More than a good story — can you really teach programming through storytelling?

Roger McDermott, Gordon Eccleston & Garry Brindley

Published online: 15 Dec 2015.

Submit your article to this journal

Article views: 16

View related articles

More than a good story – can you really teach programming through storytelling?

Roger McDermott, Gordon Eccleston, Garry Brindley
<rm@comp.rgu.ac.uk>, <ge@comp.rgu.ac.uk>, <gb@comp.rgu.ac.uk>
The Robert Gordon University, School of Computing, Aberdeen, UK

Abstract

The difficulties that students have acquiring programming skills are compounded when they enter a course of study with little confidence in their own ability to use symbolic reasoning. The idea, therefore, that programming should be understood primarily as an algorithmic process often produces severe anxiety and a consequent rapid disengagement with the subject. The recent development of visual programming environments has led to the claim that this algorithmic metaphor can be replaced, at least initially, by one that draws on a correspondence between programming and storytelling. It is asserted that this allows more productive scaffolding to occur around students' prior experience and consequently that anxiety is reduced and learning is enhanced. This paper investigates such a claim in the context of an introductory programming module taught to first year Computing undergraduates at the Robert Gordon University. It also examines the problem of transition to more conventional code-based environments.

Keywords

Programming Metaphors, Algorithm, Storytelling, Alice, Transition.

1. Introduction

At its most basic, the traditional approach to learning the elements of a programming language focuses on assimilating the different structural elements of the language, before trying to integrate these with skills involving the analysis and decomposition of problems into their component parts. This methodology works reasonably well when students have had substantial exposure to such ideas at school (usually through the medium of mathematics), but it has become apparent that, for whatever reason, many students entering an undergraduate Computing programme, such as the one at the Robert Gordon University, may not arrive at university with the required degree of experience in these practices. The difficulty in acquiring fluency in a programming language, especially the modern types of languages commonly taught at university level, is generally seen as a major obstacle for new undergraduates and such a perception of difficulty is commonly cited (Morrison et al, 2003; McCracken et al, 2001) as a reason for disengagement with the subject as a whole.

In order to address the impact of these factors, the School of Computing at the Robert Gordon University undertook a major redevelopment of its introductory programming module around the Alice programming tool (Dann et al, 2006; Cooper et al, 2000; Cooper et al, 2003). Developed at Carnegie-Mellon University, USA, Alice is a multimedia

programming environment in which students populate a 3D virtual world with a range of different graphical objects such as animals, knights, and ninjas. These can be programmed to follow instructions from the student and the result of code execution is to produce an animation in which the objects act out the programmed scenario.

Among the many intriguing claims (Cooper et al, 2003) made for this approach is that the underlying subject metaphor for the process can be can be transformed from one with a logico-mathematical (i.e. algorithmic) basis to a narrative-cinematographic (i.e. storytelling) one. It is asserted that this description of programming activities in terms of storytelling rather than formal algorithmic design is more directly accessible to, and engages more positively with, the prior experience of students. This, in turn, promotes more productive scaffolding and easier assimilation of relevant techniques.

These positive claims have not, however, gone unchallenged. Powers et al (2007) contend that, whatever the short-term gains for some at-risk students, deficiencies in Alice's design framework coupled with complete freedom from syntax errors mean that the inherent difficulties of the subject may be simply postponed until subsequent study, and, moreover, that adverse affective reaction at this point is just as damaging to further progression.

This paper reports on an investigation of the use of Alice in the context of the RGU introductory programming module and considers general lessons that have been learned with respect to teaching elementary programming skills. It seeks to comment on two specific claims relating to the use of the user interface to prevent syntax errors and the notion that narrative may provide a better conceptual framework for the subject than that of algorithm. It also addresses the challenging issue of the transition to programming languages such as Java which have more conventional (text-based) development environments.

2. Background

The case-study detailed here was carried out in a compulsory introductory programming module taught to all first year computing undergraduates in the School of Computing at the Robert Gordon University. The intake was made up of students studying for a number of different BSc degrees ranging from the more technically challenging Computer Science degree through specialist Graphics and Animation, and Internet and Multimedia degrees to a BSc in Computing for Business and eCommerce. About eighty students registered for the undergraduate programme, the majority of these being Scottish school-leavers but also including a number of mature students who entered directly or via the pre-semester access programme.

Prior to the current session, the module was taught as a fairly standard introduction to structured programming using Java. Initial emphasis was placed on the acquisition of the correct rules of syntax, and use of appropriate data types, followed by basic control structures. Explicit reference to elementary ideas about object orientation (classes, objects, etc) was only made at the end of the module (an objects-late approach). The majority of programming examples involved coding simple mathematical or lexicographical algorithms

(e.g. finding averages, sorting strings), such programs being written in a text editor rather than an integrated development environment. On completion, students undertook a second programming module which concentrated on using object oriented elements of the language in a more systematic manner.

The introductory module had for some time been considered problematic. Both student engagement and student achievement were historically thought poor despite the introduction of a number of pedagogical innovations to counter the situation. Weekly computer aided formative assessment to underpin comprehension of the theoretical issues had been tried, as had a greater emphasis on reusability of code. However student achievement, measured not so much in terms of summative assessment results, but rather in terms of the ability of students to cope with the successor module, had not substantially improved. In addition, engagement in the module remained low (apart from those occasions when use was made of the Java graphics classes).

An aspect of the module appraisal process, which had appeared consistently over a number of years, was that many students reported a major difficulty with the (perceived) mathematical nature of the subject, a perception often reinforced by the set of examples used in the module. A Scottish Higher Computing qualification is not an entrance requirement for the RGU Computing programme and a substantial number of students taking the introductory programming module would also not have any Higher Mathematics qualification. In informal interviews conducted at the start of each teaching session, most of those questioned expressed, to some degree at least, a lack of confidence in their own mathematical abilities. Accompanying this was the widespread belief among students that programming is essentially a mathematical skill and that "being good at maths" was a de facto, if not formal, pre-requisite for success in the subject. Moreover, this tendency was more pronounced among students with lower mathematical qualifications, many of whom freely admitted that they had not enjoyed the subject when studied at school level, and for whom it carried a much greater affective dissonance. This problem was often combined with the barrier to immediate feedback caused by syntax errors which tended to block progress before the semantic aspects of the problem could be tested.

3. Module Redesign Principles

The main educational driver in the redesign of the module was to increase engagement in the subject by trying to circumvent some of the affective as well as cognitive reactions that accompanied a pedagogical emphasis on the standard algorithmic metaphor. It is certainly the case that, in its more mature forms, the discipline of software development requires a highly algorithmic approach and the application of a host of high-level skills. However, it was felt that there is a serious question about whether such an emphasis is the most productive approach to take in the initial stages of a course of study.

A basic principle of applied educational constructivism is that learning activities should make contact with the learner's prior experience. The position for many students was that although the module did indeed make contact with their prior learning, this was done in a way which did not take into account the negative affective component that accompanied such experience. To use a superficially trite example, when, at the start of the module, the

lecturer used a word such as "algorithm", many student would associate this with other words such as "algebra" or "logarithm" and hence with the negative experiences of school mathematics. However, it is important to stress that this is not just a superficial phonetic correlation. For the majority of the class who had not studied computing at secondary school, the only place in the school curriculum where they would have come across algorithmic ideas is their mathematical studies and this had a significant impact upon the presuppositions and preconceptions that they brought to the module.

In developing a replacement for the module, several curricular desiderata were formulated.

- The primary aim was to make the module engaging, with the inclusion of graphics and animation content if possible.

- The conceptual model of programming presented should make contact with students' prior experience in a positive way.

- The programming vehicle should be syntactically forgiving.

- It should allow observable skill progression.

- It should introduce basic object oriented vocabulary (e.g. class, instance, method, …) but this should be contextualised in terms of an appropriate software system.

- The teaching vehicle should allow a smooth transition to a more conventional text-based Java development environment in the second semester (e.g. via the BlueJ environment).

-

In addition to this, several desirable implications for the assessment process were also enunciated.

- The module should allow assessment that was consonant with the curricular framework proposed above.

- It should facilitate open-ended, divergent assessment.

- The module should reduce the assessment burden on students.

-

In terms of the pedagogical approach of the teaching team, this latter condition translated to a requirement that the mode of assessment should be portfolio-based (from weekly lab exercises) and that there should also be some form of group project.

In determining the teaching vehicle for the new module, account was taken of the fact that increases in the speed and sophistication of computer hardware mean that students now have access to new programming environments that do not use textual input at the primary method for the creation of programs. The application finally chosen was Alice 2.0 (Dann et al, 2006), a microworld-type environment developed at Carnegie Mellon University, USA. This software package allows users to create virtual 3D worlds populated by a wide variety of objects such as animals, buildings, fantasy characters, landscapes and people. The code needed to control these objects is assembled by using the mouse to select pre-defined program building blocks. When the code is executed, the result is an animation of the objects inside the microworld.

4. Claims about the Alice Software

The RGU study sought to assess two important claims which have been made (Cooper et al, 2003) concerning the usefulness of Alice in the teaching of elementary programming skills. The first of these is essentially concerned with the software's imposition of correct syntax (syntactic forgiveness). Alice only allows assembly of well-formed program blocks effectively preventing the student from making syntax errors and consequently allows more time to be spent on issues of object control. This argument has two threads: the elimination of syntax errors and the freeing-up of learning time which can then be devoted to the investigation of semantic issues such as object control. This latter point leads onto a second major claim which has to do with the integrative conceptual framework presented to the student. The claim is that, by allowing manipulation and control of objects in a virtual 3D world, Alice promotes a change in the supporting metaphor for the process itself. The application area, at least in the initial teaching module, is removed from the classical context of numerical or symbolic manipulation and transformed into that of 3D animation. The basic premise underlying this is that the manipulation of recognisable and intuitive objects affords better feedback and so allows the learner to develop greater programmatic control of objects.

4.1 Syntactic Forgiveness

The first part of the operational claim concerns the fact that the software only allows assembly of syntactically correct statements. From an RGU perspective, this was important because students on the original module spent a great deal of time struggling with Java syntax, either straightforwardly trying to identify such errors or, more subtly, dealing with the relationship between language syntax and the manipulation of program components, e.g. maintaining correct use of data types. Syntactic mistakes resulted in compilation errors that would often give the student little understandable feedback with which to try to remedy the situation and this was a major cause of frustration at early stages. The counterclaim in (Powers et al, 2007) on this point is that such initial stress on syntax is actually necessary and that it makes the student unprepared for a syntactically less-forgiving environment. Moreover these authors assert that the later negative affective reaction to this is just as damaging to the student, perhaps more so.

RGU results, taken over both the initial and the subsequent programming module, suggest that the use of the drag-and-drop code interface was, in general, a positive thing. The report by students of difficulties on the acquisition of the correct use of syntax did not appear to be greater than in previous years and, for at-risk students, the situation appeared to have improved. Nevertheless, a common statement from students was that it would have been advantageous if Alice had been designed with some facility to allow textual input of code (something that was present in earlier versions of the software).

The second element of the claim addresses the issue of motivation. Since the primary reason for writing programs is to perform complex computational tasks, the focus of instruction should move, as quickly as possible, from a syntactic focus to that of semantics, i.e. control, (although obviously the speed at which this is done is tempered by pedagogical

considerations). This not only allows greater exploration of possible areas of application but, since errors in logical structure often result in a mismatch between desired and exhibited program behaviour, it is more likely that systematic testing will give useable formative feedback.

The findings of the RGU study on this point were, on the whole, positive. Participation in the module (measured in terms of attendance and submission of portfolio exercises) increased as did student engagement (defined both in terms of participation and as a measure of how often students would go beyond the bare minimum needed to fulfil assessment requirements). Discussion with students revealed that this was due primarily to the fact that they were able to take intuitive control of software objects at an early stage of the module.

4.2 Change of Metaphor

A precise measure of usefulness of a change of metaphor is harder to gauge as it is difficult, without further research, to disentangle its effects from those resulting from additional time devoted to programming semantics. However, informal student interviews tended to suggest that this factor did indeed have an important impact on increased student achievement. This appeared to happen both through an increase in coherence of the cognitive framework (i.e. students more readily integrated the storytelling concept with their prior learning) and also through generating a more a positive affective disposition (i.e. they were less anxious about the subject when it was viewed through that particular perspective).

From an educational point of view, the use of animation presented a range of innovative options for the student to exhibit both proficiency in the relevant programming skills and creativity in their execution. The use of the storyboarding process to develop a hierarchy of program narratives, based on different degrees of storyline detail, facilitated the design and generation of code by a process of stepwise refinement, while the ease of running the animation process allowed the student to develop a testing methodology based on the systematic investigation of additions to existing code.

In terms of assessment structures, the basic objective was to devise an assessment framework in which the use of the important programming constructs would arise naturally in the course of the story design. The divergent nature of narrative – the fact that it is possible to imagine an endless series of scenarios which have the same set of core structural features but differ completely in narrative content – meant that students were more easily able to inject aspects of personal creativity and ingenuity into their assessment responses. As a consequence, rather than insist on the solution to a particular, fixed problem, it was possible simply to describe the specific set of programming constructs – user-defined classes, control structures, event-driven input, etc – that should appear in, say, the group project assessment submission and then allow the student to negotiate with lecturing staff and with fellow group members how to develop their own animated scenarios which would satisfy the learning objectives. This increased peer collaboration, while at the same time, it meant that plagiarism was easier to detect.

5. The Transition to Java

Although the use of Alice was generally found to be beneficial for the introductory programming module, there are still questions about how it integrates into a curriculum in which such a module functions as the precursor to subsequent study of a standard object oriented language like Java. Some concerns about this type of arrangement were articulated by Powers et al (2007) who drew attention to, among other things, ambiguities in the object model used by Alice, as well as the inability to instantiate objects programmatically. These factors, they contend, make true assimilation of object oriented concepts difficult to attain and, as a consequence, cause problems with the transition to more advanced software development in standard languages such as Java or C++.

While experience at RGU suggests that Powers may be overstating the case, there are aspects to the use of Alice that do require careful thought. An objects-first approach is strongly suggested (although not actually imposed) by the nature of the software. This is appealing from a pedagogical standpoint, but the object model used by Alice is more reminiscent of a prototype-based rather than a standard class-based object oriented language. This does still allow the introduction and discussion of object oriented terminology but it is true that careful thought needs to be given when explaining the distinction between classes and objects or discussing the nature of inheritance. Similar considerations apply to some of Powers' other comments, e.g. on the definition and use of variables. Also, due presumably to the informal way in which Alice's object gallery has been compiled, there are irritating inconsistencies in the use of capitalisation for class names which can lead to confusion.

At RGU, the successor module was taught using BlueJ (Kolling, 2003) and aimed at developing a greater appreciation of the object oriented aspects of Java. The BlueJ IDE was chosen because, like Alice, there is some degree of object visualisation, albeit in a UML class diagram framework. Moreover, like Alice, it is possible to create instances of objects and execute methods using mouse control. The two main issues concerning Alice and the transition to BlueJ are whether its use hindered the subsequent writing of code and whether it promoted transferability of skills from the context of the initial module.

As mentioned previously, the ability to write code, rather than assemble it, did not seem to deteriorate in the second module. While it is true that some students still had major problems with correct Java syntax, the numbers involved and the severity of the problems were not more serious than those which occurred in previous years.

The main point of discussion with regard to the second issue is whether the skills in creating interesting and effective animations in the virtual Alice environment translated to an equivalent competence in conventional software development in Java. Specifically the question is whether proficiency in developing and implementing algorithms, smuggled into Alice under the guise of maintaining narrative control of virtual 3D objects, translates into a corresponding expertise when the storytelling metaphor reverts to the conventional algorithmic one.

The overall conclusion from studies involving analysis of student and staff responses, from both introductory and follow-up software development modules over a two year period, is that the use of Alice, accompanied by an emphasis on group work and assessed by divergent exercises, definitely increases student enthusiasm for programming activities and enhances student participation in the module. However, on the specific issue of transferability of algorithmic competence from Alice to Java, the interpretation of the results is more problematic. It would be nice, given the perceived successes Alice enjoyed in the initial module, to report that there was substantial translation of skills from one context to the other. However, the situation with regard to this is more ambiguous and it appears that the context in which such algorithmic thinking is applied is important. Quantitative examination of results based on module pass rate and retention data shows an increase, although it would be argued that analysis based solely on such figures is superficial since it does not take into account the change in pedagogical approach which accompanied the introduction of Alice and therefore does suitably distinguish between transferability of skills and other factors. Certainly, the use of Alice does not appear to have had an adverse impact on the development of skills used in algorithmic thinking, this finding being corroborated by use of questionnaires which asked students to reflect on their own experience of learning Java and through direct discussion by staff with students. Stated in this way, however, this is not a very dramatic conclusion, although not completely fatal to the success of the project given that gaining competence in programming skills (equivalent to that taught on the module in previous years) was just one motivation for the use of Alice. Nevertheless it seems clear from staff feedback that a closer examination of the issues surrounding transition needs to be made. In particular, the details of the processes by which students acquire expertise in constructing algorithms needs closer scrutiny.

For example, some students were able to assemble algorithms in Alice which involved application of fairly complex proximity methods but still found it problematic to write standard sorting algorithms in Java. Discussion with students about the reasons for this invariably brought up the fact that, in Java, they felt there was a lack of transparency in how they should begin to formulate the algorithm and their understanding of the way in which the programming statements would be executed. Work done investigating sorting algorithms in the Alice environment, while limited, appeared to show that the mental processing involved in working through the sorting procedure seemed to be made easier because of the explicit visualisation of the process that this system afforded. This was true despite the extra effort required to code a simple example (e.g. one in which the positions of objects in the virtual world had to be rearranged by the objects actually moving from one location to another) and seemed to reflect the fact that students could see the state of the sorting procedure at any stage in the algorithm. This suggests that understanding and competence are strongly bound to the ability to visualise algorithms and that use of tools to promote this (e.g. Jeliot (Moreno et al, 2004) in Java), is something that should be encouraged.

Nevertheless, for many students, such knowledge was fragile, both in the general sense used by Perkins and Martin (1986) as being easily dislodged, and more specifically of not being robust in the transition from one context to another. Preliminary investigation into the fragility of understanding examples of algorithmic thinking in Alice, exemplified by sorting, suggests that further work should focus not just on how a particular concept is mapped from one environment to another but on the mapping of the relationships between that structure

and the network of auxiliary concepts which link it to the specific context in which it is applied. Work on using such an extended mapping to provide better transitional scaffolding is now underway.

Finally, some of the problems encountered in the transition from Alice to Java may have been due to issues of presentation and resourcing which were unresolved at the time of module delivery. These should also be addressed by the time of the next presentation of the module.

6. Conclusions

Reports from both teaching staff and students indicate that the introduction of Alice into the first year RGU undergraduate computing curriculum has been successful. Both participation and student engagement has increased in the introductory module, due partly to the Alice user interface's prevention of syntax errors and partly to innovative assessment practices afforded by the change in metaphor. At this stage, based on the RGU evidence, it is difficult to give a clear-cut answer to the specific question of whether Alice succeeds in promoting the transferability of algorithmic skills to a full object oriented language in the subsequent module, but initial results look promising and suggest that it is certainly made no worse. What appears clear, however, is the fact that students enjoyed their introduction to programming more and went into the Java module with less affective reactions against the subject.

7. References

Barnes, D.J. and Kolling, M., Objects First with Java: A Practical Introduction Using BlueJ, 3rd ed, (2006), Pearson Prentice Hall, London.

Cooper, S., Dann, W. and Pausch, R., (2000), Alice: A 3-D Tool for Introductory Programming Concepts. J. Comp. Sci.. 15(5), 108-117.

Cooper, S., Dann, W. and Pausch, R., (2003), Teaching Objects-first in Introductory Computer Science, Proc. 34th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '03, 191-195.

Dann, W., Cooper, S., and Pausch, R., Learning to Program with Alice, (2006), Pearson Prentice Hall, Upper Saddle River, New Jersey.

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., (2003), The BlueJ System and its Pedagogy, Computer Science Education, 13(4), 249-268.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T., (2001), A Multinational, Multi-institutional Study of Assessment of Programming Skills of First-Year CS Students. SIGCSE Bulletin 33(4).

Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M., (2004), Visualizing programs with Jeliot 3, Proceedings of the working conference on Advanced Visual Interfaces, May 25-28, 2004, Gallipoli, Italy, 373-376.

Morrison, M. and Newman, T. S., A Study of the Impact of Student Background and Preparedness on Outcomes in CS1, (2001), Proc. 32nd SIGCSE Technical Symposium on Computer Science Education, SIGCSE '01.

Perkins, D.N., and Martin, F., (1986), Fragile Knowledge and Neglected Strategies in Novice Programmers. In Empirical Studies of Programmers, edited by Soloway, E. and Lyengar, S. Norwood, N.J., Ablex.

Powers K., Ecott S., and Hirshfield, L. M., (2007), Through the looking glass: teaching CS0 with Alice, ACM SIGCSE Bulletin, Proc. 38th SIGCSE Technical Symposium on Computer Science Education, SIGCSE '07. 39(1),