the tool to use in their own programming work. One participant commented on a small visual bug with the field view that manifested in her use of the prototype.

The relevant written comments[3] are listed below.

- Which side *[of the detail slider]* is high and which is low?
- Can be difficult to distinguish between lines *[in the ArrayList view that are]* close together.
- Much better than original BlueJ. I love it.
- Clearer *[in Novis than in BlueJ]* to see how objects are created and how methods are implemented.

7.6 Limitations of the study

A number of limitations have been identified during the design of the study. They have been categorised into two groups: principle limitations that arise from the very nature of the study, and incidental limitations that might arise from the specifically methodology used.

7.6.1 Principle limitations

The usability study demonstrates whether the design of the notional machine, and its corresponding implementation, is usable by its target audience. By implication, this should show that the above does not produce any obvious obstacle to learning.

[3]Comments that were not related to the notional machine were omitted: one referred to text colours in use and another referred to placing quotation marks around strings in Java.

However, no claim of a learning effect can be made based on this study. Whilst such a study could yield very interesting results, a study that reliably measured a long-term, longitudinal learning effect was deemed out of scope for this work. A smaller scale comprehension study could have been performed, however this would have been extremely limited. At best, it would only provide a brief snapshot of learning, and does not show (or necessarily even correlate towards) an increased retention across any length of time.

The study also had no "control" system with which the usability was tested. The logical system to use as a control measure was BlueJ, since Novis is based on BlueJ and it is a system with which the students chosen for the study were already familiar. The use of a control system causes its own limitations however, since either two separate groups of students must be used, or the same set of students must be used with a learning effect (of whatever system is tested first.)

In the former case (using a different set of students for the control system), it is not sufficient to choose the sample opportunistically, as then no meaningful comparison could be made between the control system and Novis. Instead, participants would need to be carefully sorted into two groups, using known grades throughout the year to ensure the groups were as closely matched in ability as possible. However, the nature of the students required means that few grades would be available since they would all be in their first year of study. Such a grouping would therefore be rather statistically unreliable, likely requiring a large number of students to bring the uncertainty down to a reasonable margin of error.

In the latter case (using the same set of students), a learning effect would be realised after completing the task with the control system. However, since the participants in this study were all previously familiar with BlueJ and had been using it for many months, the relative learning effect from using this as a control system would likely be small. If such a usability study were to be performed, this would then enable the results from Novis to be meaningfully compared with the BlueJ.

7.6.2 Incidental limitations

As stated in section 7.2.1, the students were chosen opportunistically for the study; willing volunteers were chosen from one of the department's computer laboratories. While the students chosen were limited to those in their first year, the fact they were working in the laboratory signifies they were likely those students that were harder working or more academically able. It is therefore reasonable to assume that the participants for the study represented top calibre CS1 students.

Some of the participants were known to the author, either through teaching or providing assistance with various programming tasks. Since the author also performed the study, some of the participants may have been biased towards Novis, rating it higher than they would have done otherwise. While it was stressed that participants should not rate the tool any differently because of personal affiliation, it would be unreasonable to argue this removed the bias entirely.

The questions themselves may have also proven a limitation. It is possible that questions were misinterpreted by the participants; in this case the ratings provided would not be accurate. We attempted to mitigate this by taking the answers of the questions into account in the analysis. No particular answers appeared to show a complete misinterpretation, but this is hard to verify. It is also possible that some participants became confused altogether by the questions and only performed few, or no tasks with any meaningful result.

The difficulty of the questions was a further limitation in testing the usability of Novis across a range of examples encountered across CS1. While the participants were all in their second term of study, the majority of the tasks posed questions that would typically be part of work completed in the first term. When combined with the assumption that the students were mainly top calibre, this degrades the usefulness of some tasks. Students earlier in the course, or of a lesser ability, would likely have taken longer to devise the answer, potentially making more use of Novis in the process.

Chapter 8

Conclusion & Future Work

8.1 Summary

The work described has proposed a novel, scalable notional machine to aid novices in learning to program. In the introduction to this work, two primary research questions were proposed:

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?

Research question 1 was examined in depth in Chapter 4, where the design of the notional machine was outlined in detail. It is scalable from few objects to many, combines traditionally separate diagrams into a single consistent visualisation, and is able to be represented in a number of formats.

These formats may be implementation based, which could be in the form of a pre-rendered custom animation, or a software tool that is able to take arbitrary code and execute it alongside a visual representation of the notional machine. Novis, also developed as part of this work, is one such example of a software tool. Equally however, the notional machine can be used in an "unplugged" paper based format, either in a very ad-hoc setting (such as scribbling down notes while explaining a program to a student) or a more formal setting (such as drawing various states of the notional machine on a whiteboard in a lecture.) In either format, the notional machine can execute examples from a range of programming textbooks aimed at first year computer science students. However, only the implemented form of the notation has been extensively tested as part of this work.

A thorough investigation into the surrounding literature (Chapter 2) found no other notional machine (nor corresponding implementation) with these properties.

Research question 2 calls for an examination of the specific design considerations and challenges around an implemented form of the notional machine. These design choices are presented and discussed in Chapter 5.

The implementation presented is an existence proof, demonstrating that a software tool (Novis) presenting the dynamic aspects of the notional machine can be developed. To assess whether the implementation is both manageable and useful, two studies were conducted. The first (Chapter 6) involved testing the implementation against a set of systematically chosen textbook examples. A rating was then assigned to the tool based on the layout and the usefulness of each visualisation. The second was a user study, conducted with the target user group of the notional machine (novice, first year undergraduate computer science students) to assess the usability of the implementation against a particular example (Chapter 7). While both evaluations had limitations (sections 6.7 and 7.6), Novis performed well for the studies performed.

8.2 Recommendation for Future Work

8.2.1 Layout algorithm

While the layout algorithm has been shown to work well with a good number of textbook examples, the textbook evaluation (Chapter 6) highlighted some areas where the layout algorithm could be improved. In particular, there were often a large amount of layout crossings, especially with examples that utilised many objects. Existing graphing algorithms were explored as part of this work that would provide an aesthetically improved layout with fewer crossings. However, no graphing algorithm was found that could both effectively minimise crossings and run in real time on an average desktop computer.

8.2.2 Longitudinal Learning Study

While the work here has devised, implemented and assessed the usability of the notional machine, no study was performed to investigate whether the notional machine causes students to learn programming more effectively over time. Such a study is out of scope for this work; however, it nonetheless remains a very interesting prospect. If the aforementioned study were to show an increased learning effect for students using the notional machine outlined here, this would strongly support the hypothesis that one or more of the novel features in use does indeed help improve students' mental model of code.

A smaller scale study on a very short term learning effect was also considered. The idea was discarded however, the results from such a study would at best provide a very weak claim on any possible learning effect.

A larger scale, longitudinal study would need a large sample and would need to be done over a long period of time, preferably over the entirety of the first year of students' learning. Two groups of participants would need to be chosen systematically, ensuring that as far as is practical both groups have the same ability. One group (the control group) would then be taught using traditional tools and explanations. The second group would be taught using the notional machine outlined in this work, both in hand drawn form (in lectures and seminars when explanations are being given), and in implemented form; completing any assignments they have in the Novis environment.

Over the course of the year, a number of assessments designed to test students' understanding of code would be devised and presented to all students simultaneously. The results from these assessments, as well as the results of any programming related course-work, would then be collected and analysed to determine if any significant difference existed.

8.2.3 Additional custom views

Section 4.3.3 introduced the concept of custom views, where expanded objects do not necessarily need to show a list of fields, but can instead show a more interesting and useful representation. In this work the concept of custom views has been introduced, and they have been defined on a small subset of library classes (notably the collection classes, `String` objects and `Date` objects.)

While this is a useful subset of library classes, it is by no means complete, and further custom views should also be defined for other library classes. Different types of custom views should also be considered; objects representing UI elements could for example replicate a thumbnail of their element in their custom view.

8.2.4 Sound

It is possible to add short sounds to Novis whenever any event occurs in the running of code – object creation, object destruction and method calls. There is little in the literature that indicates whether it is possible for such an addition to provide a learning benefit, or equally whether sounds would simply annoy or frustrate users.

A further study in this area could thus be to investigate the influence of sound in the implemented form of the notional machine, gauging the effect is has on first year undergraduate students.

8.2.5 Automatic Scaling

At present, the implementation makes the user responsible for choosing both the speed and the detail level of the visualisation. This provides the user with the most control, but also serves as a potential for distraction – to make a change to the properties of the visualisation, focus is necessarily taken off of the main visualisation area. Such a distraction does not occur in any hand drawn version of the notional machine, since here the educator or user will implicitly choose such "settings" based on the program they are about to "execute".

If there was a method by which the software could reasonably choose the speed and detail settings according to the currently executing program, this mental step could then also be removed from the implemented version of the notional machine.

8.3 Conclusion

This work has answered both research questions set out in the introduction. A notational machine for effectively modelling first year undergraduate Java programs has been developed and presented here. Detailed design choices involving the notional machine have been set out, discussed and justified. An implementation has also been developed as an existence proof and presented in detail.

Through the use of two evaluations, the usefulness of both the model and the corresponding implementation has also been demonstrated. Additionally, a number of possible areas have been outlined for future work on this topic.

Bibliography

A quick introduction to uml sequence diagrams. `http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams`. Accessed: 2016-02-08.

AlgoVis.org (2012). AV catalog.

Baldwin, L. P. and J. Kuljis (2000, June). Visualisation techniques for learning and teaching programming. In *Proceedings of the 22nd International Conference on Information Technology Interfaces, 2000. ITI 2000*, pp. 83–90. IEEE.

Barnes, D. J. and M. Kölling (2011, September). *Objects First with Java: A Practical Introduction Using BlueJ* (5 edition ed.). Boston: Pearson.

Bennedsen, J. and M. E. Caspersen (2007, June). Failure rates in introductory programming. *SIGCSE Bull. 39*(2), 32 – 36.

Brown, M. H. and M. A. Najork (1993). Algorithm animation using 3D interactive graphics. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, UIST '93, New York, NY, USA, pp. 93 – 100. ACM.

Byrne, M. D., R. Catrambone, and J. T. Stasko (1996). *Do Algorithm Animations Aid Learning?*

Cañas, J. J., M. Teresa Bajo, and P. Gonzalvo (1994). Mental models and computer programming. *International Journal of Human-Computer Studies 40*(5), 795 – 811.

Cadenhead, R. (2012, August). *Sams Teach Yourself Java in 21 Days* (6 edition ed.). Indianapolis, Ind: Sams.

Carson, E., I. Parberry, and B. Jensen (2007, March). Algorithm explorer: visualizing algorithms in a 3D multimedia environment. *SIGCSE Bull. 39*(1), 155 – 159.

Chandler, P. and J. Sweller (1991). Cognitive load theory and the format of instruction. *Cognition and instruction 8*(4), 293–332.

du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research 2*, 57–73.

Faulkner, L. (2003). Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers 35*(3), 379–383.

Ford, L. (1993). *How programmers visualize programs.* Citeseer.

Frick, A., A. Ludwig, and H. Mehldau (1995). A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). *894*, 388–403.

Fronk, A., A. Bruckhoff, and M. Kern (2006). 3D visualisation of code structures in java software systems. pp. 145. ACM Press.

Fruchterman, T. M. and E. M. Reingold (1991). Graph drawing by force-directed placement. *Softw., Pract. Exper. 21*(11), 1129–1164.

Gestwicki, P. V. and B. Jayaraman (2004). JIVE: java interactive visualization environment. pp. 226. ACM Press.

Girgis, H. Z., B. Jayaraman, and P. V. Gestwicki (2005). Visualizing errors in object oriented programs. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* OOPSLA '05, New York, NY, USA, pp. 156 – 157. ACM.

Goto, S. (1981, Jan). An efficient algorithm for the two-dimensional placement problem in electrical circuit layout. *Circuits and Systems, IEEE Transactions on 28*(1), 12–18.

Green, T. and M. Petre (1996). Usability analysis of visual programming environments: A ‘ cognitive dimensions ’ framework. *Journal of Visual Languages & Computing 7*(2), 131 – 174.

Greevy, O., M. Lanza, and C. Wysseier (2006). Visualizing feature interaction in 3-d. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005*, pp. 1–6. IEEE.

Gries, D. (2008, February). A principled approach to teaching OO first. *ACM SIGCSE Bulletin 40*, 31.

Gurka, J. and W. Citrin (1996, September). Testing effectiveness of algorithm animation. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pp. 182 –189.

Heath, M. T., J. Etheridge, et al. (1991). Visualizing the performance of parallel programs. *Software, IEEE 8*(5), 29–39.

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

Hundhausen, C., S. A. Douglas, and J. T. Stasko (2002, June). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing 13*(3), 259–290.

ISO (2012). ISO 7000:2012.

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

Kölling, M. (1999a). Teaching object orientation with the blue environment. *Journal of Object Oriented Programming 12*, 14–23.

Kölling, M. (1999b). *The Design of an Object-Oriented Environment and Language for Teaching.* Ph. D. thesis, Basser Department of Computer Science, University of Sydney.

Kölling, M., B. Quig, A. Patterson, and J. Rosenberg (2003, December). The BlueJ system and its pedagogy. *Computer Science Education 13*, 249–268.

Kosara, R. (2007, July). Visualization criticism - the missing link between information visualization and art. In *Information Visualization, 2007. IV '07. 11th International Conference*, pp. 631–636. IEEE.

Liang, Y. D. (2012, June). *Introduction to Java Programming, Comprehensive Version* (9 edition ed.). Harlow: Pearson Education.

Lister, R., E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, et al. (2004). A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, Volume 36, pp. 119–150. ACM.

Lowe, R. (2003, April). Animation and learning: selective processing of information in dynamic graphics. *Learning and Instruction 13*(2), 157–176.

Ma, L., J. Ferguson, M. Roper, and M. Wood (2007, March). Investigating the viability of mental models held by novice programmers. *SIGCSE Bull. 39*(1), 499 – 503.

Malmi, L., V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti (2004). Visual algorithm simulation exercise system with automatic assessment: Trakla2. *Informatics in Education*, 48.

Malmi, L. and A. Korhonen (2004, September). Automatic feedback and resubmissions as learning aid. In *IEEE International Conference on Advanced Learning Technologies, 2004. Proceedings*, pp. 186– 190. IEEE.

Mayer, R. E. (2002). Multimedia learning. In *Psychology of Learning and Motivation*, Volume 41, pp. 85–139. Academic Press.

Mayer, R. E. and R. Moreno (2003). Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist 38*(1), 43–52.

McGrath, M. (2011, August). *Java In Easy Steps 4th Edition* (4th Revised edition edition ed.). Southam, Wickhamshire, U.K.: In Easy Steps.

Montgomery, L., J. H. Cross,II, T. D. Hendrix, and L. A. Barowski (2008). Testing the jGRASP structure identifier with data structure examples from textbooks. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, New York, NY, USA, pp. 198 – 203. ACM.

Moreno, A. and M. S. Joy (2007, July). Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science 178*(0), 51–59.

Moreno, A., N. Myller, E. Sutinen, and M. Ben-Ari (2004). Visualizing programs with jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces*, pp. 373. ACM Press.

Myller, N., R. Bednarik, and A. Moreno (2007, July). Integrating dynamic program visualization into BlueJ: the jeliot 3 extension. In *Seventh IEEE International Conference on Advanced Learning Technologies, 2007. ICALT 2007*, pp. 505–506. IEEE.

Naps, T. L. (1990, February). Algorithm visualization in computer science laboratories. *SIGCSE Bull. 22*(1), 105 – 110.

Naps, T. L. (2005, October). JHAVE: supporting algorithm visualization. *IEEE Computer Graphics and Applications 25*(5), 49–55.

Naps, T. L., G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. n. Vel á zquez-Iturbide (2002, June). Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull. 35*(2), 131 – 152.

Nielsen, J. (1995, December). Ten usability heuristics. *Fremont: Nielsen Norman Group.*, 10.

Nielsen, J. and T. K. Landauer (1993). A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pp. 206–213. ACM.

Parker, Franck, and Ware (1998). Visualization of large nested graphs in 3D: navigation and interaction. *Journal of Visual Languages & Computing 9*(3), 299 – 318.

Pauw, W., E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang (2002). Visualizing the execution of java programs. In S. Diehl (Ed.), *Software Visualization*, Volume 2269, pp. 151–162. Berlin, Heidelberg: Springer Berlin Heidelberg.

Pauw, W. and J. Vlissides (1998). Visualizing object-oriented programs with jinsight. In S. Demeyer and J. Bosch (Eds.), *Object-Oriented Technology: ECOOP ' 98 Workshop Reader*, Volume 1543, pp. 541–542. Berlin, Heidelberg: Springer Berlin Heidelberg.

Pekrun, R., T. Goetz, W. Titz, and R. P. Perry (2002). Academic emotions in students' self-regulated learning and achievement: A program of qualitative and quantitative research. *Educational psychologist 37*(2), 91–105.

Purchase, H. (2000, December). Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers 13*(2), 147–162.

Ragonis, N. and M. Ben-Ari (2005). On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, SIGCSE '05, New York, NY, USA, pp. 226 – 230. ACM.

Ramalingam, V., D. LaBelle, and S. Wiedenbeck (2004, June). Self-efficacy and mental models in learning to program. *SIGCSE Bull. 36*(3), 171–175.

Reiss, S. P. and M. Renieris (2005). Jove: java as it happens. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, New York, NY, USA, pp. 115 – 124. ACM.

Rogers, C. (2011, October). *Sams Teach Yourself Java in 24 Hours* (6 edition ed.). Sams Publishing.

Roman, G. C. and H. C. Cunningham (1990, December). Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering 16*(12), 1361–1373.

Sajaniemi, J., M. Kuittinen, and T. Tikansalo (2008, January). A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *J. Educ. Resour. Comput. 7*(4), 3:1–3:31.

Saraiya, P., C. A. Shaffer, D. S. McCrickard, and C. North (2004). Effective features of algorithm visualizations. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, New York, NY, USA, pp. 382 – 386. ACM.

Schildt, H. (2011, October). *Java, A Beginner's Guide, 5th Edition* (5 edition ed.). New York: McGraw-Hill Osborne.

Schulz, H. J. and H. Schumann (2006, July). Visualizing graphs - a generalized view. In *Tenth International Conference on Information Visualization, 2006. IV 2006*, pp. 166–173. IEEE.

Sorva, J. (2013, July). Notional machines and introductory programming education. *Trans. Comput. Educ. 13*(2), 8:1–8:31.

Sorva, J. and T. Sirkiä (2010). UUhistle. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pp. 49–54. ACM Press.

Stasko, J. et al. (1993a). Do algorithm animations assist learning?: an empirical study and analysis. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, New York, NY, USA, pp. 61 – 66. ACM.

Stasko, J. et al. (1993b, August). Three-dimensional computation visualization. In

, *Proceedings 1993 IEEE Symposium on Visual Languages, 1993*, pp. 100–107. IEEE.

Stasko, J. T. (1990, September). Tango: a framework and system for algorithm animation. *Computer 23*(9), 27–39.

Stasko, J. T., C. Kehoe, and A. Taylor (2001). Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies 54*(2), 265 – 284.

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review 22*(2), 123–138.

Sweller, J., J. J. Van Merrienboer, and F. G. Paas (1998). Cognitive architecture and instructional design. *Educational psychology review 10*(3), 251–296.

Teyseyre, A. R. and M. R. Campo (2009, February). An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics 15*(1), 87–105.

Travers, M. D. (1996). Programming with agents: New metaphors for thinking about computation.

Utting, I., N. Brown, M. Kölling, D. McCall, and P. Stevens (2012). Web-scale data gathering with bluej. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, New York, NY, USA, pp. 1–4. ACM.

Weiss, R. E., D. S. Knowlton, and G. R. Morrison (2002, July). Principles for using animation in computer-based instruction: theoretical heuristics for effective design. *Computers in Human Behavior 18*(4), 465–477.

Wiedenbeck, S., V. Ramalingam, S. Sarasamma, and C. Corritore (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers 11*(3), 255 – 282.

Yan, T. and M. D. Wong (2010). Recent research development in pcb layout. In

Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on, pp. 398–403. IEEE.

Yee, K.-P., D. Fisher, R. Dhamija, and M. Hearst (2001). Animated exploration of dynamic graphs with radial layout. In *infovis*, pp. 43. IEEE.

Appendices

Appendix A

Source code for textbook examples

This appendix contains the full source code for each example used in Chapter 6.

A.1 Motorcycle

```
public class Motorcycle {
    String make;
    String color;
    boolean engineState;
    void startEngine() {
        if (engineState == true)
            System.out.println("");
        else {
            engineState = true;
            System.out.println("");
        }
    }
```

```
    void showAtts() {
        System.out.println("This motorcycle is a "
            + color + " " + make);
        if (engineState == true)
            System.out.println("The engine is on.");
        else System.out.println("The engine is off.");
    }

    public static void main (String args[]) {
        Motorcycle m = new Motorcycle();
        m.make = "RZ350";
        m.color = "yellow";
        //System.out.println("Calling showAtts...");
        m.showAtts();
        //System.out.println("--------");
        //System.out.println("Starting engine...");
        m.startEngine();
        //System.out.println("--------");
        //System.out.println("Calling showAtts...");
        m.showAtts();
        //System.out.println("--------");
        //System.out.println("Starting engine...");
        m.startEngine();
    }
}
```

A.2 Modem

```
public class AcousticModem extends Modem {
    String method = "acoustic connection";

    public void connect() {
```

```
        System.out.println("Connecting to the Internet
            ...");
        System.out.println("Using a " + method);
    }
}
```

```
public class CableModem extends Modem {
    String method = "cable connection";

    public void connect() {
        System.out.println("Connecting to the Internet
            ...");
        System.out.println("Using a " + method);
    }
}
```

```
public class DslModem extends Modem {
    String method = "DSL phone connection";

    public void connect() {
        System.out.println("Connecting to the Internet
            ...");
        System.out.println("Using a " + method);
    }
}
```

```
public class Modem {
    int speed;

    public void displaySpeed() {
        System.out.println("Speed: " + speed);
    }
}
```

```
public class ModemTester {
    public static void main(String[] args) {
        Modem surfBoard = new CableModem();
        Modem gateway = new DslModem();
        surfBoard.speed = 500000;
        gateway.speed = 400000;
        System.out.println("Trying the cable modem:");
        surfBoard.displaySpeed();
        //surfBoard.connect();
        System.out.println("Trying the DSL modem:");
        gateway.displaySpeed();
       // gateway.connect();
    }
}
```

A.3 Vehicle

```
class Vehicle {
  int passengers=7; // number of passengers
  int fuelcap=16;     // fuel capacity in gallons
  int mpg=21;         // fuel consumption in miles per
     gallon

  // Display the range.
  void range() {
    System.out.println("Range is " + fuelcap * mpg);
  }
}
```

```
class VehicleDemo {
  public static void main(String args[]) {
    Vehicle minivan = new Vehicle();
    int range;
```

```
    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelcap = 16;
    minivan.mpg = 21;

    // compute the range assuming a full tank of gas
    range = minivan.fuelcap * minivan.mpg;

    System.out.println("Minivan can carry " + minivan.
       passengers +
                       " with a range of " + range);
  }
}
```

A.4 Clock Display

```
/**
 * The ClockDisplay class implements a digital clock
    display for a
 * European-style 24 hour clock. The clock shows hours
    and minutes. The
 * range of the clock is 00:00 (midnight) to 23:59 (one
    minute before
 * midnight).
 *
 * The clock display receives "ticks" (via the timeTick
    method) every minute
 * and reacts by incrementing the display. This is done
    in the usual clock
 * fashion: the hour increments when the minutes roll
    over to zero.
```

```
 *
 * @author Michael K\"{o}lling and David J. Barnes
 * @version 2011.07.31
 */
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString;    // simulates the
       actual display

    /**
     * Constructor for ClockDisplay objects. This
        constructor
     * creates a new clock set at 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    /**
     * Constructor for ClockDisplay objects. This
        constructor
     * creates a new clock set at the time specified by
        the
     * parameters.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
```

```
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }

    /**
     * This method should get called once every minute -
         it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled
            over!
            hours.increment();
        }
        updateDisplay();
    }

    /**
     * Set the time of the display to the specified hour
         and
     * minute.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }

    /**
```

```
     * Return the current time of this display in the
        format HH:MM.
     */
    public String getTime()
    {
        return displayString;
    }

    /**
     * Update the internal string that represents the
        display.
     */
    private void updateDisplay()
    {
        String h = hours.getDisplayValue();
        String m = minutes.getDisplayValue();
        displayString = internal.Updator.getTimeString(h
           ,m);
    }
}
```

```
/**
 * Write a description of class MyDisplay here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MyDisplay
{
    private NumberDisplay disp1;
    private NumberDisplay disp2;
    private NumberDisplay disp3;
    private NumberDisplay disp4;
```

```
    private NumberDisplay disp5;
    private NumberDisplay disp6;
    private NumberDisplay disp7;
    private NumberDisplay disp8;

    public MyDisplay() {
        disp1 = new NumberDisplay(5);
        disp2 = new NumberDisplay(5);
        disp3 = new NumberDisplay(5);
        disp4 = new NumberDisplay(5);
        disp5 = new NumberDisplay(5);
        disp6 = new NumberDisplay(5);
        disp7 = new NumberDisplay(6);
        disp8 = new NumberDisplay(2);
    }

    public void go() {
        for(int i=0 ; i<10 ; i++)
            disp1.getValue();
        disp2.getValue();
        disp3.getValue();
        disp4.getValue();
        for(int i=0 ; i<5 ; i++)
            disp5.increment();
        disp6.getValue();
        disp7.getValue();
        for(int i=0 ; i<10 ; i++)
            disp8.increment();
    }

}
```

```
/**
 * The NumberDisplay class represents a digital number
   display that can hold
 * values from zero to a given limit. The limit can be
   specified when creating
 * the display. The values range from zero (inclusive)
   to limit-1. If used,
 * for example, for the seconds on a digital clock, the
   limit would be 60,
 * resulting in display values from 0 to 59. When
   incremented, the display
 * automatically rolls over to zero when reaching the
   limit.
 *
 * @author Michael K\"{o}lling and David J. Barnes
 * @version 2011.07.31
 */
public class NumberDisplay
{
    private int limit;
    private int value;

    /**
     * Constructor for objects of class NumberDisplay.
     * Set the limit at which the display rolls over.
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }
        public void c(int a, int b, int c) {
    }
```

```
/**
 * Return the current value.
 */
public int getValue()
{
    return value;
}

/**
 * Return the display value (that is, the current
   value as a two-digit
 * String. If the value is less than ten, it will be
    padded with a leading
 * zero).
 */
public String getDisplayValue()
{
    if(value < 10) {
        return internal.Updator.pad(value, true);
    }
    else {
        return internal.Updator.pad(value, false);
    }
}

/**
 * Set the value of the display to the new specified
    value. If the new
 * value is less than zero or over the limit, do
   nothing.
 */
public void setValue(int replacementValue)
```

```
    {
        if((replacementValue >= 0) && (replacementValue
           < limit)) {
            value = replacementValue;
        }
    }

    /**
     * Increment the display value by one, rolling over
       to zero if the
     * limit is reached.
     */
    public void increment()
    {
        value = (value + 1) % limit;
    }
}
```

A.5 Circles

```
public class SimpleCircle
{
  double radius;

  public SimpleCircle()
  {
    this.radius = 1.0D;
  }

  public SimpleCircle(double newRadius)
  {
    this.radius = newRadius;
  }
```

```
  public double getArea()
  {
    return this.radius * this.radius * 3.141592653589793
       D;
  }

  public double getPerimeter()
  {
    return 2.0D * this.radius * 3.141592653589793D;
  }

  public void setRadius(double newRadius)
  {
    this.radius = newRadius;
  }
}
```

```
public class TestSimpleCircle {
private static SimpleCircle circle1;
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    circle1 = new SimpleCircle();
    //System.out.println("The area of the circle of
       radius "
      //+ circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    SimpleCircle circle2 = new SimpleCircle(25);
    //System.out.println("The area of the circle of
       radius "
      //+ circle2.radius + " is " + circle2.getArea());
```

```
    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    //System.out.println("The area of the circle of
       radius "
      //+ circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    //System.out.println("The area of the circle of
       radius "
      //+ circle2.radius + " is " + circle2.getArea());
  }
}
```

A.6 Car

```
public class Car
{
    private String maker ;
    private String color ;
    private String bodyType ;

    public Car()
    {
        maker = "Porsche";
        color = "Silver";
        bodyType = "Coupe";
    }

    private String accelerate()
    {
        String motion = "Accelerating..." ;
```

```
        return motion;
    }

    public void setCar( String brand, String paint,
       String style )
    {
        maker = brand ;
        color = paint;
        bodyType = style;
    }

    public void getCar()
    {
        System.out.println( maker +" paint is " + color
           );
        System.out.println( maker + " style is " +
           bodyType );
        System.out.println( maker + " is " + accelerate
           () +"\n" );
    }
}
```

```
public class Constructor
{
    public static void main(String[] args)
    {
        Car Porsche = new Car();
        //Porsche.getCar();

        Car Ferrari = new Car();
        Ferrari.setCar("Ferrari","Red","Sport");
        //Ferrari.getCar();
    }
```

```
}
```

A.7 Giftshop

```
import java.util.*;

public class Storefront2 {
    private ArrayList catalog = new ArrayList();

    public void addItem(String id, String name, String
       price,
        String quant, String discount) {

        Item2 it = new Item2(id, name, price, quant,
           discount);
        catalog.add(it);
    }

    public Item2 getItem(int i) {
        return (Item2)catalog.get(i);
    }

    public int getSize() {
        return catalog.size();
    }

    public void sort() {
        Collections.sort(catalog);
    }
}
```

```
public class GiftShop2 {
   public static void main(String[] arguments) {
```

```
        Storefront2 store = new Storefront2();
        store.addItem("C01", "MUG", "9.99", "150", "
           FALSE");
        store.addItem("C02", "LG MUG", "12.99", "82", "
           TRUE");
        store.addItem("C03", "MOUSEPAD", "10.49", "800",
            "FALSE");
        store.addItem("D01", "T SHIRT", "16.99", "90", "
           TRUE");
        store.sort();

        for (int i = 0; i < store.getSize(); i++) {
            Item2 show = (Item2)store.getItem(i);
            System.out.println("\nItem ID: " + show.
               getId() +
                "\nName: " + show.getName() +
                "\nRetail Price: $" + show.getRetail() +
                "\nPrice: $" + show.getPrice() +
                "\nQuantity: " + show.getQuantity());
        }
    }
}
```

A.8 BookDemo

```
public class Book {
  private String title;
  private String author;
  private int pubDate;

  Book(String t, String a, int d) {
    title = t;
    author = a;
```

```
    pubDate = d;
  }

  public void show() {

  }
}
```

```

public class BookDemo {

    private Book books[] = new Book[5];

    public static void main(String args[]) {
        new BookDemo().go();
    }

    public void go() {
        books[0] = new Book("Java: A Beginner's Guide",
            "Schildt", 2011);
        books[1] = new Book("Java: The Complete
           Reference",
            "Schildt", 2011);
        books[2] = new Book("The Art of Java",
            "Schildt and Holmes", 2003);
        books[3] = new Book("Red Storm Rising",
            "Clancy", 1986);
        books[4] = new Book("On the Road",
            "Kerouac", 1955);

        for(int i=0; i < books.length; i++) books[i].
           show();
    }
```

```
public void x() {
if (scheme.equals("file") || scheme.equals("jar")) {
    InputStream stream = getInputStream(uri);
    stream.close();
    isConnected = true;
    contentType = MediaUtils.filenameToContentType(
       uriString); // We need to provide at least
       something
}

if (isConnected) {
    // Check whether content may be played.
    // For WAV use file signature, since it can
       detect audio format
    // and we can fail sooner, then doing it at
       runtime.
    // This is important for AudioClip.
    if (MediaUtils.CONTENT_TYPE_WAV.equals(
       contentType)) {
        contentType =
           getContentTypeFromFileSignature(uri);
        if (!MediaManager.canPlayContentType(
           contentType)) {
            isMediaSupported = false;
        }
    } else {
        if (contentType == null || !MediaManager.
           canPlayContentType(contentType)) {
            // Try content based on file name.
            contentType = MediaUtils.
               filenameToContentType(uriString);
```

```
                    if (Locator.DEFAULT_CONTENT_TYPE.equals(
                       contentType)) {
                        // Try content based on file
                           signature.
                        contentType =
                           getContentTypeFromFileSignature(
                           uri);
                    }

                    if (!MediaManager.canPlayContentType(
                       contentType)) {
                        isMediaSupported = false;
                    }
                }
            }

            // Break as connection has been made and media
               type checked.
            break;
        }
    }
}
```

A.9 World of Zuul

```
/**
 * This class is part of the "World of Zuul" application
    .
 * "World of Zuul" is a very simple, text based
    adventure game.
 *
 * This class holds information about a command that was
     issued by the user.
```

```
 * A command currently consists of two strings: a
    command word and a second
 * word (for example, if the command was "take map",
    then the two strings
 * obviously are "take" and "map").
 *
 * The way this is used is: Commands are already checked
     for being valid
 * command words. If the user entered an invalid command
     (a word that is not
 * known) then the command word is <null>.
 *
 * If the command had only one word, then the second
    word is <null>.
 *
 * @author  Michael K\"{o}lling and David J. Barnes
 * @version 2011.08.08
 */

public class Command
{
    private String commandWord;
    private String secondWord;

    /**
     * Create a command object. First and second word
        must be supplied, but
     * either one (or both) can be null.
     * @param firstWord The first word of the command.
        Null if the command
     *                  was not recognised.
     * @param secondWord The second word of the command.
     */
```

```
    public Command(String firstWord, String secondWord)
    {
        commandWord = firstWord;
        this.secondWord = secondWord;
    }

    /**
     * Return the command word (the first word) of this
        command. If the
     * command was not understood, the result is null.
     * @return The command word.
     */
    public String getCommandWord()
    {
        return commandWord;
    }

    /**
     * @return The second word of this command. Returns
        null if there was no
     * second word.
     */
    public String getSecondWord()
    {
        return secondWord;
    }

    /**
     * @return true if this command was not understood.
     */
    public boolean isUnknown()
    {
        return (commandWord == null);
```

```
    }

    /**
     * @return true if the command has a second word.
     */
    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }
}
```

```
/**
 * This class is part of the "World of Zuul" application
    .
 * "World of Zuul" is a very simple, text based
   adventure game.
 *
 * This class holds an enumeration of all command words
   known to the game.
 * It is used to recognise commands as they are typed in
    .
 *
 * @author  Michael K\"{o}lling and David J. Barnes
 * @version 2011.08.08
 */

public class CommandWords
{
    // a constant array that holds all valid command
       words
    private static final String[] validCommands = {
        "go", "quit", "help"
    };
```

```
    /**
     * Constructor - initialise the command words.
     */
    public CommandWords()
    {
        // nothing to do at the moment...
    }

    /**
     * Check whether a given String is a valid command
       word.
     * @return true if it is, false if it isn't.
     */
    public boolean isCommand(String aString)
    {
        for(int i = 0; i < validCommands.length; i++) {
            if(validCommands[i].equals(aString))
                return true;
        }
        // if we get here, the string was not found in
           the commands
        return false;
    }

    /**
     * Print all valid commands to System.out.
     */
    public void showAll()
    {
        for(String command: validCommands) {
            System.out.print(command + "  ");
        }
```

```
        System.out.println();
    }
}
```

```
/**
 *  This class is the main class of the "World of Zuul"
    application.
 *  "World of Zuul" is a very simple, text based
    adventure game.  Users
 *  can walk around some scenery. That's all. It should
    really be extended
 *  to make it more interesting!
 *
 *  To play this game, create an instance of this class
    and call the "play"
 *  method.
 *
 *  This main class creates and initialises all the
    others: it creates all
 *  rooms, creates the parser and starts the game.  It
    also evaluates and
 *  executes the commands that the parser returns.
 *
 * @author  Michael K\"{o}lling and David J. Barnes
 * @version 2011.08.08
 */

public class Game
{
    private Parser parser;
    private Room currentRoom;

    /**
```

```
     * Create the game and initialise its internal map.
     */
    public Game()
    {
        createRooms();
parser = new Parser();
    }

    /**
     * Create all the rooms and link their exits
        together.
     */
    private void createRooms()
    {
        Room outside, theater, pub, lab, office;

        // create the rooms
        outside = new Room("outside the main entrance of
            the university");
        theater = new Room("in a lecture theater");
        pub = new Room("in the campus pub");
        lab = new Room("in a computing lab");
        office = new Room("in the computing admin office
           ");

        // initialise room exits
        outside.setExit("east", theater);
        outside.setExit("south", lab);
        outside.setExit("west", pub);

        theater.setExit("west", outside);

        pub.setExit("east", outside);
```

```
        lab.setExit("north", outside);
        lab.setExit("east", office);

        office.setExit("west", lab);

        currentRoom = outside;  // start game outside
    }

    /**
     *  Main play routine.  Loops until end of play.
     */
    public void play()
    {
        printWelcome();

        // Enter the main command loop.  Here we
           repeatedly read commands and
        // execute them until the game is over.

        boolean finished = false;
        while (! finished) {
            Command command = parser.getCommand();
            finished = processCommand(command);
        }
        System.out.println("Thank you for playing.  Good
            bye.");
    }

    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
```

```
{
    System.out.println();
    System.out.println("Welcome to the World of Zuul
       !");
    System.out.println("World of Zuul is a new,
       incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help
       .");
    System.out.println();
    System.out.println(currentRoom.
       getLongDescription());
}

/**
 * Given a command, process (that is: execute) the
    command.
 * @param command The command to be processed.
 * @return true If the command ends the game, false
    otherwise.
 */
private boolean processCommand(Command command)
{
    boolean wantToQuit = false;

    if(command.isUnknown()) {
        System.out.println("I don't know what you
           mean...");
        return false;
    }

    String commandWord = command.getCommandWord();
    if (commandWord.equals("help")) {
        printHelp();
```

```
        }
        else if (commandWord.equals("go")) {
            goRoom(command);
        }
        else if (commandWord.equals("quit")) {
            wantToQuit = quit(command);
        }
        // else command not recognised.
        return wantToQuit;
    }

    // implementations of user commands:

    /**
     * Print out some help information.
     * Here we print some stupid, cryptic message and a
       list of the
     * command words.
     */
    private void printHelp()
    {
        System.out.println("You are lost. You are alone.
            You wander");
        System.out.println("around at the university.");
        System.out.println();
        System.out.println("Your command words are:");
        parser.showCommands();
    }

    /**
     * Try to in to one direction. If there is an exit,
       enter the new
     * room, otherwise print an error message.
```

```
     */
    private void goRoom(Command command)
    {
        if(!command.hasSecondWord()) {
            // if there is no second word, we don't know
                where to go...
            System.out.println("Go where?");
            return;
        }

        String direction = command.getSecondWord();

        // Try to leave current room.
        Room nextRoom = currentRoom.getExit(direction);

        if (nextRoom == null) {
            System.out.println("There is no door!");
        }
        else {
            currentRoom = nextRoom;
            System.out.println(currentRoom.
               getLongDescription());
        }
    }

    /**
     * "Quit" was entered. Check the rest of the command
         to see
     * whether we really quit the game.
     * @return true, if this command quits the game,
        false otherwise.
     */
    private boolean quit(Command command)
```

```
    {
        if(command.hasSecondWord()) {
            System.out.println("Quit what?");
            return false;
        }
        else {
            return true;  // signal that we want to quit
        }
    }
}
```

```
import java.util.Scanner;

/**
 * This class is part of the "World of Zuul" application
    .
 * "World of Zuul" is a very simple, text based
   adventure game.
 *
 * This parser reads user input and tries to interpret
   it as an "Adventure"
 * command. Every time it is called it reads a line from
    the terminal and
 * tries to interpret the line as a two word command. It
    returns the command
 * as an object of class Command.
 *
 * The parser has a set of known command words. It
   checks user input against
 * the known commands, and if the input is not one of
   the known commands, it
 * returns a command object that is marked as an unknown
    command.
```

```
 *
 * @author  Michael K\"{o}lling and David J. Barnes
 * @version 2011.08.08
 */
public class Parser
{
    private CommandWords commands;  // holds all valid
       command words
    private Scanner reader;         // source of command
        input

    /**
     * Create a parser to read from the terminal window.
     */
    public Parser()
    {
        commands = new CommandWords();
        reader = new Scanner(System.in);
    }

    /**
     * @return The next command from the user.
     */
    public Command getCommand()
    {
        String inputLine;   // will hold the full input
           line
        String word1 = null;
        String word2 = null;

        System.out.print("> ");     // print prompt

        inputLine = reader.nextLine();
```

```
        // Find up to two words on the line.
        Scanner tokenizer = new Scanner(inputLine);
        if(tokenizer.hasNext()) {
            word1 = tokenizer.next();      // get first
               word
            if(tokenizer.hasNext()) {
                word2 = tokenizer.next();      // get
                   second word
                // note: we just ignore the rest of the
                   input line.
            }
        }

        // Now check whether this word is known. If so,
           create a command
        // with it. If not, create a "null" command (for
            unknown command).
        if(commands.isCommand(word1)) {
            return new Command(word1, word2);
        }
        else {
            return new Command(null, word2);
        }
    }

    /**
     * Print out a list of valid command words.
     */
    public void showCommands()
    {
        commands.showAll();
    }
```

```
}
```

```
import java.util.Set;
import java.util.HashMap;

/**
 * Class Room - a room in an adventure game.
 *
 * This class is part of the "World of Zuul" application
    .
 * "World of Zuul" is a very simple, text based
    adventure game.
 *
 * A "Room" represents one location in the scenery of
    the game.  It is
 * connected to other rooms via exits.  For each
    existing exit, the room
 * stores a reference to the neighboring room.
 *
 * @author  Michael K\"{o}lling and David J. Barnes
 * @version 2011.08.08
 */

public class Room
{
    private String description;
    private HashMap<String, Room> exits;        //
       stores exits of this room.

    /**
     * Create a room described "description". Initially,
        it has
```

```
     * no exits. "description" is something like "a
       kitchen" or
     * "an open court yard".
     * @param description The room's description.
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<String, Room>();
    }

    /**
     * Define an exit from this room.
     * @param direction The direction of the exit.
     * @param neighbor  The room to which the exit leads
       .
     */
    public void setExit(String direction, Room neighbor)
    {
        exits.put(direction, neighbor);
    }

    /**
     * @return The short description of the room
     * (the one that was defined in the constructor).
     */
    public String getShortDescription()
    {
        return description;
    }

    /**
     * Return a description of the room in the form:
```

```
     *     You are in the kitchen.
     *     Exits: north west
     * @return A long description of this room
     */
    public String getLongDescription()
    {
        return "You are " + description + ".\n" +
           getExitString();
    }

    /**
     * Return a string describing the room's exits, for
        example
     * "Exits: north west".
     * @return Details of the room's exits.
     */
    private String getExitString()
    {
        String returnString = "Exits:";
        Set<String> keys = exits.keySet();
        for(String exit : keys) {
            returnString += " " + exit;
        }
        return returnString;
    }

    /**
     * Return the room that is reached if we go from
        this room in direction
     * "direction". If there is no room in that
        direction, return null.
     * @param direction The exit's direction.
     * @return The room in the given direction.
```

```
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }
}
```

A.10 Student

```
import java.util.Date;

public class Student {
    private String name;
    private int id;
    private Date dateCreated;

    public Student(int id, String newName) {
        this.name = name;
        dateCreated = new Date();
    }

    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    public Date getDateCreated() {
        return dateCreated;
    }
}
```

Appendix B

Source code for usability study

This appendix contains the full source code for the project used in the usability study, in Chapter 7.

```
import java.util.*;
import java.text.*;

public class Student
{
    private String name;
    private Date dateOfBirth;

    public Student() {

        dateOfBirth = new Date(0);

    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public Date getDOB() {
        return dateOfBirth;
    }

    public void setDOB(String dob) {
        try {
            this.dateOfBirth = new SimpleDateFormat("dd/
               MM/yyyy").parse(dob);
        }
        catch(Exception ex) {
            //Never mind.
        }
    }

    public void setDOB2(String dob) {
        try {
            dateOfBirth.setTime(new SimpleDateFormat("dd
               /MM/yyyy").parse(dob).getTime());
        }
        catch(Exception ex) {
            //Never mind.
        }
    }
}
```

```
import java.util.*;

public class StudentCollection
{
```

```
    private ArrayList<Student> students;

    /**
     * Constructor for objects of class
        StudentCollection
     */
    public StudentCollection()
    {
        students = new ArrayList<>();
        Student s1 = new Student();
        s1.setName("Bob");
        students.add(s1);
        Student s2 = new Student();
        s2.setName("Mary");
        students.add(s2);
        Student s3 = new Student();
        s3.setName("Alice");
        students.add(s3);
    }

    public void setName(int arrayListIndex, String
       newName) {
        students.get(0).setName(newName);
    }
}
```

Appendix C

Question sheet for usability study

This appendix contains the questionnaire used in the usability study, in Chapter 7.

Participant ID:

Novis – Usability Study

Thank you for agreeing to take part in this short study, which aims to evaluate the effectiveness of Novis, a novel visualisation system built into the BlueJ IDE.

You will first be given a brief introduction to the tool (around 5-10 minutes), and then asked to complete a series of tasks given below. After each task, please note the difficulty of using Novis to perform the task (from easy to hard) and the usefulness of the task (from useful to useless) on the scale provided. If you have any other particular comments about each task, please note those down as well.

- **Open the project provided, and create an object of type "Student". How many objects are created in total as a result of instantiating this student?**

  Answer: ______________________________

  Use of Novis:

Easy	☐	☐	☐	☐	☐	Difficult
Useful	☐	☐	☐	☐	☐	Useless

- **How many fields does this student object have?**

  Answer: ______________________________

  Use of Novis:

Easy	☐	☐	☐	☐	☐	Difficult
Useful	☐	☐	☐	☐	☐	Useless

- **What is the current value of the "name" field?**

  Answer: ______________________________

  Use of Novis:

Easy	☐	☐	☐	☐	☐	Difficult
Useful	☐	☐	☐	☐	☐	Useless

- **Set the student's name to a value of your choice by calling the setName() method. What, if any, field updates as a result of this method call?**

  Answer: ______________________________

  Use of Novis:

Easy	☐	☐	☐	☐	☐	Difficult
Useful	☐	☐	☐	☐	☐	Useless

  Please turn over to continue

Participant ID:

- **Set the detail to its highest level. Now set the date of birth of the student to "01/05/1986", first using the setDOB() method, then using the setDOB2() method. Is there a difference between how these methods work? If so, what is it?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

- **Remove the student object. Does any other object disappear also? If so, why do you think this is the case?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

- **Create a "StudentCollection" object. How many objects are created in total this time?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

- **What is the name of the student at position 1 in the ArrayList?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

- **Set the detail to its highest level. Now call the setName() method on the "StudentCollection" object to change the name of the student at index 1 to "Hannah". Does the name change as expected?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

- **Try calling the setName() method to set the names of the students at other indices. Do these change as expected? If not, can you work out what is wrong without looking at the source code?**
  Answer: ____________________________
  Use of Novis:

Easy						Difficult
Useful						Useless

Appendix D

Publications

Berry, M. (2012). The design and implementation of a notional machine for teaching introductory programming. In *Proceedings of the Psychology of Programming Interest Group Annual Conference 2012*, PPIG '12, pp. 189.

Berry, M. and M. Kölling (2013). The design and implementation of a notional machine for teaching introductory programming. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, WiPSE '13, New York, NY, USA, pp. 25–28. ACM.

Berry, M. and M. Kölling (2014). The state of play: A notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation &; Technology in Computer Science Education*, ITiCSE '14, New York, NY, USA, pp. 21–26. ACM.

Berry, M. and M. Kölling (2016). Novis: A notional machine implementation for teaching introductory programming. In *Proceedings of the 2016 Conference on Learning &; Teaching in Computer and Engineering*, LaTiCE '16, pp. To appear.