Flexible Low-cost Activities to Develop Novice Code Comprehension Skills in Schools

Peter Donaldson
School of Education
University of Glasgow
Glasgow, Scotland
peter.donaldson.2@glasgow.ac.uk

Quintin Cutts
School of Computing Science
University of Glasgow
Glasgow, Scotland
quintin.cutts@glasgow.ac.uk

ABSTRACT

The lack of code comprehension skills in novice programming students is recognised as a major factor underpinning poor learning outcomes. We use Schulte's Block Model to support teachers' understanding of how to break the skill down into component parts that are more manageable for a learner. This analysis is operationalised in three code annotation-based learning/assessment exercise formats, two helping students to identify and describe programming concepts and the third enabling them to parse code correctly and carry out desk executions. A great benefit of the activities is that they are low cost and can be applied to any imperative style code and so can be easily adopted by schools anywhere; furthermore, they are active, not passive, an issue with some animation-based visualisation approaches. The exercise formats were included as part of a national schools computing science professional learning programme (PLAN C).

CCS CONCEPTS

• Social and professional topics~Computer science education • Social and professional topics~K-12 education • Social and professional topics~Student assessment

KEYWORDS

Notional Machine; Block Model; Program Comprehension; Formative Assessment.

ACM Reference format:

Peter Donaldson, Quintin Cutts. 2018. Low-cost Activities to Develop Novice Code Comprehension Skills in Schools. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education (WiPSCE'18). ACM, Potsdam, Germany, 4 pages.* https://doi.org/10.1145/3265757.3265776

1. INTRODUCTION

This short work-in-progress paper outlines the rationale for, and design of, three exercise formats that can be used in school classrooms to aid code comprehension. The formats are independent of any particular language and can even be delivered as paper-and-pencil exercises to reduce complexity, cost and enable flexibility to suit teachers' needs. This is timely since computing education, programming in particular, is rapidly expanding into the secondary school sector and even into primary. Schools traditionally have relatively low levels of resourcing to support their teaching; there is no standard teaching language used by all schools, neither across nations, nor sometimes even within a single city; and there is no universal curriculum or set of lesson plans. Finding relatively language agnostic exercise formats that can be quickly and cheaply tailored to specific languages and contexts is therefore crucial. The formats presented here were included in a nationwide teacher professional development programme (PLAN C) [5] and incorporated by teachers into Computing Science courses for learners mainly in the 14 to 16 age range.

WiPSCE '18, October 4–6, 2018, Potsdam, Germany

https://doi.org/10.1145/3265757.3265776

2. RATIONALE

The development of code comprehension skills is increasingly viewed as a key developmental stage in becoming a competent programmer [11–13, 21]. Benedict du Boulay is generally recognised as the first to refer to code comprehension when he introduced the concept of a *notional machine* as an issue in learning to program [6]. This is one of five overlapping potential sources of difficulty for novices that he defined, another of which is *notation*, the syntax and semantics of a particular programming language. Du Boulay brings these two together in his definition of a notional machine as "an idealised conceptual computer whose properties are implied by the constructs in the programming language". The importance of novices' developing such a machine model is the consequent ability to see a program as a *white box*, the operation of whose statements may be understood, rather than as a *black box* around which only inputs and outputs are visible. In particular, by observing a large number of these white boxes, for a range of different programs, novices can start to disassociate individual constructs from the particular contexts in which they appear, a key learning step, following the *no-function-in-structure* principle [10].

Du Boulay suggested that a concrete tool would be valuable to enable the machine model to be observed. Sorva [18] developed such a tool in software, UUhistle, which makes visible a notional machine model for Python. Others have also developed computer and paper-based tools to help students develop notional machine models. For example, Berry and Kolling [2] have developed a paper or animation-based extension to the BlueJ environment for

visualizing Java programs; Holliday and Luginbuhl's [8] use of memory diagrams highlights incorrect learner models of how data is stored in memory, and Hertz and Jump's [7]trace-based model for memory allocation and update models both a stack and an object heap.

Notation and the notional machine [16] form a key part of code comprehension, and are captured in the two *Structural* dimensions of Schulte's Block Model [14], as shown in Table 1. The model considers the understanding of code at four levels of detail. The Atom level, which considers programs at the level of smallest program components, may be exemplified by a single assignment statement, or an input or output statement, or even the

expressions contained within these components. The Block level considers contiguous lines within a program that are bounded in a logical way, for example the body of a loop or a branch in a selection statement, or a subprogram body. The Relations level considers separated lines of code that are related together: this could be relevant to variable roles [17], where, for example, an one-way flag variable typically has an initialization, a test in a loop header, and an assignment statement within the loop body; alternatively more complex relational behaviour might consider function calling between code blocks, and parameter passing. At the top level, Macro Structure concerns the understanding of the whole program.

The Block Model, across all four levels, is divided orthogonally according to two dimensions - Structure and Function.

***M*acro structure**	Understanding the overall structure of the program text	Understanding the "algorithm" of the program
***R*elations**	References between blocks, e.g.: method calls, object creation, accessing data	Sequence of method class, "object sequence diagrams"
***B*locks**	'Regions of Interests' (ROI) that syntactically or semantically build a unit	Operation of a block, a method, or ROI (as sequence of statements)
***A*toms**	Language elements	Operation of a statement
	***T*ext surface**	***P*rogram execution (data & control flow)**
	"Structure"	

Table 1: Structure aspect of Schulte's Block Model

They reflect the dual role that any program has, and the need to understand the program in either of those roles. The Function dimension concerns the way a program addresses the problem it is designed to solve. From the point of view of code comprehension, how does this line, this block, these components contribute to solving the whole problem? Which parts do they play? By comparison, the Structure dimension is associated principally with the syntax of the code, and is hence referred to as the "text surface"; and also with the operation of the code, what it does with respect to some machine model – this is "program execution". As noted, text surface and program execution are directly related to two of the five issues raised by du Boulay- notional machine and notation.

In this paper, we are principally concerned with building novices understanding of the Structural dimension of the Block Model given the increasing evidence that a combination of identifying, tracing and describing code strongly influence novice's ability to explain its purpose and write code of their own [11–13, 20, 21].

3. THE EXERCISE FORMATS

The exercises presented here were designed with the following aims

- Reducing the high-level of cognitive load of code comprehension faced by novices [15, 19] by distributing cognition between the mind of the novice and the code as it is annotated [4] and reducing the number of details that need to be maintained in working memory [9].
- Overlaying information on, or near, the code itself to avoid novices having to split their attention and coordinate between two completely different representations.
- Able to be focused and targeted at different levels and aspects of the Structural dimension of the Block Model. This means that assistance can be provided with parsing, dynamic control flow and expression evaluation, as well as variable tracking.
- Largely language agnostic so that they could be shaped to suit a teacher's particular context. For widespread adoption of improved teaching methods, this aspect is crucial.
- Considering Chi and Wylie's learner engagement model [3], ensuring that the learner is constructively engaged in the activity, enabling deeper learning than via just listening or observing, and preparing for even deeper interactive engagement via group or classroom discussion.

3.1 Code Identification and Description

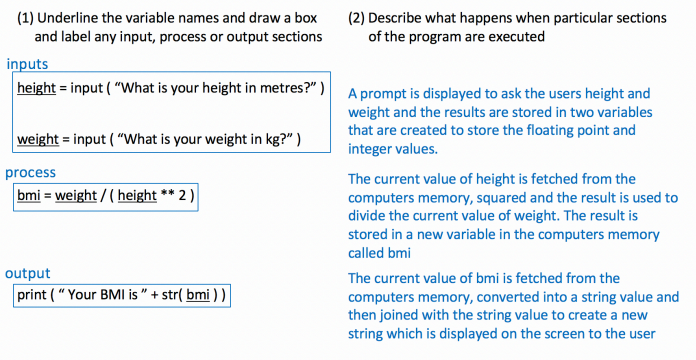

Figure 1: Python 3 identification & description exercise for a simple sequential program

The first two exercise formats presented directly target the *understanding* of code required across the various levels and two dimensions of the Block Model. They can either be used separately or in combination. A reference-sheet of programming concept terms is also provided to aid learners in identifying particular aspects and describing their execution.

- *Code Identification* (CI format) can be used to target the Atom, Block and Relations level of the Text Surface dimension. An example is shown in the exercise labelled (1) in Figure 1. Code identification involves the learner taking a code fragment and highlighting specific programming concepts. For example, they might highlight all variables, or expressions at the Atom level; or they could determine the extent of looping and selection control structures, identifying sub-sequences inside, at the Block level; or they could be identifying variable roles, noting related uses of a single variable across a program, at the Relations level. We
- note that code highlighting is not new, and is probably a technique in use by many educators already. Here we aim to show how it can be used to explicitly meet the breadth of code comprehension learning aims prompted by the levels of the Block Model.
- *Code Description* (CD format) helps to develop the core understanding of what code constructs do, hence relating to the Program Execution dimension at any level. A typical exercise here requires the learner, for atoms on one line or blocks covering multiple lines in a program, to write or say the name of the construct(s) involved and give a description of its operation. Such a task exercises a blend of Text Surface and Program Execution skills. An example for this in shown in the exercise labelled (2) in Figure 1.

3.2 Augmented Tracing using TRACS

The third exercise format, TRACS, is an augmented tracing technique acting principally as a significant cognitive assistant for developing the skills required in the Program Execution side of code comprehension. The tracing process is broken down into clear stages, to help avoid cognitive overload, and the final representation captures all aspects of the learner's notional machine model [16].

First, two finished examples of the technique are given, along with an explanation to show how the technique operates. Remember, though, that a learner would construct an example like this step by step, thoroughly exercising their understanding. After that, a number of observations are given about the design of the technique and how it meets the needs of the Block Model.

Figure 2 on the next page shows a completed trace using the TRACS tracing model on a very simple program expressed in Python 2, executed against the input data 5 and 7. The program is as follows:

```
value1 = input()
value2 = input()
total = value1 + value2
print total
```

In this case, the trace has been developed using a drawing tool, to ensure its readability at the reduced size for this paper, whereas in normal classroom use it is expected that it should be completed using pen and A4 (roughly Letter) or A3 (double Letter) sized paper.

Here are the three main steps to create a TRACS trace:

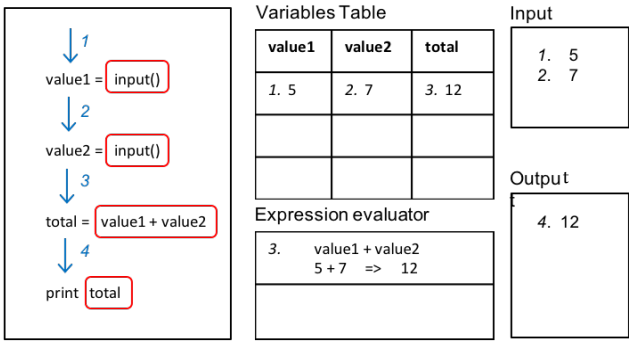

Figure 2: TRACS trace of a simple sequential program

1. Boxes are drawn round each expression in the code provided in the left-hand side. These are shown in red in Figure 2.
2. Arrows are drawn between the lines to represent the static control flow. If the code contains conditional branching points with two outgoing arrows, these are marked with T and F to indicate which route should be followed at run-time, depending on the calculated value of the Boolean expression. An initial arrow is drawn to the first line to be executed.
3. Execution now begins. The arrow leading to the next line to be executed is numbered with a step number. Lines of code executed repeatedly will attract a series of step numbers. The steps below are carried out for each line as it is executed:

3.1 As with a typical trace table, if a new variable is created, a new column is used in the variables table. Note that every update of the variable in the table, including the initialisation, is numbered on the left with the step number at which the update occurred. In Figure 2, no variable is updated after initialisation, so we only see one entry for each.

3.2 Input and output are recorded in the relevant boxes to show at which step number the operations occurred, and what values were involved. Here, the value 7 was read in on step 2, and the value 12 written out on step 4.

3.3 When an expression is encountered, if it is just a manifest constant value, it can be used directly. If it is a simple variable reference, the variables table can be consulted to find the value. If it is an I/O operation, like input, then the appropriate value is retrieved as in the previous step. If a calculation is required, it is copied into the expression evaluator, for example on step 3 for the right-hand side of the assignment. The step number is noted, and values for variables in the expression are substituted in from the variables table, and the resulting value determined. This can then be used in the surrounding statement.

A teacher can use this format to demonstrate to students how particular constructs operate – giving them a precise reference model for future use.

The teacher can then prepare a number of examples for students to work on. Printed sheets can be prepared with the code in place and the right number of entries in variables table and expression

evaluator, or else the teacher can display the code on a projector and let them draw up the whole model on paper.

A more complex instance of the technique is shown in Figure 3 for the following program:

```
total = 0
nextInput = input()
while nextInput != -1:
   total = total + nextInput
   nextInput = input()
   print total
```

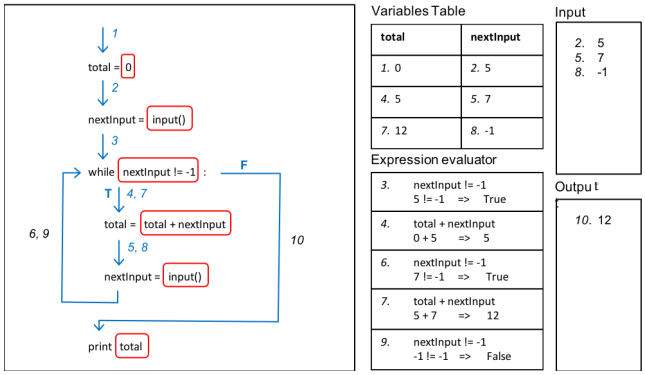

Figure 3: A more complex TRACS trace

The following observations are noted about this representation:

- Student's understanding of how expressions are different from statements is exercised, as is their knowledge of static control flow (Atom and Block levels of the Text Surface dimension). Taking place prior to the dynamic execution of the code, this should relieve cognitive load in the execution phase.
- The full dynamic control flow information is recorded in one representation, via the sequence of step numbers, as well as the data flow in the form of the variable updates and expression evaluation (Atom, Block and Relations level of the Program Execution dimension).
- This record can be used by a teacher to check for correct understanding. A quick scan of the step numbers adorning the program shows whether the student's control flow was correct. At the first sign of an incorrect sequence, the expression evaluations and variable updates can be checked by the teacher to diagnose the exact cause of the problem and fed back to the student.
- This is a learning exercise to be used to ensure that students are developing a consistent and correct mental model of how code executes [1]. Once students have internalised aspects of how code executes this level of annotation is unlikely to be required.

4. CONCLUSIONS

These exercises have been used by hundreds of teachers following their introduction in a professional development programme. While we do not have rigorous experimental data on their efficacy yet, questionnaire and interview feedback indicates that teachers have seen significant learning gains and changes in how their pupils talk and reason about programs compared to their prior methods that didn't have as strong a focus on code comprehension.

5. REFERENCES

[1] Ben-Ari, M. 2001. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*. 20, 1 (2001), 45–73.

[2] Berry, M. and Kölling, M. 2016. Novis: A notional machine implementation for teaching introductory programming. *Learning and Teaching in Computing and Engineering (LaTICE), 2016 International Conference on* (2016), 54–59.

[3] Chi, M.T.H. and Wylie, R. 2014. The ICAP Framework: Linking Cognitive Engagement to Active Learning Outcomes. *Educational Psychologist*. 49, 4 (Oct. 2014), 219–243. DOI:https://doi.org/10.1080/00461520.2014.965823.

[4] Cunningham, K., Blanchard, S., Ericson, B. and Guzdial, M. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. (2017), 164–172.

[5] Cutts, Q., Robertson, J., Donaldson, P. and O'Donnell, L. 2017. An evaluation of a professional learning network for computer science teachers. *Computer Science Education*. 27, 1 (Jan. 2017), 30–53. DOI:https://doi.org/10.1080/08993408.2017.1315958.

[6] Du Boulay, B. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*. 2, 1 (1986), 57–73.

[7] Hertz, M. and Jump, M. 2013. Trace-based teaching in early programming courses. (2013), 561.

[8] Holliday, M.A. and Luginbuhl, D. 2003. Using Memory Diagrams When Teaching a Java-Based CS. *ACM Southeast Conference*. (2003), 6.

[9] Kirschner, P.A., Sweller, J. and Clark, R.E. 2006. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist*. 41, 2 (Jun. 2006), 75–86. DOI:https://doi.org/10.1207/s15326985ep4102_1.

[10] de Kleer, J. and Seely Brown, J. 1981. Mental Models of Physical Mechanisms and Their Acquisition. *Cognitive Skills and Their Acquisition*. Taylor & Francis Group.

[11] Lister, R., Fidge, C. and 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* (2009), 19–26.

[12] Lister, R., Fone, W., McCartney, R., Seppälä, O., Adams, E.S., Hamer, J., Moström, J.E., Simon, B., Fitzgerald, S., Lindholm, M., Sanders, K. and Thomas, L. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *ACM SIGCSE Bulletin*. 36, 4 (2004), 119–150. DOI:https://doi.org/10.1145/1041624.1041673.

[13] Lopez, M., Whalley, J., Robbins, P. and Lister, R. 2008. Relationships between reading, tracing and writing skills in introductory programming. *Proceeding of the fourth international workshop on Computing education research - ICER '08* (Sydney, Australia, 2008), 101–112.

[14] Schulte, C. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. *ICER '08 Proceedings of the Fourth international Workshop on Computing Education Research* (2008), 149–160.

[15] Siegmund, J., Peitek, N., Parnin, C., Apel, S., Hofmeister, J., Kästner, C., Begel, A., Bethmann, A. and Brechmann, A. 2017. Measuring neural efficiency of program comprehension. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017* (Paderborn, Germany, 2017), 140–150.

[16] Sorva, J. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)*. 13, 2 (2013), 8.

[17] Sorva, J., Karavirta, V. and Korhonen, A. 2007. Roles of Variables in Teaching. *Journal of Information Technology Education*. 6, (2007), 407–423.

[18] Sorva, J. and Sirkiä, T. 2010. UUhistle: a software tool for visual program simulation. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli, Finland, 2010), 49–54.

[19] Vainio, V. and Sajaniemi, J. 2007. Factors in Novice Programmers' Poor Tracing Skills. *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education* (Dundee, Scotland, 2007), 236–240.

[20] Xie, B., Nelson, G.L. and Ko, A.J. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. (2018), 344–349.

[21] Yamamoto, M., Sekiya, T., Mori, K. and Yamaguchi, K. 2012. Skill hierarchy revised by SEM and additional skills. (Jun. 2012), 1–8.