ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #20938

# Improving introductory programming courses by using accurate metal models for the key abstractions.

**Mr. Robert A Ward IV, Everett Community College**

Robert Ward received his Bachelor of Science from Washington State University graduating summa cum laude. He served as the chair of WSU's ACM chapter and received the outstanding senior in Computer Science award.

He went on to his graduate work at University of Hawaii at Manoa and received his Master of Science in Computer Science. While at University of Hawaii he was a teaching assistant and developed a strong interest in Computer Science education specifically introductory programming.

After completing his master degree he was adjunct instructor at Kapi'olani Community College for introductory programming. He then went on to be a full-time instructor at Everett Community College in Everett, WA. He teaches introductory Java and C++ as well as advanced data structures.

©American Society for Engineering Education, 2017

# Improving introductory programming courses by using accurate mental models

Robert A. Ward IV
Everett Community College, robward@everettcc.edu

*Abstract* - **Computer programming has become increasingly important to most science and engineering disciplines. Unfortunately, introductory programming courses historically have a high failure rate. In addition, it is vital that computer programming be accessible to a broader range of students, and to provide a more diverse group of students the foundation necessary to succeed in programming.**

**The goal of this paper is to investigate solutions to improve the pass rate of introductory programming courses. These solutions should provide students with the foundation in the key concepts of programming that allow them to succeed in subsequent courses, should provide multiple practice opportunities to reinforce and automate skills, and should reduce cognitive load. This paper discusses several key abstractions, the mental model for each, and how the abstraction is connected to the operation of a physical computer. Examples of the exercises used to reinforce these models are discussed.**

**If we don't provide a learner with an accurate mental model, they will create their own mental model, which is often inaccurate. As they learn more about the topic, their mental model will fail, and they are forced to create a new model to accommodate the new information. Students confronted with the need to rebuild their worldview of how programs work often decide, "I just can't do this" and drop out. If, instead, students had an accurate mental model from the outset, they could more easily assimilate new information and ideas and extend the existing model. Because the new information would still fit with their existing mental model there is less frustration, less mental effort on the part of the learner and fewer barriers to continuing to study programming.**

**To provide introductory programming students with a sound foundation, we focus on providing accurate mental models for the basic abstractions of programming: variables, conditionals, loops, and function calls. Early in the introductory class, we introduce the concept of the fetch-decode-execute cycle to connect at a high level the operation of the CPU and program counter to the code. Each model is a given a direct connection to the deterministic nature and actual operation of a computer and to the machine code generated by source code. We couple this focus with multiple skill-building exercises on the abstractions.**

**For each key abstraction a simplified, but accurate, mental model is provided to the learner which must be simple enough for an introductory student to understand, while being accurate enough to allow the student to assimilate new information into the model as they expand their understanding of programming. To help students understand the mental model, skill-building exercises are done in class to reinforce the concepts and to provide skill automation that reduces the overall cognitive load required to program. Reducing cognitive load is vital to being an effective programmer as programming complexity increases. By automating performance of certain operations through repeated practice, the student can limit expending cognitive resources on those operations and can focus on learning new concepts or extending old concepts in new ways.**

## AUDIENCE AND GOALS

The goal of this paper is to introduce a new approach to teaching introductory programming courses which will be useful to those who teach introductory college programming courses, as well as to those who are designing and implementing secondary school curricula to prepare students for college engineering.

To help give students a solid foundation we must provide accurate, if simplified, mental models to students of how the key abstractions of programming work, so that learners will be able to extend their programming abilities without having the cognitive load of unlearning incorrect mental models they have created themselves. In addition, we must give them practice using these mental models to reduce the cognitive load when the students are called on to use them later in their studies.

Learners who are not provided a mental model will create their own mental model, which is often inaccurate. As they learn more about the topic their model will fail, and they are forced to create a new model or revise the existing one to accommodate the new information, which becomes increasingly difficult as their mental model differs more and more from the actual mental model. If they had been given an accurate mental model in the first place, they could assimilate new information as it came to light without recreating the mental model, which requires less mental effort on the part of the learner. This in turn creates less stress for students and less frustration that they just "don't get it and aren't cut out to be programmers."

To provide a sound foundation of understanding for the students in our introductory programming courses, we have added a focus on providing mental models for the basic abstractions of programming and repetitive practice with these models. Each model is also given a direct connection to

the deterministic nature of a computer and to the machine code generated by source code. The example basic abstractions discussed in this paper are: conditionals, loops, variable scope and function calls.

## PREVIOUS WORK AND THEORETICAL BASIS

Wulf looked at constructive pedagogical approaches to teaching programming [1]. He feels that by leveraging such approaches "resulting courses are accessible to a wider range of students and incorporate active learning." While I agree that such approaches have merit, I am cautious about focusing on the "higher cognitive levels of Bloom's taxonomy." I think we should first ensure we have built a solid base from which to proceed.

In Krathwhol's revision of Bloom's taxonomy, he discusses all the levels [2]. The higher levels, where learners analyze and evaluate, are built on the previous levels. Wulf seeks to reach these higher levels. Reaching these levels are long-term goals for our students, but they cannot attain them without progressing through Bloom's lower levels of remembering and comprehending. This is not straightforward in introductory programming, which requires the understanding of many complex and interlineated concepts to create even a simple program and for which many introductory students have no previous experience whatsoever.

Bloom himself suggested that each discipline should create its own taxonomy. "Ideally each major field should have its own taxonomy in its own language - more detailed, closer to the special language and thinking of its experts, reflecting its own appropriate sub-divisions and levels of education, with possible new categories, combinations of categories and omitting categories as appropriate."[3] Based on this we should look to create our own version of the taxonomy. Before we look to improve our student's abilities to analyze and create we must provide effective ways for them to remember and comprehend the world of programming and computing.

Mordechai Ben-Ari suggested that constructivism could serve as a solid approach to teaching computer programming [4]. He pointed out that this was a more active approach that required instructors to be aware not just of the content being delivered, but also of the current knowledge that learners would use to build new understanding.

Specifically, he suggested that, "A (beginning) CS student has no effective model of a computer." I agree with this assertion. He was specifically referring to the "object first" approach to programming, and thought that a model of a computer must be taught for students to be successful. He felt this was not compatible with an "object first" approach to teaching programming.

As he suggests, a model of the computer is vital and should be provided. I would go a step further and say that a model should be provided for all approaches, object first or not. It is vital that introductory learners have a viable model for all the key abstractions of programming, such as conditionals, loops and functions. It is irrelevant whether this is an object-oriented approach or not.

While I agree with the basic idea, Ben-Ari falls short by not discussing the general nature of the model or an example of a model. The model should be accurate enough to be extended later and simple enough for students to comprehend at their current level of understanding. The next section will discuss the basis for these requirements. Later in the paper, several mental models are provided of both the basic function of a computer and of some of the key abstractions: conditionals, loops and functions.

The reason that these abstractions must provide a clear and effective model is that students no longer have a basis to understand them. When Dykstra wrote his then famous letter to the editor: "go to statement considered harmful" [5], and introduced the term structured programming, everyone understood the issue he was trying to overcome. The overuse of go-to statements had created unmanageable, difficult to decipher code.

The creation of conditionals, loops and functions were very effective at abstracting away the use of go-to statements. So much so the people today are often unaware of how the code they write works. Understanding these key abstractions has become arbitrary and therefore difficult. We must include clear effective models for students with these key abstractions to provide the needed foundation to comprehend how these key abstractions work in a program running on a computer.

The educational psychologist Piaget theorized that there were two main avenues for people learning new information to update their schema: assimilation and accommodation [6]. When learning via assimilation the learner adds new information to an existing mental schema, because their underlying theory of how something works is accurate enough to allow simply updating their model to include the new information. In contrast, during accommodation a learner must revise or replace their underlying theory. While both methods are vital to be a successful learner, accommodation is more difficult for learners [7].

When people learn programming for the first-time they must create an entirely new schema for the world of programming, if one is not given to them. This can be a massive undertaking. It can lead to using accommodation (model replacement) more often than they are otherwise accustomed to, as they must continuously replace their model of how programming works when new facts invalidate a previous schema. This can be very frustrating and confusing for students. Therefore, our approach involves providing students with accurate, if simplified, schema from the beginning.

Donald Norman discussed this idea in terms of mental models [8]. Users of software have a mental model about how a system works. This model is the user's internal idea about how the machine operates, and it allows the user to interact with the system effectively. In his words: "These models need not be technically accurate but they must be functional."

Norman's point was that when using a system, you need to understand more than just the visible parts to use it. Often at least some understanding of how the machine does its job is vital both to learning how to use it and to using it effectively [8].

While some students who are introduced to programing create models that are both effective and accurate enough to be extended, many create models that are not. A key part of an introductory programming course should be to provide extendible accurate mental models. Norman refers to these models provided to the user as a conceptual model [9].

With this conceptual model as a starting point, students will develop and evolve their own version which Norman calls a mental model [9]. We also incorporate Norman's idea of explaining what is happening inside the machine in these mental models. The remainder of this paper will give several examples of creating such models.

### INTRODUCTION OF SIMPLIFIED, ACCURATE, EXTENDABLE MENTAL MODELS

For each key abstraction, a simplified, but accurate, model is provided for the learner. The mental model must be simple enough for an introductory student to understand, while being accurate enough to allow the student to assimilate new information into the model as they expand their understanding of programming, without the need to create a new mental model. To help students understand and internalize these mental models, repetitive exercises have been developed to solidify their understanding of them and to reduce the future cognitive load when the student must use it in more complicated problems in the future.

This remainder of this paper discusses several key abstractions, the mental model provided for the abstraction using the fetch-decode-execute cycle, and its connection to the operation of a physical computer.

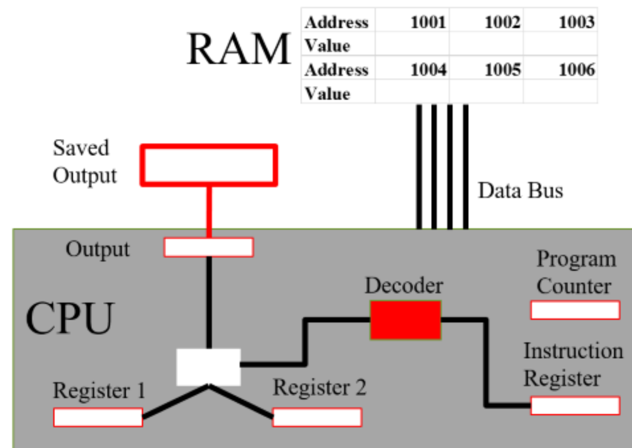
### FETCH DECODE EXECUTE CYCLE

To establish a basis for these mental models the concept of the fetch-decode-execute cycle is introduced early in the introductory programming class. The fetch-decode-execute cycle is connected to the operation of the CPU and the program counter. This is key to making the mental model connect with the actual operation of the machine.

Before this idea can be introduced, however, a learner must be aware of the build process. To effectively explain the fetch-decode-execute cycle, the process of loading the executable into RAM is discussed. This allows a clear discussion of the process of executing a specific instruction.

At this point we can introduce our first mental model. In it we provide a simplified model of a computer, including only RAM and the CPU with only a few registers. This simplified model can be seen in ***figure 1***. This model is not intended to be complete but rather complete enough to provide a basis for other concepts. This includes both related concepts like conditionals and a more complex model of the CPU's operation.

FIGURE 1
SIMPLIFIED COMPUTER FOR
FETCH DECODE EXECUTE MENTAL MODEL




At the same time, we introduce a simplified instruction set. Each instruction has three parts: the instruction followed by operand one and operand two. The instruction set and the parts of an instruction are shown in ***figure 2***.

The simplified instruction set provides only the ability to perform simply mathematical operations. Each instruction is comprised of an opcode and two operands. The goal is not to provide a complete model of how a CPU operates but instead provide a functional model that is extendable. The model should provide enough information to allow the learner to understand the basic operation but not so much that they are overwhelmed at their current level of understanding.

FIGURE 2
SIMPLIFIED INSTRUCTION SET

Instruction Set

| | Instruction |
|---|---|
| 0 | Add |
| 1 | Subtract |
| 2 | Multiply |
| 3 | Divide |

Instruction Format

| Instruction | Operand 1 | Operand 2 |
|---|---|---|
| 0 | 0 | 0 |
| Instruction put in IR | Operand 1 Put in R1 | Operand 2 Put in R2 |

To help students retain this model, simple exercises are provided that require students to load a simple program and show the changes to the registers in several sequences of the fetch-decode-execute cycle. This model emphasizes several key concepts to the learner, (a) the connection source code has with instructions, (b) the sequential nature of all programs, and (c) the importance of the program counter to the execution of a program. Examples of these exercises are included in the addendum to this paper.

Supporting future learning by allowing us to extend our understanding from previous knowledge is a key advantage to accurate mental models. As we will see in later examples, understanding the role of the program counter is essential to understanding conditionals, loops and functions.

## CONDITIONALS

Flow control in a program was originally done by conditional jumps in the program. Most modern languages have abstracted this away to remove the need for most jumps in source code (together with the confusion and spaghetti code that jumps created). However, jumps still occur in the machine code generated by the source code.

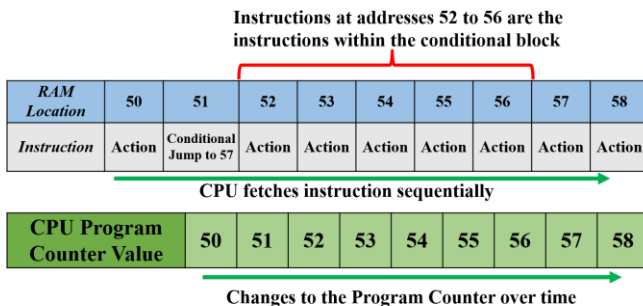
Today most learners are merely taught that the code block after a conditional is skipped if the Boolean expression associated with the conditional is false. Often students will create their own mental models of how this happens on the machine. Their model is often inaccurate and will need revision (accommodation) later in their development as a programmer. Instead of forcing students to make their own model, it is far better to give them a more accurate model. We therefore build on our previous example of the simplified CPU and connect the concept of a conditional to the program counter.

The following is the description of a simplified mental model provided for the learner, with diagrams to help clarify it. Visual learners especially might find that diagrams are invaluable to understanding the mental model.

A program is really a long list of machine instructions that reside in memory. One by one the instructions are fetched, decoded, and executed by the CPU. The program counter provides the location of the next instruction to fetch. Conditionals work by altering the program counter to skip one or more instructions. Changes to the program counter are done by machine instructions called jumps. While often referred to as branching, decisions are made by some form of jumping over a block of instructions by altering the program counter.

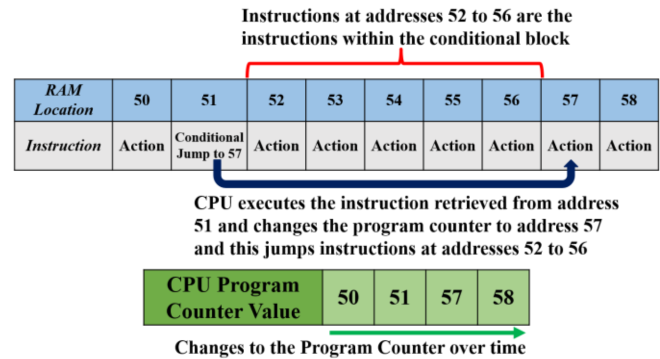
If the Boolean expression of a conditional is true, then the program counter merely increments as it normally does, increasing by one memory address at a time, as seen in ***Figure 3***. In the figure, the instruction **Action** represents a generic instruction whose specific effects are not material other than they do not alter the program counter. The first instruction inside the conditional code block is fetched.

FIGURE 3
INSTRUCTIONS FETCHED SEQUENTIALLY WITH TRUE EXPRESSION




If, however, the Boolean expression of a conditional is false, then the program counter is changed to the first instruction after the conditional code block. This alters the normal sequential nature of the change to the program counter, and the instructions within the code block associated with the conditional are skipped or "jumped" over as shown in ***Figure 4***.

FIGURE 4
INSTRUCTIONS NOT FETCHED SEQUENTIALLY WITH FALSE EXPRESSION




## LOOPS

Just like conditionals, loops in a program were originally done by conditional jumps. Most modern languages abstract this away, much in the same way they do with conditionals. Most learners are merely taught that the code block of a loop is repeated if the Boolean expression associated with it is true, without being given a model for how it works within the machine.

The previous model introduced for conditionals can also be used for the mental model of loops. This allows learners to leverage the previous understanding and reinforce the idea of the program counter and how the machine operates. The chief difference between the model for a conditional and that of a loop is that for a loop the program counter will often be set to a previous instruction. ***Figure 5*** shows the operation of a pretest loop when the expression is true and the loop continues.

FIGURE 5
PRETEST LOOP CONTINUES WITH TRUE EXPRESSION

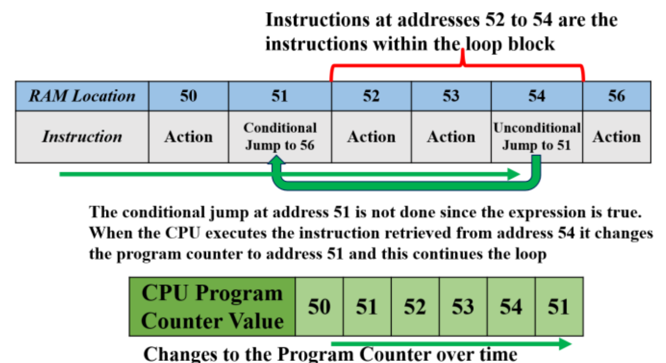
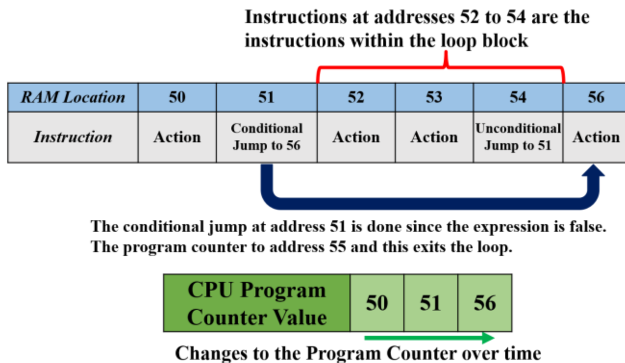


***Figure 6*** shows the operation of a pretest loop when the expression is false and the loop exits.

FIGURE 6
PRETEST LOOP EXITS WITH FALSE EXPRESSION




Again, we connect the changes in the program flow to the changes that occur to the program counter. This gives us another opportunity to connect the source code with the way the machine code operates on a physical machine. This repetition of the same underlying concept in a later lesson also provides spacing between these learning experiences, which will improve students' retention of the concept [10].

### VARIABLE SCOPE'S CONNECTION TO CONDITIONALS

Block variable scope related to the body of a conditional or loop is often a subtle and difficult concept for new programmers, because the reason for this scope is often difficult to connect to the students' previous knowledge. This can be ameliorated if students have a firm understanding of how conditionals work at the machine level.

Variables defined within a conditional block are only in scope inside the block. It will be much easier to explain why this is the case using the previous mental model for conditionals. By injecting a hypothetical variable declaration in the conditional block, the uncertainty of whether the variable was declared or not can be clearly demonstrated. If the conditional expression is true, the code after the conditional block could see it as a valid variable. If it was false, however, this code would be skipped, and the instruction declaring it would be skipped. There is no way for the later instruction to create the declared variable, therefore any variable declared within the body of a conditional or loop can only be valid inside that code block.

### FUNCTION CALLS

Another important concept introduced in introductory programming classes is the ability to create and use callable units (functions or methods). This idea is often misunderstood by the novice programmer, who then often creates inaccurate and difficult to understand mental models of the process happening when a function is called.

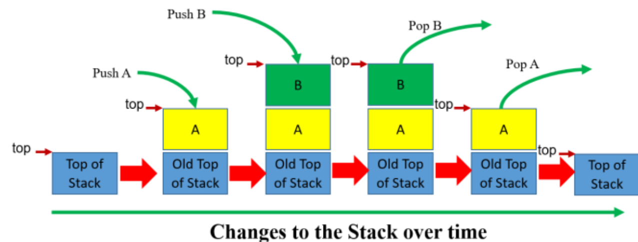
While this model requires more information than the ones previously presented, it still connects to them. Prior to explaining how a function is called, the idea of a stack and stack frame must be explained. Again, the goal of this is to provide only the information needed to understand the process of calling a function, not a full explanation of the stack.

To that end a stack frame is described as a block of data that contains (a) the memory location of the instructions of the function and (b) the local variables of the function. It should be made clear to the students that this is a simplification of a stack frame.

The idea of the program stack and its key functions should also be introduced. These functions include: keeping track of the current function that is executing, restoring the current instruction when you return to it, and passing information from one callable unit to another in the form of arguments and return values.

Now we can provide a simplified model of a stack to students, followed by the basic mental model of the program stack as shown in ***Figure 7***.

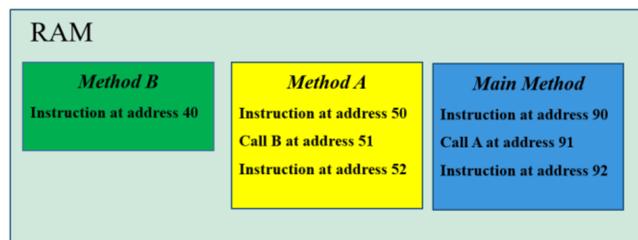
FIGURE 7
SIMPLIFIED STACK MODEL




The bottom item in the stack is always the main function. If the main function calls a function, a stack frame for that function is pushed onto the stack, and the address of the first instruction for that function is placed in the program counter. The main method waits for the called function to complete its job and give it the information it was designed to calculate (return value). In some cases, the function has no information to return, but it has a side effect (for example: print something). When a function completes its last instruction, the stack frame associated with that function call is popped off the program stack. The address of the instruction after the instruction that called this function is put in the program counter.

This model must be supported with a visual representation to be effective. ***Figure 8*** below provides a simple set of functions that can be used to demonstrate the operation of the program stack.

FIGURE 8
METHODS OF A PROGRAM IN RAM

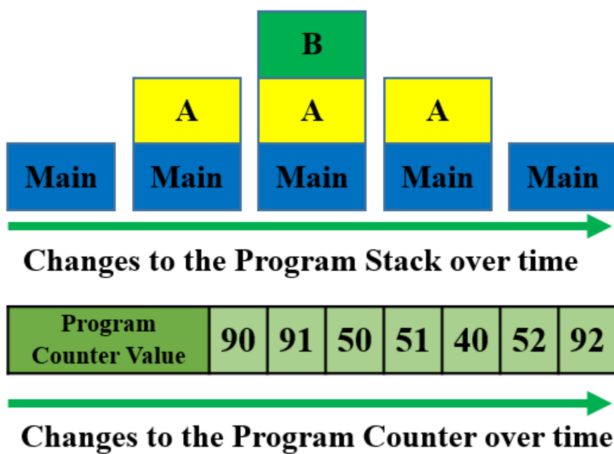


This mental model required the most additional information but it is also vital to a learner's understanding of how a program runs on a machine. It also supports other key concepts such as local variable scope and the processing cost of function calls. While at this early stage they may not fully understand this model, it provides an important foundation without which they will struggle as they continue to extend their understanding of programming.

The student's comprehension of this model can be reinforced by exercises like the previous example. Given a set of methods they are asked to show the changes to the program stack and program counter.

***Figure 9*** below shows both the changes to the program stack and the changes to the program counter based on the methods in ***Figure 8***. Example exercises are in the addendum to this paper.

FIGURE 9
CHANGES TO THE PROGRAM STACK AND PROGRAM COUNTER BASED ON METHODS OF FIGURE 7




**USE OF THE MODELS IN PRACTICE**

These mental models and others have been introduced and used in the CS0 class at Everett Community College for the past two years. Currently there is only anecdotal evidence to indicate they are effective, but they appear to have been useful in helping students be successful in the introductory and subsequent courses.

The course that first uses this approach is the prerequisite for CS 1 Java and CS 1 C++. Instructors for both courses felt that students who had been exposed to this approach in the previous quarter were more successful.

The CS 1 C++ instructor stated:

"*My CS 1 C++ classes in the winter quarter had a majority of students who had taken the CS 0 course using this method during the fall quarter. I noticed a night and day difference with these students in comparison to my previous students. The two areas that stood out with these students, was that they were competent with the basics of programming, but more importantly they were comfortable and confident with programming. This comfort level with programming freed them to continue to explore and grow in programming abilities*"

The author of this paper was instructor for two CS 1 Java sections. I also noticed a difference in ability between students from the CS 0 course using this approach and students from different CS 0 sections that did not use this approach. The students exposed to this approach had a very clear understanding of the basic concepts of conditionals, loops and functions. Many of the students from the other groups that did not use this approach struggled to keep up with the pace of CS1 and in general did not perform as well.

There are many differences in the approaches used in teaching the two groups of students. The instructor, the depth of coverage and the specific material covered could all be contributing factors to the difference in performance. Specific data would need to be measured to make a true comparison.

**SUMMARY**

The key to a mental model being effective is that it is accurate enough that later information can be integrated into it without extensive replacement or modifications to the underlying model. However, it must also be simple enough that it does not introduce excessive cognitive load that overwhelms the learner. Lack of simplicity would make it difficult for learners to understand the material and would be an impediment to them learning the original concept. This approach has been effective when used in introductory programing courses and is based on the concept of mental models developed by Donald Norman and of schemas developed by Piaget.

To avoid students creating their own inaccurate mental models we provide them with accurate, sometimes simplified, models. These mental models also demonstrate the connection between source code and the actions of the machine, a step often omitted in introductory programming courses. At the same time, these models clarify specific concepts that have been abstracted away in modern programming languages. Conditionals, loops and function calls are all accomplished at the machine level by changes to the program counter. By demonstrating this in a clear but simple model, a solid foundation is provided from which students can extend their understanding. This eliminates the confusion, effort and need for unlearning that is created by incorrect mental models.

While CPU architectures may change, the advantage of the models described here is that they functionally model the operation of a Turing machine. As Norman states “Models need not be technically accurate but they must be functional.” They should be an effective starting point for learners for the foreseeable future.

Further investigation of the long-term impact of effective mental models would be useful in future research. While the short-term impact may be significant, the impact over several courses may be even more substantial and impact retention of computer science and engineering students.

### REFERENCES


Wulf, T, "Constructivist approaches for teaching computer programming", *Proceedings of the 6th conference on Information Technology Education,* 2005 Oct 20. pp. 245-248.

Krathwohl, D. R., "A revision of Bloom's taxonomy: An overview", *Theory into Practice,* Vol 41, No. 4, 2002 Nov 1, pp. 212-8.

Anderson L.W., Krathwohl D.R., Airasian P.W., Cruikshank K.A., Mayer R.E., Pintrich P.R., Raths J., and Wittrock M.C., *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. 2001, pp.137-75

Ben-Ari, M, "Constructivism in computer science education", *ACM SIGCSE Bulletin,* Vol. 30, No. 1, 1998 Mar 1, pp. 257-261.

Dijkstra, E, W, "Letters to the editor: go to statement considered harmful." *Communications of the ACM, Vol 11, No.3,* 1968, pp. 147-148.

Piaget, J, Part I: "Cognitive development in children: Piaget development and learning", *Journal of Research in Science Teaching*, Vol 2, No. 3, 1964 Sep 1, pp. 176-86.

Piaget J., "Problems of equilibration", *Topics in Cognitive Development*, Vol. 1, 1977, pp. 3-13.

Norman, D, *The Design of Everyday Things: Revised and expanded edition.,* 2013 Nov 5, pp. 24-34.

Norman, D. A., "Some observations on Mental Models", *Mental Models*, Vol. 7, No. 112, 1983 May; pp. 7-14.

Kornell, N, "Optimising learning using flashcards: Spacing is more effective than cramming.", *Applied Cognitive Psychology*, Vol.  No. 9, 2009 Dec, pp. 11297-317.