Language-based Support for Computational Thinking

Katy Howland, Judith Good, Keiron Nicholson
IDEAs Lab, Department of Informatics, University of Sussex
{K.L.Howland, J.Good, K.Nicholson}@sussex.ac.uk

Abstract

This paper explores the potential for simplified programming languages to support the development of computational thinking skills in non-programmers. We suggest that novice programming languages might offer a starting point for non-programmers to engage with a substantial subset of computational thinking concepts, and assess a number of languages from this perspective. We outline four key computational thinking skills and examine the support provided by existing languages. We then describe additional characteristics which would be important for a new language aimed specifically at developing computational thinking skills.

1. Introduction

The notion of "computational thinking" has recently been highlighted by Wing, who describes it as "solving problems, designing systems and understanding human behavior, by drawing on the concepts fundamental to computer science" [1, p.33]. She later adds to this definition, describing it as "defining abstractions, working with multiple layers of abstraction and understanding the relationships among the different layers", along with the ability to 'automate' these abstractions [2, p.3718].

Many researchers have since put forward their own views on the concept. Some have focused on the relationships with, and applicability within, other disciplines [3], whilst others have examined the implications for computing as a field [4]. The importance of teaching computational thinking skills from a young age, and to people who do not intend to learn to program, is a theme which has captured the imagination of some researchers [5, 6].

Since Wing's 2006 article [1] there has been limited progress in clarifying what computational thinking actually is, however, in order to progress with a curriculum for computational thinking, it is necessary to first determine the core skills, and define them in sufficient detail to begin associating them with learning activities. Secondly, it is important to determine *how* computational thinking can be taught. To some extent, the acquisition of computational thinking skills has to date been a side effect of learning to program: as computer science students learn programming languages, they also seem to pick up a number of higher level skills which are more broadly applicable outside of computer science. However, the question arises as to whether students *must* learn to program in order to acquire these skills.

It is likely that learning a general purpose programming language is not the best route, given the steep learning curve associated with programming, and the fact that many of the target audience will not go on to become programmers. Furthermore, computational thinking is broader than programming, and likely operates at a higher level of abstraction.

Nonetheless, if people are to gain knowledge of computational concepts, they need to be able to access and interact with computational structures. Hence, there needs to be a language or form of representation in which these structures can be expressed. It has been suggested that specially designed programming languages could help a wider range of people to engage in computational thinking [6]. Visual languages and mini-languages aim to minimise language syntax and thus complexity, and could act as a starting point for designing such a language. Conversely, languages which focus on allowing users to program in a way which is more intuitive and which obscures the underlying computation taking place may in fact hinder the development of computational thinking skills.

We outline four key computational thinking skills, and reflect upon their usefulness to non-programmers. We discuss the need for representational support for the acquisition of these skills, and consider the extent to which existing programming languages provide this support. Finally, we conclude with requirements for a language designed specifically to support computational thinking.

2. Four key computational thinking skills and existing language support

We have identified four key computational thinking skills which have been selected firstly, because they

represent key concepts in computer science and secondly, because their applicability is much broader than computer science and hence they are of value to non-programmers. Although these are unlikely to comprise the full set of skills, they are intended as a first step towards defining a concrete set of computational thinking skills which can support efforts to bring computational thinking to a wider audience.

As noted in the introduction to this paper, some form of representation is necessary in order to scaffold the acquisition of computational thinking skills. Although the languages we examine below were designed to facilitate novice programming, we nonetheless consider their potential to teach the key computational thinking skills we have identified.

2.1 The ability to define clear, specific and unambiguous instructions for carrying out a process

This skill relates to the computer science practice of writing step-by-step instructions for a program, forcing the programmer to resolve any ambiguities or logical inconsistencies that were not apparent when considering it at a higher level. It also relates to the computer science concept of control flow. By using programming constructs like conditionals, loops and branching statements to control the order in which the instructions in a program execute, the programmer ensures that there are clear instructions for dealing with different conditions, and that those conditions are themselves clearly defined. This is a skill that easily translates to non-computing contexts: consider the frustrations of unclear assembly instructions for furniture, or the dangers of unclear dosage instructions for medicine. A related skill is the ability to identify instructions which are unclear, vague or ambiguous.

In order to develop this skill, users need to be able to specify instructions precisely. Most languages can teach this to some extent, as it is one of the basic skills necessary to create commands in a language. Whilst simplified text based languages such as Squeak [7] can address this skill, they also require substantial knowledge of syntax which is not necessary for developing the ability to specify unambiguous instructions. Other languages such as Alice [8] aim to provide syntax which is more in line with the way that humans. Although an excellent approach for languages which aim to empower users [9], it may be counter-productive for learning to think computationally.

Visual languages reduce the need for users to learn syntax. As such, languages such as Alice [8], AgentSheets [10], Scratch [11] (and languages in the same family such as StarLogo TNG [12]), offer an excellent opportunity for users to learn how to specify commands unambiguously without the additional challenge of learning syntax.

Some languages attempt to support understanding of control flow through representations of the run-time progress of the program. ToonTalk [13] and RAPTOR [14] are visual programming environments which provide a visualisation of program execution. This visualisation could make it easier for a non-programmer to gain an understanding of the way in which commands are carried out. ToonTalk makes use of metaphors to provide this visual representation. Although metaphors run the risk of focussing too much on translating computational concepts to human concepts, they may also provide concrete examples to aid a developing understanding of computation.

2.2 The ability to design a system made up of distinct components with clearly defined areas of responsibility that they do not stray from

This relates to the computer science concept of separation of concerns, which involves ensuring that constituent parts of a program are distinct and do not overlap in functionality, with each part having an inherent well-defined and unique area of responsibility. This principle encourages the design of systems which are easier to understand and manage, and in which the failure of one component has a reduced effect on other components in the system. Illustrative examples include object-oriented systems (featuring distinct objects with defined areas of responsibility); subroutines (clearly labelled portions of code which perform specific tasks); and the separation of presentation from content, for example, separating a user interface from the database it accesses. This skill translates to non-computing contexts involving the design or administration of systems. For example, any large business will split its operations into separate departments, independently governing human resources, public relations, marketing and so on. Each department has a clear area of responsibility. They will not directly involve themselves with the work of any other department, nor will they abdicate responsibility for anything in their own area.

Many languages aimed at novices could be considered object-oriented, including Alice [8] and Scratch [11]. The user is implicitly encouraged to think about a program in terms of objects, particularly when any kind of metaphor is employed, such as the scenarios used with Greenfoot [15] where, for example, a koala moves around a simple ecosystem. However, such metaphors do not encourage users to think about areas of responsibility; an ecosystem has no objective, so organisms in an ecosystem have no responsibility other than to their own survival.

Most languages allow procedures to be given meaningful names when defined (e.g. Alice [8]), however there appears to be no encouragement to consider what a meaningful name is, or what would be an appropriate size of task to embody. Playground [16] separates presentation from content, giving objects changeable 'costumes' which set their appearance. This has promise as a way of encouraging users to distinguish appearance from functionality.

2.3 The ability to design a system made up of components which deliberately reveal certain information and operations related to their purpose, and hide everything else

This relates to the computer science concept of encapsulation, which involves software components being represented solely in terms of their overall functionality, whilst details of how the component carries out the functionality are hidden from other components. It allows components to be changed or replaced completely with minimal effect on the rest of the system. This skill translates to non-computing contexts involving the design or administration of systems. For example, if an employee wants to check their payroll history they can contact the payroll department and request the desired information. The payroll department would provide the information without revealing the details of how they had retrieved it. By contrast, in a system without encapsulation, the employee would go directly to the filing cabinet holding the information and retrieve it themselves. The inner workings of the payroll department would be completely exposed, and if the filing cabinet were to be moved to another building, the employee's method for retrieving the information would no longer work. In the previous example, the employee would have been unaware of and unaffected by this change.

There is little support for encapsulation in existing languages for novices, and almost no scaffolding for teaching it. Greenfoot [15] encapsulates objects and procedures to an extent, with internal mechanisms only displayed at the user's request. However, even languages which intend programmers to use encapsulation cannot enforce it. This choice is ultimately up to the programmer, and it relies on their understanding that although making details of components' functionality accessible to other components may be more convenient in the short term, it can create problems in the long term. This is a difficult idea for novices to take on board. An alternative strategy might be to consider introducing programming in a way that encapsulation seems fundamental, and going against the principle of encapsulation would seem unnatural, similar to the "objects first" approach. The object-oriented BlueJ [17] environment explicitly represents the relationships between objects graphically: this could be useful for promoting the 'system level' as the most natural level of abstraction at which to think.

2.4 Understanding that a complex system of behaviours can arise from simple interactions

This relates to the concept of emergence, which is the way in which a complex system of behaviours can arise from interactions based on simple rules. Understanding this concept allows us to identify the unique structures, patterns and properties of a particular emergent system, and to recognise them as emergent. The concept of emergence occurs in many fields, including biology, mathematics, philosophy, cognitive science, economics or urban planning. The ability to understand emergence can engender useful insights, for example, emergence helps to explain the phenomenon of birds flocking. Finally, it is valuable to be able to recognise emergent behaviour when analysing systems, as unexpected emergent features of a system could interfere with its intended purpose in ways that are counterproductive or even dangerous.

Languages and environments which allow users to create objects with rules for interaction, such as Alice [8] and Stagecast Creator [18], have potential for allowing users to gain an understanding of emergence. To teach emergence more extensively, a language should make it as simple as possible to place many objects with modifiable rules of behaviour in an environment in which they can interact. StarLogo TNG [12] does just this by providing explicit support for users to create models where complex patterns arise from simple interactions between multiple objects.

3. Conclusion

In this paper we've described a set of four key computational thinking skills, and looked at whether existing languages can support the development of such skills. The need to define clear, specific and unambiguous instructions for carrying out a process is heavily featured across most languages. There is some support for teaching the principle of separation of concerns, while support for encapsulation is relatively rare. None of the languages explicitly teach emergence, although some offer opportunities for creating emergent behaviour. In languages which do support a particular computational thinking concept or skill, they are rarely scaffolded or made explicit to the user.

This would suggest the need for a language designed specifically to teach computational thinking skills. Requirements for such a language could be:

- present a task at whichever level of abstraction is most useful for drawing out the higher-level concepts, without necessarily having to relate it back to programming.
- if lower levels of abstraction are necessary, e.g. to the level of programming instructions, fundamental programming concepts and structures should be presented in the service of illuminating the higher-level concepts.
- provide a blend of the most useful aspects of languages which aim to teach programming and languages which aim to empower users. These would include: ensuring that the language is syntax-light to avoid forcing the user to learn a formal syntax; and minimizing syntax errors so as to avoid errors based on a misunderstanding of syntax discouraging the user in their pursuit of semantic understanding.
- in the absence of an enthusiasm to learn programming, provide an incentive to engage with the language; placing it in the context of a motivational design-based activity (such as building 3D games or making music) should excite a user's interest and provoke curiosity about the underlying concepts.

In terms of the wide range of computational thinking concepts and skills that could be taught using a specialised language, it is clear that there are more to be identified within programming, and many more beyond that in the various fields of computing. Our ongoing work consists of attempting to simultaneously map out the space of computational thinking concepts whilst looking at the most appropriate ways in which to provide support for learning these concepts.

4. References

[1] J. Wing, "Viewpoint-Computational Thinking," *Communications of the ACM-Association for Computing Machinery-CACM,* vol. 49, no. 3, pp. 33-35, 2006.

[2] J. Wing, "Computational thinking and thinking about computing," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences,* vol. 366, no. 1881, pp. 3717-3725, 2008.

[3] A. Bundy, "Computational thinking is pervasive," *Journal of Scientific and Practical Computing Noted Reviews,* vol. 1, no. 2, pp. 67-69, 2007.

[4] P. Denning, "Computing is a natural science," *Communications of the ACM-Association for Computing Machinery-CACM,* vol. 50, no. 7, pp. 13-18, July 2007, 2007.

[5] G. Fletcher, and J. Lu, "Human computing skills: rethinking the K-12 experience," *Communications of the ACM-Association for Computing Machinery-CACM,* vol. 52, no. 2, pp. 23-25, 2009.

[6] M. Guzdial, "Paving the way for computational thinking," *Communications of the ACM-Association for Computing Machinery-CACM,* vol. 51, no. 8, pp. 25-27, 2008.

[7] D. Ingalls, T. Kaehler, J. Maloney *et al.*, "Back to the future: The story of Squeak, A practical Smalltalk written in itself," in Proceedings of OOPSLA '97, ACM SIGPLAN Notices, 1997, pp. 318-326.

[8] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-D tool for introductory programming concepts," *Journal of Computing Sciences in Colleges,* vol. 15, no. 5, pp. 107-116, 2000.

[9] C. Kelleher, and R. Pausch, "Lowering the barriers to programming," *ACM Computing Surveys,* vol. 37, no. 2, pp. 83-137, 2005.

[10] A. Repenning, A. Ioannidou, and J. Zola, "AgentSheets: End-User Programmable Simulations," *Journal of Artificial Societies and Social Simulation,* vol. 3, no. 3, 2000.

[11] J. Maloney, Y. Kafai, M. Resnick *et al.*, "Programming by choice: urban youth learning programming with scratch," in Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, Portland, Oregon, 2008, pp. 367-371.

[12] K. Wang, C. McCaffrey, D. Wendel *et al.*, "3D game design with programming blocks in StarLogo TNG," in Proceedings of the 7th International Conference on Learning Sciences, Bloomington, Indiana, 2006, pp. 1008-1009.

[13] K. Kahn, "ToonTalkTM—An Animated Programming Environment for Children," *Journal of Visual Languages and Computing,* vol. 7, no. 2, pp. 197-217, 1996.

[14] M. Carlisle, T. Wilson, J. Humphries *et al.*, "RAPTOR: a visual programming environment for teaching algorithmic problem solving," in Proceedings of 36th SIGCSE Technical Symposium on Computer Science Education, St. Louis, Missouri, USA, 2005, pp. 176 - 180

[15] M. Kölling, and P. Henriksen, "Game Programming in Introductory Courses with Direct State Manipulation," *ACM SIGCSE Bulletin,* vol. 37, no. 3, pp. 59-63, 2005.

[16] J. Fenton, and K. Beck, "Playground: an object-oriented simulation system with agent rules for children of all ages," *ACM SIGPLAN Notices,* vol. 24, no. 10, pp. 123-127, 1989.

[17] M. Kölling, B. Quig, A. Patterson *et al.*, "The BlueJ system and its pedagogy," *Computer Science Education,* vol. 13, no. 4, pp. 249-268, 2003.

[18] D. Smith, A. Cypher, and L. Tesler, "Novice programming comes of age," *Communications of the ACM,* vol. 43, no. 3, pp. 75-81, 2000.