# Predictive Modelling for HCI Problems in Novice Program Editors

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

Michael Kölling
School of Computing
University of Kent
Canterbury, UK. CT2 7NF
*m.kolling@kent.ac.uk*

**We extend previous cognitive modelling work to four new programming systems, with results contributing to the development of a new novice programming editor. Results of a previous paper, which quantified differences in certain visual languages, and feedback we had regarding interest in the work, suggested that there may be more systems to which the technique could be applied. This short paper reports on a second series of models, discusses their strengths and weaknesses, and draws comparisons to the first. This matters because we believe "bottlenecks" in interaction design to be an issue in some beginner languages – painfully slow interactions may not always be noticeable at first, but start to become intrusive as the programs grow larger. Conversely, text-based languages are generally less viscous, but often use difficult symbols and terminology, and can be highly error-prone. Based on the models presented here, we propose some simple design choices that appear to make a useful and substantive difference to the editing problems discussed.**

*Programming, Scratch, Alice, StarLogo, CogTool, viscosity, patterns.*

## 1. INTRODUCTION

Programming education is highly topical, and there are several actively-developed novice programming tools that have been widely used, and cited in the literature. These range from child-user "block" building systems (like Scratch) to Greenfoot – a Java game-based development tool used in schools – to "pure" visual programming systems, based on flow-chart-style diagrams (such as Lego Mindstorms). There are other systems that sit between the above, such as Alice, StarLogo TNG, and numerous other variations on the "block" metaphor. There are also "mainstream" programming languages that are judged to be the simplest of their kind, used to teach beginners (such as Python, Java, or variants of Basic). All of these systems look, superficially, very different – they range from toy-like graphics, to monospace text, to complex flow diagrams and lines. However, there are interactions that are common to several of the differently-styled editor types, and there are also systems that **look** similar, but behave very differently in terms of interaction design.

In this paper, we extend previous cognitive modelling work to four new programming systems (McKay 2012). The initial goal of that study was to compare several "benchmark" systems to a new editor in development, as part of the design process. Results of the previous paper, which highlighted differences in some visually-similar visual languages, and feedback we received, suggested that there may be other systems that could be approached in this way. This short paper reports on a second series of models, discusses their strengths and weaknesses, and compares them to each other, and to the systems in the first set. We acknowledge, for the record, that viscosity, through task time, is only one of the issues in novice programming systems. A system with low viscosity would not necessarily meet the other (educational) requirements for beginner systems, but observations suggest that excessively viscous interactions may still be problematic for some types of novice user.

## 2. RELATED WORK

### 2.1 Psychology of programming

One important idea from the psychology of programming is viscosity. When working with notations – in this case, computer programs – viscosity is defined as resistance to change (Green 1989). For example, once a program has been (partly) written, viscosity might be encountered when editing an existing statement, rearranging the entered program, or inserting new code somewhere in that which already exists.

Green & Blackwell (1998) define six “cognitive activities” – “incrementation” (adding new code), transcription (copying a design into code, or copying code from somewhere else), modification, exploratory design, searching, and exploratory understanding. The primary activities dealt with in our previous paper were incrementation and modification. Together, these cover adding and modifying statements, moving them once in place, and removing them. In the *Time Scale of Human Action* terms used by Newell and Card (1985), these tasks take place in the “task” and “unit task” scales of the rational and cognitive bands.

### 2.2 Previous paper

In a previous work, we used CogTool to simulate human-like task completion times, for a variety of programming tasks (McKay 2012). That paper was based on cognitive models of 46 broadly-chosen program editing tasks. For simplicity, the tasks have been grouped into five categories (to avoid separately listing 46 tasks $\times$ 8 systems): adding new statements (n=6), modifying part of a statement (n=8), deleting it (n=12), moving it to somewhere else in the program (n=13) and removing/replacing it with another statement (n=7). Where some groups are larger than others, it is because there are multiple variants of the task, applying to different types of statements. The tasks were all simulated using the same cognitive architecture/agent – the software used to conduct the simulations is explicitly designed to facilitate comparing two or more designs like-for-like. As well as task times, we were able to observe the number of steps involved in completing a task. From the differences in simulated times between different systems, we noticed trends in types of task **generally** required more steps and/or time in similar systems, and which required less.
The prior paper covered only Scratch, Alice, Greenfoot, and a new prototype design (which was being described in that paper).

### 2.3 Novice program editors

Programming languages appear frequently; as with any other language domain, there are a great number of languages that have been considered educational. Kelleher & Pausch (2005), for example, categorised 87 educational programming systems, in a paper nearly a decade old. Indeed, two of the triad of “major” systems often discussed in the computing education world – Greenfoot and Scratch (the other being Alice) – were not around at that time. It would not have been possible, here, to investigate every possible variation. However, we have chosen a small number of systems that exemplify certain editing and notational styles, and that are used in a “real” educational context (that is, that they are not purely research languages). The original selection of Scratch, Alice and Greenfoot was based on their respective similarities to the new editor we were developing at the time. It was hypothesised that the main differences would occur between Greenfoot (representing text, in general) and Scratch and Alice (as a pair, since they are superficially similar in structure). There were, in fact, several areas in which they behaved very differently (for example, when deleting or moving an existing statement, but not adding new ones). The rationale for selecting the four additional example systems is discussed in the methodology section, but it is appropriate to describe the distinguishing features of each of the systems here, for readers who are unfamiliar with them.

#### *2.3.1 Alice*
Alice (Cooper, Dann & Pausch 2003) programs are composed of drag-and-drop blocks that represent program statements. Because drag-and-drop allows for validation, syntax errors can be prevented (it is not possible to drop an invalid statement at a given point). Adjustable parameters for a statement can be added or changed through context menus. The structure of the statement remains intact, and cannot be broken up. The block has to be entirely removed, and replaced, if the programmer wants to modify the type of block.

#### *2.3.2 Scratch*
At first, Scratch (Maloney et al. 2004) appears visually similar to Alice, though it uses a much stronger colour scheme. Programs are composed of blocks, which must be dragged to the composition area using a mouse. One difference between Alice and Scratch, noted in the prior paper, is that Scratch 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 overall results.

#### *2.3.3 Greenfoot (inc. Java)*
Greenfoot (Henriksen, Kölling 2004) is a Java-based system that emphasises object-oriented programming (it is closely related to BlueJ) through games. Though Greenfoot’s Java text editor uses font colour and background to some effect in code presentation, its interactions are essentially the same as other text editors’ (Greenfoot’s focus is on the games-based approach, rather than the specific program code used).

#### *2.3.4 StarLogo TNG*
StarLogo TNG blocks are visually similar to Scratch’s. However, there are some differences in the effects alignment and layout have on a block’s meaning. More importantly, for this work, Scratch allows text literals to be entered (into a textbox)

from the keyboard. In StarLogo literals need to be added as separate blocks. From the results in this paper, we can see that that makes StarLogo more viscous to use, and this is discussed later.

### 2.4. Cognitive models

Keystroke-level models can be used to measure the “overt”, or mechanical, 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 complex, and difficult to construct accurately by hand. Non-experts, in particular, can introduce errors into the calculations (John 2010). CogTool (John et al. 2004) is a prototyping tool that automates the creation of cognitive models for specific tasks. The evaluator leads CogTool through screenshots or storyboards step-by-step, demonstrating the end-user’s workflow for the chosen task (such as clicking a button or menu item). CogTool uses a computer model of human cognition to generate a model of the task, and makes predictions for overall task time, for sub-tasks. CogTool automates error-prone parts of the modelling process, improving the accuracy of the prediction considerably, compared to manually-created models (John 2010).

## 3. MODELS

The CogTool models here are based on 46 exemplar tasks, each of which was modelled in all eight systems. The tasks were chosen to cover each of the cognitive activities found in the literature. They are based on use-cases – all of the places where a statement can be entered, or moved from one context to another, and so on. They are not necessarily “equal” in terms of how frequently they occur in real tasks. We hope to have additional data in future that would show which use-cases are the most frequent in real-world novice programs.
In the previous paper Scratch, Alice and Greenfoot were modelled against a prototype editor. Those systems were selected because of their (apparent) similarities to the prototype. To augment those findings, we modelled StarLogo TNG (a block language that is similar to Scratch) and Mindstorms NXT (a diagrammatic visual language, also used in education). While Greenfoot was the only text-based system tested in the previous work, NetBeans’s Java editor has now been included. This separates any effects that are unique to (the current version of) Greenfoot from those that are related to Java syntax (or perhaps, to text in general). The final system used here is Python. Python is used in some teaching contexts – though to a lesser extent than Java – and is included to provide an additional point of comparison for text languages. Python’s syntax is considerably different from Java’s, and it uses fewer special symbols (such as semicolons or braces).

## 4. RESULTS

Mean task times produced by the model, grouped

***Table 1:*** *Old and new predictions for task types (times in seconds)*

| | **Previous paper** | | | | **New predictions** | | | |
|---|---|---|---|---|---|---|---|---|
| | *Alice* | *Scratch* | *Greenft* | *Proto* | *MStorm* | *Python* | *NetB* | *StarLg* |
| Insertion | 6.560 | 4.868 | 3.803 | 1.644***** | 15.950 | 3.950 | 5.085 | 12.501 |
| Modification | 7.051 | 5.613 | 5.836 | 5.005***** | 9.103 | 5.438 | 5.532 | 8.288 |
| Deletion | 2.555 | 5.440 | 6.530 | 2.418***** | 6.508 | 5.530 | 7.820 | 5.586 |
| Moving | 3.093***** | 5.480 | 12.197 | 4.843 | 3.819 | 5.179 | 6.008 | 4.968 |
| Replacement | 8.902 | 9.796 | 4.693 | 2.289***** | 18.546 | 5.162 | 5.102 | 11.547 |
| * = least viscous/most efficient | | | | | | | | |

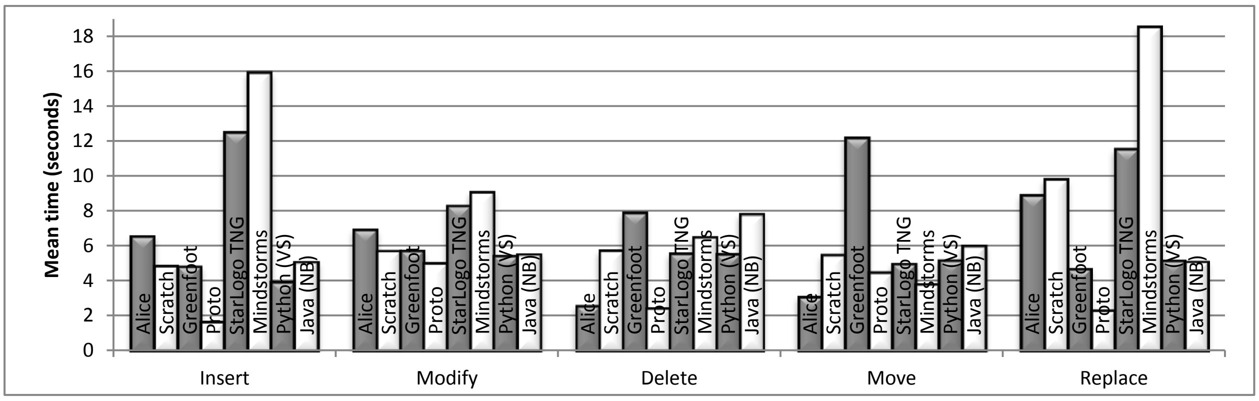

***Figure 1:*** *Mean system times for each task type*

by task type, are shown in Table 1, and illustrated in the figure. As shown, no system is universally "best" across all task types, and scales of the differences between systems (for a given task type) vary. Mean times for the "insert" group, for example, range from 1.644s to 15.950s, with some systems clustered around 3-6 seconds. The "replace" group has a similar distribution. There is less variation in the "modify" group. An analysis of variance (ANOVA) finds that the differences between systems are significant in all groups except "modify". Variance in the "modify" group is not significant overall.

## 5. DISCUSSION

### 5.1 Presentation vs. interaction

Although Scratch, Alice and StarLogo TNG could be referred to as "block" languages, and in some ways **look** very similar, they are very different in terms of interactions.

An example of the relationship between language/notation and editing system is best seen by comparing Greenfoot and NetBeans. Both are Java systems, but, as seen above, have produced differing results in some groups.

In the previous paper, we had not expected the relatively long task times for selecting and moving code in Greenfoot. The CogTool graphs subsequently showed that much of the time and (virtual) effort involved came from the user having to select exactly the right delimiting characters in a Java program construct (semicolons, { } braces, etc.). These are relatively small mouse targets, compared to the systems where a user manipulates whole blocks – an example of Fitts' (1954) law. Though the notation differs visually from the block-based systems, Mindstorms NXT has a similar overall task-time profile as those. Most of the differences occur in tasks which involve manipulating the very small "wires" that connect NXT's circuit-like symbols. In these cases, the fine manipulation required appears to increase task time – similar to the bracket/non-bracket manipulation effect from the text systems here.

Modelling another Java editor – NetBeans – provides an additional set of results. In tasks that require the user to select around a statement, NetBeans is still more viscous than some of the alternatives. However, it is less viscous than Greenfoot in the "move" task group. A detailed look at the two shows that while the two environments approach selection in a similar way, and give similar results, the extra time cost in Greenfoot occurs in the "move" part of the task. In some text editors (including NetBeans) it is possible to cut and paste a highlighted portion of code with drag and drop, though it requires small/fine movement at the end location. Because Greenfoot does not do this, the user must "manually" cut and paste – most quickly done through a right-click context menu. Modelling Python – another text language, one that does not feature C/Java-style braces – demonstrates the difference. Visual Studio's Python editor supports the highlight-and-drag feature mentioned above – making the "move" task type behaviour similar to NetBeans. However, it was more efficient at selecting the code in the first place, because Python lacks the small delimiting characters. Another factor in Java (or languages with a similar syntax) is that pairs of braces must often be "closed up" when part of the code is moved away – surplus braces might need to be removed, and/or extra ones added. The cognitive dimensions refer to these situations as "knock-on" viscosity, where a single change has an indirect effect on other parts of the program (Green, Blackwell 1998).

### 5.2 Comparison to previous results

#### *5.2.1 Viscosity and "sticky" blocks*

One of the biggest causes of viscosity in Scratch, and StarLogo, is the way blocks "stick" to their neighbours when moved. When a block is dragged from the middle of a program, any blocks that are below it are dragged along too. Therefore, when manipulating a single statement, a novice programmer has to detach any neighbouring blocks, carry out the main task, and then reattach the "unwanted" blocks in their original positions (closing the gap that was left). This was noticed by users when we previously conducted a qualitative pilot study using Scratch (McKay, Kölling 2012). It adds several (tedious, or unhelpful, in their opinions) steps which are not found when we compare Scratch or StarLogo to a system like Alice. The pattern is not defined by the visual structure of the blocks (which is often useful), but by the "sticking" effect when interacting with more than one unit of code.

The net effect of this "sticking" varies from program to program. However, the models have been used here to compare a number of tasks in their "with-follower" or "without-follower" variants. This is, for example, the difference between deleting two identical statements – one at the end of a scope/stack/method, without any others trailing underneath; the other mid-program, with neighbours both above and below. This was applicable in fourteen of the modelled tasks, and the effect is summarised in Figure 2. Task time appears unaffected by end-of-stack positioning in Alice (the total difference is less than 0.04% in only one task). In both of the affected systems, the task time increased for all tasks.

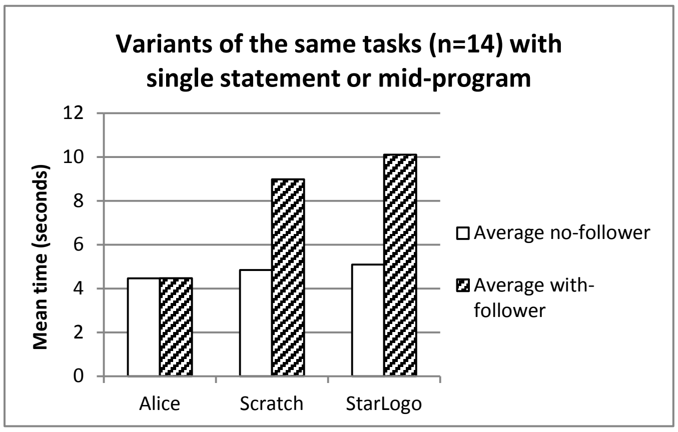

**Figure 2:** *Task times whether or not mid-program*

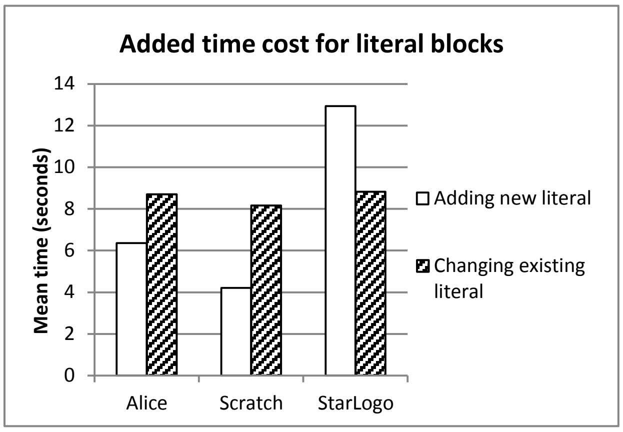

**Figure 3:** *Adding blocks to hold literals*

*5.2.2 Text vs. block-based literals*
Block languages differ from each other in their treatments of text and numeric literals. Though the program statements in these languages are shown as individual blocks, there is some difference between those that use “plug-in” value blocks in those statements, and those that have character-based text entry slots as parameters instead.
Scratch allows literals to be entered as text, from the keyboard. In StarLogo literals are added as separate blocks (Figure 4), and this makes StarLogo more viscous to use. Scratch allows plain text and drag-and-drop value blocks to be used together in the same expressions. In StarLogo, as noted, literals are created as special blocks, which are snapped together horizontally (whereas statements are arranged vertically) to create expressions. There is no direct way to enter the text without a block. Alice’s design approach is different still – while allowing drag-and-drop style blocks, it also uses selection from hierarchical menus. Thus, clicking on the empty space for a parameter opens a (large) context menu, from which the user can choose any possible values. This makes menus very large in programs with many variables, or when writing complex multi-nested expressions.
StarLogo’s block-only approach is conceptually consistent, but it increases task times for those tasks that require extra blocks for each literal. Once the block is in place, changing its value is less problematic. In Figure 3, the mean task times for

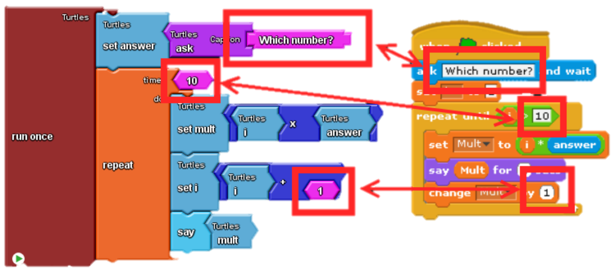

**Figure 4:** *Equivalent literal values in Scratch/StarLogo*

changing a literal, once its block has been added, are approximately the same for all three systems (since this involves entering text from the keyboard, as normal).

*5.2.3 Incidental to the interface*
There are some results that indicate a problem with the general UI of the editor, rather than a problem with the actual program notation. Some of the differences between Scratch and StarLogo TNG are like this; StarLogo, for example, uses a series of panels, on the left of the screen, which must be cycled through in a fixed order. There are panels for blocks that apply to the individual object, apply to that class of object, and that represent control statements.
A further example is Greenfoot’s cut and paste options. In most text editors (whether for programming or other domains) selected text can be dragged to another point in the file, effectively interpreted as a combined cut and paste operation. In the version of Greenfoot that was tested, this interaction is not present. Although ostensibly trivial, this increases the task time for Greenfoot tasks that rely on moving statements, or rearranging the order of different program parts.

## 5.3 Limitations of the predictive model

The modelling approach used here does not take into account the time a user might spend designing a program, or attempting to understand existing code. Programming is a more cognitively complex activity than most Internet browsing, for example and involves additional processes. We proceed on the basis that though this – the program design element of the task – means that, in practice, program-writing will take longer than the sum of the individual cognitive/ “mechanical” tasks, there is still value in comparing those aspects of tasks like-for-like in different systems.
It is obvious that the choice of tasks affects mean task times. If we were studying the overall effect that these HCI problems have on writing a whole

program, it would be important to weight the task groups appropriately – taking account of the proportion of their time the “real” programmer is likely to spend on different activities. Jadud (2006) observed such a user, and provides programmer workflows of the sort that could be used to determine the importance of different tasks.

## 6. CONCLUSIONS

This paper extends previous cognitive modelling work to compare different block-based and traditional text programming languages. The discussion has concentrated on the viscosity incurred when statements cling to each other during editing, and the ways in which parameters, particularly literals like numbers or text, are handled in the (semi-) visual notations. Comparisons of those systems that, superficially, **look** alike, have proved particularly useful. The differences between literals in Scratch and StarLogo TNG, for example, cause an observable effect in task completion time for the same sets of tasks. Mindstorms NXT, again, **looks** very different from the block-based languages, but its treatment of literals and Scratch’s are closer to each other than Scratch’s is to Alice or StarLogo (and this overall approach appears more usable). We are developing an editor that has some resemblance to block-based editors, and our findings suggest that a Scratch-style textbox approach would be preferable to the StarLogo horizontal blocks design.

Compared to Alice, Scratch and StarLogo are very viscous when editing existing statements. In this case, we believe that the Scratch/StarLogo “sticky” block design would be detrimental.

Overall, while some of these factors might be noticed through system use, we believe that this work is a first step towards quantifying them in a systematic way. Predictive models, based on their accuracy elsewhere, give us reason to believe that the trends, at least, would be similar in real user testing, and this is a step which we now intend to pursue.

## 7. REFERENCES

Card, S.K., Moran, T.P. & Newell, A. 1980, "The keystroke-level model for user performance time with interactive systems", *Communications of the ACM,* vol. 23, no. 7, pp. 396-410.

Cooper, S., Dann, W. & Pausch, R. 2003, "Teaching objects-first in introductory computer science", *ACM SIGCSE Bulletin,* vol. 35, no. 1, pp. 191-195.

Fitts, P.M. 1954, "The information capacity of the human motor system in controlling the amplitude of movement.", *Journal of experimental psychology,* vol. 47, no. 6, p. 381.

Green, T.R.G. 1989, "Cognitive dimensions of notations", *People and computers V: proceedings of the fifth conference of the British Computer Society Human-Computer Interaction Specialist Group*, Cambridge University Press, p. 443.

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.*

Henriksen, P. & Kölling, M. 2004, "Greenfoot: Combining object visualisation with interaction", *Conference on Object Oriented Programming Systems Languages and Applications,* ACM, New York, October 24-28 2004, p. 73.

Jadud, M.C. 2006, "Methods and tools for exploring novice compilation behaviour", *Proceedings of the second international workshop on Computing education research,* ACM, p. 73.

John, B.E. 2010, "Reducing the Variability between Novice Modelers: Results of a Tool for Human Performance Modeling Produced through Human-Centered Design", *Proceedings of the 19th Annual Conference on Behavior Representation in Modeling and Simulation*, p. 22.

John, B.E. et al. 2004, "Predictive human performance modeling made easy", *Proceedings of the SIGCHI conference on Human factors in computing systems,* ACM, p. 462.

Kelleher, C. & Pausch, R. 2005, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers", *ACM Computing Surveys,* vol. 37, no. 2, pp. 83-137.

Maloney, J. et al. 2004, "Scratch: A Sneak Preview", *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing,* IEEE Computer Society, p. 109.

McKay, F., 2012, "A Prototype Structured but Low-viscosity Editor for Novice Programmers", *Proceedings of BCS HCI 2012: People & Computers XXVI*, p. 363.

McKay, F., Kölling, M. 2012, "Evaluation of Subject-Specific Heuristics for Initial Learning Environments: A Pilot Study", *Proceedings of the 24th Psychology of Programming Interest Group (PPIG) annual conference.*

Newell, A. & Card, S.K. 1985, "The prospects for psychological science in human-computer interaction", *Human-Computer Interaction,* vol. 1, no. 3, pp. 209-242.