2015

Using Cognitive Load Theory to select an environment for teaching mobile apps development

Raina Mason
Southern Cross University, raina.mason@scu.edu.au

Graham Cooper
Southern Cross University, graham.cooper@scu.edu.au

Barry Wilks
Southern Cross University, barry.wilks@scu.edu.au

Publication details

Mason, R, Cooper G &Wilks, B 2015, 'Using Cognitive Load Theory to select an environment for teaching mobile apps development', in D D'Souza & K Falkner (eds), *Proceedings of the 17th Australasian Computing Education Conference*, Sydney, Australia, 27-30 January, The Conference in Research and Practice in Information Technology (CRPIT) series; 160, Australian Computer Society, Sydney, Australia, pp. 47-56. ISBN: 9781921770425

Using Cognitive Load Theory to select an Environment for Teaching Mobile Apps Development

Raina Mason
Southern Cross University
raina.mason@scu.edu.au

Graham Cooper
Southern Cross University
graham.cooper@scu.edu.au

Simon
University of Newcastle
simon@newcastle.edu.au

Barry Wilks
Southern Cross University
barry.wilks@scu.edu.au

Abstract

After considering a number of environments for the development of apps for mobile devices, we have evaluated five in terms of their suitability for students early in their programing study. For some of the evaluation we devised an evaluation scheme based on the principles of *cognitive load theory* to assess the relative ease or difficulty of learning and using each environment. After briefly presenting the scheme, we discuss our results, including our findings about which mobile apps development environments appear to show most promise for early-level programming students.

Keywords: mobile apps, programming education, computing education, cognitive load theory

1 Introduction

The teaching of programming is generally situated in the context of some sort of programming language environment. There have been, and possibly still are, courses that teach programming concepts in the abstract, with no writing or execution of code; but when a course involves code writing and execution, it must necessarily carry out these steps in some sort of environment, whether it be a command-line environment with a simple text editor or a comprehensive Integrated Development Environment (IDE).

Programming environments incorporate not only programming language processors but also tools for many ancillary tasks such as editing, debugging, and file management. We suggest that some of these environments may be so complex as to have an adverse impact on learning outcomes. Professional development tools such as Eclipse (www.eclipse.org) and Visual Studio (www.visualstudio.com) incorporate facilities for advanced programming concepts such as code sharing, versioning, profiling, and more, giving them the potential to be overwhelmingly complex for beginning students. It is important to note that such advanced concepts and capabilities are unnecessary for the purpose of teaching novices introductory programming concepts and skills. These are not only extraneous to the task of teaching and learning introductory level programming, but may well be distracting to learners' focus of attention, overloading their cognitive resources and reducing the capacity of their cognitive processes for learning.

On the other hand there are environments specifically designed for teaching purposes, such as Alice (www.alice.org) and BlueJ (www.bluej.org), which provide the essential tools for learning application development. Between these extremes there are tools such as LiveCode (livecode.com) and App Inventor (www.appinventor.org), which appear to be designed for ease of use, but also to suit continued use by more experienced programmers.

The choice of a programming environment for a particular course hinges upon many factors (Mason & Cooper, 2014; Simon & Cornforth, 2014), including

- the programming language to be used
- the desire to give students experience in a professional development environment
- the availability of teaching aids such as textbooks
- the personal preferences and expertise of the people designing and teaching the course
- cost to students and/or the institution
- access to suitable hardware
- suitability for the purpose of teaching and learning

In this paper we focus on the last of these criteria, pedagogy. In a recent survey of Australian and New Zealand computing academics (Mason and Cooper, 2014), this was the top ranked criterion for selection of a programming language for teaching an introductory programming course, and one of the top four reasons for choosing a development environment.

In considering a possible course in programming apps for mobile devices, we have investigated a number of relevant environments in terms of their usability, including both ease of learning and ease of subsequent use. In this paper we concentrate on the usability of programming environments without considering the many other factors that must contribute to the choice. We go further by developing a method for evaluating usability based on cognitive load theory (CLT) (Sweller, 1994; Sweller, Ayres & Kalyuga, 2011). This method is then applied to several programming environments that were available at the time of writing, and the results discussed.

Other researchers have evaluated IDEs in various ways. For example, Dujmović and Nagashima (2006) used a highly quantitative approach to compare three Java

IDEs, but from the perspective of professional developers. And Kline and Seffah (2005) survey a number of projects that used interviews, questionnaires, or observation to evaluate particular IDEs. However, we are not aware of any prior work using a quantitative approach based on cognitive load theory to assess and compare IDEs.

2 Cognitive Load Theory

2.1 Human cognitive architecture

Humans are limited to a working memory capacity of about seven items (Miller, 1956). Miller observed that this capacity was effectively standardised across our senses. Irrespective of whether the mode of presentation was visual, auditory, taste, or smell, people reliably demonstrated a working memory capacity of 7 (plus or minus 2) for unrelated, 'random' items of information (or stimuli). This was an important observation because it suggested that our capacity to perceive and process the world around us is channelled through a central executive, associated with consciousness that is strictly bounded and relatively small, at least compared to our long-term memory store.

2.2 Expert performance

The kernel of cognitive load theory lies in the argument that the architecture of human cognitive processes, with its limited working memory capacity, may be easily overloaded, whereupon cognitive performance will falter (Sweller, 1988).

There is an apparent contradiction in this when one considers the expert performance of people such as the readers of this paper in the area of computer programming. As you work in an IDE, surely you are attending to and processing and organising and manipulating and coordinating many more than just seven items of information associated with sequences, selections, iterations, objects, variables, functions, properties, methods, and so on.

There are, however, some important riders to this. You do not perceive all of these items as random, disassociated items, but rather, as deeply intertwined and interacting. You have acquired a vast knowledge base regarding the area of programming concepts and skills, and so you are able to effectively work around the limitations of working memory because each of the items that you attend to and process is in fact a highly complex array of information that could be unpacked and deployed into many constituent components. This is a key feature of expertise: you have developed complex schemas that hold well defined, hierarchically structured organisations of knowledge (Chi et al, 1982).

The second key feature of expertise is that experts can attend to, and process, activities in their area of expertise with very low levels of conscious attention (Kotovsky et al, 1985). This is akin to being able to perform on 'automatic pilot', and the terms 'automation' and 'automaticity' have been used to reflect this (Cooper and Sweller, 1987; Shiffrin and Schneider, 1977).

In contrast, novice learners in programming lack both the schemas and their consequent automation that are held by experts. Novices, when seeking to attend to the same information as the expert, must carry it as many more, smaller, packets of information. The fact that they are smaller in size does not help their cause. It is the *number* of elements that is critical, and for novices, this number will probably approach, and possibly exceed, their critical threshold level of cognitive capacity.

2.3 Sources of cognitive load

Cognitive load stems from three sources: intrinsic, extraneous and germane (Sweller, 2010). *Intrinsic cognitive load* is that load imposed by the inherent complexity of the material to be learnt. This is highly dependent upon the level of element interactivity between the individual elements of the information to be learnt, rather than the numerical count of elements per se (Chandler and Sweller, 1991; Sweller et al, 1990). For example, manipulating an array within a loop will impose a higher intrinsic cognitive load than assigning a value to a variable. A common teaching practice is to work from tasks with low levels of element interactivity, and thus low levels of intrinsic load, to those with higher levels of element interactivity, and thus higher levels of intrinsic cognitive load.

Extraneous cognitive load is the load imposed by the way in which information is presented, and depends upon the format of instructional materials and the nature of student activities (Ayres & Sweller, 2005). In the context of teaching programming, extraneous load will also be imposed by the interface that the student is required to navigate in undertaking the instructional materials and learning activities.

Germane cognitive load is the load that occurs as a result of the learners' conscious focus of attention to deliberately remember and understand the learning material (Paas & Van Merriënboer, 1994). That is, germane cognitive load is applied to the actual process of learning.

These three sources of cognitive load are additive, and combine to produce a total cognitive load for each instant of time during an instructional event or learning activity. If, at any point in time, the total cognitive load exceeds the capacity of cognitive resources, then by definition, some aspects of information being attended to must be dropped from consciousness; comprehensions will be lost and learning will be impeded.

Cognitive load theory posits that while the intrinsic complexity of a task remains fixed, the extraneous load may be reduced through re-engineering the instructional materials and/or the learning activities (Sweller et al, 2011). With extraneous cognitive load thus reduced, the released cognitive resources may be re-allocated to the germane aspects of schema acquisition and automation, thus facilitating learning (Paas et al, 2003).

Researchers have identified several specific instructional design principles based on cognitive load theory and have empirically demonstrated their effectiveness. These principles include the worked examples effect (Sweller & Cooper, 1985), the goal free problem effect (Ayres & Sweller, 1990), the split attention effect (Chandler & Sweller, 1991), the redundancy effect (Chandler & Sweller, 1991), the modality effect (Tindall-Ford, Chandler & Sweller,

1997), and the expertise reversal effect (Kalyuga, Ayres, Chandler & Sweller, 2003).

2.4 IDE as a source of cognitive load

Cognitive load theory specifically addresses situations where students are tasked with *learning*. Learning requires cognitive processes to attend to information in working memory, then to organise this new information and manage its transmission to long-term memory, where it will become embedded and organised into existing knowledge, evolving as an increasingly complex network of schemas. The limitations of working memory can become a bottleneck, constricting the interplay of information between the various memory stores.

To make matters worse, the cognitive resources required for the learning process need to actively compete with the demands placed upon resources for attending to and processing other matters. For novices working on a programming task, this will include attending to the relatively many conceptual items of information associated with programming, along with the means of accessing and implementing them by way of components of the IDE.

The instructional design considerations for teaching and learning programming will thus need to consider the extent to which the organisation and presentation of tools in an IDE either increase or decrease the extraneous cognitive load associated with accessing and implementing programming concepts and tasks.

2.5 Assessing cognitive load

The cognitive load experienced during a learning transaction is often assessed by means of a questionnaire or similar instrument. An early instrument was that of Paas (1992), which has formed the basis for numerous studies (Paas et al, 2003). Morrison et al (2014) propose a version specific to computer programming, and present a preliminary report on its use to assess the cognitive load of lectures in an introductory programming course.

If a lecture is found to entail a high cognitive load, it can generally be redesigned to reduce that load. However, this scope for redesigning does not apply to programming environments, which are essentially fixed and invariant. With such environments, the instructor who is aware of cognitive load theory would want to choose an existing environment that offers a low cognitive load when used for the types of task that are typically undertaken by novice students. To this end, we propose a means of assessing the cognitive load associated with programming environments, and apply it to a number of development environments for mobile apps.

When designing teaching and learning resources, teachers should consider the extent of the students' prior knowledge in the content domain. The selection of a development environment will depend on many elements of prior knowledge, for example of the

- environment's programming languages
- target device hardware
- target operating system
- environment's operating system
- general programming concepts

This is a baseline that must be determined when evaluating the suitability of an environment. In the method described in the following section we will assume that this baseline has been clearly accepted and already considered when selecting products for evaluation. This will ensure that consideration and evaluation of the cognitive load factors will be normalised across the evaluation of different products with respect to the level of prior knowledge.

3 Mobile App Development Environments

Just a few years ago it seemed reasonable when discussing the infrastructure for a mobile app development course to consider just one option for Apple iOS devices and one option for Android devices. Goadrich and Rogers (2012) did this, considering XCode and Eclipse. Other platforms and environments were acknowledged, but these two were considered to offer sufficient coverage of the field.

Since then many more development environments have appeared, and the Windows phone has started to make some inroads on the market. The choice of development environments and programming languages is no longer so straightforward.

In the first instance we identified 15 environments that might be worth considering as the basis for an early course on developing mobile apps. This might be a first programming course, or it might be a first mobile apps development course following a more generic introductory programming course. We had decided that we were interested in the development of native apps rather than Web apps; that is, the apps should run directly on the device rather than through a browser.

Initial exploration of these environments led us to narrow the field to just five; in the following subsections we explain our reasoning.

3.1 Environments discarded as too simple

A number of environments appear to be designed for non-programmers, to the point where we did not consider them useful for teaching programming. Having established this, we did not further investigate these environments.

BuzzTouch (www.buzztouch.com) is a medium for designing screens using interface elements and preset behaviours. The actual code generation is carried out in another environment.

Socialize appmakr (www.appmakr.com), Infinite Monkeys (www.infinitemonkeys.mobi), and Orbose (orbose.com) are all menu-based applications with limited functionality, and do not appear to facilitate 'coding' as it is generally understood in computing education.

3.2 Environments discarded as too complex

Several environments appear very much targeted to the professional developer. These environments might well be suitable for students at higher levels of study, but we considered that they would prove too daunting for novice programmers. Indeed, even to install some of them was a major undertaking, requiring multiple reboots and the interpretation of enigmatic error messages. This is

something that we would prefer to avoid with beginning students.

Netbeans with Google Android plugin, Eclipse with Google Android libraries, and IntelliJ IDEA with Google Android libraries (www.jetbrains.com/idea), all fell into this category of professional development environments in which there were problems installing either the environments themselves or the supplementary libraries.

Telerik Icenium (www.icenium.com) was ruled out of consideration because it appears to rely on existing advanced HTML5/CSS/Javascript skills, which we cannot assume novice programming students will have.

3.3 Environments discarded for other reasons

GameSalad (gamesalad.com) is designed specifically for writing game apps as opposed to general app programming.

AIDE (www.android-ide.com) was discarded because its code must be written on an Android device: this environment offers no way of writing code on a computer and transferring it to the device. This renders it unsuitable unless we can be sure that every student in the class will have access to an Android device.

3.4 Environments selected for further study

Having eliminated the environments listed above, we were left with just five environments for further investigation.

Visual Studio is an environment used in more than 15% of introductory programming courses in Australia and New Zealand (Mason & Cooper, 2014). **Visual Studio Express for Windows Phone** is a recent variation that permits development on a Windows computer of apps for a Windows phone.

App Inventor (appinventor.mit.edu) is a web-based environment that is used to develop apps for Android phones. In a similar way to Scratch (Maloney et al, 2010), code is built from jigsaw-like code-snippet blocks by dragging them to an editing screen and fitting them together.

TouchDevelop (www.touchdevelop.com) is a web-based environment designed to develop apps for Windows phones, and has recently been extended to include Android devices. Unlike App Inventor, TouchDevelop has a textual form for its code; but because the language was designed to be programmed from smart phones, most of the text entry is carried out by tapping screen buttons rather than from a character-based keyboard.

LiveCode (livecode.com) runs on a Windows, Macintosh, or Linux system and produces mobile apps for Apple and Android devices, as well as desktop applications. LiveCode is a more traditional text-based language, with code entry from a normal keyboard. While LiveCode is designed for writing mobile, desktop, and server applications, its current promotion appears to be aimed primarily at the development of mobile apps.

Xamarin Studio (xamarin.com) runs on a Windows computer to develop apps for Apple, Android, and Windows devices, using C# in an IDE somewhat similar to that of Visual Studio.

Of these environments, App Inventor, TouchDevelop, and Xamarin have been designed specifically for mobile applications development. In the case of Visual Studio a specific version was available for mobile development (Visual Studio Mobile 2012) at the time of evaluation.

All five of these environments are undergoing rapid change at the time of writing. In the rest of this paper we shall report on the environments as we found them in the first half of 2014, expecting that aspects of them will have changed, perhaps substantially, by the time this paper is published.

3.5 Other considerations

There are many reasons why a specific mobile application development environment may be chosen for teaching. These include

- cost to students
- relevance to industry
- number of phone features (eg gyroscope, camera, phone book) available via the environment

Table 1 shows a comparison of these features, and more, for our five chosen environments.

4 Assessing Cognitive Load in Mobile App Development Environments: Method and Application

We set out to investigate the usability of these five environments, with the goal of assessing how suitable each might be as an environment for teaching mobile app development to reasonably inexperienced programming students. We devised a four-step process for the evaluation of the programming environments:

1. choose a selection of small problems whose solutions offer coverage of various aspects of the target environments;
2. record a video screen capture and verbal narration of an experienced teacher solving each problem;
3. view and evaluate the recordings using the cognitive load theory factors discussed below;
4. analyse and compare the evaluations to determine a scoring and ranking of the products studied.

We will describe our method in a general sense as we believe that it will be useful for any instructor selecting a programming environment for any development task. At the same time we will work through our application of the method to the specific task of choosing suitable environments for developing mobile apps.

4.1 Selection of problem set

A programming environment usually contains a rich source of components or tools for development. To evaluate the complexity of an environment, problems were chosen to use small but distinct sets of individual components. Although it would be possible to devise problems that exercise large numbers of components, the interaction between components would increase complexity and complicate the final analysis, so we chose tasks that exercise as few components as possible.

The task descriptions do not have to be strict. The aim is to observe and analyse the use of components of the development environment. Minor variations in interpreting the task will not significantly restrict analysis of its presentation and use. Similarly, while different problem solvers might chose different solutions to the

problem, all will entail the targeted environment component.

For example, in targeting mobile apps development environments, we considered three distinct areas. First there is the coding itself. Second, these environments tend to incorporate a separate graphical area for designing the user interface. Finally, different environments will have different ways of managing external media, which are of high importance in mobile apps. So for our evaluation of mobile development environments we chose three separate tasks that together exercise the coding, layout, and media aspects of the environments.

Task 1: Hello world program

This task requires the developer to create an application that displays a "Hello, world!" message when a button is clicked. This was considered to be the simplest mobile application that requires processing of user actions, and that does not correspond to a default application provided in any of the development environments. A key facet of this task is the ability to create a testable application using the environment's application building tools.

Task 2: Animal display

This task requires the display of four animal images as chosen by a selection widget. The selection widget type is not specified, as this might vary with environment and target platform, but the developers were expected to use the simplest possible widget for this task. The animal images are to be in common graphic formats. This problem exercises the environment's media processing ability, in this case with images. The selection widget also had to be more complex than a simple button, in order to cater for the four-way choice.

Task 3: Hello world permute

This task requires the display of permuted versions of the string "Hello, world!" on the press of a button. This task exercises the programming language part of the environment by requiring a small but non-trivial text-manipulation computation to be programmed. This task has previously been used by Goadrich and Rogers (2012) to compare two environments, and by Simon and Cornforth (2014) to further compare those with a third environment.

These tasks were selected on the basis that they would progress from the simplest possible task, through one with a little more complexity in both the graphical user interface and the coding, to one in which the algorithm and the coding might appear fairly complex to a novice programmer.

4.2 Recording the solution

Solutions to the problems identified above were recorded by screen capture software, with the recording including a think-aloud narration to help the evaluators understand why particular actions are taken during the process.

Three of the four authors were each allocated one or two environments, in which they undertook all three tasks, producing 15 screencasts in total.

Some screencasts included steps that are not strictly necessary for completion of the task, such as changing the default names of interface objects. These steps were not counted as part of the activity or included in the evaluation. We also excluded any debugging steps apart from the standard steps required to build and test the final product. Our goal was to compare the environments themselves, and for this we required a straightforward bug-free coding of the simplest solution for each activity.

It would not have been appropriate to have students make the screencasts. Firstly, any problems with the use of the IDE would then be conflated with the students' learning problems. Second, such an approach would entail having students use five different IDEs with different programming languages, which would be a somewhat unusual approach to selecting the IDE and language for a course.

4.3 Evaluation of the recordings

Analysis of each screencast began by iteratively breaking

Table 1: big-picture considerations of the environments

Environment	App Inventor	Touch-Develop	Live-Code	Xamarin	Visual Studio
Cost to students	free	free	free	free	free
Visual cues	yes	yes	yes	yes	yes
Visual debugger	no	yes	yes	yes	yes
Graphical user interface	yes	yes	yes	yes	yes
Difficulty of installation	low	low	medium	high	medium
Cross-platform development	no	yes	yes	yes	no
Relevant to industry	yes	yes	yes	yes	yes
Open source	no	no	yes	no	no
Available support material	high	high	medium	high	medium
Can port to more than one platform	no	yes	yes	yes	no
Number of files user has to manage	0	0	1	many	many
Degree of textuality (block-based → typed code)	low	medium	high	high	high
Phone features available via the environment	high	high	medium	high	high
Required prior knowledge . . .					
Procedural algorithms	yes	yes	yes	yes	yes
Object orientation	no	no	no	yes	yes
GUI widgets and event processing	no	no	yes	yes	yes
Target mobile operating system	no	no	no	yes	yes
Specific programming language	no	no	no	yes	yes

the program development into a series of steps. The same task will usually require different numbers of steps in different environments, and steps that appear in more than one environment may vary in complexity.

To evaluate the steps we devised a series of cognitive load factors that are either directly observable from the screencasts or readily deduced; see Table 2. While these factors might vary according to the activity being addressed, we posit that they are applicable to a broad range of software development tasks.

Table 2: cognitive load factors

CLT Factor	Description
Factors that add to cognitive load; scored as low/medium/high	
EC: Environment schema complexity	Breadth/depth of environment and/or language-specific schemas that are required to perform this step
PC: Programming schema complexity	Breadth/depth of general programming schemas that are required to perform this step
TB: Think back	Number of elements from previous steps needing to be kept in mind to perform this step
I: Interactivity	Complexity of the interactions between environment schemas, programming schemas, and think back required to perform this step
PE: Relevant physical elements	Number of relevant physical elements appearing on screen that may be chosen as part of performing this step
D: Distractors	Number of physical elements in view but irrelevant to performing this step
WP: Windows/ palettes	Number of windows/palettes that are visible and active on screen while performing this step
SA: Split attention source	Extent of physical separation between elements of information or interaction that need to be mentally integrated to order to perform this step
Factors that can reduce cognitive load; scored as present/absent/NA	
PH: Prompts/hints	Instructions viewable in text or graphical form for performing this step
GS: Guiding search	Attention drawn to next element required for performing this step; for example, by highlighting text instructions or target entry field
CS: Context-sensitive help	Help available as scaffold for performing this step; for example, tool tips or other prompts indicating the purpose of an element
G: Groupings	Clustering of elements into related functionality associated with performing this step; for example, automatic indentation, clustering of menu items

We then examined the screencasts in detail, assessing each step according to each evaluation criterion. This resulted in 15 two-dimensional tables, one for each screencast, with rows representing the steps and columns representing the CLT factors.

Finally we describe the scoring system. As indicated in Table 2, factors that add to cognitive load were rated as low, medium, or high, while factors that can diminish cognitive load were rated as present, absent, or not applicable. These ratings were all assigned numerical values, as shown in Table 3.

The scoring scale in Table 3 is obviously non-linear. This is consistent with CLT theory in that a difficult step, imposing high cognitive load, will impact considerably more upon cognitive resources than a simple step that entails lower cognitive load, and might even block progress completely. Given this effect, the choice of the value 4 rather than, say, 5 or 10 is discretionary, but has proven useful for our analysis in these mobile app development environments. Likewise, we note the presence of a cognitive load reduction factor by subtracting 2 from the score for the step. Again, this judgement is discretionary, and is based on the supposition that a factor that reduces cognitive load would typically offset a medium-level factor that increases cognitive load. Both of these factor levels require a level of mindfulness from the user rather than just an acknowledgment of their existence.

The evaluations were arrived at by consensus. Two of the authors jointly evaluated the recordings and then allocated an agreed level (low, medium, or high for loading factors, and present, absent, or NA for the reduction factors) for each entry in the Environment/CLT Factor tables. The other authors checked the evaluations and initiated discussion on any values they were not in agreement with.

There was no assessment of the inter-rater reliability of the method, essentially because of the time constraints in choosing the IDE for a proposed course. Future work would certainly include checking the inter-rater reliability.

A highly reduced example table is shown in Appendix 1. This includes all of the steps in the hello world task in App Inventor, one row for each step; but because of the limited space, only a small sample of the columns, showing five of the load-adding factors and three of the load-reducing factors.

4.4 Analysis of data

In this section we suggest ways of evaluating the data gathered in the previous section. Bearing in mind the overall goal of ranking a number of environments, some standard summative statistics can be applied to each program development task and comparisons made

Table 3: step/factor score scale

Score	Description
1	low – minimal or no cognitive load
2	medium – requires consideration
4	high – substantially present
2	reduction factor present (subtracted)
0	reduction factor not present
0	reduction factor not relevant to this step

between environments. The same measures can also be applied to all three tasks combined, regarding them as a single, more complex, task. Here are the measures that we have devised.

Number of steps: this is the total number of steps required to complete the task in the development environment. It does not consider the difficulty of each step.

Cognitive load score per step (CLSS): the cognitive load score for a step is calculated as the sum of the numeric values of the step's individual CLT factors using the scoring system in Table 3.

The measure is determined by:

$$CLSS = EC + PC + TB + I + PE + D + WP + SA - (PH + GS + CS + G).$$

This measure is important because a step with a high CLSS is likely to overload novice programmers and hence block progression on the programming activity.

Minimum and maximum CLSS: the maximum CLSS over all the steps involved in carrying out the task can be used as a measure of the expertise required to use this environment for this activity. The minimum is provided for completeness.

Mode and median CLSS: the mode and median of CLSS across all steps allow a comparison of the central tendencies of CLT difficulty of each step between environments and between tasks.

Threshold score: this is the proportion of CLSS scores above a threshold value intended to represent high cognitive load for a particular cohort of learners. After examining the scores of each task in each environment, we chose a threshold value of 10 as indicating a step with a relatively high cognitive load *for novice learners*. The proportion of steps with a CLSS over this value is therefore indicative of how much of the task is cognitively taxing for this cohort in each environment.

Average cognitive load per factor (ACLF): averaging the scores for each individual cognitive load factor across all steps allows the relative contribution of various cognitive load factors to be determined. For example, the ACLF for think back (TB) for a particular environment would be calculated by the formula $ACLF_{TB}$ = *sum(TB) / steps*. ACLF will always be between the low and high scores given for each factor. For the scoring that we have used (Table 3), ACLF will be between 1 and 4 inclusive.

Proportion of steps assisted (PSA): for each cognitive load factor whose presence reduces cognitive load (Table 2), this is the proportion of steps in a task that are assisted by that factor, and is therefore something of an offset to ACLF. The PSA for a factor is calculated by the count of the steps in which the reduction factor was present, divided by the number of steps. *PSA = count (reduction factor present) / steps*

Table 4 shows the number of steps, CLSS measures and threshold score for each task in each environment, ordering the environments by increasing number of steps for each task. The table clearly shows how the number of steps and the complexity of steps must be considered together. For example, in the hello world task, LiveCode scored lowest in both the number of steps and the proportion of steps that exceeded our threshold, indicating that it presented the least cognitive load for novice users. However, in the animal display task, Visual Studio presented the lowest number of steps but the highest proportion of steps that exceeded our threshold. This indicates that even though there were fewer steps, a clear majority of the steps would provide excessive cognitive load to a novice user.

Table 5 shows the average cognitive load per factor, highlighting the highest value of each factor for each task. This table allows us to judge the average cognitive load per CLT factor, independent of the number of steps. For example, in the string permute task, Visual Studio provided the highest average cognitive load per step for the think back (TB) factor (2.06), while LiveCode provided the lowest (1.25). This supports our intuition that Visual Studio requires a much higher degree of memory of previous actions than LiveCode, which provides a larger amount of contextual information to its user. In contrast, for the same task, LiveCode provided the highest average environmental schema complexity (EC) factor and Visual Studio provided the smallest. This supports our intuition that more complex algorithms can be more concisely expressed in a traditional programming language than in the more verbose programming

Table 4: measures of cognitive load per step

		CLSS				Threshold
1: Hello World	**STEPS**	**Min**	**Max**	**Mode**	**Median**	**Score**
App Inventor	18	2	10	9	8.5	6%
LiveCode	13	5	8	7	8	0%
Touch Develop	22	2	13	8	7	14%
Visual Studio	18	4	13	8	8	28%
Xamarin	13	8	13	8	10	54%
2: Display Animals						
App Inventor	61	2	16	8	9	41%
Livecode	54	5	16	8	8	20%
Touch Develop	61	2	13	4	6	13%
Visual Studio	48	4	26	16	15	83%
Xamarin	50	6	26	11	11	64%
3: Permute						
App Inventor	60	2	16	9	10	52%
LiveCode	20	5	12	8	8	30%
Touch Develop	88	2	17	8	8	22%
Visual Studio	36	4	23	15	15	83%
Xamarin	26	7	27	8	10	58%

Table 5: average cognitive load of each factor, highlighting the highest value for each task

	ACLF (Average Cognitive Load per Factor)							
1: Hello World	**EC**	**PC**	**TB**	**I**	**PE**	**D**	**WP**	**SA**
App Inventor	1.33	1.11	1	1	***1.56***	1.5	***1.39***	1
LiveCode	1.23	1.08	1.08	1	1.08	1.54	1.31	1
Touch Develop	***1.64***	1.27	1.14	1.18	1.41	1.41	1	***1.32***
Visual Studio	1.33	1.28	1.11	1.11	1.11	***1.72***	1.22	1.06
Xamarin	1.62	***1.62***	***1.23***	***1.31***	1.46	1.38	1.15	1.15
2: Display Animals								
App Inventor	1.56	1.44	1.46	1.51	1.57	1.2	1.16	1.07
Livecode	1.54	1.28	1.24	1.26	1.35	1.59	1.35	1.19
Touch Develop	1.49	1.46	1.3	1.36	1.74	1.05	1	1.15
Visual Studio	***1.96***	1.46	1.58	1.38	***2.6***	***3.29***	***3.19***	1.71
Xamarin	1.76	***2.34***	***2.14***	***2.1***	2.54	2.3	2.08	***1.74***
3: Permute								
App Inventor	1.37	1.85	1.48	1.68	1.48	1.32	1.18	1.17
LiveCode	***1.5***	1.3	1.25	1.4	1.5	1.6	1.65	1.05
Touch Develop	1.4	1.77	1.53	1.67	1.59	1.65	1.01	1.43
Visual Studio	1.19	1.94	***2.06***	1.97	***2.64***	***3.44***	***3.11***	1.44
Xamarin	1.38	***2.19***	1.85	***2.04***	2.19	2	1.65	***1.5***

language of LiveCode. Both Visual Studio and Xamarin scored highly on relevant physical elements (PE), distractors (D) and windows/palettes (WP). For more experienced programmers with developed schemas about programming and the environment, having a large range of palettes and options available on screen will be an advantage as all options have easier access. For novices, the availability of options and palettes provides extra cognitive load which may add to other load and impede learning.

Table 6 shows the proportion of steps assisted by CLT factors which reduce cognitive load. This is useful to consider separately because it provides a judgement on how explicit design choices of the IDEs actually contribute to cognitive load reduction during solution of the chosen problem set. As an overall comparison it can be seen from this table that Touch Develop used all four factors to better reduce cognitive load in all three problems. It is also interesting how the contributing factors varied according to problem type. For example, the string permute task, which required the user to design an algorithm, showed App Inventor providing the lowest amount of cognitive load reduction in its grouping factor. This was not the case for this IDE in the hello world task, where it provided the highest contribution to cognitive load reduction from its grouping factor. This suggests that App Inventor's grouping design was aimed at facets of mobile app development other than the algorithm design.

As a final step we computed the average threshold score of each environment over all three tasks. While an instructor trying to choose between these environments should carefully examine all of the measures, this average has the advantage of being a single measure for an environment, thus permitting a very quick high-level comparison of the environments. The computed averages are as follows:

- TouchDevelop: 16%
- LiveCode: 17%
- App Inventor: 33%
- Xamarin: 59%
- Visual Studio: 65%

These average scores show a clear clustering of results, which will be discussed in the following section.

Table 6: proportion of steps assisted by each load-reducing factor, highlighting highest values

	PSA (Proportion of Steps Assisted)			
1: Hello World	**PH**	**GS**	**CS**	**G**
App Inventor	11%	11%	6%	***78%***
LiveCode	54%	8%	8%	54%
Touch Develop	***64%***	***50%***	***18%***	50%
Visual Studio	11%	17%	0%	50%
Xamarin	15%	0%	0%	38%
2: Display Animals				
App Inventor	23%	3%	5%	51%
Livecode	19%	7%	22%	50%
Touch Develop	***74%***	***38%***	38%	67%
Visual Studio	11%	17%	0%	50%
Xamarin	50%	10%	***48%***	***74%***
3: Permute				
App Inventor	30%	10%	2%	47%
LiveCode	30%	10%	45%	***75%***
Touch Develop	***72%***	***31%***	***50%***	69%
Visual Studio	36%	3%	28%	67%
Xamarin	35%	0%	35%	69%

5 Discussion and Conclusion

In this paper we have addressed the usability criterion for selecting an environment for teaching programming to relative novices. We have introduced a method for evaluating and comparing usability among a selection of candidate products, and have applied this method to the selection of a programming environment for teaching the development of apps for mobile devices.

The mobile development environments that we evaluated fell clearly into three groups. TouchDevelop and LiveCode, with threshold scores of less than 20%, permitted the development of code with the least relative cognitive load. Despite the fact that it was designed for, and is typically used for, novice programmers, App Inventor had double the threshold score of these two environments, indicating a substantially higher relative cognitive load. Both Visual Studio and Xamarin Studio had threshold scores of around 60%, nearly double again, indicating another substantial leap in the relative cognitive load required to develop mobile apps in these environments.

It would be incorrect to conclude that all we have achieved here is to confirm an intuitively obvious result. While different people have different intuitions, we nevertheless imagine that many readers would expect the block-based App Inventor to impose the least extraneous cognitive load, and that is not what we have found. Things that are "obviously" the case sometimes turn out to be incorrect. The tool and methodology presented here is a movement towards enabling objective analysis and comparisons between dissimilar IDEs using Cognitive Load Theory.

Readers interested in applying this method to their own selection of programming environments should remain aware that this single figure is derived from many conflicting factors, and that each factor should be considered in its own right before a decision is made. For example, Table 4 shows that TouchDevelop typically requires more steps than the other environments to carry out the same task. Because coding in this environment is carried out by way of tapping buttons on the screen, coding a loop might be counted as half a dozen separate steps, whereas coding the equivalent loop in a keyboard-based environment might count as a single step. On the other hand, Xamarin and Visual Studio tend to score quite high on cognitive load score per step.

Therefore it does not necessarily follow that early courses in mobile apps development should choose between TouchDevelop and LiveCode. There are many factors involved in selection of a program development environment, and instructors should consider all of the factors that pertain to their circumstances. However, it does follow that if the other factors are more or less equivalent, one of these two environments might be a good choice, especially if targeting novice programmers.

App Inventor has been used for at least one university-level course (Robertson, 2014), but only at the outset,

with a move during the course to a more traditional development environment. We tend to concur with Robertson that App Inventor is in one sense too simple and in another sense too frustrating to form the basis for a full university course.

Visual Studio or Xamarin Studio might be chosen for a mobile apps development course that is not the first programming course, particularly if the same environment had been used for the introductory course, or if the students were already familiar with the programming language chosen. They might also be chosen if the instructors particularly wanted to introduce the students to these environments. However, instructors should be alert to the substantially higher cognitive loads imposed by these environments, and should be prepared to scaffold their novice students appropriately.

The design and application of the cognitive load analysis method described here offers a prospective way for instructors to assess different IDEs in a wide range of contexts. The method is open to modification with respect both to the cognitive load factors that are considered and to the calibration of scoring used to assign values.

6 References

P. Ayres and J. Sweller (1990). Locus of difficulty in multistage mathematical principles. *American Journal of Psychology*, 105(2):167-193.

P. Ayres and J. Sweller (2005). The split attention principle in multimedia learning. In *The Cambridge Handbook of Multimedia Learning*, ed. Richard Mayer, Cambridge University Press, 135-146.

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

M. Chi, R. Glaser, and E. Rees (1982). Expertise in problem solving. In *Advances in the Psychology of Human Intelligence*, Erlbaum, Hillsdale, NJ, 7-75.

G. Cooper and J. Sweller (1987). Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79:347-362.

J. Dujmović and H. Nagashima (2006). LSP method and its use for evaluation of Java IDEs. *International Journal of Approximate Reasoning*, 41:3-22.

M.H. Goadrich and M.P. Rogers (2012). Smart smartphone development: iOS versus Android. *ACM SIGCSE Technical Symposium (SIGCSE'12)*, 607-612.

S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller (2003). Expertise reversal effect. *Educational Psychologist*, 38:23-33.

R. Kline and A. Seffah (2005). Evaluation of integrated software development environments: challenges and results from three empirical studies. *International Journal of Human-Computer Studies*, 63:607-627.

K. Kotovsky, J.R. Hayes, and H.A. Simon (1985). Why are some problems hard? Evidence from tower of Hanoi. *Cognitive Psycholology*, 17:248-294.

J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):16.

R. Mason and G. Cooper (2014). Introductory programming courses in Australia and New Zealand in 2013 – trends and reasons. *16th Australasian Computing Education Conference (ACE2014)*, Auckland, New Zealand, 139-147.

B.B. Morrison, B. Dorn, and M. Guzdial (2014). Measuring cognitive load in introductory CS: adaptation of an instrument. *Tenth International Conference on Computing Education Research (ICER2014)*, Glasgow, Scotland, 131-138.

F.G. Paas (1992). Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *Journal of Educational Psychology*, 84(4):429.

P. Paas, A. Renkl, and J. Sweller (2003). Cognitive load theory and instructional design: recent developments. *Educational Psychologist*, 38(1):1-4.

F. Paas, J.E. Tuovinen, H. Tabbers, and P.W. Van Gerven (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist*, 38(1):63-71.

F.G.W.C. Paas and J.J.G. Van Merriënboer (1994). Variability of worked examples and transfer of geometrical problem-solving skills: a cognitive-load approach. *Journal of Educational Psychology*, 86:122-133.

J. Robertson (2014). Rethinking how to teach programming to newcomers. *Communications of the ACM*, 57(5):18-19.

R. Shiffrin and W. Schneider (1977). Controlled and automatic human information processing II. Perceptual learning, automatic attending and a general theory. *Psychological Review*, 84:127-190.

Simon and D. Cornforth (2014). Teaching mobile apps for Windows devices using TouchDevelop. *16th Australian Computing Education Conference (ACE2014)*, 75-82.

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

J. Sweller (1994). Cognitive load theory, learning difficulty and instructional design. *Learning and Instruction*, 4:295-312.

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

J. Sweller, P. Ayres, and S. Kalyuga (2011). *Cognitive Load Theory*. Springer, New York.

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

J. Sweller and G.A. Cooper (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2:59-89.

S. Tindall-Ford, P. Chandler, and J. Sweller (1997). When two sensory modes are better than one. *Journal of Experimental Psychology: Applied*, 3:257-287.

Appendix: Example evaluation (partial): App Inventor environment with the hello world task.
A number of columns have been removed in order to fit the table on this page.

Step	Description	Notes	Prior knowledge	Complexity of schema required	Relevant physical elements	Distractors	Windows/ palettes	Prompts/hints	Guiding search	Groupings
1	Turn off intro help screen		low	low	low	low	low	yes	no	yes
2	second help screen close		low	low	low	low	low	no	no	no
3	Create new project		low	low	low	low	low	no	yes	yes
4	Name project		low	low	low	low	low	yes	yes	no
5	Add button	Drag and drop	low	low	medium	medium	medium	no	no	yes
6	Change label of button	in properties palette	low	low	medium	medium	medium	no	no	yes
7	Add textbox	drag and drop	low	low	medium	medium	medium	no	no	yes
8	Rename textbox		low	medium	low	medium	medium	no	no	yes
9	Go to blocks	To start editing code	medium	low	low	medium	medium	no	no	yes
10	Go to Button 1	to choose event associated with Button 1	medium	low	medium	low	low	no	no	yes
11	Choose click event	"When Button1 click do .."	low	low	medium	medium	low	no	no	yes
12	Go to Textbox blocks	to choose action associated with Label 1	medium	low	medium	low	low	no	no	yes
13	Choose set text block	"set label1.text to .."	low	low	medium	medium	medium	no	no	yes
14	Go to text blocks	to choose string block	medium	low	medium	low	low	no	no	yes
15	Choose string block	go to text, choose first option (string), drag and drop to right position.	medium	medium	medium	low	medium	no	no	yes
16	set value to "Hello World"		low	low	low	medium	low	no	no	no
17	Connect Emulator	Connect menu - emulator	medium	low	medium	medium	low	no	no	yes
18	Test application	click on button in app	low	low	low	low	low	no	no	no
	Counts:	low	12	16	8	9	11	2	2	14
		medium	12	4	20	18	14	11%	11%	78%
		high	0	0	0	0	0			
	Average (weighted):		1.33	1.11	1.56	1.50	1.39			