Learning, Media and Technology: a doctoral research conference

04 July 2011, London

Four Approaches to Teaching Programming

Cynthia C. Selby
University of Southampton
Southampton, England
C.Selby@soton.ac.uk

Abstract

Based on a survey of literature, four different approaches to teaching introductory programming are identified and described. Examples of the practice of each approach are identified representing procedural, visual, and object-oriented programming language paradigms. Each approach is then further analysed, identifying advantages and disadvantages for the student and the teacher. The first approach, code analysis, is analogous to reading before writing, that is, recognising the parts and what they mean. It requires learners to analyse and understand existing code prior to producing their own. An alternative is the building blocks approach, analogous to learning vocabulary, nouns and verbs, before constructing sentences. A third approach is identified as simple units in which learners master solutions to small problems before applying the learned logic to more complex problems. The final approach, full systems, is analogous to learning a foreign language by immersion whereby learners design a solution to a non-trivial problem and the programming concepts and language constructs are introduced only when the solution to the problem requires their application. The conclusion asserts that competency in programming cannot be achieved without mastering each of the approaches, at least to some extent. Use of the approaches in combination could provide novice programmers with the opportunities to acquire a full range of knowledge, understanding, and skills. Several orders for presenting the approaches in the classroom are proposed and analysed reflecting the needs of the learners and teachers. Further research is needed to better understand these and other approaches to teaching programming, not in terms of learner outcomes, but in terms of teachers' actions and techniques employed to facilitate the construction of new knowledge by the learners. Effective classroom teaching practices could be informed by further investigations into the effect on progression of different toolset choices and combinations of teaching approaches.

Introduction

The Royal Academy of Engineering (2009, p. 17) report, "ICT for the UK's Future" states "It is essential that a significant proportion of the 14-19 age group understands Computing concepts – programming, design, problem solving, usability, communications and hardware" . This, along with current policymaker's focus on a redesign of the National Curriculum (DfE 2011) and the report on vocational education (Wolf 2011), highlights the importance of providing opportunities for learners to acquire knowledge, understanding, and skills associated with programming. This setting provides the context for an investigation into different approaches to teaching programming.

In reading the literature concerned with teaching computer programming, varied approaches began to emerge. The characteristics of the different approaches were identified. By grouping the characteristics, these approaches coalesced into the four categories discussed below. It was then possible to identify further recently published literature where the different approaches, defined by their characteristics, were employed in classroom settings to teach programming. They were chosen to represent a range of programming paradigms, so are not language specific. The classroom or research based findings presented below appear in editor-reviewed media or in peer-reviewed conference proceedings.

Mastering the skills to write programs requires that learners be able to build solutions both in their minds and on a computer. This interpretation of programming can be supported by both a learning theory and a pedagogical method. The idea of knowledge not being passively absorbed, but being actively constructed by learners based on their own experiences, fits into the learning theory known as Constructivism (Ben-Ari 1998). This theory forms the foundation for Papert's Constructionism, which advocates that learning is accomplished through the creation of artefacts (Ackermann 2001). In the situation of learning to program, the writing of code (Constructionism) facilitates the building of knowledge (Constructivism). Some appropriate constructivist activities for teaching programming are identified by Wulf (2005) as including "... code walkthroughs, code reading, code debugging, and scaffolded code authoring" (p. 246).

These suggested tasks and the theory of actively constructing learning underpin the four different approaches to teaching programming as discussed below. The four individual approaches, code analysis,

building blocks, simple units, and full systems are each described and illustrated with examples from current practice. The presented literature allows for the identification of some advantages and disadvantages of each approach. Using the results reported in the literature, it is possible to extrapolate further and suggest how a combination of the approaches may be used effectively to teach programming.

Programming Terminology

For those readers not familiar with the technical terminology used in the teaching of programming, a small amount of background vocabulary may be beneficial. The high-level definitions provided below are derived from their common usage by educators and learners in classrooms.

Code	Sequences of textual or symbolic instructions, composed by a learner, which are interpreted by a device to affect its behaviour
Code reading, walkthrough, debugging	Employing various techniques to identify overall logic or causes of incorrect behaviour
Pseudocode	A language used to plan logic before implementing that logic in a device
Structured English	A type of pseudocode that uses recognisable English words such as *if*, *while*, *for*, *begin*, *end*, *input*, and *output*
Constructs Building blocks	Programming language specific keywords or symbols, which can be combined to form sequences of instructions; basic constructs include variable (name a place in memory), assignment (set a variable to some value), conditional (test if a condition is true or false), and repetition (repeat instructions for a number of times)
Structured programming	Instructions are interpreted and executed one after another in a sequence; data and code are usually kept separate
Object-oriented programming	An object is a collection of both code and data; instructions are interpreted and executed based on the passing of messages from one object to another

Code Analysis

In the code analysis approach to teaching programming, learners read and understand programming logic before writing their own. This approach is based on the use of pseudocode so it is not programming language dependent. Three studies at university level have supported this approach and another study suggests that this approach is especially appropriate for weaker students. The ability to explain programming logic and code appears to be a prerequisite for, although it does not imply, the ability to write code. This approach allows problem-solving skills to be developed, which are applicable to many domains. This approach may be disappointing to learners who want to engage with the physical computers and may not be suitable for independent learning situations because of the lack of feedback.

The code analysis approach is analogous to learning to read before learning to write. Learning to read exposes the rules of grammar for the target language. Attempting a language in this way should allow the learner to become familiar with the way in which components and constructs of the language are combined to produce meaning before attempting the construction themselves.

An implementation of this approach may involve providing learners with practice exercises. These exercises could be prepared using some type of acceptable pseudocode or structured English. These exercises might exist only on paper or could be displayed in an appropriate development environment, provided one exists for the pseudocode implementation. However, as a lone approach, code analysis does not require learners to interact with the code in an environment on a computer.

At least four recent studies support the use of the code analysis approach. At Victoria University Melbourne, Miliszewska and Tan (2007) redesigned one of the first-year computer science courses to include the object-oriented paradigm, but teaching structured programming first. The new approach to teaching the course includes studying examples of well-written code. They propose that the students will learn by imitating the examples of good practice. This approach is also supported by Kölling and Rosenberg (2001) who are proponents of studying code to internalise styles and idiom usage. Campbell and Bolker (2002) express agreement by their own belief that students learn more by reading programs written by experienced programmers than by starting with writing their own code. Code walkthroughs and code reading, both

elements of constructivist computer programming instruction (Wulf 2005), were employed by Lui, et al (2004) in their study on weaker programming students. They suggest that the editing, compiling, and executing cycle can drain the patience and confidence of weaker students. They advocate that weaker students work with paper and pencil, which presents an initial low risk environment.

There may be potential problems using the code analysis approach if employed under false assumptions. Glaser, Hartel, and Garratt (2000) point out that their students often have difficulty constructing a program, even if the same students can read and understand the solution when it is presented to them. However, a more recent study of introductory undergraduate programming students (Lister, Fidge, and Teague 2009) found that the ability to trace and explain code had a statistically significant relationship to the ability to write code. Of course, together, these findings could simply imply that the ability to write code requires some skill in tracing and explaining, but that skill in tracing and explaining may not beget the ability to write code.

Although from a learner's perspective learning to program without using a computer may seem perverse, there are advantages for the introduction of code analysis from the beginning. First, this approach means that there are no tools or environments to master in preparation for learning programming logic. Logic can be tackled straight away. Second, any implemented programming language may be used, but perhaps the most appropriate is no defined language at all. Pseudocode or structured English could be presented. Both are independent of any implemented language. Third, analysis first allows the development of skills involved in debugging and tracing to correct or ensure program behaviours. Fourth, it also provides the opportunity to expose learners to the logic behind simple foundation algorithms such as linear search and bubble sort. Early exposure to code analysis may help prepare learners for examinations that include questions involving tracing or dry running of code.

The code analysis approach may not be suitable under all circumstances. Learners may feel cheated by a technology class in which the tools are pencil and paper. In addition, depending on the age and capabilities of the learners, it may be challenging for an educator to develop a set of meaningful and understandable pseudocode instructions. For example, should words or symbols be used? Analysis first could be problematic for independent learning because of the lack of immediate feedback. There is also no verification that the pseudocode logic, identified by the learner, is a correct interpretation of a problem solution. In addition, there is the possibility that the understanding of the pseudocode may not result in skills that can be transferred to an understanding of a language defined by a strict syntax.

Although not suitable in every situation, the code analysis approach has been shown by recent studies to be beneficial to novice programmers. It has the advantage of not requiring a particular programming language or computer environment. It also promotes the development of problem-solving skills and logical thinking required in many domains. Its use can enhance the ability to understand program code, which forms part of the underlying foundation necessary to facilitate the writing of program code.

Building Blocks

In the building blocks approach to teaching programming, learners develop an understanding of individual pieces before combining the pieces to create meaning. This approach requires a set of developmental tools and a defined programming language. Language constructs are introduced and understood one at a time, in isolation, before combining them. This approach is not limited to single language paradigms and has been used in procedural, functional, and object-oriented languages environments. The building blocks approach has the advantage of the introduction of a precisely defined language syntax and the immediate feedback from a syntax checking tool. Unfortunately, learning how to use a code editor, a syntax checker, and perhaps a compiler can be challenging for beginners. In addition, they may be disappointed to find that syntactically correct code may not result in correct logic.

The building blocks approach is analogous to learning to speak individual parts of a language before combining them firstly in the verbal form and later in the written form. Young learners usually acquire verbal language skills before written skills. Foreign languages are usually taught using verbal skills first then written skills. When attempting to master a language, individual components (nouns, verbs, adjectives) are understood to some extent before putting them together to create meaning (sentences). Complex meaning is built up based on the understanding of the smaller pieces.

A specifically chosen set of tools, consisting of a programming language and development environment, is required for this approach. The choice of programming language is beyond the scope of this paper, but it will be assumed that the language and environment chosen is suitable for the capabilities of the learners. Specific language code is introduced one construct at a time. Understanding the behaviours of the constructs will be enhanced by writing the constructs in an appropriate development environment where syntax errors may be highlighted by the tool. In addition, execution of these single constructs may be possible in a development environment where the learners can comprehend, perhaps by visualisation, the

behaviours of each construct.

Research indicates that the building blocks approach is not limited to a particular programming language. It has been used with procedural, functional, and object-oriented languages. In an attempt to address high attrition rates in the computer science courses at the University of Denver, educators made the decision to switch to a games first approach (Leutenegger and Edgington 2007). In their approach, they use Flash ActionScript and introduce individual control constructs in the context of moving visual objects around the screen. Matthew Flatt, an advocate of the Racket programming language, reports using this type of approach with liberal arts students (Roy, et al 2003). Although a customised programming environment is used, only six concepts have been identified as necessary. The same approach can be used with the object-oriented paradigm, as illustrated by Sajaniemi and Hu (2006). They choose to introduce the behaviours of variables and control structures prior to objects. Again, this represents the building up of small pieces into more complex logic.

This approach incorporates the advantages of early introduction of precisely defined language syntax and early introduction of tools to assist in the programming endeavour. The different constructs of the language, broadly speaking, variable, assignment, conditional, and repetition can be taught in isolation. Further exploration of the different constructs can be accomplished by demonstrating how to vary their behaviours by changing their syntax. This approach also ensures that developed algorithms, incorporating multiple blocks, should execute, providing they are syntactically correct. This, of course, does not imply that any particular combination of blocks will be logically correct. Further, mastering the tools needed to express and verify the behaviours of the blocks provides learners with debugging tools and strategies that will be needed to develop real solutions. The immediacy of the feedback when working in a development environment can be a benefit to learners. Errors in implementing the syntax of a programming language led to frustration for Al-Imamy, Alizadeh, and Nour (2006). To overcome this, they developed a customised environment that makes use of construct templates where learners fill in the blanks. For example, the template for a conditional block can be presented with the correct syntax structure laid out. The learner only needs to fill in a box representing the test.

The building blocks approach may present challenges for novice learners. Recall from above that interactive learning requires mastering some complex tools at the same time as mastering the behaviours of the blocks. The scale of this task should not be underestimated. As stated above, Lui, et al (2004) recognised that the failure to compile, i.e. construct syntactically correct code, can have a derogatory effect on novice programmers. In addition, individual block constructs may not execute or may not be meaningful to execute even if the syntax is correct. For example, an empty repetition or an empty conditional are syntactically allowed in some languages but are not logically meaningful. Mastering individual block behaviours may not transfer to the building activity required to produce an algorithm that performs a meaningful task. It is possible for a learner to explain and understand a concept, yet be unable to apply that understanding in the construction of an algorithm (Lahtinen, Ala-Mutka, and Järvinen 2005). In addition, the relationships and interactions between blocks may be difficult for learners to grasp without the context of a problem to solve. However, the introduction of a simple problem context could be a follow-on activity.

Although the use of the building blocks approach may be challenging for learners and educators alike, it does provide for the early introduction of a precisely defined language syntax and the early introduction of a toolset to aid code writing. Introducing each language construct, one at a time, allows for natural exploration until its behaviour is understood. Once a basic behaviour is understood, experiments in varying the syntax of the constructs can be undertaken to deepen learning.

Simple Units

In the simple units approach to teaching programming, learners master set phrases using a limited vocabulary before combining the phrases to create meaning. Useful and reusable units of programming code are constructed in a development environment with a defined language syntax. Two studies exemplify this use of simple units that result from the solution of small set tasks. These reusable units can be held in a programmer's toolbox, for use in more complex solutions. Small, highly focused contexts can be used to combine constructs to create units useful in larger solutions. In common with building blocks, the simple units approach requires mastering tools and constructs at the same time. In addition, new problem solving skills can be mastered.

The simple units approach is analogous to learning to speak a language from a phrase book with a limited vocabulary. In this approach, the individual nouns, verbs, and adjectives are put together to form useful phrases. It is possible for the meaning of the phrase to be understood without complete understanding of the meaning of the individual components. However, decomposing the meaning of the phrase can lead directly to a better understanding of the individual components.

In this approach, constructs are put together to form useful and reusable units of code. It is a way, not necessarily the only way or even the best way, of building commonly occurring units of code. For example, a set of constructs could be put together in some way to sort a list or to find the maximum of a set of numbers. Small, well-specified problems, such as making a visual object bounce when it touches a maze wall, could be solved with the use of a small number of constructs. In themselves, these units may seem trivial. However, they become more powerful when joined together to effect solutions to larger problems. As with the building blocks approach, the learners will necessarily need a development environment and, unfortunately, as mentioned above, will be exposed to all the problems and issues associated with it.

This approach has been successfully employed in two studies whereby learners created reusable units of code to provide a basis for complex solutions. Reges (2006) at the University of Washington, Seattle uses the idea of small set tasks to increase both enrolments and student satisfaction. Although the course is taught in Java, the approach relies heavily on mastering the basics of problem solving. Small set tasks are designed to incorporate the use of constructs to create a solution. However, each small task, such as displaying an output, repetition, taking input, or reading a file, can be used as a simple unit. Once the units have been written and understood, they can be joined together to create the solutions to more complex assignments. This approach of unit building is extended by Lui, et al (2004) who suggest that creating a toolbox of key program segments is valuable for weaker students, who find programming from scratch challenging. This toolbox gives learners, especially weaker ones, a starting point from which to develop solutions.

The ability to introduce small, manageable, and highly focused contexts is an advantage of this approach. If the required focus is a single construct, then a context can be chosen which will best illustrate its behaviour. For example, when introducing the repetition construct, the concept of average could be employed. Constructing a simple unit that finds the average of a set of numbers can easily be written using a repetition construct but will also result in a useful code segment that can be incorporated into programs that are more complex. Combining individual constructs leads to simple units; combing simple units leads to more complex units. An example of this is a unit that finds the maximum of a set of numbers. To implement this, a repetition and a conditional are required. Again, this simple unit can be used in more complex logic. Using these simple, but meaningful contexts also helps develop problem solving and debugging strategies. It is easier to identify a logic error if the outcome is well known, i.e. the maximum number.

As with the building blocks approach, interactive learning requires mastering some complex tools at the same time as learning to construct the simple units. This is often a hindrance to beginner programmers. In addition, the introduction of a context does not remove the necessity of mastering the behaviours of the individual constructs, so some overlap with the building blocks and/or code analysis approaches may be unavoidable. Furthermore, mastering the problem solving skills necessary to resolve the set tasks must occur simultaneously with learning the tools and mastering the block behaviours. As the objectives of the approaches become more complex, so do the set of skills necessary to address the complexity. In a worst-case scenario, this approach may degrade to repeated cycles of differing combinations of simple units, code analysis, and building blocks.

By combining basic constructs into simple units that solve small well-defined problems, learners can create toolboxes of useful and reusable code fragments. Although a defined language and a development environment are required for this approach, it has the advantage that the resulting units provide a starting point for learners to begin writing code solutions. The useful skill of decomposition, breaking a problem down into small solvable parts, is also developed using this approach.

Full Systems

In the full systems approach to teaching programming, learners are immediately immersed in the full use of language constructs and tools. This approach involves a non-trivial problem solution. Programming concepts and language constructs are introduced as needed to solve the problem. Four studies report success when using a full systems, problem-solving approach in object-oriented and visual language environments. Although it might seem counterintuitive given their complexity, real problems and their full solutions motivate some learners. This approach, in common with some of the above approaches, requires mastering many skills and tools at the same time.

The full systems approach is analogous to learning a language by immersion. This type of learning is encountered by learners in foreign language classrooms where only the target language is used, in English speaking classrooms where learners understand limited English, and often when living in or visiting a country where the native language is not understood. Learning by immersion usually requires the leaner to master reading, listening, speaking, and writing all simultaneously.

In the full systems approach, learners design or help design a solution to a non-trivial problem. The

programming concepts and language constructs are introduced only when the solution to the problem requires their application. For example, a 2-player game of tic-tac-toe could serve as an introductory problem. To produce this, learners would need to demonstrate skills in decomposition (understanding the problem), handling inputs from the keyboard, displaying outputs in some format, tracking turns (variables/assignments), determining availability of position (selection/conditional), and identifying a winning move (repetition). The language constructs, show in ( ), would be introduced only when that part of the problem needs to be solved.

Four different studies have indicated success in using a full systems approach, incorporating problem solving, to teaching programming. Duke, et al (2000) employ this approach in their beginner classes with Java. In order to hide some of the object-oriented complexities, they provide a set of customised classes for their students. In one assignment, the students were asked to implement the game of 4-in-a-row. The constructs the students were expected to master were the use of repetition and Boolean expressions to maintain a system's internal logic. The students reported that this approach allowed them to learn Java effectively. The objective for Nuutila, et al (2008) is acquiring system design skills. For their students, they set tasks such as a robot in a maze. Students are expected to decompose the problem, design classes, and develop algorithms to create an effective solution. Interestingly, the actual production of the solution is not always part of the task objective, although practice using real programming tools is also included in the course. Storytelling with Alice, a 3D visual programming world, is the context chosen by Sattar and Lorenzen (2009) for introducing their learners to programming using a full systems approach. Students are presented with a set storyboard scenario and expected to develop the code to animate the story. Concepts (object, world, and scene) and constructs are introduced when needed to progress the animation. Another example of a full systems approach is used by Campbell and Bolker (2002). They employ immersive techniques and ask their beginners to read and alter a bank ATM simulation. Their approach focuses immediately on interfaces, architecture, and design. The syntax of the chosen programming language is addressed only when needed. They admit that this approach is not easy, but it places greater emphasis on design and skills than on mastering syntax.

Although it may appear that this approach should be daunting for the learners, the advantage is that they can be presented with real problems for which they should already have conceptual models, for example, a vending machine or a telephone billing system. In addition, they may feel motivated and empowered that they are learning to program a system that is representative of real life. They can also attribute the mastering of small steps to the eventual solution of a real problem. For example, the vending machine code may need to be modified to give the correct change from a £2 coin. If the learner can make this change, then the full system has become more functional and effective.

By definition, the immersion or full system approach requires mastering many skills simultaneously. The programming environment tools, individual block behaviours, interactions between blocks, and debugging must be confronted together. This could be demoralising for some beginners, as Lui et al (2004) has identified. From a teacher's perspective, the problem selection must be well considered to fulfil the underlying requirements of mastering the necessary constructs and acquiring the skills to work in the programming environment.

Although it may seem counterintuitive, the full systems approach may motivate some learners. Being able to understand how changes to code make a solution more effective can quickly move some learners on to further investigation. Real problems, in an already known context, may help develop more refined problem-solving skills, such as decomposition. Although, in common with the simple units and building blocks approaches, a defined language syntax and development tools must be mastered, this approach has the advantage of working on non-trivial solutions right from the beginning.

Suggested Combinations of Approaches

Effective programmers have some level of skill in explaining, tracing, and writing programming code. Each of the four previously described approaches to teaching programming allow development of one or more of these skills. Combining different approaches provides for progression toward becoming an effective programmer. Some possible combinations of approaches are illustrated in Figure 1. Different toolsets lend themselves to different paths around the model, based on the capabilities and needs of the learner.

The ability to write code requires some skill in tracing and explaining code (Lister, Fidge, and Teague 2009). Each of the four approaches identified above encompasses development of one or more of these skills. Effective programmers exhibit skills in code analysis (tracing and explaining), in understanding block behaviours (tracing and explaining), in constructing simple units (writing), and in combining simple units to create full systems (writing).

By combining the approaches to teaching programming, it is envisioned that learners will be able to evidence

progression toward becoming effective programmers, regardless of their age and capabilities. Figure 1 is an attempt to represent possible orderings of these approaches that could facilitate progression. The linear approach from building blocks to simple units to full systems is perhaps the most obvious line of progression as it represents stepped movement in complexity. However, the reverse path is also a type of progression and can represent the acquisition or deepening of knowledge by the learner.

Orders of approaches presented in this model are dependent only on the starting point. A final decision about the approaches used and their order of presentation should be based on the requirements of the course, the age appropriateness of the programming environment or language, the capabilities and prior experiences of the learners, and the confidence of the teacher.

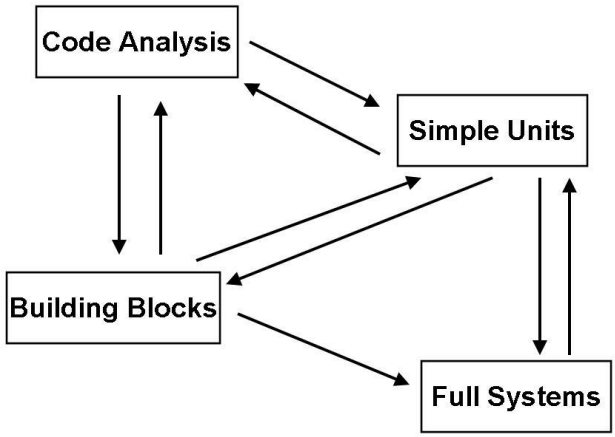

Figure 1: Ordering of Approaches Showing Progression

The use of a visual programming environment such as Scratch can be used to illustrate the approach of building blocks, simple units, and full systems. In this environment, the learner gives instructions, represented by puzzle pieces, to cause a visual object to move around a stage and interact with its environment. The behaviours of a subset of the available puzzle pieces could be introduced and demonstrated to the learner. After mastering the behaviours of this subset, the leaner could put the blocks together to control the actions of the object. Once a sufficient number of these simple units have been constructed, they could be assembled to create a full system. An objective for a full system using this type of environment could be a short animation or a simple interactive game.

Visual environments also lend themselves to the order: simple units, building blocks, full systems. In the Alice programming environment, learners instruct a visual object to move around and interact with other visual objects represented in the world. Introducing simple units first might entail the learner being supplied with several objects possessing predefined behaviours. The simple units, in this case methods belonging to the Alice objects, could be combined to create interactions between objects. Progression toward building blocks could be accomplished by being able to understand and being able to modify the code in a simple unit to change the interactive behaviours between the objects. Once units can be constructed and the blocks can be understood, learners could create a full system, such as a story animation or a simple interactive game.

Full systems, as described above, may be the most challenging approach for learners, but by giving them access to a fully or partially functional system from the beginning, the students may be more motivated to engage with the learning. A functional system, for which the logic is well known, such as the arcade game Asteroids, could be presented to learners directly in the programming environment. This could be achieved in an environment such as Greenfoot, which is based on the Java programming language. New or modified behaviours could be explored by changing the code in a simple unit or by introducing new simple units. For example, a simple modification might require five hits before the asteroid explodes or a more complex introduction could involve growing asteroids when they collide. As cautioned above, this approach entails simultaneous learning of problem decomposition, a programming environment, language syntax, and debugging strategies.

Another option incorporates code analysis, building blocks, simple units, and full systems. This order has the attraction of exposing the learners to programming concepts in the analysis phase. The code analysis could be based on the use of pseudocode similar to a text-based language. When attempting to master the building block constructs of the chosen language, the learners could be asked to use a programming environment appropriate to that language. Because the tools for developing in text languages are targeted

toward professional programmers, learners can find the mechanics of program creation challenging (Kölling, et al 2001). One programming environment based on Java that may address this issue is BlueJ. The next step, simple units, can give learners the opportunity for constructing parts of their toolbox of reusable program segments. For example, useful segments might include sorting a list of numbers or validating an email address. A full system gives the learner the chance to incorporate real problem solving with the use of blocks and their simple units. At this level, a full system might consist of a logic game, modelling the behaviour of a vending machine, or a quote generation system.

Conclusion

Upon reading published literature devoted to the learning of computer programming, common teaching approaches began to emerge. The characteristics common to these approaches were analysed and refined to fit into one of four identified approaches to teaching programming: code analysis, building blocks, simple units, and full systems.

Some research (Lister, Fidge, and Teague 2009) indicates that that, in order to write code, the learner must possess some skill in tracing and explaining code. Learners' development of these skills can be enhanced by engagement with any or all of the previously defined approaches to teaching programming. Progression toward becoming an effective programmer could be evidenced by movement between the approaches: analysing code to divine logic, understanding block behaviours of language constructs, constructing simple units of useful and reusable code, and combining simple units to create full systems.

A visual representation of possible progression through the different approaches is presented in Figure 1. The final choice of approaches and their order of presentation should be based on the requirements of the course, the age appropriateness of the programming environment or language, the capabilities and prior experiences of the learners, and the confidence of the teacher. Regardless of the starting point, movement through the approaches provides learners with opportunities to progress and construct new knowledge.

Further research is needed to better understand these and other approaches to teaching programming, not in terms of learner outcomes, but in terms of teachers' actions and techniques employed to facilitate the construction of new knowledge by the learners. Effective classroom teaching practices could be informed by further investigations into the effect on progression of different toolset choices and combinations of teaching approaches.

References

ACKERMANN, E. 2001. Piaget's constructivism, Papert's constructionism: What's the difference? *Future of learning group publication MIT* [Online]. Available: http://learning.media.mit.edu/content/publications/EA.Piaget__Papert.pdf.

AL-IMAMY, S., ALIZADEH, J. & NOUR, M. A. 2006. On the Development of a Programming Teaching Tool: The Effect of Teaching by Templates on the Learning Process. *Journal of Information Technology Education,* 5**,** 12.

BEN-ARI, M. 1998. Constructivism in computer science education. *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education.* Atlanta, Georgia, United States: ACM.

CAMPBELL, W. & BOLKER, E. 2002. Teaching programming by immersion, reading and writing. *Frontiers in Education, 2002. 32nd Annual.*

DEPARTMENT_FOR_EDUCATION. 2011. Review of the National Curriculum in England: Remit. Available: http://www.education.gov.uk/schools/teachingandlearning/curriculum/b0073043/remit-for-review-of-the-national-curriculum-in-england [Accessed 26-04-2011].

DUKE, R., SALZMAN, E., BURMEISTER, J., POON, J. & MURRAY, L. 2000. Teaching programming to beginners - choosing the language is just the first step. *Proceedings of the Australasian conference on Computing education.* Melbourne, Australia: ACM.

GLASER, H., HARTEL, P. H. & GARRATT, P. W. 2000. Programming by numbers: a programming method for novices. *The Computer Journal,* 43**,** 253-264.

KÖLLING, M., QUIG, B., PATTERSON, A. & ROSENBERG, J. 2001. The BlueJ System and its Pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology,* 13.

KÖLLING, M. & ROSENBERG, J. 2001. Guidelines for teaching object orientation with Java. *SIGCSE Bull.,* 33**,** 33-36.

LAHTINEN, E., ALA-MUTKA, K. & JÄRVINEN, H.-M. 2005. A study of the difficulties of novice programmers. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education.* Caparica, Portugal: ACM.

LEUTENEGGER, S. & EDGINGTON, J. 2007. A games first approach to teaching introductory programming.

Proceedings of the 38th SIGCSE technical symposium on Computer science education. Covington, Kentucky, USA: ACM.

LISTER, R., FIDGE, C. & TEAGUE, D. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education.* Paris, France: ACM.

LUI, A. K., KWAN, R., POON, M. & CHEUNG, Y. H. Y. 2004. Saving weak programming students: applying constructivism in a first programming course. *SIGCSE Bull.,* 36**,** 72-76.

MILISZEWSKA, I. & TAN, G. 2007. Befriending Computer Programming: A Proposed Approach to Teaching Introductory Programming. *Issues in Informing Science and Information Technology,* 4.

NUUTILA, E., TÖRMÄ, S., KINNUNEN, P. & MALMI, L. 2008. Learning Programming with the PBL Method. *In:* BENNEDSEN, J., CASPERSEN, M. E. & KÖLLING, M. (eds.) *Reflections on the Teaching of Programming Methods and Implementations* Berline: Springer-Verlag.

REGES, S. 2006. Back to basics in CS1 and CS2. *Proceedings of the 37th SIGCSE technical symposium on Computer science education.* Houston, Texas, USA: ACM.

ROY, P. V., ARMSTRONG, J., FLATT, M. & MAGNUSSON, B. 2003. The role of language paradigms in teaching programming. *Proceedings of the 34th SIGCSE technical symposium on Computer science education.* Reno, Navada, USA: ACM.

SAJANIEMI, J. & HU, C. Year. Teaching Programming: Going beyond “Objects First”. *In:* ROMERO, P., GOOD, J., CHAPARRO, E. A. & BRYANT, S., eds. 18th Workshop of the Psychology of Programming Interest Group, September 2006 University of Sussex. Psychology of Programming Interest Group, 255-265.

SATTAR, A. & LORENZEN, T. 2009. Teach Alice programming to non-majors. *SIGCSE Bull.,* 41**,** 118-121.

THE_ROYAL_ACADEMY_OF_ENGINEERING. 2009. ICT for the UK's Future: the implications of the changing nature of Information and Communications Technology. Available: http://www.raeng.org.uk/news/publications/list/reports/ICT_for_the_UKs_Future.pdf [Accessed 26-04-2011].

WOLF, A. 2011. Review of Vocational Education - The Wolf Report. Available: http://www.education.gov.uk/publications/standard/publicationDetail/Page1/DFE-00031-2011.

WULF, T. 2005. Constructivist approaches for teaching computer programming. *Proceedings of the 6th conference on Information technology education.* Newark, NJ, USA: ACM.