The Effectiveness of Screencasts and Cognitive Tools as Scaffolding for Novice Object-Oriented Programmers

Mark J. W. Lee and Sunam Pradhan
University of Ballarat, Ballarat, Victoria, Australia

m.lee@ballarat.edu.au
s.pradhan@ballarat.edu.au

Barney Dalgarno
Charles Sturt University, Wagga Wagga, NSW, Australia

bdalgarno@csu.edu.au

Executive Summary

Modern information technology and computer science curricula employ a variety of graphical tools and development environments to facilitate student learning of introductory programming concepts and techniques. While the provision of interactive features and the use of visualization can enhance students' understanding and assist them in grasping fundamental ideas, the real difficulty for many students lies in making the transition from relying on the graphical features of these tools, to actually writing programming code statements in accordance with a set of plain English instructions.

This article opens with a systematic review of the literature on alternative approaches to teaching object-oriented programming (OOP) to novice programmers. It then describes the rationale behind an "objects first, class user first" approach to introducing OOP, arguing for the use of interactive GUI-based visualization tools such as BlueJ as cognitive tools to allow learners to represent and manipulate their mental models or schemas. Finally, it reports on a study involving a cohort of students undertaking an introductory OOP unit in Java. The study investigated the effectiveness of: (i) the graphical features of BlueJ as a cognitive tool while performing coding tasks as part of a test; and (ii) the use of screencasts (video screen captures) of BlueJ to provide scaffolding during learning, which involves the provision of temporary support structures to assist learners in attaining the next stage or level in their development. The screencasts were used in conjunction with a series of structured exercises by providing an intermediate stepping stone to ease the transition to the writing of program code.

The study found no significant effect of screencasts during the learning phase of the study, and no significant effect of BlueJ during testing. This result runs counter to theoretical predictions and consequently is important both for researchers focusing on the pedagogy associated with learning programming as well as those interested in the broader applications of animated instructional resources and cognitive tools.

In the article, the authors postulate a number of reasons for the lack of significant effects to sup-port their hypotheses. Firstly, it is possible that

Editor: Linda Knight

some, or perhaps many, participants who had access to BlueJ during the testing phase did not actually use it to assist them in answering the test questions. Secondly, since the screencasts and BlueJ were intended to ease students' transition to code, the data collection was conducted immediately following the participants' initial exposure to code statements. This gave rise to the possibility that they may not have been ready to attempt the questions framed at a high level of abstraction, which accounted for a majority of the test marks. The authors had hypothesized that the most benefit in providing the screencast-based scaffolding and the use of BlueJ as a cognitive tool was likely to be gained in assisting students with writing code for English instructions at this high level of abstraction; however, at this point in the semester they may not have been adequately prepared to undertake these types of questions, which required them to interpret the high-level task requirements and decompose them into individual object and class operations that would achieve the desired outcome (object state).

Further research will need to be carried out to determine whether these hypothesized reasons for the lack of an identifiable difference between conditions can be supported, whether other factors are responsible, or whether in fact neither BlueJ screencasts nor the use of BlueJ as a cognitive tool actually enhance learning. One possible approach to a follow-up study would involve using a smaller number of students, but carrying out intensive observation during the experiment in order to determine the degree to which, and ways in which, BlueJ is used. This may include an oral component incorporating think-aloud protocols (Ericson & Simon, 1993) and/or follow-up interviews to gain deeper insight into and understanding of the participants' thought processes as they attempt the various questions in the test, as well as to identify gaps in their understanding in relation to the test questions. In addition to informing on the value of screencasts and cognitive tools for the learning of programming, such a study would also reveal in greater depth the nature of the cognitive stages involved in learning to write object-oriented program code from English instructions.

Keywords: Object-oriented programming, programming pedagogy, objects first, Java, BlueJ, screencasting, visualization, cognitive tools, scaffolding.

Introduction

The present article describes a project aimed at investigating the use of structured, screencast-based exercises, in conjunction with the popular BlueJ (Barnes & Kölling, 2002; Kölling, Quig, Patterson, & Rosenberg, 2003; Kölling & Rosenberg, 2002) development environment, to teach introductory programming in Java using an "objects first" approach that begins by introducing object-oriented (OO) programming from the perspective of a class user.

The goal was ultimately to help students become competent in taking a description or set of instructions, written in plain English, and implementing it in programming code. Developing the exercises entailed capturing a series of screencasts, or screen recordings to be delivered over the Web. Each recording showed classes and objects being manipulated graphically in BlueJ, which uses a notation based on the Unified Modeling Language (UML). The exercises required the students to view the actions being performed in the screencasts and compose semantically equivalent lines of Java code. They were intended as a stepping-stone to assist students in moving from interacting with classes and objects graphically at runtime through a point-and-click interface, to writing lines of Java code to achieve similar effects.

Teaching Introductory Object-Oriented Programming

Objects First

Many teachers have found that an “objects first” or “objects early” approach is most effective when teaching OO programming to beginners (Barnes & Kölling, 2002; Cooper, Dunn, & Pausch, 2003; Kölling et al., 2003; Kölling & Rosenberg, 2001; Machanick, 2007; Proulx, Raab, & Rasala, 2002). This is in contrast to the older, more traditional approach of beginning with structured programming “in the small,” which is suited to teaching procedural languages, but is considered by many to be less appropriate to the OO paradigm. In an introductory Java programming unit, this approach is typified by the use of a “main” method within which students focus on writing code, commencing with the all-too-familiar “Hello World” example. The initial emphasis is on basic/primitive data types and control structures. Object-orientation is deferred to a later stage, usually no earlier than the fifth or sixth week of the semester, at which time students are forced to suddenly make the awkward leap to OO and begin to view the procedural statements they have learned to write in the “big picture” context of methods that implement the responsibilities or behaviors of classes and objects.

Notwithstanding the popularity of the objects first approach, there are certainly critics and skeptics of the approach (Astrachan, Bruce, Koffman, Kölling, & Reges, 2005; Bailie, Courtney, Murray, Schiaffino, & Tuohy, 2003; Bruce, 2005; Lister et al., 2006; Reges, 2006). There is also no doubt that objects first adds a level of complexity to the teaching and learning process. By diving right into object-orientation, students must come to grips with “many different concepts, ideas, and skills... almost concurrently. Each of these skills presents a different mental challenge” (Proulx et al., 2002, p. 66, para. 3). Myriad software tools and even full-blown languages have been developed to help meet the challenge of an objects first approach (see for example, Barnes & Kölling, 2002; Bergin, Stehlik, Roberts, & Pattis, 2006; Cooper et al., 2003; Kölling, 1999a, 1999b; Kölling et al., 2003; Kölling & Rosenberg 1996, 2001, 2002; Proulx et al., 2002). These tools generally have a strong visual/graphical component and incorporate interactivity to reduce the complexity that novice programmers must overcome, helping them “see” objects in a meaningful context.

Class User versus Class Developer

Students can be required to view OO programming from two perspectives or roles: that of the “class user” and that of the “class developer.” The class user makes use of the classes developed by the class developer. In doing so, he/she simply views classes and their instances (objects) as “black boxes” exposing a set of publicly available operations (methods) that implement the required functionality. Since these public methods are published in the form of an Application Programming Interface (API), the class user need not understand the internal workings of the classes/objects, including the programming code within each of their methods and the private members they encapsulate. The API serves as a form of contract or agreement between the class user and class developer specifying the name, input (parameters), and output (return type) of each method by means of a method signature.

Many teachers who adopt an “objects first” approach use graphical user interfaces (GUIs) as a means of introducing Java, thereby placing students in the role of class developers from the outset. Although this approach capitalizes on the high appeal of GUIs, in Java it introduces a level of complexity that makes it prohibitive for all but the strongest students to fully grasp the required concepts (Koffman & Wolz, 1999). Graphics libraries such as those described in Bruce, Danyluk, and Murtagh (2001) and Roberts and Picard (1998) may alleviate this problem to an extent.

Smith and Boyd (2001) suggest having students act as class users before requiring them to develop classes. A “class user first” approach (or what is referred to by authors like Meyer (1993) and Duke (1997) as an “inverse curriculum” approach) is logically sound since in all but the most trivial applications, class developers need to also make use of classes developed by others when creating their own classes. Students need not know the details of the classes being used – It is only necessary to know what types of messages can be sent to a class or object, and the way that class or object will respond. Moreover, this illustrates and stresses the importance of abstraction and re-use, two of the main goals of object-oriented software development.

BlueJ as a Cognitive Tool for Novice Object-Oriented Programmers

Like Smith and Boyd (2001), the authors advocate the use of BlueJ as an environment for teaching Java using an objects first, class user first approach. This is achieved by requiring students to examine the APIs of provided classes and to explore, investigate, and experiment with the classes and their instances through the graphical interface of BlueJ. BlueJ provides an integrated environment in which the student generally starts with a previously defined set of classes. The project structure is presented in notation similar to UML. The student can create objects and invoke methods on those objects to explore their behavior. Methods are invoked through context menus accessed by right-clicking on the relevant class (for static methods, including constructors) or object (for instance/non-static methods) (See Figure 1). Parameters and return values are entered and displayed by means of dialog windows. This strategy is effective because it allows students to familiarize themselves with OO concepts and develop sound intuitions about objects and OO programming before becoming concerned with Java language syntax. It is consistent with the general consensus amongst teachers that the ultimate goal should be to teach programming and not a programming language, focusing on the paradigm and methodology rather than the specifics of a particular language (Kölling, 2005; Luker, 1989, 1994; Zhu & Zhou, 2003).

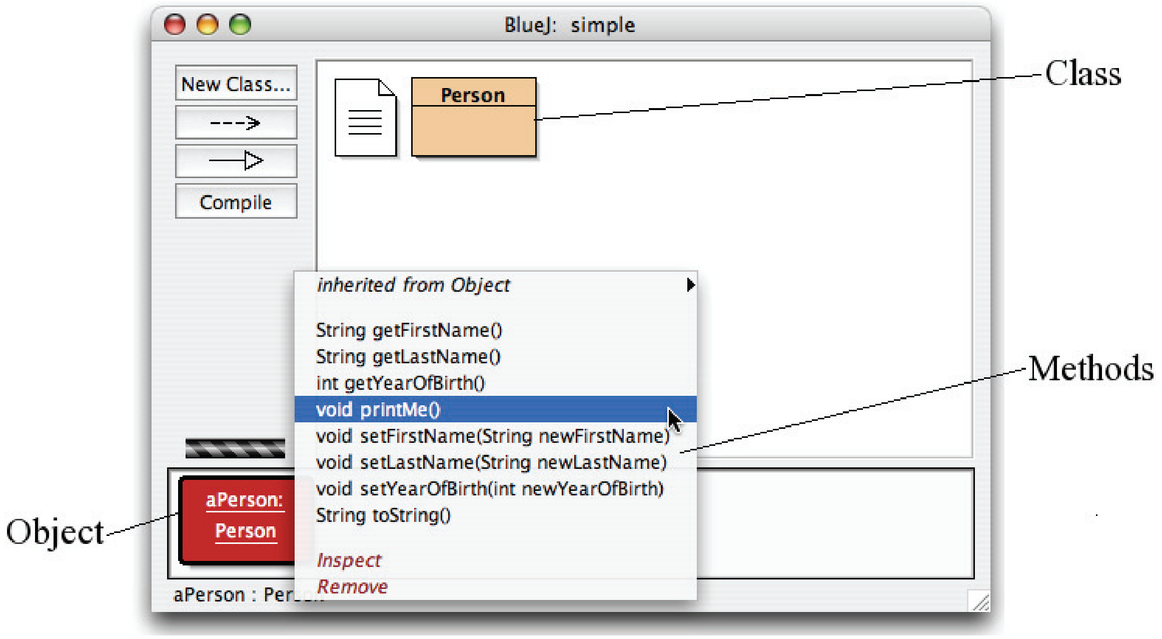

Figure 1. Methods invoked through context menus

BlueJ provides a graphical interface in which the user can interact directly with classes and objects and visually observe the resulting effects, thereby making it an excellent tool for demonstrating object-oriented concepts through both instructor-led demonstration, as well as learner-directed exploration, investigation, and experimentation. In addition, these features make BlueJ an ideal cognitive tool by allowing learners to represent and manipulate their cognitive structures or "mental models" of the object-oriented programming domain. Jonassen (1994) argues for the use of educational technologies as cognitive tools that provoke "mindful" engagement, rather than simply as conveyors, tutors, or repositories of information. This mindful engagement can occur when learners use a computer application to represent their knowledge. Furthermore, cognitive tools facilitate learning through knowledge construction rather than knowledge reproduction, encouraging learners to engage in generative information processing that involves activating the appropriate mental models, using them to interpret new information, then integrating the new information and reorganizing the mental models to form aggrandized models that can be used to explain, interpret, and/or infer new knowledge (Rumelhart & Norman, 1978).

A major hurdle, if not the most significant hurdle, for many students is making the transition from manipulating classes and objects within BlueJ's GUI, to writing Java code statements that achieve equivalent results. Students are typically expected to do this by writing lines of driver code placed in one or more methods within a "Test" class. For the novice programmer, writing his/her first lines of code can be an especially daunting task, and this transition should not be introduced abruptly, but rather should be a gradual process that is carefully planned and thought out to maximize the chances of success.

Using Screencast-Based Exercises to Assist Students in Making the Transition to Code

Overview of Screencasting

The term "screencast" was coined by Udell (2004). A screencast is a digital recording of the activity on a computer screen and may be thought of as a sequence of screenshots captured in rapid succession, to be played back at high speed in order to represent the motion that occurs on a user's screen over a period of time. In addition to video, screencasts may also contain audio tracks, which may consist of sound output from the computer whose screen is being recorded, or from an external source, such as voice narration or music.

While screen recording software products have been in existence for many years, early solutions produced large files and offered very limited editing capabilities. EDUCAUSE (2006) asserts that good screencasts depend not only on careful planning, but also on thoughtful and judicious editing to re-sequence lesson elements, eliminate awkward and unnecessary portions, and craft a focused, easy-to-follow presentation that makes efficient use of students' time. Recent products like Camtasia Studio (Techsmith Corporation, 2007) support more compact, cross-platform file formats suitable for web-based delivery such as Macromedia Flash, and have more sophisticated editing features allowing changes in sequence, mouse movement, and audio.

Because screencasts can be produced in a variety of formats such as Macromedia Flash, Apple Quicktime, Windows Media Player, and AVI, they are suited to delivery on a variety of platforms. Screencasts can be delivered via streaming or downloaded in their entirety for later viewing. Downloaded content can be transferred to a variety of portable devices such as personal media players (PMPs) and video-capable iPods (Apple Computer, 2006), creating opportunities for mobile learning, although the small screens of many portable devices impose restrictions on the types of material that lend themselves to this kind of application.

Perhaps the most obvious and common uses of screencasting involve demonstrations or tutorials of software packages. In the programming arena, a natural application of this technology would be in the creation of web-based lectures demonstrating and explaining, step-by-step, the process of writing, testing, and debugging code. Bennedsen and Caspersen (2005) highlight the value and cost-effectiveness of using “process recordings” of an expert programmer (for example, the teacher) solving a concrete programming problem, thinking aloud as he/she moves along. As compared with traditional classroom teaching methods such as programming on a blackboard, presenting finished code on transparencies, or live programming via a data projector, process recordings have the advantage of allowing students to revisit, in part or whole, the development process for clarification, reinforcement, and revision purposes. Bennedsen and Caspersen (2005) suggest that these recordings should not be “perfect” – That is, they should not be artificially planned, scripted, or crafted, but rather should capture the actual programming process as naturally and authentically as possible. This includes the making of errors and their resolution, the integrated use of the development environment (IDE), referring to online documentation, and so on. An alternate view is that the process recordings should be scripted and crafted (as in Gries & Gries, 2001, for example), but in such cases, deliberate errors could be introduced to demonstrate testing and debugging techniques and processes.

While such applications hold tremendous value, Poindexter (2003) and Pendergast (2006) emphasize the importance of using active learning techniques and strategies when teaching programming, consistent with a Constructivist philosophy, which emphasizes that students learn best when they construct a personal understanding based on a continuous process of experience and critical reflection upon experience (Jonassen, 1991; Piaget, 1973). However, the problems of pure discovery learning without adequate instructional support have been well documented (see, for example Mayer, 2004) and accordingly the authors believe that screencasting technology has considerable potential as a supporting instructional resource, allowing learners to obtain the benefits of active learning tasks without the confusion that can occur when such tasks are employed in the absence of adequate support. The old mantra “the only way to learn programming is through practice” comes to mind here. In the authors’ experience, as well as those of numerous colleagues, weaker students tend to spend disproportionate amounts of their study time on activities of secondary importance, favoring behaviorally passive tasks such as reading the textbook/notes and watching or listening to recorded lectures. They go through great lengths to avoid the substantial leap from reading about programming to actually writing their own programs, except where explicitly dictated by the assessment scheme, preferring to attempt theory exercises (e.g., multiple-choice questions) or play with the GUI tools provided.

For this reason, the screencast-based exercises should gently but surely encourage students to begin to write their own code, with the recordings providing the necessary *scaffolds* for learner-centered, learner-driven activities to be carried out. Scaffolding is a concept that has its roots in Vygotskyan theory (Vygotsky, 1978), in which a learner’s zone of proximal development (ZPD) is the gap between a learner’s current or *actual* development level – at which he/she is able to solve problems without assistance – and his/her emerging or *potential* level of development. Scaffolding involves systematically providing learners with supportive aids in the form of tools, strategies, and guides within the parameters of their ZPDs, to assist them in progressing to their next, potential level of development (Brush & Saye, 2001; Dabbagh, 2003). This is typically accomplished by initially limiting the complexity of the assigned task, then gradually introducing complexity by removing or “fading” the support provided as learners make progress and acquire the knowledge, skills, and confidence required to handle the task independently and in its authentic context (Young, 1993).

Design of Scaffolded Exercises Using BlueJ and Screencasting

As mentioned earlier, the ultimate aim is to have students become competent in taking a description or set of plain English instructions, and implementing it in Java code. This involves translating between two different representations of a program, and can be denoted *English => Java*. The use of BlueJ as described in the previous section is effective in helping students understand object-oriented concepts and achieves this by introducing a third, UML-like graphical view or representation of a program, which will simply be denoted hereafter as *BlueJ*. The use of the BlueJ GUI alone as a cognitive tool for teaching and learning activities only requires the student to translate between plain English and BlueJ's graphical representation (*English* => *BlueJ*) but does not provide the necessary scaffolds to gently and gradually introduce students to code. The classic work of Spohrer and Soloway (1986) indicates that students have difficulties not so much in misconceptions about language constructs (construct-based problems) as in putting the "pieces" of a program together (plan composition problems). A staged approach incorporating fading levels of conceptual and procedural scaffolding (Hill & Hannafin, 2001) is needed to assist with the transition to code.

Table 1 summarizes the distinct features of each of the three representations of a program. Students must be conversant in all three representations in order to be able to effectively translate between them.

Table 1: Differences between the Plain English, BlueJ, and Java code representations of a program

ENGLISH	BLUEJ	JAVA
Plain English description or set of instructions	UML-like representation of classes and objects	Java code statements
Text-based	Visual / graphical	Text-based
Compile-time view	Run-time view, allowing for interactive object creation and method calls	Compile-time view
Syntax unimportant – Rules not enforced by software	Representation is symbolic. Adherence to rules is responsibility of BlueJ software – Violations are prevented by user interface constraints and internal program logic of BlueJ	Adherence to syntax rules is responsibility of programmer, but syntax errors are detected by Java compiler through the process of parsing

The researchers hypothesized that by having students complete exercises in which they are given screencasts showing classes and objects being manipulated in BlueJ's GUI and are required to produce snippets of equivalent Java code (*BlueJ* => *Java*), they would be better placed to tackle writing code from plain English instructions (*English* => *Java*). For example, the action being performed in BlueJ in Figure 1 equates to the following line of Java code:

```
aPerson.printMe();
```

It was expected that students would find writing code from plain English instructions easier having carried out the screencast-based exercises, because they would then make use of an intermediate step (*English* => *BlueJ* => *Java*). During the intermediate step, the GUI features of BlueJ are used as a cognitive tool to aid students in conceptualizing the classes, objects, and actions through interaction and visualization, before they make an attempt to translate the concepts into code.

The authors also believed that with sufficient practice, students would be able to "remove the training wheels" and translate a plain English description directly into code (*English* => *Java*). The intermediate step should become second nature as students become decreasingly reliant on the cognitive tool, and develop unconscious competence in constructing internal (mental) representations of the classes and objects, from which they can derive lines of Java code. Whether or not this is actually the case will need to be investigated through further research that is beyond the scope of the present study.

In addition to the above, it should be noted that the plain English instructions may be framed at varying levels of abstraction. For example, consider following "high level" plain English instruction:

> John Doe purchases Peter Smith's house for its current value, through a direct debit between their bank accounts.

This can be expressed as four "lower level" instructions:

1. Get the current value of Peter Smith's house;
2. Withdraw the amount from John Doe's bank account;
3. Deposit the amount into Peter Smith's bank account;
4. Change the ownership of Peter Smith's house to reflect John Doe as the new owner.

The authors believed that the most benefit in providing the screencast-based scaffolding and the use of BlueJ as a cognitive tool was likely to be gained in assisting students with writing code for English instructions at a "high level" of abstraction. This is because the use of scaffolding to simplify object-oriented concepts (conceptual scaffolding) and the planning of the steps/order in which the various program actions – class instantiation, method invocation, variable assignments, etc. – are to be executed (procedural scaffolding) frees up some of the learner's limited cognitive resources, making them available for allocation to higher-order tasks such as problem-solving and resolving task requirements into lower-level instructions that are directly implementable as code statements.

Method

The researchers set out to test the aforementioned hypotheses through a simple experiment, designed to determine whether the following had an effect on students' ability to translate a set of plain English instructions into lines of Java code:

1. During a learning phase, completing a series of exercises requiring them to watch screencasts of actions being performed in BlueJ and coming up with semantically equivalent lines of code; and
2. During a testing phase, making use of BlueJ as a cognitive tool to plan and visualize their responses graphically before attempting to write code.

To this end, participants were recruited and randomly allocated into four groups, depicted in Table 2.

Table 2: Groups used in experiment – 2 x 2 factorial design

		USE OF BLUEJ IN TEST TASKS (FACTOR B)	
		No (0)	Yes (1)
USE OF SCREENCAST-BASED EXERCISES IN LEARNING TASKS (FACTOR S)	No (0)	Group S0B0	Group S0B1
	Yes (1)	Group S1B0	Group S1B1

All four groups were asked to complete a series of learning tasks designed to teach them to write simple lines of Java code to instantiate unseen classes and invoke instance (i.e., non-static) methods given the relevant API documentation. They were then tested on their mastery of this learning outcome and the results of the groups compared. The experiment was conducted across three tutorial class groups (sections), leading to an overall 3 x 2 x 2 design.

Participants and Context

The participants in this study were information technology students studying at one of two campuses located in the Central Business District of Sydney, Australia. They were enrolled in a first-year introductory programming unit, in which Java was used as the teaching language. A vast majority of them were international students hailing from upper-middle-class families in the Indian subcontinent, with medium levels of household income. Most of the participants were enrolled in a Graduate Diploma of Information Technology or Master of Information Technology program, having completed an undergraduate degree in a different discipline in their home country. International students enrolled in these degree programs generally apply to the Australian Government Department of Immigration and Citizenship for permission to work, and upon being granted this permission are allowed to do so for maximum of 20 hours per week during the academic semester (Department of Immigration and Citizenship, n. d.).

The study took place in Weeks 3, 4, and 5 of a 12-week semester. At this point in the semester, the students had undergone an orientation to BlueJ, which included creating a project as well as adding a provided class to the project and compiling it. They had also been introduced to the theoretical concepts of classes, objects, attributes, and methods, had exposure to the purpose and format of Java API documentation, and learned how to instantiate classes and invoke instance methods using BlueJ's GUI interface. Although in Week 4 they attended a lecture introducing basic Java code for instantiation and instance method invocation, they had not had hands-on practice in writing code themselves.

In all, 38 students participated in the study, 36 males and 2 females. The participants in each tutorial class were randomly allocated to the 4 groups: 8 were allocated to the S0B0 group, 10 were allocated to the S0B1 group, 9 were allocated to the S1B0 group, and 11 were allocated to the S1B1 group. (S0B0 and S1B1 contained one female participant each.) The slightly unequal group sizes were due to the lack of attendance of a small number of students.

Learning Tasks and Procedure

During the Week 3 lab class, all students were asked to complete a set of exercises, which required them to complete a set of tasks in BlueJ, based on a plain English description (*English => BlueJ*). These tasks involved using BlueJ's GUI interface to instantiate unseen, provided classes and invoke methods on the instances interactively. The students had access to the API documentation for the provided classes while completing the exercises. Solutions to the exercises were provided to allow them to check their work. The solutions were presented as screencasts demonstrating the actions that needed to be performed in BlueJ to accomplish the required tasks. Most of the screencasts lasted no more than one minute each.

The main data collection was carried out during the Week 5 lab class. Students were not notified of the exercise in advance, in order to minimize the possibility of certain participants undertaking additional study in their own time that would give them an advantage over other participants, thereby confounding the results. This class began with a 30-minute "chalk and talk" session reviewing how to write Java code to perform instantiation and instance method invocation, as well as how to refer to API documentation to facilitate such tasks. This session incorporated a demonstration in which the instructor used a sample BlueJ project to model the process of implementing a plain English instruction as an action in BlueJ's GUI (*English => BlueJ*), then writing semantically equivalent line(s) of Java code (*BlueJ => Java*). Questioning and active listening techniques were employed by the instructor to promote student engagement and involvement. Students were also asked to review the Week 4 lecture notes, which summarized the pertinent points relating to writing Java code statements for instantiation and instance method invocation. They then proceeded to attempt to write Java code satisfying the requirements of the aforementioned Week 3 exercises. For each exercise, members of the S1B0 and S1B1 groups were asked to read the plain English description and view the screencast (i.e., the solution from Week 3 – *English => BlueJ* provided as screencast). They were then required to write equivalent lines of Java code (*BlueJ => Java*). The aim was to emphasize the relationships between point-and-click actions carried out through the GUI interface and the lines of corresponding Java code. The S0B0 and S0B1 groups were not provided with access to the screencasts, and so were required to write the Java statements without the intermediate step (*English => Java*). Once again, in the Week 5 exercises the API documentation for the provided classes was supplied to the students in all four groups, and the (Java code) solutions were made available to allow them to self-check their answers. The seating arrangements prevented the members of the control groups from accidentally or intentionally observing the screencasts on the other students' screens.

Copies of the data collection instruments used in the learning phase of the study, including the exercises completed by participants and the accompanying screencasts, are available on a web site maintained by the authors (Lee, 2008). (A web browser with the Adobe Flash plug-in is required to view the screencasts.)

Test Tasks and Procedure

On completion of the learning phase of the study, all participants undertook a paper-based test, copies of which can be downloaded at Lee (2008). This test consisted of eight items, each comprising plain English instructions requiring the participants to write lines of Java code to perform a number of tasks involving instantiation and instance method invocation. The test items were based on a set of provided, unseen classes, for which all participants were supplied with hard copies of the API documentation.

Groups S0B1 and S1B1 were asked to first use BlueJ on a computer to visually demonstrate how the English instructions would be implemented, before attempting to derive equivalent lines of Java code on paper (*English => BlueJ => Java*). (Note, they were not allowed to enter code into the computer to compile or test it in any way. Moreover, the procedures carried out in BlueJ were not taken into account when scoring responses.) Groups S0B0 and S1B0 were not allowed access to a computer and were required to write the code directly based on the plain English instructions (*English => Java*).

The paper-based responses were scored, with one mark being awarded for each correctly implemented code feature. Half marks were not awarded for partially correct code. The marking scheme also distinguished between high abstraction level ("H"-type) and low abstraction level ("L"-type) tasks. The highest possible score was 46, including 13 marks for "L"-type tasks and 33 marks for "H"-type tasks.

To minimize risk to students, neither this test nor the Week 3 and Week 5 learning exercises contributed to their final grade. In addition, all students in the cohort were provided with access to the learning exercises and their solutions shortly following the data collection required for this study.

Results

SPSS (Statistical Package for the Social Sciences) was used to perform an analysis of the data collected. The initial analysis undertaken consisted of a univariate analysis of variance (ANOVA) using a 3 x 2 x 2 design with access to BlueJ during the test, access to Screencasts during learning, and Tutorial Group as dependent variables. The test scores ("L"-type, "H"-type, and Total) were the dependent variables. This analysis, shown in Table 3, indicates that there was no main effect of BlueJ availability on either "H"-type Score, "L"-type Score, or Total Score. Similarly, there was no main effect of Screencast availability on either "H"-type Score, "L"-type Score, or Total Score. There was also no significant interaction effect of access to BlueJ and access to Screencasts, nor was there a significant three-way interaction between BlueJ, Screencasts, and Tutorial Group.

The means and standard deviations of scores obtained by students with and without access to Screencasts are shown in Table 4. The means and standard deviations for students with and without access to BlueJ are shown in Table 5.

Table 3: Analysis of Variance for BlueJ x Screencast x Tutorial Group

	df	"L"-TYPE SCORE		"H"-TYPE SCORE		TOTAL SCORE	
		F	p	F	p	F	p
Corrected Model	11	0.900	0.553	3.964	0.002	3.031	0.010
Intercept	1	281.005	0.000	109.755	0.000	202.360	0.000
Screencasts	1	0.417	0.524	1.181	0.287	1.087	0.307
BlueJ	1	0.048	0.828	0.963	0.336	0.636	0.432
Tutorial Group	2	2.054	0.149	13.903	**<.0005**	10.314	**0.001**
Screencasts x BlueJ	1	0.142	0.709	0.830	0.371	0.261	0.614
Screencasts x Tutorial Group	2	1.004	0.380	3.289	**0.053**	2.726	**0.084**
BlueJ x Tutorial Group	2	1.380	0.269	0.250	0.781	0.650	0.530
Screencasts x BlueJ x Tutorial Group	2	0.069	0.933	0.972	0.392	0.572	0.571
Error	26						
Total	38						

Table 4: Comparison of test scores for students provided with screencasts and those not provided with screencasts during the learning phase

	"L"-TYPE SCORE (out of 13)	"H"-TYPE SCORE (out of 33)	TOTAL SCORE (out of 46)
NO SCREENCASTS (S0) *(n=18)*	Mean: 8.111 Std. Dev.: 2.698	Mean: 9.333 Std. Dev.: 8.338	Mean: 17.444 Std. Dev.: 10.291
SCREENCASTS (S1) *(n=20)*	Mean: 7.550 Std. Dev.: 2.800	Mean: 7.450 Std. Dev.: 5.491	Mean: 15.000 Std. Dev.: 7.434
SCREENCASTS MAIN EFFECT	p=0.524 (not significant)	p=0.287 (not significant)	p=0.307 (not significant)

Table 5: Comparison of test scores for students with access to BlueJ and those without access to BlueJ during the testing phase

	"L"-TYPE SCORE (out of 13)	"H"-TYPE SCORE (out of 33)	TOTAL SCORE (out of 46)
NO BLUEJ (B0) *(n=17)*	Mean: 7.882 Std. Dev.: 2.619	Mean: 8.941 Std. Dev.: 7.084	Mean: 16.824 Std. Dev.: 8.791
BLUEJ (B1) *(n=21)*	Mean: 7.762 Std. Dev.: 2.730	Mean: 7.857 Std. Dev.: 6.981	Mean: 15.619 Std. Dev.: 9.102
BLUEJ MAIN EFFECT	p=0.828 (not significant)	p=0.336 (not significant)	p=0.432 (not significant)

As can be seen in Table 3, there was a significant main effect of Tutorial Group on "H"-type Score and Total Score. Carrying out a Post Hoc analysis using Tukey's Honestly Significant Difference (HSD) test showed that students in Tutorial Group 1 performed significantly better than students in the other two groups on "H"-type questions (p<0.0005), and on Total Score (p<0.0005), while there was no significant difference between Tutorial Groups 2 and 3 on "H"-type questions (p=0.691), or on Total Score (p=0.928). The means and standard deviations of scores for students in each Tutorial Group are shown in Table 6.

Table 6: Comparison of test scores for students in each Tutorial Group

	"L"-TYPE SCORE (out of 13)	"H"-TYPE SCORE (out of 33)	TOTAL SCORE (out of 46)
TUTORIAL GROUP 1 (T1) *(n=9)*	Mean: 9.444 Std. Dev.: 1.740	Mean: 16.667 Std. Dev.: 8.832	Mean: 26.111 Std. Dev.: 10.216
TUTORIAL GROUP 2 (T2) *(n=12)*	Mean: 7.667 Std. Dev.: 2.995	Mean: 4.833 Std. Dev.: 2.480	Mean: 12.500 Std. Dev.: 4.890
TUTORIAL GROUP 3 (T3) *(n=17)*	Mean: 7.059 Std. Dev.: 7.816	Mean: 6.412 Std. Dev.: 4.048	Mean: 13.471 Std. Dev.: 6.296
TUTORIAL GROUP MAIN EFFECT	p=0.524 (not significant)	**p<0.0005 (highly significant)**	**p=0.001 (highly significant)**

There was also an interaction effect between Tutorial Group and access to Screencasts on "H"-type Score and Total Score, but not "L"-type Score, that approached significance (p=0.053 for "H"-type Score and p=0.084 for Total Score). Table 7 and Table 8 show the means and standard deviations of "H"-type and Total Scores respectively, for students in the Screencasts and No Screencasts groups for each Tutorial Group.

Table 7: Comparison of "H"-type Scores for students provided with Screencasts and those not provided with Screencasts, for each Tutorial Group

	NO SCREENCASTS (S0) *(n=18)*	SCREENCASTS (S1) *(n=20)*	SCREENCASTS MAIN EFFECT WITHIN EACH TUTORIAL GROUP
TUTORIAL GROUP 1 (T1) *(n=9)*	T1S0 *(n=5)* Mean: 20.600 Std. Dev.: 6.348	T1S1 *(n=4)* Mean: 11.750 Std. Dev.: 9.811	p=0.215 (not significant)
TUTORIAL GROUP 2 (T2) *(n=12)*	T2S0 *(n=5)* Mean: 4.400 Std. Dev.: 1.517	T2S1 *(n=7)* Mean: 5.143 Std. Dev.: 3.078	p=0.324 (not significant)
TUTORIAL GROUP 3 (T3) *(n=17)*	T3S0 *(n=8)* Mean: 5.375 Std. Dev.: 4.307	T3S1 *(n=9)* Mean: 7.333 Std. Dev.: 3.808	p=0.436 (not significant)
TUTORIAL GROUP MAIN EFFECT WITHIN SCREENCAST / NO SCREENCAST GROUP	**p<0.0005 (highly significant)**	p=0.194 (not significant)	*Screencasts x Tutorial Group interaction effect:* **p=0.053 (approaching significance)**

Table 8: Comparison of Total Scores for students provided with Screencasts and those not provided with Screencasts, for each Tutorial Group

	NO SCREENCASTS (S0) *(n=18)*	SCREENCASTS (S1) *(n=20)*	SCREENCASTS MAIN EFFECT WITHIN EACH TUTORIAL GROUP
TUTORIAL GROUP 1 (T1) *(n=9)*	T1S0 *(n=5)* Mean: 31.000 Std. Dev.: 6.595	T1S1 *(n=4)* Mean: 20.000 Std. Dev.: 11.431	p=0.182 (not significant)
TUTORIAL GROUP 2 (T2) *(n=12)*	T2S0 *(n=5)* Mean: 12.800 Std. Dev.: 3.347	T2S1 *(n=7)* Mean: 12.286 Std. Dev.: 6.020	p=0.896 (not significant)
TUTORIAL GROUP 3 (T3) *(n=17)*	T3S0 *(n=8)* Mean: 11.875 Std. Dev.: 6.621	T3S1 *(n=9)* Mean: 15.000 Std. Dev.: 7.434	p=0.416 (not significant)
TUTORIAL GROUP MAIN EFFECT WITHIN SCREENCAST / NO SCREENCAST GROUP	**p=0.001 (highly significant)**	p=0.327 (not significant)	*Screencasts x Tutorial Group interaction effect:* **p=0.084 (approaching significance)**

Discussion

Further research will need to be carried out to determine the precise reasons why the results of the study did not support the original hypotheses. The authors envisage that this will entail conducting a follow-up study in which a smaller number of students will undergo the learning and testing phases. This time, they will be subject to intensive observation while they complete the test to determine the degree to which, and ways in which, BlueJ is used. This may include an oral component incorporating think-aloud protocols to gain deeper insight into and understanding of the participants' thought processes as they attempt the various questions in the test. Additionally, the participants will be interviewed to ascertain where there were gaps in their understanding, and/or where they went wrong in relation to the test questions.

At this stage, the authors are only able to speculate as to why the use of BlueJ in the test and the use of screencasts during learning did not appear to have a significant effect on students' performance. One possibility is that there was no way of ensuring that students who were given access to BlueJ during the testing phase of the study (i.e., groups S0B1 and S1B1) actually made use of it to help them answer the questions in the test. This could perhaps have been addressed by requiring the students to submit their BlueJ models as part of the test. Such a strategy to encourage compliance with the intentions of the study is likely to have been particularly effective if the test had been part of the students' assessment for the subject. The provision of differentiated learning opportunities for a formal assessment task would, however, have been ethically problematic. The planned follow up study, which will involve intensive observations of students during the learning task and during the test, will help to determine whether the lack of advantage for those provided with BlueJ during the test was in fact due to many students choosing not to use it.

Additionally, if a student chose not to use BlueJ in the test, any value gained from watching the screencasts as part of the exercises during the learning phase is likely to have been negated. The relatively passive nature of the screencasts as a learning resource may have meant that on their own they were of relatively limited value. That is, it is possible that their primary value would have been as a scaffold to help students learn to use BlueJ as a cognitive tool during coding. In fact, for students who chose not to use BlueJ, or who were not given access to BlueJ, the use of screencasts as scaffolding during learning may have put them at a disadvantage. Such students carried out the test by going directly from English to Java, but those of them exposed to screencasts during learning had experience only in going from BlueJ to Java.

Moreover, when retrospectively returning to their hypothesis that the independent variables were likely to have the greatest effect on students' ability to complete the high abstraction level ("H"-type) questions, the authors considered the possibility that the students may not yet have been adequately prepared to tackle these types of questions, which accounted for a majority of the marks on the test. If this was true, the study may have yielded markedly different results if it had been conducted later in the semester, rather than immediately following the students' initial introduction to programming code.

Although the differences between the Screencast and No Screencast groups were not significant when considering students across all Tutorial Groups, or when considering students within either Tutorial Group 1 or 2 individually, the magnitude of the difference between the Screencast and No Screencast groups within Tutorial Group 1 deserves some attention. The students in this group performed substantially better than the students in the other Tutorial Groups, and also had a much larger variance. It is possible that students with a better understanding of the material find the screencasts unnecessary – and in fact a hindrance – because of the time spent viewing them and also because their inclusion denies them of practice in going directly from English to Java. Caution must be exercised in drawing such a conclusion because the sample sizes in individual groups were relatively small. Nevertheless, this may be something that could be explored in further research.

It should also be noted at this point that Tutorial Group 1 was based on a separate campus to the other two Tutorial Groups. It is possible that there is great disparity between the profiles of the typical student at each campus, since the differences in the recruitment systems and processes, in campus cultures, as well as other factors may have contributed to the attraction and development of different types and calibers of students. It is also possible that the large variance in ability levels across the student population meant that a larger sample size than 38 would have been required to be able to see a significant difference between groups, especially if the effect size was relatively small.

A possible improvement in the learning phase of the study could have been to emphasize to students the relationships between the BlueJ actions performed in the screencasts and the corresponding, individual Java code statements. The solutions to the Week 5 exercises were provided as code blocks or snippets containing several lines of Java code, which were simply presented as text files. A useful enhancement could have been to show each line of code being typed within the screencast, timed in such a way as to be synchronized with the corresponding action being performed in the BlueJ GUI. In implementing this enhancement, however, cognitive load and "split-attention" considerations must be taken into account, so as to avoid overwhelming the student and therefore adversely affecting his/her performance and learning (Chandler & Sweller, 1991, 1992; Mayer, 2003; Sweller, 1988; Sweller, Chandler, Tierney, & Cooper, 1990; Sweller, van Merrienboer, & Paas 1998).

An explanatory voice narration track could also have been added to the screencasts as a further aid to students' understanding of the object-oriented concepts involved and their realization in both graphical (*BlueJ*) and program code (*Java*) forms. Appropriate use of voice narration would be consistent with Paivio's (1986; Clark & Paivio, 1991) dual coding theory and Mayer's (2001) cognitive theory of multimedia learning, which both advocate the presentation of information in both visual and verbal form. Like Mayer, the work of Sweller et al. (1998) found that multimedia instructions are more effective when verbal information is presented auditorily rather than visually (known as the *modality effect*). Voice narration could also have been used as part of a separate set of instructional screencasts, designed to supplement or replace the "chalk and talk" session that preceded the Week 5 learning exercise. There is a vast body of literature available to help guide instructional design decisions when combining spoken and/or visual text with animation in the creation of such instructional multimedia applications (for example, see Kalyuga, 2000; Mayer, 1999; Mayer & Moreno, 1998; Sweller et al., 1998).

To summarize then, there are a number of possible reasons that could explain why the study was unable to find a significant difference in the learning of students provided with screencasts during learning and those who were not and between students provided with access to BlueJ during testing and those who were not. These include the sample size, the large variance in student abilities, and most importantly a possible lack of compliance with the intention of the study by students in the BlueJ group in the way that they completed the test. The planned follow-up study, which will include intensive observations, will specifically explore this latter issue.

Conclusion and Future Work

The study found no significant effect of the provision of screencasts during learning, nor of the use of BlueJ during testing, on ability to write Java code given a problem specification in English. However, the authors are not ready to dismiss the value of screencasts as scaffolding during learning, nor of BlueJ as a cognitive tool. Rather, more work is required to more intensely explore the way students go about their learning and coding tasks and the ways in which attending to screencasts and the use of BlueJ affect their cognitive processing.

Additionally, the researchers plan to study whether their findings generalize to other programming teaching scenarios, including those involving the use of other languages and/or tools. For example, the research discussed in this article has focused on scaffolding exercises to encourage students to use BlueJ's UML-like representation of a Java program as a cognitive stepping-stone. Although the BlueJ GUI provides for interactive class instantiation and method invocation by the user, it does not provide visualization of the messages sent between objects and intermediate changes in object state, that result from the user-initiated actions. What is displayed at any particular point in time is a snapshot – or *static* picture – of the current state of the objects and classes within the project in a UML-like class diagram format. Authors like Ragonis and Ben-Ari (2005) assert that visualizations of the *dynamic* aspects such as those supported by the code animation tool Jeliot (Ben-Ari, Myller, Sutinen, & Tarhio, 2002) are critical to students' understanding of program flow and execution. Jeliot is capable of displaying animations of method calls, variables, and operation at each step during the execution process of a Java program. It would be interesting to see whether similarly structured exercises based around screencasts of such dynamic representations are of value to students' learning of object-oriented programming.

Acknowledgements

The authors would like to acknowledge the support and assistance provided by Professor Sid Morris of the University of Ballarat, and Dr. Robyn Pierce of the University of Melbourne (previously with the University of Ballarat). They would also like to thank Dr. Michael Kölling of the University of Kent, the creator of BlueJ, for his valuable advice and comments in relation to the project during its early planning and inception stages.

References

Apple Computer. (2006). Apple – iPod Family. Retrieved March 27, 2006, from http://www.apple.com/ipod/

Astrachan, O., Bruce, K., Koffman, E., Kölling, M., & Reges, S. (2005). Resolved: Objects early has failed. Paper presented at the *36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE-05)*, St. Louis, MO, February 23-27.

Bailie, F., Courtney, M., Murray, K., Schiaffino, R., & Tuohy, S. (2003). Objects first – does it work? *Journal of Computing Sciences in Colleges, 19* (2), 303-305.

Barnes, D., & Kölling, M. (2002). *Objects first with Java – A practical introduction using BlueJ*. Englewood Cliffs, NJ: Prentice-Hall.

Ben-Ari, M., Myller, N., Sutinen, E., & Tarhio, J. (2002). Perspectives on program animation with Jeliot. In S. Diehl (Ed.), *Lecture notes in computer science, Volume 2269: Software visualization: International seminar, Dagstuhl Castle, Germany, May 20-25, 2001; revised papers* (pp. 618-621). Berlin: Springer-Verlag.

Bennedsen J., & Caspersen, M. E. (2005). Revealing the programming process. In J. Dougherty (Ed.), *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '05) (pp. 186-190). New York, NY: ACM.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (2006). *Karel J. Robot: A gentle introduction to the art of object-oriented programming in Java.* Retrieved October 10, 2006, from http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

Bruce, K. B. (2005). Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *SIGCSE Bulletin, 37* (2), 111-117.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2001). A library to support a graphics-based object-first approach to CS 1. In I. Russell (Ed.), *Proceedings of 32nd SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '01) (pp. 6-10). New York, NY: ACM.

Brush, T., & Saye, J. (2001). The use of embedded scaffolds with hypermedia-supported student-centered learning. *Journal of Educational Multimedia and Hypermedia, 10* (4), 333-356.

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

Chandler, P., & Sweller, J. (1992). The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology, 62*(22), 233-246.

Clark, J. M., & Paivio, A. (1991). Dual coding theory and education. *Educational Psychology Review, 3*(3), 149-170.

Cooper, S., Dunn, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In W. Dann (Ed.), *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '03) (pp. 191-195). New York, NY: ACM.

Dabbagh, N. (2003). Scaffolding: An important teacher competency in online learning. *TechTrends, 47*(2), 39-44.

Department of Immigration and Citizenship. (n.d.). *Conditions for working while studying*. Retrieved March 3, 2008, from http://www.immi.gov.au/students/students/working_while_studying/conditions.htm

Duke, R. (1997). In search of the inverse curriculum. In H. Søndergaard & J. Hurst (Eds.), *Proceedings of the 2nd Australasian Computer Science Education Conference* (ACSE '97) (pp. 65-70). New York, NY: ACM.

EDUCAUSE. (2006). 7 things you should know about screencasting. Retrieved June 5, 2006, from http://www.educause.edu/ir/library/pdf/ELI7012.pdf

Ericson, K. A., & Simon, H. (1993). *Protocol analysis: Verbal reports as data*. Cambridge, MA: MIT Press.

Gries, D. & Gries, P. (2001). *ProgramLive*. New York, NY: Wiley.

Hill, J. R. & Hannafin, M. J. (2001). Teaching and learning in digital environments: The resurgence of resource-based learning. *Educational Technology Research and Development, 49*(3), 37-52.

Jonassen, D. H. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development, 39*(3), 5-14.

Jonassen, D. H. (1994). Technology as cognitive tools: Learners as designers. Retrieved June 3, 2005, from http://itech1.coe.uga.edu/itforum/paper1/paper1.html

Kölling, M. (1999a). The Blue language. *Journal of Object-Oriented Programming, 12*(1), 10-17.

Kölling, M. (1999b). Teaching object orientation with the Blue environment. *Journal of Object-Oriented Programming, 12*(2), 14-23.

Kölling, M. (2005). Using BlueJ to start an OO intro course. Paper presented at the *36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE-05)*, St. Louis, MO, February 23-27.

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

Kölling, M., & Rosenberg, J. (1996). An object-oriented program development environment for the first programming course. In K. J. Klee (Ed.), *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '96) (pp. 83-87). New York, NY: ACM.

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. In D. Finkel (Ed.), *Proceedings of the 6th Annual Conference on Integrating Technology into Computer Science Education* (ITiCSE-01) (pp. 33-36). New York, NY: ACM.

Kölling, M., & Rosenberg, J. (2002). *BlueJ – The hitchhiker's guide to object orientation*. Mærsk McKinney Moller Institute for Production Technology, University of Southern Denmark, Technical Report 2002, No. 2. Retrieved January 10, 2007, from http://www.mip.sdu.dk/mik/papers/hitch-hiker.pdf

Kalyuga, S. (2000). When using sound with a text or picture is not beneficial for learning. *Australian Journal of Educational Technology, 16* (2), 161-172.

Koffman, E., & Wolz, U. (1999). CS1 using Java language features gently. In B. Manaris (Ed.), *Proceedings of the 4th Annual SIGCSE/SIGCUE Joint Conference on Integrating Technology into Computer Science Education* (ITiCSE '99) (pp. 40-43). New York, NY: ACM.

Lee, M. J. W. (2008). *BlueJ and screencasting study at UB: Data collection instruments*. Retrieved 16 February 2008, from http://uob-community.ballarat.edu.au/~mlee/bluej_screencasting/

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., et al. (2006). Research perspectives on the objects-early debate. In M. McNally (Ed.), *Working group reports from ITiCSE on Innovation and technology in computer science education* (ITiCSE-WGR '06) (pp. 146-165). New York, NY: ACM.

Luker, P. A. (1989). Never mind the language, what about the paradigm? *SIGCSE Bulletin, 21*(1), 252-256.

Luker, P. A. (1994). There's more to OOP than syntax! *SIGCSE Bulletin, 26*(1), 56-60.

Machanick, P. (2007). Teaching Java backwards. *Computers and Education, 48*(3), 396-408.

Mayer, R. E. (1999). Multimedia aids to problem-solving transfer. *International Journal of Educational Research, 31*(7), 611-623.

Mayer, R. E. (2001). *Multimedia learning*. New York, NY: Cambridge University Press.

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

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American Psychologist, 59*(1), 14-19.

Mayer, R. E., & Moreno, R. (1998). A cognitive theory of multimedia learning: implications for design principles. Paper presented at the *15th ACM SIGCHI Conference on Human Factors in Computing Systems (CHI-98)*, Los Angeles, CA, April 18-23. Retrieved November 5, 2006, from http://www.unm.edu/~moreno/PDFS/chi.pdf

Meyer, B. (1993). Towards an object-oriented curriculum. *Journal of Object-Oriented Programming, 6*(2), 76-81.

Paivio, A. (1986). *Mental representations: A dual coding approach*. Oxford, England: Oxford University Press.

Pendergast, M. O. (2006). Teaching introductory programming to IS students: Java problems and pitfalls. *Journal of Information Technology Education, 5*, 491-515. Retrieved February 14, 2008, from http://www.jite.org/documents/Vol5/v5p491-515Pendergast128.pdf

Piaget, J. (1973). *To understand is to invent: The future of education*. New York, NY: Grossman.

Poindexter, S. (2003). Assessing active alternatives for teaching programming. *Journal of Information Technology Education, 2*, 257-265. Retrieved February 14, 2008, from http://www.jite.org/documents/Vol2/v2p257-265-25.pdf

Proulx, V. K., Raab, R., & Rasala, R. (2002). Objects from the beginning – with GUIs. In D. Finkel (Ed.), *Proceedings of the 7th Annual Conference on Integrating Technology into Computer Science Education* (ITiCSE '02) (pp. 65-69). New York, NY: ACM.

Ragonis, N., & Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. In J. Dougherty (Ed.), *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '05) (pp. 226-230). New York, NY: ACM.

Reges, S. (2006). Back to basics in CS1 and CS2. In M. Schneider (Ed.), *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '06) (pp. 293-297). New York, NY: ACM.

Roberts, E. & Picard, A. (1998). Designing a Java graphics library for CS1. In G. Davies & M. OhEigeartaigh (Eds.), *Proceedings of the 3rd Annual SIGCSE/SIGCUE Joint Conference on Integrating Technology into Computer Science Education* (ITiCSE '98) (pp. 213-218). New York, NY: ACM.

Rumelhart, D. E., & Norman, D. A. (1978). Accretion, tuning and restructuring: Three modes of learning. In J. W. Cotton & R. L. Klatsky (Eds.), *Semantic factors in cognition* (pp. 37-53). Hillsdale, NJ: Lawrence Erlbaum.

Smith, P. A., & Boyd, G. (2001). Introducing OO concepts from a class user perspective. *Journal of Computing Sciences in Colleges, 17*(2), 152-158.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*(2), 257-285.

Sweller, J., Chandler, P., Tierney, P., & Cooper, M. (1990). Cognitive load as a factor in the structure of technical material. *Journal of Experimental Psychology: General, 119*(2), 176-192.

Sweller, J., van Merrienboer, J., & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review, 10*(3), 251-296.

Techsmith Corporation. (2007). *Camtasia Studio screen recorder for demos, presentations and training*. Retrieved April 28, 2007, from http://www.techsmith.com/camtasia.asp

Udell, J. (2004). *Name that genre: Screencast*. Retrieved February 17, 2005, from http://weblog.infoworld.com/udell/2004/11/17.html

Vygotsky, L. S. (1978). *Mind and society: The development of higher mental processes*. Cambridge, MA: Harvard University Press.

Young, M. F. (1993). Instructional design for situated learning. *Educational Technology Research and Development, 41*(1), 43-58.

Zhu, H. & Zhou, M. (2003) Methodology first and language second: A way to teach object-oriented programming. In R. Crocker & G. L. Steele, Jr. (Eds.), *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '03) (pp. 140-147). New York, NY: ACM.

Biographies

Mark J. W. Lee is an Honorary Research Fellow with the School of Information Technology and Mathematical Sciences, University of Ballarat, and an Adjunct Lecturer with the School of Education, Charles Sturt University. He was previously a Lecturer with the School of Information Studies, Charles Sturt University. Prior to this he was Head of the Faculty of Computing and Information Technology, Martin College. His research focuses on educational technology and e-learning, most recently instructional uses of "Web 2.0" technologies, mobile devices and computer games, as well as on the pedagogy of computer programming. Lee is a Senior Member of the Institute of Electrical and Electronics Engineers (IEEE), the Australian Computer

Society (ACS), and the Association for Computing Machinery (ACM). He serves as Chair of the New South Wales Chapter of the IEEE Education Society.

Sunam Pradhan is a Lecturer with the School of Information Technology and Mathematical Sciences, University of Ballarat, where he coordinates the Master of Information Technology and Master of Information Technology Studies programs. He was previously a Program Manager in Information Technology at the Melbourne Institute of Technology, and has extensive industry experience as an analyst/programmer. Pradhan's current research interests are in mobile and web technologies, specifically mobile transactional middleware to support web services, as well as in the pedagogy of computing, particularly in the field of programming. He is a Member of the Australian Computer Society (ACS).

Barney Dalgarno is an Associate Professor with the School of Education, Charles Sturt University, and a Research Fellow with the Centre for Research in Complex Systems (CRiCS) at the same university. His research interests lie in desktop virtual reality learning environments, as well as constructivist computer-assisted learning theories, techniques and tools. His Ph.D. work examined the characteristics of 3D environments and their potential contributions to spatial learning. Dalgarno is currently studying the application of brain imaging though Functional Magnetic Resonance Imaging (fMRI) to interactive multimedia research, and is also involved in a project funded by the Carrick Institute for Learning and Teaching in Higher Education on the implications of teaching the "Net Generation." He is a Member of the Australasian Society for Computers in Learning in Tertiary Education (ASCILITE), of which he serves on the Executive Committee.