A Prototype Structured but Low-viscosity Editor for Novice Programmers

Fraser McKay
School of Computing, University of Kent,
Canterbury, UK. CT2 7NF.
fm98@kent.ac.uk

This paper presents work in progress on a prototype programming editor that combines the flexibility of keyboard-driven text entry with a structured visual representation, and drag-and-drop blocks. Many beginners learn with Java, a traditional text-based language. While text entry is ideal for experts desiring speed and efficiency, there is evidence in the literature that a significant portion of novice errors are related to syntax. Some beginners learn with Scratch, Alice and Star Logo, all of which have drag-and-drop, "block"-based interfaces. Validation makes them less prone to syntax errors, but they are very "viscous" – there is resistance to changing or rearranging statements once they have been entered. The new system combines keyboard input with statements that can still be manipulated with the mouse as whole blocks. Standard text idioms can be used – highlighting code by dragging the mouse, copying & pasting (as text), etc. With CogTool cognitive/keystroke models, we show that the new system effectively overcomes the viscosity found in block-based languages, but it retains much of the error-proofing. Work is ongoing, but there are implications for the design of a new novice programming system.

Programming, Greenfoot, Java, Scratch, Alice, CogTool, viscosity, cognitive dimensions.

1. INTRODUCTION

This paper introduces a new prototype novice programming editor. Its target users are those who have picked up enough programming skill to know what they want to do, but who sometimes make mistakes in the execution of their ideas. The system combines the keyboard-driven editing used by most experts – and intermediate learners, like those who use Greenfoot and BlueJ – with the structured view and drag-and-drop blocks used in Alice and Scratch. Greenfoot is Java-based, with a code editor that uses some highlighting for emphasis, but that is still based on text. In Alice and Scratch (and a few less-common systems, like Star Logo TNG) the programmer uses the mouse to drag-and-drop statement blocks into the editor space. This prevents syntax errors caused by incorrect/missing characters (the most common type of errors that beginners make (Robins, Haden & Garner 2006, Denny et al. 2011)), but is very "viscous" – program statements are difficult to edit once they have been entered.

The new prototype is less viscous than block-based peers, but still presents the program as structured blocks, visually, and prevents many kinds of syntax error. The keyboard controls seem similar, at first, to autocompletion, so they do not feel out of place or completely new. The programmer does not need to remember all of the syntax (unlike pure text), but can have a selection of statements to choose from (as in Alice or Scratch). Factors like the prototype's visual style (its "look") are being designed in parallel with new usability heuristics for this domain (McKay 2012), which we are currently evaluating. Future versions of the prototype are intended to be compatible with the existing Greenfoot tool.

Though there are a range of novice programming systems written about in computing education, we are aware of no systematic studies investigating their specific interactions. Through CogTool – an HCI tool for measuring and predicting user behaviour – it is possible to estimate the time it would take to do something. Since viscosity can be described partly as the effort expended in doing a certain task, this paper uses CogTool models to explore viscosity in Greenfoot, Alice, Scratch, and the prototype. For a selection of entry and manipulation tasks, the new prototype is less viscous than Alice and Scratch, and is similar to Greenfoot. After explaining the prototype's interaction style, this paper describes the tasks used to test the prototype in CogTool. The results are summarised, and the prototype is then discussed in relation to the three other systems.

2. RELATED WORK

2.1 Viscosity

Green (1989) introduced viscosity as one of the "cognitive dimensions" (CDs) of notational systems

(such as programming languages). Viscosity is a measure of local resistance to change. In this context, viscosity can refer to the amount of effort, or the number of steps, needed to add a new statement to a program, to change the condition of, say, an if statement, or to find and delete a particular method or block. It is a particular problem, for example, in diagram-based languages, where inserting something new can force the programmer to rearrange much of the diagram (Green, Blackwell 1998). It is referred to in Pane & Myers's usability heuristics for novice programming systems (Pane, Myers 1996), and is discussed in the new heuristics (McKay 2012). Excessive viscosity frustrates expert programmers – who know what they want to do, and get held up by the system – but after a time it can also demotivate novices. Simply operating the interface can take up effort meant for the task at hand.

2.2 Existing systems

Greenfoot (Henriksen, Kölling 2004) (www.greenfoot.org) is a Java-based system developed at Kent, used for teaching object-oriented programming through games. In Greenfoot, the programmer enters code in a colour-highlighted text editor, similar to that in many IDEs. A similar code editor is used in BlueJ (from which Greenfoot originates) to teach undergraduates. The editor uses subtle background colours to highlight structure, putting boxes around if statements, loops, and other constructs. It also highlights keywords.

In Alice (Cooper, Dann & Pausch 2003) (www.alice.org), the programmer drags-and-drops "blocks" of code into the editor. The blocks can be rearranged with the mouse. Because of the drag-and-drop validation, syntax errors are avoided (it is not possible to enter an invalid statement). Parameters can be added or changed through blocks' context menus, but the structure of the statement itself cannot be changed, nor can its label/caption. To "change" the statement, it has to be removed and replaced with something else. The drag-and-drop editor makes it relatively easy to insert new blocks, but it is more tedious to rearrange or replace blocks once they are in place.

The third main system in this area is Scratch (Maloney et al. 2004) (www.scratch.mit.edu). It has a visually-similar block-based interface, with drag-and-drop, but fewer right-click context menus. Another major difference in Scratch is that blocks "stick" to the blocks above them when they are dragged. This means that additional steps are needed to move a single block, since it must be detached from its neighbours first (so as not to bring them with it). As shown later in this paper, this is a critical point in discussing Scratch's viscosity. StarLogo TNG, developed by some of the same group, uses visually similar blocks, and exhibits the same "sticky" effect. Like Alice, these systems prevent text-based syntax errors.

2.3 CogTool

Keystroke-level models can be used to measure the "overt" movements that a user makes (Card, Moran & Newell 1980). Cognitive models additionally measure hidden "mental" operators, like eye-movement, and reading- and thinking-time. These models, however, are quite complex, and difficult to construct accurately by hand. Non-experts, in particular, can easily introduce errors. It can be difficult to know which mental operators to include and where/when to use them (John 2010).

CogTool (John et al. 2004) is a prototyping tool that automates the creation of cognitive models. The evaluator leads CogTool through screenshots or storyboards step-by-step, demonstrating what the user would do for the task being measured (e.g. clicking a certain button or menu item). CogTool then uses the "Adaptive Control of Thought – Rational" (ACT-R) architecture – a computer model of human cognition (Anderson et al. 2004) – to generate a model of the task. CogTool automates error-prone parts of the modelling process, improving the accuracy of the prediction considerably (John 2010).

2.4 Cognitive activities

The cognitive dimensions refer to several activities: "incrementation", transcription (copying code from a design), modification, exploratory design, searching, and exploratory understanding (Green, Blackwell 1998). Not all cognitive activities are relevant to all systems. Different programming tasks can be mapped to the different activities: adding a statement, deleting a statement, modifying a statement, rearranging the program structure, etc. In this paper, viscosity is primarily measured in the incrementation and modification activities – that is, adding and modifying statements, and moving/rearranging them once they are in place.

3. PROTOTYPE

Figure 1 is a screenshot from the new prototype. In this notation, the cursor can be in either of two states: either horizontal (between blocks) or vertical (the normal text caret, used in text areas). The text caret behaves as standard. When focus moves off of a line of text, either through Enter, the down arrow key, or clicking elsewhere, the cursor becomes horizontal (marked as "1" in the figure). This cursor indicates an insertion point between blocks. When arrow keys are used to move again, focus moves inside the next block. The focus can be either on a block (or part of it), or between blocks. Inside a block's text areas (marked as "2"),

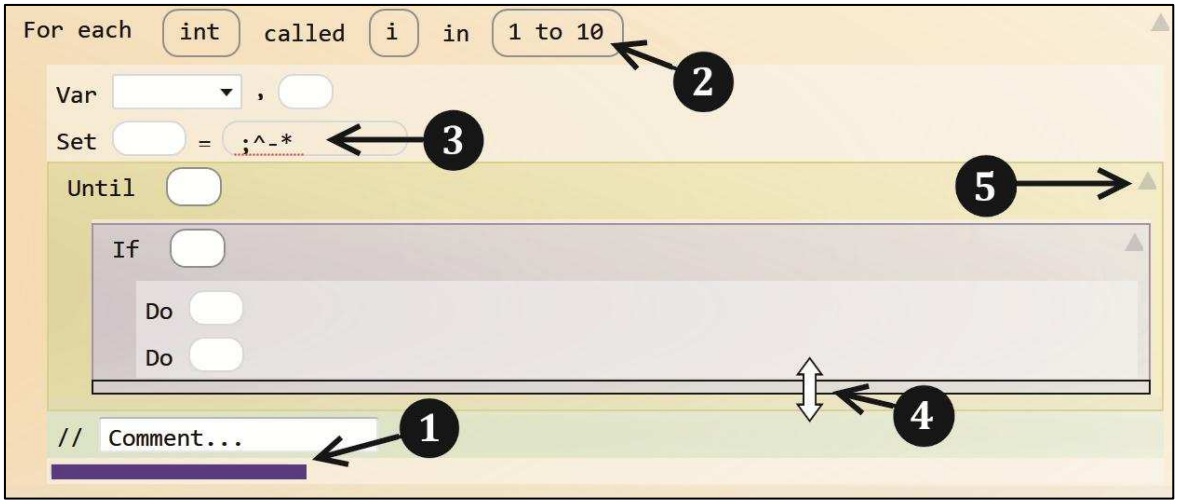

Figure 1. Annotated screenshot of the prototype

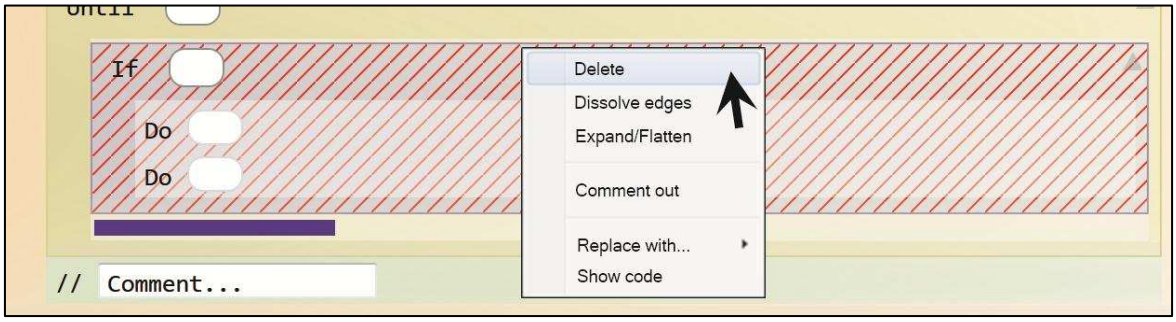

Figure 2. Context menu, delete preview

pressing a key enters text as normal. Pressing Space or a left/right arrow key moves focus to any bordering textbox, as in the "var" and "set" statements in the figure. Between blocks, pressing a key inserts a new block, the type depending on the letter pressed. The pseudocode language that the editor "understands" has a relatively short (but complete) list of statements, each of which has its own accelerator-like letter. Pressing 'F' inserts a "for" block, 'V' declares a new variable, 'I' adds an "If" statement, and so on. The block is added immediately; there is only one block for each character key. It is possible, of course, that the key pressed is not one that is mapped to a block. In that case, we have experimented with three different strategies. Either:

- The character (and any subsequent text) is put in a text prompt-style block. A red underline shows that this is not syntactically correct, like a word processor's spell-checker (marked 3 in Figure 1);
- Focus moves to a popup window that suggests alternatives; this filters results with a live search from the keyboard, to save the programmer having to switch between keyboard and mouse;
- In the first instance, the system ignores the event, or shows only a passive indicator that there is a mistake. If the programmer presses a second, successive incorrect key, the popup dialog appears. This means that a focus-taking popup does not have to appear for every single incorrect key (there is a chance to correct it, by pressing a valid key again).

Delete and Backspace behave as they would with text; in an empty textbox, they delete the block. As with most text editors, Shift+Up and Shift+Down selects a range.

The prototype also has a rich set of mouse interactions. Dragging over a block, or blocks, selects and highlights a range (the same as drag-selecting over text in most editors). Control+click adds an individual block to the selection, and Shift+click adds a range. Blocks can be rearranged with drag-and-drop. Though it is not the primary use case, a Scratch/Alice-like block palette could be used to insert blocks (meaning that statements do not have to be remembered). Nesting blocks, like loops and conditional statements, can be resized along their bottom edge (marked 4). Moving the border moves subsequent statements in or out of scope. Individual blocks can also be collapsed or expanded if desired (marked 5).

Right-clicking a block (or selection range) opens a context menu (Figure 2). Most of the menu items have live mouse-over previews, showing which block(s) will be affected. As well as the extra

information this provides, it is a deliberate error avoidance mechanism – showing a confirmation dialogue every time something is deleted would disrupt the programmer's flow; the live preview is unobtrusive, but does give some warning that this will change the program (though it can always be undone). Deleting a block deletes the whole block, and any other blocks inside it. Many can also be "dissolved": dissolving a loop or if-statement, for example, deletes the parent block, but leaves its contents intact. Dissolving the block in Figure 2 would leave the two "Do" statements alone, but delete the "If" that surrounds them. A block can also be commented-out (surrounded by a comment), or replaced with another, similar, kind of block. The latter makes it possible to swap a loop, for example, with another construct (another kind of loop, or an "if"). The context menu operations reduce viscosity by providing shortcuts to operations that usually require a number of mouse or keyboard actions (like replacing a block).

4. COGTOOL MODELS

CogTool task scripts are called "demonstrations". The demonstrator carries out each task, taking screenshots or video along the way. Key frames are loaded into CogTool, and then made into storyboards. CogTool is walked through the storyboards, similar to how they would be demonstrated to a human user who had not used that interface before. CogTool uses statistical norms to simulate an "average" user's experience, predicting how long would be spent on each sub-part of the task. Because it uses standardised values, it can be used to compare several alternative designs that tackle the same task (so, for example, the Greenfoot, Alice and Scratch designs for adding a new loop). Results can be visualised as a "trace" in CogTool, or, more usefully here, exported to a spread sheet. The trace shows state-changes that occur between sub-tasks – moving a hand from the keyboard to the mouse, for example. The spread sheet contains values measured in seconds, the estimated time(s) it would take a user to complete each sub-task (moving the mouse to a particular place, etc.).

184 CogTool tasks were compared in total: 46 tasks in each of Greenfoot, Alice, Scratch and the prototype. Additional sets of tasks have previously been tested with Greenfoot, Alice and Scratch. Those tasks are very similar, and produced a similar pattern of results in those three systems, but they have not yet been used with the prototype (the work is ongoing). The tasks can be divided into five groups: adding/inserting a statement (n=6), modifying part of the statement (n=8), deleting it (n=12), moving it to somewhere else in the program (n=13), and removing and replacing it with another kind of statement (n=7). Adding and modifying statements map to the cognitive dimensions' "incrementation" and "modification" activities, respectively. In the dimensions framework, viscosity is usually considered "harmful" for these, especially for modification (Green, Blackwell 1998). The complete task list is lengthy, but examples are listed as Table 1. Similar programs, notwithstanding the language syntax, were used in each system for carrying out the same tasks (there are equivalent statements in the languages). The tasks were chosen as examples of real life edits that are likely to be made in many programs. The reason there are more tasks of some types is that adding a statement, for example, takes broadly the same effort whatever and wherever that statement is; replacing or deleting a loop with contents can be more complex than deleting a simple one-line statement – this can also depend on whether any other statements are listed after it (see Scratch discussion).

Table 1. Example task types

Type	Example
Insertion	*Declare a new variable*
Modification	*Change a string parameter in a method call*
Deletion	*Delete the whole of a loop structure*
Moving	*Reverse the order of two variable declarations*
Replacement	*Replace a "for" loop with "while"*

When analysing the individual tasks, some design(s) stand out as especially good, or bad, compared to others. It is possible to qualitatively look at each result for a short selection of tasks. In a longer task set, to produce a concise analysis, it might be beneficial to filter out the most significant results for further investigation. On the data for these tasks, we calculated a z-score to highlight the most important differences. For the z-score of a result, x:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the mean for that task, and σ is the standard deviation, we calculated a score for each task in each system. We may use a filter to pick out results for which the absolute of z was greater than 1. This highlights those results that are significantly higher or lower than their peers, so that they can be further investigated. In a longer task set, this may help to pick out the most significant results quickly. However, for completeness, each of the 46 tasks has been looked at here anyway. That kind of filter is more of an extension, which might be appropriate to consider in future, longer, work.

5. RESULTS

Comparing a single value of viscosity that encompasses all tasks does not provide a

complete picture; they all have particular strengths and weaknesses (none are "better" all of the time). As already discussed, the tasks can be grouped into different categories. In Table 2, the results are organised in these groups. This is more meaningful for discussing broad trends in the different notations.

Table 2. Mean task time summary

	Alice	Scratch	G.ft	New
Insertion	*6.560*	*4.868*	*3.803*	*1.644**
Modification	*7.051*	*5.613*	*5.836*	*5.005**
Deletion	*2.555*	*5.440*	*6.530*	*2.418**
Moving	*3.093**	*5.480*	*12.197*	*4.843*
Replacement	*8.902*	*9.796*	*4.693*	*2.289**
** = least viscous/most efficient*				

6. DISCUSSION

6.1 Comparison – Greenfoot

Greenfoot programs are written in Java syntax; Greenfoot is the development environment. Though the tests were conducted in Greenfoot, the results could be expected to be similar in any Java text editor. In general, text editing was less viscous than the block-based models; however literature elsewhere suggests that its error-proneness is a significant drawback (Robins, Haden & Garner 2006, Denny et al. 2011).

Like many text editors, Greenfoot has an auto-completion feature. This is interesting for comparing to the key-character-driven entry in the prototype. Both are less viscous than the block syntaxes (Table 2), but the prototype is actually less viscous than Greenfoot (for those tasks, at least). Although Greenfoot can auto-complete typing, there are two things that mean this is not quite as fast as the alternative interface: firstly, it requires some level of confirmation. Pressing 'a' for the common Greenfoot "act()" method does not automatically add the whole statement – even assuming that the first suggested method is the one that is wanted, the user has to either press Enter or double-click. Secondly and already touched on, there might be dozens of valid statements that begin with any given letter. That is primarily because Java's syntax (like many languages) means assignments and method calls do not require an initial keyword like "call" or "let". This is generally a good thing, and makes sense in plain Java (it removes extraneous words and is faster to type), but it has been helpful to reintroduce these kinds of keywords for the new kind of interface. It makes it possible to have only one kind of statement associated with each key.

Also of note is that for the moving/rearranging tasks, Greenfoot was significantly more viscous than the block syntaxes. From following the detailed traces from those tasks, it becomes clear that there are two factors involved: when moving several statements, rather than just one, it is necessary to highlight all the statements before they can be dragged anywhere else (which is not unusual, of course). If, however, the selection end-point is a small punctuation mark like a bracket or semicolon (and it is likely to be, in Java), this is a comparatively small target, and the user must slow down to accurately drag over it (an example of Fitts's law, whereby smaller targets are harder to hit (Fitts 1954)). In a language like BASIC, there are not as many punctuation marks. The second factor is that it is not possible to drag selected text to move it – the programmer must cut/paste from either a context menu or a key combination. Overall, these factors add time to the task.

6.2 Comparison – Alice

In Alice, it is necessary to go through a (very large) menu tree to change the parameter in a method call, the counter variable in a loop, etc. Whereas Scratch has small text boxes for these, in Alice they must be entered via the mouse. Complex expressions make this even more viscous – the menus are hierarchical, so that, instead of changing only the "-" to "+" in the expression "5 - (y * 2)", the whole expression has to be re-written.

To change a Java keyword in Greenfoot, it is relatively trivial to overwrite a few characters of text. It is therefore possible to change the keyword of an if-statement to make it a while loop, keeping the brackets and the loop body as they already are. In Alice, the whole block needs to be replaced. For the same if-to-while example, the condition part and the block contents both have to be put aside and then, afterwards, put back into the new block. As seen in Table 2, it takes longer to do this in the block languages than in Greenfoot or the prototype. In the latter, some blocks have menu shortcuts for this (if -> until, until -> for, etc.), removing the need for excess steps.

6.3 Comparison – Scratch

As with Alice, it is not simple to replace a block with another. This even means that it is not possible to quickly replace "turn left" with "turn right", as these are separate statements (separate types of block). This seems a particularly egregious example.

Unlike in Alice, parameters appear as text boxes (in Alice they have to be entered from menus). This makes changing simple literals (numbers and strings) similar to doing so in a text language; less viscous than Alice's menus (as seen in the table).

In Scratch, there is no way to select a specific range of statements at once. When a block is selected/moved from the middle of a program, all of the blocks below it "stick" to it. Moving a block, or range of blocks, means splitting the program just after the part being moved, dragging away the first block of the range, and reattaching the original split

part to close up the gap. This takes much more mouse effort than just dragging a box around some specific blocks (which can be done with text and the prototype's blocks).

Another difference, though less critical, is that Scratch (also Alice) has no way to move the end of a nesting block, like a loop or an 'if'. The equivalent, in Greenfoot, is to insert a new closing brace. In the prototype, the end "border" of those blocks can be dragged up or down to change the end point, and move nearby blocks in or out of scope. Though this is a more occasional task than moving statements (which is quite a basic interaction), it is a task that is more viscous in Scratch than in other editors.

7. CONCLUSIONS

The code editor presented in this paper is still under development. Simulations with CogTool have shown that the new interface can be less viscous than other block-based systems, but its design retains the clear visibility of structure, and the error prevention, normally found in those systems. There are still areas where the constraints need further investigation and refinement – for example, what tone of feedback, if any, is appropriate when an unexpected key is entered? It is also still possible to produce **some** syntax errors in-line, but this appears to be better than the alternative Alice-like menu hierarchy looked at in CogTool. The research has implications for the design of new systems; it is not a given that editors such as these **can only ever be** viscous by design. Certain small enhancements can have an effect on the feel of the system, and can greatly speed up editing for more intermediate to advanced users. The context menus in the prototype, that allow a block to be swapped for something similar, are unobtrusive, but provide quick shortcuts for otherwise-viscous tasks. Block-based languages can still make use of the keyboard without feeling unnatural, and without the constant need to change hands from keyboard to mouse. Even without the validation that drag-and-drop supplies, removing minor punctuation marks like semicolons and braces makes the code more efficient to enter, and importantly, manipulate later. Automatic previews give simple feedback that might avoid the need to repeatedly try something, and then undo it, just to see what it does. We believe that these design changes would suit existing Greenfoot users, as one example.

8. ACKNOWLEDGEMENTS

The author wishes to thank his supervisor, Michael Kölling, and others at the University of Kent who have given feedback on the prototype.

9. REFERENCES

Anderson, J. R. et al. (2004) An integrated theory of the mind. *Psychological Review,* 111 (4). 1036-1060.

Card, S. K., Moran, T. P., & Newell, A. (1980) The keystroke-level model for user performance time with interactive systems. *Communications of the ACM,* 23 (7). 396-410.

Cooper, S., Dann, W., & Pausch, R. (2003) Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin,* 35 (1). 191-195.

Denny, P. et al. (2011) Understanding the syntax barrier for novices. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education.* 208-212.

Fitts, P. M. (1954) The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology,* 47 (6). 381-391.

Green, T. R. G. (1989) Cognitive dimensions of notations. In: *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group.* 443-460.

Green, T. R. G., & Blackwell, A. F. (1998) Design for usability using cognitive dimensions. *Tutorial Session at British Computer Society Conference on Human Computer Interaction HCI'98*.

Henriksen, P., & Kölling, M. (2004) Greenfoot: Combining object visualisation with interaction. In: *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications.* 73-82.

John, B. E. (2010) Reducing the variability between novice modelers: Results of a tool for human performance modeling produced through human-centered design. In: *Proceedings of the 19th Annual Conference on Behavior Representation in Modeling and Simulation*. 22-25.

John, B. E. et al. (2004) Predictive human performance modeling made easy. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 462-470.

Maloney, J. et al. (2004) Scratch: A sneak preview. In: *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*. 104-109.

McKay, F. (2012) HCI for beginner programmers. *Interfaces,* (90), British Computer Society. 22-23.

Pane, J. F., & Myers, B. A. (1996) *Usability issues in the design of novice programming systems* CMU-CS-96-132. Pittsburgh, Pennsylvania: Carnegie Mellon University.

Robins, A., Haden, P., & Garner, S. (2006) Problem distributions in a CS1 course. In: *Proceedings of the 8th Australian Conference on Computing Education*. 165-173.