Heuristic Evaluation for Novice Programming Systems

MICHAEL KÖLLING, University of Kent
FRASER MCKAY, University of Kent

The last few years has seen a proliferation of novice programming tools. The availability of a large number of systems has made it difficult for many users to choose between them. Even for education researchers, comparing the relative quality of these tools, or judging their respective suitability for a given context, is hard in many instances. For designers of such systems, assessing the respective quality of competing design decisions can be equally difficult.

Heuristic evaluation provides a practical method of assessing the quality of alternatives in these situations, and of identifying potential problems with existing systems for a given target group or context. Existing sets of heuristics, however, are not specific to the domain of novice programming, and thus do not evaluate all aspects of interest to us in this specialised application domain.

In this paper, we propose a set of heuristics to be used in heuristic evaluations of novice programming systems. These heuristics have the potential to allow a useful assessment of the quality of a given system, with lower cost than full formal user studies and greater precision than the use of existing sets of heuristics. The heuristics are described and discussed in detail. We present an evaluation of the effectiveness of the heuristics that suggests that the new set of heuristics provides additional useful information to designers not obtained with existing heuristics sets.

• Human-centered computing~Heuristic evaluations • Social and professional topics~Computing education.

Additional Key Words and Phrases: HCI, heuristic evaluation, introductory programming tools

ACM Reference Format:

Michael Kölling, Fraser McKay, 2016. Heuristic Evaluation for Novice Programming Systems. *ACM Trans. Comput. Educ.* 16, 3, Article 12 (June 2016), 30 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. EVALUATING NOVICE PROGRAMMING TOOLS

Software tools specifically designed for programming education have a long tradition in our discipline. They take various forms, from languages to custom-made libraries to micro-worlds and fully fledged integrated development environments (IDEs).

In the domain of languages, various systems, including Logo, Basic and Pascal, were developed specifically for teaching and learning. In parallel, a small number of libraries and micro-worlds became popular. Throughout the 1970s and 80s, a few systems dominated the educational space. One of the most successful was Turtle Graphics – an abstraction first implemented for the Logo language [Papert 1980] and later ported and adapted to countless other languages [Caspersen and Christensen 2000; Python Software Foundation 2012; Slack 1990]. Turtle Graphics introduced the concept of a micro-world, in this case with a single actor (a turtle) and the ability to produce graphics.

Author's addresses: M. Kölling, University of Kent, School of Computing, Canterbury, Kent, CT2 7NF, UK; F. McKay, University of Kent, School of Computing, Canterbury, Kent, CT2 7NF, UK.

DOI:http://dx.doi.org/10.1145/0000000.0000000

In 1981, Karel the Robot expanded on this idea, introducing a micro-world in which a robot could be programmed to collect "beepers" [Pattis 1981]. Again, the system was ported, with little change, to many languages.

With the popularisation of object orientation in introductory teaching from the late 1990s, these tools were adapted to the new paradigm, and new teaching tools started to appear. At the turn of the century, however, the choice was still fairly limited. Teaching tools existed mostly in the form of these kinds of libraries; educational languages had taken a backseat to the rise of the use of industry-strength languages (such as C++, Java and Visual Basic) in education. Dedicated full educational development environments were few and far between.

Since then, however, the situation has dramatically changed. Within just a few years, a large number of educational environments were released, mostly supporting the object-oriented paradigm, and often incorporating complete IDEs highly specialised for rich media programming and interactive experimentation. An early example was Blue [Kölling 1999], published in 1995, followed by BlueJ [Kölling et al. 2003] and GameMaker [Overmars 2004], both systems published in 1999; Alice [Cooper et al. 2003], published in 2000; DrJava [Allen et al. 2002] published in 2002; Jeroo [Sanders and Dorn 2003] published in 2003; Scratch [Maloney et al. 2010] and Greenfoot [Kölling 2010], both published in 2006; StarLogo TNG [Begel and Klopfer 2007] from 2007 and Kodu [MacLaurin 2009], published in 2009.

The explosive proliferation of this type of educational system clearly indicates a belief – at least on the part of the developers – in the benefit that those systems can bring to the teaching of young beginners. Motivation is greatly increased through the use of interactive graphical systems, dealing with the complexity of the underlying language and system is made much easier, and programming principles are understood much better. So, at least, go the hypotheses.

One of the major problems, from a point of view of programming education research (with emphasis on the *research* part of this term) is that these hypotheses have rarely been appropriately tested. Belief in the benefit of these kinds of system is based much more on anecdotal folklore and individual experience than on formal studies that could withstand scientific scrutiny.

Initially, the lack of scientific validation, or at least somewhat more formal evaluation, did not pose a significant problem. With systems such as Logo and turtle graphics, subjective teacher and learner satisfaction was high enough to be convincing for a large number of instructors, and choosing to use such a system (or not) was a manageable question.

With the proliferation of competing systems, however, the problem has become more complicated. Not only should we ask the question whether such kinds of tools are helpful at all (which many instructors strongly believe them to be, even in the absence of hard evidence), but we need to decide which of a significant number of competing systems is "better" for a given task in a given context. Educators have to make choices, not only between using an educational IDE or not, but between a number of direct competitors.

At the same time, it is time that designers of such systems take the questions of suitability, usability and effectiveness more seriously. More and more schools and universities use educational IDEs, and seat-of-the-pants design methods are becoming increasingly hard to justify.

Studies evaluating the actual learning benefit of the use of a specific system are rare. This is not for lack of interest or realisation of the usefulness of such studies, but because they are difficult to conduct with a high degree of scientific reliability.

A common experimental study setup of effects of pedagogical interventions consists of a pre-test and post-test to measure learning effects and use of a control group taught with different tools to compare to relevant alternatives. The study would have to be conducted in a realistic setting (school, college or university class) with users of the actual user group (i.e. actual students in real learning situations) and compensate for other competing variables (such as different teachers, differences in prior knowledge, age, motivation, etc.).

Running the two groups (experiment group and control group) in parallel is usually difficult to resource: the teacher almost doubles the workload and has to avoid bias. It also introduces an ethical problem: If we expect one variant to be superior, and the setting is an actual examined part of a student's education, then we would knowingly disadvantage a group of students. However, if we run the two trials sequentially, it becomes very difficult to compensate for possible other factors influencing the outcome, such as difference in teachers or populations.

In short: Using comparative experiments to study educational effects of educational environments is often impractical, and other methods of study are required to fill the gap. A number other approaches are available, including empirical and qualitative studies and systematic, formal, or semi-formal evaluation following various methods. Of the alternatives, heuristic evaluation is attractive as an early, approximate method of analysis: while it does not reach the same level of reliability in assessing the quality of a complete system as some other methods of evaluation can do, it is relatively quick and easy to perform and – as a result – may be performed more often in practice than other types of study.

Heuristic evaluations can give a good indication of many of the problems or potential successes of a system, and they are able to collect some useful and reliable information about the quality of a system design. Heuristic evaluation will not allow us to draw conclusions about pedagogical benefits of a system, but it can help make reliable judgements about usability and suitability of a tool. In the trade-off between difficulty and usefulness of evaluative studies, heuristic evaluation advances to becoming an attractive alternative method available to us.

We advocate a significant increase in more formal or semi-formal evaluations of the quality of educational programming systems. With the increase in the number of systems on the market, it is becoming increasingly important to be able to make informed decisions, backed by formal argument, about the relative quality of these tools.

Heuristic evaluation is a method that is both useful and practical to make such an assessment of many aspects of educational programming systems. However, the state of the art of heuristic evaluation for this specific purpose can be improved.

The quality of insights derived from heuristic evaluation depends on two variables: The experience and background of the evaluators, and the quality of the heuristics themselves. This paper is concerned with the latter.

Some of the most frequently used sets of heuristics are application-area neutral. They specify goals and guidelines for software systems in general. However, results of evaluations can be improved for specific application areas if the set of heuristics is adapted specifically to the area under investigation. In that case, application-area specific requirements can be included in the heuristics, and deeper insights might be gained. This has been done for a variety of application areas (see our discussion in Section 2) with good success.

In this paper, we propose a set of heuristics specific to novice programming systems. This new set aims to improve evaluations of such systems by combining

general usability criteria with aspects relevant to our specific application domain, such as motivational and pedagogical effects. After discussing some background of heuristic evaluation and its variations we propose, describe and discuss the heuristics, followed by an evaluation of their effectiveness.

2. BACKGROUND

2.1 Heuristic evaluation

Heuristic evaluation was introduced as a "discount" method of analysing user interfaces without full user testing [Nielsen and Molich 1990]. When performing a heuristic evaluation, experts compare interfaces to sets of heuristics – general principles that describe aspects of an ideal system. Nielsen lists ten separate heuristics, formulated in the style of positive guidelines such as "*The system should always keep users informed about what is going on, through appropriate feedback within reasonable time*." [Nielsen 2005].

Heuristics can also be used in designing interfaces. In this case, designers pay conscious attention to the set of heuristics during the design process.

Nielsen's heuristics are intended to be generally applicable to a wide variety of software interfaces, but other authors have identified more specialised, domain-specific sets of heuristics. Pane and Myers [1996] defined a set of heuristics aimed at novice programming systems. These extend Nielsen's set with additional heuristics aimed at identifying issues with problems specific to this target domain.

Another area relevant to our work are heuristics for interaction in computer games. Malone [1980] described heuristics for predicting "fun" in games. These include notions of challenge (an important factor in one of our heuristics, described later). Malone's work has been further developed in the Structured Expert Evaluation Method (SEEM), which combines heuristics for both usability and fun. SEEM has subsequently been shown to be a useful way of predicting reactions to children's educational games [Bekker et al. 2008].

The research presented here, being concerned with programming, also conceptually overlaps with the Cognitive Dimensions framework [Green 1989]. The dimensions describe concepts relevant to cognition and Human-Computer Interaction (HCI) in programming notations, but – contrary to heuristics – are not phrased as instructions. The cognitive dimensions provide descriptions of concepts such as "viscosity" (resistance to code changes) and "secondary notations" (such as colour and spacing).

Heuristic evaluations rely heavily on human evaluators' findings, and these findings are driven by the set of heuristics provided. Therefore, it is critically important that the heuristics are valid in themselves. While the concept of "validity" is not usually well defined in terms of evaluating heuristics, it is generally taken to mean "shown to be useful in uncovering actual usability problems".

Pane and Myers's heuristics, for example, are argued from the literature of the time, but are not evaluated in practice. Nielsen and Molich's original heuristics were tested in user studies to refine them and to assess their validity [Nielsen and Molich 1990]. SEEM, based on Malone's heuristics, has also been validated experimentally. Hartson, Andre, and Williges [2001] present a method of comparing evaluation methods, including three standard metrics (thoroughness, validity and reliability) based on concepts named by Bastien and Scapin [1995] and further refined by Sears [1997]. At the end of this paper, we discuss an experiment performed based on these metrics to assess our own heuristics.

2.2 Novice programming tools

Novice programming tools take many different forms, and take up a large part of the CS education literature. For younger learners, tools such as Scratch, Alice and Greenfoot are among the major systems in wide-spread current use. However, the novice/learner spectrum extends to "older" tools as well. BlueJ, for example, is commonly used in universities. Mainstream programming languages, such as Java, are often used, as well as custom ones aimed at education (e.g. [de Raadt et al. 2002; Schulte and Bennedsen 2006]). BlueJ and Greenfoot, for example, both use standard Java, while Alice and Scratch use their own, custom designed languages.

Many programming tools share broadly similar interaction styles – programs are entered as streams of raw text from the keyboard, presented in a two-dimensional text layout on screen, and stored in a text file. Some systems mark up the text with "signalling" annotations (cues like colour, background and font), but the meaningful program specification consists purely of text. Many novice systems use the same text-based styles as standard industry languages, such as Java or C.

A popular alternative in early education are drag-and-drop block-based interfaces. Scratch, Alice and StarLogo TNG are examples of this kind of interaction. Visual blocks of code are stacked together like toy building bricks. This has the advantage of preventing syntax errors, but the disadvantage of being less flexible than traditional text. Program entry and manipulation is less efficient in these systems when writing long programs.

3. HEURISTICS FOR NOVICE PROGRAMMING SYSTEMS

3.1 Rationale

The need for a definition of new domain specific heuristics arose during an on-going effort to design a novel beginners' programming tool. To aid in the design of this new system, we initially applied existing heuristics as design guidelines. We also conducted evaluations using the cognitive dimensions framework, in their role as discussion tools, to study some existing models. We observed a number of different problems with these frameworks.

One drawback we encountered with Pane and Myers's heuristics is their length. Using them in depth, they produced very long – and quite cumbersome, thus less useful – evaluations. In total, Pane and Myers have 29 heuristics, grouped under eight of Nielsen's headings. Differences between the heuristics were not always clear-cut; there is duplication across some categories, and using them to categorise some of the problems is distinctly difficult. Some problems seemed to fit two categories equally well, and others did not neatly fit any. Yet, when using Nielsen's original heuristics, in order to avoid the length of Pane and Myers', the evaluation missed some crucial areas that we consider important for early learners' programming systems. While shorter, these heuristics do not achieve the same domain specific precision.

This experience led to the definition of a new set of heuristics with a number of specific goals of improvement. The goals were:

(1) *Manageable length*. The number of heuristics should be limited; it should be much closer to Nielsen's 10 rules than Pane and Myers' 29.
(2) *Domain specific focus*. The heuristics should cover aspects specific to novice programming.
(3) *Avoidance of redundancy and overlap*. As much as possible, problems should fit clearly into a single category.

(4) *Clarity of categorisation*. The categories should be clear to evaluators, cover all possible issues and be easily distinguished.

The new heuristics are neither sub- nor superset of any previously existing set, and have been developed from scratch using well established principles for this specific development domain.

3.2 Development Methodology

The definition of the new set of heuristics followed a systematic methodology, based on criteria defined to ensure usefulness and practicality of each heuristic, as well as the set as a whole. Once defined, the heuristics were tested by the authors, followed by tests with independent evaluators (described towards the end of this paper).

Given our experiences, we hypothesise that a smaller, more concise, more orthogonal set of heuristics is easier to use than a large one. No evidence has been presented in the literature about an ideal size of a heuristics set to be manageable. This has to be balanced with the goal of approaching completeness (finding as many faults as possible), which introduces a force towards larger sets.

The criteria for individual heuristics are:

— Each heuristic must be able to uniquely identify a set of actual known issues in existing systems from the target domain.

— Each heuristic must be sufficiently orthogonal to the remaining set to avoid ambiguity in classification of identified faults.

The criteria of the set as a whole are:

— The set of heuristics must be small enough to remain manageable.

— The set of heuristics must support identification of all major known problem areas found in software systems of the target domain.

These criteria for the new set of heuristics were derived from experiences both with the immediate evaluation of the software system under development, as well as many years of experience of system design in this target domain. Each criterion is the result of one or more specific problems encountered in actual practical experience with the use of existing heuristic sets in the application domain, several of them hinted at above. The first three criteria all address aspects of size, vagueness, overlap and lack of clarity of heuristic categories. Our experience has shown that, as heuristic sets grow in size, their usefulness suffers. Evaluators have increasing difficulty in assigning observed problems to specific heuristics as categories overlap or are unclear, leading to frustration and reduction of confidence of the evaluator. Well fitting heuristics not only make the evaluators life easier in categorising observed shortcomings, they also guide the observations of the tester: different heuristics lead to different observations. The result of being unsure of a fitting category leads to potential problems remaining unreported, while apparent overlap of category leads to lack of confidence and reduction of progress of the evaluator, and reports that are less clear than they should be lead to the danger of the designer dismissing a point because of miscategorisation.

The last criterion addresses the observation that generic (non-application specific) heuristics regularly overlook issues that are important to us, and that more specific heuristic sets could pick up.

Candidates for heuristics were developed in various ways: Where heuristics from existing sets proved useful and clear in practice, these were selected for inclusion. Where existing heuristics seemed to overlap, these were either merged or reworded to make them distinct. Heuristics from existing sets that were found not to be useful, either because they were too rarely needed, could be subsumed into an existing category without much loss of precision, or because they were unclear, were removed. Finally, issues we discovered through experience and informal evaluation of our own and other systems, and which we considered important in the design, but which were not addressed by any existing heuristic, were translated into new, additional heuristics.

After identifying these target candidates for heuristics, the authors evaluated the resulting set by applying it to existing systems from the target domain. This process resulted in observations about problems with the target heuristic set, including, at times, overlap of categories, unclear wording, and missing aspects. The heuristics were then adapted to address the problems.

This process was repeated over several iterations to refine and improve the set.

The systems used for systematic evaluation of the heuristics were Scratch, Alice, BlueJ, Greenfoot and Visual Basic. Partial evaluations using the new heuristics have also been carried out for Lego Mindstorms NXT, StarLogo TNG, Kodu, and C#. The first group of systems are the ones that were used for testing each iteration of the heuristics; the second group are systems that we have evaluated using the heuristics. We have also used the heuristics to evaluate new designs, which we are working on as part of a wider project to design a new system.

3.3 Context

The heuristics assume a context of the learner and learning situation that influences the content and focus of the heuristics themselves. It is assumed that the learner uses the system with the intention of learning to program (as opposed to systems that aim at quick achievement of specific implementation results without the need of deeper understanding). The learning goal takes precedence over the characteristics of the artefact under creation.

The learning situation may be a traditional, formal, teacher-led setting, or it may be a self-guided, informal learning process. The heuristics assume that a system may be used in both of these scenarios, and they evaluate characteristics accordingly.

It is assumed that the content at the core of the learning endeavour consists of traditional programming concepts.

3.4 The new heuristics

There are thirteen heuristics in our set. They are used to evaluate both the programming environments and programming language.

They are:

(1) **Engagement:** *The system should engage and motivate the intended audience of learners. It should stimulate learners' interest or sense of fun.*
(2) **Non-threatening:** *The system should not appear threatening in its appearance or behaviour. Users should feel safe in the knowledge that they can experiment without breaking the system, or losing data.*
(3) **Minimal language redundancy:** *The programming language should minimise redundancy in its language constructs and libraries.*

(4) **Learner-appropriate abstractions:** *The system should use abstractions that are at the appropriate level for the learner and task. Abstractions should be driven by pedagogy, not by the underlying machine.*
(5) **Consistency:** *The model, language and interface presentation should be consistent – internally, and with each other. Concepts used in the programming model should be represented in the system interface consistently.*
(6) **Visibility:** *The user should always be aware of system status and progress. It should be simple to navigate to parts of the system displaying other relevant data, such as other parts of a program under development.*
(7) **Secondary notations:** *The system should automatically provide secondary notations where this is helpful, and users should be allowed to add their own secondary notations where practical.*
(8) **Clarity:** *The presentation should maintain simplicity and clarity, avoiding visual distractions. This applies to the programming language and to other interface elements of the environment.*
(9) **Human-centric syntax:** *The program notation should use human-centric syntax. Syntactic elements should be easily readable, avoiding terminology obscure to the target audience.*
(10) **Edit-order freedom:** *The interface should allow the user freedom in the order they choose to work. Users should be able to leave tasks partially finished, and come back to them later.*
(11) **Minimal viscosity:** *The system should minimise viscosity in program entry and manipulation. Making common changes to program text should be as easy as possible.*
(12) **Error-avoidance:** *Preference should be given to preventing errors over reporting them. If the system can prevent, or work around an error, it should.*
(13) **Feedback:** *The system should provide timely and constructive feedback. The feedback should indicate the source of a problem and offer solutions.*

Below, we name and formulate each heuristic in turn, followed by a presentation and discussion of some relevant examples from existing systems and interfaces to illustrate their main aspects. Examples are drawn both from development environments and programming languages. As always in design work, there are no clear right and wrong answers, and some of the heuristics present competing forces that can contradict each other and must be weighed in specific design cases. These competing forces are discussed where appropriate.

1 Engagement

> *The system should engage and motivate the intended audience of learners. It should stimulate learners' interest or sense of fun.*

Aspects of the system that are designed to engage the learner should be pitched at the appropriate level – be that in terms of creativity, emotional/aesthetic appeal, subject/theme, imagination (if appropriate) and the degree of challenge and/or competitiveness against the computer or other learners. A system that is engaging to one type of user is not always going to be engaging to others. It is therefore critical that system designers have a clear idea of their target audience. Many systems aim to appeal to beginners' creative interests. Storytelling systems appeal to some groups of learners – Alice, and Storytelling Alice in particular [Kelleher and Pausch 2007], have shown success in engaging girls, fewer of whom have traditionally taken up programming. Systems like Greenfoot and Kodu have a focus on developing games.

Scratch programs have been used for storytelling [Burke and Kafai 2010], as well as games. There are a wide variety of systems, with different styles, and engaging to different groups. One study of out-of-school Scratch users involved asking them what school subject they would most associate Scratch with. Most related it to creative arts subjects, rather than "traditional" maths or computer science classes. Most of the users, when asked, had not made a connection between Scratch and computer programming. However, these users enjoyed being actively engaged in creating projects with Scratch [Maloney et al. 2008]. Thomas et al. [2003] refer to "code warriors" and "code-a-phobes" being present together in one class – individual personalities being a factor beyond age and educational setting. These students had different interests and motivations from each other. Real-world usefulness is also important for some students. Discovering that a language is used in one particular system or website, or in a "cool" company (such as Google, or in one case the *Star Wars* special-effects firm ILM), motivated some Greek high school pupils [Konidari and Louridas 2010].

Programming is a challenging activity; an engaging system should provide motivation for the beginner to overcome the challenges they are likely to face. "Classic" programming systems like Karel The Robot were based heavily around problem-solving tasks. Karel also featured a character for beginners to identify with.

2 Non-threatening

> *The system should not appear threatening in its appearance or behaviour. Users should feel safe in the knowledge that they can experiment without breaking the system, or losing data.*

The system should not be intimidating in its apparent size or complexity. It should use interactions that are familiar or natural, or that can be learned easily. It should be a safe place where mistakes do not have drastic consequences. The beginner might wonder, in a more threatening kind of system, whether their mistakes can be undone, or if they should be sure before they commit to anything. They might also spend too long concentrating on learning to navigate the environment, rather than thinking about code. The interaction found in most programming systems – typing plain text into a large editor (often without autocorrect options) – is not one that we would expect young beginners to be familiar with. Drag-and-drop interactions, on the other hand, are used in many other contexts. Similarly, the brick/block metaphor used in some systems has a real-world counterpart in construction toys or jigsaw puzzles. If an interaction style is not familiar, it should be easily learnable. Kodu [MacLaurin 2009] is unusual for using a game controller for program entry. On the Xbox 360, Kodu works in a familiar environment; it is treated by the console's interface in the same way as any other game, accessed from the same menus and having the same look and feel. It appears, for all intents and purposes, to be like any other "normal" game.

Secondly, the user should not have to fear making a mistake. Actions should be undoable, to facilitate easy exploration for a keen beginner. The novice should know that they are "safe" – that they cannot accidentally break either their own work or the computer. Microworlds, in particular, are usually sandboxed environments – a Karel program, for example, is unlikely to need network or (programmatic) file system access. Greenfoot users can upload their programs to the Greenfoot Gallery website, but these run with fewer permissions than a native program running on the user's machine. A sandboxed high-score mechanism is provided to abstract away details of file storage and network communication.

3 Minimal language redundancy

> *The programming language should minimise redundancy in its language constructs and libraries.*

The notional "ideal" beginners' programming language should have one way – and only one way – of achieving a particular task. The system should only include those features that it actually needs. A language for school children can (and should) be smaller than one for university students. In turn, a university-level development environment should still be smaller than an enterprise IDE.

Redundancy, which can be an advantage for experts since it adds flexibility, is usually a disadvantage for learners, who cannot choose between multiple, subtly different alternatives. Choosing which feature to use can be a distraction. "*Selection barriers"* – having to decide which, of many, features to use for part of the design – are a difficulty in programming [Ko et al. 2004]. If there is a pedagogical reason for having particular abstractions, to teach a data-structures course, for example, then the system designer might need to reconsider how the alternative features are presented – to at least make it clear what the differences are. .NET supports a large number of collection classes, arrays, generic and non-generic lists, sets, etc. Scratch, in the opposite extreme, represents all collections in one kind of list, with no arrays, sets, or other collection types. Many languages have a range of number types (signed/unsigned, single/double-precision, floating-point) that could represent the values 1, -13, or 3.14.

4 Learner-appropriate abstractions

> *The system should use abstractions that are at the appropriate level for the learner and task. Abstractions should be driven by pedagogy, not by the underlying machine.*

Mainstream programming languages have built up a range of reasonably common abstractions, which may include ones the novice programmer does not need to use. The abstractions a system uses should be selected carefully, based on the programmer's level of ability and the requirements for their programs.

Some abstractions closely match the programmer's mental model. Others represent lower levels of execution – they are useful for experienced programmers, but require the programmer to understand the details of the machine. Some languages' abstractions are pragmatic, balancing "ideal" design with facilitating transition to other languages. Many abstractions relate more to the language than the editing environment.

Pane [2002] worked with children to record natural-language pseudocode in order to identify abstractions novices use to reason about programming tasks. He found a clear preference for use of sets and collections over indexed arrays in mental models of young learners. The data structures were not assumed to be ordered. Built-in sorting operations were expected to be available. The novices did not refer to storage mechanisms, further supporting the idea of abstract sets rather than bounded arrays. Scratch, matching this model, has a single list structure, rather than a "machine-centric" array.

Pane also found that children expressed many of their pseudocode programs using objects. However, these children's programs were object-*based*, and did not make use of classes or inheritance. There are times when the absence of a class construct introduces viscosity; for instance, if changes to a logical kind of thing have to be made manually to each instance. Design difficulties can occur when choosing whether to use classes [Sanders and Thomas 2007; Thomasson et al. 2006], however, these could

be explained as *design* or *selection* barriers (formulating a high-level approach to solving a programming problem, and choosing which specific features to use to implement the solution, respectively) [Ko et al. 2004] that might be overcome.

Pane found a mix of imperative and declarative styles, with declarative statements used to "set up" the program, such as "There is a ball". Events were then used, to provide structure for imperative statements. That research found many (54%) of the children's programs were driven by events, more than any of the other styles that were identified. The child-oriented systems HANDS (Pane's own), Scratch and Alice implement simple types of event. Frameworks like .NET and Java support events, but there are often conceptual difficulties in teaching how they are implemented and used [Bruce et al. 2001; Milner 2010]. Therefore, teaching systems could make use of events, where domain-appropriate, but the Java-style "listener" metaphor may be too complex.

5 Consistency

> *The model, language and interface presentation should be consistent – internally, and with each other. Concepts used in the programming model should be represented in the system interface consistently.*

Consistency is included in most usability guidelines, for example in regards to user-interface terminology, expected behaviours, layout conventions, and so on [Nielsen 2005]. In addition to these areas, this heuristic applies to the programming model, libraries, the terminology used in the programming language, as well as the design of the environment.

Consistency should feature in how, and when, a code editor makes use of secondary notations (Heuristic 7) – text and background colour, typeface, and other graphical and typographical features. A common example is keyword highlighting. We would expect the colour scheme to be applied consistently, including consistency with the programming model. That is, highlighting should match conceptual abstractions rather than emphasise technical parsing detail.

Many object-oriented languages make a distinction between object types and primitive types. The reasons for this are technical; the distinction does not benefit users of a *novice* language. In Java, to compare two identical strings, one must use a *.equals()* method. To compare two integers, the programmer uses the == operator. There is a similar issue with switch/case statements, which in Java only handle integers (or enumerations) but not object types (an exception being made since Java 7 for strings, introducing another inconsistency). The distinction adds another rule the novice has to learn.

In many languages, statements are separated with a semicolon. However, this is not consistently applied for *all* statements (for, if, for example, in C-type languages). This forces a novice to learn (and remember) another exception to the usual rules of the language.

6 Visibility

> *The user should always be aware of system status and progress. It should be simple to navigate to parts of the system displaying other relevant data, such as other parts of a program under development.*

As with consistency, keeping the user informed – or at least, not hiding status information where it will be hard to find – is a component in several wider usability guides [Nielsen 2005]. In the domain of programming languages, the Cognitive Dimensions refer to "dependencies" [Green 1989]: sections of code often refer to

others, with multiple complex relationships between segments spread-out throughout the program. In these situations, it can be difficult to keep track of all these separate statuses, and it is likely that they will not all be visible at the same time. Hidden dependencies are those which the programmer is not made aware of by the system. Poor visibility can result in errors – either because programmers do not know something that they should (as with hidden dependencies), or because they are asked to remember too much at a time, rather than being assisted with appropriate display of relevant information.

Hidden dependencies should be avoided. Related pieces of code should be kept close together (or linked by navigation aids, if not physically close), so that they can be viewed together, or accessed with minimal effort. A programmer should be able to navigate their code logically, not just by linear scrolling.

Where dependant code is not easily visible, changing one piece might have knock-on effects on its non-local dependants [Green and Petre 1996]. One example is C/C++'s use of separate header files, with forward declarations that are not visible at the same time as their corresponding definitions. C# and Visual Basic also allow "partial" classes to be split across more than one file (but still be treated as one class). These mechanisms can make it more difficult to find related code. By contrast, Java requires all of a class's code to be put into one file. In Scratch, stacks belonging to the same sprite are visible side-by-side, and can be rearranged on the screen.

Navigation is supported by text editors to varying degrees of sophistication, from basic search support to a variety of more elaborate techniques. Related pieces of code can be linked, allowing quick navigation from one location to a relevant other. Visual cues – either graphical, or through structuring of the text – can be used as markers to help recognizing relevant code. Display techniques include the use of fisheye, bifocal, or "focus+" technology [Jakobsen and Hornbæk 2006] to emphasise a segment on screen while retaining navigational visibility that places the code in context. These involve a display with two areas, one of which is the normal editing space, and another, a zoomed-out view that shows the whole document, with an indication of the part the user is currently focussed on. The scroll bar is a common graphical user interface (GUI) control, and it usually indicates where in the current file the user is working. In a "fisheye" display, this could be replaced with a sidebar, containing a scrollable thumbnail image. This shows an overview of the document's content, in addition to the usual position marker. This is already seen in other domains (Adobe Reader, for example), and is used in the BlueJ and Greenfoot code editors. We are not aware of it being common in other IDEs.

Another technique commonly used to support this goal is the provision of navigation functionality to move from the use of an identifier to its definition (or *vice versa*), independent of the locality of the target location.

However, though good visibility is necessary, the way in which it is incorporated must be balanced with simplicity, avoiding cluttering an interface and making it harder to use (Heuristic 8).

7 Secondary notations

> *The system should automatically provide secondary notations where this is helpful, and users should be allowed to add their own secondary notations where practical.*

Secondary notations are auxiliary channels of information that are embedded in the main program – either as an optional part of the primary notation, as written by the programmer (whitespace, comments, layout, etc.), or by the system as part of the

presentation (colour, size, typeface). They do not affect the computer's understanding of the program, but affect the ability to read the program text for the human programmer. Secondary notations generated by the system are sometimes called "signalling" [Pane and Myers 1996]. Signalling includes techniques such as auto-indentation [Miara et al. 1983], colour [Rambally 1986], shape (in visual languages), grouping, typeface and text size. Programmer-generated notations include comments, whitespace, naming conventions, and layout on the page [Green and Petre 1996; Pane and Myers 1996]. Media-rich comments have sometimes been added; for example, images, videos and interactive animations that explain the program [Ko and Myers 2006].

There are varying schemes regarding time and place of appropriate signalling. In the novice context, highlighting should be applied to code that is conceptually important to the student's understanding, not to language keywords distinguished by their token-type in the parser [Pane and Myers 1996]. The programmer should know why one piece is highlighted and another is not – they should *know* that it is not arbitrary. There should also be a clear distinction between the primary and secondary notations. Pane and Myers [1996] also highlight the risk of novices assuming that the secondary notation affects the computer's understanding – there might be a danger of the novice thinking that the computer will infer what to do with a variable, for example, because it has been given a meaningful name (rather than some arbitrary characters). It is also important that secondary notation does not interfere with the visual aspects of readability and clarity (see Heuristic 8).

Programmer-generated secondary notations should be allowed were possible, if they are of no consequence to the computer. Even if a naming convention is desired, it seems counterproductive to force a "hard" error message if a legal (though inadvisable) name is entered. This is a pedagogical issue, but not necessarily one that has to result in a catastrophic error message from the compiler. Scratch makes heavy use of secondary notations with its bold background colours. However, Scratch also, effectively, blocks the use of whitespace within its scripts as a secondary notation. Although individual stacks (groups of statements) can be laid out freely on a page, there is no whitespace within a stack. This would make it difficult to leave space around one particularly important line, for emphasis, for example, or difficult to leave a temporary space where more code is to be added in future. The ability to freely position complete stacks in relation to each other, however, provides an important opportunity for programmer-generated secondary notation.

8 Clarity

> *The presentation should maintain simplicity and clarity, avoiding visual distractions. This applies to the programming language and to other interface elements of the environment.*

This heuristic asks whether there is too much on the screen at once, whether there are too many colours or interface elements, and how well different visual artefacts can be told apart. Visual stress should be avoided, though this must be balanced, in each case, with the need for visibility. This does not always mean that less visible information is preferable, but designers must consider *how* they present things.

Blackwell [1996] addresses the assumption that pictures and diagrams are inherently easier to understand (that visual programs were easier to read than textual ones), observing that there is no actual evidence to support this, and that the assumed verbal/visual distinction is not a correct application of the relevant

psychology. Visual notations should not be *assumed* to be clearer just because they are visual.

The clarity of any interface can also be discussed in terms of visual "feature channels" [Ware 2008]: these include the elements of size, shape, texture, colour, alignment, sharpness and motion. If there is too much variation across too many channels, the design becomes more visually stressful to process. If there is not enough, however, it becomes difficult to "pick out" differences. For example, a colour coding system is only useful if the colours are sufficiently different from each other that they can be told apart. Colour systems are used, of course, in many secondary notation schemes. Literature summarised by Ware [2008] suggests that no more than twelve values can be usefully differentiated from each other in any colour-based system. The most distinctive colours are those at the opposite ends of the three colour channels: black and white, red and green, and yellow and blue. Scratch, as a highly-graphically marked-up language with strong colours shows that bold colours work well to emphasise elements of a small program with a few statements. However, as the program gets longer, what was helpful information can turn into visual noise on the screen, and the strong colours can introduce a distraction.

To reduce use of screen space, many large IDEs support code-folding, where small sections of code can be minimised individually. However, this must be done carefully, so that information visibility is not impeded.

Visual languages may also be less accessible, or not accessible at all, to screen-reading or Braille software – a relatively fundamental accessibility problem [Siegfried 2006]. Editors that appear, primarily, to be text-based can also have visual features that are not accessible – GUI editors and class diagrams, for example. There are various ways to overcome this, but the system must implement existing standards to allow itself to be read by accessibility software.

9 Human-centric syntax

> *The program notation should use human-centric syntax. Syntactic elements should be easily readable, avoiding terminology obscure to the target audience.*

Programming language keywords should be clear, simple, and easy to understand. Nielsen's heuristics encourage us to *"speak the users' language"* [Nielsen and Molich 1990]. Additionally, symbols and punctuation should not be tedious to manipulate, should be readable rather than confusing, and should exist for the benefit of the programmer rather than the machine. This heuristic may lead to favouring keywords over symbols for programming languages.

Constructivist theory tells us that novices bring their own understandings with them when they approach programming for the first time [Ben-Ari 1998]. It is beneficial if their preconceptions do not interfere with the learning of keywords that have existing meaning in natural language. Compatibility with problematic keywords should not be kept for solely historical or conventional reasons. Keywords such as "final", "static", "void" and "explicit" are obscure in everyday language. "Print" is redefined in programming in that it usually writes something to the screen, rather than to paper. Scratch uses relatively simple words – "repeat" rather than "for", which is based on *doing a thing for each value of a given counter*, and is not an obvious word choice for a non-programmer. Pane [2002] also suggests replacements for particular keywords – that "afterwards" might be less ambiguous than "then", and "otherwise" less than "else".

It is important, however, to emphasise that the decision for a specific syntax is highly dependent on the target group, and that no single right solution exists for all types of programming. Many keywords – "afterwards", for example – are longer than their equivalents in more professionally oriented languages, such as C. Professionals possess a different, richer vocabulary, and the use of concise symbols may be entirely appropriate. Short syntax forms and operators like "++" take less time to code, and to read, and they are usable once they have become part of one's language. For beginners, a more meaningful name is often required, but for experts, short names might be the most appropriate design choice.

Some languages rely on "magic" phrases – "public static void main," for example – that have to be memorised and entered verbatim at a particular time or place. Ideally, that formula would not be the first thing that we would teach in Java, but it is part of every program. A format like Pascal's program...begin...end at least uses more apparently-meaningful words. Some languages have removed these completely, and some more-abstracted editors can hide or auto-generate them. BlueJ, Greenfoot, Alice and Scratch all hide these kinds of statement.

Novices [Denny et al. 2011; Konidari and Louridas 2010; Robins et al. 2006] and more experienced programmers alike [Ko et al. 2005], can frequently introduce errors when manipulating delimiter characters like brackets or semicolons. Some languages separate statements with semicolons – something that really only exists in the world of programming. Other text languages – such as Python and Visual Basic – take a more human-language approach of using line breaks instead.

10 Edit-order freedom

> *The interface should allow the user freedom in the order they choose to work. Users should be able to leave tasks partially finished, and come back to them later.*

Green describes edit-order flexibility as "decoupl[ing] the generative order from the final text order" [Green 1989]. That is, the order in which sections are written should not reflect where they have to fit in the final program. The user should be free to write the program in whichever order they like – they should not have to complete one task before moving on to another, and they should not have to complete them in some fixed sequence. It should be possible to leave a fragment unfinished, as a placeholder, stub or draft implementation. It should be easy to change one's mind after decisions have been made, especially without having to lose any changes made since then.

New programmers can be classed as either "planners" or "tinkerers" (also called "bricoleurs" [Turkle and Papert 1992]). The second group do not work to any particular sequence, jumping about and experimenting by adding to the program in chunks [Berland and Martin 2011]. Therefore, while planning support is important, for those students who depend on it, the system must also accommodate programmers who do not stick to a plan. It must be possible to tinker, or to change the plan. In an object-oriented program, behaviour that is implemented in a class does not need to be specified for each individual object of that type. However, in instance-based systems where this is not available, as in earlier versions of Scratch, it can take foresight to create a well-designed "master" object that can be statically copied-and-pasted to be used more than once. Without this careful foresight, changes have to be made individually to every copy of that object. This is an example where, because late changes are difficult, there has to be a degree of certainty before committing to copying and re-using a particular piece of code. Tinkering becomes

difficult. (This difficulty was recognised as a problem by the Scratch developers and addressed with the provision of a programmatic copy operation in Scratch 2.) If affected objects are referred to elsewhere in the program, it may also force the programmer's hand in planning which part has to be written before another – an example of the abstractions discussed in Heuristic 4 affecting this writing flexibility.

Jadud [2006] observed students moving back and forth between different parts of a program when they encountered an error. They left behind incomplete sections. Birnbaum and Goldman [2005] show that a program should not need to be in a run-ready state after every individual edit. This has implications for some error-prevention constraints used in strict syntax-directed editors (see Heuristic 12). Placeholders should be marked, but this can be done somewhat passively. It could be achieved through highlighting, or an open space or slot (as in Scratch), or a metaphor from outside programming, such as the red underline used to mark spelling mistakes commonly seen in word processors [Birnbaum and Goldman 2005; Ko and Myers 2006]. Some systems make it difficult to leave code half-written when outlining. Alice 2, for example, requires that a condition be entered immediately when adding an if statement. If the condition has not yet been set up, a dummy value has to be entered even if it is not sensible, to satisfy the system constraints. This is an example of error-prevention constraints forcing the programmer to follow a specific process. The programmer will have to remember to change this later. Because this is forced to be *syntactically* correct, there is no leftover marker to remind them that this is incomplete.

11 Minimal viscosity

> *The system should minimise viscosity in program entry and manipulation. Making common changes to program text should be as easy as possible.*

Viscosity is a measure of local resistance to change [Green and Petre 1996]. Examples of viscosity in relevant systems are 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 statement or group of methods.

Knock-on viscosity is incurred when a "primary" change has side-effects for other parts of the program. Pane and Myers [1996] discuss BASIC line numbers as an example. However, modern systems do not tend to use these. Knock-on viscosity could occur when an object or method is renamed – every occurrence of the name might have to be updated. Of the systems discussed in these heuristics, the larger editing environments are more likely to employ refactoring functionality for these knock-on changes, either automatically, or through a menu prompt. Another example is that of renaming a class in Java – a language that requires the filename to be the same as the class name. In addition to the effort of changing the class name every time it is referenced in code, more viscosity is caused by having to rename the text file accordingly.

As mentioned above, viscosity is more important for operations that are more common in a given system. The kinds of operations that are counted as common may vary in different kinds of system, and judging the importance of non-viscous aspects requires a judgement about frequency and importance of the operation affected.

12 Error-avoidance

> *Preference should be given to preventing errors over reporting them. If the system can prevent, or work around an error, it should.*

Some program errors are relatively easy to prevent. The system should try to prevent any errors that can easily be avoided – trivial, largely syntax-related errors are a distraction from the broader concepts a new programmer has to learn. Firstly, the system can use various editing constraints to reduce the likelihood of syntax errors. Where these fail, or are not appropriate, the system can attempt to make use of some reasonable default value, or behaviour. One attractive approach in some systems, which aim to emphasise tinkering more than correctness, is a "fail-soft" approach that involves finding a way around an error, and continuing.

The engineering principle of "poka-yoke" (Japanese, "mistake proofing") (Shingō, 1986) creates a design that can only be manipulated in a certain, always-correct, way. It is simply not possible to perform any "incorrect" action, because there is never any opportunity to do it. Maloney et al. [2010] observe that a child playing with plastic Lego bricks will not encounter error messages – either the blocks fit together, or they don't. Extending that analogy now, a Lego house could be said to be "syntactically" valid if the blocks fit together, but it may, or may not, be a well-*designed* house. A program may be syntactically correct and still do (from the programmer's perspective) the logically "wrong" – or an unexpected – thing. Poka-yoke constraints prevent one important class of errors – syntax errors – that novices often struggle with.

Drag-and-drop block interfaces have been one attempt to implement this for software development. Scratch, Alice and Star Logo TNG all have similar block-based syntaxes, where only syntactically correct blocks can be put together. In the 1980s, various attempts at designing syntax-directed editors had the same goal for text based languages. Most never became very popular, in part because this goal conflicted strongly with Heuristic 10, enforcing an edit order that felt restrictive to many users.

At times it might be appropriate to make an educated guess about the programmer's intent. Cooper, Reimann, and Cronin [2007] refer to errors where the system claims to know what is wrong, where, and why, and then tells the user this without attempting to work around the problem, or simply offering to solve it. With messages such as "; expected" (in Java, for example), it might be reasonable to infer what it is the programmer meant, or at least to ask "would you like me to fix this?" Another example originates in type systems. Implicit conversion from a short integer to a long one, or from an integer to a floating-point, would be reasonable inference in some scenarios. Auto-correcting, for some errors, and advisory "warnings" (as in Visual Basic and C#) might be used to prevent hard errors. Office packages are usually expected to use auto-correct, as are major search engines. Working around a simple error does not need to encourage lazy programming practices – thinking the computer will pick up the slack – and it can still emphasise the fact that a segment has been autocorrected.

In preventing errors, there is also a danger that either the constraints become too tight (a "dead" keyboard that does not react at all when invalid keys are pressed), or that the system allows anything to be written, but then ignores it. Constraint must be balanced with editing flexibility (Heuristic 10, the ability to "dip in" to part of the program) and local viscosity. The level of constraint clearly depends on the user; a more experienced programmer might feel more comfortable having enough latitude to do something error-*prone*, that is not recommended, but that would not be certain to fail. Entirely removing access to a certain threaded operation, for example, might prevent a common error, but there might be times where it is perfectly appropriate to carefully use threaded code. There are clearly different levels of constraint

appropriate for a university C++ student than for a beginning Scratch or StarLogo programmer.

Constraints should be soft; they should guide the programmer in their writing, but they should not force repeated "hard" modal error messages on the user when they get something wrong. Scratch is notable for ignoring ambiguities in incomplete code, such as an empty condition. In the first instance, it tries to insert a reasonable default (which is visible in the code). If this is not appropriate, it carries on and ignores the incomplete code. Though this means that there are no "hard" errors, it does mean that there is no obvious solution when the error is logical, and the program is auto-corrected to by syntactically valid but might behave unexpectedly.

13 Feedback

> *The system should provide timely and constructive feedback. The feedback should indicate the source of a problem and offer solutions.*

Two key elements of feedback are *demand* and *helpfulness*. Feedback should exist to help the programmer. Feedback should be given as close to the source event as it can be – in terms of both where and when it is shown. These considerations apply equally to feedback originating from the environment and editing process, and feedback from the compiler and runtime system. Novices frequently review their programs, so it should be available at least as frequently as the programmer wants it [Green and Petre 1996; Jadud 2006]. When feedback is requested, the system's response should be appropriately quick.

Continuous compilation is generally regarded as very useful, analysing program text continuously in the background and offering feedback early and quickly. An extension of this idea is continuous execution, which some systems provide in an "immediate window" or sandbox. Hundhausen and Brown [2007] however, argue against such a feature. BlueJ, Greenfoot and Scratch let the user invoke individual segments of code directly, providing opportunities for earlier feedback through easier – and thus potentially more frequent – testing.

The second element of feedback is helpfulness. Feedback should be specific about *what* the problem is, *where* it is, and if possible, how to fix it. Unhelpful error messages are a particular problem for novices [Ko et al. 2004]. Working in Java, novices have been observed to often make repeated small changes, suggesting they may not understand how to appropriately respond to an error message [Jadud 2006]. Messages such as "identifier expected", or "class, interface or enum expected" have been shown to be problematic, as they require prior knowledge. Novices have problems deciphering their meaning. "Symbol not found" is clear only once the programmer knows what a "symbol" actually is. In the context of DrRacket, a beginners' environment for programming in Scheme, Marceau, Fisler, and Krishnamurthi [2011] have analyzed categories of errors that students struggle with and error messages that are frequently not understood by students. They come to the same conclusion: Error messages do not help, even if they are timely and technically correct, if they are not understood by the users of the system. Failure to understand error messages is a common occurrence in beginners' programming system. Weinberg [1998] laments, "*how truly sad it is that just at the very moment when the computer has something important to tell us, it starts speaking gibberish*".

Scratch differs from these systems in that it does not show error messages. Instead, it uses a fail-soft policy (Heuristic 12). However, this behaviour is not unproblematic either: The automatic and silent modification of the user's code to circumvent an error may result in the program exhibiting unexpected behaviour that

beginners then may struggle to understand. The absence of any messages in this case can make the location of the problem more difficult to determine.

4. EVALUATION

It is hoped that use of the new set of heuristics provides better insight into the quality of novice programming tools than use of previously existing heuristics sets would yield. To test this hypothesis, the heuristics themselves must be evaluated.

Sears [1997] defines the concepts of *thoroughness*, *validity* and *reliability* for evaluating inspection methods. Similar concepts are reviewed by Hartson et al. [2001]. Thoroughness is a measure of how many of a given set of "reference problems" a method finds. Validity is a measure of how many of the candidate problems found are "real" problems (as opposed to false positives). Reliability measures consistency in the number of (correct) problems found when different evaluators apply the method.

Our method of evaluating these heuristics consists of two stages:

(1) In a first stage, the heuristics are applied in the evaluation of three different educational programming systems by the authors themselves.
(2) The second stage consists of the evaluation of the heuristics using a group of independent evaluators on a single system.

Both of these evaluations are described in this section.

The first stage is designed to test the hypothesis of increased effectiveness of the new heuristics set. Of the effectiveness measures identified – thoroughness, validity, and reliability – it is mainly thoroughness that will be assessed with this experiment design. The evaluation of different systems by the authors, and comparison to a known set of reference problems, can give a good indication of the number and kind of problem that the set can potentially identify. Measures of validity are likely to be too biased in this design to yield useful results, and reliability cannot be measured with this experiment. However, another goal of the heuristics set design was *practicality*, as expressed in ease of dealing with the set (and leading to the goal of *orthogonality* of the heuristics as the measurable indicator). Orthogonality can be measured with this stage of the experiment.

The second stage of the evaluation is designed to reduce the bias introduced by the authors being involved in the evaluation. It extends the results from the first stage by being able to measure thoroughness not only per evaluator, but also for a set as a whole, across a number of individual evaluators, therefore leading to more reliable results. Validity can then also be measured. Orthogonality can be evaluated in this stage via interviews, which can also yield additional information about other aspects of practicality.

4.1 Evaluation Stage 1: Three Systems

A first evaluation of the heuristics by applying them to a number of known systems with a known set of reference problems can give valuable first insights into the suitability and effectiveness of the heuristics, and help direct the refinement of the heuristics themselves. It is also useful in the preparation of the study in stage 2. Thoroughness and – to a lesser extent – validity can be investigated this way. The set of reference problems is known from various sources, including earlier evaluations of the same systems using various different sets of heuristics, which were carried out in early stages of this research to test and compare existing heuristic sets. More detail

is given below. Risks and limitations resulting from the authors' own involvement in performing the evaluation are stated below.

Method

For this stage of the evaluation, we collected and recorded interaction problems in three systems: Greenfoot (Java), Scratch, and Visual Basic. The aggregate problem sets were derived from the literature, from our experience of using these systems, from predictive cognitive modelling (discussed in a separate paper [McKay 2012]), and from the heuristic evaluations themselves. We recorded 65 possible problems in Greenfoot (many in the Java language), 58 in Scratch, and 57 in Visual Basic. They are representative of the kinds of problems that we would expect these heuristics to find.

We had already conducted a Nielsen-type evaluation before the new heuristics were developed, and then a further evaluation based on Pane and Myers's extension of Nielsen's set. These problem sets therefore predate our new heuristics. As noted in Section 3, difficulties using those sets inspired the new heuristics' creation. We applied the new heuristics, and then added new problems found by them to the reference set. To mitigate for the fact that the new heuristics came last (and therefore, we already knew of problems that we had uncovered with the other sets), we took several passes through the evaluation reports, attempting to match the newest problems to the existing sets. We attempted to show that problems *could* be findable with Nielsen's and Pane and Myers's sets, even if we did not find them on our first pass.

In Section 3, we stated that, as far as possible, we aimed for the heuristics to be orthogonal, and that the set, as a whole, should cover all of the problem areas. We therefore measured thoroughness (t), defined as the number of problems found, divided by the total number of problems that we know to exist. We also counted problems that indicated the heuristics were not orthogonal – that is, that the problem overlapped one or more heuristics. For example, Java naming conventions seem to be covered in more than one of Pane and Myers's categories (signalling, misleading appearances, and secondary notations). We measured the proportion of problems that were not ambiguous, and refer to that measure here as orthogonality (o).

Results

The problem counts for Nielsen's set, Pane and Myers's, and the new heuristics, are given in Table 1. The results of this experiment show that we could match more problems to the new set than to the previous two (a total of 176 compared to 136 for Nielsen and 134 for Pane and Myer). The new set is also more orthogonal, with significantly more problems fitting clearly into one category. Thoroughness and orthogonality are recorded in Figures 1 and 2, which show the individual 't' and 'o' values for each system, as well as the overall totals.

Table 1: Number of problems identified with different sets of heuristics (Nielsen, Pane and New) in three different systems (Greenfoot, Scratch, Visual Basic). (In parentheses: number of problems that were ambiguous, i.e. could not clearly be assigned to one single heuristic.)

	Nielsen	Pane	New
Greenfoot/Java (out of 65)	52 (20)	50 (23)	62 (1)
Scratch (out of 58)	39 (17)	37 (10)	57 (0)
Visual Basic (out of 57)	45 (10)	47 (23)	57 (1)

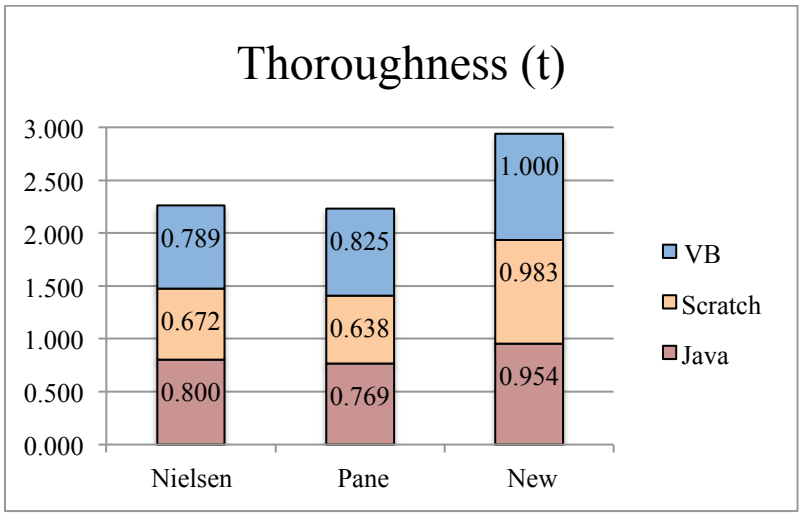

Figure 1: Thoroughness of different sets (larger is better)

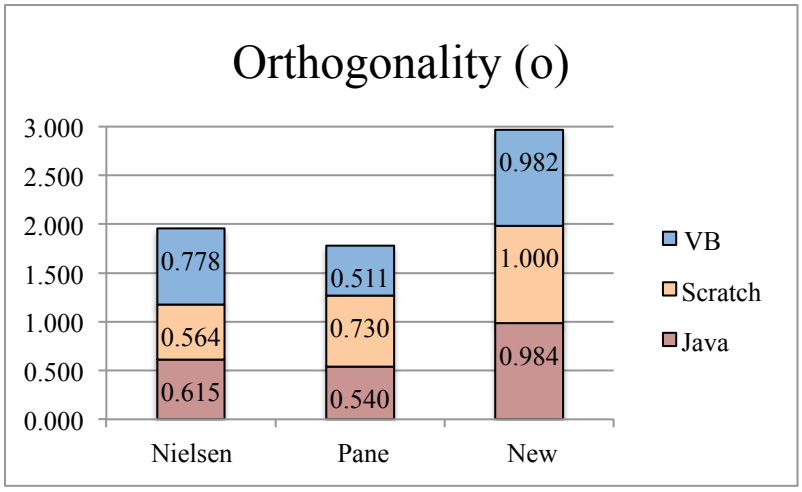

Figure 2. Orthogonality of different sets (larger is better)

Surprisingly – given that Pane and Myers's set is presumed to be a domain-specific improvement on Nielsen's – the evaluations using Pane and Myers's set seem marginally less effective than those using Nielsen's heuristics. However, the difference is too small to present a useful result. It may be that the larger number of heuristics in Pane and Myers's set makes overlap more likely.

Limitations

An evaluation of the heuristics by the original authors is most likely biased, and therefore does not provide conclusive evidence of the effectiveness of the heuristics in general use by independent evaluators. The experiment measured mostly thoroughness and orthogonality. Validity could theoretically be measured, but is most likely significantly biased by the authors' own involvement and the prior knowledge of the reference set. It is clearly less likely to find false positives in this study design, and therefore this study should not be taken as a useful measurement of validity. (Thus, a measurement of validity has not been included in the results.)

Reliability cannot be measured with this experiment.

To ameliorate these limitations, a second study was performed.

4.2 Evaluation Stage 2: Multiple Evaluators

The second phase of evaluating the new heuristics consisted of an experiment where several independent evaluators applied the heuristics to the same software system.[1] Some evaluators used our new heuristics, while a control group used an existing set, allowing us to compare the effectiveness of the new set relative to the existing one.

Method

The participant group consisted of undergraduate and postgraduate students from the School of Computing at our university. They were recruited via an email to undergraduate students who had completed an HCI module (which included study of and exercises with heuristic evaluation), and a second email to all postgraduate students. 13 students in total participated in this study (five undergraduate and eight postgraduate). The students volunteered their time; the activity was not linked to any study.

All participants had significant programming experience, and all had encountered heuristic evaluation previously.

Participants were asked to perform a heuristic evaluation of Scratch, using a given set of heuristics. All of the participants had heard of Scratch, however, only one participant had experience in using Scratch.

A set of tasks was handed out, together with a set of the heuristics, and participants were asked to evaluate the system and record any usability problems they encountered. About half of the evaluators (N=6) were given Nielsen's heuristics, while the other half (N=7) used the new heuristics.

Participants were not told that the purpose of the study was to evaluate the heuristics themselves. They were not aware that other participants might use a different set of heuristics. Participants were given a short "refresher" talk on the heuristic evaluation process as part of their instructions. Between one and three participants at a time worked independently, in different parts of a large, quiet room. A copy of the task and a list of the heuristics to use were made available on paper. The new heuristics were presented as above (the short, summary form at the beginning of section 3.3), and the existing set was taken from Nielsen [Nielsen 2005]. Both sets were formatted in the same plain style, were similar in length and level of detail provided. Participants were asked to record usability problems on paper pre-printed with columns for recording the problem and the heuristic violated by the problem. We did not disclose until afterwards that the purpose of the study was to compare two sets of heuristics.

At the end of the sessions, seven participants who had time (three using the new set, four using Nielsen's set) participated in short, informal individual interviews. The subject of the interview were the heuristics themselves, rather than the system under evaluation. The participants were asked open questions about ease or difficulty to understand and apply the heuristics; how they would describe the

[1] A subset of these results was reported by the authors in a workshop of the Psychology of Programming Interest Group as *McKay, F. and Kölling, M.: Evaluation of Subject-Specific Heuristics for Initial Learning Environments: A Pilot Study, PPIG, London, 2012*. The results presented here include and extend the previous data.

heuristics in their own words; and whether they thought any of the problems they found would have fitted with none of the heuristics, or more than one.

Table 2 summarises the details of evaluators in this study.

Table 2: Reviewer profiles

Reviewer	Study stage	Scratch experience	Heuristics	Interview
A	Undergraduate	No	New	Yes
B	Undergraduate	No	New	Yes
C	Postgraduate	No	New	No
D	Postgraduate	No	New	Yes
E	Postgraduate	No	New	No
F	Postgraduate	No	New	No
G	Undergraduate	No	New	No
H	Postgraduate	No	Nielsen	Yes
I	Postgraduate	No	Nielsen	Yes
J	Undergraduate	Yes	Nielsen	Yes
K	Undergraduate	No	Nielsen	No
L	Postgraduate	No	Nielsen	No
M	Postgraduate	No	Nielsen	Yes

Results, Part 1: Number of Problems Found

The number of problems found by each group showed no statistically significant difference.

Figure 3 is a diagrammatic representation of problems found per user. In the figure, every column represents a usability problem, and every row represents one reviewer. If that reviewer found the particular problem, the grid is marked with a black square. The first seven evaluators used the new heuristics, and the last six used Nielsen's set (separated by a horizontal line). Similar diagrams are used in Nielsen's online material [Nielsen 2005b]. Problems to the left of the vertical centre-line were found by more than one evaluator. Problems on the right were only found by one person. Problems are numbered (horizontal, bottom).

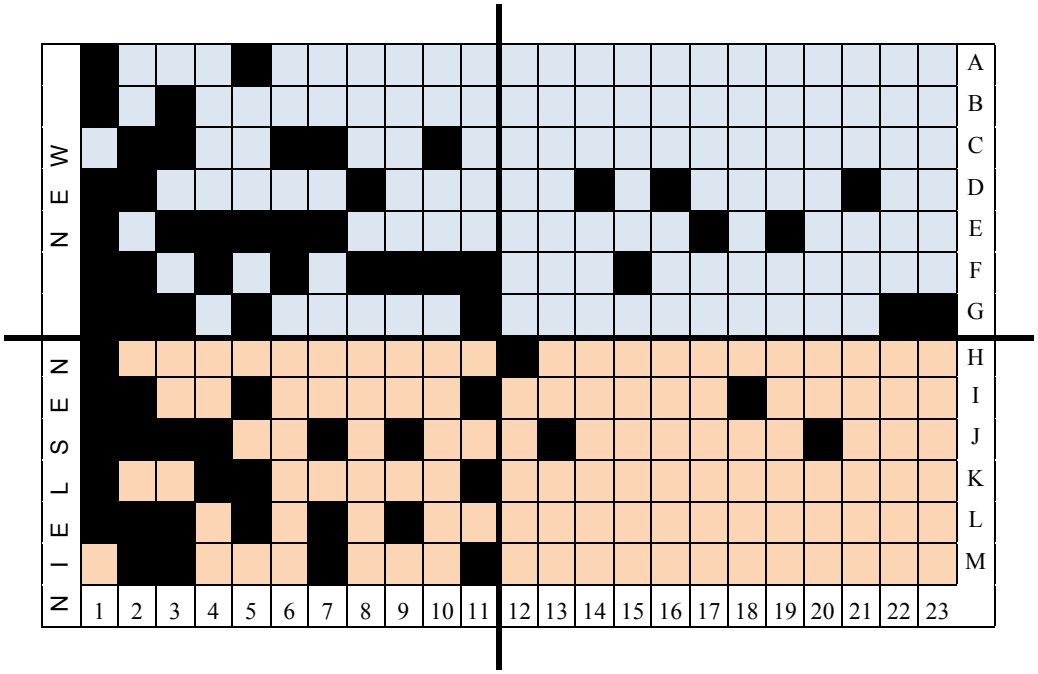

Figure 3.Problems identified per reviewer. Problems (horizontal) are numbered, and participants' (vertical) identifiers correspond to Table 2. Top: new heuristics; bottom: Nielsen's. A black square marks a particular problem as identified by a particular reviewer.

Results, Part 2: Effectiveness of Heuristics Sets

Figure 4 summarises the problems found by each heuristics set. Any problem which was found at least once, for the given set of heuristics, is marked in black. Some problems were found with only one set, and not the other. A total of 11 problems were identified only with the new set (problems 6, 8, 10, 14, 15, 16, 17, 19, 21, 22, 23). Four problems were found only with the Nielsen set (problems 12, 13, 18, 20). Of problems which were identified by at least two evaluators, three were uniquely identified using the new set (6, 8, 10), and none with Nielsen's set. This indicates that the new set may be better able to identify some valid problems.

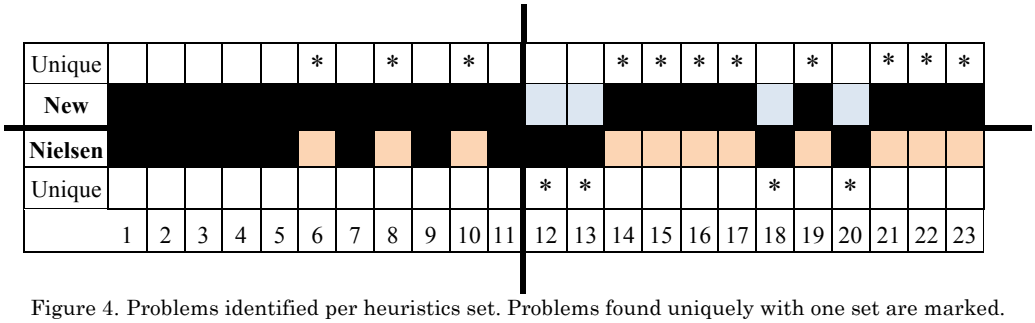

Figure 4. Problems identified per heuristics set. Problems found uniquely with one set are marked.

To further interpret the results we used validation methods proposed by Sears [1997] and Hartson et al. [2001], which rely on comparing "thoroughness" and "validity" per

review(er). Using equations 1 and 2, thoroughness and validity can be calculated for the issues identified by the reviewers.

Thoroughness varied little between the two groups (Table 3).

Calculating validity does not produce anything useful with these results – none of the comments were clear false positives.

$$t = \frac{\#\ Real\ problems\ found}{\#\ Total\ problem\ set} \qquad v = \frac{\#\ Real\ problems\ found}{\#\ Suggested\ problems\ found}$$

Equation 1 and 2: Thoroughness (t) and validity (v)

Table 3: U-test of thoroughness (p=0.792)

Group	N	Median	Avg Rank	Min	Max
New	7	0.175	5.166	0.065	0.290
Nielsen	6	0.163	4.672	0.065	0.264

Results, Part 3: Comments Made

While there was little significant difference in the raw number of issues detected using the two sets of heuristics, we discovered another – and unexpected – difference: The length and content of the comments made accompanying the issue reports varied significantly.

In total, participants in the study made 71 written comments. 37 of these comments were made by evaluators using the new set of heuristics and 34 were made by those using Nielsen's heuristics (Figure 5). The mean number of comments per evaluator was 5.29 and 5.67 respectively.

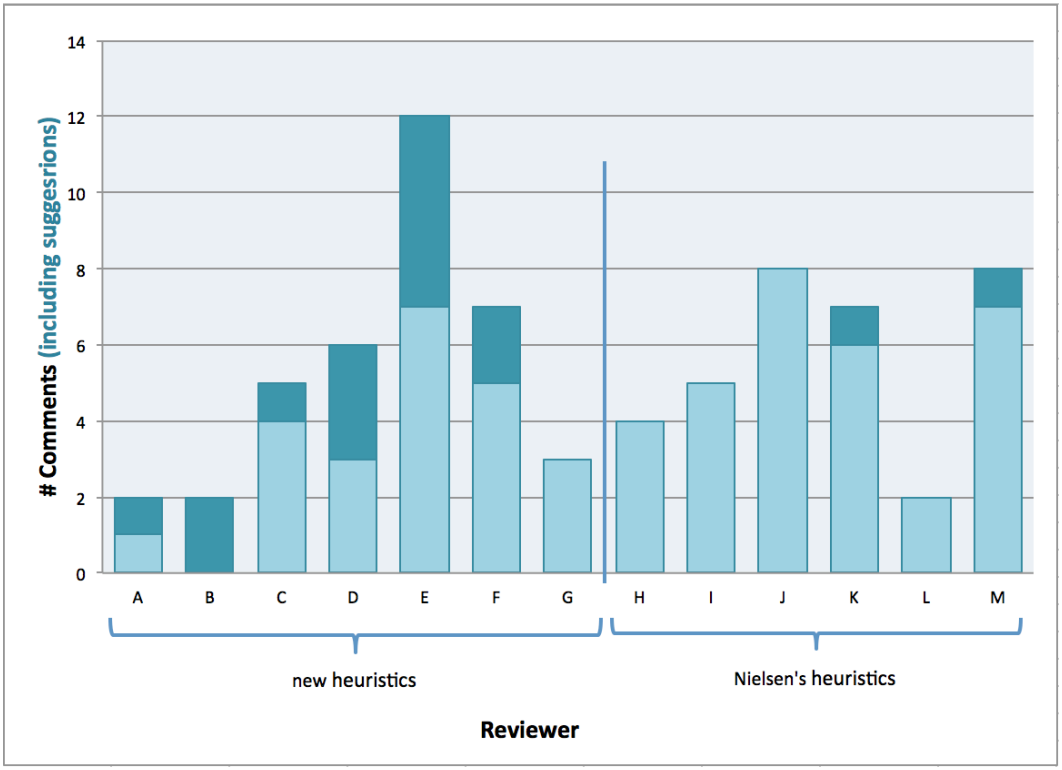

Figure 5. Number of written comments made per reviewer (darker: comments including suggestions).

This data shows that the *number* of comments was broadly the same; however, the *length* of comments – and the qualitative information supplied in these – differed significantly. Table 4 shows the length of comments in characters. From this data we can see that comments made by participants using the new set of heuristics are significantly longer.

Table 4: Comment length by group (t-test p-value = 0.018)

Group	N	Mean	Min	Max
New	37	114.4	28	499
Nielsen	34	48.0	21	113

Two typical comments, from different reviewers, are,

> *"H8 simplicity. There is something strange with the way a whole block of blocks is moved, and one has to separate the first of these in."* (Reviewer B)

> *"H10, 8. I can not delete an statement by right clicking it if it's in between a sequence therefore I had to do 2 steps (separated them, deleted and put the rest back together)."* (Reviewer F)

The longest comments with the new heuristics are paragraph-length. For instance,

> *"H11. Dragging and dropping the blocks felt slow when doing a sequence of 10, doing the edit on both cars separately was annoying, and also replacing blocks was annoying, though I found a trick that helped (first add the new block beneath the old block, then drag it (and all thus all that follows it) to*

> *the right spot above the old block, then remove the old block which as it's the last item, doesn't uncouple the other blocks). Still, a right-click--delete option would have saved a lot of time."* (Reviewer D)

In contrast, a longer comment from the Nielsen group reads,

> *"H8. Some of the control colours are very similar, slowing down rate that you can guess where commands are stored."* (Reviewer I)

We are aware that having longer comments, per se, is not the aim of the new heuristics. However, in addition to being longer, the comments contained more detailed and descriptive feedback. Below, the content of the comments is discussed in more detail.

Types of comments and their content

All comments collected were coded for the problems they described and the nature of the comment. The coding for types of comment was open – there were no predetermined categories of response.

Even though not explicitly instructed to do so, many of the evaluators included suggestions (sometimes quite detailed) in their problem reports. The following are examples of suggestions, all from the new heuristics (emphasis added):

> "Had to switch tabs a lot, *process would be more efficient if they were rearranged, or more visible at once".*
>
> [About there being a work-around to shuffling blocks] *"Still, a right-click delete option would have saved a lot of time".*
>
> *"maybe have a way of integrating the left hand, e.g. by being able to switch between block-menus (which is done by clicking a button in the top-left square) with the key-board?".*
>
> "[Would be] *Easier to make changes then duplicate rather than editing each one".*
>
> *"Maybe try, and balance the sprite pictures so there are a bit more girlish ones…"*
>
> "When the last change is compiled the button is still active (even though there is a state indicating). *However it would be better to block it".*

Reviewers made roughly the same number of comments using each set. However, the difference in suggestion frequency was significant. There were only two suggestions made with the existing heuristics, out of 34 comments, while the 37 comments from the new set included 14 suggestions. Only one reviewer using the new set did not make a suggestion for improvement.

Some of the comments with the new set were phrased as cause-effect pairs. The problems were described as in other reports, but was also linked to an underlying issue or concept in the system. For example:

> *"No types on the variables, so string/int confusion of 90/ninety could not be prevented".*
>
> *"[…] as it's the last item, doesn't uncouple other blocks".*
>
> *"The dragging blocks is very slow, [because] having to use the mouse a lot makes me use the right hand a lot more than the left."*

Post-experiment interviews

The post experiment interviews were semi-structured and lasted between 12 and 27 minutes. The interview questions and topics covered the nature of the heuristics, problems with their use and interpretation, and ease (or otherwise) of their application.

The qualitative data identified issues mostly related to wording of heuristics and interpretation of some concepts or terms. The most relevant are:

— New set, Heuristic 11: The term or concept of "viscosity" was not clear to all evaluators, and some found it more difficult to assess than other heuristics. The reason was a lack of colloquial understanding of the term.

— New set, Heuristic 8 and 11: The trouble with the term "viscosity" crystallised especially in some evaluators' difficulty distinguishing it from simplicity (in the sense of visual simplicity). One evaluator asserted there to be no difference.

— Nielsen's set, Heuristic 8: "Aesthetic and minimalist design". One evaluator did not think that it was appropriate judge a violation of the guideline "Dialogues should not contain information which is irrelevant or rarely needed" [Nielsen 2005] as a violation of this heuristic, and also had trouble finding a match using any of the other heuristics.

— New set, Heuristic 1: The phrase "intended audience of learners" was criticised as being ambiguous.

Overall, most comments in the interview were positive, and the evaluators felt largely confident in applying the heuristics.

5. FUTURE WORK

The present data resulting from the evaluation of the heuristics, as described here, is promising. The sample size, however, is still fairly small, and the interpretation of the results was carried out by the authors. Both these aspects introduce risks to the validity of the study.

To increase the confidence in the results, we plan to repeat the study using a different software system and more evaluators. An additional goal is also to remove the heuristics authors from the organisation and implementation of the experiment, and from the interpretation of the results. This serves to remove possible confirmation bias.

A yet more reliable result would be obtained by a study planned and carried out entirely independently from the authors; we hope that others will be interested enough to set up such an experiment.

In a future study, it may also be interesting to include severity ratings in the investigation – part of the "canonical" heuristic evaluation method described by Nielsen, but not included in all variants of heuristic evaluation. We did not use them in our studies to simplify comparison and interpretation of results; inclusion of this additional dimension would have added another variable that complicates the analysis. However, we envisage the heuristics as also being used during a design process, not only for evaluation after the fact. In that case, numerically rating problems is much less important since the heuristics are typically applied by the designers themselves.

6. DISCUSSION AND CONCLUSION

The authors' experience with using existing sets of heuristics to evaluate a number of educational programming systems led to the hypothesis that heuristic evaluation for

such systems could be improved with a new set of domain specific heuristics. Such a set of heuristics was then developed, following a set of formulated goals for the individual heuristics and the set as a whole. The goals aimed at maintaining both effectiveness and practicality of the heuristics set, while the hope was that the application of the new set of heuristics would be more effective for our target application domain than using existing sets.

Effectiveness of the heuristics is measured in thoroughness, validity and reliability. Following the formulation of the heuristics, the authors conducted two evaluative studies: one by applying the new heuristics to a number of systems and measuring the outcomes against a known set of reference problems, and a second one comparing the new heuristics set to Nielsen's heuristics using a group of independent evaluators.

The outcome of the first study suggests that the new set of heuristics provides a marked improvement in thoroughness and also shows a greater degree of orthogonality. Reliability cannot be measured with this study.

There are several risks and limitations inherent in the study performed. Since the study was performed by the authors of the heuristics, there is an implicit risk of bias. The problems used as the reference set were known in advance, introducing further risk of bias. Reliability cannot be measured without using a larger set of independent evaluators.

The second study, using independent evaluators, showed that the new heuristics set can be used to find actual verifiable problems at least as well as Nielsen's set, and possibly better.

While the raw number of issues reported was similar using both sets, the new heuristics set uncovered more different problems and included more issues not identified with Nielsen's set than *vice versa*.

Descriptions of issues produced using the new set were significantly longer and more detailed. Qualitative analysis of the issue reports identified significant amounts of additional useful information included in the evaluations using the new set, which were not found using Nielsen's set. The useful information provided falls mainly into two categories: suggested fixes and identification of cause-effect relationships.

While it is too early to assert the superiority of the new set over existing heuristics – both general and domain specific – with a high degree of confidence (because of the risk of confirmation bias), our results seem to indicate a clear advantage, and we hope that others will replicate and confirm our findings.

REFERENCES

Allen, E., Cartwright, R., and Stoler, B. 2002. DrJava: a lightweight pedagogic environment for Java. *SIGCSE Bull.* 34, 1 (2002), 137-141. DOI=http://dx.doi.org/10.1145/563517.563395

Bastien, J. M. C. and Scapin, D. L. 1995. Evaluating a user interface with ergonomic criteria. *International Journal of Human-Computer Interaction*, 7(2), 105-121. DOI:http://dx.doi.org/10.1080/10447319509526114

Begel, A. and Klopfer, E. 2007. StarLogo TNG: An introduction to game development. *Journal for E-Learning*, 2007.

Bekker, M. M., Baauw, E., and Barendregt, W. 2008. A comparison of two analytical evaluation methods for educational computer games for young children. *Cognition, Technology & Work*, 10(2), 129-140. DOI:10.1007/s10111-007-0068-x

Ben-Ari, M. 1998. Constructivism in Computer Science Education. *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, 257-261. DOI:http://dx.doi.org/10.1145/274790.274308

Berland, M. and Martin, T. 2011. Clusters and patterns of novice programmers. *The Meeting of the American Educational Research Association*, New Orleans.

Birnbaum, B. E. and Goldman, K. J. 2005. Achieving flexibility in direct-manipulation programming

environments by relaxing the edit-time grammar. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 259-266.

Blackwell, A. F. 1996. Metacognitive theories of visual programming: What do we think we are doing? *Visual Languages, 1996. Proceedings., IEEE Symposium on Visual Languages*, 240-246. ISBN:0-8186-7508-X

Bruce, K. B., Danyluk, A. P., and Murtagh, T. P. 2001. Event-driven programming is simple enough for CS1. *ACM SIGCSE Bulletin*, 33(3), 4. DOI:http://dx.doi.org/10.1145/507758.377440

Burke, Q. and Kafai, Y. B. 2010. Programming & Storytelling: Opportunities for learning about coding & composition. *Proceedings of the 9th International Conference on Interaction Design and Children*, 348-351. DOI:http://dx.doi.org/10.1145/1810543.1810611

Caspersen, M. E. and Christensen, H. B. 2000. Here, there and everywhere-on the recurring use of turtle graphics in CS1. *Proceedings of the Fourth Australasian Computing Education Conference* (ACE 2000), 34-40.

Cooper, S., Dann, W., and Pausch, R. 2003 Teaching objects-first in introductory computer science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE 2003), 191-195, ACM. DOI:http://dx.doi.org/10.1145/792548.611966

Cooper, A., Reimann, R., and Cronin, D. (Eds.) 2007. *About Face 3: The Essentials of Interaction Design*. Chapter 25, errors, alerts and confirmation. (pp. 529-550). Indianapolis: Wiley. ISBN: 0470084111

de Raadt, M., Watson, R., and Toleman, M. 2002. Language trends in introductory programming courses. *Informing Science: Where Parallels Intersect*, 329-337.

Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. 2011. Understanding the syntax barrier for novices. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, 208-212. DOI:http://dx.doi.org/10.1145/1999747.1999807

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*, 443-460.

Green, T. R. G. and Petre, M. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2), 131-174.

Hartson, H. R., Andre, T. S., and Williges, R. C. 2001. Criteria for evaluating usability evaluation methods. *International Journal of Human-Computer Interaction*, 13(4), 373-410.

Hundhausen, C. D. and Brown, J. L. 2007. An experimental study of the impact of visual semantic feedback on novice programming. *Journal of Visual Languages and Computing*, 18(6), 537-559. DOI:http://dx.doi.org/10.1016/j.jvlc.2006.09.001

Jadud, M. C. 2006. Methods and tools for exploring novice compilation behaviour. *Proceedings of the Second International Workshop on Computing Education Research*, 73-84. DOI:http://dx.doi.org/10.1145/1151588.1151600

Jakobsen, M. R. and Hornbæk, K. 2006. Evaluating a fisheye view of source code. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 377-386. DOI:http://dx.doi.org/10.1145/1124772.1124830

Kelleher, C. and Pausch, R. 2007. Using storytelling to motivate programming. *Communications of the ACM*, 50(7), 58-64. DOI:http://dx.doi.org/10.1145/1272516.1272540

Ko, A. J., Aung, H. H., and Myers, B. A. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, 1557-1560.

Ko, A. J. and Myers, B. A. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 387-396. DOI:http://dx.doi.org/10.1145/1124772.1124831

Ko, A. J., Myers, B. A., and Aung, H. H. 2004. Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages and Human Centric Computing*, 199-206. DOI:http://dx.doi.org/10.1109/VLHCC.2004.47

Kölling, M. 1999. *The Design of an Object-Oriented Environment and Language for Teaching*. PhD thesis, University of Sydney, Basser Department of Computer Science.

Kölling, M. 2010. The Greenfoot Programming Environment. *ACM Transactions on Computing Education* (TOCE), 10(4), 14. DOI:http://dx.doi.org/10.1145/1868358.1868361

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. 2003. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, Special Issue on Learning and Teaching Object Technology, 13(4), 249-268.

Konidari, E. and Louridas, P. 2010. When students are not programmers. *ACM Inroads*, 1(1), 55-60. DOI:http://dx.doi.org/10.1145/1721933.1721952

MacLaurin, M. 2009. Kodu: End-user programming and design for games. *Proceedings of the 4th International Conference on Foundations of Digital Games*, 2. DOI:http://dx.doi.org/10.1145/1536513.1536516

Malone, T. W. 1980. What makes things fun to learn? Heuristics for designing instructional computer

games. *Proceedings of the 3rd SIGSMALL Symposium and the First SIGPC Symposium on Small Systems*, 162-169. DOI:http://dx.doi.org/10.1145/800088.802839

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education* (TOCE), 10(4), 16. DOI:http://dx.doi.org/10.1145/1868358.1868363

Maloney, J. H., Kafai, Y. B., Resnick, M., and Rusk, N. 2008. Programming by choice: Urban youth learning programming with Scratch. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 367-371. DOI:http://dx.doi.org/10.1145/1352135.1352260

Marceau, G., Fisler, K., and Krishnamurthi, S. 2011. Measuring the Effectiveness of Error Messages Designed for Novice Programmers. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 499-504. DOI:http://dx.doi.org/ 10.1145/1953163.1953308

McKay, F. 2012. A prototype structured but low-viscosity editor for novice programmers. BCS-HCI '12 Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers, 363-368.

Miara, R. J., Musselman, J. A., Navarro, J. A., and Shneiderman, B. 1983. Program indentation and comprehensibility. *Communications of the ACM*, 26(11), 861-867. DOI:http://dx.doi.org/10.1145/182.358437

Milner, W. W. 2010. A broken metaphor in Java. *ACM SIGCSE Bulletin*, 41(4), 76-77. DOI:http://dx.doi.org/10.1145/1709424.1709450

Nielsen, J. 2005. Ten usability heuristics. Retrieved May 08, 2015 from http://www.useit.com/papers/heuristic/heuristic_list.html

Nielsen, J. 2005b. How to conduct a heuristic evaluation. Retrieved May 8, 2015, from http://www.useit.com/papers/heuristic/heuristic_evaluation.html

Nielsen, J. and Molich, R. 1990. Heuristic evaluation of user interfaces. CHI '90 *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Empowering People*, 249-256. DOI:http://dx.doi.org/10.1145/97243.97281

Overmars, M. 2004. Teaching Computer Science through Game Design. *Computer* 37, 4, 81-83. DOI:http://dx.doi.org/10.1109/MC.2004.1297314

Pane, J. F. 2002. A programming system for children that is designed for usability. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Pane, J. F. and Myers, B. A. 1996. Usability issues in the design of novice programming systems. Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132, Pittsburgh, PA, August 1996, 85 pages.

Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. New York: Basic books. ISBN 0465046746

Pattis, R. E. 1981. *Karel the Robot: A gentle introduction to the art of programming* (1st ed.). New York, USA: John Wiley and Sons. ISBN 0471597252

Python Software Foundation. 2012. Python standard library: Turtle graphics for tk. Retrieved May 08, 2015 from http://docs.python.org/library/turtle.html

Rambally, G. K. 1986. The influence of color on program readability and comprehensibility. *Proceedings of the 17th SIGCSE Symposium on Computer Science Education*, 18(1) 173-181. DOI:http://dx.doi.org/10.1145/953055.5702

Robins, A., Haden, P., and Garner, S. 2006. Problem distributions in a CS1 course. *Proceedings of the 8th Australian Conference on Computing Education*, 165-173. ISBN 1-920682-34-1

Sanders, D. and Dorn, B. 2003. Jeroo: A tool for introducing object-oriented programming. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 201-204. DOI:http://dx.doi.org/10.1145/792548.611968

Sanders, K. and Thomas, L. 2007. Checklists for grading object-oriented CS1 programs: Concepts and misconceptions. *ACM SIGCSE Bulletin*, 39(3), 166-170. DOI:http://dx.doi.org/10.1145/1269900.1268834

Schulte, C. and Bennedsen, J. 2006. What do teachers teach in introductory programming? *Proceedings of the Second International Workshop on Computing Education Research*, 17-28. DOI:http://dx.doi.org/10.1145/1151588.1151593

Sears, A. 1997. Heuristic walkthroughs: Finding the problems without the noise. *International Journal of Human-Computer Interaction*, 9(3), 213-234.

Shingō, S. 1986. *Zero quality control: Source inspection and the poka-yoke system*. Productivity Press, New York, NY. ISBN 0915299070

Siegfried, R. M. 2006. Visual programming and the blind: The challenge and the opportunity. *ACM SIGCSE Bulletin*, 38(1), 275-278. DOI:http://dx.doi.org/10.1145/1124706.1121427

Slack, J. M. 1990. *Turbo Pascal with turtle graphics*. St. Paul: West Publishing Co.

Thomas, L., Ratcliffe, M., and Robertson, A. 2003. Code warriors and code-a-phobes: A study in attitude and pair programming. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 363-367. DOI:http://dx.doi.org/10.1145/792548.612007

Thomasson, B., Ratcliffe, M., and Thomas, L. 2006. Identifying novice difficulties in object oriented design. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 28-32. DOI:http://dx.doi.org/10.1145/1140123.1140135

Turkle, S. and Papert, S. 1992. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1), 3-33.

Ware, C. 2008. *Visual thinking for design*. Burlington, MA: Morgan Kaufmann Publishers.

Weinberg, G. M. 1998. *The psychology of computer programming* (2nd ed.). New York, USA: Dorset House Publishing.