Cognitive influences on learning programming

Author

Zagami, J

Published

2023

Book Title

Teaching Coding in K-12 Schools: Research and Application

Version

Accepted Manuscript (AM)

DOI

10.1007/978-3-031-21970-2_26

Cognitive Influences on Learning Programming

Zagami, J. (2022). Cognitive Influences on Learning Programming. In T. Keane & A. Fluck (Eds.), *Teaching Coding in K-12 Schools: Research and Application* (pp.389–399). Springer Nature. https://doi.org/10.1007/978-3-031-21970-2_26

Abstract

A theoretical exploration of cognitive load to guide the teaching of computer programming by tailoring the use of different programming language types (visual vs textual) to the developmental needs of students relative to the complexity of the cognitive concepts being taught so that the cogitative processing capacity of students is not exceeded.

Introduction

With hundreds of programming languages available to teachers, choosing which programming language or languages to use when teaching students how to program is often the first concern many teachers have. Educators often look to industry for measures of which languages are the most popular, the recommendations of other teachers, and their own experiences in learning programming. There are, however, reasons to use different programming languages in different circumstances, depending on the age and experience of the learner with programming, the type of problem being solved, or, as will be focused on in this chapter, the complexity of the problem being solved.

Learning a programming language is a difficult undertaking; some argue that it is akin to learning a new foreign language (Prat, Madhyastha, Mottarella, & Kuo, 2020), and while there are similarities, there are also significant differences. While both require mastery of grammatical rules and syntax, programming languages have far fewer such rules, though conversely, these must be used more precisely than with natural languages.

Previous experiences with programming languages are also important. While all programming languages have similarities in basic operations of sequencing instructions, branching, looping, modularity, and the available data structures, each language also has its own syntactical rules or grammar and different thematic structure, akin to language categories. Learning French or Italian, having learnt English, is relatively straightforward, much less so is learning Chinese or Navaho. Learning a programming language that has similar genealogical roots to a common language (such as the Java and Python languages,

both developing from the C programming language, is relatively straightforward. Likewise, various versions of SQL developed from SEQUEL, and have a range of commonalities, but learning Python, having learnt SQL, would involve learning quite different grammatical, syntactical and logical language rules (Rigaux, 2020).

Each programming language also has its approach to abstracting instructions into a form processable by a computer. These are commonly framed around a language metaphor - flow charts, filmmaking, circuit/wire diagrams, card decks, etc. This variation is seen especially in programming languages developed for teaching young children to program, with languages such as Alice relying on a film making metaphor, RoboLab relying on a metaphor of circuit/wire diagramming, and many simple coding languages for young children using a playing card metaphor to depict the processing of a sequence of instructions. Professional programming languages also have such categorisations, primarily: procedural, functional, object-oriented, scripted, and logic-based languages, but over 50 categories exist, and these relate to the types of problems the programming language has been designed to solve.

The metaphor used by a language can influence how effectively the language can be learnt, but this is influenced by each learner's affinity to that metaphor and their previous experience with similar programming languages or the concepts incorporated into the metaphor. Within the wide range of available programming languages, learners will find some languages more straightforward to learn than others. A set of languages have been designed to support learning, generally using more familiar metaphors and simplifying programming processes to reduce syntactical and logical errors. Beyond this, however, there is a body of research from cognitive science that can shed some light for teachers on why students have difficulty in learning programming and why some programming languages are easier for students to learn than others. In this chapter, we will focus primarily on one such theory, that of cognitive load.

Cognitive Load

Cognitive Load Theory (Sweller, 1988) suggests that we each have a limited capacity to hold different concepts in ‘working memory’ when problem-solving, with the implication that when programming problems involve too many different elements, this capacity can be exceeded. Students will then have increasing difficulty in solving such problems. Working memory differs from long term memory in that what is held in this memory is done so temporarily, in combination with other memories, so that these can be associated and then reinforced in long term memory along with the associated links between such memories. Generally, this capacity limit is described as 7 plus or minus 2. When faced with problems of such complexity that they exceed this capacity, students would find it increasingly difficult to develop solutions to these problems because additional memories of these concepts could not be accommodated in their working memory. 7 plus or minus 2 belies the implications of such a capacity limit that represents a range from 5 to 9 elements, with some students having almost twice the working memory capacity of other students. The implications of such a difference for education are profound and unappreciated.

This concept of cognitive load has subsequently developed into three facets:

1. *Intrinsic Cognitive Load* (Chandler & Sweller, 1989) reflects the inherent level of difficulty of the problem that students may be attempting to process in their working memory in relation to their intrinsic capacity, i.e. between 5 and 9 elements. The theory does not expand upon when or how this capacity is determined but suggests that beyond this point, it is an inherent capacity constraint that cannot be modified by education or effort. There are, however, techniques that can be taught to ameliorate the limitation, such as synthesising a range of memories into a single new concept (chunking) when the complexity exceeds the working memory capacity of the student.

2. *Germane cognitive load* (Sweller, 2010) involves the development of schemas that support student understanding of the problem and provide another technique to overcome the limitations of working memory. Flowcharts are an example of an external schema, providing students with a means of processing how various programming instructions are executed. Germane cognitive load also extends to the creation of internal schema such as mental models can again be limited by the complexity of mental models that can be held in working memory, and decomposition of such models into manageable schemas is necessary to process complex models.

3. *Extraneous cognitive load* is an effect that worsens the impact of working memory constraints. It is produced by the manner in which information is presented to or learnt by students, an example being the mixing of instructional media such as text and images, each requiring memory recall and processing in different parts of the brain and, in combination, more likely to exceed working memory capacity (Zagami, 2012).

Intrinsic Cognitive Load

While Intrinsic Cognitive Load is the least malleable by teachers, being an intrinsic capacity unique to each student, the effect on student learning can be significant when teachers provide programming problems of increasing complexity without considering their intrinsic cognitive load. These problem sets generally increase in cognitive difficulty as the number of concepts involved increases the problem complexity. At some point, students' working memory capacity will be exceeded as they attempt to hold the range of different concepts in their working memory simultaneously to process a problem to a solution. Chunking is a cognitive process of abstraction, taking a set of elements that would have needed to be included in working memory and essentially creating a new concept for this collection in a similar way that we would create a subroutine or module in programming, and then only needing to hold this new process in working memory. In teaching, this cognitive process can be mirrored by the computational thinking processes (Tsarava, et al. 2022) of decomposition, generalisation and synthesis, breaking down complex problems into manageable components, generalising these, and synthesising the generalised components back together to solve the overall problem. Once students are taught to decompose, generalise and synthesise, they can

rely upon the successful completion of previous problems they have mastered to solve more complex problems.

Unfortunately, computational thinking skills are not well addressed in computer education - decomposition, Synthesis, Generalisation, Pattern Recognition, etc. Even fundamental processes such as Algorithmic Thinking are usually taught in an ad hoc way, with little understanding of why such processes are necessary for problem-solving in relation to their impact on reducing cognitive load.

Likewise, the creation of external schemas, while commonly taught, starting with flowcharts, data flow diagrams, UML, relational schemas, etc. as an aid to understanding programming, involves little consideration of why these aid understanding and student capacity to solve more complex problems, or how they contribute to the development of internal schemas such as mental models depicting the interaction between various elements and concepts.

While simple programming concepts can be held in working memory and processed without difficulty, more complex real-world problems involving a range of programming concepts, such as the use of various data structures and operations, can easily exceed student working memory, requiring them to approach the problem in more manageable parts before an overall solution can be synthesised.

The ability to decompose problems should be taught as soon as students begin having difficulty synthesising complete solutions to problems, even though they may be able to articulate an understanding of individual components successfully. As the working memory capacity of students can vary significantly, from 5 to 9 elements, it is difficult for teachers to plan learning activities to support this range of students fully. While a differentiated approach can assist students with lower working memory capacity while enabling students with greater working memory to work to their capacity before needing to rely upon decomposition and chunking processes, generally, all students will need to use techniques to break problems down into manageable parts reasonably quickly. Once this complexity of problem-solving is reached, the differences in programming ability between students with respect to their intrinsic working memory capacity are reduced significantly.

A problematic zone exists, however, between simple problems involving 5 or fewer elements that all students generally have the capacity to process, and problems involving 10 or more elements, that all students need to use chunking and decomposition techniques. Within this zone, student ability is significantly influenced by their intrinsic working memory capacity and not their learning, effort or intelligence. This presents a problem for programs of instruction that gradually increase the complexity of problems through this zone while delaying the teaching of decomposition and synthesis techniques to combine the various solved elements into an overall solution to the problem under investigation.

To address intrinsic cognitive load, decomposition and synthesis techniques should be taught as soon as students have difficulty solving problems of reasonable difficulty. Without such techniques, it may be impossible for students to solve the problem simply because of intrinsic limitations in their working memory. They may become increasingly frustrated by the ability of their peers to do so, regardless of their efforts, simply because they have a different intrinsic capacity.

Germane Cognitive Load

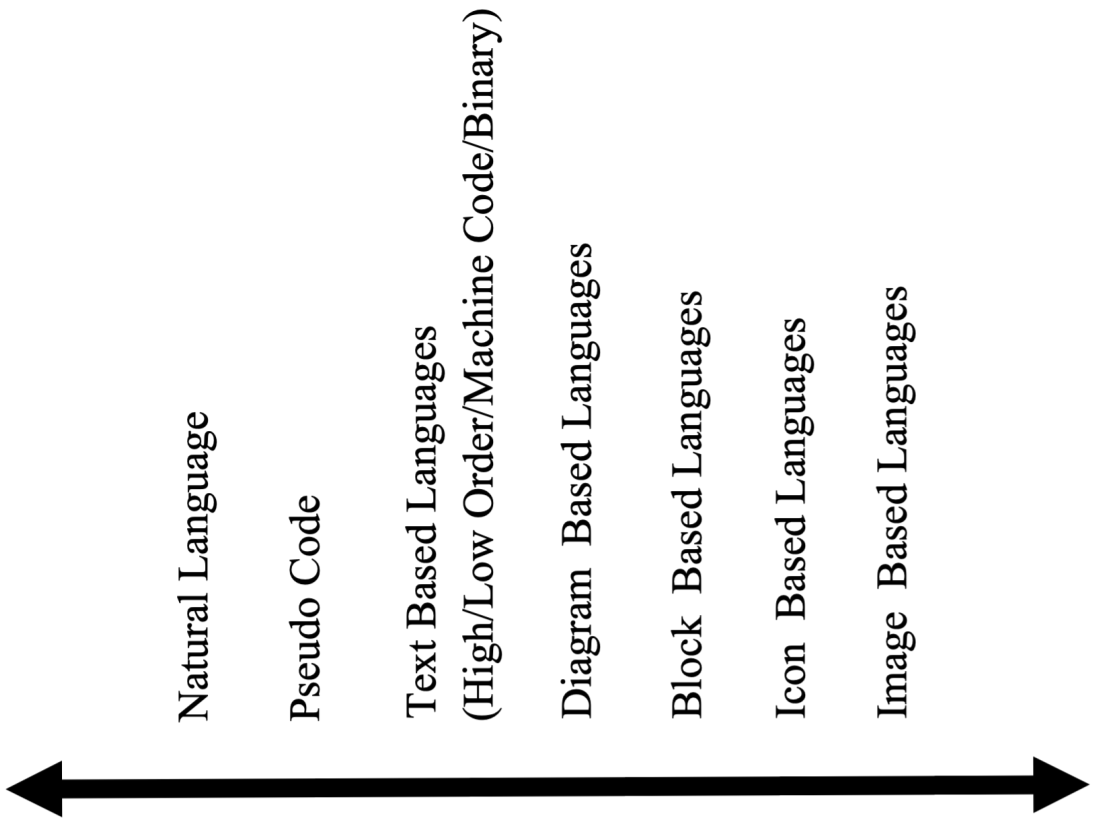
Germane cognitive load provides an opportunity to shift some of the burdens of maintaining elements in working memory to an external mechanism. While this can be in the form of textual narratives, instructions, pseudo-code or programming code, these have their cognitive constraints, and it is more often expressed in the form of visualisations that can depict information and the relationships between information more efficiently, essentially reflecting the idiom “A picture is worth a thousand words". Flowcharts, Nassi-Shneiderman Diagrams, Data Flow Diagrams, Entity-Relationship Diagrams, and Conceptual Schemas each depict various relationships, with the Universal Modelling Language attempting to address these collectively by incorporating a range of diagramming approaches, including behavioural diagrams - use case, activity, interaction, timing, state machine, communication and sequence diagramming; and Structural Diagrams - class, object, component, composite structure, deployment, package and profile diagrams. These visualisations approaches are used to assist the programmer in understanding what is occurring within the complex system of a software application. Human brains are not well suited to comprehensively understanding every interaction in reasonably complex software solutions. Still, by using abstraction, and in particular visualisation of these abstractions, we can generalise a complex problem to the point where we can understand the overall processes and interactions occurring, delving into the detail of parts of the whole as needed to explore the underlying complexity. This process of abstraction supports germane cognitive load, using external representations to reduce the load on working memory.

All programming languages involve abstraction, binary being an abstraction of the transistors turning on and off in a microchip, machine language being an abstraction of binary, and higher-level programming languages being abstractions of lower-level ones. Visual programming languages continue this process, using graphical techniques to represent the function and relationship between coding statements. Block-based coding languages such as Scratch and Blockly (Papadakis, Kalogiannakis, Orfanakis, Zaranis, 2019) are thus commonly used in primary education because the strict syntactic rules required by textual programming languages are managed by the physical placement of blocks in relation to one another within a block-based diagram. They also provide a diagramming process in which the complexity of a program is reduced by using abstracted blocks placed in relationship to other blocks, reducing germane cognitive load.

The differences between visual and textual programming languages can be considered along a continuum.

Figure 1.

Continua of textual and visual programming languages



Natural language programming that can interpret coding instructions from human language expressions of what the program should do can be considered one extreme of abstraction on a continuum of textual and visual programming approaches. Pseudocode provides the next stage, representing programming instructions more abstractly than textual programming instructions of traditional programming code. Within textual programming languages, there are various levels of abstraction, described as orders. Higher-order languages tend to have more complexity, abstracting various simpler instructions in lower-order languages that tend to be less abstract from machine code instructions, leading to the most concrete, binary command sequences. The continua do, however, extend in another direction of abstraction, of increasing visual, rather than textual abstraction. Diagram based coding involves the use of interactive visualisation tools such as flow charts, Nassi-Shneiderman diagrams, and structured design charts to represent programming elements and process flow, with automated translation generally occurring to generate a textual programming language code from the

diagram. Block-based languages extend this to distinct programming languages that abstract textual commands into labelled blocks, allowing the placement of these commands to incorporate the syntax of textual languages. Further abstraction involves removing labels, and using icons that represent textual commands to be arranged, often with ‘wires’ linking icons and abstracting the instructional flow. While entirely image-based languages do not generally yet exist beyond some very specific interactive visualisations, they would represent the other end of the spectrum from natural language programming, where a single interactive image depicts the entirety of a program, with the images changing content, placement, tones, colour, etc. reflecting an abstraction of the code, processes and data structures.

Extraneous Cognitive Load

Cognitive processing of textual and visual elements occurs in separate areas of the brain, with textual processing involving a translation into auditory language structures and processing in the auditory cortex, while visualisation is processed in the visual cortex. This has implications for how instructional material is presented to students and the choice of the programming language used.

Diagram and Block-based coding can require the processing of textual labels and visual placements, increasing extraneous cognitive load with an increased demand on working memory. Such demand is not problematic for simple problems that include both textual and visual processing; indeed, it may improve student learning by being able to draw upon connections made in these widely separate areas of the brain, triggering recall as the result of not just textual cues but also visual cues. The problem is that the increased demand in processing elements from auditory and visual elements, assuming these have not been chunked into a single conceptual element, can likely overload working memory faster than if the instructional content had included only text or visualisations. Likewise, programming languages that rely upon a mix of textual labels and visual elements, while improved for simple problems, may exceed working memory because this extraneous cognitive load increases more quickly than programming languages relying upon only text or visuals.

The use of combined visual and textual visual programming languages in early programming has the potential to improve student learning of simple concepts, but this needs to be offset with potential difficulties in subsequently learning purely textual or visual programming languages.

Where concepts are encoded and stored using visualisations, focusing on the spatial relationships of entities, etc., then the shift to text-based coding, relying primarily on auditory processing and the decoding of textual commands in a similar manner to human language decoding, students may find the transition to be more difficult. Essentially, students will need to relearn, encoding a different area of the brain, albeit with the advantage of being able to draw upon their learning of the concepts through visualisation. The reverse would also be true in learning a purely visual programming language.

A balance needs to be made between the advantages of learning from rich media, textual and visual, based programming languages, and the impact this may have on the learning of subsequent languages that rely upon a single media. Extraneous Cognitive Load is, however, considered the weakest of factors influencing working memory, and the support visualisations can provide to germane cognitive load can outweigh the impact of Extraneous Cognitive Load.

Generally, however, students can process problems involving fewer than 5 elements more effectively using programming languages with both visual and textual elements, problems with between 5 and 9 elements are more effectively understood using purely textual or purely visual programming languages, and for problems of greater than 9 elements, there is little difference, as all require chunking techniques such as decomposition to address problems regardless of the programming language used. There are, however, differences resulting from the capacity of the programming languages themselves, with most languages combining visual and textual elements not well designed to display and manipulate large complex solutions. In this case, purely visual programming languages have some advantages over purely textual programming languages in supporting student understanding of the complex interactions occurring (Zagami, 2012).

Cognitive Fit

Cognitive Fit theory (Vessey, 1991) supports the use of visualisation independent of working memory theory by considering the fitness of various approaches to problem-solving and learning, in particular, the comparison between visual and textual representations. In exploring the use of visualisation tools for problem-solving, Cognitive Fit theory compares the effectiveness of spatial (visual) and symbolic (textual) representation of information, with clear advantages in time and accuracy established when processing information visually over textually. This has been subsequently demonstrated in the learning of a range of programming concepts, such as iteration and recursion (Sinha, & Vessey, 1992) and objects (White, 2001).

Cognitive walkthroughs

Also supporting the use of visual programming languages is their potential interactivity. Developed initially to evaluate the effectiveness of visual programming languages (Green, et al., 2000), cognitive walkthroughs now more commonly provide a mechanism by which students can explore the operations, flow and data of an executing program. While walkthrough mechanisms exist for text-based programming languages, those incorporated into visual programming languages are generally more interactive and can show multiple threads occurring at once.

Cognitive walkthroughs can be used for the:

Representation of operations

This is the most incorporated walkthrough in visual programming languages, highlighting the programming operations in sequence much as a flowchart. The ability of visual programming languages to replace the need for diagrammatic planning tools such as flowcharts, is incorporated into the spatial layout and connectivity of block-based programming languages.

Representations of flow
Representing which operations are currently active, this represents a desk check process to visually see how interaction, branching, looping and subprocesses are executing. In debugging, students can see the flow of operations and identify discrepancies with expected flow and terminations.

Representations of data
Representing the data values of variables and more complex data types, combined with representations of flow, changes in data values can be observed in relation to the operations acting upon them, further assisting in debugging, by comparing expected changes in values in real time with representations of flow, as values are displayed visually and change because of visually highlighted operations.

Collectively, cognitive walkthroughs using visual programming languages, can be dynamically compared with students' mental models of how programming code should execute, and discordances assist in challenging students to reconsider their understanding of cognitive concepts being learnt.

Conclusion

The choice of programming languages used in teaching programming can be a significant factor in the effectiveness of this instruction. Generally, younger students benefit from programming languages that mix text and visual cues to support students in solving relatively simple problems. Very young pre-reading students, however, benefit more from simple, purely visual programming languages and tactile programming tools.

More experienced students tasked with solving complex real-world problems need well established computational thinking skills, in particular, decomposition, generalisation, and synthesis, to support them in breaking down problems into manageable elements and combining these into overall solutions. The choice of programming language is less relevant, but a purely textual or purely graphical language is likely to be more effective than a block-based language, for example, that requires both visual and textual processing. In balance, purely visual programming languages have more advantages in conceptualising elements of a problem and understanding the synthesis involved in generating a solution from sub-elements, but such languages are less commonly available and tend to be specific language types that have a focus on multitasking and event triggers, such as used in control systems and robotics. In general, text-based programming languages are more common, and their disadvantages in visualisation can be compensated for by using a range of diagrammatic

visualisation tools to supplement the textual code in the cognitive processing of the problem to a solution.

The difficulty exists between these groups, often occurring in the middle years of school, lower secondary. Students at this age are pushing the design constraints of most block-based programming languages, designed for instruction, and focused on solving relatively simple problems. Students are beginning to explore real-world problems that involve more concepts than their working memory can process because of the increased demands of their extraneous cognitive load by block-based tools. Generally, they have insufficient computational thinking skills to chunk these processes effectively to reduce intrinsic cognitive load, and their skills in the use of diagramming tools to reduce germane cognitive load may also be limited. Combined, middle years students may face the greatest challenges in learning to program while also being most formative in decisions about future career pathways.

Supporting middle years students with their transition from block-based programming to purely visual or textual languages needs to be conducted in a considered manner to reduce the range of cognitive loads impacting students. First and foremost are their computational thinking skills, particularly decomposition and synthesis, without which no programming language will support them in solving programming problems beyond a relatively low complexity level because of their intrinsic cognitive load.

Teachers can also support them in ensuring they master and use a wide variety of diagrammatic tools to supplement their use of textual programming languages (these are generally incorporated into purely visual programming languages) to reduce their germane cognitive load.

Finally, in considering working memory, the use of block-based and other visualisation-based programming languages should not be discounted even for the most experienced programmer. Simple programming tasks can be more efficiently conceptualised and developed with these tools, generally automatically incorporating the diagramming processes conducted when using text-based programming. Particularly when students have well developed computational thinking skills to enable them to decompose problems, generalise, and synthesise programming solutions, coupled with a mastery of a range of diagramming tools, the limitations of any type of programming language will fade in comparison to their strengths in solving various problems.

Teachers are recommended to provide students with experience with a wide range of language types, optimised for different purposes, and this can only support the flexibility that students will need in the future. Initially, programming languages designed to be easy to learn will be prominent, but over time students should learn a range of languages optimised for various problem solutions. Robotics and control problems are well supported by specific languages, as are solutions involving web-based applications, information system problems, artificial intelligence applications, etc. The use of text and visuals are not the only

classification of programming languages, but there is a lack of research into the various cognitive loadings other programming language characteristics impose upon programmers.

Students will inevitably learn a wide variety of programming languages over their lifetime as approaches to programming are refined and developed. AI and networks will have immediate impacts, with augmentation and virtualisation technologies presenting entirely new approaches to visualisation that have not yet been explored in-depth regarding programming. Likewise, we will continue to better understand cognitive processes and make improvements in our understanding of learning and teaching. However, the fundamental computational thinking skills that students develop, combined with their experiences with programming to solve problems, will better prepare students to face these challenges and those for which we cannot yet conceive.

References

Chandler, P., & Sweller, J. (1991). "Cognitive Load Theory and the Format of Instruction". *Cognition and Instruction, 8*(4): 293–332.

Green, R., Burnett, M., Ko, A., Rothermel, K., Cook, C., & Schonfeld, J. (2000). Using the cognitive walkthrough to improve the design of a visual programming experiment. In *Proceeding 2000 IEEE International Symposium on Visual Languages* (pp. 172-179). IEEE.

Papadakis, S., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2019). The appropriateness of scratch and app inventor as educational environments for teaching introductory programming in primary and secondary education. In *Early childhood development: Concepts, methodologies, tools, and applications* (pp. 797-819). IGI Global.

Prat, C., Madhyastha, T., Mottarella, M., & Kuo, C. (2020). Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports*, *10*(1), 1-10.

Rigaux, P. (2020). Diagram and history of programming languages. Retrieved February 24, 2022, from http://rigaux.org/language-study/diagram.html

Sinha, A., & Vessey, I. (1992). Cognitive fit: an empirical study of recursion and iteration. *IEEE Transactions on Software Engineering, 18*(5), 368.

Sweller, J. (1988). "Cognitive Load During Problem Solving: Effects on Learning". *Cognitive Science*, *12*(2): 257–285.

Sweller J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review, 22*: 123–138.

Tsarava, K., Moeller, K., Román-González, M., Golle, J., Leifheit, L., Butz, M. V., & Ninaus, M. (2022). A cognitive definition of computational thinking in primary education. *Computers & Education*, *179*.

Vessey, I. (1991). Cognitive fit: A theory based analysis of the graphs versus tables literature. *Decision sciences, 22*(2), 219-240.

White, G. L. (2001). *Cognitive characteristics of learning Java, an object-oriented programming language*. The University of Texas at Austin.

Zagami, J. (2012). *Seeing is understanding: The effect of visualisation in understanding programming concepts*. Lulu.com.