Developing Computational Thinking through Coding

Weng Kin Ho
wengkin.ho@nie.edu.sg

Keng Cheng Ang
kengcheng.ang@nie.edu.sg
National Institute of Education
Nanyang Technological University
Singapore 637616
Singapore

August 30, 2015

Abstract

The turn of the 21st Century sees renewed emphasis on STEM (Science, Technology, Engineering and Mathematics). Building a Smart Nation is now a buzzword for many developed countries. Compelled by the advancement of information and communication technology, the ability to code in a programming language is a skill that is now, more than ever, urgently called for. In addition, it is widely believed that coding enhances problem solving abilities. In this paper, by underscoring the disciplinarity of coding, we construct a curricular framework for inculcating computational thinking in authentic classroom situations. Our proposition that computational thinking can only be imparted through doing, and this is supported by episodes of a graduate course, taught by the second author, in which graduate students develop coding competencies in VBA.

1 Introduction

The turn of the 21st century sees a renewed emphasis on STEM (Science, Technology, Engineering and Mathematics) globally, and consequently a re-alignment of educational directives that respond to this ([6], [1], [7], [12]). The transformation technology has brought about over the past few decades, be it the way we live, work or play, is tremendous. Notably, technological advancement manifests itself in the forms of industrial automation, digital communication and web-based information. In response to such advancement, it is only natural to anticipate that the future workforce be equipped not only with an *instrumental understanding* to operate technology but also a deeper *relational understanding* of how technology works and how new technology can be continually created – extending Skemp's terms used in mathematics learning ([11]) to STEM education.

Worldwide permeation of 'smart' mobile devices has now set the stage for a full-fledged integration of 'smart' technology into our everyday life. For Singapore, the call to build a 'Smart Nation' is no longer a futuristic aspiration from yesteryears' science fiction as this global buzzword is now rapidly turning into a reality. In his official speech at Smart Nation launch on 24 November 2014 ([10]), Singapore Prime Minister Lee Hsien Loong envisioned a Smart Nation as one "where people live meaningful and fulfilled lives, enabled seamlessly by technology, offering exciting opportunities for all. ... where networks of sensors and smart devices enable us to live sustainably and comfortably". In the same speech, schools were specifically urged to expose students to Information Technology and basic *coding* skills.

Education plays a critical role in nurturing the 'smart' generation to be ready for the digital world. Different countries are responding, in varying degrees, to this urgent need to teach their students the new *discipline of computing*. Some countries took a large-scale change; for instance, the United Kingdom has taken a radical (and applaudable) approach, through the pioneering efforts of Computing At School (CAS), to implement computer science as a school subject in the primary school curriculum in 2015. Some countries adopted small-scaled approaches; the Ministry of Education in Singapore, for instance, launched a new national programme, known as *Code for Fun*, which involved some 12 secondary schools and 4 primary schools. While helping to demystify the notion that coding is challenging, the Fun for Code pilot programme exposed pupils to "the use of algorithms and logical reasoning to solve problems by breaking them down in a fun way" ([13]).

The word 'discipline' associates to long-term values rather than short-term 'skills'; i.e., a discipline is characterised by "a way of thinking and working that provides a perspective on the world that is distinct from other disciplines" ([4, p.2]). We refer to this particular way of thinking as *computational thinking* – a coinage first put forth and expounded by Jeanette Wing ([14]).

What precisely is computational thinking? Though many scholars have proposed different interpretations of this over a growing body of literature ([2], [3], [5], [8]), there does not seem to be a single "crisp and meaningful definition" and yet, "the term clearly has resonance" ([4]). One thing stands in common to most interpretations: computational thinking is a paradigm of tackling large complex problems by systematically (i) breaking these down into smaller tasks, (ii) finding representations for them so as to make them tractable, and in most cases, (iii) creating efficient algorithms to solve them. Because the aforementioned three processes are intrinsically linked to the application of logic and creative thinking, one anticipates that computational thinking "shares elements with other types of thinking such as ... mathematical thinking" ([8]). However, to the best knowledge of the authors, such a link has not be explicated.

The conference theme "Celebrating Mathematics and Technology" for ATCM this year calls for a celebration of the success as a direct result of the synergy between mathematics and technology. Thus, the authors see no better opportunity than this to make explicit the link, perhaps for the first time, between mathematical thinking and computational thinking. By so doing, we hope to illustrate the strong interplay between the disciplines of mathematics and information technology. Furthermore, we aim to make use of this connection to craft a pedagogical framework for a workable classroom implementation that embeds computational thinking into teaching and learning of mathematics.

We organize this paper as follows. Since coding is a convenient vehicular activity that calls for computational thinking, we describe, in Section 2, the disciplinarity of coding. Based

on this description, we draw the parallels between coding and mathematical problem solving, and hence underscore the strong connection between computational thinking and mathematical thinking. The upshot here is to make clear the interaction between these two paradigms. This comparison is done in Section 3, culminating with a possible theoretical framework for infusing computational thinking into a mathematics curriculum. Finally in Section 4, we describe how this framework was implemented by the authors in a mathematics curriculum for a graduate course.

2 Disciplinarity of coding

It is difficult to divorce the thought processes and behaviourial patterns of a programmer from the coding activity because they are intimately interweaved together. So, we see no better way to expound on the disciplinarity of coding than to bring the reader through an excursion of the mind of a programmer who is actively engaging in some coding activities.

Here's a question on a type of number sequence, popularly known as the *Morris Number Sequence*, that is said to have been used as an interview question by companies such as Google (see [9], p. 96). Consider the sequence below and write down the next two terms.

$$1, 11, 21, 1211, 111221, 312211, 13112221, \ldots$$

It does not take too long for one to recognize that this is a visual sequence, i.e., the production of the terms in this sequence depends on a visual observation: the next term records the digits that occur in consecution within the current term and their corresponding frequencies of occurrence. For example, the second term records one 1 in the first term; the third term records two 1's in consecution in the second term; the fourth term records one 2 followed by one 1 in the third term, and so on.

2.1 Problem posing and solving

One natural question comes to your mind: Can you produce the nth term of the sequence? With a computational mind-set, the *Problem posing and solving* mode is activated:

1. Is it always possible to write down the nth term of the sequence for any given natural number n? How can one establish this possibility or impossibility without doubt?
2. By what means can we produce the nth term?
3. Can this job be performed by a computer?
4. How fast can this be done?
5. Can one determine efficiently whether an arbitrarily given string of digits is a particular term of this sequence?

Since the rule for generating the next term of this sequence makes use of the current term, the job of producing the nth term in this sequence is of a recursive nature. This indicates that we have a good chance to tackle Problems (1)–(3) collectively. We can rephrase these to a single problem:

Problem 1 *Write a program that takes an integer n and returns the nth term of the following visual sequence:*

$$1, 11, 21, 1211, 111221, 312211, 13112221, \ldots$$

A program can be thought of as a function p that assigns to each input x a unique output y, often denoted by $f(x)$. The origins of functional programming paradigm can be traced back to early development in theoretical computer science, such as the λ-calculus. This paradigm is essentially a style of constructing structure and elements of computer programs by treating computation as evaluation of mathematical functions; and, in particular, avoids state-transitions and mutable data. Some popular functional programming languages include LISP, SCHEME, OCAML and HASKELL. Apart from an excuse of convenience, we choose HASKELL because of its versatility in dealing with two important data types that will be invoked in solving the above problem: namely, the function type and the list type.

2.2 Understanding the problem

In any endeavor of coding a program that meets the required specifications spelt out in a problem, the programmer must begin with the phase of *understanding the problem* at hand. It is very common to see beginners in programming to be staring blankly at the computer screen or trying their luck by lifting codes blindly from textbooks or web-pages. This is not to say that there is no understanding at all – at least, a basic comprehension of the problem that the desired program is a function that admits an integer input and prints a 'sequence' output. Computer scientists call this type checking, i.e., in HASKELL we expect the desired function, denoted by `funseq`, to have the following type:

```
funseq :: Int -> [Int]
```

The functional style of programming codes every program as a function `f` that expects such input `x` and returns a unique output `f(x)`. As in the mathematical definition of a function, we expect that the domain and co-domain of the function to be specified. So, the program `funseq` takes in an integer argument (of type `Int`) and returns a list of integers (of type `[Int]`). The astute reader would have immediately recognize that we have employed a coding heuristic of choosing a *representation*. More precisely, a term in the sequence is represented as a list of integers; for instance, 13112221 is identified with `[1,3,1,1,2,2,2,1]`. Note that HASKELL dictates that the head of the list be defined as the zeroth term of the list, e.g., `[1,3,1,1,2,2,2,1]!!0` returns `1` but `[1,3,1,1,2,2,2,1]!!0` returns `3`. The list representation is more meaningful than the decimal representation because the odd positioned entries record the score (i.e., the symbols occurring in consecution within the previous term) and the even positioned entries record the corresponding frequency of occurrence.

2.3 Devising a plan

The problem at hand is still too complex, and computational thinking points towards breaking up this task into smaller ones. This can be invoked by *acting out* the rule which is used to generate the terms. We now demonstrate this particular coding heuristic. Suppose the current term consists of

```
[1,3,1,1,2,2,2,1]
```

The program must read the elements in the given list starting from the head `1`, proceeding through its tail `[3,1,1,2,2,2,1]`. As it scans the head `1`, it starts to check how many of this symbol occur in consecution. It does so by moving on to the next element and check for equality with the head. This movement to the next element and checking for equality with the head continues until the equality fails or when the list is exhausted, whichever comes first. The program then counts the frequency of occurrence of the head together with a record of the head. After this is completed, the preceding procedure is the applied to the first symbol in the tail which is different from the head, and so on till the entire list is exhausted.

By acting the rule out, the problem solving programmer identifies an important sub-task:

Problem 2 *Design programs to (1) determine the* depth of initial repetition*, i.e., the length of the longest initial segment of an input list consisting of consecutively repeating elements, and (2) record the element that is being repeated.*

2.4 Carrying out the plan

Carrying out the plan derived from 'acting-it-out', a program `counteq` (see Algorithm 3) which meets the specification (1) of the above sub-task is then scripted as follows:

Algorithm 3 (Determining the depth of repetitions)

```
counteq :: [Int] -> Int
counteq []         = 0
counteq [x0]       = 1
counteq (x0:x1:xs) = if (x0==x1) then 1 + counteq(x1:xs)
                                 else 1
```

Notice that `counteq` makes use of case considerations via *pattern matching*, i.e., it knows precisely what to do when presented with (i) an empty list `[]`, or (ii) a list with only one element `[x0]`, or (iii) a list consisting of more than one element. In the last case, a recursive call for `counteq` is invoked on a contracted list `x1:xs`. Hence the treatment is applied recursively on shorter lists. Together with the addition of 1, this recursive call implements a counter that calculates the frequency of the consecutively repeated elements which are equal to the head of the list. In other words, `counteq` returns the length of the longest initial segment of an input list that comprises consecutively repeating elements. For instance, applying the program `counteq` to the list `[0,0,0,1,3,3]` yields `3`. Running a *test* program on a number of data, though incapable of establishing the correctness of the program, is useful to check if it is incorrect; usually the bugs, if they exist, are detected at this verification check.

Meeting specification (2) is comparatively easy; this essentially calls for the in-built function `head` which when applied to any list of the form `(x0:xs)` decapitates it to yield `x0`.

The knowledge of how deep one needs to scan, starting from the head of the input sequence, before you hit a new symbol gives you a handle to break up the input sequence into segments each of which contains the same symbol such that no two adjacent segments have the same constant symbol. We call this intermediate program specification (3) the *segmentation* procedure for the moment. At this juncture, it is important to recognize that the desired output consists of a list of lists. To do this, it would be handy to fetch, from within an individual's coding

resource or otherwise that of some HASKELL community, an in-built function `splitAt` that does the job of breaking an input list into an ordered pair of two lists at a specified position. For instance, `splitAt 3 [0,1,2,3,4,5]` returns `([0,1,2],[3,4,5])`.

Coordinating all these together, one naturally arrives at the program `breakseq` performs segmentation (see Algorithm 4).

Algorithm 4 (Segmentation)

```
breakseq :: [Int] -> [[Int]]
breakseq []      = []
breakseq (x0:xs) = (fst (p) : breakseq (snd (p)))
  where p = splitAt (counteq (x0:xs)) (x0:xs)
```

So, applying `breakseq` to the list `[0,1,1,2,2,2]` expectedly yields `[[0],[1,1],[2,2,2]]`.

Now it is easy to make use of this output of segmentation to complete the task. To each member (a list, of course) of this list, we assign a list whose head is the length of the list (that would take care of the frequency of repetitions) and whose tail is just the symbol that appears in this segment. This *histogrammatic* assignment can be easily realized in Algorithm 5:

Algorithm 5 (Histogram)

```
histo :: [Int] -> [Int]
histo l = (counteq l:[head l])
```

For instance, `histo` applied to `[2,2,2]` yields `[3,2]` as there are 3 consecutive 2's.

Given a program `f :: a -> b`, one can lift this program to the function type `[a] -> [b]` at the list level using the following in-built program `map`:

```
map :: (a -> b) -> ([a] -> [b])
map f []     = []
map f (x0:xs) = (f(x0):map f (xs))
```

Applying `map` on `histo`, Algorithm applies the histogrammatic assignment throughout the segmented list of lists so that, for instance, `genfun` yields `[[1,1],[1,3],[2,1],[3,2],[1,1]]` when applied to `[1,3,1,1,2,2,2,1]`.

Algorithm 6 (Generating function)

```
genfun :: [Int] -> [[Int]]
genfun l = map histo (breakseq l)
```

Aware that the desired output is a list instead of a list of lists, we now flatten this list of lists using Algorithm 7 to yield the desired list:

Algorithm 7 (List flattening)

```
flatten :: [[Int]] -> [Int]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

Here, the syntax `++` is just the concatenation of two lists (into one). Thus, we can, for convenience, *compose* `flatten` with `genfun` using the HASKELL composition operator `.` by coding as follows:

Algorithm 8 (Operating function)

```
opvisseq :: [Int] -> [Int]
opvisseq = flatten.genfun
```

This program operates on an input list to yield the next term of the visual sequence.

At this stage, the programmer sees the light at the end of the tunnel because all the necessary tools for generating the next term of the visual sequence given a current term of the sequence. All needs to be done is to create an auxiliary program (Algorithm 9) which when given a natural number `k` and a list `l` returns the `k`th fold application of the operating function `opfunseq`, i.e., $f^{(k)}$, where f stands for `opfunseq` and k is the meaning of `k`.

Algorithm 9 (Auxilary program) `auxvisseq :: Int -> [Int] -> [Int]`

```
auxvisseq 0 l = l
auxvisseq k l = auxvisseq (k-1) (opvisseq l)
```

Now to obtain the kth term of the visual sequence, one just applies the above auxiliary program on the seed list `[1]`. The programmer finally arrives at the desired program (Algorithm 10) does the advertised specification described in Problem 1.

Algorithm 10 (kth term of the visual sequence)

```
visseq :: Int -> [Int]
visseq k = auxvisseq k [1]
```

2.5 Checking and extending

Looking back, we realized immediately that the main program `visseq` was constructed by building subprograms. This can be seen as a (local) version of *modularized* programming. While this programming style evolves from the basis of decomposing a complex problem into simpler tasks, it has a pleasant consequence that the programmer's job of *debugging* the program (in case it contains bugs) is made easier because of the possibility of problem isolation. Frequently, one needs to review the logical flow by relying on *flow charts* when programs are taken in isolation or *structure charts* when things are taken in entirety, e.g., in modularised programs.

As we have pointed out earlier, checking the correctness of each algorithms is a constant activity that a programmer engages up to the point when he or she is satisfied that the coded program does what it is supposed to do. Usually, this is accomplished by using random data checks. What is often missed out at this phase is that the programmer does not (or has no such ability) to reason about program correctness. By reasoning about programs, we mean a rigorous (in fact, mathematical) proof that affirms that the written code meets the given specification.

A crucial, but often overlooked, cornerstone of computer science is programming language semantics. In semantics, one gives meaning to the syntax by means of evaluating the terms, i.e., syntactically legal strings, to some mathematical universe. As an example, we can reason how the program `map` does its job. In the *denotational* approach, we can think of data types σ (e.g., `a`, `b`, ...) as certain kind of partially ordered sets[1]. The denotational model we use here is the *Scott domains model* invented by Dana Scott in the 1970s. Given that each data

[1]In the Scott model of computation, these partially ordered sets are called domains.

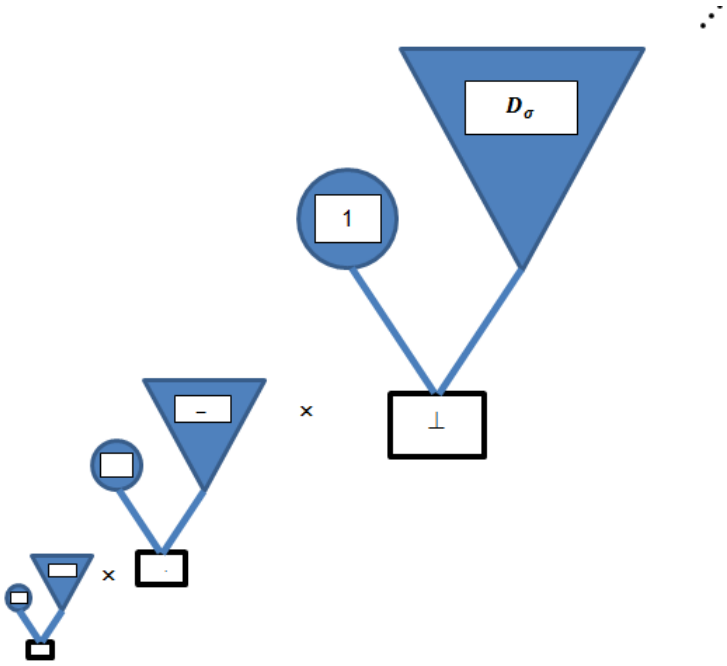

Figure 1: Data structure of list type $[\sigma]$

type σ is interpreted as some set D_σ, one then interprets the associated list type $[\sigma]$ as the set $D_{[\sigma]} = 1 + D_\sigma \times D_{[\sigma]}$. Here the data type 1 just contains non-termination, signifying that an element of a list may not return any output in finite time. The above recursive definition of $D_{[\sigma]}$ gives rise to the fractal (data) structure as shown in Figure1. In the above model, we can now reason out how `map` works. Given a program `f :: a -> b` which denotes the mathematical function $f : A \to B$, where $A = D_{\mathtt{a}}$ and $B = D_{\mathtt{B}}$. This function preserves certain order-theoretic structures of A. Thus, the program `map f` can be thought of as the mathematical function $[f] : [A] \to [B]$, where $[A] = D_{[\mathtt{a}]}$ and $[B] = D_{[\mathtt{b}]}$, and

$$[f] : (a_0, \vec{a}) = (f(a_0), [f](\vec{a})),\ (a_0, \vec{a}) \in [A].$$

Note that $[f]$ is the least fixed point solution to the above recursive equation because the Scott denotational model guarantees so. Moreover, the function correctly outputs as 'head' of the list the element $f(a_0)$, which is what `map f` is supposed to do. Though the precise mathematics is left out of this present description, it is still evident that a mathematical method is available to the programmer who wishes to know *for certainty* whether the program he/she has coded is correct. This then completes the process of *formal verification*.

3 Framework for computational thinking

The experiential session that we have just brought the reader through allows us to tease out various salient aspects of the disciplinarity of a programmer. In order to implement a pedagogically sound curriculum for computational thinking, a framework which illustrates the various fundamental components, together with a central theme, must be be called for. Informed of the

Notions	**Competencies**	**Procedures**	**Disposition**	**Metacognition**
data structure	mathematics	heuristics	precision	self-regulation
algorithm	error handling	problem reduction	perseverance	control
systems	code organization	problem decomposition	belief	awareness of requirements
database	code readability	error handling	interest	
languages	scripting	modularisation	confidence	
semantics	communicating	verification	knowledge of upcoming technologies	awareness of resources
automation	tool knowledge	build automation	integrity	
complexity	logical reasoning	system decomposition	creativity	
program logic	testing	documentation	resourceful	
recursion	debugging	defensive coding	meticulous	
security	judgement	encryption and risk assessment	analytical	awareness of environment

Table 1: Framework for computational thinking

peculiar disciplinarity of programming, we now give a sketch of what possibly such a framework may look like.

The heart of computational thinking is *problem solving*. This problem solving theme transcends through all levels and components of the activity of computing. To empower a computational thinker and problem solver, we propose that the five essential components are (1) Notions, (2) Competencies, (3) Procedures, (4) Disposition, and (5) Metacognition. This framework is somewhat reminiscent of the Singapore Mathematics Framework, except for the different items categorized under each component that are particular to the computational thinking.

Notions refer to key concepts salient to computational reasoning and coding, e.g., data structures and algorithms. Competencies are the core skills essential for a programmer when he or she engages in coding activities. Computational thinking exhibited through coding also involve procedures and processes, such as modularisation, and risk management. Coding calls for a special disposition characterised by precision, creativity, etc. Lastly, metacognitive or self-regulatory aspects are invoked in computational thinking.

We recommend that any lesson design that aims to encourage computational thinking, especially through coding activities, be crafted around the aforementioned five components. Because problem solving is central in this framework, coding problems should be designed with a focus of solving real-life problems.

4 Professional Development for Teachers

Despite the importance of coding and computational thinking and their relationship to the learning of mathematics, many mathematics teachers in Singapore are not taught computing or programming either as an undergraduate or as a student teacher. In fact, some of them sometimes feel overwhelmed by the abundant and rising use of technology in mathematics classrooms. In this section, we showcase how the framework for computational thinking developed in the preceding section can be implemented in authentic classroom situations. Here, we intentionally work with an imperative language VBA to illustrate language-independence in the teaching of computational thinking.

4.1 A course on computing for teachers

A course entitled "Computing and Programming Techniques" was made available to mathematics teachers in Singapore by the National Institute of Education (NIE). This course is an introduction to programming using a common programming language. The focus will be on writing computer programs for mathematical computations and problem solving. Topics include computer basics, data, statements, control flow and structures, arrays, functions and subroutines, recursive techniques, testing and debugging.

The broad objectives of this course include introducing participants to the basics of computing and programming for the purpose of mathematical problem-solving, exposing participants to the use of a common IT tool in performing mathematical computations and solving mathematical problems, including problems involving mathematical modelling, and developing the ability to construct solutions to problems on a common platform using the concepts and tools learnt.

The course serves to meet the need for a professional development programme in coding, with an aim to providing opportunities for mathematics teachers to link the essential skills involved in programming and computing to those required for mathematical problem solving.

In principle, the actual language used to deliver this course is irrelevant and any suitable programming language (such as Python, java, C++, MATLAB and so on) may be used. In 2014 when this course was last conducted, it was decided that the platform used would be Microsoft Excel's Visual Basic Applications (VBA). The main reason for the choice is that MS Excel is both available and readily accessible to most school teachers in Singapore. Any other choice may pose problems of either licensing issues or difficulties in accessibility. In addition, most, if not all, school teachers in Singapore are already familiar with using Excel as an electronic spreadsheet, and thus using VBA in Excel would be a natural and logical extension of their current knowledge and skills in this area.

The course was run over a period of 13 weeks, with three hours of contact time each week. Lectures were conducted in a computer laboratory, where hands-on activities may be carried out, and were sometimes supported with videos, especially on topics that involved basic skills. An example is a short video clip on using Excel to generate a scatter plot, which is available at : `https://youtu.be/bHncSzKaolw` .

Videos were also provided for to help participants with certain parts of the exercises. For instance, one of the exercises had involved constructing animations using Excel with VBA. Since animation was involved, a video showing the intermediate stages as well as the end product would be most helpful for participants to know what was actually required. An example of a

video for an exercise on developing a "projectile game" (not unlike the popular *Angry Birds* app) is available at: `https://youtu.be/oIS9bUOdmhc` .

4.2 Mini Project

Participants in this course were assessed continuously throughout the course. Besides the usual quizzes, which involve practical components, and a final test, assessment for this course also includes a small project which constitutes 30% of the final assessment. In fact, feedback from participants seems to suggest that this was the "highlight" of the course.

In this "mini project", the task is to develop an Excel VBA application worksheet using the techniques and skills covered in the course. The application should be sufficiently substantial to demonstrate appropriate and correct use of the programming techniques of VBA. This task is to be completed individually.

Participants are free to choose any topic or problem they wish to solve. However, the topic should fall in one of the following four categories.

1. Mathematical modelling (involving data manipulation, simulation, or animation)
2. Solving a mathematical problem
3. Mathematics Education (on teaching of some mathematical topic)
4. Games

The scope of the chosen problem should involve a good range of the techniques taught and through their application, participants are required to demonstrate the use of Form Controls, Subroutines and Functions, different data types, built-in VBA functions, some program control statements (such as the "if then else" construct, or the "for" and "while" loops) and some form of interaction with the user. In addition, the application should be properly designed to provide the best user experience. Participants were given about 6 weeks to complete this project; a written report and an oral presentation formed part of the overall assessment for this project.

Of the many mini projects that were turned in, two have been chosen to be discussed in this paper.

Example 1: Teaching of Probability

This project contains a few applications which simulate certain experiments that may be used in the teaching of probability, and has an attractive opening interface (See Figure 2). This project therefore falls under Category 3 (Mathematics Education).

As an illustration, clicking the "Tossing of 3 coins" brings up the worksheet in which the user may simulate the outcomes if three coins, possibly with some bias, are tossed. The worksheet shows the experimental probability as the simulation iterates and a simple tree diagram illustrates the concept further (see Figure 3).

From the VBA code written to run this application, it is clear that apart from coding skills, some demands of mathematical problem-solving skills were made. For instance, in constructing the simulation of tossing a *biased* coin, one has to know how make use of the usual pseudo-random number generator provided in VBA. In addition, the programmer also needs to check the output, and verify that the outcome is as expected.

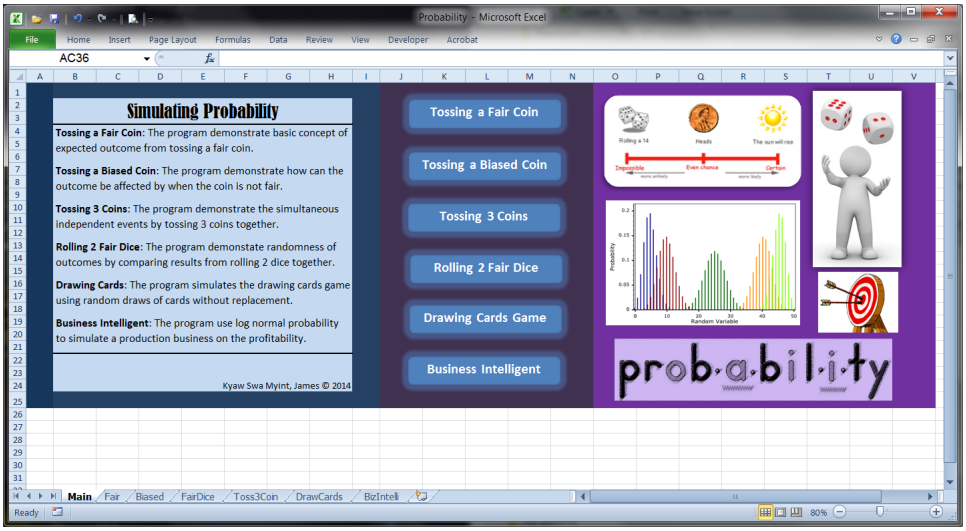

Figure 2: "Teaching of Probability" project interface

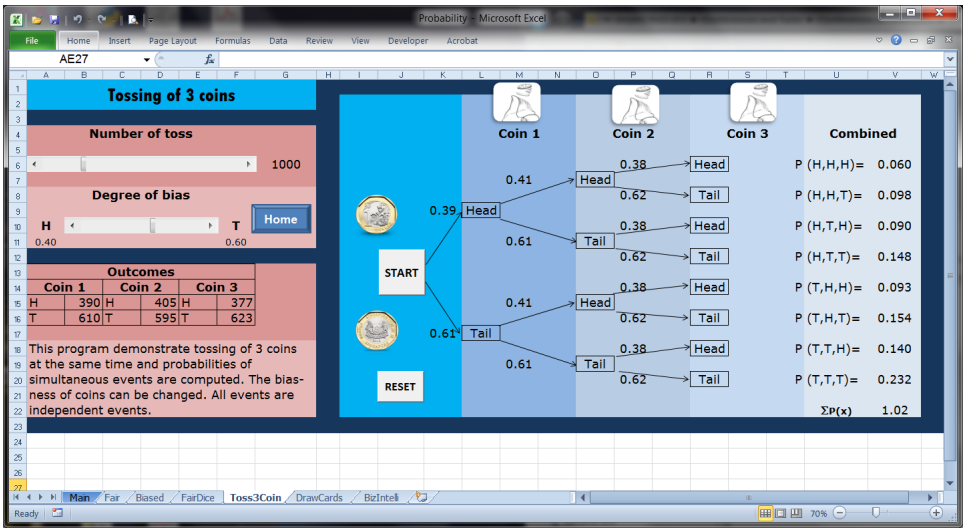

Figure 3: One of the applications within the "Teaching of Probability" project

This project involved running simulations to provide some form of verification of certain laws of probability. While constructing this application, the programmer will need to think about the laws of large numbers, present the experiment for human consumption, and conceptualize the whole experience for the user. In the words of Wing [14], *computational thinking* was very much in the thick of the action as the participant carried out this project.

Example 2: MineSweeper

This project uses Excel with VBA to implement the once popular and common game known as "Mine-Sweeper", and naturally falls under Category 4. The application allows the user to choose the size of the board (from 2×2 to 30×30), and the level of difficulty of the game ("Easy", "Hard", "Insane"). A screenshot of a typical game is shown in Figure 4

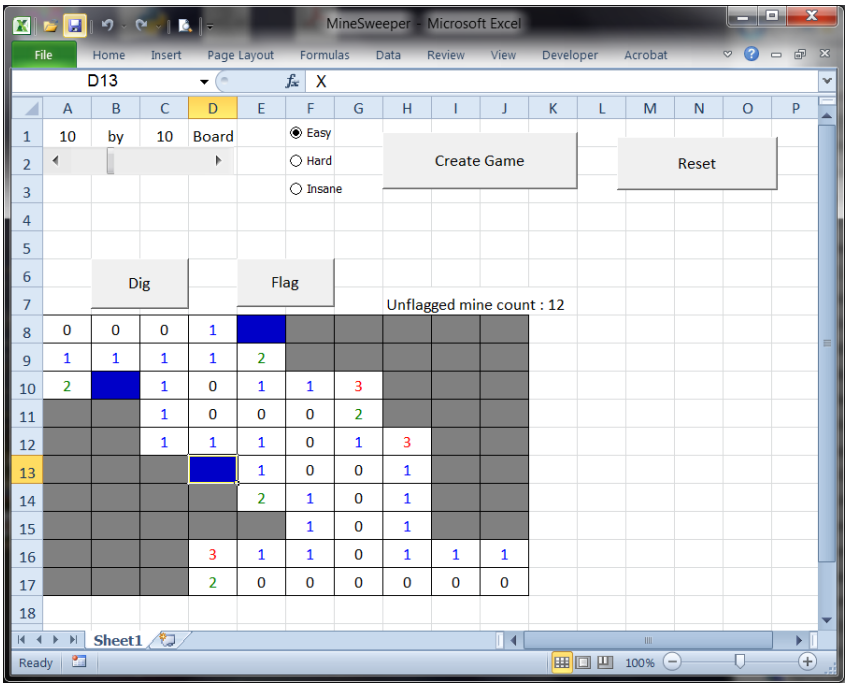

Figure 4: Screenshot of the "MineSweeper" project

To successfully create this game on VBA Excel, the code required will need to be carefully written to take into account the design of the board, keeping track of the variables, checking various conditions as the game progresses. The systematic and meticulous approach towards constructing this game on Excel with VBA is evident in the code submitted by the participant. This is also one important attribute that is not only related to mathematics learning but also one that is valued by mathematics educators.

From the examples presented above, we can see that the course has somewhat enabled participants to develop certain skills, attributes and competencies. What is equally important but not explicit in these examples is the fact that many of the participants have had to face initial struggles in completing this part of the course. However, almost all successfully produced a reasonably acceptable piece of work. In fact, out of 17 participants in the course, only one was deemed to have not met the standards required for this project.

More importantly, the course has successfully provided participants with the opportunity to learn programming or coding, albeit within the Excel VBA platform. Through the process, and

in particular the mini project, participants have acquired and developed their computational thinking, which, we hope, will in turn help them further develop their mathematical thinking.

5 Reflection and conclusion

Computational thinking is best learnt and taught through doing. By planning the curriculum around the proposed framework, the second author implemented a course in computing and programming techniques to a class of graduate students. Throughout the course, the students had ample chance to develop their computational thinking via coding. At the end of the course, students displayed competencies in computing and programming techniques and developed VBA application worksheets.

In this paper, we have highlighted the disciplinarity of coding by using an experiential walk-through with a programmer at work (with HASKELL). Based on this, we build a curricular framework that supports computational thinking in the classroom. Computational thinking can be developed through the practice of coding. This proposition is supported by episodes of a graduate course, taught by the second authors, in which students develop coding competencies VBA. The positive feedback from these students gave the authors a reassurance that the framework so drawn has indeed addressed certain salient aspects of computational thinking.

References

[1] Malaysia aims for 60 percent of children to take up STEM education - Najib (2014, September). *Bernama*. Retrieved from `http://english.astroawani.com/malaysia-news/malaysia-aims-60-percent-children-take-stem-education-najib-44509`

[2] Barr, V., & Stephenson, C., "Bringing computational thinking to K-12: what is involved and what is the role of the computer science education community?", ACM Inroads, vol. 2, no. 1, 2011, pp. 48-54.

[3] Bundy, A. Computational thinking is pervasive. Journal of Scientific and Practical Computing, Vol. 1, No. 2.

[4] Jones, S. P., Mitchell, B. & Humphreys, S., "Computing at school in the UK", CACM Report, 2013.

[5] Lu, J. J., & Fletcher, G. H. L., "Thinking about computational thinking," Proceedings of the 40th ACM technical symposium on Computer science education, ACM, New York, NY, USA, pp. 260.

[6] Mervis, J., Obama Advisers Call for Greater Emphasis on STEM Education (2 September 2010). *Science*. Retrieved from `http://news.sciencemag.org/2010/09/obama-advisers-call-greater-emphasis-stem-education`

[7] McComas, W. F., "STEM: Science, Technology, Engineering, and Mathematics", *The Language of Science Education*, 2014, pp. 102-103.

[8] Perkovic, L., Settle, A., Hwang, S. & Jones, J. (2010). A framework for Computational Thinking across the Curriculum, Proceedings of the 2010 Conference on Innovation and Technology in Computer Science Education, 2010, pp. 123-127.

[9] Poundstone, W., (2012). Are you smart enough to work at Google? New York, Little, Brown and Company.

[10] Transcript of Prime Minister Lee Hsien Loong's speech at Smart Nation launch on 24 November, 2014. *Prime Minister's Office, Singapore*. Retrieved from http://www.pmo.gov.sg/mediacentre/transcript-prime-minister-lee-hsien-loongs-speech-smart-nation-launch-24-november

[11] Skemp, R. R., "Relational Understanding and Instrumental Understanding", *Mathematics Teaching*, 77, 1976, pp. 20-26.

[12] Lee, P., Science, technology, engineering, math skills crucial to Singapore for next 50 years: PM Lee (8 May 2015). *The Straits Times*. Retrieved from http://www.straitstimes.com/singapore/education/science-technology-engineering-math-skills-crucial-to-singapore-for-next-50

[13] Teng, A., More kids to learn programming in Smart Nation push (2014, December). *The Straits Times*. Retrieved from http://origin-stcommunities.straitstimes.com/education/primary/news/more-kids-learn-programming-smart-nation-push

[14] Wing, J. M., "Computational Thinking", Communications of the ACM, Vol. 49, No. 3, March 2006, pp. 33-35.