Chapter 6
Why Is Teaching Programming Difficult?

Carlton McDonald

Abstract Teaching 1st year programming is a major challenge at all universities. It doesn't seem to matter what programming language is used, how much support is provided to the students, or how the students are assessed, or at which university the teaching and learning takes place. Learning to program is hard enough as it is (Pine, in Learn to program. The Pragmatic Programmers, 2009). Given that an A level in Computer Studies is still not a prerequisite for admission to a course teaching computer programming although A level Mathematics often is. We look at a number of issues around the difficulties of learning to program, the vast change in the nature of programming language concepts, libraries and application areas, and ask the question, are expectations of beginner programmers realistic given the short amount of time given to learning to program?

Keywords Continuous assessment · Formative feedback · Student motivation Portfolios · Learning to program · Teaching programming

6.1 Introduction

Teaching beginners programming is riddled with challenges:

- What's the point of programming?
- Motivating Students
- Why can't I dive straight into programming?
- Language Learning
- Cooperative learning
- Weekly, consistent learning

C. McDonald (✉)
University of Derby, Kedleston Road, Derby DE22 1GB, UK
e-mail: c.g.mcdonald@derby.ac.uk

J. Carter et al. (eds.), *Higher Education Computer Science*,
https://doi.org/10.1007/978-3-319-98590-9_6

- Incremental Learning
- Language Problems
- Assessment versus Learning.

This chapter looks at these challenges and how there is a good case to review the expectations of students particularly on degree courses, given the changes of programming languages, increased language complexity, and what can be learnt and applied in the contact time given to students new to programming. Programming skills are applied to web pages, web servers, mobile, distributed, cloud and systems programming to name just a few. However, many of the new application contexts need a new programming language in a browser context. JavaScript has a unique document object to become familiar with. It is neither obvious or intuitively understood, Programming the document object needs to be taught/learned. We look at the issues at each stage of the learning programming process up to undergraduate programming and some of the reasons that learning to programme is difficult.

6.2 What's the Point of Programming?

Imagine teaching origami. One of the first things you want to do is to show your students the things that can be produced with origami. Seeing these creative objects inspired some pupils to want to learn how to do it. As a seven or eight year old at primary school, the author remembers being excited by Origami and the different animals and objects that could be made. The discipline of folding the edges as precisely as possible, following written instructions, is effectively being a computer executing written instructions. The process of reflecting on origami exercises enables pupils to see the importance, not just of the quality of the instructions, but also of the accuracy of the execution. Many of the classmates struggled, at times, to follow the 2D paper based instructions attempting to illustrate folding movements, however the more one follows instructions, as difficult as they are at first, the easier they become to understand. Those that invest the time become experts at understanding the language in which the instructions are written. This is only half of the skill required to produce origami artefacts.

In addition to being able to understand the program, the robot following the program instructions needs to be able to meticulously and precisely follow the instructions. Those children that fold a square in half in order to make a fish or a bird, need to produce an accurate square to start with, and a precise diagonal fold in half, in order to have a reasonable looking and flapping bird at the end of the process of paper folding. These observations enabled the author to recognise the importance of following instructions carefully. The experience of following origami steps from books also occasionally meant that the occasional program didn't work because the instructions describing what tasks needed to be completed were unclear.

Traditional *sequential* programming is akin to writing the origami procedure in a step by step manner for a robot (the computer) to follow. The average programmer can manage to produce a program with less than 20 language constructs (if—then—else, while, for, arrays, procedures, classes, constructors, procedure and function calls etc.). This is much more than the Basic programming language (variable declaration, types, if—then—else, while, for, arrays, and procedures) that could be used to solve any computer problem, although not in the most efficient way. This relatively tiny set of instructions meant that programs took tens of thousands of lines of code, in the same way that a human language (like Tok Pisin, the *lingua franca* of Papua New Guinea) requires many more words to describe concepts than a language like English with its 100,000+ words.

The purpose of programming is to provide instructions to a machine to carry out a task or series of tasks. For example, book an airline flight, play a game, make a bank transaction, and billions of other tasks.

6.3 Motivating Students

The approach used by the author to get students excited about programming is to introduce the first lesson as an opportunity to program their own robot. This piques their attention. Students that have already started to nod off, wake up; those that are slumped, sit up excited with anticipation to see their robot. In preparation for programming their robot, they must first learn the simplest set of instructions that the robot will understand:

Forward *n* steps
Turn left
Turn right

The robot is to be navigated from any one corner of the classroom to the diagonally opposite corner of the classroom using a sequence of the above three instructions. It is surprising how these three instructions can be used to fulfil all navigation tasks in a single storey building, or even a city on foot.

6.3.1 Introducing the Robots

In order to see how good they are at programming these instructions need to be provided to a robot. Students are then paired off, if there is anyone that is not paired off they work with the tutor. Once they have settled down, they introduce themselves to each other and are notified that their partner is their "robot". They have to take it in turns to be programmed in an interpretive manner, i.e. receive instructions, one at a time, until they are not able to complete the instruction, or the program has completed.

The results of testing their robots indicate a number of issues. Firstly, is the need for initialisation instructions, e.g. stand up, face the front. These aren't in our very restricted language but all programs require initialisation statements. The second issue is how dumb robots are! Why can't they see that the desk they are about to walk into is going to cause them damage? Robots, computers, automatons do exactly as they are told by their programmers. If something goes wrong, it is always the programmers fault. This requires the programmer to revisit the program and edit the instructions. Programming robots to perform physical tangible tasks is comparatively easy in contrast to algorithmic intangible tasks where formulae and variable manipulation is required (see Variables later in this chapter).

6.3.2 Testing

How important is it to test the program? Having constructed the program, we cannot just assume that it will work. Testing, getting their robot to carry out the tasks called out to them is how the programmer is able to see if the program is ready to be distributed to all robots.

6.3.3 Single Solution

When asked, what are the limitations of their program, it is identified that the program only works in the room we are currently in. It is not a program that can be generalised to navigate any room. It is at this point that the purpose of decision making in programs is required. In addition to decision points it is also pointed out that we also need to be able to repeat tasks. We then go through a couple of iterations of modifying the program to add to the instruction set:

If condition is true **then** do task1
If condition is true **then** do task2 **else** do task3
Repeat task4 **while** condition is true

Programs are then modified, robots deployed, crashed, reviewed and students leave understanding programming languages, instructions, initialisation, testing and incremental development having had a fun first class.

6.3.4 Learning Attitudes

Most programmers are extremely confident and having written the program assume that it is correct after the first attempt. Many students are lazy. When asked to write their program, it is surprising how many of them have to be told to get out their pen and paper, tablet or laptop and put away their mobile phone! It seems that many of the students educated in the UK are just waiting for the tutor to provide the solution. This is a major problem.

Tell me, and I will forget, show me and I will remember, let me do it and I will understand.

This spoon fed generation just want the answer. Don't make me think! It is at this stage of the first class that those that are going to do well become apparent to the tutor. Programming is a practical activity, those that immediately set about writing the instructions are those that will be natural programmers. They are prepared to have a go, confident *they* can do it. They come into the class enthusiastic and are eager to start. When the program fails, they are completely undeterred, even more determined they can do it.

Other students, even though the task is straightforward and not a task that can only be completed by degree students, immediately start to look around to see what others are doing. It is as if they are not sure what they are supposed to be doing. After discussion, it emerges that what they want to know is there no instructions that say: "stand up", "turn away from the desk". They don't know how to initialise their robot. When told they can indicate some initialisation steps, this second group of students still continue to look around, typing/writing nothing until told to write something. It is surprising how tentative and uncertain these students are, as if they daren't get anything wrong lest they be expelled or imprisoned should anything go wrong. The third group of students are those that are in class but not involved in class. They seem reluctant to speak, reluctant to answer questions, reluctant to get into pairs and seem embarrassed as they pace out the instructions when it is their turn to be the "robot". Once in a while one or two of these students work well on their own and feel a little awkward in public. This third group of students are a challenge to motivate and will be the group that ask, "what do I need to do to pass"? "What do I need to do to get a first"? More interested in the qualification than the learning associated with the course.

6.3.5 Variables

One of the most difficult concepts to teach a complete beginner is the programming concept of a variable. Having attempted to teach 16 year olds programming in Papua New Guinea, the author spent a two hour lesson trying convey the idea of a variable. Why do we need a box to put things in? Was a question raised in trying to convey the concept of a storage place for things we want to remember in the program. Why

does the robot need a pocket to store a single item? why doesn't it just remember the value? The computer has memory doesn't it? The questions are a mass of confusion and are akin to the difficulties encountered by children attempting to learn algebra: "what are x and y for?", I asked my parents and they said they have never used simultaneous equations in their life, why do I need to understand it?

Unfortunately, it is true that not everyone who registers for a Computer Programming course, having never attempted programming before, will get past this stage. If they have never grasped algebra (even the word sounds alien, they think to themselves), it is an extremely challenging hurdle to overcome, because it involves abstraction: not having a physical or concrete entity. This is perhaps the reason that the course with the highest drop out rate of all UK degrees is Computer Science (Pine 2009). Admission to a Computer Science course therefore requires GCSE mathematics, not because programming is a mathematical activity but because GCSE requires the ability to think in an abstract manner, manipulating concepts in the mind, passing mentally through each instruction of the program with the aid of variable values on paper in order to determine where we are in the program and what is happening at each point in the program.

6.3.6 First Class Reflection

Programming robots has always proved a fun activity to get the class experiencing and reflecting on programming processes and concepts. Unfortunately, when the rest of the course is focussed on programming a machine, the kinaesthetic learning is perceived by some as simply typing on a keyboard. That's barely kinaesthetic, otherwise typing is the way in which kinaesthetic learners can learn anything, nevertheless writing programs in a computer laboratory is referred to as a practical! And that is what it is… to some students! Students build programs, which interact with users and produce results displayed on a screen. Kinaesthetic learners are able to make that association fairly easily. They find completing programming tasks quite fulfilling, whereas others just see text and pictures on a screen and it doesn't evoke positive feelings other than: "I have the marks required of this week's practical, and can still pass the module". For them programming is not a labour of love, just labour.

In the same way that with any practical skill, the more time that is spent doing the activity the better one becomes. Consider origami, playing badminton, driving a car… The real challenge for the programming lecture is to motivate those students that aren't natural problem solvers, that took a while to get into the robot programming and find that the keyboard programming is very unfulfilling. It is for this reason that it is the author's view that students are far more likely to spend time producing programs if those programs enable them to create programs for themselves or their family and friends to use. A tiny percentage of the world's population use DOS (whatever that is), Windows or programs that are executed on the command line. It is time that such introductory programming courses were commanded to be put on a line and executed. Everyone interacts with a browser, so although not a rich

instruction set, well structured, and limited Object style programming is available, JavaScript is the most important language in the world. Write JavaScript programs and everyone in the world is able to use your program. This inspires many student, whereas a C# program in 2018 seems hardly relevant to hand held environments from which people interact with software. Even server side programming requires a good grasp of JavaScript concepts.

In a similar vein, the web is a quarter of a century old, and mobile devices are ubiquitous, permitting access to a browser but if students are able to develop apps that they can show to family and friends this too might motivate them much more than DOS programs.

In the 1980's Computer Science purists were adamant: the only way to teach programming is to teach an assembly language. It is a while since such nonsense has been vocalised in a university but the Computer Science purists are now insisting that the only way to learn to program is with C++, C# and Java. Such arguments are way out of date and have some basis in the fact that many lecturers believe that the way in which they learnt is the best, the languages they are experts in, are the best for all time. It is amazing that the most profitable computing company in the world ditched one of these antiquated tools relatively recently. Apple devices were programmed in Objective-C, a language way ahead of its time when it was first created at the Stepstone company in the early 1980s by Brad Cox and Tom Love. It was licensed by Steve Jobs' NeXT Computer Inc. in the late 1980s but way behind the times when finally superseded by Swift in 2013. Yes, speaking to the die hard purists, an expert Objective-C programmer can make the program far more efficient than the best optimising compiler. However, the best Objective-C program could cause the greatest harm when memory access is to an unintended area. We need the best high level programmers, not low level programmers. When was the last time a modern developer wrote an assembler program? Only a very small percentage need learn this interesting programming paradigm. Nowadays we need the best web programmers and mobile programmers. As a result, the first environment in which introductory programming, in the module Principles of Programming, is taught is MIT's AppInventor. AppInventor was a Google Project, which they passed to MIT in 2012, and made open source (The Verge 2018) (Figs. 6.1 and 6.2).

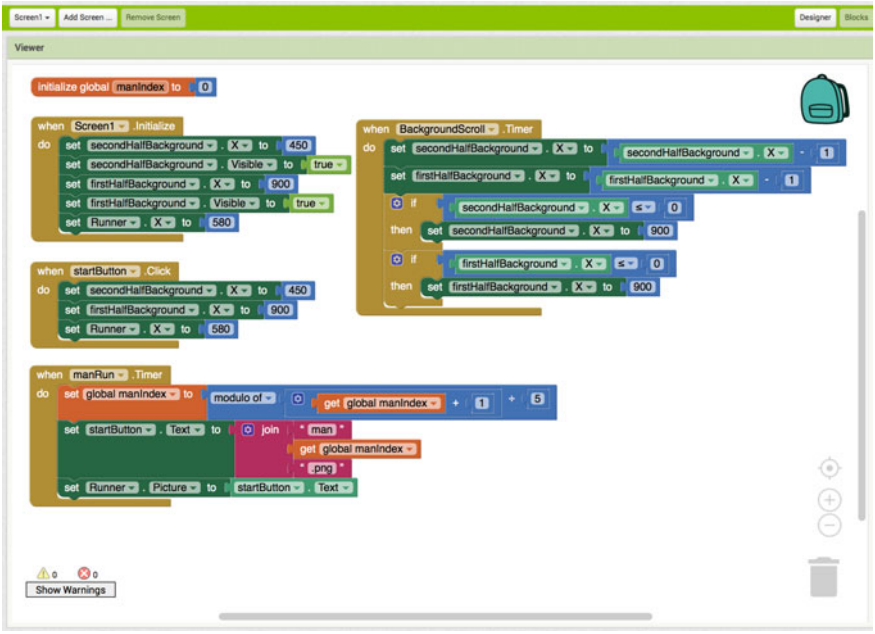
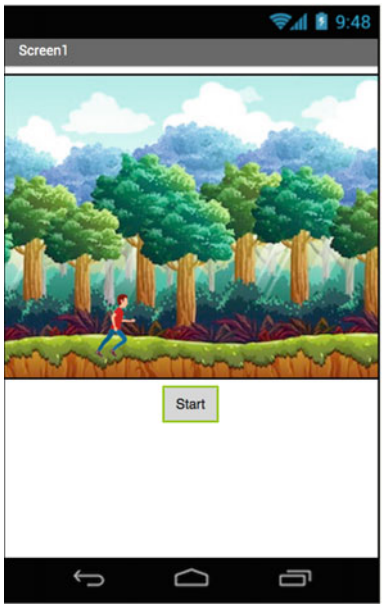


Fig. 6.1 A complete AppInventor program that animates a runner in front of a scrolling background

Fig. 6.2 Visual representation of the animated runner and scrolling background



AppInventor is a colourful jigsaw puzzle style programming environment for Android devices only. It is so easy to program Android devices that schools around the world are using AppInventor to teach mobile programming to very young children and upwards. AppInventor uses pieces to construct programs by dragging visual components and dropping them onto other visual pieces called "blocks" in a blocks editor.

At the time of writing, AppInventor is being ported by a couple of the original AppInventor creators Arun Saigal and WeiHua Li, to iOS devices through a platform Thunkable (Thunkable Thoughts—The How and Why You Would Make Your Own Beautiful App 2018a). The interface and programming constructs currently work, however, sensors, GPS, Accelerometer etc do not currently work in an iOS environment. Read about the exciting Thunkable platform developments here: https://blog.thunkable.com/ (Thunkable Thoughts—The How and Why You Would Make Your Own Beautiful App 2018b).

One of the best things about AppInventor is that there is a really easy method of designing User Interfaces or the visual representation of the app. It is the visual design that determines a high percentage of the blocks that are needed to construct the program.

Students require no real technical skills (other than drag and drop) in order to put the interface together and so in a short space of time they can have all screens designed before attempting to write the code for the app.

The feedback from the students is much more complimentary with regards to learning to program. Students still find it difficult to solve problems in the blocks editor where the programming is performed. With the logic of if-then-else and loops being the main hurdles. This is particularly difficult when these are "nested" within other if-then-else or loop statements.

AppInventor 2 (AI2) is not as good for learning to program as the original AppInventor Classic because AI2 does not allow the program to be debugged (going through the program step by step and examining values). This inability to debug, by manually stepping through the code, one instruction at a time, makes it impossible for students to learn how debug a program. Stepping through a program, step by step, statement by statement, is an excellent way of understanding the flow through a program. A beginner is able to go through the program step by step, over and over again, until they are able to predict what happens next. Debugging also allows the debugger to examine the values of variables, UI component states, values and sensors at each point in the program.

This is the only downside of using AppInventor. This lack of debugging was a serious factor as to whether to continue to teach AppInventor. However, the reason it was continued, is that students feel empowered to develop programmes at a high level and quickly. This makes AppInventor the ideal platform for Agile Development Methodologies. It is also perfect for scrums and sprints. Take a break, watch Agile in 5 min http://www.youtube.com (Krishna 2018), Watch Scrums and Sprints in 7 min: http://www.youtube.com (Youtube 2018).

In the early 1980s learning to program on mainframe computers was difficult because students could only practice at their place of learning. In the early 1990s

introductory programming was inevitably producing command line programs despite being in a Windows environment. This meant that students were producing programs that were not up to date as far as the beginner was concerned. Most universities used Windows Operating Systems, but Windows programming was not for the faint hearted, so students would often finish a degree and not have written a single Windows program. In the late 1990s the browser was the platform of choice for many users, browsing (or surfing as it used to be called) really took off. Nevertheless, few universities taught Applets or JavaScript as the introductory programming language so that the beginner could develop software for themselves, their family and friends. We were still producing command line programs.

AppInventor has changed all of that, students are able to design software for the ubiquitous Android platform. Some of them show what they have done to their family and friends and invest time, outside of class, practicing their skills. Others, too many others, do nothing from one class to the next.

Motivating students so that they want to come to the practical is most of the battle to learn programming. Ease of development of something meaningful in a few minutes rather than "hello world!" is much more of an inspiration to get out of bed. Has anyone noticed how many students struggle to get to class at 9:00 am? AppInventor, enables students to produce apps that can solve all sorts of problems that students can feel a real sense of accomplishment. A tiny number of absolute beginners, each year, really appreciate and express how much they love programming in AppInventor.

6.4 Why Can't I Dive Straight into Programming?

Students often dive straight into writing their program without thinking what they actually want to accomplish. This is like an engineer putting a big plank wood across a river and saying that he has built a bridge as he walks across the plank successfully. When he rides his motorbike across and ends up in the river below it shows the wisdom of "thinking before you program"! When learning to speak a new language, no one would attempt to construct sentences and paragraphs in the new language without first thinking what they would say, in their native tongue, and then translating that into the new language. Even if the new language is understood it is better to write a series of instructions to get to the supermarket for your Spanish neighbour. In programming terms this would mean writing in one's native tongue, in addition to drawing what one wants to happen in the program and then converting the written natural language instructions to programming language instructions. This early thinking, design stage, helps one to attempt to tackle each of the events that the user might generate, or environmental events (location, altitude, device orientation etc).

Students want to do things as efficiently as possible in terms of the time required by them to complete a task. They feel that trial and error is a good way of solving problems… after all it has served them well so far (with the exception of the rubic's cube which they went on to YouTube after the first unsuccessful attempt to see

how it was done). This approach results in the idea that it is better use of time to produce the program straight away rather than spending time designing it on paper because whatever goes on paper still needs to be transcribed into a computer program, this, for many beginners, is a waste of time, why not just write the program using a language that is unfamiliar, for a machine that is unfamiliar.

Even expert programmers put some ideas down on paper, helping to organise the difficult parts of the program in their head rather than producing code that may need to be reorganised. It is better, always better, to structure the program on paper. It is similar to planning the construction of building, any builder who can save you money by not "wasting time" with drawings, calculations and designs is someone you will find has lots of dissatisfied customers living in houses that still aren't finished, or with the bathroom opening out into the dining room. It works as a house but it clearly was not thought through the way in which the components of the house would fit together.

6.5 Language Learning

Learning a human language requires a number of stages unless one is born and hears the language from birth. The first stage is to hear the language over and over and over again. One firstly, begins to recognise a few words, followed by recognising the context in which they are used. Learning the grammar of a language is as important as learning the words making up the language. In programming, the syntax (grammar) of languages has continued to grow, with modern languages perhaps reaching 60 or 70 words and symbols, whereas early programming languages had perhaps as little as 20 words and symbols. Almost any program can be written with the following 3 components: sequence, selection and iteration. Sequence is the way in which one statement is followed by a second statement. Selection are the decision points in a program. All selection points in a program are effectively a combination of a number of if-then-else statements. However even the else is unnecessary if every statement is preceded by its antecedent:

```
if age > 18 then canVote = true
if age <= 18 then canVote = false
```

It is more efficient to use an else:

```
if age > 18 then canVote = true
else
    if age <= 18 then canVote = false
```

The reason using an "else" is more efficient, is that the second 'if' will only be performed if the first 'if' statement fails. We actually use the expression "executed" rather than "performed", this is frightening, subconsciously, to the beginner programmer, it slightly adds to the anxiety, particularly that of people that grew up when a personal computer cost £3000, and one daren't break something that cost as much as a used car.

A reason that some students don't make the transition to programming is whereas everyone learns the language of communication from birth, not everyone learns the language of logic or problem solving. Those students that enjoy solving problems find it easier to learn programming than those that never play strategy games, or solve puzzles. The skills can be learnt but many cannot see the point, they much prefer the answer to be provided to them than have the joy of solving the problem themselves. Most university computer science courses have a separate module covering mathematics, very few have a separate problem solving skills module.

Programming may be applied in numerous contexts: desktop, browser, mobile, server-side, distributed, cloud, server and system are the main areas expected of most Computer Science graduates in 2018. Eight different types of programming, few of which the students can be experts in given the time constraints of undergraduate degrees. However, there are other languages for AI and natural language processing, robotics, more libraries than one could learn in a lifetime, so the problem is not just learning a language as it was 40 years ago. The diversity of all of these application areas mean that the base from which the students need to build needs to be much wider than that of 30 or 40 years ago. There are many more concepts expected to be understood by today's programmers and the academic revalidation cycle means that concepts are changing far quicker than the curriculum is changing. There are so many programming libraries that students have to have an idea of where to find things. However, many computer science courses seem to be little different to the approach of 40 years ago. The author did his introduction to programming in 1981. Pascal programming culminated with arrays and pointers at the end of a year, we wrote our own procedures and there were no objects. The ability to apply today's programming skills to, say, a browser environment manipulating components as well as the document is non-trivial, and a different language is used for programs in a browser environment, yet another for mobiles, and still another for server-side programming…

6.5.1 Language Problems

It depends which lecturer one speaks to as to what their preferred approach is. What's more, there is bound to be an adverse comment about the language, tool or platform:

"Java is the best language", says the module leader, Professor Indonesia, "it is the most widely used in industry".

"No, it isn't", says Dr. Australia, taking the practical on Tuesday, "Python is the language to use, it is functional", Wednesday's tutor, James Gosling, insists on C# because it is the most widely used in industry.

"I thought Java was" says the student, "that's what Professor Indonesia told us".

"He's a bit out of date", says Gosling, "C# is sort of Java with reliability, productivity and security deleted" (Joy 2002), he continues, "it has few restrictions and is therefore more flexible".

The tutor for Thursday's group, is the most popular with the students but they don't understand anything he says. "Ruby is a dynamic, interpreted, reflective, object-

oriented, general purpose programming language". says Yukihiro Matsumoto. "Ruby was influenced by Perl, Smalltalk, Eiffel, Ada and Lisp" (En.wikipedia.org 2018).

"We don't know who any of those people are", say the class. On Friday, the lecturer nearing retirement, Carlton McDonald, says the best Programming language for complete beginners is Prolog, Programming in logic. Research has shown that if you have never programmed before Prolog is easy to learn because you say what you want, rather than how you want it done. This is declarative programming.

Such a diversity of opinions further adds to the confusion amongst students, but the lack of an agreed best first programming language and development environment is just the beginning of the language problems. The amount of terminology used in programming alone is enough to make every medical student respect the drugs that the computer programming students have to have a working knowledge of:

Protocol (sounds like cortisol)
Inheritance
Binding
Scope
Closure
Override
Exceptions
Polymorphism
Parameter
Public
Private
Protected
And very many more…

Medical students spend years learning their drugs but are able to practice with a working knowledge of a subset of significant drugs within a few years. It is equally hard for the programmer because many of these terms are abstract, refer to relations between components (organs, vessels and bones) and systems. A program has a complicated structure, and there are patterns that can be used to put the body together. Programs can contain hundreds of thousands of lines, and are very complicated systems yet we expect students to learn programming in one or two 20 credit modules in the first year of their degree. This is not realistic.

A medical student must take and pass A levels Biology, Chemistry and Maths with the highest grades. This is not the case for computer science courses. Perhaps its time to review the effort required to learn how to program. There are an awful lot of students that manage to pass the programming modules, sometimes by referral, but never want to be programmers after graduation.

Programming languages preferences, and the sniping from one camp to another, confuses students in the friendly fire. The programming concepts that are expected of modern high level programming languages have not been estimated in terms of the time taken to learn these concepts. If one is struggling to understand the languages and concepts it is impossible to explain the difficulty one is having at a particular

time. If the student can't express the difficulty, how can the tutor help the student? Repeating what was said five times, only louder isn't going to help. Many of the concepts need practical examples, followed by practical applications in order to fully grasp them. 12 hours per week is nowhere near enough for learning programming. Two or three hours of supervised practicals are nowhere near enough for beginners, especially when 180 minutes in a class of 20 is only 9 min of one to one conversation on average per student per week. Over 12 weeks this is a little over an hour. What chance do the first year students have?

Are the expectations of the tutors of students too ambitious given the enormous changes in computer programming concepts, languages, libraries, application areas and paradigms? This review is overdue and may lead to a reduction in the 10% of students that fail Computer Science courses in the UK. However, it needs to happen at a high level, perhaps coordinated by HEA?

6.6 Cooperative Learning

The author has for decades insisted on individual assignments for an introduction to programming to be certain that students understand programming rather than understanding written programs. However, in 2017 the Programming Principles module had two cohorts: one from the department of Computing and Mathematics and a second from the Business School doing an IT for management as part of a business course. It was here, 32 years after first teaching programming, that it was observed that the students that work together on solving their problems were those that achieved the best grades. There is no need to insist that students work solely alone. Learning actually takes place more when students are learning in groups. Students prefer the safety of asking their friends more than asking the tutor in front of the rest of the class. As long as students understand what they are producing at the end, there should be no hesitation in permitting cooperative learning.

Those that explain to their friends however, sometimes have a program that works but not a program that works well. It is at this point that we have to address efficiency. We can write a statement seven times or, have a loop that goes from 1 to 7 performing the same statement each time around the loop. "It works", says the student, yes but what if you need to do the same statement 100 times, or 100,000 times? We don't just want something to work, on a mobile device there is not much worse than using an app that consumes excessive amounts of battery because it works inefficiently, or that has run out of memory because the programmer couldn't find an efficient was of writing the program to be compact and also to perform the tasks with as few statements as possible. Once this has been explained to the small group, or the entire class, if many of them are making the same mistake, or had the same hiatus, they will have learnt something. We often learn from our own mistakes; a wise person will learn from the mistakes of others.

6.7 Weekly, Consistent Learning

Nine times out of ten, the portfolio approach to 100% coursework requires the students to demonstrate all of the work in the final week or just after the end of the module and all in one go. During these demonstrations many of the students have no idea how the little program they wrote ten weeks earlier, or in some cases, a few days earlier, works.

There is no doubt that if students are assessed at the end of the session at which they are instructed they will remember more if:

- They have had to apply the knowledge through practical activity
- The more they have had to apply it, the more they will remember what they did (repetition deepens impression)
- They have not had time to forget what they were taught, if the knowledge was delivered 60 seconds earlier students will remember more than 60 min earlier, and far more than if the knowledge was give 60 days earlier.

It is for this reason, it is better to not allow students to wait until the end of the module before they attempt to apply the knowledge and practical skills they learnt at the start of the module. Assess them as early as possible by means of a few questions at the end of each session. These can be discussion questions, multiple choice questions, short written answer questions. The weekly assessment for programming could be: can you solve the following problems… extend the program in the following ways? Etc.

Weekly exercises require a lot of thought and preparation. A concern raised by many is that we are over assessing the students. The student concern is: will I get any marks for this? This failure to see the importance of answering questions is quite concerning. It seems that Computing students, and men in particular, are less articulate than their humanities peers. Actually, that is not entirely true. In the 1990s there was a discussion of software development research papers in week 10, of a 12 week course on Advanced Programming Methodologies. At the time there were approximately a third of the class from Austria. The Austrian students were far more articulate than the UK students (there were no females in the Advanced Programming Methodologies course in the 1990s). There is a body of research that should be done to determine the difference in the Austrian culture versus the UK culture, and why the Austrians significantly outperformed their UK counterparts.

When teaching programming it is not unusual to ask questions, and not receive responses. Students need to be encouraged to engage in class discussions and reminded that if they don't know the answer to a question, or don't understand something then half of the class doesn't know or understand either.

The more programs students write, the better they are able to understand programming challenges. However, if there is no one around to guide them as they attempt to program they are less inclined to invest that time, because one can easily spend hours or even days trying to get something working, without guidance. The approach that an increasing number of students are taking is to search for a complete solution

to the programming task. This is easier than trying to build it oneself and spending days getting nowhere. Once a student is demoralised it is difficult to get them back on track.

6.8 Incremental Programming, Incremental Learning

"**It's not that I'm so smart, it's just that I stay with problems longer**."—Albert Einstein (1879–1955)

If you want to try and solve anything, don't try and solve everything.

All programming problems are more easily solved if they are broken into smaller pieces. 40 years ago, Jackson Structured Programming structured programs around the structure of the data. To a small extent we have come full circle: When designing a mobile app, a series of storyboards, or screen designs, will identify all of the inputs and outputs for a screen, i.e. all the data the app has to deal with. Incremental development says start with a screen and generate the outputs required by utilising the user inputs and device sensor values required for each output. Do one at a time. This approach means that the screen designs are created first, then the transitions between screens, finally each output on each screen is implemented.

The result should be that one always has a program that is working, the only feature that may not work should be the one being worked on. The other features have not been started yet.

Programming is not unique in terms of production projects, in that products with numerous components have component tests and also dependencies. One is able to have a program that is incomplete but can undergo component tests. If one is building a car incrementally, the seats have to be complete before the steering wheel and the brakes must be working before a test drive. In a program there are data dependencies. All the data must be read before the calculations on all the data can be performed. Which parts of the app should one develop first? Agile methodologies say, work on the functions that one feels comfortable and confident to develop first. The delayed gratification personality says work on the hardest functions first. The entrepreneur, or consultant, works on the parts that will impress the client the most. Object Oriented suggests the one should work on the least connected objects, then there are bottom up approaches and top down approaches. There isn't a natural place to start.

This poses a problem for the beginner, where do I start? What should I do first? Just tell me what to do. This flexibility is a source of consternation initially because on the one hand programming appears to be a linear sequence of steps, in the event handling approach we don't know which event is going to happen first. It is not at all linear. On the other hand, component development of classes and functions, seems to be completely random.

As it happens, at the time of writing, the author's son completed a degree in Biochemistry and has been offered a job to become a software developer. The first three months are unpaid because he is receiving training. 12 weeks of training. 480 hours of development training is worth far more than 3 month's salary (although

it doesn't feel that way to a 22 year old) and opens up a lifetime of development opportunities. It will be interesting to see what approach they take, and how much he is expected to know, and what he is expected to be able to do at the end of the three months.

6.9 Assessment Versus Learning

There is an unfortunate dilemma that students face. They don't have enough time to do all the exercises required to become fully conversant with the programming techniques that build on each other, each week. Many of them have lost sight of the reason they are at university: to learn. This is partly because the external pressure on UK universities to achieve and maintain TEF gold. The relationships between staff and students are becoming entirely formalised because there is a perception that academic staff aren't working hard enough. It is the student experience that suffers. If lecturers are overworked and stressed out they don't have time to be available for students.

Students also have time constraints, not entirely due to course expectations. The biggest impact appears to be in the form of part-time work. A few years ago, at the author's institution, a report came out from the Students Union indicating that the average student was working 16 hours per week. If the average student is working 16 h per week, and not every student works, there are students that are working 20, 25 h, even as much as full-time night shifts, in some cases. This is possible because the emphasis on student-centred learning means that the bulk of student effort is inevitably away from the classroom. They feel that this allows them to study at times when they are not at work or university.

It is financially poor students that are affected most by the need to work but it is not just poor students. This academic year, the author has three children at university, two at undergraduate level and one at post graduate level. As the parental income of both parents as educators is quite high, the children have reduced maintenance loans, and need to work. Their loans do not cover their accommodation costs. The Student Academic Experience Survey Report 2018 indicates that students at post-92 institutions have relatively high volumes of [course related] work outside the university (Neves and Hillman 2018).

Surveys have shown students are increasingly sleep deprived. There is a generation of teens growing up chronically sleep-deprived. According to a 2006 National Sleep Foundation poll, the organization's most recent survey of teen sleep, more than 87% of high school students in the United States get far less than the recommended eight to 10 hours, and the amount of time they sleep is decreasing—a serious threat to their health, safety and academic success (News Center 2015).

As a result of the changing culture and the high demands of the subject, students manage their time by doing the activities that will not only be assessed but actually accumulate marks in order to pass the module. It is for that reason students are increasingly asking: "What do I need to do to pass?" or "What do I need to do to get a 1st"? This selective learning is from those students that are struggling to manage

the demands of modern computer science degrees with those of being able to keep their debt low.

It is therefore understandable that students don't have time to complete all the tasks we prepare for them. It's the main reason students want video recordings of lectures rather than documents to read. It takes effort, a lot of effort, to read unless it is one's preferred learning style. As students are sleep deprived reading sends many of them to sleep, so they find themselves drugged up with caffeine or reading the same paragraph 3 or 4 times between snoozes.

6.10 Summary

Computer programming languages along with computer technologies are unrecognisable in comparison to 40 years ago. The computer was the size of a classroom, the programming language BASIC had a small set of instructions with which to assemble programs. Today the computer in our pocket or handbag is able to do far more than several computers could do in several rooms. The programming languages and programming concepts are significantly more complicated, and if one doesn't know fully how to utilise an ArrayList for example, it is impossible to be able to pass an array as a parameter, or to process an array of elements polymorphically. Two or three hours of supervised practicals per week are nowhere near enough for beginners. The changing culture and the high demands of the course are the reason the drop-out rate for Computer Science is the highest of all degrees in the UK. It is time to conduct a review of the expectations of the challenges of teaching and learning to program.

References

En.wikipedia.org (2018) Ruby (programming language). [online] Available at: https://en.wikipedia.org/wiki/Ruby_(programming_language). Accessed 17 June 2018

Joy B (2002) Microsoft's blind spot. cnet.com. Retrieved 12 Jan 2010

Krishna R (2018) Intro to agile under 5 minutes. [online] YouTube. Available at: https://www.youtube.com/watch?v=N2hDKpgzdIE. Accessed 17 June 2018

Nevevs J, Hillman N (2018) The student academic experience survey report. [online] Higher Education Academy. Available at: https://www.heacademy.ac.uk/knowledge-hub/student-academic-experience-survey-report-2018. Accessed 21 June 2018

News Center (2015) Among teens, sleep deprivation an epidemic. [online] Available at: https://med.stanford.edu/news/all-news/2015/10/among-teens-sleep-deprivation-an-epidemic.html. Accessed 17 June 2018

Pine C (2009) Learn to program, 2nd edn. The Pragmatic Programmers, ISBN 978-1-93435-636-4

The Verge (2018) MIT brings Google App Inventor back from the dead as open-source project. [online] Available at: https://www.theverge.com/2012/1/21/2723656/google-app-inventor-open-souce-code-released-by-mit. Accessed 16 June 2018

Thunkable Thoughts—The How and Why You Would Make Your Own Beautiful App (2018a) Thunkable Thoughts—The How and Why You Would Make Your Own Beautiful App. [online] Available at: https://blog.thunkable.com. Accessed 16 June 2018

Thunkable Thoughts—The How and Why You Would Make Your Own Beautiful App (2018b) Mark Friedman, Cofounder of App Inventor, Joins Thunkable!. [online] Available at: https://blog.thunkable.com/mark-friedman-founder-of-app-inventor-joins-thunkable-ee3f130a2835. Accessed 17 June 2018

YouTube (2018) Introduction to scrum—7 minutes. [online] Available at: https://www.youtube.com/watch?v=9TycLR0TqFA. Accessed 17 June 2018