https://doi.org/10.34074/rere.00408

Conceptual Article

OBJECT-EARLY VERSUS OBJECT-LATE: PERSPECTIVES ON CONCEPT ACQUISITION IN UNDERGRADUATE SOFTWARE ENGINEERING COURSES

Barry Dowdeswell, Tariq Khan, Asanthika Imbulpitiya, Waruni Hewage, Kathiravelu Ganeshan and Farhad Mehdipour

OTAGO POLYTECHNIC AUCKLAND INTERNATIONAL CAMPUS

ABSTRACT

Teaching Object-Oriented programming is a core element of undergraduate software programming courses. Using a custom-developed Ontology of programming language elements as well as case studies from three institutions, this article explored the order of presentation of both procedural and abstract concepts, including Object-Orientation. The findings are considered with reference to issues raised in both current and historical literature that examines concept acquisition in this context.

The literature suggests that in the past, first-year failure rates were often woefully high. Teaching abstract object concepts too early can hinder the establishment of a clear procedural understanding of coding. Similarly, presenting concepts in the wrong order is confusing and counter-productive for novice developers. The article concludes that the choice of programming language is not necessarily as significant as the order and way in which language concepts are presented and reinforced.

Keywords: Software Languages, Procedural Programming, Object-Oriented Programming, Teaching Paradigms, Concept Reinforcement.

INTRODUCTION

Part of the motivation for this research was the shared experience of the authors, who all teach first-year programming courses together at Otago Polytechnic Auckland International Campus (OPAIC). When developing new courses or revising existing material, we are all cognizant of how important it is to introduce programming concepts in a logical, consistent order. When writing program code to address a requirement, it is vital to help learners acquire both a theoretical understanding as well as know how to make the best choice of constructs and commands from what the language provides. That is an iterative process which can be undermined whenever irrelevant or unrelated concepts are introduced at the wrong time. Computing languages also evolve over time, introducing new capabilities and complexities. The literature review in the next section contains examples of how practitioners in other educational institutions addressed these challenges.

During introductory courses, students learn how programming languages store information in entities called variables. A computer program is a set of instructions which manipulate or process the information stored in variables. As the program runs, it can generate more information or solve problems. Mathematical operations are performed on numeric variables while functions such as string concatenation change data held in character type variables. Variables are defined by assigning them a unique name and, in type-specific languages, specifying the type of data they will hold (Omar et al., 2014).

Figure 1: A chess board used in the author's classes

Object-Oriented paradigms emerged in the early 1960s to address the issues of complexity and scalability (Gabbrielli & Martini, 2010; Saidova, 2022). Objects can be used more efficiently to implement individual entities such as the pieces in a chess game. Multiple instantiations or instances of an object represent the Queens, Bishops, and other pieces. Each object instance holds the unique board rank and file of that piece, its colour, and other properties. Objects also contain or encapsulate code functions such as validating a proposed move for that type of piece. However, the literature shows that for many students, mastering Object-Oriented paradigms is still a daunting task. Related object concepts such as Polymorphism and Inheritance are often challenging to understand and implement.

The primary research question explored in this article is "What is the optimal order for introducing procedural and object-oriented concepts to novice students?". The literature often highlights the issue of Object-Early versus Object-Late approaches (Astrachan et al., 2005; Kiper & Abernethy, 1996). A number of related concerns are implicit in this: how well do students emerge from this time and what are they proficient at? What skills do they lack? How well do they acquire and hone the skills required to design more complex programs later, apply appropriate software architectures, and become proficient with a range of different languages? More importantly, how do they develop the native instincts and cunning required to debug applications effectively?

Understanding the prior experiences of other faculties is invaluable to those contemplating a change to their own chosen language. In the context of these questions, how well does the programming language promoted by a faculty support the learning outcomes of their course? The skills that institutions help students build in their introductory courses often directly affects their level of achievement in subsequent years. How effectively students acquire foundational programming concepts at this early stage is a critical contributor to their long-term ability to be productive and later, employable.

BACKGROUND HISTORICAL PERSPECTIVES FROM THE LITERATURE

Severance (2012) comments that at some point in the 1960s and 1970s, academic computer scientists transitioned from concentrating all their effort on building hardware to also include learning how to write software. They recognised that they also needed to learn how to teach their students to write code. Kemeny and Kurtz (1964) had developed their Beginner's All-Purpose Symbolic Instruction Code (BASIC) programming language at Dartmouth College in 1964. It provided a foundational teaching tool that became well-accepted both in education and industry (O'Regan & O'Regan, 2012). Today, the True BASIC language (Bantchev, 2008) and Microsoft Visual Basic .NET (Balena, 2004) still embody the core language defined by Kemeny and Kurtz.

Literature was chosen that profiled a faculty's choice of a teaching language, including their experience migrating from one or more languages to another. Studies provided both empirical and qualitative feedback on these processes. Concerns about introducing objects too early and how to mitigate the effects were contrasted. The range of options about object paradigms and the merits of teaching them is diverse. There is also evidence that the failure rate in first-year courses during this time was disturbingly high.

Agder College in Norway was a typical example (Hadjerrouit, 1998). In 1998, they migrated from Simula and C++ to Java. They cite their primary reasons as being the need to have a language that supported object-orientation, concurrent

programming, and had applicability to web design. Agder's experience needs to be seen in the context of other early adopters, coming only three years after Java's first commercial release in 1995 (Horstmann, 2021). The Simula language (Dahl & Nygaard, 1966) had been taught at Agder since 1967, supporting classes and object creation since its inception. Hence the opportunity to teach object concepts using an appropriate paradigm, examples, and language had not been lacking.

Hadjerrouit explains that Agder's key learning requirements included their desire to teach algorithmic thinking as well as general programming principles. They emphasized design and modular program construction as well as the acquisition of problem-solving skills. Like many others, Agder believed that when choosing a new language, pedagogical considerations such as simplicity and support for object-orientation should take precedence over the pressure to choose a language just because it is favoured in industry. Hadjerrouit states that this was the main reason they originally taught two languages. However, they found that after a three-year investigation, only 60% of their students felt comfortable programming in Simula. The majority also had great difficulty with C++, even though Agder used a limited subset of the language (Stroustrup, 1986).

The literature reports common themes about what constitutes a good teaching language and what the difficulties are with the course languages available. The need to acquire clear, proven debugging skills is imperative. However, bug finding is hard when learning in command-line environments. An Integrated Development Environment (IDE) provides additional debugging capabilities that support C++ and Java better (Hadjerrouit, 1998). Gosling, the creator of Java, states that the language was designed to be simple enough that programmers can achieve fluency in the language quickly and that it has novice usability (Gosling et al., 1996). Hadjerrouit countered saying that Java is not simpler than Simula. While Java is syntactically very similar to C/C++ and Simula, it is not necessarily easier to learn.

Figure 2 shows examples of programs which all print "hello, world" and the current value of a variable which manages a counter. The examples include code written in BASIC, the C language, Java, and Microsoft C#. Why do students find these languages so hard to master? Part of the answer lies in the differing ways students grasp concepts while learning. Procedural programming in BASIC and C is more concrete, since they do not support objects. This encourages the incremental step-by-step introduction of ideas. Each concept, such as creating and using an integer variable, is small and easy to illustrate. However, objects are complex data types. They are entities that are like variables but have multiple data types and values that are changed by mutator methods. This is an abstract concept that is not intuitive for beginners. This issue of teaching object principles early in a course, referred to as "Object Early", is a common thread in the studies examined (Astrachan et al., 2005; De Raadt et al., 2002). The BASIC and C examples are not Object-Oriented: they print using language keywords. In contrast, the Java and invoke the more complex display console object methods. Students using these programs may not be aware they are using objects when they begin to program in these languages.

Figure 2: Programming Language examples including BASIC, C, Java, and C#

```
REM BASIC EXAMPLE
10 LINE = 0
20 FOR COUNTER = 1 to 10
30    LINE = LINE + 1
40    PRINT "hello, world ", LINE
50 NEXT COUNTER
```

```
// C EXAMPLE
#include <stdio.h>

int main() {
   int line = 0;
   for (int counter = 0; counter < 10; counter++) {
      line++;
      printf("hello, world %i \n", line);
   }
   return 0;
}
```

```
// Java EXAMPLE
class HelloWorld {
   public static void main(String[] args) {
      int line = 0;
      for (int counter = 0; counter < 10; counter++) {
         line++;
         System.out.println("hello, world " + line);
      }
   }
}
```

```
// C# EXAMPLE
namespace HelloWorld {
    internal class Program {
        static void Main(string[] args) {
            int line = 0;
            for (int counter = 0; counter < 10; counter++) {
                line++;
                Console.WriteLine("hello,world " + line);
            }
            Console.ReadLine();
        }
    }
}
```

However, the lack of complex C-style pointers in Java is perceived to be an advantage over C and C++. Milne cites pointers as being the single most difficult concept for students in each of the language construct categories they surveyed

(Milne & Rowe, 2002). More than one article recommended the use of a reduced subset of C++ for teaching (Dale & Weems, 2014; Hadjerrouit, 1998; Hasker, 2002).

Faculties often describe their approach to teaching coding as requiring students to write English-like pseudo-code statements that they later translate into program code (Hadjerrouit, 1998; Van Rossum et al., 1999). In many introductory courses, that approach worked well. Conditional branching control structures are readily understood if they are expressed first as English sentences in the form: if this is true then do that. Otherwise do this. Building indented statement blocks reinforces the concept that debugging becomes easier if the structure is clear. Since the Python language mandates indentation in control structures, it enforces this practice; there is no option but to indent correctly (Van Rossum & Drake Jr, 1995).

However, Robins (2012) notes that typical introductory programming courses demonstrate an unwanted bi-modal grade distribution, exhibiting a greater than expected aggregation of both high and low marks. Dehnadi and Bornat 2006 reported first-year programming failure rates between 30% and 60%. When Bennedsen and Caspersen analysed data from a range of smaller sized classes, it showed a much wider distribution. They saw the pass rates vary from 0% and 60% across 67 institutions between from 2004 to 2007. This represents an overall failure rate of 33% across these institutions for that period. Their later research in 2019 showed a decrease in the failure rate to 28% that they contrast with the 42% to 50% failure rate seen in US college algebra courses (Bennedsen & Caspersen, 2019).

In contrast, DeClue (1996) proposes that object instances may be an easier concept to grapple with since they are more analogous to real-world objects. Osborne and Johnson (1993) argue that objects are intuitively more like co-operating, real-world machines from the student's point of view. This should be advantageous, since the effect is to raise the level of abstraction for novice programmers at a time when they need it most (Osborne & Johnson, 1993). However, there is still vigorous debate about the issue of teaching Objects-Early versus Imperative-Early. Koffmann commented on the introduction of object-orientation into courses, stating that it reinvented the 'new math' syndrome and that many practitioners were not aware that they had (Astrachan et al., 2005). Gabriel, commenting in the midst of the academic debate that followed Dehnadi and Bornat's findings, asks what it means when a programming paradigm fails: "it can fail when the narrative it embodies fails to speak truth or when its proponents embrace it beyond reason" (Gabriel, 2002, page 2).

While Object-Orientation is important, is the paradigm still too abstract for beginning students? What features are most desirable in a teaching language? Students who have experience with at least one programming language often perform better on introductory courses (Hagan & Markham, 2000). They propose that the more languages a student is familiar with, the steadier their overall performance is over the duration of their studies.

RESEARCH METHODOLOGY

This research presents a framework for classifying and analysing the concepts that are taught and the order in which they are presented by employing terms organised into a custom ontology. Ontologies are common vocabularies of terms built by researchers to enable them to classify and share codified information about a problem domain (Noy & McGuinness, 2001). Ontologies are built using well-defined and clear rules so that concepts can be codified as classes. Figure 3 is an extract of the final ontology which is available for download from our public GitHub repository (Dowdeswell, 2024).

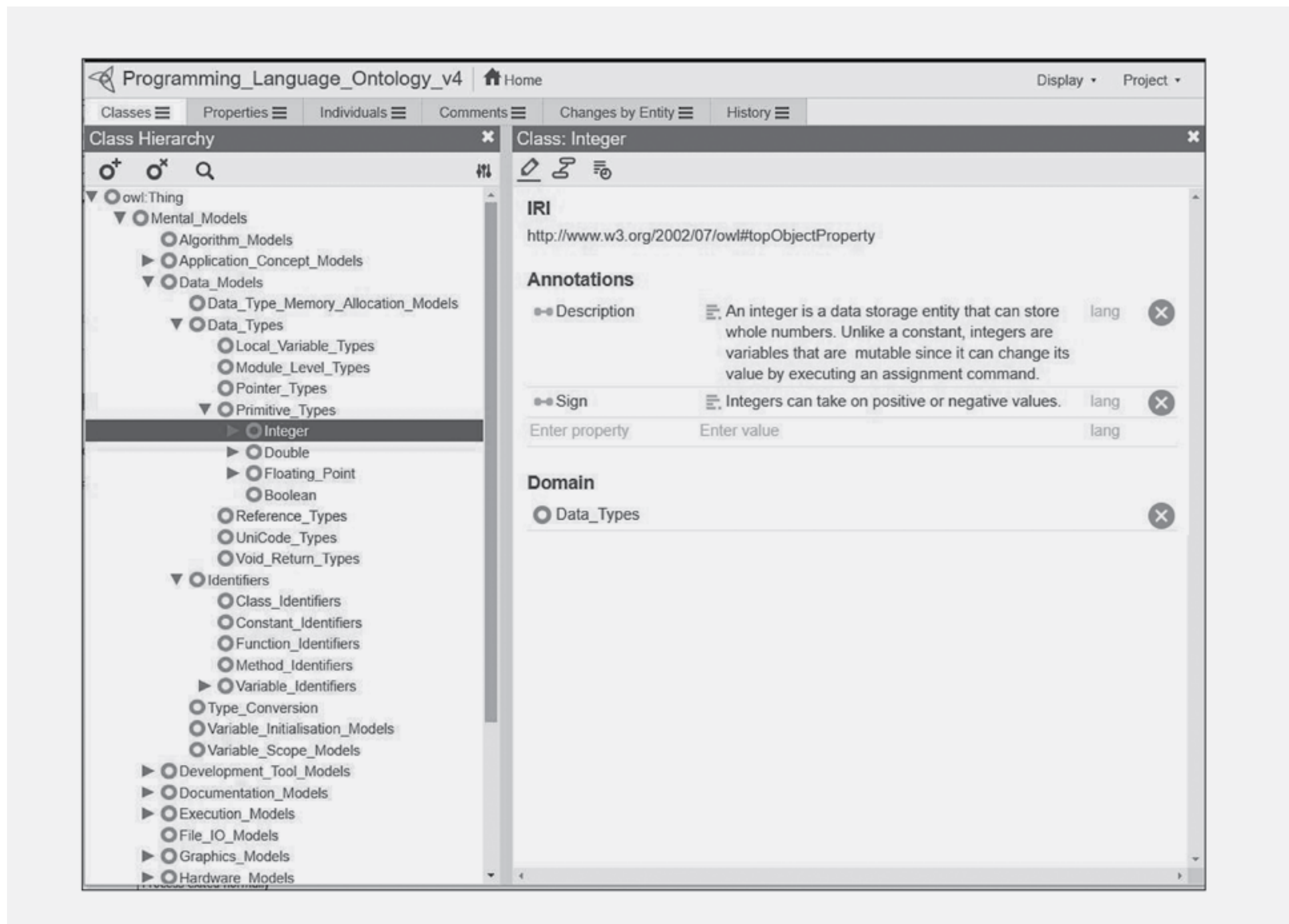
Course material consisting of lecture notes, textbooks, and hand-outs from three university first-year programming courses were obtained. A pilot study was conducted using Auckland University of Technology (AUT) lecture material from all 28 lectures that were delivered in a single semester (Skene, 2014). Each lecture transcript and presentation slide was analysed and summarised into a brief paragraph. OPAIC, AUT, and Stanford also provided extensive background information on the material and how it was developed over time.

Stanford also provided extensive background information on the material and how it was developed over time. AUT's courses catered for 350 students per semester. OPAIC offers two introductory courses, Programming 1 and Programming 2, which are often taken consecutively by students who are predominantly international and have English as their second language. OPAIC classes are smaller, with no more than 30 students in each cohort. A third set of international lecture notes, videos, and hand-outs were obtained from Stanford University. Their undergraduate classes are some of the largest in the world with over 650 students in their CS106A course (MacKay, 2014). Stanford also offered their CS106X course that only teaches C++, focusing on game design. It is targeted at more advanced beginners.

Stanford University's Protégé Ontology Editor was used to capture the concept classes. It has a built-in code generator that can take a´ collection of prototype classes and classify them using automated processes and custom rules (Stanford, 2024). Once an ontology has become sufficiently rich and mature, it is possible to automate the mining of large amounts of textual data. This codification speeds up the analysis of the underlying structure for large amounts of textual data. Each of the key concepts identified in the AUT University pilot study represented a potential candidate for inclusion in the ontology. One of the tasks in the pilot analysis was to identify topics which were mentioned in sufficient detail to be considered important. These represent thresholds between discussions that are not highly significant in relation to the curriculum but are present in the transcripts and those that are considered to be key concepts.

Ethics Approval was discussed initially for this research, but an Ethics Committee Approval was not sought since individual students were not involved or interviewed during the data collection and analysis. No individual student assignments or marks were provided by the faculties.

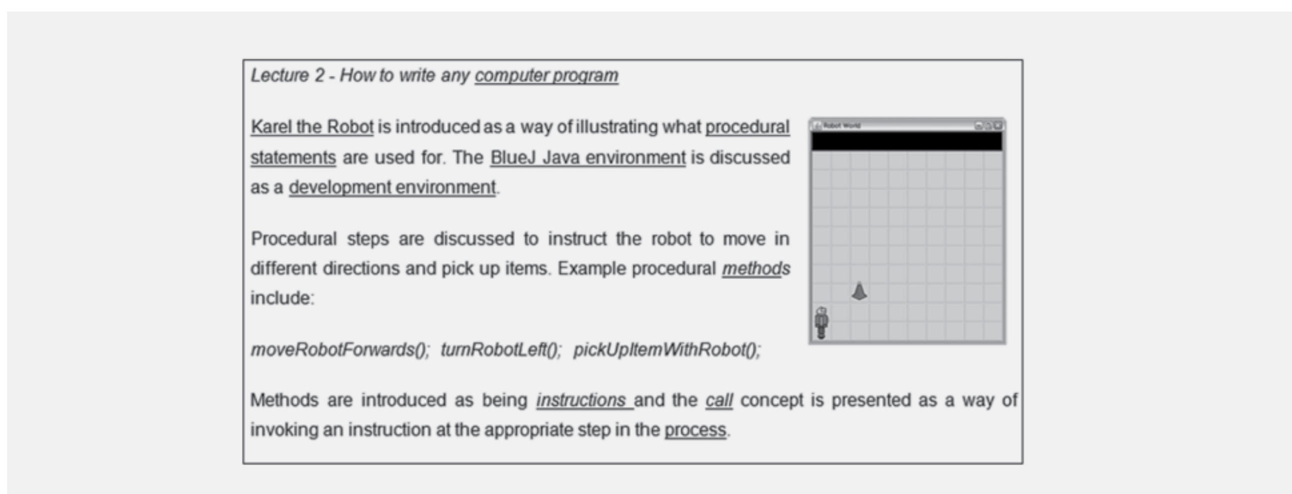
Figure 3: Example Classes from the Ontology in Protégé showing the properties of an individual class



COMPARING COURSES: THREE CASE STUDIES

Robins (2010) suggests that faculties should pay close attention to the order in which related topics are presented. Hence the order in which each concept was introduced in a course, how it was developed, and how often it was reinforced later was of particular interest. Do courses that are more successful introduce concepts in different, unexpected orders? Are particular concepts represented more intuitively in some programming languages than others?

Figure 4: Karel the Robot, used to introduce procedural concepts at AUT and Stanford



Ontology concepts were identified within the context of discrete topics and the lecture they were presented in. The full comparison available from the repository compares the three institutions, showing how these concepts were developed as the course progressed. Many key concepts were illustrated with reference to Karel the Robot, shown in an extract from the course notes in Figure 4. Karel was a programmable entity used in many code examples. Instructing Karel to perform a task illustrated procedural statements, methods, and function calls. However, it was not explained at this stage that Karel was an object. In this context, the students used Karel as a tool with multiple capabilities without needing to know that it was an object.

During the pilot studies, several example ontologies from different disciplines were examined. The most promising example was that of Lee et al. (2005) who developed an ontology for the Java language. They codified entities present in the Java language by classifying each of the language constructs for branching, control, variable declaration, and method construction separately. In our study, the concepts that students were being taught were more important than the actual implementation of the language statements. Hence the classes evolved to represent Mental Models of the concepts that the students were being exposed to. For example, when two different instances of variable declaration statements were encountered in lectures, such as the BASIC Dim aNumber as Integer or the Java int aNumber = 22; both were codified as members of the Integer class in the Variable Identifiers branch of the ontology tree. Instead of being language-specific, the ontology abstracts language features with a granularity that is sufficient to allow both the Java and the equivalent BASIC or C statements to be identified using the same classes. In this way, each class codifies a separate Mental Model for each teaching concept.

Noy and McGuinness (2001) point out that ontologies are seldom written in one pass. Rather, draft ontologies are trained and refined by testing them against sample data until they begin to more accurately classify concepts. This was observed while building the current version of the Programming 1 and 2 ontology that was used here. Generating it required four iterations, adapting it by generating similar classes in Protégé´ that represented the mental models of the concepts that the students were being exposed to. The most common change was to rename a class to make its purpose more obvious or to move it to a more logical branch of the ontology tree. Abstract concepts such as Inheritance, Recursion, Polymorphism, and Data Hiding moved several times, as did the data type classifications for variables.

ANALYSIS AND CONCLUSIONS

The original research question was concerned with identifying the optimal order in which concepts should be introduced in first-year programming courses. Table 1 is an extract from the comparison of the three courses analysed using the ontology. For each institution, the classification codes of the lecture content taught in that week are shown in parallel. A brief summary of the content discussed in the lecture is included in Table 1 below.

All three faculties begin their courses by presenting similar content, focusing on variables, procedural programming, and conditional control statements. Later, OPAIC and Stanford teach method calls by value and reference in Week 9 while AUT teaches it in Week 14. Objects are explicitly discussed in Week 4 at Stanford, in conjunction with inheritance. By Week 5 at Stanford, classes are presented as being complex variables while AUT defers this until the second half of the course in Week 22. The emphasis in both AUT and OPAIC is to develop procedural concepts that allow conditional control structures to be used simply. Karel was used at AUT to illustrate how to issue commands via program statements. Looping constructs are taught as a way to allow Karel to repeat steps. The value of understanding syntax is emphasised in AUT's system more emphatically than the Stanford approach, where they teach graphics methods and coding examples in-parallel with their Karel illustrations. By Week 7, AUT has taught clear examples of control structures and ways of writing code that instructs Karel. By the same point, Stanford is demonstrating how to invoke object methods and explains return values.

Whenever an Object-Oriented concept was identified, the class is highlighted in red in the analysis. Stanford is Object-Early; they begin teaching objects in their fourth week. This suggests they align with Osborne and Johnson (1993), believing that objects are more intuitive when taught at the same time as fundamental procedural code constructs. In contrast, OPAIC and AUT are Object-Late and do not explicitly mention objects until Weeks 12 and 17 respectively. The approach taken in the early OPAIC C# lectures is to use objects such as the intrinsic Random object as well as the Integer and String classes without identifying them as objects. Students learn to use parsing methods for type conversion without having to know that they are calling class methods.

Table 1: Extract of the First-Year Programming Course Comparison from the Three Institutions
Italics to provide additional contextual information that may not always be obvious from the choice of ontology classes applied.

LEC	AUT UNIVERSITY	OTAGO POLYTECHNIC	STANFORD
1	Procedural_Programming_models Computer_Models Hardware_Models Stored_Program_Models Programming_as_a_Process_ Models *AUT introduces programming as being like the steps in making a cup of tea.*	Programming_as_a_Process_ Models Visual_Studio_IDE_Models *Explanation of the course and how it flows. Illustrating what programming is by considering the steps required to make a cup of tea. Writing the first program Hello_world*	Karel_the_Robot_Models *Stanford is an introductory lecture with mostly administration and protocols*
2	Procedural_Programming_Models Karel_the_Robot_Models Immediate_Command_Models BlueJ_IDE_Models Call_Statements Langauge_Syntax_Models Conditional_Expressions If_Structure Loop_Structure *AUT introduces program commands without mentioning variables.Illustrates the effect of immediate commands and conditional tests.*	Procededural_Programming_Models Variable_Identifiers Data_Types Langauge_Syntax_Models *Becoming confident in Visual Studio and writing programs that run in the console, not the GUI.Declaring and assigning variables.*	Algorithm_Models Stored_Program_Models Karel_the_Robot_Models Conditional_Expressions *Introduces to conditional expressions using Karel*
3	Hardware_Models Primitive_Types Java_Langauge Assembly_Language *Comparing langauge construcs and attributes*	If_Structure *Operator precedence and evaluating boolean conditions for if conditions.* *Formatting variables.* *Casting*	Karel_the_Robot_Models Java_Langauge In-Line_Comments Conditional_Engineering_Models Quality_Models *Introduction to Java and quality concepts.*
4	Text_Identifiers Semi_Interpreted_Langauge_Models Debugging_Models *First mention of debugging to find errors.*	Constants_Models *Coding styles and displaying information correctly*	Historical_Background_Models Compiler_Models Object_Orientated_Models Inheritance_Models Java_Langauge Decomposition_Models *Introduces Object Models and Java as an object-orientated langauge, including inheritance.They also examine their first complete Java program.*
5	Langauge_Syntax_Models *Why syntax and formatting is important.How it relates to control structures*	Loop_Structure Random_Numbers. *Using Random numbers and loops to generate Values that can be used in simple dice roll games.*	Variable_Identifiers Data_Types Classes_as_Types_Models Objects_as_Variables_Models Method_Invocation_Models Graphic_Models Coordinate_Models Expressions *Objects and complex variables. Introduce methods that act on classes that manage graphics.*
6	Control_Structure_Models Indentation Nesting *Why indentation and formatting is important how it relates to control structures.Nesting control structures and flow.*	If_Structure Loop_Structure Logical_Operators *Branching and logical conditions. Introducing or and AND NOT conditions.*	Precedence_Rules_Expressions Floating_point Boolean Type_Conversion Variable_Scope_Models Conditional_Expressions If_Structure Nesting *Variable types and operator precedence.*

This explicit introduction to objects at Stanford exposes students to abstract concepts very early while they are still learning procedural coding. AUT and OPAIC appear to be more evenly paced, teaching similar concepts which are related to each other at the same time. Concept acquisition in technology courses relies on Robins (2010) concept of the Learning Edge Momentum effect. Students learn on the edges of what they already know, so teaching similar concepts closer together helps to make them progressively less abstract as more and more concrete concepts are understood. However, this momentum can also be lost if unrelated concepts are taught together since learners fail to understand how they are related. Positive examples from the study include the first introduction to variables which is immediately reinforced by learning about integers and doubles as examples of related Data Types.

Osborne and Johnson (1993) stress that becoming object-orientated is not about learning more syntax rules. Rather, it introduces abstractions that require a whole new way of thinking. Instancing, references, polymorphism, and inheritance are deeply abstract concepts. It is interesting to consider how early Stanford explores these topics while many students are still wrestling with procedural statements. Osborne observes that in his classes, abstraction is only presented after students have had concrete experience with the underlying concepts. This reinforces the OPAIC and AUT belief that static objects can be successfully taught without needing to introduce instances and references early on.

In conclusion, the analysis suggests that the way concepts are presented is more important than the choice of language. Much of the early literature cited about migrating to Java describes difficulties with the language that may no longer be an issue today. Java has evolved to incorporate many of the features of C and C++ without the need to introduce pointers. Even in 2006, the study by Chen et al. (2006) found few differences in what they called the outward layout or surface characteristics in the design of students' programs. Further, these differences were not attributable to Object-Early or Imperative-Early paradigms. Rather, the literature suggests that the problems lie with both the order and the way topics are taught and how well the hierarchy of related concepts is delivered.

Nicklaus Wirth, the creator of the Pascal language, asserted that people seem to misinterpret complexity as sophistication (Wirth, 2002, page 2). That bears further consideration in this context. Wirth went further stating that "As computing professionals, it is our duty to speak up against a culture that equates computer literacy with mastering the intricacies of a production programming language".

FUTURE RESEARCH DIRECTIONS

This topic suggested a number of future avenues of research, some of which may prove to be quite extensive. OPAIC runs courses in blocks rather than semesters with smaller class sizes. Do shorter courses with smaller classes have a significant effect on students' learning and retention of concepts?

The programming ontology provided a valuable way of classifying the content of the material examined. However, it was not possible within the scope of this article to use the classes to analyse what assignment and examination questions were really testing. This could then be used to correlate the anonymised student marks to gauge the success of teaching individual concepts identified in the curriculum. Analysing this over a number of iterations of a course while factoring in course changes would provide valuable insights.

Modern software engineering addresses the needs of a range of computing platforms, including desktop, web, and mobile. How should we address the differing characteristics of these technologies when students' progress from their first courses to learn about programming in other environments?

The authors would like to gratefully acknowledge the help provided by Dr James Skene for providing access to his AUT course material.

REFERENCES

Astrachan, O., Bruce, K., Koffman, E., Kolling, M., & Reges, S. (2005). Resolved: objects early has failed. *ACM SIGCSE Bulletin, 37*(1), 451–452. https://doi.org/10.1145/1047124.1047359

Balena, F. (2004). Programming Microsoft Visual Basic. *NET Version 2003.* Antonio Faustino.

Bantchev, B. (2008). *The True 'True BASIC'*. http://www.math.bas.bg/bantchev/misc/ttb.html

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming: 12 years later. *ACM inroads, 10*(2), 30–36. https://doi.org/10.1145/3324888

Chen, T.-Y., Monge, A., & Simon, B. (2006). Relationship of early programming language to novice generated design. *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, 495–499. https://doi.org/10.1145/1121341

Dahl, O. J., & Nygaard, K. (1966). SIMULA: an ALGOL-based simulation language. *Communications of the ACM, 9*(9), 671–678. https://doi.org/10.1145/365813.365819

Dale, N. B., & Weems, C. (2014). Programming and problem solving with C++.

De Raadt, M., Watson, R., & Toleman, M. (2002). Language trends in introductory programming courses, 1–9.

DeClue, T. (1996). Object-orientation and the principles of learning theory: a new look at problems and benefits. *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, 232–236. https://doi.org/10.1145/236452.236546

Dehnadi, S., & Bornat, R. (2006). The camel has two humps (working title). *Middlesex University, UK*, 1–21.

Dowdeswell, B. (2024). Object-Early vs Object-Late Dataset version 5. https://github.com/badger-dowdeswell/Object-Early-vs-Object-Late

Gabbrielli, M., & Martini, S. (2010). *Programming languages: principles and paradigms*. Springer Nature. https://doi.org/10. 5555/1805886

Gabriel, R. P. (2002). Objects have failed - notes for a debate. *OOPSLA Debate November*.

Gosling, J., Joy, B., & Stelle, G. (1996). The Java Language Specification.

Hadjerrouit, S. (1998). Java as first programming language: a critical evaluation. *ACM SIGCSE Bulletin, 30*(2), 43–47. https://doi. org/10.1145/292422.292440

Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, 25–28.

Hasker, R. W. (2002). HiC: a C++ compiler for CS1. *Journal of Computing Sciences in Colleges, 18*(1), 56–64. https://www.researchgate.net/publication/234821062_HiC_a_C_compiler_for_CS1

Horstmann, C. S. (2021). *Core Java: Fundamentals, Volume 1*. Pearson Education.

Kemeny, J. G., & Kurtz, T. E. (1964). Basic: A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System. http://bitsavers.trailing-edge.com/pdf/dartmouth/BASIC Oct64.pdf

Kiper, J. D., & Abernethy, K. (1996). Language Choice for CS1 and CS2: Experiences from Two Universities. *Computer Science Education, 7*(1), 35–51. https://doi.org/10.1080/0899340960070103

Kolb, A. Y., & Kolb, D. A. (2005). Learning styles and learning spaces: Enhancing experiential learning in higher education. *Academy of management learning & education, 4*(2), 193–212. https://doi.org/10.5465/amle.2005.17268566

Lee, M.-C., Ye, D. Y., & Wang, T. I. (2005). Java learning object ontology. *Fifth IEEE International Conference on Advanced Learning Technologies (ICALT'05)*, 538–542. https://doi.org/10.1109/ICALT.2005.185

MacKay, R. F. (2014). The Stanford Programming Classes, the bigger the better. http://engineering.stanford.edu/news/stanfordprogramming-class-bigger-better

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming - views of students and tutors. *Education and Information technologies, 7*, 55–66. https://doi.org/10.1023/A:1015362608943

Noy, N. F., & McGuinness, D. L. (2001). Ontology Development 101: A guide to creating your first ontology.

Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., & Aldrich, J. (2014). Safely composable type-specific languages. *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014*. Proceedings 28, 105–130. https://doi. org/10.1007/978-3-662-44202-9 5

O'Regan, G., & O'Regan, G. (2012). History of programming languages. *A Brief History of Computing*, 121–144. https://doi.org/10.1007/978-1-4471-2359-0_9

Osborne, M., & Johnson, J. L. (1993). An only undergraduate course in object-oriented technology. *ACM SIGCSE Bulletin, 25*(1), 101–106. https://doi.org/10.1145/169070.169358

Robins, A. (2010). Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education, 20*(1), 37–71. https://doi.org/10.1080/08993401003612167

Robins, A. (2012). Learning Edge Momentum. In *Encyclopaedia of the sciences of learning* (pp. 1845–1848). Springer US. https: //doi.org/10.1007/978-1-4419-1428-6 1716

Saidova, D. E. (2022). Analysis of the problems of the teaching object-oriented programming to students. *International Journal of Social Science Research and Review, 5*(6), 229–234. https://doi.org/10.47814/ijssrr.v5i6.418

Severance, C. (2012). The Art of Teaching Computer Science: Niklaus Wirth. *IEEE Computer Journal, 45*(7), 8–10. https://ieeexplore.ieee.org/document/6228569

Skene, J. (2014). Auckland University of Technology (AUT) Programming 1 Lecture Notes and Handouts.

Stanford. (2024). The Protégé´ Open-Source Ontology Editor. https://protege.stanford.edu

Stroustrup, B. (1986). An overview of C++. *Proceedings of the 1986 SIGPLAN workshop on Object-Oriented programming*, 7–18.

Van Rossum, G., et al. (1999). Computer programming for everybody. *CNRI: Corporation for National Research Initiatives, 1*.

Van Rossum, G., & Drake Jr, F. L. (1995). Python reference manual Centrum voor Wiskunde en Informatica Amsterdam. *Computer Science Department of Algorithmics and Architecture, Report, CS-R9525*.

Wirth, N. (2002). ACM SIGCSE *Bulletin, 34*(3), 1–3. https://doi.org/10.1145/637610.544415