Understanding Conceptual Transfer in Students Learning a New Programming Language

Ethel Tshukudu

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School Of Computing Science
College of Science and Engineering
University of Glasgow

June 2022

Abstract

There is a large literature from at least as early as 1985 on the difficulties encountered in learning programming languages, and in particular additional programming languages. This thesis concentrates on how students transfer their knowledge from their first programming language to their second. The central idea is to adapt and use theories from linguistics of how people learn second natural languages to illuminate the problems of learning second programming languages. The major claim of this thesis is that: Semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages; the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.

This thesis uses mixed methods to investigate how students transition from procedural Python to object-oriented (OO) Java. It includes a sequence of nine research studies building on each other. First, an exploratory qualitative study is carried out on how semantic transfer in natural language applies to programming language transfer; secondly, a Model of Programming Language Transfer (MPLT) is developed based on the first study's findings; thirdly, four quantitative studies are carried out to validate the model; fourthly, a study that collects school teachers' views and experiences on second language learning is carried out; fifthly, a study is conducted to explore transfer interventions with students; and the last study builds and investigates a pedagogy for transfer deriving from the MPLT.

The findings support the thesis claim that semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages. The transfer can be positive when the first programming language (PL1) and the second programming language (PL2) share similar syntax and semantics, negative when PL1 and PL2 share similar syntax but have different semantics, and there is little or no transfer when PL1 and PL2 have different syntax but share similar semantics. The results also reveal that transfer teaching interventions based on the MPLT could improve conceptual transfer and understanding in students learning a second PL.

The contribution of this thesis is two-fold: First, a validated model of programming language transfer that has three categories that reflect the types of potential transfer students encounter when learning a second programming language. The model provides a unified way to measure

transfer in second language learning. Second, a validated unified pedagogical guideline for promoting transfer in programming languages derived from the MPLT. Researchers, educators and curriculum designers can use these instruments to advance research, teach, and design teaching materials. First, the researchers can use the instruments to further programming language transfer research by adopting them in other programming language contexts. Second, educators can use the instruments as a guideline for improving second and subsequent programming language teaching. Lastly, Computer Science (CS) curricular designers can draw on these instruments as guidance to design teaching material that promotes transfer as students transition to new programming languages. They can also use them for teacher professional development.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to appreciate and thank God for granting me grace, strength, and knowledge to accomplish the thesis finally.

I want to thank my esteemed supervisor, Professor Quintin Cutts, for his invaluable supervision, support, knowledge, and guidance during my Ph.D. journey. I want to thank him for giving me the opportunity and trust to pursue a Ph.D. degree under his supervision. Professor Cutts, thank you for teaching me to be a good researcher. Your advice and the lessons are invaluable. Thank you for providing cheerful moments when times were hard, especially during the Covid-19 pandemic.

Special thanks to all the lecturers that supported me, provided me with access to their students, and were willing to assist with implementations of my interventions in their classrooms: Dr. Mary Ellen Foster, Ms. Siri Annethe Moe Jensen, Dr. Matthew Barr, and Dr. Gerardo Aragon Camarasa. Without such support, this research would have been impossible.

I want to give a big thank you to all my research participants (the teachers and the students). Your support is invaluable.

Special thanks to Professor Felienne Hermans and Dr. Steve Draper, who supported me and gave their valuable time to review and provide constructive feedback on my thesis. Professor Felienne Hermans, thanks for your mentorship, for believing in my work, and for always being eager to do research collaborations with me. Dr. Steve Draper, thank you for showing interest in my work and providing a platform for stimulating discussions. You have both been a great source of encouragement.

To my co-authors and research collaborators, thank you. Collaborating with you gave me new knowledge, skills, techniques, and new friends and mentors in the academic field.

I would like to thank Dr. Sue Sentance for her support and encouragement. She is one of the most influential people in my life. She has had a positive impact on me academically and personally.

I thank my fellow CCSE lab colleagues from the University of Glasgow. Thank you for the support, advice, stimulating discussions, and refreshing hangout sessions.

Last but not least, I would like to give my most enormous thanks to my family. Firstly, my husband, Thabo Tshukudu, has shown the most significant support during my Ph.D. journey. I thank him for his selfless sacrifice of quitting his job to come with me to the UK to support my

Ph.D. endeavor. I thank him for his patience and tolerance over the last four years. Thabo, I could not finish this work without your support. Thank you for being proud of me for what I accomplished and always looking for ways to help bring out the best version of myself. Secondly, I would like to thank my children for their support, prayers, and patience. I apologize for not spending enough time with you while writing my thesis.

I dedicate this work to my late mother Priscilla Ndiko Daniel.

Declaration

I declare that all the work in this thesis was carried out by the author unless otherwise explicitly stated.

Chapter 1

Introduction

Students learning to program in K-12 (primary to secondary school) and higher education are not only faced with the challenge of developing and implementing solutions for problem-solving, but they also have to understand the programming languages (PLs) they use to solve different problems. The research that attempts to investigate and understand the source of these difficulties has reported that students struggle with syntactic, semantic, clerical, and logical errors when writing programs [1, 2]. Prior work has also reported that, in the early stages of learning programming, students tend to struggle more with the syntax of programming language [2, 4], therefore focus more on the syntax than on understanding the program's function [3]. Stefik et al. [4] reports that some of the reasons students struggle with syntax are the types of word choices and symbols used in programming languages which may not be intuitive or easy for them to understand. For example, they found that for the programming concept of iteration, programming language keywords that are common in English and literal such as `repeat` are more intuitive for students than keywords such as `for`.

Efforts to mitigate these challenges have resulted in the evolution of multiple programming languages explicitly designed to ease how students learn to program. The various programming languages introduced in the K-12 and higher education classrooms result in students transitioning from one PL to the other during their education. For example, students may start learning to program using block-based languages and then transition to text-based languages later on or transition between paradigms within text-based languages.

Research relating to the learning of second and subsequent programming languages has reported the difficulties experienced by students when they transition. The challenges have been reported in the context of block-based languages to text-based languages [11,18,29,30], paradigm shifts [2,16] and within text-based languages [15,16,20–22,33]. This line of research has largely not been theory-driven, hence not describing the cognitive processes and mechanisms involved in learning new PLs. Given that students learn different sets of common underlying PL concepts during their education [5, 6], the overarching question, therefore, is how do students transfer their understanding of concepts between PLs? Uncovering the mechanisms involved in

PL transfer is a necessary step for a detailed understanding of second PL learning and assists educators in understanding the success and failure of transfer in the classroom.

The following introductory section lays out the prior research relating to transfer in programming languages. This is followed by the objectives and statement of this thesis. Finally, the structure and content of the forthcoming chapters are presented.

1.1 Transfer in Programming Languages

Prior work has explored aspects of novice programmers transferring knowledge when learning their first PL. For example, students' understanding of natural language may interfere with their understanding of the meanings of programming language constructs when learning their first PL [4, 174–176]. Prior knowledge of mathematics can also interfere with learning programming concepts; for example, the concept of a *variable* is used in both mathematics and programming but has significantly different meanings in each [177]. These interferences could occur because when learning the first PL, students learn new concepts and use their intuition to try and understand these concepts. Intuition relies on the *unconscious* transfer of previously acquired knowledge and therefore is not apparent to the student during learning. This transfer has been seen as an essential source of error in computing education, leading to misconceptions in understanding programming concepts as summarised in [176]. Stefik et al. [4] explored which words (e.g. the word `for` that represents iteration) and symbols (e.g. `++` in the `for loop`) novices find intuitive in PLs. Their study found that variations in the first PL syntax matter to novice programmers such that they find some syntax more intuitive than others. For example, the novice programmers rated `x=x+1` as more intuitive than the shorthand `x++`, which the experienced programmers found intuitive.

By comparison, however, there is minimal research on how students move on to learn a second and subsequent PL when they already have established conceptual knowledge from their first programming language. Finding the answer to this question is particularly important because several studies reported transition challenges relative novices face as they switch between programming languages [8–17, 178].

Early work on transfer to second and subsequent programming languages mainly focused on experiments that observed experienced programmers solving programming problems in a new language. Participants in these studies were reported to start solving problems using familiar plans from the prior language. This transfer was negative when they had to implement the plans in a new language that had different constructs from the prior language [19, 20]. However, it was positive if the new language had similar constructs as the prior language [21].

For instance, Scholtz and Wiedenbeck [19] reported negative transfer of plans on experienced programmers with C and Pascal knowledge planning in the new Icon language. They discovered that in the early stages of learning a new language, experienced programmers use

a top-down and depth-first approach to solve a problem [22]. Programmers then change their plans repeatedly when they reach implementation as they become familiar with the new language constructs, resulting in delays in solving the problem. This fits with the observations of transfer of plans made by their later studies [20]. However, in investigating novice programmers, Scholtz et al. [23] reported that novice programmers tend to use a bottom-up approach in solving problems in a new language.

Wu and Anderson [21] reported that experienced programmers experienced the positive transfer of plans between LISP and PROLOG and between LISP and PASCAL. Participants were reported to use the familiar algorithms in the first language to cut down the planning in the second language, which reduced the number of revisions needed to finalize the plan. They concluded that the positive effect was because of the three languages' commonalities in some concepts, such as recursion. Syntactic interference was reported to be minor.

Computer science education researchers have tried to address essential questions of how transfer occurs when problem-solving in new languages, thus not focusing on the interaction between the linguistic elements of known PL and new PL in the programmer's brain when reading code in a new language. Learning programming is more than problem-solving and plans; it is also about understanding the underlying programming language constructs used to write programs and how they behave when executed. In addition, prior work gives little attention to how relative novices move on from their first or second language, considering that novice programmers often only have limited knowledge of plans [23, 24]. When learning to program, novice programmers tend to focus more on the syntax of a programming language [3, 4] which is intimately tied to their understanding of programming language constructs [6]. Therefore, this thesis is interested in investigating how a novice programmer's knowledge of concepts in their first PL transfers to learning a second PL. This research draws from second language acquisition models in natural languages to understand programming language transfer as a starting point.

1.2 Transfer in Natural Languages

Jiang [25, 26] has developed a model of lexical development in second language acquisition, explaining how the lexical entries in the second language lexicon evolve. A lexicon is the collection of all words in a language in linguistics [201]. Each word in the lexicon is referred to as a lexical item and is described in the dictionary by a lexical entry [201]. In turn, each lexical entry comprises information about the lexical item and is divided into two components: the lemma and the lexeme [25]. The lemma consists of the syntax and semantics components (e.g., parts of a sentence and their meaning), and the lexeme consists of the morphological and formal components (e.g., how a word is formed and its context, spelling, and pronunciation). Further clarification of the lexeme and lemma components with examples is presented in Section 2.5.

This thesis focuses on the lemma because syntax and semantics apply to programming lan-

guages. Programming languages, just like natural languages, use syntax, which is a set of rules that inform us how to combine words and symbols to create well-formed sentences/programs. One crucial aspect emphasized by Jiang is that the lexical representation components (in this research context, only syntax and semantics) are highly connected. The activation of one component in the mind of a language learner results in automatic simultaneous activation of the other components.

Jiang's model has provided valuable insights into how learners learn new languages based on their first languages. Empirical work has confirmed the validity of this model in the natural languages context [25–27]. One of the ways that lexical association may occur when learning new languages is through cross linguistic-similarities. Ringbom [28] proposes that comprehension of a new language can start with perceiving lexical similarities to elements of the language they already know, which is followed by the assumption of the associated semantic or functional similarity.

From the theories of natural languages reported above, it seems clear that when learners learn their second language, the semantic transfer will occur [25–27]. In some cases, the semantic transfer can be influenced by cross-linguistic similarities [28].

1.3 Thesis Statement

Given the semantic transfer principles in natural languages research, the major claims of this thesis are as follows: **Semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages; the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.**

This research uses Jiang's definition of semantic transfer. They define that semantic transfer occurs when the semantic content in a second language word is transferred from the first language [26]. It means that the first language semantic structures influence the second language semantic development in the mind of a language learner. In the context of this research, a relative novice programmer is a programmer who has knowledge of one programming language and has, on average, one year or less of programming experience.

1.4 High-level Research Questions

In order to investigate the hypothesis about semantic transfer in the thesis statement, high-level research questions were designed. The first research question addressed whether the semantic transfer principles in natural languages are also applicable to the context of programming language transfer. Understanding how relative novice programmers transfer in PLs can be help-

ful for educators to better account for the success and failure of PL conceptual transfer in the teaching and learning of second and subsequent programming languages.

- **RQ1: How are principles of semantic transfer in natural languages applicable to patterns of transfer in the context of relative novices transferring from first to subsequent PLs?**

The qualitative study yielded results that led to the development of a Model of Programming Language Transfer (MPLT) tailor-made for relative-novices transfer in programming languages. This model is used to investigate transfer in programming languages quantitatively. Therefore the second research question was:

- **RQ2: Do relative-novices transfer their programming language conceptual and semantic knowledge to a new programming language during code comprehension as proposed by the MPLT?**

In addition to RQ1 and RQ2, this thesis explored how second and subsequent programming languages are taught, given that transition challenges reported in prior work are experienced in the classroom. This thesis investigated the transfer strategies and attitudes of teachers towards second language learning and teachers' awareness, from their own classroom experiences, of the transfer issues. Therefore the third research question was:

- **RQ3: How do teachers experience PL transfer for relative novices in the classroom?**

As mentioned in RQ1, understanding relative novices' transfer and how teachers approach transfer in the classroom can help educators account for the success and failure of transfer in the second PL classroom. Therefore, the last question considers how such an understanding could help build transfer interventions that can help improve second language learning in the classroom.

- **RQ4: How can transfer teaching interventions based on our understanding of semantic transfer improve second PL learning?**

RQ1 will be addressed in Chapter 4, and this chapter consists of an exploratory qualitative study conducted over ten weeks with students transitioning from procedural Python to object-oriented Java. RQ1 and RQ2 are addressed in Chapter 5 that involves the designing of the model of PL transfer based on the exploratory study results. Chapter 6 addresses RQ2, and it includes experiments that validate the model. RQ3 is addressed in Chapter 7 and consists of studies that explore teachers' experiences with the transfer. Finally, Chapter 8 and Chapter 9 address RQ4 and both consist of investigating transfer interventions.

1.5 Thesis Contributions

This thesis offers a number of contributions:

Model of PL Transfer: The first contribution is a model of PL transfer (MPLT) presented in Chapter 5, drawn from existing literature on programming and natural language transfer principles. The model has been empirically validated in several studies in students transitioning between Python and Java in Chapter 6. This model helps us answer RQ1 and RQ2. The model demonstrates the role of syntax similarity and subsequent semantic transfer as a bridge to learning new PLs. It also offers an understanding of the relationship between PLs in the mind of a novice programmer. Furthermore, students transfer partial or no semantics to concepts that do not have similar syntax but have identical semantics. It demonstrates that semantic transfer mainly occurs from PL1 to PL2 but can also happen bi-directionally from PL2 to PL1 depending on experience and knowledge. The model can be used to explain the learning processes involved in language transfer and aid in guiding teachers who teach new programming languages.

Teacher transfer practices on second PL learning: The second set of contributions help answer RQ3 by reporting on the teacher transfer practices on second PL learning; See Chapter 7. The findings reveal that many teachers use simple programming languages initially, creating an opportunity for transfer learning. Furthermore, teachers reported both benefits and problems of transfer from PL1 to PL2, as predicted by the MPLT. Regarding transfer strategies, it was observed that teachers mostly don't see the need to implement and don't capitalize upon the opportunity transfer could bring. These findings opened up opportunities to build a pedagogy that can guide teachers in PL transfer to improve learning outcomes.

A pedagogy for PL Transfer: The final contribution is a second programming language pedagogical design rooted in both general and specific theory and a detailed study of applying this pedagogy in an actual classroom setting; See Chapter 9. This pedagogy helps answer RQ4. The pedagogy uses implicit, explicit, and bridging techniques aligned with the MPLT's predictions. This thesis investigated the effects of this pedagogy on the learning of concepts in a new language. Furthermore, the lecturer reported on their experiences of using the pedagogy. The findings suggest that the transfer pedagogy can benefit second language learning and can be of value in teaching second programming languages.

A summary of the contributions mapping between research questions, contributions and chapters is presented in Table 1.1.

Table 1.1: A summary table of the thesis contributions mapped to the chapters and research questions

Contributions	Chapter	Research questions
Model of PL Transfer	5, 6	RQ1, RQ2
Teacher transfer practices on second PL learning	7	RQ3
A pedagogy for PL Transfer	8, 9	RQ4

1.6 Publications

This section summarizes the published work that addresses the research questions. So far, this research has resulted in paper publications, conference presentations, reading group talks, seminar presentations, and research newsletter features.

Long papers and Long Abstracts

1. *Publication 1*: Ethel Tshukudu and Quintin Cutts. 2020. Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. 307–313. [29].

This publication presents the initial links between semantic transfer in natural languages and semantic transfer in novice programming context, laying the foundations for a model of PL transfer in CS which is the first major contribution of this thesis. This study addresses RQ1 and aspects of this publication are presented in Chapter 4.

1. *Publication 2*: Ethel Tshukudu and Quintin Cutts. 2020. Understanding Conceptual Transfer for Students Learning New Programming Languages. In Proceedings of the 2020 ACM Conference on International Computing Education Research. 227–237. [30]

This publication presents the first main contribution of this thesis, the model of PL transfer, which demonstrates the role of syntax similarity and subsequent semantic transfer when relative novices learn second and subsequent PLs. In addition, this publication presents two empirical studies that validate the model's hypothesis. This study addresses RQ1 and RQ2 and aspects of this publication are presented in Chapter 5 and Chapter 6.

In addition to the above, an additional Doctoral Consortium paper and a symposium paper presents a summary of the above publications. These were presented at the ICER conference and the Cambridge Computing Education Research Symposium, respectively. These studies address RQ1 and RQ2 and aspects of these publications are presented in Chapter 4 and Chapter 6.

1. *Publication 3*: Ethel Tshukudu and Quintin Cutts. [n. d.]. Understanding conceptual transfer in second and subsequent programming languages. In Cambridge Computing Education Research Symposium 18 [31].

2. *Publication 4*: (Doctoral consortium) Ethel Tshukudu. 2019. Towards a Model of Conceptual Transfer for Students Learning New Programming Languages. In Proceedings of the 2019 ACM Conference on International Computing Education Research. 355—356. [32].

Papers that presented the teachers' experiences (RQ3), initial transfer interventions, and the final pedagogy aligned with the model were published. These papers include qualitative and quantitative research methods. These studies demonstrate how interventions can help students transition to second or subsequent PLs and show the second and third thesis contributions.

1. *Publication 5*: Ethel Tshukudu and Siri Annethe Moe Jensen. 2020. The Role of Explicit Instruction on Students Learning their Second Programming Language. UKICER '20: United Kingdom Ireland Computing Education Research conference. 10–16 [33]. This study addresses RQ4 and aspects of this publication are presented in Chapter 8.

   [*This paper was co-authored with one other researcher: I led and guided the study by coming up with the research idea, research questions, study design, developing the data collection methods (e.g., transfer quizzes), data analysis, and discussion. My co-author, Siri Annethe Moe Jensen, contributed by reviewing the study material and implementing the study design in their classroom at the University of Oslo.*]

2. *Publication 6*: Ethel Tshukudu, Quintin Cutts, Olivier Goletti, Alaaeddin Swidan, and Felienne Hermans. 2021. Teachers' Views and Experiences on Teaching Second and Subsequent Programming Languages . In Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021), Au- gust 16–19, 2021, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446871.3469752. This study addresses RQ3 and aspects of this publication are presented in Chapter 7.

   [*This paper was co-authored with other researchers as follows: I led and guided the study by coming up with the research idea, research questions, study design, developing the data collection methods (e.g., interview scripts), interviewing the Scottish teachers, data analysis (thematic analysis), and writing the discussion of the results. My co-authors' contributions are as follows: Felienne Hermans, Olivier Goletti, and Alaaeddin Swidan contributed to the data collection by conducting the teacher interviews in the Netherland. In addition, they helped in conducting Thematic analysis (which includes data cleanup, coding, and creating themes). In addition, Felienne Hermans guided the study and also used her expertise to guide the design of thematic analysis methods. My supervisor provided the overall support and guidance.*]

3. *Publication 7*: Ethel Tshukudu, Quintin Cutts, Mary Ellen Foster. 2021. Evaluating a Pedagogy for Improving Conceptual Transfer and Understanding in a Second Programming

Table 1.2: A summary table of the thesis contributions mapped to the chapters and research questions

Publication	Chapter	Research questions
1. Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices.	4	RQ1
2. Understanding Conceptual Transfer for Students Learning New Programming Languages.	5, 6	RQ1, RQ2
3. Understanding Conceptual Transfer in second and Subsequent Programming Languages.	5, 6	RQ1, RQ2
4. Towards a Model of Conceptual Transfer for Students Learning New Programming Languages.	5, 6	RQ1, RQ2
5. The Role of Explicit Instruction on Students Learning their Second Programming Language.	8	RQ4
6. Teachers' Views and Experiences on Teaching Second and Subsequent Programming Languages.	7	RQ3
7. Evaluating a Pedagogy for Improving Conceptual Transfer and Understanding in a Second Programming Language Learning Context	9	RQ4

Language Learning Context . KoliCalling '21. This study addresses RQ4 and aspects of this publication are presented in Chapter 9.

[*The paper was recognised at the conference as* ***Best Paper Runner-up****. This paper was co-authored with two other researchers: I led the study by coming up with the research questions, study design, transfer pedagogy, data collection methods (e.g., transfer quizzes), data analysis, and discussion. My co-author, Mary Ellen Foster contributed by reviewing the study material and implementing the study design in their classroom at the University of Glasgow, and lastly my supervisor gave the overall review and guidance of the study.*]

The publication of these papers were at high quality peer-reviewed conferences in the research area hence providing a degree of value in the work of this thesis. The presentation of the details of these studies will be in the later chapters of this thesis. A summary of the publications mapping with the chapters and the research questions is presented in Table 1.2.

1.7 Thesis Structure

This section will provide the structure of the rest of this thesis. **Chapter 2** reviews the detailed literature on research in programming language transfer which touches on both the transition challenges and cognitive aspects of programming language transfer. Next will be the theoretical perspectives of second language learning from programming languages and natural languages research. The gaps in existing research work that this thesis aims to address are presented last. **Chapter 3** is presented next with a broad description of the philosophical underpinning to the

chosen research methods. It includes the study design, issues of validity and reliability, and ethical considerations. **Chapter 4** provides the analysis and findings of the first pilot study that explored the aspects of transfer based on the existing transfer theories from both natural and programming language transfer. **Chapter 5** presents the MPLT, drawn from existing literature on programming and natural language transfer principles and Chapter 4 study findings. **Chapter 6** presents four experiments that validate the MPLT. **Chapter 7** presents a study on teachers' experiences and transfer strategies. **Chapter 8 and 9** present interventions and pedagogy for transfer, respectively. Lastly, **Chapter 10** presents the discussions, contributions, implications, limitations and conclusions.

Chapter 2

Literature Review

This chapter provides a broad overview of the most relevant programming language and natural language transfer research. This thesis investigates how students transfer knowledge between programming languages; therefore, this chapter starts by reviewing the empirical studies that address transition challenges experienced by students who learn new programming languages and why investigating transfer matters. From there, empirical studies that specifically address knowledge transfer between programming languages are reviewed. As this thesis investigates PL transfer drawing from both programming languages and natural language theories, the next part of this chapter examines the theories on language transfer from programming languages and natural languages. This thesis also investigates if implementing transfer strategies based on semantic transfer improves the learning of a second PL; therefore, the last part of the chapter reviews empirical work that addresses pedagogical approaches in PL transfer and their effects on learning new PLs. The terms used in this thesis are used as they are explained in the literature.

2.1 Transitioning between Programming Languages

This section reviews empirical research that reported the challenges of programming language transition. This research is not particularly concerned with designing the underpinning theories behind programming language transfer but is concerned with gaining knowledge on the practical experiences and effects of second and subsequent programming language learning. This research reports various language transition experiences such as block-based languages (e.g., Scratch) to text-based languages, within textual languages, and between programming language paradigms.

2.1.1 Transitioning from Block-based to Text-based Languages

The first set of experiments discussed are researchers investigating the transition from block-based to text-based languages. Unlike text-based languages, which allow learners to write lines

of code to build a program, block-based languages enable learners to manipulate visual elements such as dragging and dropping blocks of instructions to create a program. Researchers have reported both positive and negative impacts of transitioning from block-based languages to text-based languages. One notable example is the study conducted by Powers et al. [34], which followed students who learned Alice for the first half of the semester, transitioning to Java in the second half of the semester. Students were given lab assignments and guided to write code in Java translating from an Alice program. The results showed that while prior Alice knowledge showed some benefits of increased confidence and retention, it also had some significant transition challenges. These were related to the students feeling intimidated by the Java textual language and syntax. In addition, the students could not recognize the connection between the Alice code and the Java code. Powers et al. [34] reported that the difficulty was due to the mismatch of the code organization in the two languages on concepts such as object declarations and methods. For example, students expected the creation of objects in Alice to be the same as in Java which was not the case; in Alice, objects are created in the IDE and then manipulated by code, which is not the case for Java objects.

In another study of block-based to text-based language transition, Armoni et al. [16] investigated students transitioning from Scratch (middle-school) to Java (high-school). The study investigated how students' learning of CS at secondary school is affected by their previous knowledge of CS learned from middle school. The study involved an group of 44 students who had taken a course in Scratch and a control group of 76 students who had not. Students wrote multiple tests covering reading and writing programs in C# or Java at different cognitive levels. The results showed no significant differences between the experimental and control groups in most concepts. Students in the experimental group recognized some concepts in the early days of learning, although the recognition was restricted to the form they learned them in Scratch. For example, explaining variables as a means to count game points. Furthermore, both groups encountered difficulties with some concepts such as static typing in C# or Java. Despite these challenges, the authors reported that students in the experimental group had higher levels of motivation and self-efficacy.

Weintrop and Wilensky [35] conducted a study to investigate the effect of different modalities (blocks-based versus text-based) on ninety novice programmers' understanding of basic programming concepts. The study was not particularly addressing transition challenges but compared students' scores side-by-side on a *Commutative Assessment* that includes short program questions represented in either a blocks-based or text-based form. The findings revealed that modality (e.g., text-based, hybrid, block-based PLs) affects novice programmers' understanding of basic programming concepts. This effect affects different concepts in different ways; it also does not seem to influence comprehension of programs in the same way it affects basic comprehension of what a construct does within a program. Students in the graphical condition performed significantly better with the blocks-based modality on questions related to iterative

logic, conditional logic, and functions. For example, students found it easier to understand the `repeat` in block-based environment as compared to a `for-loop` in a text-based language.

In a another study, Weintrop and colleagues [36] further investigated how introductory programming tools (block-based, text-based, and hybrid blocks/text) prepare learners for the transition to a text-based language (Java). Their study followed 90 students in an introductory programming course that used three introductory programming environments: Text, Blocks, and Hybrid condition in the Pencil Code platform [1] for the first five weeks before transitioning to Java for the remaining ten weeks. Their findings are based on the results of programs written by students in the Java portion of the class. They concluded that there were minor differences in the programming patterns adopted by novices as they transitioned from different programming environments to Java. They also discovered that students who used the Hybrid and Text programming environments made slightly more common errors during the transition than the Blocks programming environment. For example, one of the most common errors experienced when switching to Java was *"cannot find symbol"*, which occurred when students tried to use a *variable* before it had been defined. The authors suggest that students made assumptions based on how their previous programming environments worked; variables did not need to be instantiated before they were used. Their subsequent studies [37] reveal that the initial difference in conceptual learning that emerged after five weeks between the block-based and text-based programming environments fades after ten weeks of students transitioning to Java. Thus, this suggests that modality impacts learners' experiences with programming only in the early stages of transitioning to Java.

Kölling et al. [178] analyzed and reported that the challenges involved in the transition from blocks to text, among others, involve: readability (block-based environments are easier to read than text-based languages), memorization of commands and syntax (students struggle less with having to remember the syntax in a block-based environment because all available commands are visually represented on screen), typing/spelling (the necessity to type in a text-based language may add an extra cognitive load), matching identifiers (students deal with case-sensitive syntax for identifiers in text-based languages), understanding types (unlike in block-based programming environments, students need to understand data types for the construction of simple, meaningful programs in text-based languages) and interpreting error messages (students may struggle with error messages in text-based languages which are not shown in block-based programming environments).

Moors et al. [38] argued that although block-based programming environments are successful at motivating students and reducing the programming barriers, they can cause students to lose confidence when they transition to text-based languages. They argue that block-based environments can make students form bad programming habits, e.g., block-based environments allow students to leave individual blocks of instructions in the script area without affecting the

[1] https://pencilcode.net

execution of the main program.

Summary of Transition Challenges from Blocks-based to Text-based Programming Languages

So far, the focus has been on studies that mainly investigate the transition between block-based to text-based languages. The overall summary of these studies is the conceptual learning and efficacy gains in the early stages of transitioning from blocks to a textual language which starts to fade away at the end of learning the textual language. The above studies have reported some transition challenges related to syntax and misconceptions. These could result from the difference in modalities (graphical versus text) in representing code. Students learning in block-based environments usually are not expected to work with syntax; hence they may be intimidated by textual languages. The thesis aims to investigate how students would transition within the same programming modality (textual languages). The following subsection will present the studies addressing this question.

2.1.2 Transitioning between Text-based Languages

Some researchers reported on the challenges students face when transitioning between textual languages. An earlier study by Walker and Schach [12] observed six junior-level software engineering students transitioning from at least one programming language (e.g., Pascal) to Ada. At the beginning of the course, the teacher taught the students the Ada programming language syntax and constructs. Students were then asked to perform some problem-solving tasks in the Ada language. The authors then analyzed the program changes to deduce the programming style in the new Ada language. The findings were that students frequently used familiar constructs to their known Pascal language and ignored the features of the new Ada language. For example, some students opted to use the more familiar Pascal *while* construct (also available in Ada) instead of the *loop-exit-end* construct of Ada (not available in Pascal). The students who chose to start developing programs in the new Ada constructs quickly abandoned them and went back to familiar Pascal-like constructs when the programs failed.

In a different approach of students transitioning to a new programming language paradigm, Nelson et al. [11] followed 19 students with various computing backgrounds (from one semester to 10 years experience) through a semester to explore how they transition to object-oriented programming (Smalltalk). They identified that students took different paths in learning the new language. The common finding across all the students was that their previous programming experience helped them understand familiar programming constructs such as loops, branches, and iteration. The more experienced students struggled less with the new language's syntax than relative novices. The relative novices found their prior knowledge of procedural programming was inadequate to help them understand common concepts in the object-oriented language.

Table 2.1: A summary table of empirical work on transitioning between text-based languages

The study	Language Transition	Population size	Challenges identifies
Walker and Schach [12]	Pascal to Ada	6 students	Previous language interference (students frequently used familiar constructs of prior language and ignored new language features)
Nelson et al. [11]	Procedural PL to object-orieted Smalltalk	19 students (novices and experienced)	Previous procedural programming experience (helped some students' to understand 00 programming, gain confidence, interfered with OO learning)
Santos et al. [39]	Racket to Java	53 students	Previous language interference (students struggled with Java concepts that work differently)
Shrestha and colleagues [40]	Different programming languages	16 experienced programmers	Previous language interference (old habits of programming, little/no mapping of concepts to previous languages)

They abandoned making connections to prior knowledge resulting in their progress being much slower than experienced students. The experienced students were more interested in identifying similarities and differences between prior knowledge and new knowledge to help them make informed decisions when solving problems.

A recent study by Santos et al. [39] reported on observations in teaching object-oriented programming in Java to 53 students who previously learned functional programming in Racket the previous semester. They reported that students struggled with Java concepts that do not work the same way in Racket, such as *string concatenation, type systems, and if-statements*.

Prior work on language transitions has also addressed transition challenges specific to experienced programmers. Shrestha and colleagues [40] conducted an empirical study of 450 stack overflow questions addressing different programming languages. They found that 61% of the 450 posts contained incorrect assumptions about the new language, and only 39% had correct assumptions. The authors reported that programmers related syntax and concepts of the new language with their previous language, which was helpful but often caused interference when differences exist between languages.

Summary of transition challenges in text-based programming languages

Table 2.1 summarises the empirical work on transitioning between text-based programming languages. In Summary, the researchers reported that prior language knowledge affected the learning of second and subsequent languages positively and negatively. Students were reported to experience previous programming language interference, such as continuing old habits and making mistakes with new language concepts that worked differently and having difficulty mapping concepts of prior languages and new languages. The prior work significantly advanced our knowledge of the transition between text-based programming languages. However, it does not use a unified model or theory to account for the programming language transition results that are beginning to appear. Each paper focuses on a specific transition challenge without producing a theory that explains the transfer process. A unified cognitive model of the programming language transfer will help educators better understand the success and failure of transfer in the classroom.

The next section explains transfer theories specific to learning second and subsequent programming languages.

2.2 Theoretical Research on PL Transfer

The researchers who theorized or defined the inner mechanisms of programming language transfer have focused on experienced programmers solving programming problems in a new language. These researchers broadly adopted a problem-solving approach to study transfer to second or subsequent programming languages. These researchers described language transfer and explained why and how it happens.

Scholtz and Wiedenbeck dominated the studies on programming language transfer in the 90s [19, 20, 22, 23]. Their first study [19] was qualitative and observed five highly experienced programmers with Pascal/C knowledge learning the Icon language. These programmers reported having developed around 55 programs in their known languages. For four hours, the programmers had to solve a text-processing problem in the new Icon language. The findings were that 40% of the transfer problems dealt with syntax and semantics, but it did not take long before the programmers resolved them. These findings on experienced programmers struggling less with syntax concur with some of the above studies presented in Section 2.1.2 [11]. The most difficulty reported was with planning algorithms in a new language. These studies revealed that programmers transfer plan knowledge from previous languages, which usually fails when implemented in the new language. For example, participants used a character-at-a-time Pascal approach to reverse a string, not the built-in Icon REVERSE function. It resulted in programmers building highly inefficient programs that did not utilize the new language's features (Icon).

In their subsequent study, Scholtz and Wiedenbeck [22] were interested in exploring experienced programmers' behavior in solving problems in a new language. They developed a model

of planning in a new language derived from an empirical study of experienced Pascal programmers solving programming problems in a new Icon language; see Figure 2.1. In the model, in the early stages of learning a new programming language, experienced programmers use a top-down and depth-first approach to solve a problem and change their plans repeatedly when they reach implementation as they become familiar with a new language construct. Their model is in two parts. In the first part (labeled figure 6 on the left-hand side of Figure 2.1), the programmer starts by implementing a strategic plan and then decomposes the problem into sub-problems. The programmer then creates a tactical plan to work with each sub-problem. The programmer then formulates an implementation plan. The second part begins with the programmer implementing the plan through coding. Their knowledge of syntax and constructs of the new language is necessary at this phase. The programmer then tests the code; if it fails, they revise the implementation plan again. The frequent changing of plans results in delays in solving the problem. However, where the plans from prior languages are similar to the new language, plan transfer helps them solve the problems in the new language. It fits with the observations of plan transfer made by their later studies [18, 20]. However, in investigating novice programmers, Scholtz et al. [23] reported that novice programmers tend to use a bottom-up approach in solving problems in a new language that is different from the prior language.

Other researchers also reported the transfer of plan knowledge from previous languages to new languages as reported in the above studies. For example, Wu and Anderson [21] reported positive transfer of plans between LISP and PROLOG and between LISP and PASCAL. The findings were based on three experiments with experienced programmers: the first two on transfer between LISP and PROLOG, and the third on transfer between LISP and PASCAL. They reported that programmers use the familiar algorithms in the first language to cut down the planning in the second similar language, reducing the number of revisions needed to finalize the plan. Overall, they concluded that the three languages that shared commonalities played an advantage. Syntactic interference was reported to be minor for these programmers.

Armstrong and Hardgrave [9] developed a mind-shift learning theory based on conceptual knowledge transfer in the context of professional programmers transitioning from non-OO to OO software development; see Figure 2.2. Their theory was derived from the categories of mind-shift in cognitive sciences research [48]. The model proposes three types of transfer that engage the programmer in cognitive processing as they transition to a new language. Concepts may be perceived as novel (i.e., not familiar to the learner), changed (i.e., similar to a known concept but a different meaning in the new context), or carryover (i.e., known concept with a similar meaning in the new context).

They evaluated this model with an empirical study that focused on concept knowledge of 81 object-oriented software developers. They were given a multiple-choice test of programming concepts definitions. The findings indicate that software developers had higher knowledge

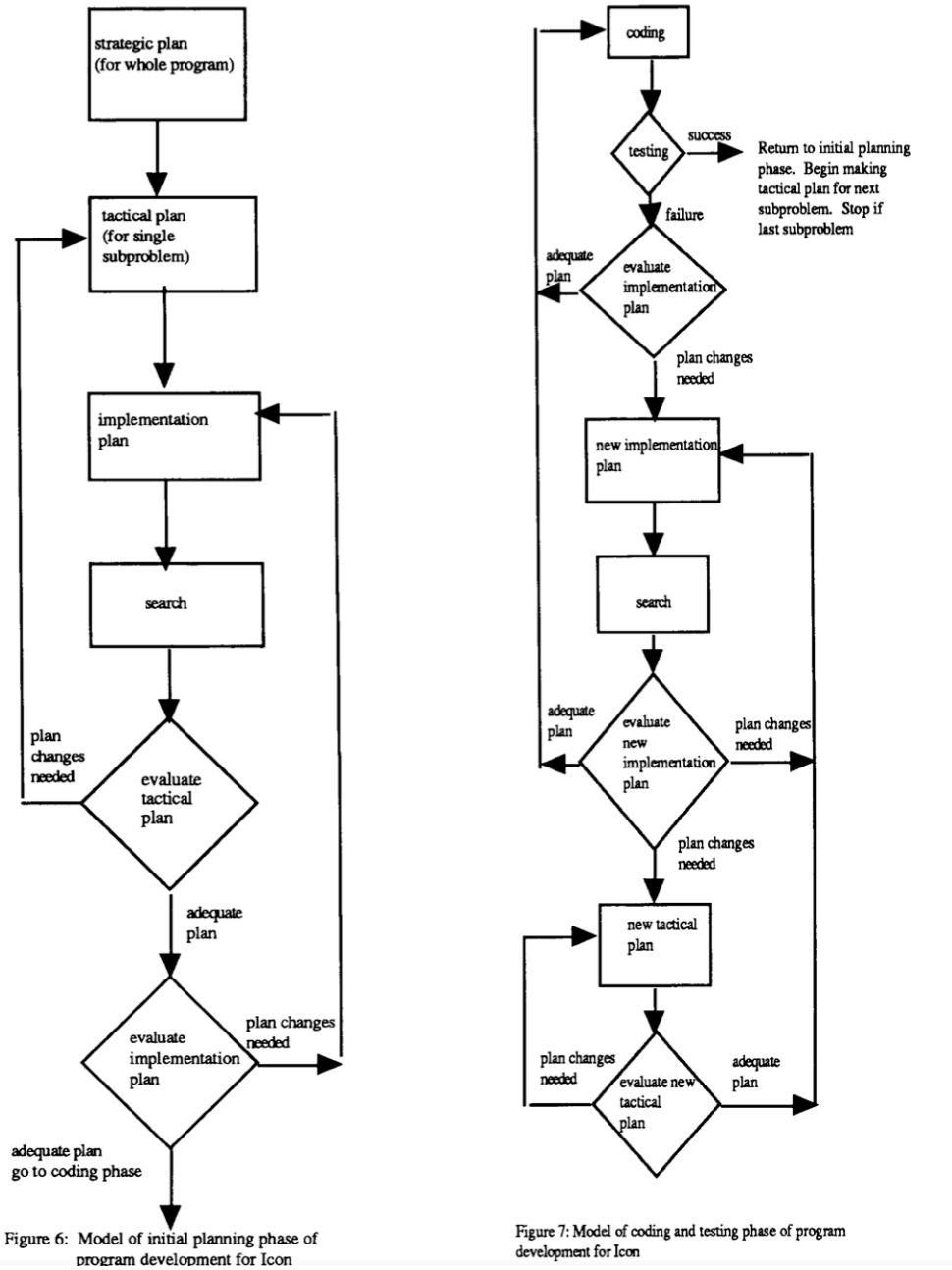


Figure 6: Model of initial planning phase of program development for Icon

Figure 7: Model of coding and testing phase of program development for Icon

Figure 2.1: Model of plan transfer for Icon by Scholtz and Wiedenbeck taken from from [22]

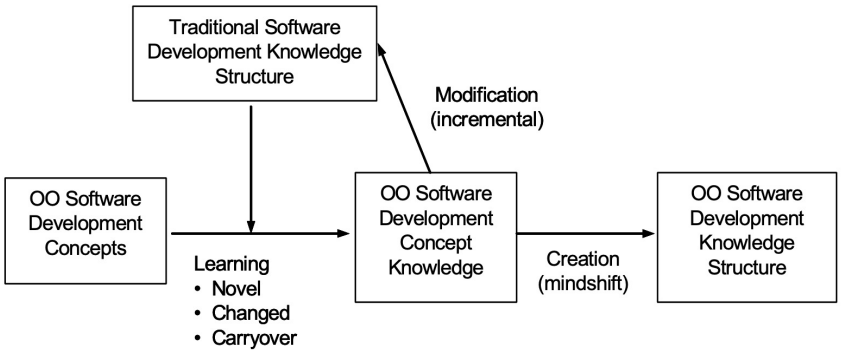


Figure 2.2: Mind-shift Learning Theory by Armstrong and Hardgrave taken from [9]

scores on the OO concepts they perceived as novel or carryover compared to those they perceived as changed.

Summary of Theoretical Research on Transfer between PLs

The previous Section 2.1.2 focused on explaining transition challenges experienced by students without focusing on theory or explaining the mechanisms involved in the learning process during transition. The just reviewed studies provide insights into the programmers' learning process when initially shifting to a new programming language/paradigm mindset. They report plan transfer from prior languages to new languages, which may be positive or negative. Additionally, Armstrong and Hardgrave [9] developed a theory that explains that experienced programmers encounter carryover, changed, and novel concepts, which affects the learning of a new programming language paradigm.

One common and underlying limitation of prior work on transfer theories is that it mainly focused on experienced programmers. Armstrong's work focused on how experienced programmers transfer conceptual knowledge at a high-level (e.g., programmers asked the meaning of the concept polymorphism). On the other hand, Scholtz's work focused on problem-solving skills transfer. Rather than making the programming language comprehension their starting point, these researchers are centrally concerned with how programmers write programs in a new language. But learning programming languages is more than problem-solving and plans; it is also about understanding the underlying programming constructs used to write programs and how they behave when executed. In addition, prior work gives little attention to the theories and cognitive processes involved when relative novices move on from their first or second language, often only with limited knowledge of plans [23, 24], compared to an experienced programmer. When learning to program, the novice programmers tend to focus more on the syntax of a programming language [3] which is tied to their understanding of programming language constructs [4]. Understanding how programming constructs work or execute in the known language is a skill that can be transferred to the new language, especially if the languages share

similar syntax and semantics. Therefore, as a starting point, this thesis aims to fill this gap by investigating how relative novices transfer their programming language knowledge of syntax and semantics to a new programming language during code comprehension.

The following section will present researchers' work that focuses on general transfer theories followed by research on code comprehension models.

2.3 Learning Transfer

Learning transfer is defined as the ability to extend what has been learned in one context to new contexts [41]. Early theoretical work on the transfer of learning emphasized the similarity between conditions of learning and conditions of transfer [42]. As one of the early researchers on transfer, Thorndike [43, 44] emphasized that new learning is facilitated by previous knowledge only to the extent that the new learning task contains elements identical to those in the previous task. Their theory suggests that training for transfer depends on the degree to which the stimuli and responses in training are identical to those in the transfer situation. Other researchers explain this type of transfer as near transfer, which means the transfer of closely related contexts and performances [45]. Perkins and Salomon [45] further explain a transfer to different learning contexts (far transfer).

Perkins and Salomon [46] propose that transfer occurs by way of two different mechanisms, namely: low-road transfer and high-road transfer. Low-road transfer occurs by automatically triggering well-learned behavior in a new context (e.g., car to truck driving skills). Low-road transfer often occurs unintentionally or implicitly and can involve personality traits, habitual behavior patterns, response tendencies, cognitive strategies and styles, expectations, and belief systems. High-road transfer occurs by explicit conscious formulation of abstraction in one situation that allows making a connection to another, e.g., a person deliberately searching for relevant knowledge already acquired. An example of such transfer would be a transfer of strategy, e.g., using a Calculus technique to solve a Physics problem. Some studies have found no transfer of students' cognitive abilities who had received programming instruction to other contexts. For example, Pea and Kurland [47] reported that students who received programming practice in LOGO had no transfer to activities such as planning or goal evaluation outside of a programming context.

Transfer can also have positive and negative effects, e.g., positive transfer is when learning in one context enhances learning in another context, and negative transfer is when learning in one context undermines learning in another context [42, 45].

Prior work has also extended the concept of learning transfer by considering analogies. Analogical transfer is the act of applying knowledge from one context to another based on shared relations [207]. Gentner [209] proposes Structure Mapping as a theory of analogical transfer which suggests that humans transfer by establishing structural alignment between two situations

and then projecting inferences. Gentner [209] proposes that achieving structural alignment is concerned with abstraction, contrast, and inference-projection learning. Abstraction learning can occur when alignment results in a common symbol system that can be used in the future. Contrast learning can occur when there are alignment differences, where differences that reside in the same role in two symbol systems are pointed out. Inference-projection involves when one member of the two relations is more complete in its structure than the other.

A recent study by Kao et al. [207] discussed analogical transfer in the context of programming languages. They propose two strategies to promote comparison and improve analogical transfer during learning based on prior work. The first one is concerned with using strategies that use comparison, such as presenting two or more problems simultaneously. The other way to compare can be by using analogical comparisons for supporting generalization to instances with a common structure and providing contrasting examples to point out where there is a similarity in surface structure but there is a difference in underlying structures. The second one is using perceptual cues as a scaffold for understanding abstract concepts. For example, common surface feature analogies can be used as a starting point for learning to support students in engaging in deeper, structural comparisons later. Kotovsky and Gentner [208] used this strategy. Their research discovered that 4-year-olds could recognize higher-order relational matches only when lower-order commonalities supported them. Another way to use perceptual cues is by presenting problems simultaneously and using surface cues to illustrate corresponding parts between them [207].

In addition to the these studies that promote transfer, Bransford and colleagues [42] explain the factors that encourage initial learning in fostering transfer. These include, among others: the degree of mastery of the initial subject, understanding versus memorization, the time it takes to learn, motivation to learn, and context.

The learning transfer research gives insights into how previous knowledge can help or hinder the understanding of new information. Additionally this research gives insights into strategies that can be used to promote transfer in learning. The next section starts by reviewing research on code comprehension to understand how programmers read code.

2.4 Program Comprehension

The researchers on program comprehension are interested in how the mind's processing mechanisms deal with reading a program written using some programming language. They are different from the previous researchers mainly concerned with the mechanisms of using a programming language to write a meaningful program. This thesis primarily focuses on relative novices' code comprehension for several reasons. Prior work has reported that understanding of the programming language (e.g., syntax, semantics/notional machine, concepts) for relative novices is still very fragile [3, 49–52]. Perkins and Martin, define fragile knowledge as "knowledge that is

partial, hard to access, and often misused" [156]. When relative novices start writing programs or engaging in problem-solving activities too early, it may delay their development of viable mental models of program execution. Furthermore, a recent study by Salac et al. [53] reported that writing programs is not necessarily a perfect reflection of the ability to understand code in novice programmers. In this section, program comprehension models are reviewed.

Early work on program comprehension was by Pennington [24]; they propose that a programmer has *text-structure* knowledge and *plan* knowledge. *Text-structure* knowledge in program comprehension models is knowledge of the grammar and syntax of the program [55]. According to Pennington, *text-structure* knowledge is the program text/syntax which consists of control-flow constructs such as sequence, iteration, and conditionals. They propose that the programmer starts reading the syntax, which provides surface clues during comprehension. The programmer then chunks statements into groups which then combine into higher-order groupings. The *plan* knowledge is concerned with the programmers' understanding of patterns of program instructions to accomplish functions/goals such as summing, hashing, and counting. Comprehension starts with the programmer recognizing and activating a familiar program plan by matching the input text to existing *plan* knowledge. To understand the role of programming knowledge in program comprehension, they investigated eighty professional programmers comprehending short computer programs and reported that syntax knowledge forms the basis of programmers' mental representation in understanding a program as compared to *plan* knowledge.

Shneiderman and Mayer [3] propose that when a programmer is reading code, they use their already existing *syntactic* knowledge to construct an internal *semantic structure* to represent the program. In their model, the programmer has a multi-leveled knowledge cognitive structure. This knowledge consists of *structured semantic* knowledge and *syntax* knowledge. *Syntactic* knowledge is concerned with the programming language, while *semantic* knowledge involves programming concepts with one or many syntactic representations. The *structured semantic* knowledge can be at multiple levels. For example, a lower level entails how a construct operates, while a higher level entails how a program functions. During reading, the programmer uses a chunking process to assimilate the code; that is, the programmer starts with recognizing the function of a group of statements and then combines these chunks to form larger chunks until the entire program is comprehended. Their studies that formed the basis of the model compared the comprehension of FORTRAN arithmetic and logical IF-statements with two groups. The groups involved 24 first-term programming students (novices) and 24 advanced programmers (experts). The results showed that novice programmers struggled with the greater syntactic complexity of the arithmetic program as compared to experts.

Lastly, Schulte [54] proposed an educational model of program comprehension, which, unlike the previous models, focuses on program understanding by novices, aimed at supporting research and practice of teaching and learning to program. The Block Model is organized as

a table consisting of three columns and four rows. The rows consist of four levels starting in sequential order from atoms, blocks, relations, and the macrostructure. The columns have two main dimensions; *structure* and *function*. The *structure* dimension consists of the *text surface* (syntax) and the *program execution*. At the same time, the *function* dimension is concerned with the intended function/goals of the program. Each cell represents the level of the programmer's comprehension of a program. During comprehension, the programmer starts with reading the syntax. The process moves from bottom to top in the table. The process starts with reading from words (Atoms) to blocks, then moves to inferences about the relations between blocks to recognizing the program's overall structure and purpose. The comprehension process is presented as bottom-up yet chaotic and flexible. Understanding of the program is influenced by the programmer's prior knowledge of the program text/syntax.

These program comprehension models explain the cognitive processes involved in the programmer's reading and understanding of a program [3, 24, 54, 55]. These models have common elements of the cognition: Knowledge structures and Assimilation processes. These models view the programmer as having two categories of static programming knowledge which according to Pennington are the *Text-structure knowledge* and *Plan knowledge*. These categories are presented with different names in these models. For example, Shneiderman [56] refers to the *text-structure* knowledge as the *syntactic structure* while Schulte [54] describes this knowledge as the *structure*. Shneiderman refers to the *plan* knowledge as the *semantic* (higher level) knowledge while Schulte calls it as the *function* knowledge.

The process of program comprehension by the reviewed comprehension models mostly draws from the natural languages text comprehension model by Kintsch [179] which describes comprehension involving chunking the text into segments that correspond to schema categories so that labels for segments will constitute the macrostructure for the text. The program comprehension models describe program comprehension as starting with identifying the building blocks of program units in the surface structure of the program and deriving their local purposes. These units then act as items that combine into higher-order program units, with higher-level functions attached to units at this level. This process continues until the highest level is a single unit with an identifiable function.

These models explain how programmers understand program code. However, they refer primarily to comprehension in the case of one language, with little attention given to how programmers understand a program in a new language. These models also emphasize the importance of prior knowledge in comprehending syntax. Research on program comprehension models also shows that novices tend to struggle more with programming language syntax, reflecting how limited their prior knowledge is to programming language concepts and plans.

In this thesis, the interest is on comprehension mechanisms in the case of learning a second language, given that the programmer has already acquired syntax, semantics, and plan knowledge in their first language. Unlike prior work, which focused on the programmer's plan knowl-

edge when writing code and how it transfers to a new language, the focus of this thesis is on the programmer's syntax and the semantic knowledge when reading code and how it transfers to a new language. The code comprehension models [24, 54, 57, 180] have drawn from natural languages text comprehension models. Programming languages and natural languages are similar, but humans design programming languages to communicate a set of instructions to the machine. The similarities are in that they both are a means of communication and have syntax and semantics [58]. Syntax is a set of rules that guide language users on how to combine words, characters, and phrases, while semantics refers to the meaning of words. Section 2.5.2 elaborates more on the similarities between natural languages and programming languages. Because of the similarities, this thesis looks at natural language transfer to understand programming language transfer. Therefore, making it a starting point.

2.5 Transfer in Natural Languages

The research in learning a second natural language (L2) strongly emphasizes the role of the first language (L1) in learning other new languages [25, 28, 59–61]. This research has reported that language learners initially learn their second language meaning through meaning already established in their first languages via lexical/word associations between the two languages, and is now discussed.

Jiang proposed a validated psycholinguistic model of vocabulary acquisition in second language learning that focuses on how the L2 mental lexicon evolves for a second language learner [25, 26]. For every language a person knows, they have a mental lexicon that captures the word structure, pronunciation, syntax, and meaning for each word they have learned [62]. According to Levelt, a person's mental lexicon is a repository of declarative knowledge about the words of their language [63]. That is, it deals with how the person stores, processes, and retrieves the words in their language. In the model, a lexical entry in the lexicon has two components, taken from Levelt's work on speech production [63]. The first component is the *lemma*, which deals with the syntax and semantics of a word. The *lexeme* is the second component concerned with the morphology and phonological-orthographic aspects of a word [63]; see Figure 2.3. These components are explained as follows:

- **The lexeme:**
  - **Phonological-Orthographic**: This component means that the language speaker can pronounce the word [64]. It deals with the lexical entry's composition in terms of phonological segments (its accent structure and pronunciation) [63]. Orthography is concerned with the spelling of the word [25]. The pronunciation is represented by different combinations of letters when written down. For example, in the English

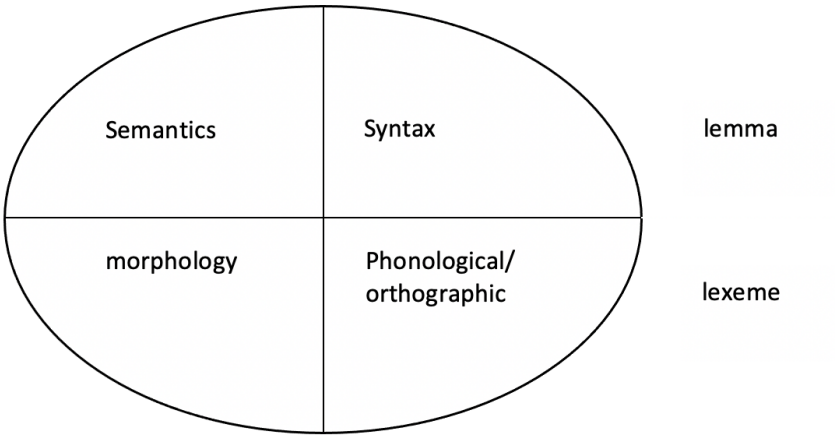


Figure 2.3: The internal structure of the lexical entry adapted from Levelt, 1989 [63])

language, the lexical entry of the word `love` is spelled as `love` and pronounced as `Luv`.

- **Morphology**: Morphology means that the language speaker knows how to add and subtract parts of the word to make new words [64]. The component deals with words, their internal structure, and how they are formed [65]. Some examples of word forms for the lexical entry `love` may include third-person singular present tense `loves` and past-tense `loved`.

• **The Lemma:**

- **Syntax**: This component means the language speaker can use the word in a sentence. Syntax involves the sentence structure or the order of words in a sentence. It includes a set of syntactic properties , which firstly includes the lexical entry (e.g the word `love` which is a **verb** (*doing word*)), the syntactic arguments it can take (**subject** (*person or thing doing the action*) and an **object** (*what is acted on*), and other properties [63]. For example, in the English language, the lexical entry for the verb `love`, may require a subject and an object in the following grammatical pattern or order where the argument **subject** comes first, the **verb** second, and the **object** third. An example of the sentence is `The girl loves food`. In other languages, this sentencing order may be different, for example, the Arabic order would be VSO (verb-subject-object).
- **Semantics**: Semantics is the meaning that is taken from words, sentences and phrases. For example, meaning can be derived from the sentence "`The girl loves food`" because it is syntactically correct in the English language, however, if it was structured as "`food girl the loves`" it would have no meaning.

According to Jiang, the two lexical entry components are highly integrated such that acti-

vation of one component results in automatic simultaneous activation of the other components. A learner can simultaneously extract the semantic, syntactic, phonological, and orthographic information upon recognizing a word. As has also been reported in word recognition experiments where phonological information is automatically activated in visual word recognition tasks [66].

Jiang proposes that second language adult learners already have a well-established mental lexicon in their L1, which they learned as children. It means that second language learners learn L2 words in the presence of extensive existing conceptual and semantic information in the L1 lexicon. When a new L2 word has little conceptual or semantic development during second language learning, the strong existing L1 lexicon becomes actively involved in L2 learning. Because of typical second language learning methods, learners rely on this already existing information in understanding L2. Jiang's model of vocabulary acquisition is based on a notion of *semantic transfer* and consists of three stages, described below in Figure 2.4:

1. Stage 1 (Lexical Association Stage): A lexical association between Language 2 (L2) word and its Language 1 (L1) translation happens when an L2 lexical entry (e.g., a word) is introduced to the learner. The L2 word has a pointer (reference) that directs attention to the L1 translation equivalent (1a.). The learner understands the L2 word meaning based on the existing semantic information in their L1 lexicon. In recognizing the L2 word, the learner activates the L1 translation and uses its semantic, syntactic, and morphological information for comprehension (1b. and 1c.). At this point, there is a weak, or no, direct link between the L2 word and the concept. The L1 syntactic and semantic meaning is transferred from L1 to L2 (1d.).

2. Stage 2 (Lemma Mediation Stage): The transferred semantic and syntactic information links the L2 word and the concepts; hence the learner does not need to rely much on the L1 translation (2a. and 2b.). At this point, the L2 words are linked to the conceptual representation both directly through the L1 lemmas (syntax and semantics) and lexical association with their L1 translation. When the learner keeps getting exposed to contextualized L2 words, it helps them develop L2-specific meanings/semantics in the knowledge structure such that the L2 lemma contains both L1 and L2 specifications.

3. Stage 3 (Full Integration Stage): Finally, at this point, there are strong links between L2 words and concepts such that L2-specific information dominates the knowledge structure with a very weak dependency on L1.

In the model, when the second language learner is introduced to a new L2 word, their attention is focused on the formal features of the word, i.e., spelling and pronunciation. Little semantic, syntactic, and morphological information is created and established at this point. The learner may be introduced to the L2 word by a pointer that draws their attention to the translation of that word in L1. So basically, the pointer connects L1 and L2 words. For example, when an English (L1) native speaker is learning Setswana (L2), they may get introduced to the word

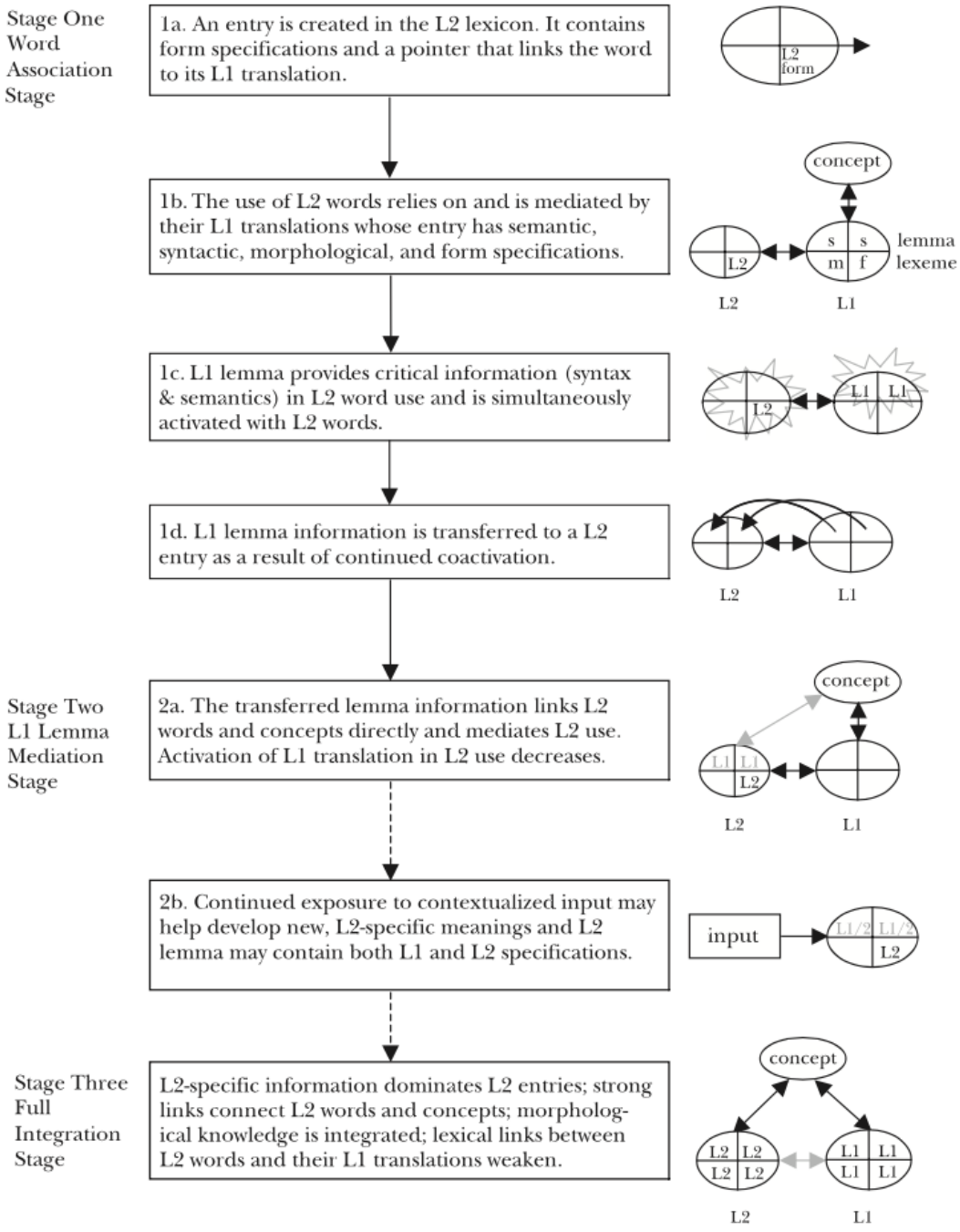


Figure 2.4: Jiang's model of second language acquisition drawn from [26]

`siana` and a pointer (it could be a reference to L2 by a teacher/textbook) that draws their attention to its English translation `run`. The Setswana learner then activates the semantic, syntactic, and morphological information from their mental lexicon about the English word `run` to assist them in comprehension of the Setswana word. They know the word in English (L1) and know `run` can be used as a verb to mean moving at a speed faster than a walk. They know how to use it in an L1 sentence, e.g., `The boy ran`. At this point, the Setswana learner understands `siana` through the mental lexicon of their L1 word `run`.

Continued exposure to L2 words and their meanings will eventually form strong links between L2 words and their concepts, weakening the reliance on L1. However, this is not always the case that all learners can reach this final stage. This could be because once the lexicon space is occupied by the L1 lemma information (syntax and semantics), it may be a challenge for the L2 lemma information to occupy the lexicon space, or else they do not spend enough time using L2 for this switch to occur. For example, while some words in L1 and L2 have an exact match on meaning, they represent exactly the same concept. Other L1/L2 pairs are close in meaning but not identical. In such a case, lexical association and L1 lemma mediation can lead to lexical errors when an L2 word and its L1 translation do not match in semantic representation [25]. For example, a native Arabic learner of a second language (English) using the English words `long` and `tall` interchangeably because both their translation in Arabic is `tawil` [67]. That is, the Arabic learner might learn that "long" and "tall" both map to the Arabic word "tawil." They will then happily say "I am a long person" in English because of the dual concepts for the word in Arabic, resolved in use by context. There are multiple such dualities in English; of course, when one says "I hammered the nail into the wood," they know by context which concept/meaning of nail to use-the metal one, not the one at the end of the finger.

Jiang's model provides useful insights into the process of second language learning which may also be applicable to programming languages. The contributions that are relevant from this model are as follows:

1. **The conceptual and lexicon mental representations:** There is a clear separation between the conceptual knowledge about the word and the lexical (syntax and semantics) knowledge in L1 and L2. Each language has its own lexicon yet they both share the same conceptual level.

2. **Reliance on L1:** L2 learners develop concepts/semantics associated with words in L1. That is, L2 learners heavily rely on their L1 lexical knowledge in order to comprehend the second language words.

3. **Semantic transfer:** Semantic knowledge about an L2 word is transferred from the associated L1 lexicon. The concept onto which the L2 word is mapped is an L1 concept.

The limitation with Jiang's model is that it is not specific on how a lexical entry forms connections/lexical associations between L1 and L2 in Stage one. For example, is it based

on the similarity of words in L1 and L2? The model also does not represent some aspects of transfer in the mental lexicon, e.g., negative and positive transfer/near and far transfer from cognitive science theories as discussed in the previous Section 2.2. In other words, the model does not cater to the different storage patterns of possible connections within the lexicon of a language user. For instance, if the L1 and L2 words look similar like `table` and mean the same thing in both French and English, how are they represented in the Mental Lexicon. It seems that the L1 and L2 translation equivalents would share very close conceptual, semantic, and lexical forms. Would each language then have a separate mental lexicon connected at concept level like in Stage three of Jiang's model or, are they intertwined?

In addition, the model has been validated by semantic judgment tasks measured by reaction time. These tasks are where participants are requested to judge whether or not the L1/L2 word pairs are related in meaning. These experiments have not paid much attention to comprehension activities, especially the language syntax. Therefore it becomes a challenge to account for how transfer occurs at the syntax level. Language comprehension demands much more than just identifying individual words [68]. It involves language learners knowing how a combination of words and symbols operate together to provide meaning.

Ringbom's research on cross-linguistic similarities helps explain natural language transfer through the lens of language comprehension. Unlike Jiang's model of vocabulary acquisition that highlights the developmental sequences of the L2 lexicon, Ringbom's research pays attention to the comprehension processes/mechanisms relating to performance during L2 learning. According to Ringbom, transfer in comprehension is influenced by cross-linguistic similarities and the perceptions between L2 input and existing L1-based lexicon knowledge [28]. Cross-linguistic similarities can be defined as the lexical (phonological, morphological, syntactic, semantic) similarities amongst languages [69]. Jiang's model does not explain how cross-linguistic similarity influences the arrangement of the mental lexicon in the mind of a language learner. However, in the early stages of L2 learning, language form (phonology, orthographic, morphology, and syntax) plays a role in the arrangement of the mental lexicon [28].

In psycho-linguistics, words that are similar and share translation equivalents are called *cognates*. These are words with similar forms and meanings in two languages. Full scale cross-linguistic similarity of both form and meaning is not common and it occurs only in closely related languages that are mutually comprehensible like Norwegian and Swedish [202] which both stem from Germanic languages [203]. In second language learning, a language that shows similarities with the already known language will be easier to learn in comparison to different languages. Ringbom proposes that comprehension of L2 can start with learners perceiving cross-linguistic similarities with L1 followed by the assumption of the associated semantic or functional similarity. While most perceived similarities will facilitate learning, there are also instances where similarity can lead to errors [28, 70]. Ringbom describes three similarity relations in second language learning:

1. Similarity Relation: This means that a form or pattern in the L2 is perceived as functionally similar to a form or pattern in the L1, e.g., cognate words in two languages that share form and meaning like `mine` in *English* and `mein` in *German*. This results in positive transfer.

2. Contrast Relation: This means that a form or pattern in the L2 is perceived differing from L1 form, though there is an underlying similarity between them. For example, the French word `copier` with a suffix ending of `er` which has the same meaning of the English word `to copy` with a suffix ending of `y`. This may mean the native English speaker may recognise the French word is related to the concept of copy in English but may have challenges using the correct suffix of the French word. This can lead to both positive and negative transfer. The similarity in these words may be because the French word has been adopted into English, due to the long history close geographic proximity of speakers.

3. Zero Relation: This means that a form or pattern in L2 appears to have little or no perceptible relation to L1 or any other language the learner knows. For example, this occurs when learning a language which looks different from L1. Native English speakers may have trouble learning languages like Zulu. E.g. `Love` Zulu translation is `Uthando`.

Ringbom's research provides insights into the mechanisms involved in second language learning and adds to what is already known about L2 lexicon development in Jiang's theory of second language acquisition. More importantly, Ringbom's descriptions of similarity relations are related to theories of learning transfer in cognitive sciences described in Section 2.2 about positive, negative, and no transfer of learning. The contributions that stand out from this work are as follows:

1. **The role of lexical similarity in L1 and L2:** Similarity plays a crucial role in making lexical associations between L1 and L2. This is in line with Thordike's theory of identical elements which describes how transfer depends on the level of similarity between what is being learnt and what is known [44].

2. **The three different types of cross-linguistic similarities:** These are similarity, contrast and zero relation between L1 and L2 and their consequences in comprehending L2.

2.5.1 Second Language Learning and Cognitive Approaches

In the previous sections, Jiang's theory of second language acquisition was presented and is concerned with the representation and the development of the L2 mental lexicon. Ringbom's similarity relations, on the other hand, is concerned with how second language learners access the L1 mental lexicon and the mechanisms they employ during second language learning.

Usually, the study of language is the area of linguistics and the study of mental processes is the area of psychology [64]. Discussing both disciplines can help understand how the mind

works and learn new information, which will also be informative in understanding second language learning theories. In this section, therefore, some of the work of cognitive theorists is reviewed.

The first cognitive model also adopted in second language learning research is Anderson's ACT* model. The ACT* model is related to the processing and the development of cognitive skills over time, just like Jiang's theory is concerned with the development of linguistic knowledge. Anderson proposes two stages in the development of cognitive skills, a Declarative stage concerned with the facts/concepts about the skill domain and a Procedural stage in which the domain knowledge is incorporated into procedures for performing the skill [71–73]. Both Declarative and Procedural knowledge is in long-term memory. Procedural knowledge develops from Declarative knowledge and requires awareness in the early stages. For example, when learning to ride a bike, the learner's mind is initially engaged in a conscious effort to understand the facts about the riding skill (pedalling, steering, balancing, and coordination). At this point, it does not mean they know how to ride a bike well. With practice and repetition, the learner learns how to perform the pedalling and steering movements while keeping balance and coordination. Eventually, the skill is activated unconsciously, without them having to think about it. According to Anderson, this transition from the Declarative phase to the Procedural phase is in 3 stages [64, 72, 73]:

1. The cognitive stage: The leaner learns the description of the procedure
2. The associative stage: The learner works out a method to perform the skill
3. The autonomous stage: The skill becomes more rapid and automatic

In applying the model to second language learning, the Declarative stage would consist of the learner's knowledge about the linguistic form (e.g., phonology, morphology, and syntax). Procedural knowledge is about being able to use a particular linguistic form to understand or produce language implicitly. For example, at an initial stage of learning a new L2 English word, `'Party'`, Arab native speakers will have to be conscious about the way that the sounds are pronounced. For example, Arabic does not have words with sound `"p"` and they may want to use `/b/` sound instead. The learners will need to be conscious of this and may verbalize the task. Eventually, however, the sound produced will become automatic and will not need to be thought about [64].

Another cognitive model that has been applied to the study of memory and learning is the Connectionism [182, 183]. The Connectionism model uses the analogy of a mind as being like a computer having neural networks that have links between the nodes [72]. In relation to language comprehension on the lexical level, the model proposes that there are connections between words within chunks with the assumption that concepts and the associative network are common to both languages (L1 and L2) [64]. For example, the concept of `animal` is assumed

to exist in both English and Setswana language. The idea that concepts are shared between languages is also adopted by Jiang and Levelt's models [25,63] discussed in the previous section. The Connectionism model describes that language input is initially processed at a phoneme level and connections are formed between the nodes at this level. These continue to be connected at the phrase and syntax levels. Through continued exposure and repetition of the linguistic patterns, these established connections between nodes are strengthened [64, 73]. The second language learner may be better than the first language learner because they already have an established lexicon from their first language experience hence might be able to process the L2 input and recognise the recurring patterns at initial stages.

The last theory is of identical elements by Thorndike that emphasizes that when a new learning task contains elements that are identical to those in the previous task, learning becomes easier [43, 44]. Their theory suggests that learning for transfer depends on the degree to which the stimuli and responses in the learning task are identical. For example, in the second language context, learners of Afrikaans whose first language is Dutch have an advantage with their Dutch background compared to students without a Dutch background. This is because Dutch and Afrikaans share a lot of identical linguistic elements, which facilitates learning. This theory is related to Ringbom's similarity relations in cross-linguistic similarities, which proposes that similarities in L1 and L2 can facilitate learning when L1 and L2 share similar functionality.

In summary, cognitive processes related to second language learning and comprehension were presented. It is evident that the natural language theories presented in this thesis are drawn from cognitive psychology. This involves cognitive skill development (related to Jiang's theory) and processing mechanisms (related to Ringbom's similarity relations) in second language acquisition. As shown in natural-language theories, the theories of connectionism and Piaget's identical elements emphasize the importance of existing language knowledge in learning new languages. While the ACT* model demonstrates that, with repetition and practice, the L2 knowledge may be strengthened and atomized.

Going back to the natural language theories specific to how people learn linguistic knowledge in second languages, it seems clear that when learners learn their second language, the semantic transfer will / is likely to occur. Ringbom's research suggests that cross-linguistic similarities can play a big part in the processing mechanisms of learners making lexical links between L1 and L2 during code comprehension in Jiang's model. In addition, Ringbom has explained the three types of similarities between L1 and L2 that learners encounter. Of course, these natural language theories cannot be adopted as they are for programming language learning. To contextualize these theories in programming languages, it is essential to know why this thesis wishes to draw from natural language theories in the first place when exploring programming language transfer. This is explained in the next section.

2.5.2 The context of Programming Languages

In natural languages, there is a theory of universal grammar by Noam Chomsky that proposes that all human beings under normal conditions develop language with certain properties innately. In universal grammar, the language consists of a lexicon and a set of rules [72]. A sentence rule in English can be `'Sentence → Subject Verb Object'` which can be applied correctly as `'Dog eats food'`. The common notation used by computer scientists to represent grammar is called Backus-Naur form (BNF), and it describes the syntax (rules) of programming languages used in computing. An assignment rule can be `<assignment statement> ::= <variable> = <expression>` which can be applied in Python as `'number=2'`. Unlike natural languages, programming languages are designed to be used by programmers to write meaningful programs with instructions that can later be interpreted into the machine language. Both languages are used to communicate among people, although programmers are usually supposed to read, understand, and modify existing programs written by other programmers.

Like natural languages, many programming languages share common structural and lexical similarities because earlier programming language concepts and grammar have influenced the design of each new programming language. Programming languages have a lexicon. The *lexeme* component is a string of characters that is the lowest syntactic unit in the programming language, e.g., the tokens, keywords, operators, and identifiers. However, unlike natural languages, programming languages do not necessarily currently focus on phonology because they are not spoken. The medium for communication in programming languages is mainly text. Lastly, morphology may not be very relevant to programming languages as their grammar is precise, fixed with formal meaning defined according to the grammar specifications, and does not change depending on context. Based on these reasons, the *lexeme* component defined in Jiang's theory is not so crucial for this research now.

Like natural languages in the *lemma* component, syntax and semantics are applicable to programming languages. The syntax of a programming language describes how to combine strings of characters to produce a valid program. The semantics of a programming language describes what syntactically valid programs mean. Hence, this thesis will focus on the syntax and semantics of the mental lexicon to explore the transfer phenomenon.

In summary, there are a lot of similarities between natural languages and programming languages [184] and therefore programming language research can draw on theories of natural language transfer to explore the transfer in programming languages. However, given that there are still many differences between the two, natural language theories cannot be applicable as they are to programming languages. For example, if people learn their native language (L1) innately as children, how would transfer occur in programming languages, given that people learn their first programming language at later stages in their lives? Could transfer be depending on the PL1 prior knowledge or it happens bi-directionally due to fragile PL1 lexicon knowledge?

This thesis aims to explore these questions as a starting point.

The overall goal of this thesis is to understand how relative novice programmers transfer knowledge from their first PL to the second PL during code comprehension. Given the semantic transfer principles in natural languages, it becomes compelling to hypothesize that semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer in comprehending new programming languages. In order to test this hypothesis in programming languages, an exploratory study is reviewed in the next chapter to gain in-depth knowledge of novice language transfer and how it relates to the hypothesis.

The second part of the thesis statement hypothesizes that the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language. Based on this part of the hypothesis, the literature survey is extended by investigating existing pedagogical approaches in both programming languages and natural languages that may inform the interventions on programming language transfer.

2.6 Pedagogical Approaches in PL Transfer

Here research on pedagogical approaches to address the transition issues is reviewed. These approaches mainly design tools that have dual interfaces representing visual and text-based languages to help learners form analogies at the text surface (syntax) as they learn new languages [8, 10, 74, 178]. For example, Kölling et al. [178] designed a frame-based programming environment that combines features of block-based and text-based programming to help students transition in a sequence of programming systems from blocks to text. The frame-based environment is implemented in Stride, which is integrated into the Greenfoot development environment. They propose that the tool will help novice programmers beginning from the age of 14 with the difficulties of transitioning to text-based languages since it combines many of the beneficial aspects of block-based and text-based systems into a single interface. The benefits include reducing the need to memorize and type syntax and tackling issues with poor error messages which students face when transitioning to text-based languages.

Homer and Noble [74] investigated students using Tiled Grace programming environment with textual and visual/block representation of code. The code is visualized as tiles that are manipulated by drag-and-drop. A tile represents a single syntactic unit in the program, such as variable assignment. One of the aims of Tile Grace is to help students make connections of analogous concepts as they transition from one language to the other by demonstrating the exact parallel between the two languages. They investigated 33 university students as they engaged with Tiled Grace. They reported that 76% of the students enjoyed using the programming environment and reported that students found the mapping between tile and text helpful and appreciated the ability to switch between the two.

Weintrop and Wilensky [8] on the other hand used a quasi-experimental design to investigate 90 high school students writing code in a hybrid interface of Pencil.cc over the first five weeks of an Introduction to Programming course. The hybrid interface of Pencil.cc gives students the ability to drag and drop commands into a program. For example, when a student drags a block onto the text canvas, the block turns into the textual equivalent and is inserted into the program in a syntactically valid way. They reported that the hybrid interface has the ease of composition of the block-based modality, which made it easy for students to quickly add commands to the program by dragging blocks into the text area to verify syntax. As students did that, they could see the syntax errors because they were not restricted on how and where commands could be added. As a result, the hybrid environment allowed students to experience both the blocks and text environments fully.

Dann et al. [10] studied learners transitioning from Alice 3 to Java. Alice 3 was designed to enable learners to switch to the equivalent Java code on the same screen. This Alice 3 to Java approach is designed to enable students and instructors to transfer concepts learned in the context of Alice animations to programming in Java. They developed a pedagogy that integrates the Alice 3 problem-solving strategy with the bridging and hugging teaching technique by Perkins. The bridging approach aims to bridge students' existing knowledge to a new context [45] while the hugging technique creates learning experiences that are similar to the future learning situations [45]. The experimental group used the Alice 3 hybrid approach before transitioning to Java, while the control group did not. They collected historical data from the Java final score exams for both groups for comparison purposes. They concluded that the experimental group's overall scores improved compared to the Control group because of the explicit transfer techniques, supported by Alice 3's ability to transfer code directly to Java.

Tabet et al. [205] also conducted a study that used the bridging and hugging teaching technique by Perkins [45]. The study investigated students transitioning from Alice (learned in grade 7) to Python they learned in grade 8. They compared students taking Python in eighth grade who had minimal Alice background (group A) with those who had extensive background knowledge in Alice programming (group B). In grade 7, students were taught Alice using the hugging technique, while in grade 8, they were taught Python using the bridging technique. It meant that students were taught by comparing programming concepts in both Alice and Python. The quantitative results revealed that in the final Python assessment, group B outperformed group A. Furthermore, they discovered that comparing similar examples of concepts in both Alice and Python helped students transfer knowledge of concepts better. Another recent study confirmed that the bridging and hugging transfer techniques were beneficial for students when they investigated transfer from the block-based programming language, MakeCode for micro:bit to text-based Python programming language on six grade students [206].

The just reviewed pedagogical approaches give great insights into the effectiveness of using transfer interventions such as bridging and hugging [45] when teaching students a second

programming language.

Most of the reviewed studies also show promising evidence of the effectiveness of using hybrid programming environments in helping students transfer from blocks to text. However, most of these approaches, except for [205], focus on using designed programming environments that assist learners in transitioning and do not explain the teacher's role in helping students transition. Careful focus on pedagogy and design is required to help students transition across notations [5].

2.7 Summary of Literature Review

This chapter reviewed prior work that underpins and informs the body of work presented in this thesis. The four main discussion points from prior work were transition and transfer in programming languages context, program comprehension models, transfer in natural language contexts, and, lastly, the pedagogical approaches in programming languages.

Firstly, this chapter reviewed prior work on students transitioning between programming languages. It was reported that relative novice programmers experience challenges when transitioning to new programming languages. However, this research did not use a unified theory to account for the programming language transition challenges.

The researchers who theorized programming language transfer have mainly paid attention to experienced programmers writing programs in a new language. This research reported that experienced programmers experience transfer of plan knowledge when writing programs in a new language which may or may not be beneficial for them. However, this work gives little attention to how novice programmers transfer knowledge to second and subsequent programming languages, often only with limited knowledge of plans. Relative novice programmers focus more on the syntax of a programming language which is tied to their comprehension of programming language concepts. The code comprehension models reviewed in this chapter explain the cognitive processes involved in the programmer's reading and understanding of a program. However, they do not cater to comprehending code in a new programming language.

This thesis, therefore, reviews and draws from second language acquisition models to understand programming language transfer. The second language acquisition models discussed in this chapter revealed that L2 learners heavily rely on L1 lexical knowledge to comprehend L2 words. It means they experience semantic knowledge transfer from L1 to L2. Furthermore, this research reported on the role of lexical similarity in learners making associations between L1 and L2.

The last part of the literature reviewed the pedagogical approaches in programming language transfer. These interventions show promising results of using hybrid programming environments to help learners transition between blocks and text-based languages. However, they do not explain the role of the teacher in assisting students in transitioning.

The prior work has given significant contributions and insights into the language transfer research and has informed what still needs to be addressed in this area. Given the identified gaps in prior work, the goal of this thesis is to investigate how relative novice programmers transfer knowledge to their second and subsequent programming languages during code comprehension. Drawing from the literature, the major claims of this thesis are as follows: **Semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages; the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.** To support this thesis statement, the following research questions are asked:

- RQ1: How are principles of semantic transfer in natural languages applicable to patterns of transfer in the context of relative novices PL transferring from first to subsequent PLs?
- RQ2: How do relative-novices transfer their programming language conceptual knowledge to a new programming language during code comprehension?
- RQ3: How do teachers experience PL transfer for relative novices in the classroom?
- RQ4: How can transfer teaching interventions based on our understanding of semantic transfer improve second PL learning?

In order to answer these research questions, this thesis explores novice programmers' language transfer, teachers' transfer practices, design a model of programming language transfer, validate the model, and design and test a pedagogy for language transfer. The following chapter presents the methodological choices used to answer the research questions for this study.

Chapter 3

Methodology

This chapter describes the mixed methods research methodology used to investigate relative novices' semantic transfer in programming languages and the role of implementing transfer interventions in fostering conceptual transfer in programming languages. A mixed-method research methodology was used to explore the semantic transfer phenomenon, build and validate a model of programming language transfer and finally develop and evaluate a programming language transfer pedagogy. This chapter discusses the methodology and the research process of this thesis. The chapter starts by explaining the research design and the rationale and assumptions underpinning it. Next, the issues of validity and reliability are discussed, followed by the ethical considerations of the research. It should be noted that this thesis is organized in a sequence of different studies developmentally building on to each other. Therefore, they each have their unique data collection and data analysis methods which will be not be discussed here but instead in the separate study sections.

3.1 The Research Design

To answer the research questions, the mixed methods design, which is a combination of qualitative and quantitative research in the same research [75] is used. This thesis used the mixed methods design to capitalize on the strengths of both qualitative and quantitative approaches and for its ability to address the research questions in-depth and breadth [76].

Creswell and Clark [75] recommend three core mixed methods designs that provide a framework for research studies which are the convergent design, the explanatory sequential design, and the exploratory sequential design. The *Convergent design* is when the researcher uses results from both quantitative and qualitative data analysis to understand the phenomenon and validate one set of findings with the other. The *Exploratory Sequential Design* is presented in 3 stages that start with collection and analysis of qualitative data followed by the development phase of interpreting the qualitative data into a theoretical model and finally testing the model quantitatively. The *Explanatory Sequential Design* starts with the collection and analysis of quantitative

data followed by the qualitative data collection to explain the quantitative results in depth.

For this thesis, two core mixed methods, the Exploratory sequential design, and the Convergent design, are combined. This thesis starts by using an *Exploratory Sequential Design* to answer RQ1 and RQ2. This design was chosen because there is no guiding framework/model of programming language transfer for relative novice programmers during code comprehension; hence, the research aim was to develop a substantively relevant model for novice programmers. The research starts with an exploratory qualitative study to understand how novice programmers transfer knowledge between programming languages. A Model of Programming Language Transfer (MPLT) is developed based on the exploratory qualitative study results, and quantitative studies to validate the model hypothesis are conducted. In the third phase, RQ3 and RQ4 are answered by first exploring teachers' experiences with transfer interventions and then developing a transfer intervention/pedagogy based on what is learned from the model validation results. Then finally, the intervention is examined using both quantitative and qualitative results. This thesis made use of the exploratory (phase 1 to phase 2) and convergent (phase 3) core designs as elaborated in Figure 3.1.

3.2 Issues of Validity and Reliability

Reliability and validity are concerned with evaluating the quality of the research. It is done by evaluating the method, technique, or test measures used and the consistency and accuracy of the measure. The discussion about the validity and reliability of this research will be discussed in two parts of quantitative studies and qualitative studies.

3.2.1 Quantitative Studies

Each experiment's sample size was considered to ensure the validity and reliability of the results. Delice [79] proposes that sample sizes of more than 30 are suitable for quantitative methods. The participants in this thesis met the conditions of sample size 30 or more. Furthermore, the students who participated in the study met all the transfer study requirements. The requirements of the students who participated were restricted to students who are transitioning to a second programming language and had an average of 1 year of programming experience.

The other issues concerning validity and reliability in research are the instruments used to collect the data. The instruments were reviewed by other researchers, knowledgeable teachers, and PL experts to ensure content validity and deemed reasonable. Lastly, appropriate statistical measures and techniques to answer the research questions were chosen by ensuring the normality of data before choosing the suitable statistical method.

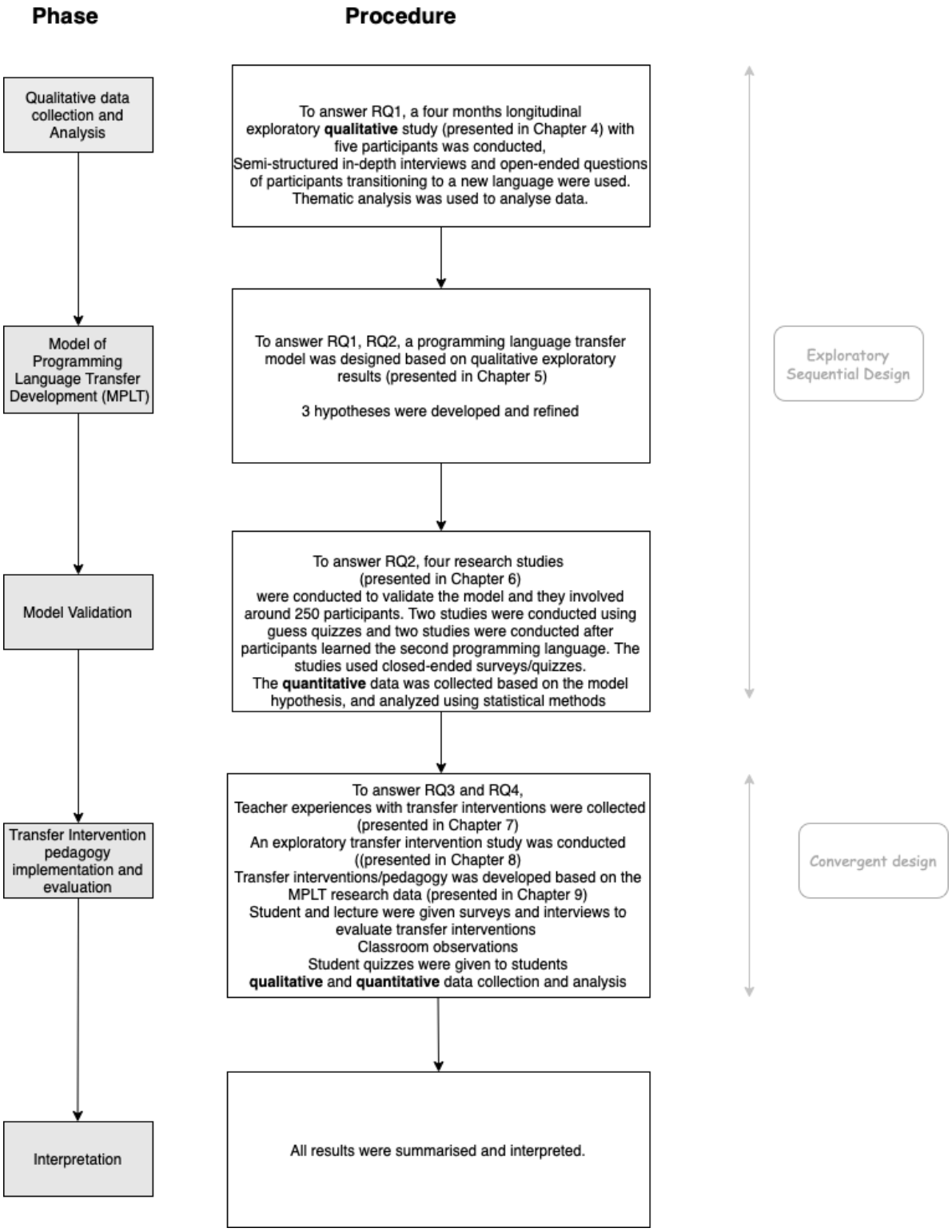


Figure 3.1: The adopted mixed methods design for this thesis. This design uses the Exploratory Sequential and Convergent design [75]. The studies were conducted for a period of 3 years.

3.2.2 Qualitative Studies

Qualitative research usually focuses on designing systematic strategies to ensure the trustworthiness of the results [78]. This thesis started with the study using qualitative methods because there is currently no guiding model of programming language transfer for novice programmers during code comprehension to guide data collection through quantitative methods. The process of analyzing the qualitative data was documented to ensure the validity of the results as advised in [78]. Furthermore, the researcher acknowledges that their viewpoints could be biased, affecting how the findings are interpreted. Other researchers were involved in the data coding process to reduce biases and increase the results' reliability. After creating themes, the researcher ensured that everyone involved arrived at similar or comparable findings. The chapters that entail these actual thesis experiments will elaborate more on how these issues were tackled.

3.3 Ethical Considerations

Ethical Considerations is one of the most integral aspects of research. It is important because this research involves interaction with human participants (i.e., students and teachers). The range of interactions in this research include; in-depth interviews, focus groups, surveys, quizzes and, even observing behavior. Following the guidelines of ethical consideration set for computing science research, formal permission was requested to conduct all the studies involved in this thesis from the Ethical Committee of the School of Computing Science before data collection. Once the Ethics Committee approved the studies (see approval letters in Appendix G), the next step was to collect data. Participants were provided with sufficient information to decide their participation. Before data collection, participants were assured that their identity (such as name/Identity number/address) was anonymous. The participants were given a research identity number that does not have their identifying information. They were also told that their participation was voluntary. They were informed they could withdraw before, during, or after this study. Participants were also given information sheets that explained the experiment's aims, objectives, and procedures. They signed a 'consent form' (see Appendix A) when they opted to be participants. Their data was only included in the study if they volunteered to participate by signing a consent form.

An ethical issue that may arise is when students may feel that if they do not permit the use of their data for research, this will reflect poorly on them and negatively impact their grade in the course because of the influence of power relationship with their lecturer. The lecturer did not know which students participated in the quiz because their personal information was not included in the surveys/quiz.

3.4 Summary of the Methodology

This chapter presented the methodology of this study and the rationale for using a mixed-method approach, followed by a discussion of validity, reliability, and ethical considerations. Since this thesis is built up sequentially with separate studies that depend on each other to answer the thesis research questions, the research tools, data collection, and data analysis will be discussed in detail separately. The first three studies in Chapter 4, Chapter 5 and Chapter 6 follow an Exploratory Study Design as shown in Figure 3.1. The first study explores semantic transfer in programming languages, and the second study designs a model for programming language transfer. The third study quantitatively investigates conceptual and semantic transfer in programming languages using the model. The last three studies in Chapter 7, Chapter 8 and Chapter 9 follow a Convergent study design as shown in Figure 3.1. The study presented in Chapter 7 explores teachers' experiences with the transfer, the Chapter 8 study explores transfer interventions, and the last study in Chapter 9 designs and investigates a pedagogy for programming language transfer. The next six chapters will elaborate on these studies and how they fit into the chosen research design.

Chapter 4

Exploratory Study on Semantic Transfer

[Aspects of this study have appeared in [29]]

This chapter presents the first phase of the *Exploratory sequential design* explained in the previous chapter. In this research phase, qualitative data is first collected and analyzed, and themes are used to drive the development of the programming language transfer model in phase two. For this thesis, a qualitative exploratory case study approach was used to get an initial understanding of how semantic transfer in natural languages applies to semantic transfer in programming language[5]. Jiang's work [26] on second natural language acquisition, presented in Section 2.5, claims that when learners learn their second natural language, the semantic transfer will occur. It means that second language learners initially base the meaning of the second language words on their first language translations. On the other hand, Ringbom's book on *Cross-linguistic Similarity in Foreign Language Learning* [28] has given insights that cross-linguistic similarity at the lexical level can be one of the ways that learners can make links between first and second languages [25].

Both natural language and programming language research are first considered as guidance in this thesis because of the similarity of the structure of these languages, both based on syntax and semantic rules. As already explained in Chapter 3, this thesis uses an exploratory sequential design to answer RQ1. This design starts by using qualitative methods because there is currently no guiding model of programming language transfer for relative novice programmers during code comprehension to guide collecting data through quantitative methods. Although some researchers may argue that deriving from existing literature and theories in qualitative studies may restrict the scope of the data, other researchers argue that this method may help organize the data and offer an initial way to explain causal relationships between the concepts [88, 89]. The analysis section will explain the analysis approach details used for this study.

This thesis hypothesizes that semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages. This chapter presents an in-depth exploratory study to test this hypothesis in programming languages to explore if se-

mantic transfer in natural languages is applicable in programming languages (RQ1). The study is an in-depth case study of five undergraduate students transitioning from procedural Python to OO Java. It is the first study conducted for this thesis and aims to provide the initial understanding of PL transfer processes of relative novices, specifically during code comprehension. The study asks the participants to read and explain their understanding of Python and Java programs. These explanations are analyzed through the lens of the contributions of key papers drawn from both programming language [9] and natural language [26, 28] transfer domains.

The thesis statement consists of two parts: transfer mechanisms and transfer interventions. To support the first part of the thesis statement *"Semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages;"*. this study aims to answer Research Question 1:

- *How are principles of semantic transfer in natural languages applicable to patterns of transfer in the context of relative novices PL transferring from first to subsequent PLs?*

The following sections thus present the methods and research design involved, participants' description, and the study context. Following this, a detailed account is provided of the data analysis procedures. Finally, the results of this study are presented, analyzed, and interpreted.

4.1 Participants

As a starting point, semantic transfer in programming languages was investigated to address RQ1. Because data collection is a crucial step in research, it was essential to make sound judgments when selecting the method of data collection and participants involved in the study [90]. Purposive sampling was used to understand semantic transfer in students learning new programming languages, which is a standard method used in qualitative studies [91]. According to Alkassim and Tran [90], Purposive Sampling is the deliberate choice of participants who possess specific characteristics that will better be able to assist in understanding the phenomenon being studied. It was, therefore, appropriate to engage participants' in transitioning to a new programming language in this study. Thus, the study used purposive sampling of five students with no Java experience who enrolled in a second-year university *Introduction to Java* class. The Java programming course ran for 11 weeks.

The course had three one-hour lectures and a two-hour lab each week. These participants are referred to as P1-P5. Four of the participants (P1-P4) had studied only Alice and non-OO or procedural Python in their first year, referred to as *novices* hereafter; one participant, P5, was an experienced programmer with six years of Python, C, and C++ experience. Their age ranged from 18 to 24 years, with three females and three males. The students were recruited by advertising for participation in the study from their Java classroom. Initially, eight students

responded; however, three withdrew from the study due to other commitments. Thus, the final number of participants was five second-year undergraduate students. Before the study, the participants were assessed on their knowledge of the Python programming language; see Listing A.2 in Appendix A. All the participants performed similarly and got the correct answer of the Python program output. Researching a small sample size is based on conducting dense observation and intensive analysis of qualitative data and changes in each student's programming language transfer mechanisms across several trials. It should also be noted that this study was not aiming to give statistical analysis but explore how the transfer works within a small population.

4.2 Procedure

This section presents the data collection approach. The data was collected by conducting individual 1-hour sessions with the students over ten weeks. The meetings were conducted outside class-time in the *School of Computing Science* meeting room in the University of Glasgow. Individual sessions were chosen instead of focus groups because they allow for the collection of detailed data by getting a single participant to talk freely and to express detailed beliefs and feelings about their transition to a new language experience [92]. Focus groups were not chosen because some participants may not participate freely and honestly because they feel intimidated by others; hence this might be a threat to validity [93].

The students in the study had learned procedural Python 3.0 and were transitioning to object-oriented Java 13. Although they were taught Java 13, they were not taught some of the language features; for example, they were taught the Java `for-loop` instead of the `foreach` loop during the time of the study. Therefore, what they had learned influenced the concepts covered in the code comprehension quizzes.

Data collection methods for this study included:

- **Think-aloud protocols**: Participants were asked to carry out code comprehension exercises in Python and Java using think-aloud techniques. According to Charters [94], using the think-aloud method is one of the most effective ways to assess higher-level thinking processes that involve working memory and evaluating individual differences in performing the same task. Therefore, this method was appropriate for assessing participants' transfer mental processes in learning new languages. Details of the interviews are in Appendix A.

- **Questionnaire**: Participants were given a questionnaire to note their confidence levels with the concepts/constructs currently being studied as they progressed with their learning. Details of the questions are in Appendix A.

A micro-genetic method was used [95] to study the knowledge transfer and development in learning a second programming language. This method was used to assess the progress in understanding cognitive developmental change mechanisms over time. It is a method well used in computer science research [96]. It involves:

- observations of individual students throughout the period of the change,
- a high density of observations relative to the rate of change within that period, and
- intensive trial-by-trial analyses intended to infer the processes that gave rise to the change.

Using this method meant that all participants' discussions were recorded and transcribed in each meeting. The transcriptions then directed the study activities, with each fortnightly group of interviews being informed by analysis of the previous interviews; these will be presented as *sessions*. In this study, Sessions 1-4 are presented as shown in Table 4.1. Session five is excluded as it does not address RQ1 but focused on helping students to correct their Java understanding as the study was coming to an end. The sessions are explained as follows:

- At the beginning of the first session, the participants were given the information sheet that explained a summary of the research project and its aims. It also had information about the duration of the project. If there were any questions the participants had, they were clarified. After reading the information sheet, the participants were required to fill out the consent form. This consent form was for them to give consent before they participated in the research. The issues of voluntary participation, anonymity, and confidentiality were explained to the participants. The demographics about their age and programming language experience were collected after.

  Participants were requested to complete a Python program comprehension task in the next activity; see Listing A.2. It was to understand their actual level of development and conceptual knowledge in their known programming language. Participants were given a 5-minute distraction puzzle to reduce the carryover effects of the Python quiz to the next activity of the Java quiz. Participants were then asked to complete a Java quiz to explore semantic transfer in programming languages; see Listing 4.1. Participants were asked to talk through as they executed the Java code line by line.

  The concepts covered in both the quizzes included variables and assignments, functions, parameters and arguments and if-statements. In the Java programs, the additional concepts of class and objects were included.

- At the second session of the study, students were presented with five multiple-choice code comprehension questions of matching Python and Java program snippets that they had already covered in the classroom; see Figure 4.1. The process involved identifying common concepts (e.g., composite types, if-statements, e.t.c) shared by the two languages

Table 4.1: The Design of the exploratory study showing all the four sessions conducted for a period of four months

Sessions	**Weeks into learning Java**	**Procedure**	**Deductive themes/codes based on existing literature**
Session 1	Week 0	Consent form, survey on experience, Python baseline test, Java guess quiz	Explore semantic transfer in programming languages [25, 26]
Session 2	Week 2	Five mapping multiple-choice questions on Python and Java. Confidence level survey.	Explore how semantic transfer occurs after learning Java concepts, explore what happens during negative transfer [9, 25, 26, 28, 45]
Session 3	Week 4	The for-loop code talk-through in Python and Java. Confidence level survey.	Explore the semantic transfer and negative transfer [9, 25, 26, 28, 45]
Session 4	Week 6	The Python dictionaries and Java objects talk-through. Confidence level survey.	Explore zero relation or novel concepts between languages and transfer interventions . [9, 28]

under study. The next step was to categorize the common concepts based on the similarity of syntax between the two languages. The language tokens that may stimulate transfer were chosen guided by the first session findings; in the first session, participants based their similarity on lexical tokens such as (print, if-else, brackets, and operators). Examples of concepts included were constrained by how the languages were taught in the classroom. For example, the `foreach` loop in Java was not used to match it to the Python `for loop` because the students had not learned it. Students were also given a Likert-scale survey on their confidence level with each learning concept.

- In the third session, the experiment involved students talking-through (discussing and explaining) how a for-loop in Python and Java works. Participants were also given a confidence survey.

- The last session explored how students view similar concepts represented syntactically differently in two languages. The session started by asking students to explain their experience of learning Java objects. Students were then given programs on Python dictionaries and the Java objects and asked to talk through their understanding of these programs. The exercise explored how to help students bridge their conceptual knowledge of these concepts. In addition, the students were given a survey to record their confidence levels.

Listing 4.1: Java Program used in session 1

```
1  public class Rectangle {
2    int length=3;
3    int width=4;
4    public static void main(String args[]) {
5      Rectangle r1=new Rectangle();
6      r1.insert(12,4);
7    }
8    public int insert(int l, int w) {
9      int result=l*w;
10     if (l > w)
11       System.out.println("The answer is " + l*w);
12     else
13       System.out.println("no result");
14     return result;
15   }
16 }
```

4.3 Analysis

Thematic analysis was chosen to identify recurring patterns of students learning a new language from their audio recordings and writings. Thematic analysis is commonly used in qualitative research to identify, analyze and report patterns in the data [88]. The themes were created using a hybrid inductive and deductive thematic analysis approach. This process merged the data-driven themes, and the themes derived from natural language and programming language literature [9,25,26,28]. The hybrid approach allows deductively exploring the concept of semantic transfer while also allowing for themes to emerge directly from the data using inductive coding.

Thematic analysis was chosen because it can be used for analysis in small or large data sets [97]. Therefore, it is suited to analyze the data from five participants in this study. The data analysis procedures for this hybrid approach followed the guidelines proposed by Fereday and Cochrane [98] and these include:

- Stage 1: Developing the Code manual
- Stage 2: Testing the reliability of the codes
- Stage 3: Summarising data and identifying initial themes
- Stage 4: Applying template of codes and additional coding
- Stage 5: Connecting codes and identifying themes
- Stage 6: Corroborating and legitimating code themes

In **Stage 1**, the kinds of transfer that occur when students read code in a new language were explored. Guided by prior work in natural languages [25, 28], the codes that involved the transfer of knowledge from a previously known programming language were observed. The codes developed include:

- Students making explicit reference to the previous language: This is derived from the idea of semantic transfer theory [25, 26] (e.g., When explaining Java, a participant mentions that Python also uses this...).
- Making assumptions and conjectures: This is derived from the idea of semantic transfer theory [25, 26] (e.g., A participant uses words like 'I assume it means this')
- Certainty: This is derived from the idea of Ringbom and Armstrong's work [9, 28] (e.g., A participant says 'Java works this way')
- Uncertainty: This is derived from the idea of Ringbom and Armstrong's work [9,28] (e.g., A participant says 'I'm not entirely sure')
- Making similarity relations: This is derived from the idea of Ringbom and Armstrong's work [9, 28] (e.g., A participant says 'Java uses brackets like Python')

In **Stage 2**, the reliability of the codes was tested by presenting them to the researcher's supervisor for comments and suggestions. The supervisor and the researcher agreed with the initial codes. Where there were disagreements, they were reconciled.

Stage 3 involved the process of reading, listening, and summarizing the raw data from interviews and written answers. This process was iterative.

In **Stage 4**, the codes developed were then matched with the results of the analyzed data. The initial codes guided the analysis of the data. Additional deductive codes were applied when the data produced new patterns.

In **Stage 5**, the themes and patterns in the data were being discovered. Differences in the four novices and the experienced programmer emerged, yet they also shared similarities.

In the final **Stage 6**, the themes identified from codes were clustered to describe the meaning that underpinned the phenomenon of programming language transfer for relative novices. There were iterations of the data analysis process before making any conclusions. The main themes guided the final discussion and paved a way to create a model suitable for programming language transfer.

The results are presented in detail per session, and each session contains a particular student/s case study similar to other student cases in that session. This kind of results analysis has been used in other computer science qualitative research [99, 100].

4.3.1 Session 1

Initially, in week 0, participants'were assessed on comprehension of a Java program, to which they had had not seen or learned yet. This program was a guess quiz, which means that the participants were required to predict the output without any knowledge of Java. A simple program to calculate the area of a rectangle (Listing 4.1) was chosen because it includes common constructs shared by Python and Java such as variable declarations, parameter passing, subprograms, calls, and if-statements. Some constructs were not familiar, such as classes and objects. Participants were asked to talk through as they carried out the program comprehension task and then give the program output. Participants had already learned these concepts the previous year in their Python course except for objects and classes.

Presenting all the students' recordings of interviews would have been a challenge. Therefore, representative cases of participants (P1, P2, and P5) are presented. These cases were chosen because they were representative (not necessarily identical) of the group of participants as a whole. That is, they capture the variation in results. Some of the participants were more vocal and explained in detail their mental processes. Other participants were not so expressive. However, the differences between the novices (P1-P4) and their level of expertise were not significant. Participants shared many themes and specific characteristics in the data.

Session 1 Results

This section presents the analyzed data using the initial codes described in the previous section as a guide. The examples given for students quotes are from Table 4.2.

Referring to the first language: With no prior exposure to Java, the participants were seen to explicitly refer to their previous language to understand Java; see Table 4.2. P1 and P2 were explicitly matching Java syntax to known Python syntax in order to understand the Java semantics [26]. For example, in their explanations of how the Java program works, P1 explains Line 4: *"It looks a bit like a Python function"* while P2 says: *"I can relate it to Python"*. P6 (the most experienced one), on the other hand, referred to several languages that seem similar to Java. For example, in explaining Line 4: *"I'm assuming since we have square brackets here it's an array-like C++"*. P5 also refereed to multiple languages in interpreting Line 10-13 (The If-statement): *"looks similar to several languages, C, C++, Python"*.

Conjectures and Assumptions: Participants were seen to be making assumptions or conjectures in explaining the Java code. For example, in explaining Line 4, P2 said: *"Maybe it's a name of a function"* while P5, in explaining Line 2, said: *"I would assume this is a declaration and an assignment"*.

Table 4.2: Session 1, analyzed students data of comprehension of a Java program in Listing 4.1 (guess activity)

Participant	**P1 (male)**	**P2 (female)**	**P5 (male)**
Experience	*one year experience- Only Python prior knowledge*	*one year experience- Only Python prior knowledge*	*six years experience- Python and C++ and other prior knowledge*
Line 1	"I don't quite know about line 1, it could look like a function maybe the function is called rectangle or it's the name of the program. **It's something I have not seen before.**"	"Hmm, **it looks completely new**. Hmm, I'm not entirely sure what line 1 is trying to specify."	**I did do C++ too and I am familiar with O-O**. I assume this will be a class you declare for an objects called rectangle.
Line 2	**Looks like** its assigning value to variable and saying that the variable is an integer, so its saying length is an integer with a value 3	It's the initial variable length set to 3 I guess. **It looks similar to Python**	**I would assume** this is a declaration and an assignment. We are declaring a variable called length which is of type integer and we are assigning it the value of 3.
Line 4	**looks a bit like a Python function**, there are brackets that seems to be a parameter.The name of the function is main. looks like its specifying a string argument. **I have never seen** public, static and main.	It says this public whatever. **Maybe it's** a name of a function because it has something in brackets. And maybe inside the brackets it's the parameters which are string. **I can relate it to Python**. It's just that there are **so many words I'm not used to**.	We are essentially here having a function called main. Im assuming it takes strings as arguments. **I'm assuming since we have square brackets here its an arrays like C++**. Im assuming void means it doesn't return a value. **I know static from C++.**
Line 5	I'm not entirely sure, **looks like its** calling a rectangle because of the brackets **which Python uses** when calling a function, but I'm not sure from where.	It **looks like** the name rectangle is something that they call. But I don't know. I mean if you have a rectangle and have empty brackets that will always be for me be calling a function without passing parameters in it. **Its similar to Python**	We are declaring a variable r1 which is of type rectangle which is the entire class, we are assigning to it an object of type rectangle. .
Line 6	**Looks like** its related to line 5. Like here,insert(12,4), maybe that's how you put arguments in a function.	We have had commands **similar to this in Python.** Maybe put something in a list maybe in position 12 and 4. I'm not sure.	This r1 we constructed earlier, we are calling the insert method with two parameters (12 and 4).
Line 8	It looks like another function, ehh , called `insert` with what seems to be integer parameters l and w, **looks like a tiny bit of Python**. It says it has to return an integer [unsure48 in the voice]. **I have never used public before.**	**It looks like** calling a function and putting in values. I think `l` and `w` are the parameters. **Its similar to Python.**	It's the public method called insert which we called earlier and it returns and integer .
Line 9	The result is going to be an integer and multiplies `l*w`, **similar to Python.**	It calculates something l and w the result is `l*w` and the result is integer.	This could be **similar to many languages like C, C++, Python without int**.
Line 10-13	It is an if-else conditional comparing if l is greater than w. Line 11 prints because it has the word print and prints the answer is l*w, if its false it goes to line 13 and prints no result.	It says if parameter l is bigger than w it should print out the answer is l*w. its an if-conditional **similar to Python. I don't now what system.out means but print is familiar**.	We have an if-statement with an expression that evaluates to true or false inside the brackets. `println` **looks similar to a number of languages, C, C++, python**. 'The answer is' is a string and l*w is an integer. .
Line 14	It returns whatever result is	The variable result is returned which is an integer	We are returning the value of result which is l*w .

Similarity Relations: Participants made similarity relations by comparing Java and previous languages based upon shared syntax similarities. Both P1 and P2 associated the brackets in Line 4 to a Python function. For example, P1 said: *looks a bit like a Python function, there are brackets that seem to be a parameter*. At the same time, P2 said: *"it's the initial variable length set to 3 I guess. It looks similar to Python"*. P5 also explained how Java looked similar to previous languages by saying: *"print in println looks similar to several languages, C, C++, Python"*.

Certainty: Sometimes the participants did not make explicit references or similar to the previous languages. In this case, they just interpreted the Java code with certainty, like they genuinely have no doubt. For example, all of them were showing confidence in how 'return' works in line 14, with P2 saying :*"The variable result is returned which is an integer"*. Line 6 also explained Line 6 by saying *"This r1 we constructed earlier, we are calling the insert method with two parameters 12 and 4"*.

Uncertainty: In some instances, participants expressed uncertainty/doubt in explaining the Java code. For example, in explaining line 1 '*Public class Rectangle*', P1 said: *"I don't quite know about line 1"* while P2 said: *"Hmm,it looks completely new "*.

Session 1 Discussion

The participants (P1-P5) made explicit references to their prior language, especially Python when explaining the Java code. P5, however, even though they knew Python, made most of their connections to C or C++. It is probably because Java is a statically-typed language like C/C++, and they have a lot of syntax similarities. In addition, C++ and Java are both object-oriented languages. This demonstrates that participants made connections between the languages and used the previous language to understand the new language, which they had not learned yet.

Participants made the connections by syntax mappings of Python and Java, or C/C++ and Java, in alignment with Ringbom's cross-linguistic similarities research [28] and subsequent semantic transfer in line with Jiang's semantic transfer theory [26]. It may mean that they interpreted the Java program based on their previous semantic knowledge. Take the interpretation of a Java method, for example. Participants mentioned that they noticed brackets and parameters, making mappings from a language structural/syntax domain onto a semantic and conceptual domain in their mental models. By this means, lines 2-4 and 8-14 were well enough matched to Python concepts of variables, function declarations, and calls, if statements, return statements, and output statements. This was also the case for the other students. All the students but one (P2) gave the correct output (The answer is 48) for the code. As they explained the program, student (P2) said:*"I think it will print no result, I mean I do not understand lines 4-7, if you take l to be length and w to be width, 3 is not bigger than 4, although I am not sure if it is being passed*

anywhere." What can be seen in (P2)'s think-aloud is that they ignored the lines of code that they could not match to Python, such as the Java main method and objects, and decided to explain the lines of code they could match, resulting in a wrong answer. This shows that perceptions of similarity between languages can vary amongst students [28].

If the participants could not make the match to Python, as in lines 1 and 5, then the syntax or construct was glossed over and perceived as Novel [9]. However, these lines of code were peripheral to how the overall program worked, hence the correct answers on the output. The more experienced student (P5) understood all lines and kept making analogies to either Python or C++.

This session presents the results of reading code in Java, new to five of the participants who have not learned yet. The themes emerging from these findings are that, during code comprehension, **conceptual and semantic transfer occurs from PL1 to PL2** as proposed by Jiang's model of second language acquisition. More importantly, the second theme is that **the similarities in syntax facilitated students in transferring the meanings/semantics and concepts to PL2** as proposed by Ringbom in cross-linguistic similarities. The following section explores the phenomenon of syntax mapping and semantic transfer further and observes any changes in the strategy students use after they have learned Java (a new language).

4.3.2 Session 2

In this session, students' development of learning a new programming language is explored after two weeks (6 hours) of learning Java in the classroom and attending two 2-hour labs. Changes in the strategy the students use when learning Java were observed too. The exercises include mapping code snippets of Python and Java programs. The students were asked to answer four multiple-choice questions; they had to pick the Java code matching the Python code. The exercise given is shown in Figure 4.1. By this time, participants were taught the Java concepts of variables, arrays, for-loops, and if-statements which they also learned in Python.

Session 2 Results

Most of the participants made correct mappings of the program execution outputs between the Python program and the corresponding Java program for concepts in questions 1 (arrays/lists) and 2 (if-statements) as seen in Table 4.3. However, students showed confusion in the mappings of questions 3 (for-loop) and 4 (static versus dynamic), especially question 3. For this mapping exercise, students were asked to fill in open-ended responses to their thoughts and rationale for their chosen answer. Unfortunately, only P2 filled in the responses. P2 got the Q4 mapping wrong, they mapped the Python program to Java version (a), and this is the rationale they gave:

- *"In Python, because you do not have to specify the type, printing out* `a` *would print 'Hello.'*

Question 1: Lists and arrays

Python		Java
mylist = [1,2,3] for x in range(len(mylist)): print (mylist[x])	a.	int array []= {1,2,3}; for (int x in range(len(array)) {System.out.println(x);}}}
	b.	int array []= {1,2,3}; for (int x=0; x<array.length; x++){ {System.out.println(array[x]);}}}
	c.	No equivalent
Answer		

Question 2: If-statements

Python		Java

a=3
b=9
if a<b:
print"a is less than b"
elif a>b:
print"a is greater than b"
else:
print"Bye"
	a.	int a=3;
		int b=9;
		if (a<b){
		System.out.println ("a less than b");}
		else if(a>b) {
		System.out.println ("a greater than b");}
		else {
		System.out.println ("Bye");}

	b.	int a=3;
		int b=9;
		if (a<b){
		System.out.println ("a less than b");}
		else (a>b) {
		System.out.println ("a greater than b");}
		else {
		System.out.println ("Bye");}

	c.	No equivalent
Answer		

Question 3: for-loops

Python		Java

for i in range(5):
if i==3:
i = 10
print(i)
	a.	for (int i=0; i<5; i++;)
		{
		if (i==3)
		{ i=10;}
		System.out.print(i);
		}

	b.	for (int i=0; i<5;)
		{
		if (i==3)
		{ i=10;}
		System.out.print(i);
		}

	c.	No equivalent
Answer		

Question 4: Dynamic versus Static

Python		Java
a=3 a="Hello" print (a)	a.	int a=3; String a="Hello"; System.out.println(a);
	b.	int a=3; String b="Hello"; System.out.println(b);
	c.	No equivalent
Answer		

Figure 4.1: Mapping multiple-choice questions on Python and Java presented to participants for the Session 2 activity.

Table 4.3: Scores of answers participants gave for the Session 2 mappings activity in Figure 4.1

Participant	Q1(arrays)	Q2(if-statement)	Q3 (for-loop)	Q4(static/dynamic)
P1	Correct	Correct	Wrong	Correct
P2	Correct	Correct	Wrong	Wrong
P3	Correct	Correct	Wrong	Correct
P4	Correct	Correct	Wrong	Wrong
P5	Wrong	Correct	Correct	Correct

> *If you need to assign the value* `a` *to a String, you need to write* `String a="Hello"`*. This is why I said answer (a) is correct. I am not sure what will actually be printed out when the Java code runs-probably "Hello" as well as in Python."*

Student P4 also did the same wrong mapping (option (a)) while the rest of the students mapped it correctly to (option (c)). P2 also got the result for question 3 wrong, and this is their thoughts of why they chose option (a) for this mapping:

- *"In code b, the fragment* `i++` *is missing in a Java loop, you need this to show that the value* `i` *should increase"*

The rest of the other three students gave the same wrong mapping of the option (a) just like P2 except for P5, who gave the correct mapping of the option (c).

Session 2 Discussion

In Session 2, just like in Session 1, participants are still referring to their prior language (Python) to perform the mappings of Python and Java. P2's explanation of re-assigning different types to variables (Q4) demonstrates this. P2 showed they made syntax mappings between Python and Java and assumed that because syntax differences are so small, the semantics are also similar. Note that P2 has the correct Python execution model for the *variable reassignment*, but carries the correct Python variable reassignment model across to the Java code showing that they base their interpretation on syntax similarities between Python and Java. The only difference they see is that Java specifies the type and Python does not therefore, both programs have the same meaning. This interpretation is evidence of semantic transfer [26]. This may mean that Python knowledge assists participants in their understanding of Java semantics in the early stages of learning Java. However, since Python semantic knowledge cannot completely reflect the semantic properties of some Java concepts, it may potentially lead to errors. In this question, variable `a` in Java is statically typed. That means this variable is initially declared to have a specific data type, and any value assigned to it during its lifetime must always have that type. Since Python is dynamically typed, variable `a` may be assigned a value of one type and then later re-assigned a value of a different type. Participant P4 also made the same mistake as P2 on this question. In addition to the two themes in Session 1, Session 2 presents two additional interesting themes

of **negative semantic transfer due to syntax similarity between PL1 and PL2** and **positive semantic transfer due to syntax similarity between PL1 and PL2** [9, 42, 45].

Another interesting finding is that the four participants (all novices) failed to map Python for-loop question to the Java for-loop. All of them mapped it to option (a). The following subsection explores the students' talk-through as they execute the for-loop in both languages to understand better why participants made the wrong mapping.

4.3.3 Session 3

Session 3 was carried out in week 4 of participants learning Java. Participants, by this time, had learned the new concepts of objects, classes, and encapsulation, not encountered in Python. The session aimed to further investigate the participants'comprehension of the for-loop in both languages as they showed misconceptions of this concept in Session 2. Students were asked to talk through their understanding of the for-loop execution of each language; the programs used are in question 3 in Figure 4.1, using Java option (a).

Session 3 Results

The results of two novices (P2 and P3) and an experienced programmer (P5) are in Table 4.4. These results show that participant P5 (more experienced) had the correct explanation and mental representation of how the Python and Java for-loop both work. P2 and P3, although they had the correct mental model for Python, they did not have the correct mental model of the Java for-loop. The participants not in the table, P1 and P4 had misconceptions in how to execute the previous language (Python) for-loop, P1 gave a wrong explanation that the `range(5)` generates a sequence of numbers starting from 1, not 0 as it should be the case. P4 terminated both the Python and Java loop at the end of the fourth iteration (when `i=10`). Both P1 and P4 already have misconceptions in understanding their first language and carried them forward to the second language, even though it is also possible that P4 might have carried the their correct understanding from Java to Python instead.

Session 3 Discussion

Participants P2 and P3 demonstrated a correct understanding of the execution of the Python for-loop. These participants then appear to have transferred the same semantic knowledge of the Python for-loop to explain the Java loop, resulting in the wrong Java output because Python and Java loops execute differently. In question 4 (Python version) in Figure 4.1, the range function returns a sequence of numbers from 0 to 4, with the loop variable taking on one of these numbers on each iteration. The loop will not terminate until all list elements have been processed, despite changing the iterating variable in the inner if statement. In Java, the for-loop variable retains its value across executions of the loop body, retaining any updates made in the loop's body.

Table 4.4: Session 3 activity: The students' talk-through in explaining the Java for-loop in Figure 4.1 (Question 4, Option a).

Participant	**P3 (female)**	**P2 (female)**	**P5 (male)**
Experience	*one year experience- Only Python prior knowledge*	*one year experience- Only Python prior knowledge*	*six years experience- Python and C++ and other prior knowledge*
Python Answer Python Explanation	**0,1,2,10,4** *"First iteration we have `i=0` and then it `i` incremented to 1 and then `i` is not 3, then it prints 1, then it is incremented to 2 and then `i` is not 3, then it prints 2, then `i=3`, then `i` changes to 10, because `i` is not 3 then it print 4..*	**0,1,2,10,4** *"The loop is supposed to loop 5 times. The first time `i=0,1,2,3,4` The first time it should print 0, then 1, then 2, the fourth iteration `i=10` so it should print 10. If `i` is 10 now, does it still go back to 4 now"*	**0,1,2,10,4** *"Especially range in Python is called a generator, it behaves as an iterable, basically what happens is. Essentially at each iteration of the loop it will get numbers 0 to 5 (5 excluded). So this is basically equivalent to 0,1,2,3,4. Now my understanding of this, it's generally bad practice to modify the loop counter. So at first `i` will be 0, then `i` is not 3 then we will print 0, `i` then will be set to 1, `i` is not 3 then it prints 1, then `i` is set to 2, its not 3 then it prints 2, then you get to `i=3`, `i=3` is true so `i` will be set to 10, then next I assume it prints 4 because of how the generator works so we get 0,1,2,10,4. We return 4 because `i` returns the next element in the generator/iterable irrespective of what you did to the loop variable."*
Java Answer Java Explanation	**0,1,2,10,4** *"The initial `i` value is 0, it will be 0,1,2,3,4 again. `i` is incremented by 1, so `i` is not equal to 3 it prints out 1, `i` is not 3, then it prints out 2, and then `i` is 3, it becomes 10, it prints out 10 [pause], then next it prints out 4 [does it though?] so it's the end of the for-loop, you stop here".*	**0,1,2,10,4** *"The Java one is a loop now, it will loop 5 times as well, `i` will be from 0 to 4. First loop it will print 0, then 1,2 and then 10 the next one may be 4 too."*	**0,1,2,10** *"So in C, to my understanding. For-loop will be an equivalent to the while loop, so basically you have i=0 then while i<5 you do the inner block then the last instruction will be `i++`. So initially we set this to 0, which runs the inner block until `i` is less than 5. At the beginning `i` is set to 0, so it checks if `i` is 3, its not then we printout 0, then we increment `i` by 1. We go to the beginning of the loop and then we check if `i<5`, its now set to `i`, then inner block, `i` is not 3 so we print 1, then `i` is incremented by 1, now `i` is 2, we check `if 2<5`, the loop body will still run, then we printout 2, then `i` is incremented by 1 and the next iteration `i` will be 3, so `3<5`, so the loop body will still run, `i` is 3, so `i` is now set to 10 so we print 10 here, so `i` is incremented by 1. Because `11>5` we terminate the loop."*

Hence the Java program exit out of the loop at the start of the fifth iteration. It appears that the participants ignored the conditional statement `(i<5)` in the for-loop that should be evaluated to true for the body of the for loop to be executed, or they did not appreciate that the update to `i` would carry errors outside the loop body.

Interesting thoughts from P2 were that: " *When I learned the for-loop, I was very confused at first because I had to add i++, but I feel it's very similar to Python* ." This participant assumed that because the syntax differences are so minor, the semantics are similar. It is evident that participants, even though it is their week 4 of learning Java, still refer to Python to understand Java, just like in Sessions 1 and 2. The similarity of some keywords like `'for'` make them make connections between the two languages and subsequent semantic transfer. These results support Jiang's model of second language acquisition based on semantic transfer [25]. The results also show that semantic transfer depends on the similarities between the languages. P5, however, understood the different forms of the for-loop well in each language, associating the Java for-loop with the identical (C++) syntax.

The results also showed that sometimes students carried misconceptions from Python (P1) or vice versa from Java (P4). It appears P4 transferred the Java semantic knowledge to Python by exiting the loop at the start of the fifth iteration for both languages. At this level, the the plausible explanation for this finding that was not derived from the natural language theories is that **learners may transfer semantic knowledge bi-bidirectionally from PL1 to PL2 and vice versa.** This may depend on where their level of learning the second language is. The other plausible explanation is that novice programmers **have fragile knowledge of the programming language semantics in their PL1**.

4.3.4 Session 4

When reviewing the students' confidence levels using a Likert scale score (level 0 being the least confident and level 5 being the most confident) with different concepts in week 6, most were very uncertain about objects as seen in Figure 4.2 and Table 4.5. This session, therefore, explores what students understood about objects. As well as students' understanding of objects, this session explores what transfer intervention can help them comprehend objects better. The concept closest to objects from their knowledge of non-OO Python is Python dictionaries. Python dictionaries and Java objects are both data structures/user-defined types although objects are more abstract [101] [58]. Having observed in the previous sessions that students transfer by syntax matching, they were given a Java objects program mapped to Python dictionaries and associated functions. Participants were asked to identify similarities and differences between the two programs and write down the program outputs.

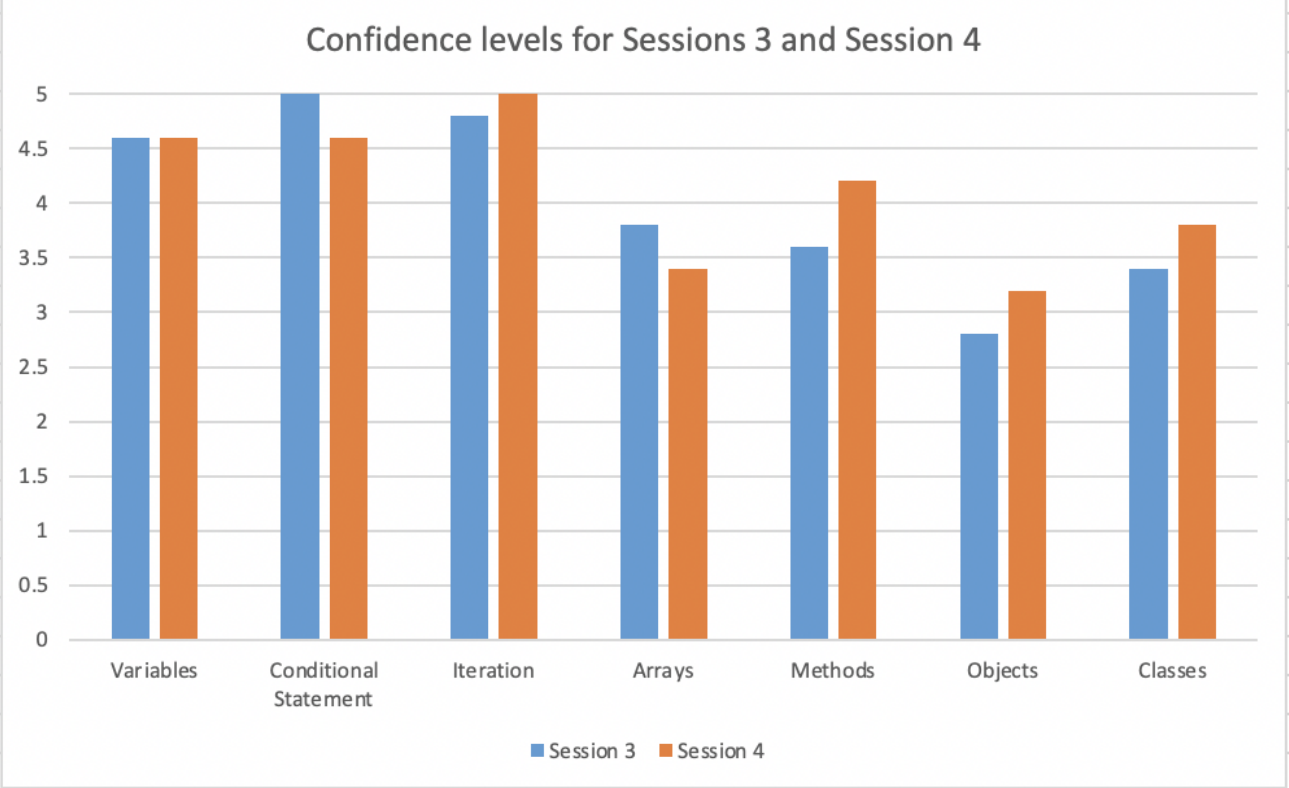


Figure 4.2: Session 3 and 4 participants' self-rated confidence levels on Java constructs

Session 4 Results

The session started with asking participants to talk through their understanding of objects since they were in week 6 of taking Java lessons. Their responses are shown in Table 4.5. All four novices showed that they struggled with the concept, expressing that it is complex and confusing to grasp. P5, who was more experienced, said that they understood objects well and still compared them to C++ at this point. Interestingly, none of the four novices referred back to their Python knowledge to explain objects this time around.

The participants in this study had not learned object-oriented Python in their previous class. That means they had not learned the Python objects and classes before learning Java. However, they had learned data structures such as lists and Python dictionaries, and learned Python functions. Therefore, the closest concepts to objects the participants knew of were the Python dictionaries. Objects and dictionaries both represent a data structure; the difference is that, unlike the dictionary, the implementation details of a Java object can be encapsulated. In order to explore the transfer intervention, Java objects program (see Listing 4.3) were mapped with the Python dictionaries program (see Listing 4.2). The Python program has a function (`p`) that returns dictionary values (`name` and `age`). Variable instances `me` and `you` are assigned returned dictionary values when the function is being called. These dictionary values can be manipulated with the `getName` and `incAge` methods. Similarly the Java program has a class `person` that

Table 4.5: Session 4: The breakthroughs, challenges with learning objects talk-through with participants on week 6 of learning Java

Participant	**P1 (male)**	**P2 (female)**	**P3 (female)**	**P4 (male)**	**P5 (male)**
Experience	*one year experience- Only Python prior knowledge*	*one year experience- Only Python prior knowledge*	*one year experience- Only Python prior knowledge*	*one year experience- Only Python prior knowledge*	*six years experience- Python and C++ and other prior knowledge*
Confidence about objects	*The whole concept of object is still difficult to understand. Can I like to create an object within another piece of code?*	*"Well, I thought an object is actually an interface or class ..what!...what! [she exclaims]. So I think its the most confusing part. I'm still not a 100 percent sure what they are. I mean our lecturer mentions objects sometimes but I still don't get it, I might have to go over our notes again."*	*"It was a very confusing concept to grasp. Like the objects have similar knowledge with the words in English. There are different ways to interpret the word so I got confused. But in Java I wonder if it could mean like variable"*	*"I still don't completely understand how you set them up and that kind of staff. I still don't understand how it works, the package the methods and everything. I get that you can use it for different things in the app and it can pass on values but I'm not a 100% sure on it."*	*"I have no confusions with objects because I did them in c++"*

defines `name` and `age`. It has object instances `n1` and `n2` with defined specific values of name and age as well as methods `getName` and `incAge` that manipulate these values. Two lecturers who taught programming courses at the University of Glasgow confirmed the equivalence of the Python and Java programs. One of these lecturers was not involved in the program.

Participants were presented with the Python and Java program in Listing 4.2 and Listing 4.3 respectively. They were asked to talk through the programs and discuss what they saw as similar or different. P2 said:

- *"An object is created in Java and then you create fields for the object in line 16 and 17 and those fields are similar to the entries in the dictionary for the Python in line 8 and 9. I see methods where fields are accessed in both programs for each person that is created ..mhh.. which has the name and the age, for example in the Java* `getName` *access the name field and in Python it gets name key for the dictionary."*

P4 also said:

- *"They both have 2 persons n1 and n2 (objects) similar to me and you (dictionaries). Each person has two fields name and age and increases age by 1. They have methods for each data and accessed differently"*

while the most experienced student (P5) said:

- *"In terms of functionality they are very similar. They have a data structure that has an int and string that get manipulated by functions"*

Listing 4.2: Python Dictionary Program given to students in Session 4

```
1
2  def p(n,a):
3      return {'name':n, 'age':a}
4  def getName(obj):
5      return obj['name']
6  def incAge(obj):
7      a = obj["age"]+1
8      return a
9  me=p("Joseph", 51)
10 you=p("Vic", 35)
11 print("your name is:", getName(me),", your age is:",
12 incAge(me))
13 print("your name is:", getName(you),", your age is:",
14 incAge(you))
```

The participants recognized some similarities between Python dictionaries/functions and Java objects as both having attributes and subprograms `getName` that manipulate them. They associated dictionary variables `me` and `you` with object variables `n1` and `n2`. Python dictionaries and Java objects share similarities in that they are both data structures, and the difference is that objects have a more abstract or hidden representation [102] while a Python dictionary has a concrete representation.

Participants were asked to discuss their understanding of objects again after making the comparisons. P2 said:

- *"It makes sense now. When I create an object, it doesn't show me the data structure or how it will be saved. Having a dictionary helped me visualize it because dictionaries are actually what I understand because, in a dictionary, you will know there will be a field called Joseph and a field called 51."*

Like P2, other participants seemed to change their views about objects after the Python dictionary and Java object comparisons, for example, *"When you say the word object, it seems like it's this abstract thing. I think now I see an object as an advanced way to store data and give you options to pass it around like lists and dictionaries with techniques to manipulate the data. it's a lot simpler to understand objects than what you kinda like think at first."* (P4). While P3 said *"I felt that an object is this abstract thing. But relating it to dictionaries makes more sense. I think objects are an advanced way of storing data and manipulating it with methods"*.

Session 4 Discussion

It was interesting to note that all the novices (P1-P4) had difficulty understanding objects in week 6 of learning Java. As they were asked about objects and their confidence levels, they

Listing 4.3: Java Program given to students in Session 4

```
1
2  public class person{
3      public String name;
4      public int age;
5      public person (String n, int a){
6          this.name=n;
7          this.age=a;
8      }
9      public String getName(){
10         return name;
11     }
12     public int incAge(){
13         int a=age+1;
14         return a;
15     }
16     public static void main(String []args){
17         person n1=new person("Joseph",51);
18         person n2=new person("Vic",35);
19         System.out.println("Your name is: "+n1.getName()+
20         " Age:" +n1.incAge());
21         System.out.println("Your name is: "+n2.getName()+
22         " Age:" +n2.incAge());
23     }
24 }
```

did not map or refer their understanding of objects to their Python knowledge, as was always the case in the previous sessions when they explained other concepts. An additional theme that was not covered in the last sessions was discovered. It was the difficulty of transfer with abstract concepts. Unlike previous sessions, participants could not make similarity relations/assumptions or even refer to their previous language to understand objects. The student misconceptions about objects are also well-documented [103, 104]. Understanding the difference between class and object seemed to be a challenge for novices.

The interventions were designed so that the Java and Python programs looked similar in Listing 4.2 and Listing 4.3 even though the syntax differed significantly. Hence they were able to transfer their existing semantics of data structures and functions across to Java, supporting Jiang's Semantic-transfer model [25]. This session suggests that it is harder for participants to carry out a conceptual and semantic transfer when similar concepts do not look the same. Mind-shift theory [9] was used to categorize the object concept as initially perceived as a Novel concept by participants. However, they later realized there were carryover concepts from their first language within the concept of an object expressed in a totally new syntax. The theme emerging from this session's findings is that **there is no semantic transfer when the syntax looks different for the same concept**. The other plausible explanation of the findings that is emerging as a theme is that **explicit teaching by comparing PL1 and PL2 may help students to activate transfer**.

4.4 Summary of Discussion

In answering the research question 1:

- *How are principles of semantic transfer in natural languages applicable to patterns of transfer in the context of relative novices PL transferring from first to subsequent PLs?*

The summary of the overall findings are presented as follows:

- Once the students identified similarities at the syntax level between the languages, the semantic and conceptual transfer occurred, and the learning process was positive, as shown in Session 1 and 2. These include concepts such as variables, conditional statements, methods, and parameter passing.

- During the initial learning stages, participants mainly relied on their syntactic matching/similarities between the languages and subsequent semantic transfer. This affected Carryover and Changed concepts as described by Armstrong [9] in Chapter 2. However, this matching meant that their mappings were wrong for Changed concepts, for example, Sessions 2 and 3. They showed they were transferring their Python semantic knowledge to (incorrectly) understand the Java concepts, e.g., the P1 and P3 examples in Session

Table 4.6: The overall themes derived from the Exploratory study

Clustered themes from Study 1 (deductive and inductive)	**Main Emergent themes**
1) Conceptual and semantic transfer occurs from PL1 to PL2	
2) Conceptual and semantic transfer can occur bi-bidirectionally from PL1 to PL2 and vice versa form PL2 to PL1	One plausible explanation of these findings is that relative novices may have one mental representation of the same concept in two languages if the syntax looks similar
3) The similarities in syntax facilitated students in connecting the meanings/semantics and concepts of the PL2	
4) Positive semantic transfer due to syntax similarity between PL1 and PL2	
5) Negative semantic transfer due to syntax similarity between PL1 and PL2	
6) There is no semantic transfer when the syntax looks different for the same concept	These findings suggest that similarity plays a crucial role in transfer between PL1 and PL2
7) Relative novices have fragile knowledge of the programming language concepts and semantics in their PL1	These findings suggest that sometimes misconceptions are transferred from PL1 to PL2
8) Explicit teaching by comparing PL1 and PL2, may play a role in helping students to transfer to PL2	These findings suggest that most often transfer happens implicitly

2. These results show that when syntax matching and corresponding semantic transfer take place for a Changed concept, they negatively impact the learning process of the new language.

- Initially, participants could not map the object concept to their existing knowledge structures because the syntax of this concept was at too high a level, not resembling any known syntax in procedural Python. The lack of syntax matching resulted in no semantic transfer, making their learning of these concepts less progressive. Participants were assisted through interventions to help them recognize that objects are not as novel as they thought and related to a concrete user-defined data type, implemented in Python using dictionaries and functions, as shown in Session 4. Introducing Python object-orientation first before teaching Java objects might have also improved the result. This was not practical in the current experiments due to constraints of the already running curriculum, but it would be good to explore in the future.

The overall themes derived from Study 1 are presented in Table 4.6. These themes capture a common, recurring pattern across a dataset from all four sessions. The clustered themes were derived from each session dataset, while the main emergent themes were derived from further clustering the themes to describe the meanings that underpinned the themes.

In summary, the results from the analysis revealed that semantic transfer did take place, in line with Jiang's model of semantic transfer for natural languages. The transfer is based on the matching/similarities of syntax [9, 28]. The principal factor influencing the success of the transfer was whether the matched syntax had the same semantic and conceptual meaning or not in the new language as in the old language, following on from Ringbom's cross-linguistic similarities [28] and Armstrong [9] 's Mindshift notions of Carryover, Changed and Novel concepts. In addition to the themes deduced from natural language theories, there were other plausible explanations from the findings that resulted in new themes emerging. These are as follows: relative novices may have one mental representation of the same concept in two languages when the syntax looks similar, the transfer may not only occur at the semantic level but may also occur at a conceptual level, the transfer may happen bi-directionally (PL1 to PL2 and from PL2 to PL1) as learning progresses, lastly, relative novices may have fragile knowledge of both concepts and semantics in PL1.

In the following study, a model of programming language transfer suitable for relative novices and guided by the exploratory study results and prior work on both natural and programming language is developed. In addition, the model derives from cognitive sciences.

Chapter 5

The Model of Programming Language Transfer

[Aspects of this study have appeared in [30]]

In this chapter, existing models of code comprehension [3, 24, 54], semantic transfer [26], cross-linguistic similarities [28] and cognitive sciences [182, 183] are merged and refined into a single developmental model of PL transfer, building on our earlier findings in Study 1. Study 1 findings revealed that semantic transfer did take place, in line with Jiang's model of semantic transfer from natural languages based on syntax similarities (derived from Ringbom's cross linguistics) between PL1 and PL2. Additional findings from the exploratory study are also presented and how they help build up the model to suit the context of relative novice programmers learning a new language. The chapter will start with the presentation of the Model of Programming Language Transfer (MPLT) and its components, and then the second subsection will explain the hypothesis of what happens when relative novices learn a second programming language.

5.1 Knowledge Structures in the MPLT

The programmer's knowledge structures are derived from code comprehension and natural language models. In code comprehension models, the programmer is represented to have both *syntactic and semantic knowledge* [3, 24, 54]. In these models, the programmer also has plan knowledge. However, plan knowledge is not the focus of this study, as explained in Chapter 2. In natural language models [25, 105], the language learner has a mental lexicon that comprises of *semantic, syntactic, phonological, and orthographic* knowledge. In programming languages, the phonological and orthographic knowledge will not be included in representing programming language knowledge, as explained in Chapter 2.

It is against this background that the MPLT is proposed to have three levels of knowledge:

concepts, semantics, and syntax, with a clear separation of concepts and semantics which is not the case for the mentioned code comprehension models. Recognizing the separation of conceptual and semantic knowledge means that when the programmer learns a second PL, they may have two representations of semantics specific to each language for a single construct. For example, the programmer who knows both Python and Java knows the *for-loop* concept shared by both Python and Java language, that is, a loop that repeats a fixed number of times. However, this concept has an *iterator-based* semantics in Python, while in Java, the semantics is *index-based*.

As proposed in Jiang's [25] second language acquisition model, which derives from Levelt's model [105], the mental representation of languages in a language learner is represented hierarchically. Also, the hierarchical knowledge structures are derived from the cognitive models of the Connectionist approach [106, 107, 182, 183]. Connectionist models represent the mental model structures as neural networks, where nodes are represented as neurons connecting to other nodes. The connections of knowledge nodes are based on prior knowledge/experience. Deriving from both natural languages and cognitive science, the programmer's knowledge is proposed to be built of a network of nodes that are connected hierarchically at different levels (syntax, semantics, and concepts), as shown in Figure 5.1.

- **The Syntax Level**: The programmer has knowledge of the syntax of a known programming language. For example, a variable declaration and initialisation in Java could be: *int mark =4;*.
- **The Semantic Level**: The semantics provide meaning to a well-formed syntax. This level is concerned with specific 'implementations' of the higher level concepts. For example, *variable **mark** is being declared as an integer and initialized to 4*.
- **The Concept Level**: This level contains the underlying concepts of programming languages known by the programmer. For example, *Variable declaration*.

The MPLT proposed that when a programmer is learning their first programming language (PL1), links between the nodes are developed, as shown on the left side of Figure 5.1. A path running from the Conceptual Level down to the Syntax Level represents; a concept, the concept semantics or implementation in the first language, and the concept representation in the syntax of that language. The *prior knowledge box* shows that when a programming language learner knows just one language, there is no branching across the semantic or syntax levels - because they have knowledge of just one semantics and syntax for a given concept. At the Conceptual level, a tree structure may emerge, given that a concept may have sub-concepts. For example, the concept of data type has the sub-concepts of primitive data types (e.g., int) and composite data types (e.g., dictionaries) (as captured in (a) and (b) in Figure 5.1, in the *prior knowledge box*). A concept presented as in (c) in the diagram could be, for example, a print instruction from a function - in a single language, there is usually only one form of this concept.

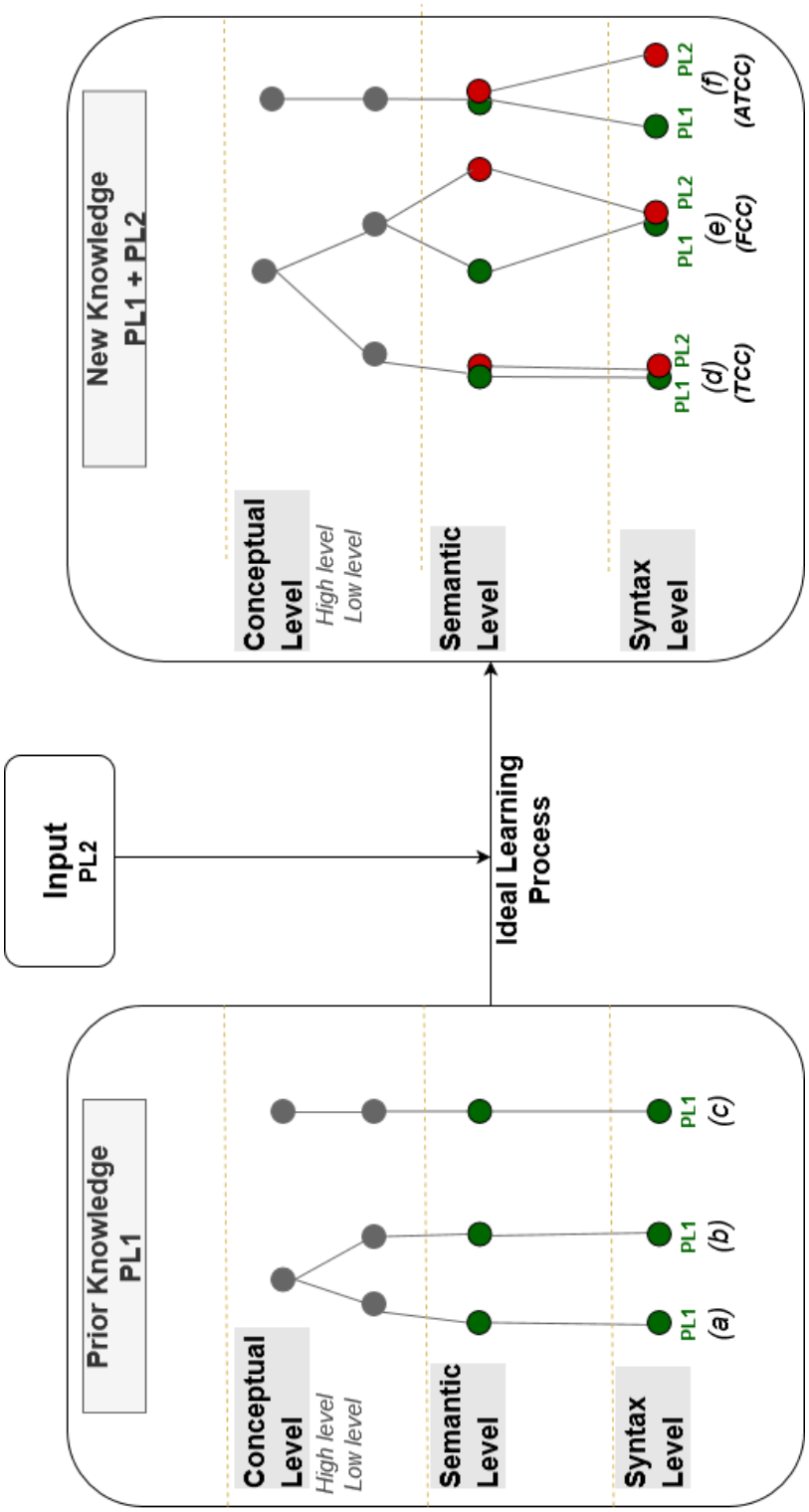


Figure 5.1: Model of PL transfer (MPLT)

When the PL learner learns the second programming language (PL2), more interesting knowledge structures are developed. Completely new concepts may be introduced, not shown in the diagram but would appear as a new' tree' at the conceptual level. Concepts (d), (e), and (f) all represent important but different kinds of ideal relationships between the concepts found in the two languages in a programmer's mental representation. Jiang's model does not cater to the different storage patterns of possible connections within the lexicon of a second language learner. In the MPLT, different storage patterns and relationships as proposed by Ringbom's cross-linguistic similarity relations [28] and Armstrong's concept categories [9] are proposed. These patterns were also confirmed to exist in the themes derived from Study 1, as shown in Table 4.6. The following concept categories represent these relationships:

- *True carryover construct (TCC)*: is a construct with *similar* syntax and same underlying semantics in PL1 and PL2, this is represented by the (TCC) concept branch in Figure 9.1. For example, a *while loop* in Python and Java.
- *False carryover construct(FCC)*: is a construct with *similar* syntax but *different* semantics in PL1 and PL2, this is represented by the (FCC) concept branch in Figure 9.1. For example an *integer division* in Python 3 and Java.
- *Abstract true carryover construct(ATCC)*: is a construct with *different* syntax but same semantics in PL1 and PL2, this is represented by the (ATCC) concept branch in Figure 9.1. Examples are constructs whose implementation details are hidden such as *data abstraction (objects)* [58] in Java which at a low level are data structures like Python *dictionaries*.

5.2 Predictions of what actually happens during Learning PL2

The learning process is derived from the theories explained in Section 2. Jiang's model of semantic transfer [25,26], Ringbom's cross-linguistic similarity relations [28], Armstrong's mind-shift theory [9] and cognitive sciences [45]. In addition, the learning process is derived from Study 1 findings. All the above models and Study 1 themes are merged and refined to suit the programming language transfer context. Study 1 findings showed that Relative novices have one mental representation of the same concept in two languages if the syntax looks similar. This means that the representation of TCC and FCC concepts should be similar. Even though Study 1 findings revealed that in some small instances transfer occurred from PL2 to PL1 (Theme 2; see Table 4.6), the transfer from PL1 to PL2 (Theme 1; see Table 4.6) is much stronger hence the hypothesis will be based on transfer happening from PL1 to PL2.

Therefore, based on the themes (Themes 4,5 and 6) derived from Study 1 in Table 4.6. I suggest that at the initial stages of learning PL2, programmers will engage in learning three

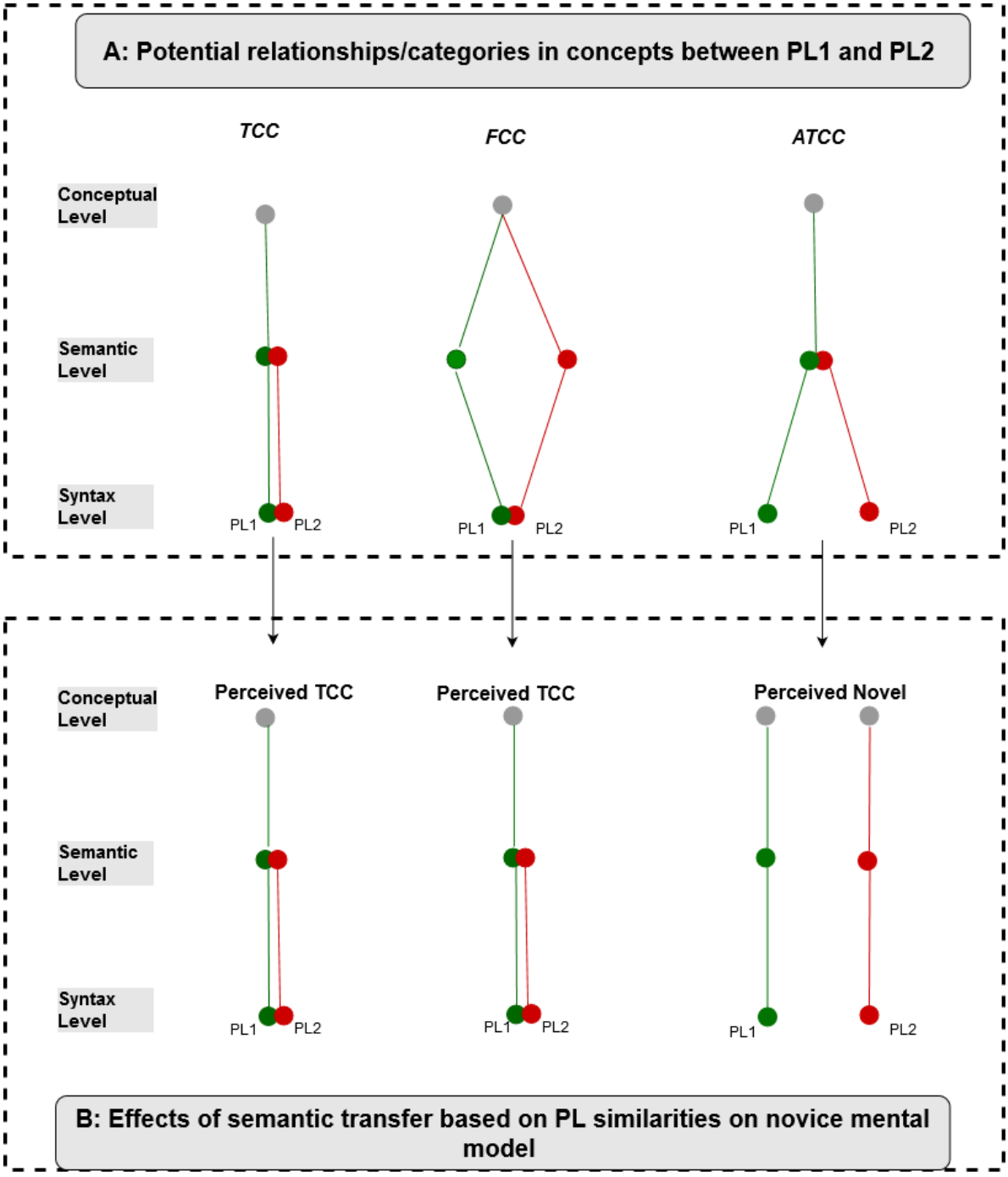


Figure 5.2: MPLT showing constructs categories a learner encounters during the learning process, and the consequences

categories of concepts, TCC, FCC and ATCC. These three categories are shown in *Box A* on Figure 5.2. The crucial point is that each of these categorised constructs can be *perceived* or *actual*. The *Box B* of Figure 5.2 predicts the learning process when students encounter these constructs during learning a new language.

1. If a learner encounters a TCC, they will perceive it as a TCC because of syntax match, hence semantic transfer will be appropriately effected from PL1 to PL2 and the learning will be positively impacted. This is represented in *Box B*, in the TCC column of the diagram. See how, even with two languages, the diagram is a straight line from concept to matched semantics to matched syntax.

2. If a learner encounters a FCC, they will perceive it as a TCC because of syntax match, hence semantic transfer will be inappropriately effected from PL1 to PL2 and the learning will be negatively impacted. This is represented in the central column of *Box B* in the FCC part of the diagram, which does not match the correct learning in *Box A*.

3. If a learner encounters an ATCC, they will perceive it as novel since there is little or no syntax match, hence there will be little or no semantic transfer . This is represented in the right column of *Box B* in the ATCC part of the diagram, which does not match the correct learning in *Box A*.

It should be noted that the MPLT hypothesis does not cater to the PL1 misconceptions that students may already have and transfer to their learning of PL2; therefore, Theme (7) in Table 4.6 is not represented. For example, if a learner's incoming knowledge of a PL1 `if-statement` is wrong, they may transfer the misconception of PL1 to PL2, which they will get wrong in both PL1 and PL2. The MPLT, however, proposes that if students learn PL2 correctly, it may strengthen their understanding of concepts as represented in Figure 5.2 in *Box A*. Theme 8 *'Explicit teaching by comparing PL1 and PL2, helps students to transfer to PL2'* is addressed in Chapters 8 and 9, which address second programming language transfer interventions.

Different possible scenarios can be represented in the MPLT. For example, a concept with similar syntax but different semantics and different conceptual roots, or a concept with different syntax and different semantics. It would be interesting for future studies to see how such different scenarios may affect the students' learning of PL2. However, for this study, the focus was on the three scenarios presented and adopted from Ringbom's similarity relations [28].

5.3 Summary

In this chapter, the qualitative (exploratory) findings from Chapter 4, together with theories from natural languages, programming languages, and cognitive sciences, were used to develop the MPLT to understand transfer in PLs. In the model, a relative novice programmer already has

knowledge of concepts, semantics, and syntax in PL1. The MPLT proposes that in the initial stages of learning PL2, a relative novice programmer will engage in learning three categories of concepts, TCCs, FCCs, or ATCCs; during the transition to PL2, they automatically transfer their semantic knowledge between PLs based on syntax matching, which may be positive or negative for their learning. The next chapter presents the quantitative studies that validate the model.

Chapter 6

Validation of the Model

[Aspects of this study have appeared in [30]]

The first research question in this thesis was to explore transfer and investigate how the principles of semantic transfer in natural languages applied to patterns of transfer in the context of relative novices transferring from first to subsequent PLs. The first study in this thesis, presented in Chapter 4, explored how novice programmers transfer knowledge between programming languages. The results showed that semantic transfer occurred while learning a second programming language based on syntax similarities.

Chapter 5, presented the development of a Model of Programming Language Transfer (MPLT) based on the first exploratory study results, natural and programming language theories, and cognitive theories. In the model, a relative novice programmer already has knowledge of concepts, semantics, and syntax in PL1. When the learner reads PL2 programs in the early stages of learning PL2, the semantic transfer occurs from PL1 to PL2 based on the syntax similarities between PL1 and PL2. While perceived syntax similarities may be helpful in understanding PL2 (TCC), they can also be harmful (FCC), and sometimes there is no transfer at all (ATCC).

This chapter concerns the validation of the MPLT. To do that, four studies (referred here as Study 2a, Study 2b, Study 2c, and Study 2d) that assessed students' comprehension of programs were designed as follows:

- It was important to give students a comprehension quiz in PL1 on concepts they learned to get the baseline of their existing conceptual and semantic knowledge.
- Students were also given a code-comprehension quiz in PL2 to assess the transfer process between PL1 and PL2. The PL2 quizzes were given in two sets of studies:
    1. The first set of studies (Study 2a and Study 2b) assess semantic transfer from procedural Python to object-oriented Java **before** the Java programming language instruction. These studies were designed to investigate the automatic/implicit transfer mechanisms based on the MPLT hypothesis.

2. The second set of studies (Study 2c and Study 2d) assess semantic transfer between procedural Python (version 3.0) and object-oriented Java (version 13) **after** Java programming language instruction. These studies are designed to investigate the transfer effects on students when they are taught the second PL, but without the teacher using specific transfer pedagogies to help them transfer. Furthermore, these studies will help investigate if transfer challenges persist even after PL2 instruction. In short, does typical PL2 teaching overcome/correct any incorrect transfers made implicitly by students and detected in the first point above? It should be noted that the curriculum restricted the concepts used in the study. For example, a Java `for loop` was used instead of a `foreach loop` because the students had not learned the `foreach loop` during the time of the study.

In order to investigate transfer using these kinds of code comprehension-based quizzes, it is necessary to analyze PL1 and PL2 to find common concepts. The second step would be to develop code snippets representing PL1 and PL2 for the same concept categorized into TCC, FCC, and ATCC. Lastly, it is necessary to find ways of administering these code comprehension exercises in PL1 and PL2 with the code snippets across the three categories of TCC, FCC, and ATCC.

This chapter begins with the proposed hypotheses that validate the model, followed by a section that describes the methods used to develop the transfer test instrument suitable for code-comprehension-based tests. The section involves the selection and analysis of common concepts, the process of designing code snippets, and the mappings of these code snippets into TCC, FCC, and ATCC between PL1 and PL2. The next section of this chapter describes the development of transfer quizzes. This section describes how the quizzes were designed from the code snippets and the way these quizzes were administered across the three categories FCC, TCC, and ATCC. The rest of the chapter describes the participants, data collection, data analysis, and study results. The last section of the chapter concludes with a discussion.

6.1 Hypotheses

The hypotheses for this study are developed based on the learning process by the MPLT. It is hypothesized that when students are assessed using code comprehension quizzes in PL1 and PL2, there will be differences in the understanding of concepts in PL2 as compared to PL1. The difference is measured by the score of individual concepts in a given quiz in each study in the three construct categories outlined by the model (TCC, FCC, ATCC). Thus the hypotheses for this study are as follows:

- Hypothesis 1: There will be no significant difference in the score for concepts involving TCC between PL1 and PL2.

- Hypothesis 2: There will be a significant difference in the score for concepts involving FCC between PL1 and PL2 such that PL2 score will be less than PL1 score.
- Hypothesis 3: There will be significant difference in the score for concepts involving ATCC between PL1 and PL2 such that PL2 score will be less than PL1 score.

6.2 Instrument

In order to explore the transfer between two languages using the MPLT, compatible testing instruments are considered a necessity. There are currently no validated instruments to measure PL transfer. Therefore, the focus of this section describes the development of the instrument and the rationale behind it.

Well-designed test instruments developed for their intended purpose have the potential to provide significant benefits to students and educators [108]. In order to ensure the quality and validity of the instrument, careful procedures were followed in its design. Some books in Education and Psychology have provided guidelines for developing valid tests. In computing education, Tew et al. [109, 110] proposed a validated language-independent assessment instrument (FCS1) for programming constructs for university students studying introductory CS1 programs. A Second CS1 Assessment (SCS1) was also developed as an isomorphic version of the FCS1 [86]. Other computing education researchers have also developed validated instruments for assessing the learning of computational concepts and constructs in different programming environments [111, 112]. However, the developed instruments do not cater for assessments that assess the transfer of knowledge between programming languages. Therefore, for this thesis, a custom PL transfer instrument for Python and Java was developed.

The approach followed in designing the instrument to validate the model is as follows:

1. Identification of the purpose for the instrument
2. Identification of common concepts in PL1 and PL2 for the instrument
3. Development of the code snippets for the PL concepts
4. Classification of common concepts between PL1 and PL2 into TCC, FCC and ATCC
5. Verification of categorisations
6. Development of the transfer quiz

1. Identification of the purpose for the instrument: The MPLT was developed based on how relative novice programmers read/comprehend PL2. To validate the model using comprehension-based quizzes, it was necessary to create an instrument that determines competency in under-

standing PL concepts relative to transfer and the MPLT. The development of such an instrument can be used as a guideline for research on other PL transfer contexts.

3. Identification of common concepts in PL1 and PL2 for the instrument: According to Sethi, [113], the origins of programming languages lie in the machines which have common concepts such as Data, Arithmetic operations, Assignments, and Control flow. High-level languages such as Python and Java were designed to allow programmers to write instructions in a language easier to understand than low-level machine language. Some common concepts underlie most of these high-level languages and may be represented syntactically in various ways.

The students in this study were transitioning from Python 3.0 to Java 15. This step, therefore, aimed to identify common concepts shared by the two languages under study (Python and Java). PL2 concepts that the students had not learned yet during the time of the study were not included in the instrument (e.g., *Inheritance* and *Polymorphism* in Java). Therefore, the existing curriculum restricted the choice of concepts during that time. For example, students had been introduced to Python functions in their first programming course and were now introduced to Java and object orientation, having not explicitly been exposed to object orientation in Python. Therefore a decision was taken to test whether students could benefit from conceptual transfer from Python functions to Java methods. Java methods are similar to Python functions, being a sequence of steps that can be referenced from elsewhere in a program. However, Java methods are scoped by their parent class, either as an instance of static methods, whereas Python functions are scoped by their parent modules. In addition, students had not yet learned some of the new features of the Java 13 version. The final list of common concepts between the two languages was as follows:

- **Variables and Identifiers** (Variable declaration and Assignments, Dynamic (Python) / Static typing (Java))
- **Primitive Types** (Types (e.g. Int, Boolean, Float) and Type conversions)
- **Operators** (Arithmetic and Logical operators)
- **Control Structures** (If-statements, For-loops, While-loop, Scoping)
- **Functions/Methods** (Parameters, Arguments and Return values, calls, Scoping).
- **Non-primitive types** (Lists, Arrays, Dictionaries (Python), Objects (Java))

3. Development of the code snippets for the PL concepts: Programming languages provide language constructs for organising computations [113]. These constructs are composed of lexical tokens and form the basic building blocks of programs that are syntactically allowable in a programming language. An example of a construct for a *Control structure* concept is the *While*

Loop. Programs can be built with one or more constructs; for example, when the program is longer, it will need more constructs and use some constructs repeatedly. In order to assess PL transfer for relative novices, it was best to select short code snippets that represent the lowest basic level of a PL concept identified in the previous step. The rationale for this was to reduce the number of difficulties the students might encounter that are unrelated to transfer. It should be noted that even with this approach, some of these basic constructs may encompass other constructs to be syntactically correct. For example, the syntax of the *While loop* consists of a *Boolean expression* and a block of code. The explanation for the semantics of a *While loop* is that when the *Boolean expression* is evaluated to true, then the block of code in the loop is executed. This continues until the *Boolean expression* is evaluated to false. The code snippets selected will be presented in the next step.

4. Classification of common concepts between PL1 and PL2 into TCC, FCC and ATCC The code snippets identified in the previous step were categorized into TCC, FCC, and ATCC as follows:

- **Identification of similarities in Syntax :** The initial step was to categorize the common concepts based on the similarity of Syntax between PL1 and PL2. The language tokens that may stimulate transfer were chosen guided by the first exploratory study findings. In this exploratory study, participants based their similarity between Python and Java on lexical tokens such as keywords (for, if-else, while), literals (int, Boolean, String), operators (+, -, /, *) and separators (, (, ;). Identifier names did not seem to affect the transfer process.

  Secondly, it was important to get the PL grammar/notation similarity. This is because even if the tokens are similar, sometimes the order of lexical tokens can be different between languages, affecting how students understand them. The Syntax of a programming language is usually specified using some context-free grammar that specifies the syntax rules of a language e.g., BNF [114]. For example, the rules of simple *addition* in Python and Java are `<addition> = <number>+<number>`. In the exploratory study, participants also based their similarities on pattern recognition.

- **Identification of similarities in Semantics:**

  The semantics of a programming language describes the meanings of syntactically valid programs or the processes a computer follows when executing a program in that language. As noted in the MPLT design, these come between concepts and syntax. The paired code snippets were compiled/executed in each language to ensure that the two languages share/do not share the same meaning for a given concept. In this scenario, the similarities in the meaning of a program represented in two languages were taken as the output it produces, rather than the formally-defined semantics embodied in the execution process

that generated the output. The limitation with this method is that the syntax in different languages may give the same output for certain examples, even though their semantics are quite different. This thesis, therefore, aims to provide the opportunity to discuss such differences in the classroom and give a deeper understanding of the underlying machine model.

It should be noted that there was no standard measure of similarity in the classification of common concepts.

5: Verification of categorisations: Academics well-versed in programming language design and the Python and Java lecturers from the two universities (Glasgow and Norway) reviewed the categorizations to provide content validity evidence and ensure that all constructs are mapped well. The initial categorizations were done by the researcher. The researcher then shared the categorizations with the supervisor for verification. After that, the categorizations were shared with two other lecturers who taught Java and Python. There were discussions between the researcher, the supervisor, and lecturers to verify the categorizations. Furthermore, some concepts in the categorizations were tested in pilot Study 1.

Each concept was categorized as belonging to FCC, TCC, or ATCC. The code snippets in both languages were made to look similar to simplify the explanation of the categorized examples, e.g., the identifier names and literal values were similar. The identifier names and literal values were not necessarily the same in the actual quizzes, as can be seen in the actual quiz (Java and Python questions) in Appendix B.

Figure 6.1 shows the example of a *While loop* concept, which was categorized as a TCC concept. Figure 6.2 is the *Type coercion* FCC concept categorised as FCC. An example of the *Data Structure* concepts is in Figure 6.3. This concept is categorized as ATCC.

The categorization process described here can be used as a guideline, but it is not rigid or a clear black and white process. The similarity of the two languages was based on the researcher's perceptions with confirmation from PL experts and lecturers. The degree of perceived similarity can differ from learner to learner and from teacher to teacher, based on several factors, such as the degree of exposure to other languages and the depth of knowledge of programming constructs. As noted early, the similarity determined by the researchers was reviewed by knowledgeable teachers and PL experts and deemed reasonable. Automated natural language tools exist, and while not used here, are described in further work.

6. Transfer Quiz Development: Given that the MPLT was developed based on relative novice programmers comprehending code in PL2, choosing a code comprehension-based quiz seemed

TCC (While Loop)	Similar syntax	Similar semantics	
		Execution of program	Output

Python	`x=0`
	`while x<2:`
	`  print(x)`
	`  x+=1`
		When the current value of the Boolean expression is evaluated to true, the embedded statement is executed. This continues until the expression becomes false.	0
			1

Java	`int x=0;`
	`while (x<2){`
	`System.out.println(x);`
	`  x++;`
	`}`		

Figure 6.1: Example of the *while loop* concept in the TCC category

FCC (Type coercion)	Similar Syntax	Different semantic	
		Execution of program	Output
Python	`'hello'+3`	Does not allow implicit type coercions	Error
Java	`'hello'+3`	Implicit type coercion *coerces* the integer value from a number into a string and then concatenates the two values together	hello3

Figure 6.2: Example of the *type coercion* concept in the FCC category

ATCC (Objects)	Different Syntax	Similar semantics	
		Execution of program	Output
Python	`n1={'age':3, 'name':'Agnes'}`	A data structure is being created with the fields (age and name) assigned data values (3 and Agnes) respectively.	

Java	`public class Person`
	`{`
	` int age;`
	` String name;`
	`public Person (int ages, String names)`
	`{  age=ages;`
	`name=names;}`
	`public static void main(String[] args){`
	`Person n1=new Person (3,"Agnes");`
	`} }`		

Figure 6.3: Example of the *data structure* concept in the ATCC category

Guess week-first lecture

At this stage in the first lecture, you are obviously not expected to know about the Java programming language as we haven't started yet. However, all of you do know at least something about programming, from your earlier studies and elsewhere, and the purpose of this exercise is to show that it can be of some use.

Write down what you *think* the output of the following Java code fragments will be, even though you may have to guess at some parts.

```
int i=0;
while (i<2){
  System.out.println(i);
  i++;
}
```

Figure 6.4: A sample TCC question on a *While-loop* construct which was given as a Java guess quiz **before** students were given the Java instruction

What will be the output of this Python program fragment?

```
x=0
while x<2:
  print("word")
  x+=1
```

Figure 6.5: A screenshot of a sample TCC question on a *While-loop* construct which was given as a Python quiz

to be justified. Each quiz consisted of tracing questions in which the participants predicted the execution of the code. These questions were chosen to determine the participants' knowledge more effectively, rather than using multiple-choice questions, where students are liable to guess.

To assess different kinds of semantic transfer, the questions in each quiz were categorized into three equal sets of TCC, FCC, and ATCC constructs. Each quiz consisted of one-third of the questions allocated to each model category. For example, each category (TCC, FCC, and ATCC) was allocated four questions in a quiz with a total of 12 questions. The four questions in each category were based on different constructs. The *identifier* names and *literal* values were different in the two languages, to reduce stimulating transfer.

An example of a TCC question in both Java and Python for a *While-loop* construct **before** students were given the Java instruction is shown in Figure 6.4 and Figure 6.5 respectively.

An example of a FCC question in both Java and Python for a *Type-checking* construct **after** students were given the Java instruction is shown in Figure 6.6 and Figure 6.7 respectively.

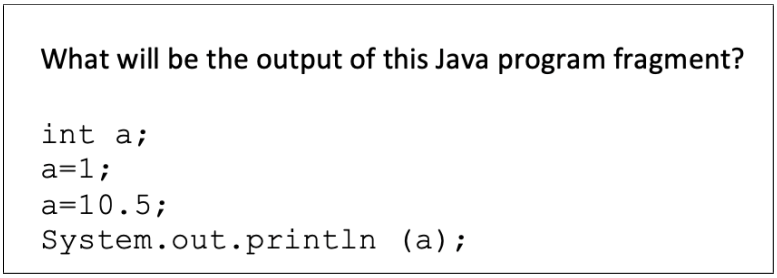


Figure 6.6: A screenshot of a sample FCC question on a *Type-checking* construct which was given **after** students were given the Java instruction

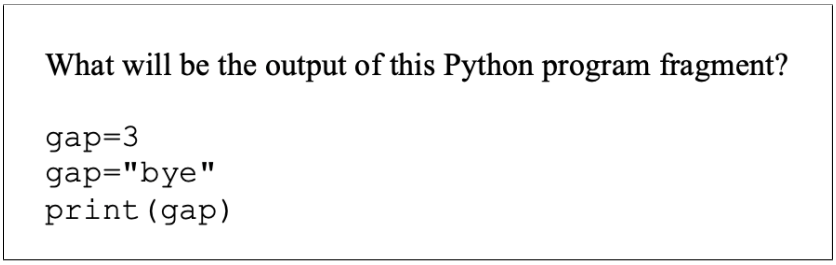


Figure 6.7: A screenshot of a sample FCC question on a *Type-checking* construct which was given as a Python quiz

An example of an ATCC question in both Java and Python for a *object-update* construct **after** students were given the Java instruction is shown in Figure 6.8 and Figure 6.9 respectively.

6.3 Participants

This section summarizes the participants involved in the studies that validated the model. It was appropriate to recruit participants transitioning from their first programming language to their second programming language to validate the MPLT's hypothesis and investigate how relative novices transfer knowledge to second programming languages. Therefore, the studies were conducted at the University of Glasgow in the Computing Science department. This university exposes students to different programming languages during their CS studies. The studies followed the order of students' enrollment at varying levels of their introductory CS courses. Therefore, the studies that included undergraduate students were conducted in their second year Java (PL2) course after they learned one year of Python (PL1) in the previous year. The study that included postgraduate students was conducted in their second semester of learning Python (PL2) after learning Java (PL1) in the first semester.

The study was advertised through emails and again in their first lesson of PL2 to recruit participants. All students were allowed to participate voluntarily and told that participation

What will be the output of this Java program fragment?

```
public class Robot {
  String label;
  int num;

public Robot(String n, int w){
  this.label=n;
  this.num=w;
}

public void agga(){
  num=num+2;
  System.out.println(num);
}

public static void main(String []args){
  Robot n1=new Robot("Nori", 51);
  Robot n2=new Robot("Alen", 22);
  System.out.println(n1.num);
  n2=n1;
  n1.agga();
  System.out.println(n2.num);
}
}
```

Figure 6.8: A screenshot of a sample ATCC question on an *Object-aliasing* construct which was given **after** students were given the Java instruction

What will be the output of this Python program fragment?

```
x = {'name':'Joseph', 'age': 51}
y = {'name':'Vic', 'age': 35}
print(x['age'])
y=x
x['age']=x['age']+1
print(y['age'])
```

Figure 6.9: A screenshot of a sample ATCC question on an *Object-aliasing* construct which was given as a Python quiz

Table 6.1: Participants in the four studies that validate the MPLT

Study	Participants	Participants/one language/no PL2 exposure	Age range	Year of study	Year of study	Language transition
Study 2a	84	46	18-23	Undergraduate (yr 2)	2020	Python to Java
Study 2b	241	101	18-23	Undergraduate (yr 2)	2021	Python to Java
Study 2c	120	70	18-23	Undergraduate (yr 2)	2019	Python to Java
Study 2d	50	33	23-37	Msc (yr1)	2020	Java to Python

would not affect their grades in the course. Students were also encouraged to use anonymous self-created identity names in the assessments. Participants received consent forms to participate in the study.

Participants were given a demographic survey to get their prior programming language experience. The research focus of the study was to find only participants with only one programming language knowledge; however, in a normal classroom setting of introductory programming courses, it is a challenge to find participants with only one programming language. Some participants reported that they know multiple languages, including some already exposed to Java programming language. Therefore, to increase the internal validity and strengthen the research design, the data that will be presented will be for homogeneous undergraduate participants who had no PL2 (Java) exposure and also had an average of one year of programming experience. The postgraduate participants had an average of six months of programming experience in PL1. The participants summary for each study is presented in Table 6.1.

6.4 Data Collection Procedures

The study consisted of two stages of studies. The first studies (Study 2a and 2b) were conducted at the first session of the Java programming language course **before** the students were given the Java instruction. The students were given an online code-comprehension guess quiz in Java. The students were then given a Python code-comprehension quiz covering similar questions. Both the quizzes consisted of a total of nine questions which consisted of three questions per concept category (TCC, FCC, and ATCC), Appendix B.

The second set of studies was conducted at week 3 (Study 2c) and week 6 (Study 2d) of students learning the second programming language; The students were given handouts of code-comprehension quizzes in both Java and Python. Study 2c had a total of 12 questions consisting of four questions per category, while Study 2d had 18 questions comprised of six questions per category, see Appendix B.

The quizzes took 20-25 minutes to complete for all four studies. All participants could finish the quizzes before or on the given allocated time without reporting any difficulty in completing the studies. Participants were given both the Python and Java quizzes on the same constructs during each study. To establish the baseline of participants' conceptual and semantic knowledge that they have already acquired from the first year, they were given a PL1 quiz. The least known language/ the second programming language was administered first, then after 10 minutes, the PL1 (most known language) was administered. It was to reduce the chances of stimulating transfer from Python to Java.

6.5 Data Analysis

This section reviews the processes followed at the data analysis stage. To validate the MPLT hypothesis, the quizzes were designed by mapping concept categories (TCC, FCC, ATCC) between the languages (Python and Java) as described in Section 6.2. This design helps find the participants' performance of different concept categories between the two languages based on the independent variables (TCC, FCC, and ATCC). This comparison in performance will give us a clearer picture of how transfer occurs when relative novices learn a second PL.

The results were recorded in an Excel document with '0' allocated for an incorrect answer and '1' allocated for a correct answer for each question. For example, a quiz with a total of 12 questions meant that each participant could be awarded a maximum of 12 marks.

The quantitative analyses for all the four studies in this chapter were conducted in an integrated development environment for R (RStudio 1.4.1103, 2009-2021). The mean score in each category was calculated, which helped analyze a more accurate score for conceptual understanding and semantic transfer between the two languages. To validate the hypotheses, the results of the Python and Java tests were compared. When comparing the significance in the score between the assessments, the Wilcoxon signed-rank test was used instead of a paired-sample t-test because the dependent variable (scores) was not normally distributed. The normal distribution was checked using the Shapiro–Wilk test. The Wilcoxon signed-rank test is a non-parametric test used to compare paired samples. These tests are appropriate for measuring the within-participant score results. To determine whether the statistical tests in these studies are significant or not, the usual consensus adopted by computer science education research [115] was followed. Thus, any p-value below .05 was considered statistically significant. The effect size was calculated too. A recommendation from Cohen's classification of effect sizes which is small (d = 0.2), medium (d = 0.5), and large ($d \geq 0.8$) was used.

6.6 Results of Transfer Before learning Java

This section presents the results of the first set of studies as outlined at the start of the chapter that evaluates semantic transfer on participants' first encounter with the second programming language **before** they were given the PL2 instruction. The primary goal of these studies was to assess the possible automatic or implicit semantic transfer when students have not yet learned the second programming language to extend the understanding of transfer between programming languages. The MPLT proposes that semantic transfer occurs due to syntax similarities between programming languages. The participants were asked to guess the Java programming language (PL2), which they had not learned before, to assess if their guess would match the MPLT's hypothesis. This section includes an analysis of a within-participant study that compares transfer between the three transfer categories of the MPLT. The section concludes with a discussion of the transfer findings presented. The two studies were conducted in the same Java course with students enrolled in two different years.

6.6.1 Study 2a Results

In this section, the findings from the Python quiz and the Java guess quiz given to 46 participants who had just completed their first year learning programming via Python and were now enrolled in the second year Java programming course in the year 2020 are presented.

The findings presented in this section include comparing the MPLT's TCC, FCC, and ATCC scores of the Python quiz and the Java quiz. The results showing the mean score grouped by concept category are presented in Figure 6.10.

A Wilcoxon signed-rank test using paired-samples indicated that there was a significant difference in performance as represented by the score data between the Python quiz and the Java quiz for the FCC ($p < 0.001$), ATCC ($p < 0.001$) and TCC ($p < 0.001$). In general, participants performed much better in Java in TCC (2.08, out of 3) than in FCC (0.14, out of 3) and ATCC (0.58, out of 3). The mean score for FCC in Python was 2.41 while in Java it dropped significantly to 0.14. A similar result was seen in the ATCC which showed a large significant drop in mean score from Python (2.15) to Java (0.58). Each category comprised of questions on different constructs, as a result, a Wilcoxon signed-rank test was computed for each construct for further analysis (See Table 6.2). The results showed that there were significant differences with medium-large effect size between all the constructs except for the TCC constructs, the *While loop construct* and the *String concatenation*.

Participants performed significantly better in the Python quiz than in the Java quiz in all the FCC constructs *variable reassignment, block scoping, and equality*. For example, in the *variable reassignment* construct, only 9% of the participants got it correct in Java while 91% of the participants got it correct in Python. Participants also performed significantly better in the

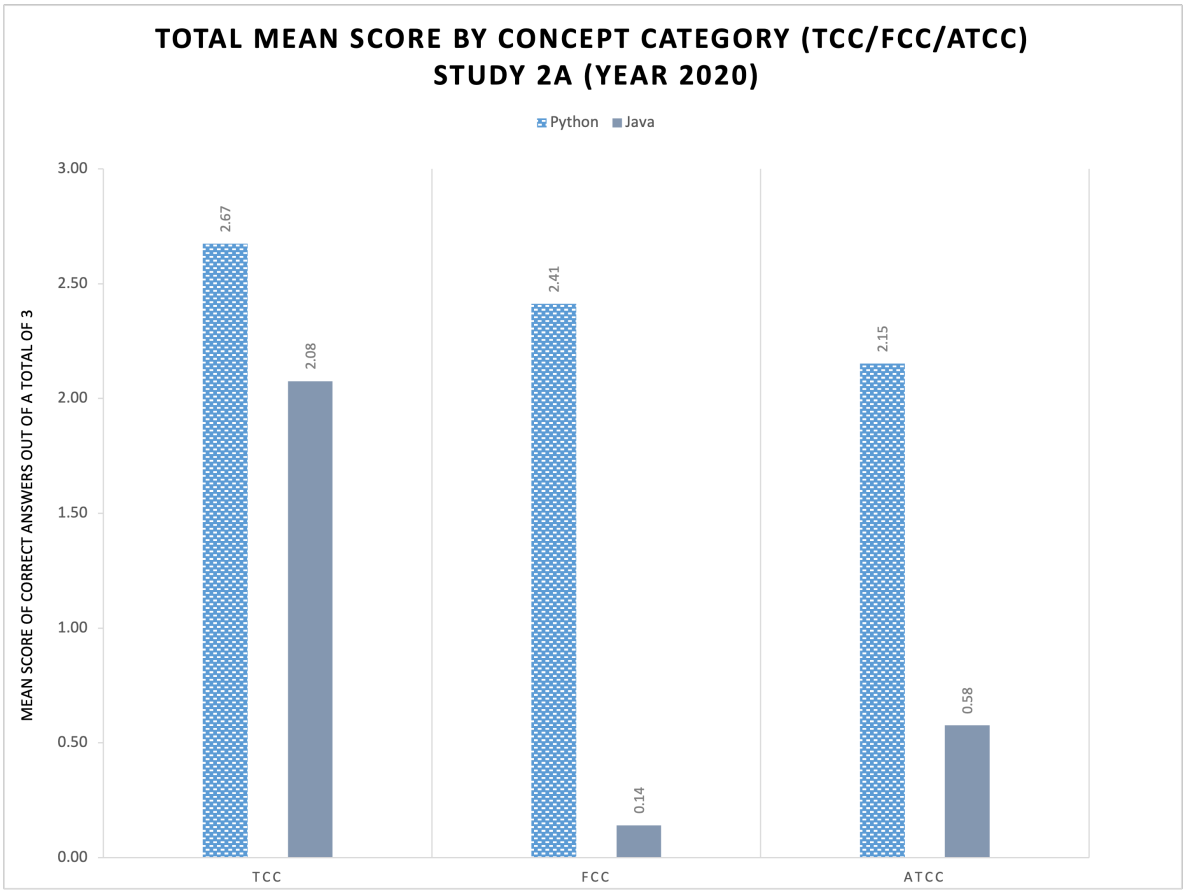


Figure 6.10: Study 2a Participants' mean score grouped by concept category and programming language: N=46

Table 6.2: Mean score, p-value and effect size of individual concepts tested in Study 2a Guess quiz: N=46

Category	**Construct**	**Python**	**Java**	**Wilcoxon P-Value**	**Effect-size**
TCC	String concatenation	0.95	0.84	0.07	0.32
TCC	While loop	0.78	0.65	0.09	0.30
TCC	Functions/methods	0.93	0.57	<0.001	0.72
FCC	String equality	0.86	0.04	<0.001	1.12
FCC	Block Scoping	0.63	0.01	<0.001	0.96
FCC	Variable Reassignment (static/dynamic)	0.91	0.09	<0.001	1.12
ATCC	Object update	0.81	0.19	<0.001	0.96
ATCC	Object retrieval	0.82	0.30	<0.001	0.85
ATCC	Object aliasing	0.51	0.07	<0.001	0.74

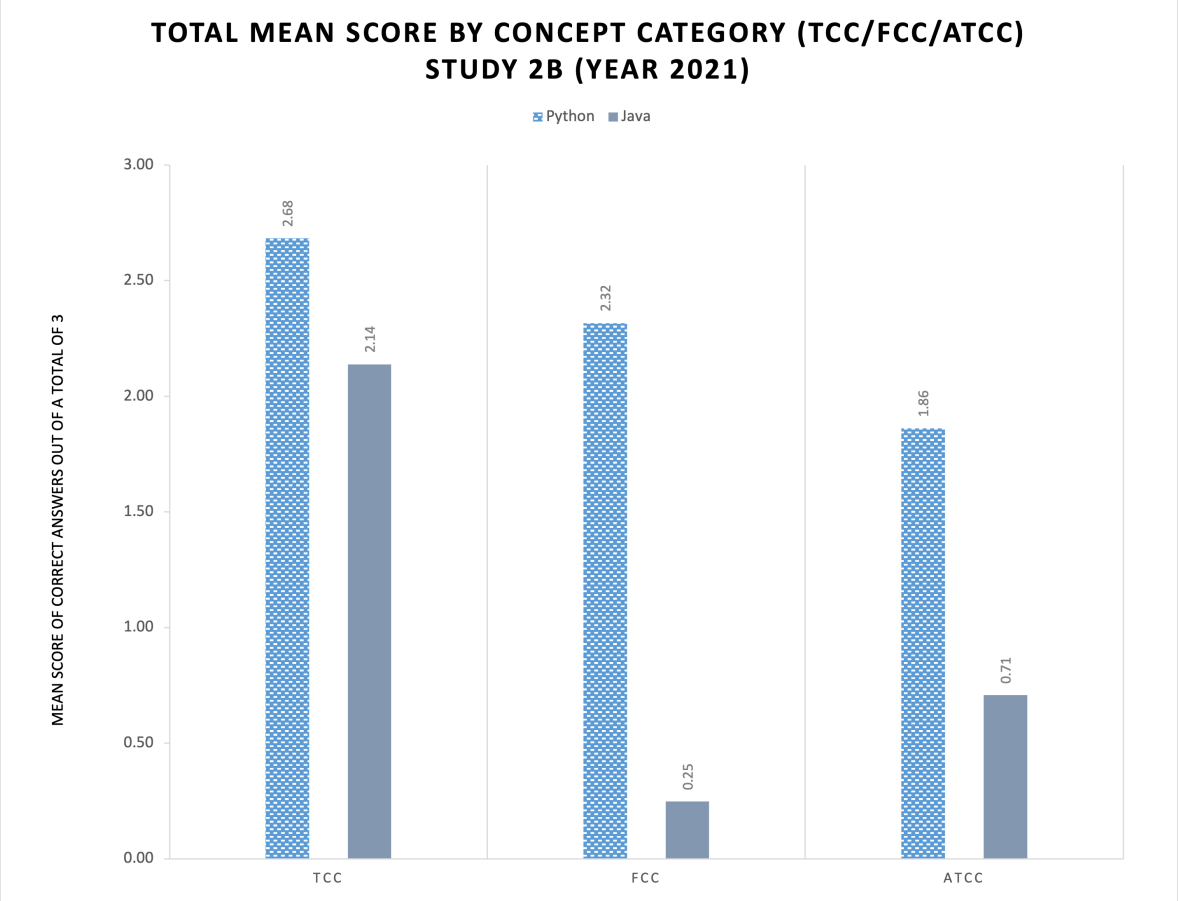


Figure 6.11: Study 2b Participants mean score grouped by concept category and programming language: N=101

Python quiz than in the Java quiz in all the ATCC constructs.

6.6.2 Study 2b Results

In this section, a replica of Study 2a is presented. The findings from the Python quiz and the Java guess quiz given to 101 participants transitioning from procedural Python to object-oriented Java in 2021 are presented. The aim is to determine if the replicated study can achieve the same or similar results as in Study 2a. This will give greater validity to the findings of implicit transfer and can mean that it is more likely that these results can be generalized to the larger population.

The findings presented included a comparison between the TCC, FCC, and ATCC scores of the Python quiz and the Java quiz, just like in Study 2a. This was to investigate how syntax similarities between Python and Java and subsequent semantic transfer assisted the participants in their first encounter with the Java (PL2) language. The results showing the mean score grouped by concept category are presented in Figure 6.11.

A Wilcoxon signed-rank test using paired-samples indicated that there was a significant

Table 6.3: Mean score, p-value and effect size of individual concepts tested in Study 2b Guess quiz: N=101

Category	Construct	Python	Java	Wilcoxon P-Value	Effect-size
TCC	String concatenation	0.94	0.87	0.02	0.41
TCC	While loop	0.83	0.62	<0.001	0.73
TCC	Functions/methods	0.91	0.64	<0.001	0.88
FCC	String equality	0.83	0.08	<0.001	1.57
FCC	Block Scoping	0.54	0.02	<0.001	1.33
FCC	Variable Reassignment (static/dynamic)	0.94	0.15	<0.001	1.60
ATCC	Object update	0.75	0.27	<0.001	1.20
ATCC	Object retrieval	0.79	0.38	<0.001	1.01
ATCC	Object aliasing	0.59	0.33	<0.001	0.90

difference in performance as represented by the score data between the Python quiz and the Java quiz for the FCC ($p < 0.001$), ATCC ($p < 0.001$) and TCC ($p<0.001$). The results revealed that participants performed much better in Java in TCC (2.14, out of a total of 3) than in FCC (0.25) and ATCC (0.71). Similarly to Study 2a, the mean score for FCC in Python (2.32) dropped significantly to 0.25 in Java. A similar result was seen in the ATCC which showed a large significant drop in mean score from Python (1.86) to Java (0.71).

A Wilcoxon signed-rank test was computed for each construct in the quiz for further analysis (see Table 6.3). The results showed that there were significant differences between all the constructs between Java and Python scores with most constructs showing a large effect size except for the TCC (*While loop* and *String Concatenation)*.

6.7 Results of Transfer After beginning to Learn Java

This thesis claims that semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages; the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language. The previous studies have already shown initial evidence of the first part of this claim. The purpose of the second set of studies (Study 2c and 2d) in this section is twofold. The first aim is to provide further evidence of semantic transfer proposed in the MPLT hypothesis in the context of the early stages of participants learning the second programming language. The second aim was to investigate the transfer process when the teacher does not use transfer strategies in the classroom.

Study 2c investigated 70 relative novice programmers (undergraduate students) transitioning from procedural Python to object-oriented Java, while Study 2d investigated 33 relative novice programmers (postgraduate students) transferring from object-oriented Java to procedural Python. At this point of the studies, the participants had learned three weeks of Java (PL2)

Table 6.4: Mean score, p-value and effect size of individual concepts tested in Study 2c: N=70

Category	Construct	Python	Java	Wilcoxon P-Value	Effect-size
TCC	String concatenation	0.99	0.96	0.424	0.14
TCC	Operator precedence	0.86	0.84	0.766	0.05
TCC	While loop	0.91	0.86	0.266	0.20
TCC	Functions/methods	0.84	0.71	0.069	0.40
FCC	Array equality	0.94	0.11	<0.001	1.34
FCC	String coercion	0.83	0.14	<0.001	1.17
FCC	String multiply	0.90	0.47	<0.001	0.82
FCC	Int division	0.73	0.54	0.024	0.41
ATCC	Object retrieval	1.00	0.59	<0.001	0.95
ATCC	Object update	0.63	0.59	0.298	0.04
ATCC	Object assignment	0.93	0.58	<0.001	0.85
ATCC	Object aliasing	0.63	0.11	<0.001	1

and six weeks of Python (PL2), respectively. The assessments allocated to the participants consisted of two code comprehension quizzes (Java and Python) categorized by MPLT concepts categories TCC, FCC, and ATCC.

6.7.1 Study 2c Results

This study aimed to determine if it can achieve similar results as in Study 2a and 2b and explore how semantic transfer persisted or changed as students started learning the PL2.

In analyzing the results of Study 2c, a comparison was made between the TCC, FCC, and ATCC scores of the Python quiz and the Java quiz. The aim was to test if semantic transfer from the known Python language affects the learning of Java positively or negatively by comparing the scores in these categories as predicted by the model. Figure 6.12 shows the findings of grouping participant mean scores of correct answers (out of a total of 4) by concept category and language from the two quizzes (Java and Python).

A Wilcoxon signed-rank test using paired samples was adopted for the analysis of the effects of transfer in the TCC, FCC, and ATCC categories. The results indicated that the difference was significant in performance as shown by the scores between the Python and the Java quiz for the FCC ($p < 0.001$), ATCC ($p < 0.001$) and TCC ($p = 0.018$). As expected, in accordance with the MPLT hypothesis, participants performed much better in Java in TCC (3.37, out of 4) than on FCC (1.27) and ATCC (1.89). The mean score for the Python FCC concepts was 3.40 but dropped significantly to 1.27 in Java, as shown in Figure 6.12. The mean score for the Python ATCC concepts also dropped from 3.21 to 1.89 in Java. These findings are similar to Study 2a and Study 2b when participants were guessing the Java quiz.

It was interesting to further investigate how the participants performed in each construct pair (TCC, FCC, and ATCC). Therefore a Wilcoxon signed-rank test was computed for each construct (see Table 6.4).

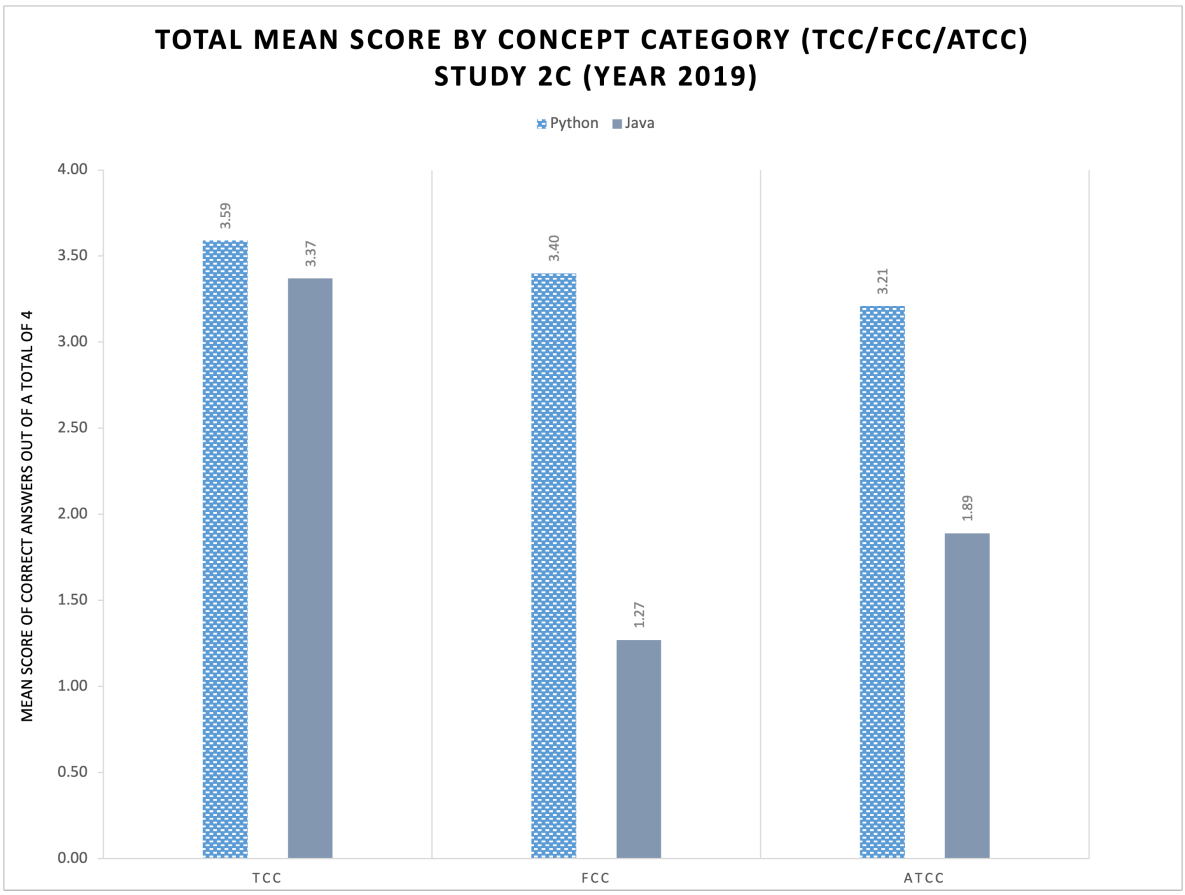


Figure 6.12: Mean scores of individual concepts tested in Study 2c when participants in week 3 of learning Java (PL2): N=70

As expected and proposed by the MPLT hypothesis, there was no significant difference in all the four individual TCC constructs between Python and Java quiz score, *while loop* (p=0.266), *functions/methods* (p=.069), *operator precedence* (p=.766), and *string concatenation* (p= .424). The effect size for all TCC constructs was also small.

Also, as expected, there was a significant difference in all the four individual FCC constructs between Python and Java quiz score FCC constructs. The scores of each construct reduced significantly in Java as compared to Python, *integer division* (p=.024), *array equality* (p<0.001), *string multiplication* (p<0.001) and *string coercion* (p<0.001). This implies that most participants performed poorly in Java (PL2) when the Java and Python syntax was similar, but the semantics were different. Three (*Array equality*, *String coercion* and *String multiply*) of the four FCC constructs showed a large effect size.

As expected, there was also a significant difference between the score of Python and Java constructs on ATCC concepts, *objects aliasing* (p<0.001), *objects retrival* (p<0.001), and *objects assignment* (p<0.001). These constructs also had a large effect size. There was no significant difference between the scores for the *object update* (p=.298) however.

The results of the analysis taken as a whole are consistent with the MPLT semantic transfer hypothesis. The participants' learning of TCC concepts was affected positively when they learned Java and negatively when they learned the FCC and ATCC concepts. These results seem to follow the same pattern as the results in Study 2a and 2b, but in this study, participants performed better in the *functions/methods* TCC concept. This is probably because unlike in previous studies, this time the participants had learned the concept in Java.

6.7.2 Study 2d Results

The previous studies revealed some fundamental issues of semantic transfer on relative novices transferring from Python to Java. In this section, the bidirectional transfer with 33 post-graduate students who have knowledge of Java (PL1) and are at their sixth week of learning Python (PL2) was investigated. This will give deeper insights into the claims that this MPLT hypothesis makes.

To examine the MPLT hypothesis, the TCC, FCC, and ATCC scores of the Python quiz (PL2) and the Java quiz (PL1) were compared. The results revealed participants scored higher in Python (PL2) in the TCC (5.33, out of 6) than on FCC (3.12) and ATCC (3.84), as presented in Figure 6.13. These findings corroborate with Study 2a, 2b, and 2c.

A mean score of the individual constructs tested was calculated as shown in Table 6.5 to gain deeper insights into the results. Just like in Study 2a, 2b and 2c, there was no significant difference in all six TCC scores between the two tests as predicted in hypothesis 1. The effect size for all the TCC constructs was also small. The p-value for each of them was, *operator precedence* (p=1), *functions/methods and parameters* (p=.233), *If-conditional* (p=1), *functions/methods and scope* (p=.789), *while loop* (p=.223) and *string concatenation* (p= 1).

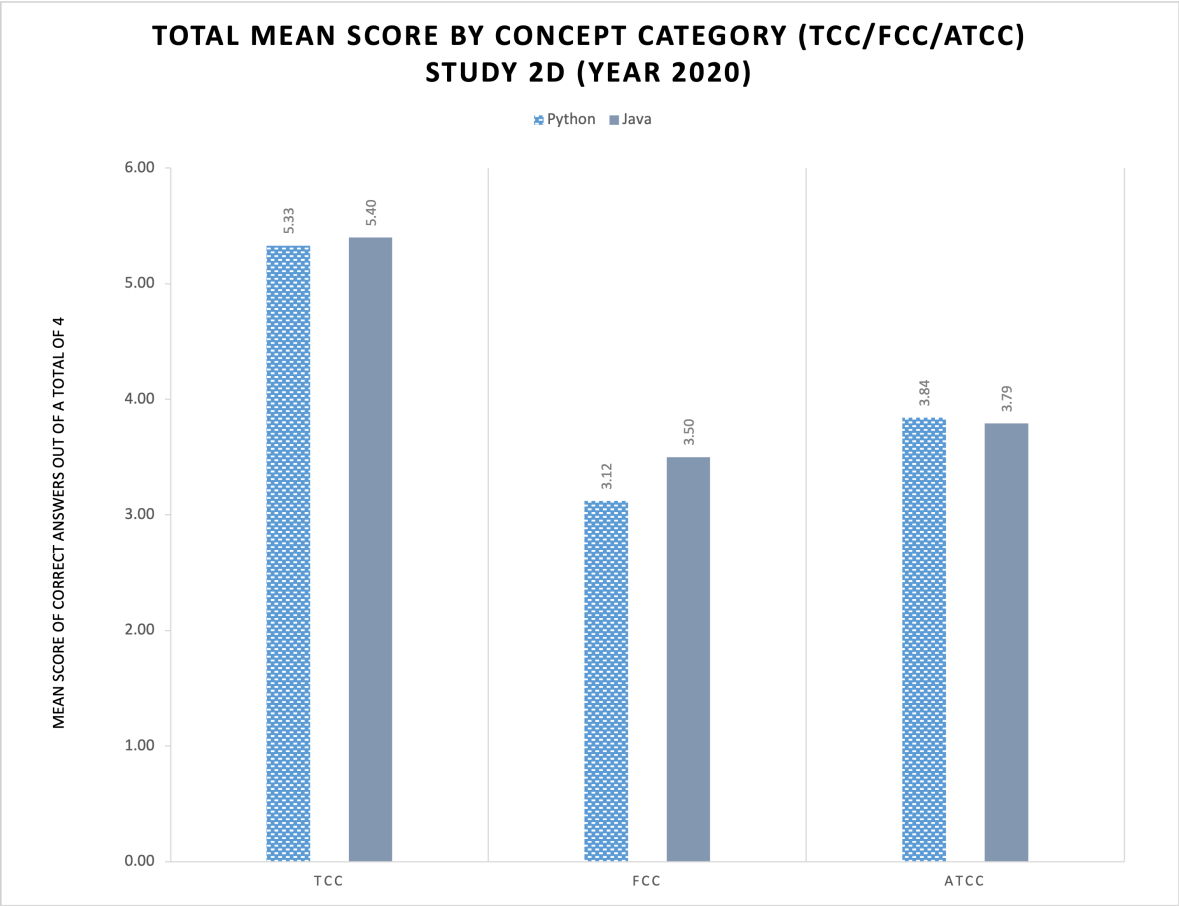


Figure 6.13: Mean scores of individual concepts tested in Study 2d of participants in week 6 of learning Python (PL2): N=33

Table 6.5: Mean score, p-value and effect size of individual concepts tested in Study 2d: N=33

Category	Construct	Java	Python	Wilcoxon P-Value	Effect-size
TCC	If Condition	1.00	0.99	1	0
TCC	String concatenation	1.00	0.97	1	0
TCC	Operator precedence	0.96	0.94	1	0
TCC	Functions/methods parameters	0.88	0.79	0.233	0.22
TCC	Functions/methods scope	0.79	0.76	0.789	0.04
TCC	While loop	0.76	0.88	0.223	0.22
FCC	For loop	0.91	0.15	<0.001	0.91
FCC	Int division	0.85	0.60	0.063	0.34
FCC	String coercion	0.67	0.09	<0.001	0.75
FCC	String multiply	0.39	0.69	0.008	0.48
FCC	Array equality	0.33	0.67	0.013	0.46
FCC	Type checking	0.33	0.90	<0.001	0.72
ATCC	Object retrieval	0.97	0.97	1	0
ATCC	Object update	0.97	0.97	1	0
ATCC	Object aliasing	0.53	0.27	0.023	0.41
ATCC	List retrival	0.58	0.73	0.145	0.26
ATCC	List update	0.56	0.73	0.145	0.26
ATCC	List aliasing	0.21	0.18	0.80	0.05

As expected and just like in the previous studies, there were significant differences in five of the FCC constructs scores between the two languages, *type checking* (p<.001), *string coercion* (p<.001), *array equality* (p=.013), *string multiplication* (p=.008) and *for loop* (p<.001) with medium to large effect size. However there was no significant difference in the FCC *int division* construct (p=.063). Interesting new trends emerged for the results, unlike in the previous studies of Python to Java transfer, the participants transferring from Java to Python had mixed results in the score of Java (PL1) and Python (PL2): for some instances as expected Java (PL1) score was higher than Python (PL2) score (e.g. *String coercion*) while for other cases unexpectedly Python score was higher than Java score (e.g. *Type checking*).

Unexpectedly again, subjects performed almost the same in Java and Python in the ATCC constructs with no significant difference in all constructs except for *object aliasing* (p=.023). The effect size for all the ATCC constructs was also small. The results revealed mixed patterns, with students scoring more in Python (PL2) than Java (PL1) (e.g. *list update* while in other instances they scored more in Java (e.g. *object aliasing*) than in Python. In these constructs, it appears students transferred better from Java to Python than from Python to Java.

6.8 Discussion

This chapter presented four studies aimed at validating the MPLT by assessing students' comprehension of programs in PL1 and PL2. The first set of studies (study 2a and 2b) evaluated the transfer from procedural Python to object-oriented Java before students were given Java (PL2)

instruction. The second set of studies (study 2c and 2d) evaluated transfer after students had been given PL2 instruction. In this latter set of studies, Study 2c investigated transfer from procedural Python to object-oriented Java while Study 2d investigated bi-directional transfer from object-oriented Java to procedural Python. The section presents the discussions of the results through the MPLT lens. Furthermore, additional emergent findings of relative-novice transfer are discussed.

Positive Effects of Semantic Transfer on TCC

Hypothesis 1 proposes no significant difference in the score for concepts involving TCC between PL1 and PL2. According to the MPLT, the rationale is that if a learner encounters a TCC while learning PL2, the semantic transfer will occur from PL1 to PL2 because of syntax matching, resulting in a positive impact on learning PL2. The findings revealed that all the four studies took the same direction in confirming this hypothesis with minor differences on the two sets of the study (before and after instruction). In all the studies, participants performed with higher scores in the TCC concepts as compared to the FCC and ATCC concepts. The participants were referring to their PL1 to understand PL2 as they also did in Study 1 (exploratory study), which positively impacted learning.

Before learning Java, participants in the first set of studies had no prior exposure to Java; therefore, they had no previous knowledge of Java syntax, semantics, and concepts. Even without the knowledge of Java, semantic transfer based on syntax similarities yielded positive results in Java for participants on the TCC concepts. Participants in the second set of studies performed even better than participants in the first set in these concepts by showing no significant difference in all the TCC constructs in both Python and Java. These results show that the positive learning effect on the TCC concepts in the early stages of learning PL2 happens rapidly.

The degree of perceived similarity between PL1 and PL2 may vary in different constructs, therefore, affecting learning PL2 on these constructs differently, yet still positively. For example, for all the four studies, participants performed equally similar (no significant difference in the scores) in both Python and Java in the *While loop*. Participants glossed over the minor differences and made connections with syntactic elements (e.g., *while, =, print,+*) and the pattern/notation of the two languages. It could be because, in addition to the token similarities between the languages, there is no difference in the pattern/notation of constructing a *While loop* in Python and Java as explained in Section 6.2. These positive effects of semantic transfer were also seen in the *String Concatenation* construct for all studies except for Study 2a.

However, participants performed with lower scores on the *functions/methods* TCC construct as compared to other TCC constructs. It meant that syntax/pattern similarity played a role in assisting semantic transfer. The reason for the participants performing lower in this construct is that there are fewer token similarities in (*functions/methods* as compared to the other constructs. Java methods can confuse some students because they have the declaration of parameter and

method return types, unlike Python. These findings show that the more similar the constructs, the easier to effect semantic transfer between the two languages. Also, another important point is that categorization of constructs can vary according to how the researcher/teacher perceives it. For example, the functions/methods could have been categorized as a TCC; however, maybe students perceived it differently. It brings in the dimension of student variations when recognizing similarities [28].

Other transfer studies in computing education have reported the benefits of language similarities when learners learn new languages [19,22,23]. For example, Scholtz and Wiedenbeck [19] reported that learners of the Icon language benefited from syntax matching of looping constructs that have similar syntax and semantics in Pascal.

Negative Effects of the Semantic Transfer on FCC

Hypothesis 2 proposes that there will be a significant difference in the score for concepts involving FCC between PL1 and PL2. According to the MPLT, if a learner encounters an FCC while learning PL2, the semantic transfer will occur from PL1 to PL2 because of syntax matching, resulting in a negative impact on learning PL2. This hypothesis was confirmed in all four studies in this chapter. Participants performed worse in the FCC construct than other constructs (TCC and ATCC). Unlike in TCC, there was no difference in performance on the FCC construct before and after the Java instruction. This means that the negative semantic transfer can be persistent even after students are given the instruction in PL2.

There was a significant difference in the score of the FCC construct in Python and Java for each individual FCC construct in all four studies, as predicted in hypothesis 2. In studies 2a, 2b, and 2c that covered transfer from procedural Python to object-oriented Java, participants performed poorly in all the FCC constructs in Java compared to Python. For example, in the *Variable reassignment* construct, participants showed they made syntax mappings between Python and Java and assumed that because syntax differences are so minor, the semantics are also similar. In Study 2a, participants had the correct Python execution model for the *variable reassignment* with 91% of them getting it right. Still, most of them (67%) carried the correct Python variable reassignment model across to the Java code showing that they base their interpretation on syntax similarities between Python and Java and then semantic transfer. It was also the case in Study 2b, with 94% of the participants getting the construct correct in Python and only 14% getting it right in Java. This shows that the concept carryover had inappropriately occurred, rather than just being unable to answer the question. In Java, a variable is initially declared to have a specific data type, and any value assigned to it during its lifetime must always have that type. Since Python is dynamically typed, a variable may be reassigned to a value of a different type.

In all the four studies, participants struggled with the *equality (==)* FCC construct. For example, in Study 2c, participants performed significantly better in the Python (94% correct)

than Java (11% correct) on this concept with (77%) of them having negative semantic transfer and giving an answer that corresponded to their Python answer. Learning was also impacted negatively on the *string coercion* concept in Java. Most of the participants transferred their semantic understanding of the Python concept and (76%) of them responded incorrectly that this Java statement will produce a Type error:`System.out.println("Friday is no:" + 1 )`. These participants gave the same response in Python (83%): `print ("Friday is no:" + 1 )`, in this case, they were correct. Unlike Python, the Java compiler can implicitly convert an integer value to a string. This provides compelling evidence of negative semantic transfer from Python to Java.

The difference in performance can be attributed to semantic differences between Python and Java on concepts that look similar. The participants perceived them as TCCs hence transferred their Python understanding of these constructs to Java. As described in the MPLT in Chapter 5, learners mostly have *one* semantic model of the FCC in two languages even if the semantics of that concept is different. The MPLT predicts that at the early stages of learning PL2, learners struggle with holding *two* semantic representations of the same concept in two languages. This means that their learning is affected negatively by these concepts. Just like in the TCC findings, these results show evidence of implicit semantic transfer, which means that, without explicit instruction, the transfer mostly happens automatically and unconsciously when the syntax between the two languages looks similar **before** or **after** having learned the PL2.

Overall, students did not benefit from mapping similarities between Python and Java when learning their second programming language in the FCC concepts. The negative effects of semantic transfer have also been reported in prior work in computer science education. For example, Walker et al. [12] evidenced negative semantic transfer of Pascal (PL1) statement, `writeln`, to a similar (but not semantically equivalent) Ada (PL2) statement `Put Line`. Weintrop and Wilensky [17] also reported errors in students' programs such as wrong relational operators (e.g., ==), undefined variables, and incompatible type errors when students transitioned from block-based languages to Java. In addition, other researchers in natural languages have reported the persistence of semantic transfer as observed in this thesis [26]. Schmitt also reported that semantic development could be a slow and unsuccessful process when learners learn new languages [116]. Schmitt's findings also found the persistence of negative semantic transfer even after students had learned the concepts in PL2.

Minimal/no Transfer on ATCC

Hypothesis 3 proposes that there will be a significant difference in the score for concepts involving ATCC between PL1 and PL2. According to the MPLT, if a learner encounters an ATCC while learning PL2, the semantic transfer will not occur from PL1 to PL2 because the syntax between the two languages looks different. Three studies of the transfer from procedural Python to object-oriented Java confirm this hypothesis. In these studies, the participants performed sim-

ilarly in the ATCC constructs **before** and **after** the Java instruction. This is evidence that when the syntax of the two languages looks different for the same concept, participants fail to implicitly transfer semantic knowledge from PL1 to PL2 on a concept they already know. Participants were unaware they were already familiar with these concepts in their PL1.

For example, there was a lack of semantic transfer from Python to Java on the *object aliasing* concept. Participants performed significantly better in Python (63% correct) than Java (11% correct). However, this was still a challenging concept for them in Python [117] as shown by 37% of the participants still struggling with this concept in Python. However, the 63% of students who knew this concept in Python still failed to transfer this knowledge to Java. The students who failed the Java *objects aliasing* gave the answer of *copy semantics* instead of *reference semantics*, however, both Python and Java use *reference semantics* in composite data structures [58] like objects. Failure to transfer this knowledge in the context of Scheme to Java was also reported in Fisler at al. [117]. The similarity in *objects* and *dictionaries* is that both of them represent a data structure, the difference is that unlike dictionaries, Java objects implementation details are hidden [114]. The implementation details can be hidden, although the learner is usually both the creator and user of a class and its objects and so may know, in principle, that it is a data structure with fields, like a record or dictionary entries. These findings showed that relative-novices have two mental representations of the same concept in two languages if the syntax looks different as proposed in the model.

The results in Study 2d (transitioning from object-oriented Java to Python) were, however, different from the other studies in that students performed almost the same in Java and Python in the ATCC constructs. This could be because students found the Python programming language easier to understand in these concepts when transitioning. This finding opens avenues for further research.

Bi-directional Semantic Transfer

Other emerging findings in Study 2d that assessed transfer from object-oriented Java to procedural Python revealed that the semantic transfer could happen in both directions. It means that semantic transfer can occur from PL1 to PL2 and vice-versa from PL2 to PL1.

For example, Participants performed significantly better (67% correct) in the Java *string coercion* versus the Python (1% correct). This is attributed to negative semantic transfer from Java (PL1) to Python (PL2). On the other hand, most students performed better in Python (67%) than Java (33%) in the *Array equality*. Other natural language transfer studies have also reported Bi-directional transfer shown in this example [118–120].

The results from Study 2d can be attributed to several reasons: students' prior knowledge on the construct may have been very fragile hence changing easily (these participants had only four months of learning Java (PL1)), the FCC construct may be more intuitive in Python than Java, or the students were taught the construct in the Python language first before the Java language.

Another factor that may influence transfer is the level of study for the participants, e.g., these participants were in their sixth week of learning the second language, which could mean they end up knowing it more than the first language. But these findings are still crucial to the validation of the MPLT hypothesis. As the model proposes, students in this study still held *one* semantic model of the two languages; they struggled with having *two* semantic representations of a single concept as elaborated in the model of PL Figure 5.2, *Box A*. These findings corroborate the previous three studies.

6.9 Summary of Discussion

The findings from this chapter are important for studying transfer in second programming language learning. First, they provide compelling evidence of semantic transfer between programming languages as stated in the thesis statement and the MPLT. The results from these various studies replicated the similar semantic transfer effects with different types of participants across different time periods. The replication effect proves that semantic transfer is common among relative novice programmers. The findings demonstrate the usefulness of the MPLT adopted in this study as an objective and reliable means of examining transfer in second programming language learning in the context of Python and Java.

In analyzing all the studies included in this chapter, we find that:

- The TCC concepts were found to be the easiest to learn when transiting from PL1 to PL2, as confirmed in hypothesis 1. These findings are attributed to the syntax and semantic similarities PL1 and PL2 share. It was also revealed that even before the students learn PL2, they can implicitly make good guesses of the behavior of PL2 because they recognize similarities in the syntax. The degree of similarity plays a role in transfer, e.g., the more similar the syntax is between PL1 and PL2, the easier it is for semantic transfer to occur. Educators can use this implicit transfer to help them effectively teach second programming languages.

- FCC is the most challenging concept to learn in PL2 in all the four studies presented in this chapter. The reason for this challenge could be because semantic transfer affects learning negatively. The students transfer semantic knowledge between PL1 and PL2 due to syntax similarities, but it becomes a problem when the PL2 semantics work differently from PL2. Semantic transfer can occur from PL1 to PL2 and bidirectionally from PL2 to PL1. The findings confirm the MPLT where it shows that relative novices hold one mental representation of the same concept in two languages that behaves differently in each. The negative semantic transfer starts when students guess PL2 before learning it and persist even after they have learned PL2. In teaching PL2, the educators can point out the differences between PL1 and PL2 and take this as an opportunity to teach students a

deeper understanding of programming concepts, and this is explored in Chapter 9.

- Lastly, the results revealed minimal/no semantic transfer on ATCC concepts between PL1 and PL2. It has been shown that this is more challenging for students transitioning from procedural Python to object-oriented Java (Study 2a, 2b, and 2c), as also confirmed by other studies [9, 11], than for students transitioning from object-oriented Java to Python (Study 2d). It could be because object-oriented languages like Java can already be a challenge for students as compare to Python language because of their abstract representation. Other challenges in abstract conceptual learning has been confirmed in psychology research by Gentner [87] and natural language research [28]. For ATCC concepts, second language educators can point out the corresponding concept representations in PL1 and map them to PL2.

These findings show that the model can be used to guide teachers of PL2. The significance of the findings also is that students may have fragile knowledge of PL1 semantics and concepts before they even start transferring to PL2; therefore, instructors can use PL2 learning as an opportunity to deepen conceptual understanding. Lastly, educational programming language designers need to consider the role of language similarities in learning.

Chapter 7

Teachers' Experiences on Transfer

[*Aspects of this study have appeared in [121].*]

The claim of this thesis is twofold, as reflected in the thesis statement. The first claim is that *semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages*. The first preliminary study of this thesis, presented in Chapter 4, confirmed this claim qualitatively and revealed that indeed semantic transfer did take place based on syntax similarities when relative-novice programmers learn a second programming language. In order to investigate the semantic transfer notion further and quantitatively, a model (MPLT) suitable for programming language transfer based on the first preliminary study findings was developed in Chapter 5. The model was validated in Chapter 6 with four experiments investigating relative-novice programmers transferring between procedural Python and object-oriented Java. The results highlighted that semantic transfer automatically occurs between PL1 and PL2 based on syntax similarities, which can result in positive transfer (TCC), negative transfer (FCC), or no transfer (ATCC). The findings revealed other problematic issues such as that students have fragile knowledge of PL1 and also that as a result of the fragile knowledge, the bi-directional transfer can occur between PL1 and PL2.

Given these challenges of PL transfer, the second part of the thesis statement proposes that *the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.* In order to explore this claim, a preliminary step is to explore PL transfer from the school teacher's perspective at the secondary school level. There are three main reasons for making teachers at the secondary school level the starting point of this investigation. Firstly, all the previous experiments in this thesis explored PL transfer from the context of university students. Therefore, further exploring the transfer phenomenon from the teacher's perspective could bring deeper insights into the transfer process as both students and teachers are the key players in the process of learning. Such research can help bring the student learning experiences and teacher teaching experiences closer together. Secondly, the previous

experiments focused on transfer at the university level; therefore, the school level research may help confirm if students also experience the reported transfer challenges proposed by the MPLT at the school level. Lastly, exploring the teachers' perspective of transfer strategies can help identify specific needs for teacher education when transitioning students to new programming languages. Specifically, the findings from the teachers can highlight the ways the MPLT can be used to improve transfer by investigating and implementing teaching strategies aligned with the Model's predictions (TCC, FCC, and ATCC) that will enhance second language learning outcomes.

Therefore, this chapter presents a study that investigates high-school teachers' current practices and attitudes towards second language learning and the teachers' awareness of the transfer issues from their classroom experiences. This chapter begins with presenting the research questions, followed by a section that describes the participants. The next section describes the interview protocol used to guide the researcher in conducting the interviews with teachers. The sections that follow describe the data collection and analysis process. These include the descriptions of the procedures of collecting, measuring, and analyzing the data. The rest of the chapter describes the results categorized by each research question. The last section of the chapter concludes with a discussion.

7.1 Research Questions

The purpose of this chapter is to answer the high-level RQ3c: *How do school teachers typically approach PL transfer for relative novices in the classroom?* In order to answer this overarching research question, the following specific research questions are formulated. They will serve as a systematic guide for the data collection, analysis, and discussion of the findings in this chapter.

It is important to gather data to help shape potential transfer teaching strategies on second programming language learning as a starting point. The transfer strategy a teacher adopts to teaching a second language in the classroom will be influenced by their belief in the value and purpose of teaching that second language, which therefore leads to the first research question:

- (RQ3a) Why do school teachers teach a second programming language?

It is also important to find out whether school students experience the same transfer challenges proposed by the MPLT when learning their second programming language as do university students. Hence the second research question for this chapter:

- (RQ3b) Do school teachers notice any transfer problems/benefits when teaching a second or subsequent PLs?

Finally, it is crucial to investigate the current transfer intervention methods that teachers use to teach for transfer and determine if these interventions cater to the positive, negative, and lack

of transfer issues associated with the transfer as proposed by the MPLT. If the findings show the teachers' uncertainty about transfer intervention, it will be a motivation for this research to design transfer intervention programs aligned with the MPLT that can lead to improved conceptual transfer and understanding in a second PL learning. Hence the final research questions:

- (RQ3c) What are the views of computing school teachers on using transfer strategies?
- (RQ3d) What types of transfer strategies do computing school teachers use?

7.2 Participants and Context

As already mentioned in the introduction of this chapter, the research presented here aims to explore transfer between programming languages from the teacher's perspective. In order to achieve this, computing teachers at the school level (K12) were suitable for this research because, unlike university teachers, school-level teachers are frequently interacting with students and engaged with their programming exercises [122] hence they can be able to identify transfer issues more effectively. Furthermore, university teaching has a highly individual nature [123], while at the school level, the curriculum is shared, which can help in making the teachers' observations more homogeneous. Purposive sampling [90, 91] was used to select the teachers. This is because to understand their experiences and views about the transfer, it was best to select teachers who teach second or subsequent programming languages at the secondary school level (K12). Twenty-three K12 computing in-service teachers from different schools in two European countries (Scotland and The Netherlands) participated in the study. The reason for choosing teachers from different schools and countries is to give better-quality evidence, and greater validity to the findings [124]. Participation in the study was voluntary, and the teachers were assured of confidentiality and anonymity of responses before agreeing to participate.

The teachers were recruited in different ways. Teachers who are usually interested in CS research in Scotland were contacted through emails. In the Netherlands, teachers were contacted through an advertisement in the weekly newsletter of i&i[1]: an association of Computer Science teachers in primary and secondary schools in the Netherlands. In addition, social media platform (Twitter) was used to contact the teachers.

A summary of the teachers' demographics and teaching-related information is presented in Table 7.1. All the teachers teach at public secondary schools; however, two teachers also teach in the final year of primary school. Their Computing teaching experience ranges from 5 years to 35 years, and they teach students with ages ranging from 11 years to 18 years. There are differences in the Scottish and The Netherlands Computing curricula. In the Netherlands, there is no national curriculum. The existing curriculum is more flexible, and the responsibility of the content and delivery method lies with the teacher. However, teachers are given guidelines

[1] "https://ieni.org/"

on what to teach in digital literacy and Computer Science subjects. These guidelines are split into different domains. Students do not have mandatory exams when following these domains throughout their school years. In Scotland, teachers follow a national CS curriculum where students in the first two years of secondary school (S1 and S2) usually have a 50-minute (more or less) Computing subject lesson per week. At this stage, there is no mandatory exam. The CS subjects become optional in the third year of secondary school (S3). Students who pursue the CS subjects at this stage are required to take between two to four 50-minute lessons per week in addition to getting mandatory examinations.

The teachers' names are presented here as research IDs: participants T1-T23 to ensure anonymity.

7.3 Interview Protocol

In order to explore the teachers' views and experiences in teaching a second or subsequent programming language, interviews were chosen as a means of inquiry. Interviews are primarily done in qualitative research and occur when researchers ask one or more participants general questions and record their answers. Structured interviews involve prepared sheets with a set of prepared closed-ended questions resulting in responses that are easy to analyze [125]. Unstructured interviews do not use any set of predefined questions. The interviewer asks open-ended questions based on a research phenomenon under study and uses prompts or probes that remind the interviewee about topics to discuss [126]. The unstructured interviews can produce deep insights by providing valuable data regarding the phenomenon under study [125]. In order to investigate teachers' views and experiences on PL transfer, semi-structured interviews were used. The reason for choosing semi-structured interviews is because it is a combination of both structured and unstructured interviewing, and it offers both of their advantages. The semi-structured interviews consisted of initial scripted questions, which were supplemented by follow-up questions, probes, and comments where needed.

A 'prompt sheet' was developed for the semi-structured interviews based on the research questions; see Appendix C. The sheet was developed to guide and initiate the conversations with the teachers. RQ3a was designed to get the teachers' motivations to teach a second programming language; therefore, the prompting question for this discussion was *"Why do you teach a second and subsequent programming language?"* As an example of a follow-up to this question, teachers were asked to justify their choice of programming languages and the learning outcomes they were pursuing when teaching them. These questions allowed for the teachers to discuss and think deeply about their motivations for multiple languages in the classrooms. At this point, the questions do not draw attention to any particular aspect of the transfer. These questions give insights into understanding what the teachers value in teaching multiple programming languages. As a result, their answers provide insights into their choice of a transfer strategy.

Table 7.1: Details over the participating teachers. Teacher's code reflects the order of their interview, where T1 is the first interviewed and T23 is the last interviewed.

Code	Gender	Country	Teaching experience	Programming languages taught
T1	Male	Scotland	21 years	Scratch, Blocky and micro:bit block editor,HTML, Python
T2	Male	Scotland	10 years	Scratch, Scratch and micro:bit block editor,HTML, Python, PHP
T3	Male	Scotland	18 years	Scratch, HTML, Python, PHP
T4	Female	Netherlands	5 years	Python, HTML/CSS JavaScript, C#
T5	Male	Netherlands	5 years	HTML, Scratch, Javascript
T6	Male	Netherlands	5 years	Processing (Java), Arduino, Web (PHP), Python
T7	Male	Netherlands	11 years	HTML, Python, Java
T8	Female	Scotland	15 years	Blocky, Scratch and HTML, Python and SQL, Python, HTML/CSS Javascript, MySQL and PHP
T9	Female	Scotland	30 years	Blocky, Scratch, Python
T10	Male	Scotland	35 years	Blocky, Scratch, Java, HTML
T11	Male	Scotland	16 years	Blocky, Scratch ,Javascript and VB, Javascript
T12	Female	Scotland	19 years	Scratch, micro:bit block editor, Turtle Python, Python
T13	Male	Netherlands	35 years	HTML, JavaScript, PHP, Python
T14	Male	Scotland	24 years	Scratch(S1), HTML/CSS SQL, VB, Javascript, PHP MySQL
T15	Male	Scotland	15 years	Scratch, Python
T16	Male	Scotland	23 years	Truebasic, Scratch,VB, Python, Java
T17	Female	Netherlands	14 years	HTML/CSS, Flowcharts Flowgarithm, Python
T18	Male	Netherlands	3 years	Python, Database SQL, JavaScript
T19	Male	Netherlands	14 years	Scratch, Python
T20	Male	Netherlands	5 years	HTML/CSS, JavaScript, AJAX and JSON, PHP MySQL, Python
T21	Male	Netherlands	32 years	Python, PHP
T22	Female	Scotland	23 years	Micro:bit block editor, Scratch, Python, Javascript SQL HTML/CSS, PHP
T23	Male	Scotland	15 years	Scratch, Python

To answer RQ3b, teachers were asked the leading question, *"Do you experience any transfer benefits or problems in the classroom when teaching second/subsequent programming languages?"*. Teachers were then asked follow-up questions to get in-depth knowledge about their experiences, e.g., which caused problems. These questions were asked to find out whether school students experience the same transfer challenges proposed by the MPLT and prior research on PL transfer.

RQ3c was designed to get the teachers' views on transfer strategies. Teachers were asked about the perceived impact of previously taught languages on students learning the second/-subsequent programming language and if they thought it was beneficial to implement transfer strategies. Finally, to answer RQ3d on teachers' transfer strategies, teachers were asked the following leading question, *"What measures or techniques do you currently use to help students transfer knowledge from first programming language to second programming language (if any)? "*. Some follow-up questions included asking them specifically how they referred to the first or previous programming language(s) during second language instruction. The answers to these questions would help inform this transfer research. For example, if the teachers use transfer strategies, that would give insights into how they can be adopted. However, if the teachers are not using transfer strategies, that would be a motivation for this research to develop a guide for teachers of transfer strategies based on the MPLT.

7.4 Data Collection

As mentioned in the previous section, semi-structured interviews were used to collect open-ended data on teachers' thoughts and beliefs about the transfer and teaching of second and subsequent programming languages. Participants for this study were from Scotland and The Netherlands, as mentioned in the Participants' section. Therefore, the data collection involved five researchers, two from Scotland and three from The Netherlands. Each researcher followed consistent steps in conducting interviews as advised in the literature in conducting interviews during qualitative research [75, 127].

The interviews were conducted through the Universities (Scotland/Netherlands) video platform, Zoom. They began by introducing the research to the teachers. It involved providing a clear summary of the purpose of conducting the research on programming language transfer. At this point, the teachers were requested to give consent to record the videos and assured of confidentiality and anonymity in any published reports. The second step involved asking a more general question about their demographics and experiences in teaching. The next step was to guide the teachers by asking them the research questions, as outlined in the previous section. After each question, there was a set of follow-up questions and probes to gain in-depth knowledge into their experiences and views. Towards the end of the interview, the interviewer alerted the teacher in order to wrap up the discussion. Each teacher interview was an average of 50

minutes. All interviews were recorded, transcribed, and then coded.

7.5 Data Analysis

Thematic analysis [77] was used to analyze and interpret themes within the data. Thematic analysis, as described by Braun and Clarke [128], is a method for systematically identifying, organizing, and offering insight into patterns of meaning (themes) across a data set. This method was chosen because it offers high levels of flexibility and simplicity in analyzing qualitative data [129]. Guidelines of analysis of this data were adopted from Braun and Clarke [128]. These guidelines include:

1. **Familiarizing Yourself With the Data:** All the interviews were stored in a password-protected shareable drive accessible to all the researchers. Each researcher was allocated one of the four research questions to code. At this stage, the researchers familiarised themselves with the data. It involved listening and re-listening to the interviews, transcribing them, and writing down notes relevant to the allocated research question.

2. **Generating Initial Codes:** The transcribed data was imported into Excel for detailed coding analysis. Each researcher coded at least one of the four research questions. The researcher then discussed the initial codes with one other researcher, and conflicts were discussed and resolved. This stage ended when the data was fully coded.

3. **Searching for Themes:** At this stage, themes were generated from the coded data. It involved reviewing the coded data to identify areas of similarity and overlap between codes. Small themes that shared unifying features were merged, and larger themes were collapsed using an iterative process.

4. **Reviewing Potential Themes:**

   This phase involved a recursive process whereby the developing themes were reviewed in relation to the coded data and the entire data set. In this process, themes that described the core issues of each research question were identified. The aim was to have at least three major themes per research question, potentially with sub-themes. The discussions between the first and second coders were about the precise grouping of themes and sub-themes.

5. **Defining and Naming Themes:** Finally, all the researchers discussed the themes. It was important to confirm if the themes have a clear focus and purpose. It was also necessary to ensure that the themes aligned with each research question. Lastly, it was essential to check if the themes provided a coherent overall story about the data.

Table 7.2: Reasons teachers provide for teaching multiple languages

Category	Number of teachers
Start with a simple language	13
- Simple language provides engagement	8
- Simple language avoids errors	5
- Simple language builds confidence	4
- Simple language supports focus on concepts	3
Different languages have different benefits	10
- Knowing different languages gives a different perspective	6
- Different languages have different applications	4
Part of the curriculum	7

6. **Producing the Report:** The purpose of this stage was to write a coherent report about the identified themes based on the analysis. Details of this report will be presented in the next section (Results).

7.6 Results

Findings from the interview data relating to the 23 teachers' views and experiences teaching second and subsequent PLs are presented in this section. The identified themes will be presented in four subsections corresponding to each research question. These subsections are shown in the following order; reasons for teaching multiple languages, problems/benefits of teaching multiple languages, teachers' views on transfer strategies, and finally, types of transfer strategies teachers use.

7.6.1 RQ3a: Reasons for Multiple Languages

The findings presented in this subsection are the reasons high-school teachers provide for teaching multiple languages. Table 7.2 shows three main themes derived from the interview data; starting with a simple language (13 teachers), different languages have different impacts on students (10 teachers), and languages are part of the curriculum (7 teachers).

Start with simple language

The biggest motivation for teaching multiple languages given by the teachers is that at the early stages of learning CS, teachers want to start with a simple language (such as block-based language) and then progress to a more formal/text-based language (such as Python). According to the teachers, a simple language means a visual or block-based programming language that lets students create programs by manipulating program elements graphically using drag and drop, e.g., Scratch. Teachers expressed that simple languages may have limitations, hence the need for a second or third language. The following four sub-themes were derived from this overarching theme:

Simple language provides engagement Eight teachers expressed that starting with a simple language helps students to engage with programming quickly. For example, T8 said that *"Scratch is easier because it uses blocks and students can throw in something quickly"* and T1 states that *"We start with a simple language, so we do not scare them."*

Simple language avoids errors Helping students avoid syntax errors is another motivation teachers have for starting with simple languages. For example, teacher T9 states *"We start with Scratch to eliminate syntax errors".*

Simple language builds confidence Four teachers revealed that a simple language increases students' self-confidence in programming. For example, teacher T16 explained that they *"start with blocks to avoid frustration"*. This reason seems to be justified because, as presented in Section 2, programming language syntax can be a barrier in learning to program for most relative-novice students.

Simple language supports focus on concepts Finally, a few teachers (3) believe that simpler languages allow students to focus on programming concepts rather than on syntactic issues. For example, T17 states *"We use Executable flowcharts because they help to explain the three basic concepts: sequence, selection, and iteration. We then "dress it up" with a textual programming language".*

Different languages have different benefits

The second main theme derived from the interview data given by ten teachers is that students benefit from being taught different languages. For example, T20 explained that they teach different languages to achieve different goals e.g. starting with HTML/CSS to show students code is everywhere. Two sub-themes were derived from this main category as follows:

Knowing different languages gives a different perspective Some teachers (6) under this category believed that students' knowledge of multiple languages can give them a better perspective and view on understanding different programming languages. For example, T21 believed that students can learn a new language quicker if they have been exposed to other languages.

Different languages have different applications Other teachers (4) believed that different programming languages have different applications that can show students the broadness of various languages. For example, T6 said *"Different programming languages help to teach different parts of programming. For example, C++ is difficult but helps to teach hardware".*

Table 7.3: Benefits and problems highlighted by teachers of teaching multiple languages

Benefit Category	Number of teachers
Positive effect on understanding programming concepts in PL2	15
- Transfer the understanding of basic programming concepts from one language to another	11
- Students make the connection/link to the previous language	7
- Happens only under specific conditions	2
Positive effect on cognitive abilities	7
- Transfer of problem solving skills and abstract thinking	4
- Broadening students perspectives	3
Problem Category	
Negative effect of semantic transfer	10
Negative effect of syntax transfer	8
- Context switching is difficult and takes time	5
- Mixing syntactic elements from two languages	5
Students make no connection to/with previous language	4
Students demotivated or less confident to learn a new language	3
- Students making generalizations on lack of ability based on a bad previous experience	3
- Stuck in one language that they know	1

Part of the curriculum or materials

Lastly, as can be seen in the responses, seven teachers revealed that they teach multiple programming languages because it is a requirement from the curriculum.

7.6.2 RQ3b: Problems/benefits of Teaching Multiple Languages

The findings presented in this subsection are the problems and benefits of teaching second and subsequent languages. The data was coded and categorised by what the teachers explained regarding their experiences. They shared experiences that varied greatly from both the benefits and problems of teaching multiple languages. For example, one teacher may experience both benefits and problems of transfer. These were grouped into common themes that came from all the teachers' experiences and are presented in Table 7.3. Table 7.3 is divided into two main categories of benefits and problems. In the category of benefits, there are two main themes; positive effects on understanding programming concepts in PL2 (15 teachers), and positive effects on cognitive abilities (7 teachers). In the category of the problems, there are four main themes, namely; negative effect of semantic transfer (10 teachers), the negative effect of syntax transfer (8 teachers), students make no connection to/with previous language (4 teachers), and students demotivated or less confident to learn a new language (3 teachers).

Benefits observed when moving to another programming language

The benefits that teachers observe when teaching second or subsequent PLs are the advantages that students have of knowing more than one language.

Positive effect on understanding programming concepts in PL2 The biggest benefit of having knowledge of other languages, highlighted by 15 different teachers is that it helps students understand programming concepts in PL2. This overarching theme is divided into three subthemes. Firstly, teachers report that students transfer the understanding of basic programming concepts between programming languages. For example, T9 highlights that students *"automatically transfer the concept of the if-statements and repeat statements from Scratch"* while T4 said that *"students understand how they can use if-statements in VB because they have learned that in Scratch, it happens quite naturally"*.

Secondly, teachers believe that students make connections to previous languages, and that helps them understand PL2. For example, T4 highlights that sometimes students say *"you would do this in Python like this or that, but in C# I do it differently now"*. Some teachers said that by students making connections to prior languages, they learn PL2 more quickly, as T6 said it: *"concept repetition, [and then] you have the basis of learning another language quickly"*. Lastly, a few teachers believe that for any benefit to carry over to PL2, there should be good instructions, peer discussion among the pupils, and focus on code comprehension questions, as highlighted by T15.

Positive effect on cognitive abilities Teachers report that knowledge of multiple languages helps students to develop reasoning or thinking skills in programming. First, teachers believe that students' knowledge of PL1 helps them transfer the problem-solving skills and higher-level thinking from PL1 to PL2. T7 expressed that *"students learn that when they have a problem they need to split it into parts and then they can solve those small parts one by one. By doing multiple programming languages students would see that the way of thinking remains the same"*. T19 also said that *"Learning multiple languages helps more in the thinking process, higher-level thinking over what programming languages do"*. Second, some teachers believe that learning multiple languages broadens students' perspectives on programming languages and their applications. For example, according to T20 *"Knowing multiple languages allows the student to do different things"*, and *"students then know that a specific programming language is not sacred"*, according to T18.

Problems observed when moving to another programming language

Teachers reported that while students benefit from the knowledge of other languages, there are problems that come with transfer from PL1 to PL2.

Negative effect of semantic transfer This main theme is concerned with the negative semantic transfer from a previous language to a new programming language. For example, T17 says *"with Structurizr you had to put an `'and'` instead of a `'+'` in Python to concatenate strings"*. Another example is when students transition between strictly and less strictly typed

programming languages. T10 explained that when students switch to Java from Python, parameter passing is a challenge because Python does not explicitly ask you to declare your data types while Java does.

This negative transfer can also be from previous Mathematics knowledge. For example, T1 expressed that *"the = symbol is a problem, they transfer the math equal"*.

Negative effect of syntax transfer Eight teachers reported syntax transfer challenges when students learn second/subsequent languages. Two sub-themes emerged from this main theme. First, students' context switching between programming languages is a challenge as context switching can take time. T1 believes that students struggle with syntax more than concepts while T2 says that a new language throws a barrier with syntax. The second sub-theme is concerned with the problems that arise from students mixing syntactic elements between the two languages. For example, T21 mentions that *" students write PHP sometimes in the python style: colon instead of, or mixed with, parenthesis for loops"*.

Students make no connection to previous language Four teachers reported that transfer is often unsuccessful as students fail to transfer their understanding of the same concept from their previous language to a new language. It is when students transition from a block-based programming environment to other programming languages. For example, T8 explained that: " I think they have transfer problems from Scratch, I think it is because they are given a block in Scratch and they do not think what's inside the blocks so they just pull them in, they do not notice it's a repeat block which is the same as a for-loop in Python".

Students demotivated or less confident to learn a new language The last main theme is that students' motivation to learn a new programming language is negatively affected (three teachers). The students' negative first encounter with programming using the first programming language affects their motivation to learn a new language. For example, T23 said that *"When pupils go onto a second language but are not confident in the concepts and structures that appear in both, it can make them less confident"*. One teacher (T19) reported that, *"Sometimes students do not want to switch languages and remain stuck in their first language. It usually happens when students feel their knowledge of the first language is better hence they may look down on the second language"*.

7.6.3 RQ3c: Views on the use of Transfer Strategies

Three main themes emerged from the answers that teachers gave on their views on the use of transfer strategies. Table 7.4 presents a summary of these views.

Table 7.4: Views on transfer strategies

Category	Number of teachers
Do not believe they have to implement transfer strategies	12
-There is not enough time to implement transfer strategies	4
-Mapping prior language to new language is difficult	2
-Assume students transfer implicitly	2
-Implementing transfer strategies confuses students	2
-Implementing transfer strategies puts students off	2
Believe that knowledge of more than one language helps in transfer	9
Do not think students have strong comprehension of the first language for any benefits to carry over to a new language	4

Do not believe they have to implement transfer strategies

The majority of the teachers (12) do not believe that implementing transfer strategies in the classroom is necessary. Their reasons are divided into five sub-themes follows:

There is not enough time to implement transfer strategies Four teachers believe that they do not have enough time to implement transfer strategies in the classroom. For example, one teacher said that *"It takes an awful lot of time to teach for transfer and use it as an opportunity for learning. It takes time to unpack the concepts students are learning but there is no time"* (T10).

Implementing transfer strategies confuses students Two teachers believed that transfer strategies will confuse students because it might become complicated when students learn new languages. For example, *If I introduce the deeper semantics of a language, it's too early, I lose too many, we don't want it to get overly complicated* (T2).

Mapping prior language to new language is difficult Some teachers expressed that they face challenges when trying to map prior languages and new languages. For example, T1 expressed that, *"Its difficult to convert a game environment from Scratch to Python"*.

Assume students transfer implicitly therefore no need for transfer strategies Two teachers do not believe that students need assistance to transfer knowledge. These teachers believe students automatically transfer knowledge from one programming language to the next. For example, one teacher said *"I believe transfer happens quite naturally for students from Scratch to VB"* (T11).

Implementing transfer strategies puts students off Lastly, two teachers believe that transfer strategies might demotivate students to learn a new language. For example, T8 expressed that,

Table 7.5: Types of transfer strategies

Category	Number of teachers
Referring to PL1	
- Explicitly referring to PL1	6
- Comparing code in different programming languages and discussing similarities	5
Favoring transition	
- Putting the emphasis on concepts	8
- Using visual representations	4
- Using dedicated activities	3
Preparing for transfer	
- Referring to everyday life	3

"We don't teach for transfer, we don't go that far. I am always worried about putting them off".

Believe transfer strategies are important and knowledge of more than one language helps foster transfer

The second main theme presented by nine teachers is that when students have knowledge of another programming language it can help them with the ability to transfer conceptual knowledge from one language to the other. For example, one teacher said, *There are analogies between languages, but students do not see that for themselves, someone needs to tell them. With more concepts and analogies, students can learn a new language faster. Students become more resilient to new materials/products because they have seen something similar, which helps more in the thinking process, higher-level thinking over what programming languages do* (T19).

Do not think students have strong comprehension of the first language for any benefits of conceptual knowledge to carry over to a new language

Lastly, four teachers believe that students have fragile conceptual knowledge in the first language and, as a result, they do not have conceptual knowledge to transfer to a new language. These teachers expressed that it is important to teach students programming language concepts more effectively in their first language. T23 expressed that, *I think the biggest risk is not teaching the first language in a comprehensive way, especially Scratch when it is sometimes seen as an "introduction" rather than a learning tool*. T3 also has a similar view and states that, *"We then don't teach the constructs of the first language, but rather the game environment".*

7.6.4 RQ3d: Types of Transfer Strategies

Table 7.5 presents a summary of the transfer strategies teachers use when teaching a second or subsequent languages. Only nine teachers out of the 23 interviewed teachers intentionally use transfer strategies.

Referring to the first programming language

The first main theme of transfer strategies that teachers use is referring to the first programming language when teaching the second one. This theme contains the two sub-themes:

Explicitly referring to the first programming language Six teachers refer to the previous programming language when teaching a second or subsequent language. For example, one teacher expressed that *I just repeat the concept briefly, but not too deep in the new language and indicate what differences and similarities are with the previous language* (T20).

Comparing code in different programming languages and discussing similarities Five teachers show different versions of one concept in the previous language and the new language. Teachers do this by presenting both languages at the same time and doing a translation process from one language to the other (e.g. block to text, text to text, flowcharts to text). One teacher said: *"For most concepts I introduce through the course I map them to the Scratch ones to show how they work in both languages"* (T13). T15 also expressed that *"I show a JavaScript program on the whiteboard and together with students and transform it to PHP"*. Another teacher said to ask students questions such as *"How would you do it in C#/Unity?"*

Favoring transition

This theme consists of three sub-themes of instructional strategies that promote the transition from a first programming language to a second one.

Putting the emphasis on concepts Most teachers (8) favoring transition strategies put the emphasis on concepts when teaching a second or subsequent programming language. For example, naming and abstracting concepts when introducing a new programming language to remind students of these concepts in the first language. One teacher said *"We teach concepts that we have taught before, but we flag what they looked like in the first language when we introduce them in the second language"* (T22).

Using visual representations Four teachers use visual representation by starting with using visuals like flowcharts to abstract the concept and make it more generic even when it is presented in the second programming language. For example, T19 said: *"I teach them the logic flow in flowcharts first and when I move to Python I keep showing the flowcharts in one charts".*

Using dedicated activities Lastly, three teachers use dedicated transfer activities such as allowing students to solve an assignment in the language they prefer. Some teachers give similar assignments in PL1 and PL2 so that students can not struggle with understanding the problem.

This gives the students a chance to only focus on the new syntax of a new language. Some teachers use programming language development environments that allow the students to switch from blocks to text and back (T11). Some teachers allow students to use both languages at the same time and be aware of differences and similarities between the two languages.

Preparing for transfer

Three teachers prepare for transfer by referring to everyday life examples such as asking students to look at websites they are familiar to familiarise themselves with code. One teacher compared objects properties in an object-oriented class to the fields of an element in the software student know (T13).

7.7 Discussion

The previous section presents the results on the views and experiences of 23 secondary school Computing teachers on teaching second and subsequent programming languages in the classroom. This section presents the discussions of the results as reflections of the research questions.

7.7.1 RQ3a: Reasons for Multiple Languages

The transfer strategy a teacher adopts is likely to be influenced by their belief in the value and purpose of teaching that second language. The results reveal that some teachers teach second and subsequent programming languages because they are required to do so by the curriculum. The teachers who believe there is value in second language learning mostly associate this benefit with increasing students' engagement and excitement in programming. These teachers start with teaching students a simple language in the early stages of learning programming before they transition them to the second programming language that is more formal or text-based. These are reasonable points considering the syntax challenges relative-novices face when learning new languages [4, 130]. These kinds of teachers usually teach students how to explore programming through games without emphasizing giving students conceptual knowledge.

7.7.2 RQ3b: Problems/benefits of Teaching Multiple Languages

Teachers revealed that school-level students experience transfer problems just like university students. Most of these challenges have been reported in prior programming language transfer research and the MPLT. Teachers reported that students experience positive effects of understanding PL2 concepts (TCC), negative effects of semantic transfer (FCC), and fail to connect common concepts between PL1 and PL2 (ATCC). The failure to connect common concepts usually occurs when they learn abstract concepts in a new language [29, 34, 132]. Other issues such

as syntax problems have been reported in prior work, which investigated near novices problem-solving in new languages [16, 17, 19]. Other positive issues such as transfer of problem-solving skills have been reported in [19, 20, 22], and they are also attributed to language similarities.

The persistent problems of transfer as reported in schools and universities give motivation for this research to design transfer intervention strategies for teaching second programming languages.

7.7.3 RQ3c: Views on the use of Transfer Strategies

The value in second language learning expressed by teachers in answering RQ3a has shaped their views on transfer strategies in second language learning. Because most teachers do not see the value of conceptual development of second language learning, most teachers also do not believe that implementing transfer strategies in the classroom is helpful in second language learning.

The problems/benefits of teaching multiple languages (RQ3b) results and the teachers' views of transfer strategies (RQ3c) results were compared. Nine teachers who do not believe they have to implement transfer strategies also reported their students experiencing both benefits and challenges in transferring to another language in the classroom. However, their most concern with implementing transfer strategies was that there was not enough time to do such interventions. Also, these teachers reported the challenges they perceived might occur when teaching for transfer which included their concern in putting off the students' interest in learning a new language as well as believing that students have fragile knowledge in PL1, which cannot be transferred to PL2. However, the experiments that validate the MPLT, presented in the previous chapter, have revealed that in most cases, even with fragile knowledge, students automatically transfer semantic and conceptual knowledge from PL1 to PL2 based on syntax similarities to most basic PL concepts. If teachers do not implement transfer strategies, it may impact the learning of the second language negatively, as shown in RQ3b results and as confirmed in prior work [10, 33].

Some teachers (39%), however, believe in the value of transfer strategies and their potential to deepen conceptual understanding. As a result, these teachers try to implement transfer strategies in the classroom as shall be presented next (RQ3d).

7.7.4 RQ3d: Types of Transfer Strategies

Teachers mostly adopt transfer strategies that refer back to the first programming language by showing different versions of the same code in the PL1 and PL2 or giving the same assignment in two different languages. These strategies of bridging have been reported to be beneficial in prior work [10, 17, 45, 178, 205, 206]. Although these are commendable efforts, only 39% of the teachers use transfer strategies, and some of these transfer strategies seem rushed and informally implemented. For example, teachers say *I repeat the concept briefly, but not too deeply* and *I flag*

concepts. Furthermore, most of the transfer strategies they use do not cater to the different types of transfer reported in the MPLT (TCC, FCC, and ATCC). These findings give a motivation for this research to design transfer intervention programs aligned with MPLT that can guide teachers and also lead to improved conceptual transfer and understanding in a second PL learning.

7.8 Summary of Discussion

The summary of the research findings of the 23 in-service teachers include that:

- Most teachers start teaching programming using a simple programming language, therefore, creating an opportunity for transfer and deepening of conceptual knowledge once learners encounter their second language.
- Teachers report both benefits and problems of transfer from PL1 to PL2. Furthermore, school students experience the same transfer problems as university students.
- Some teachers do not believe they have to implement transfer strategies hence they do not use second language learning as an opportunity to help the students understand concepts better.
- Some teachers adopt transfer strategies that are known in computing literature. This willingness can be used as an opportunity for researchers to work with these teachers to develop, implement and evaluate a pedagogy where transfer interventions tackle specific student transfer problems (e.g. concepts, syntax, and semantics).

These study findings open up opportunities for further research to be conducted in this thesis to explore transfer interventions aligned with how students transfer knowledge as proposed by the MPLT. These interventions will be presented in the next chapters.

Chapter 8

Exploring Explicit Interventions on Transfer

[*Aspects of this study have appeared in [33].*]

Chapter 7 highlighted that teaching students multiple programming languages is a common practice in schools as it is in universities. The common rationale for teaching multiple languages given by teachers is to engage students by starting with simple programming languages at the early stages of learning programming and then transitioning the students to more formal programming languages. The teachers highlighted that the transition process is not as straightforward. Students sometimes face both positive and negative transfer effects between the programming languages as hypothesized by the MPLT. Despite recognizing the lack of transfer (ATCC) or the negative effects (FCC), most teachers do not implement transfer strategies in their classrooms when teaching a second or subsequent programming language. The few teachers who adopted transfer interventions relied on their practical knowledge shaped by experience rather than drawing on a theory/model of how students transfer knowledge from one programming language to the other.

The MPLT presented in Chapter 5 was designed to explain how students transfer semantic and conceptual knowledge between programming languages, therefore, can assist teachers to better account for the failure or success of transfer in the classroom. Chapter 6, which validated the MPLT, highlighted that before and after learning PL2, learners automatically transfer semantics from PL1 to PL2 based on similarities between PL1 and PL2. The negative effects (FCC) or the lack of transfer (ATCC) can remain persistent if the teacher does not implement explicit strategies of transfer interventions. These empirical studies also provided evidence that some students have fragile knowledge of both PL1 and PL2. They hold only one mental representation of constructs that look similar in both programming languages even though they have different semantics.

The second part of the thesis statement proposes that *the implementation of deliberate se-*

mantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language. The initial exploratory transfer interventions implemented on students transitioning from procedural Python to object-oriented Java presented in Study 1, in Chapter 4 seemed to give the initial support to this claim. However, they were implemented on a small sample size of just five students. Therefore, this chapter focuses on investigating this claim quantitatively by designing transfer interventions that draw on the MPLT of how students transfer knowledge between programming languages. The transfer interventions in this chapter are based on the MPLT because it provides a basis to understand, make predictions and analyze how students transfer. As a starting point, this chapter focuses on exploring transfer interventions that are based on the TCC and FCC concepts only. This is because the experiment was conducted in a real classroom which meant that there was restricted time to implement transfer strategies aligned with all the MPLT construct categories. Given this reason, the FCCs were chosen as a starting point because the previous chapters revealed that they are the most problematic constructs that affect learning of PL2. The TCCs were selected as they were reported to be the least troublesome; therefore, implementing their transfer strategy would not consume time.

The transfer interventions adopted in this study were student-focused. The student-centered approach was chosen because it involves active learning that allows students to reflect in their learning [52, 137] as well as work collaboratively to facilitate their learning [137]. Furthermore, the transfer interventions were based on code comprehension activities for the following reasons. Firstly, they were chosen because the MPLT was developed based on code comprehension. Secondly, they allow students to focus on the programming language concepts and semantics and develop viable mental models of program execution, before going ahead and attempting to use the concepts in problem solving.

The chapter begins with the research questions followed by the participants' details. The section that follows describes the instrument used and the rational for using it. The data collection is presented next. This section includes the process of gathering and measuring data on transfer interventions, that helps to answer the research questions. The rest of the chapter describes the data analysis and results of the experiment. The chapter concludes with a discussion.

8.1 Research Questions

The previous study presented in Chapter 7, gave insights into the transfer challenges teachers face in the classroom and also revealed that despite these challenges, the majority of teachers interviewed do not implement transfer strategies in the classroom. In order to build transfer interventions that can guide teachers and can help improve second language learning in the classroom, this thesis poses the following over-arching RQ4: *How can transfer teaching interventions based on our understanding of semantic transfer improve second PL learning?* In order

to answer this overarching research question, the following specific research questions are formulated. They will serve as a systematic guide for the data collection, analysis, and discussion of the findings in this chapter.

The empirical studies that validated the MPLT revealed that the TCC constructs were the easiest to learn when transiting from PL1 to PL2 due to positive semantic transfer based on the syntax and semantics similarity between PL1 and PL2. Therefore it is essential to investigate how teachers can use transfer interventions to use this implicit transfer to help them effectively teach second programming languages. To investigate this, the following RQ is asked:

- (RQ4a): Can students' learning of True Carryover Concepts (TCC) be improved by student-centred transfer interventions?

The previous studies also revealed that the FCC constructs were the hardest to learn when transiting from PL1 to PL2 due to negative semantic transfer. Therefore it is important to investigate how teachers can use explicit transfer interventions to help students avoid negative semantic transfer as well as deepen conceptual understanding. To investigate this, the following RQ is asked:

- (RQ4b): Can students' learning of False Carryover Concepts (FCC) be improved by student-centred transfer interventions?

With these research questions in mind, the next section elaborates the design of the study by explaining the methods and procedures used to collect and analyze data.

8.2 The Research Design

A between-subject study design was adopted in order to investigate how the transfer teaching interventions based on the MPLT can improve second PL learning. In a between-subjects design, every participant experiences only one experimental condition, and group differences between participants in various conditions are compared [138]. Participants in this study were allocated randomly into either the Intervention group or the Control group. The reason for random allocation was to prevent selection bias so that any difference in the outcome can be explained only by the transfer interventions. Both groups were homogeneous, as described in detail in the next section.

To summarize the design, the Intervention group received transfer interventions that allowed them to carry out student-centered code-comprehension tasks in Python (PL1) and Java (PL2). The details of this intervention will be explained in the following sections. The Control group did not receive the transfer interventions.

The benefit of choosing a between-subject design is that it increases internal validity by reducing the carryover effect in within-participant designs [139]. The carryover refers to any lin-

gering effects of a previous study condition that may affect a current study condition. Between-subject study designs are also beneficial because they do not require long periods, which makes them suitable for this research context that was carried out in a restricted classroom environment. However, the disadvantage of this design may arise when a cohort is split and exposed only half to a potentially beneficial intervention. This was mitigated by offering the Control group the same intervention after the experiment.

8.3 Participants and Context

In order to investigate the transfer interventions and answer the proposed research questions, the study consisted of participants who had just completed their first semester having learned object-oriented Python and now enrolled in object-oriented Java in their second semester. Unlike the previous studies that were conducted at the University of Glasgow, this study was conducted in Norway, at the University of Oslo. There are two main reasons for conducting the transfer interventions research at this University. Firstly, the lecture teaching both these courses requested guidance from the researcher on how to transfer the students successfully to a second programming language. Secondly, the study allows validating the MPLT in a different context, therefore providing evidence of generalizability across the conditions. The University enrolls students in the Introduction to object-oriented programming using Python in the first semester. Most of these students progress to object-oriented programming using Java in their second semester.

A total of 230 students participated in this study. The data was collected only from students who completed the quiz, had less than one year of programming experience, and only had knowledge of one programming language. It was done to reduce the noise and enhance the data analysis as much as possible. Therefore, the actual number of participants whose results were considered for analysis was reduced to 97 students. The chosen participants had an average of five months of programming experience using the Python programming language. Their age range was 18 to 24 years. They were recruited through emails and during the lecture period. They were informed that the tests were voluntary and could exit at any point of the study.

The study was a between-subject design. Therefore, 50 participants were randomly assigned to the Intervention group and 47 participants to the Control group. The Intervention and Control groups were given a Java pre-quiz in the second week of learning Java (PL2) on the TCC and FCC constructs in order to ensure that the groups were homogeneous with similar proficiency levels in programming knowledge. The results revealed no significant differences in their TCC scores (P-value = 0.71) and FCC scores (P-value = 0.74) as seen in Table 8.1.

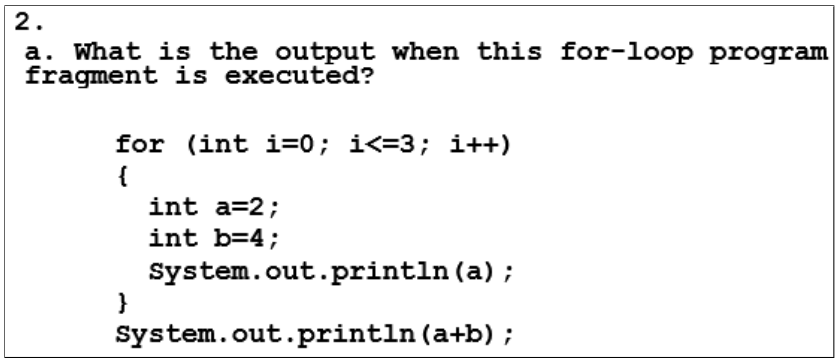


Figure 8.1: A screenshot of the sample Java question in the quiz given to the students after the interventions

8.4 Instrument

The research instrument used to obtain, measure, and analyze data from the participants was a Java code comprehension quiz that covers only the TCC and FCC constructs. The details about designing this instrument are already elaborated in Section 6.2. There are ten questions in the quiz consisting of five questions per construct category (TCC and FCC). It covers concepts that the participants were taught in both the first semester of the Python programming course and the second semester of the Java programming course. The quiz included the following concepts: types, expressions, operators, control structures, and functions. An example of a FCC question in the Java quiz for a *for-loop* construct is shown in Figure 8.1.

8.5 Data Collection Procedures

In the first week of the Java course, students attend two lectures, each having a duration of 90 minutes. After the lectures, students attend a 90-minute seminar/tutorial led by a Teaching Assistant (TA). Participants were randomly allocated to the seminar groups, and the groups were divided into the Control and the Intervention groups. The Control group had 50 participants, while the Intervention group had 47 participants. The data collection lasted for the duration of the seminar session (90 minutes) for both groups. It was divided into three stages, the pre-quiz, the interventions, and the post-quiz. The details of these three stages are as follows:

8.5.1 Pre-quiz

At the beginning of the first week of seminars, the Intervention group and the Control group were given a Java code-comprehension quiz that lasted for a duration of 20 minutes. The quiz covered the programming concepts in Chapter 6. The participants were taught these concepts in their first two lectures of the Java lesson. The rationale for the pre-quiz was to establish the baseline of participants' conceptual knowledge at this stage. An online platform for data

collection, Mentimeter, was used to anonymously collect both the pre-quiz and post-quiz data.

8.5.2 The Transfer Intervention (Student Centered Approach)

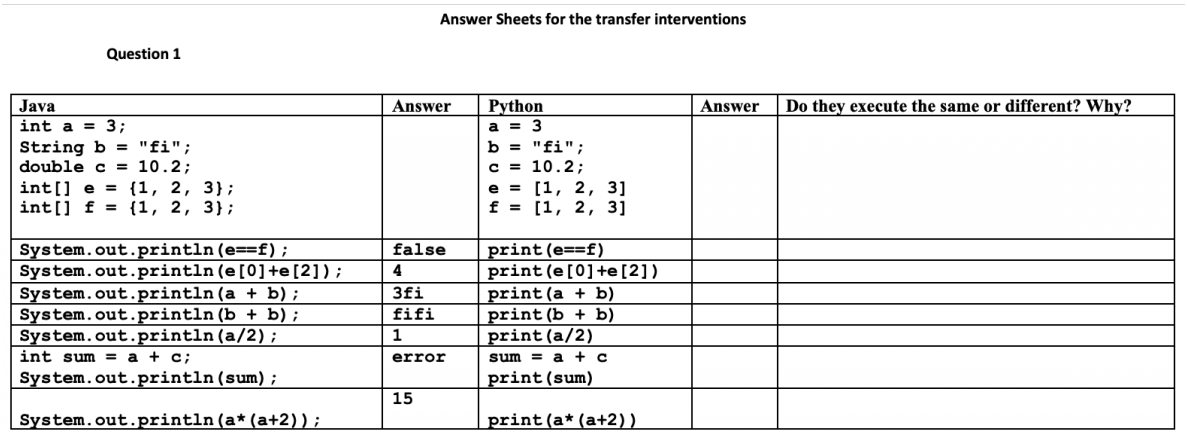
The transfer interventions were not given to the Control group. This group followed the usual seminar activities that the lecturer designed. These included the students solving problems using the Java programming languages. The solutions required the students to use the same concepts and constructs in Appendix 8. These were also covered in the pre-quiz and were also given to the Intervention group. The TA facilitated the activities for this session.

The Intervention group received the transfer interventions that were student-focused. The student-centered approach involves active learning that allows students to reflect in their learning [52, 137]. This active learning approach is also used elsewhere in computer science education research. For example, Parson's puzzles allow students to solve code comprehension programming puzzles and sort them in order [52]. By engaging in code comprehension activities, students get the opportunity to understand the programming language semantics rather than focusing on syntax errors when they try to write code. The student-centered approach can also involve collaborative learning, which allows students to work together to facilitate their learning [137]. In this study, collaborative learning was adopted by the Think Pair Share teaching strategy that helps promote students' teamwork and critical thinking skills [137, 140].

The intervention activity was as follows: At the beginning of the seminars, the tutors took 10 minutes to make the students aware of the transfer between programming languages based on syntax similarities. They explained how this transfer might be positive or negative by showing students examples of such scenarios. The students were then handed out a sheet that consisted of Java code-comprehension activities as shown in Figure 8.2. They were encouraged to work in pairs for 40 minutes. Their task was to map the Java constructs to their equivalent Python constructs. Specifically, the students were given instructions to identify semantic similarities and differences between Java and Python and explain their choice. The students were encouraged to do their investigations using the computer (either the programming language compilers or the internet). The tutors were moving around the seminar session and were available for any students' questions or clarifications.

8.5.3 Post-quiz

At the end of the seminar activities, students from both groups were given a post-quiz that took an average of 20 minutes. The Java code comprehension quiz covered concepts in Section 9.5, see Appendix D.



Answer Sheets for the transfer interventions

Question 1

Java	Answer	Python	Answer	Do they execute the same or different? Why?

int a = 3;
String b = "fi";
double c = 10.2;
int[] e = {1, 2, 3};
int[] f = {1, 2, 3};
		a = 3
		b = "fi";
		c = 10.2;
		e = [1, 2, 3]
		f = [1, 2, 3]		

System.out.println(e==f);	false	print(e==f)		
System.out.println(e[0]+e[2]);	4	print(e[0]+e[2])		
System.out.println(a + b);	3fi	print(a + b)		
System.out.println(b + b);	fifi	print(b + b)		
System.out.println(a/2);	1	print(a/2)		
int sum = a + c; System.out.println(sum);	error	sum = a + c print(sum)		
System.out.println(a*(a+2));	15	print(a*(a+2))		

Figure 8.2: A screenshot of an activity sheet given to students in the 90 minutes seminar activities for Intervention group at the University of Oslo. Students were asked to compare Java and Python semantics.

Table 8.1: Pre-quiz comparisons: Mean scores grouped by concept category

Category	Intervention group	Control group	P-Value
TCC	3.78	3.77	0.71
FCC	1.40	1.40	0.74

8.6 Data Analysis

The data for both pre-quiz and post-quiz was downloaded into a spreadsheet from Mentimeter.com. IP addresses and timestamps were removed from the dataset. A score of 0 was allocated for an incorrect answer, and a score of 1 was allocated for a correct answer on each question. For this study, each participant could receive a maximum of 10 scores per quiz.

The data analyses steps described in Section 6.2 in Chapter 6 will be followed in this chapter as well. The only difference was that this study used the non-parametric Mann-Whitney U test to compare the Control and Intervention groups' scores for both pre-quiz and post-quiz.

8.7 Results

The research questions of this study aim to evaluate if students' learning of TCC and FCC can be improved by transfer awareness-raising interventions. This section, therefore, presents the results and analysis to answer this research question.

Table 8.2: Post-quiz comparisons: Mean scores grouped by concept category

Category	Intervention group	Control group	P-Value
TCC	4.22	3.98	0.164
FCC	3.1	1.53	<0.001

Table 8.3: Post-quiz mean comparisons between the Control group and the Intervention groups using the Mann-Whitney U test.

Category	Construct	Intervention group	Control group	P-Value	Effect-size
TCC	String concatenation	1.00	0.98	0.32	0
TCC	While loop	0.86	0.84	0.75	0
TCC	Methods	0.51	0.35	0.12	0.24
TCC	Array addition	0.92	0.94	0.72	0.05
TCC	Operator precedence	0.98	0.98	0.98	0
FCC	String coercion	0.49	0.02	<0.001	0.81
FCC	For-loop-scope	0.55	0.04	<0.001	0.87
FCC	Typing (static vs dynamic)	0.59	0.45	0.16	0.19
FCC	Array equality	0.75	0.85	0.08	0.31
FCC	Integer division	0.96	0.19	<0.001	1.0

In analyzing the pre-quiz using the Mann-Whitney U test comparisons, it was revealed that there were no significant differences between the Control group and the Intervention group in the FCC and TCC constructs; see Table 8.1. These results show that the two groups were homogeneous in their Java programming language knowledge, therefore, were comparable at the beginning of the experiment. It should be noted that as predicted in the MPLT hypothesis in Section 6.5, both groups performed with higher scores in the TCC (Intervention=3.78, Control=3.77) as compared to the FCC. This provides further validation for the MPLT.

Table 8.2 shows the findings of grouping participants' post-quiz mean scores by concept category. The results show that the Intervention group (mean score=3.1) outperformed the Control group (mean score=1.53) in the FCC constructs with a significant difference (P-value=<0.001).

A Mann-Whitney U test was computed for each construct for further analysis as shown in Table 8.3. Consequently, to explore whether the difference between the means scores for each construct is significant, a Mann-Whitney U test was conducted, which indicates no difference with a small effect size between the groups in all the five TCC constructs, *String concatenation* (p=0.32), *While loop* (p=0.75), *Methods* (p=0.12), *Array addition* (p=0.72), *Operator precedence* (p=0.98).

However for the FCC concepts, the results revealed that the Intervention group significantly outperformed the Control group with a large effect size in three of the five FCC concepts, *integer division* (<0.001), *For-loop and Scoping* (<0.001), and the *String coercion* (<0.001).

The results revealed that students benefited more from the FCC interventions as compared to the TCC interventions. Nonetheless, students still performed better in the TCC concepts before and after the intervention. The next section discusses these findings and implications.

8.8 Discussion

The previous section reported on the results that answer the following research questions:

- (RQ4a): Can students' learning of True Carryover Concepts (TCC) be improved by student-centered transfer interventions?
- (RQ4b): Can students' learning of False Carryover Concepts (FCC) be improved by student-centered transfer interventions?

In answering RQ4a, the results of this study revealed that the student-centered transfer interventions did not improve participants' learning of TCC concepts. However, the findings revealed that the Intervention and Control group participants still performed better in these concepts than the FCC concepts. These findings do not imply that the TCC concepts are easier than the FCC concepts. However, they show the effect of semantic transfer based on syntax similarities. According to the MPLT hypothesis, if a learner encounters a TCC while learning PL2, the semantic transfer will occur from PL1 to PL2 because of syntax matching, resulting in a positive impact on learning PL2. This shows that semantic transfer that resulted in positive learning of Java occurred early, as reflected in the pre-quiz results, resulting in ceiling effects. It resulted in not much room for improving performance in these constructs in the post-quiz. This means that the participants in this study benefited from their knowledge of Python (PL1) constructs which helped them learn Java constructs effectively in the first week of learning Java. A high proportion of the participants in the study had high scores in the TCC concepts because of positive transfer. These results show the positive learning effect of the TCC concepts in the early stages of learning PL2 happens rapidly due to implicit learning. This may mean that in these concepts, interventions are not necessary. However, participants from both groups still struggled with understanding the Java methods in week 1 of learning the Java language, although the Intervention group performed slightly better than the Control group. This could mean that the methods may not be perceived as a TCC concept, as categorized by the researcher. There are significant syntax differences between Python functions and Java methods. For example, Java methods have return types and type declarations that Python does not explicitly have. Students have been reported to struggle with understanding Java methods in other studies [4, 141].

The Intervention group and the Control group performed equivalently poorly in the FCC concepts in the pre-quiz even after the constructs semantics were taught in class. According to the MPLT, semantic transfer affected learning negatively because the students transferred semantic knowledge from Python to Java due to syntax similarities, however it became a problem because Java behavior is different from Python in these constructs. In answering RQ4b, the results revealed that the Intervention group benefited significantly more in the FCC concepts than the Control group in most FCC concepts in the post-quiz. This could mean that as the students discussed code comprehension questions of Java and Python mappings they managed to help each other learn and reflect [52, 137] on the similarities and differences in behaviour of the two languages. Engaging in making comparisons and connections to prior knowledge helps students organise their conceptual knowledge structure [133], and understand concepts and se-

mantics better [142]. Other studies have used this type of learning in other transfer pedagogies like bridging and hugging [10, 45, 84, 85].

These findings also show the importance of explicit instruction in programming language transfer which gave the Intervention group the opportunity to become conscious about the programming language syntax and semantics [45, 143]. The activities that the Control group were engaged in, in this first week were problem-solving activities in the new Java language this does not give them the chance to focus and understand the semantics and concepts of the new language as they may struggle with syntax errors of the new language as this early stage [49–51,53]. This could mean students do not get the opportunity to engage deeply with the new language as their cognitive capacity is focused on problem-solving not understanding the meanings of constructs.

Overall, the findings in this study with regards to semantic transfer corroborate with Chapter 6 findings where the students were at a different university country and taught by different lecturers. This increases the validity of the MPLT. The results revealed that the transfer interventions are not helpful on the TCC concepts as the students learn them implicitly without much assistance. The results have also shown the potential value of explicit interventions in teaching students second and subsequent PL concepts in the FCC category. Although the Intervention group benefited from student-focused interventions in the FCC, they still performed lower in these concepts than the control group. This might imply that more vigorous interventions are necessary for these concepts. The next section, therefore, explores transfer interventions further.

Chapter 9

The Pedagogy for Transfer

[*Aspects of this study have appeared in [144].*]

Chapter 4 and Chapter 6 have confirmed that semantic transfer based on syntax similarities plays a crucial part in relative novices transferring between PLs as proposed by the first main claim of this thesis. The transfer can be positive, negative or there may be a lack of transfer as proposed in the MPLT in Chapter 5. The subsequent Chapter 7, revealed that despite students facing transition difficulties between PLs, most of the interviewed teachers do not pay attention to implementing transfer interventions in the classrooms. The second main claim of this thesis is that the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language. To assist teachers and encourage them to use transfer interventions, in Chapter 8, this thesis explored how explicit transfer interventions can help improve the learning of the TCC and FCC concepts. The results suggest that if pedagogy for second language learning is adjusted in alignment with the MPLT, learning of second programming languages can improve, especially for the FCC concepts. However, no pedagogy integrated with the larger learning design of a transfer course has yet been developed.

This chapter, therefore, proposes an integrated pedagogy that is aligned to all the three categories of the MPLT (TCC, FCC, and ATCC) that will improve conceptual transfer and understanding. The MPLT draws from both the cognitive and linguistic theories; hence this integrated pedagogy uses Implicit, Explicit, and Bridging interventions, which are also adopted in cognitive and linguistic disciplines [45, 185]. The previous study mildly implemented the transfer intervention for only 90 minutes in a seminar session. This study, however, embeds the transfer interventions in an integrated pedagogy of second language learning. It allows for the interventions to be used to improve PL conceptual understanding and not only correct misconceptions. Furthermore, these interventions are teacher-focused because the teacher is mostly more knowledgeable about programming languages and concepts. Therefore, they can guide students on relevant focus areas to deepen their conceptual understanding. Also, the teacher can follow an

integrated pedagogy in a controlled and orderly manner, ensuring that students do not miss out on important details about the transfer.

This chapter begins with the research questions followed by the design of the pedagogy. The design of the pedagogy section involves the pedagogical approaches (Implicit, Explicit, and Bridging) adopted and how each of them can be implemented in alignment with the three categories of the MPLT. The section that follows is the instrument used which is adopted from Chapter 6. The data collection follows next and includes the process of gathering and measuring data on implementing the pedagogy. The data was collected quantitatively and qualitatively. The rest of the chapter describes the data analysis and results of the experiment. The chapter concludes with a discussion.

9.1 Research Questions

The overarching Research Question 4 of this thesis is: *How can transfer teaching interventions based on our understanding of semantic transfer improve second PL learning?*

According to the MPLT, when students learn PL2, there is positive semantic transfer when they learn TCC constructs, negative semantic transfer when they learn FCC constructs, and no/minimal transfer when they learn ATCC constructs. Therefore, it is important to investigate how implementing a pedagogy aligned with theory on how students transfer as proposed by the MPLT will help them effectively learn second programming languages and improve their conceptual understanding. To investigate this, the following specific RQ is asked:

- (RQ4c): Can students' learning of TCC, FCC and ATCC be improved with an explicit transfer pedagogy?

Obtaining the students' and teacher's feedback regarding the pedagogy can give insights into what works and does not work, therefore helpful to improving the pedagogy. In addition, if the teacher's feedback is mostly positive, this can encourage other teachers to reflect and improve on their own teaching of second languages by adopting the pedagogy. Therefore, the next two RQs are as follows:

- (RQ4d): What are students' views on learning the PL concepts through the transfer pedagogy?
- (RQ4e): What are the teacher's views on teaching using the transfer pedagogy?

The next section elaborates the design of the pedagogy and how it aligns to the MPLT.

9.2 The Development of the Transfer Pedagogy

This section describes the process of developing the Transfer Pedagogy that fosters conceptual transfer and understanding. It starts with a description of the pedagogical approaches used and then presents the framework of the Transfer Pedagogy.

9.2.1 The Pedagogic Approaches used for the Interventions

Pedagogy is a method that is used to promote learning, and it includes techniques, learning activities, and strategies that provide the environment where learning may take place [115].

In a traditional classroom, the teachers usually teach by giving students information and exercises with an assumption that they will receive and revise their mental models and store the new learning in a coherent framework [115, 145]. As elaborated in the previous section, this assumption can be wrong when learners learn second programming languages because their understanding may remain partially, or they may incorrectly carry over learning from the first language. In this section, this thesis describes specific pedagogic practices within programming languages and natural languages.

Constructivist Instruction

Constructivism has had significant influence in the teaching of programming in computer science. It is a theory of learning that suggests that learners actively construct new understandings and knowledge, integrating these with their background knowledge [186]. Constructivist instruction is based on the belief that learning occurs as learners are actively involved in a process of meaning and knowledge construction as opposed to passively absorbing it from textbooks and lectures [50, 187]. It involves classroom activities such as higher-level problem solving, critical analysis, enquiry-based learning and collaborative learning. The teaching practices that have successfully adopted this method in programming courses include students live-coding and problem solving during class [188, 189]. This approach can also be used in code-comprehension activities [197]. The only challenge that may arise with this approach is when constructivist learning problem solving activities precede explicit teaching strategies and teachers incorrectly assume that students have sufficient prior knowledge to generate new learning constructively [146]. If relative novices start writing programs or engaging in problem solving activities too early it may delay their development of viable mental models of program execution, given that their understanding of the programming language (e.g syntax, semantics/notional machine, concepts) is still very fragile [3, 49–52]. Furthermore, it has been reported that without explicit instruction, the fragile knowledge of the first language negatively impacts their learning of the subsequent programming languages [29, 30].

Implicit Instruction

Implicit learning is a method of learning that may be passive/unconscious [195], as compared to active construction of knowledge. The implicit learning process can happen naturally and intuitively for learners without much consciousness. It then means that the learner usually learns without metalinguistic awareness. Early work on implicit learning was from cognitive sciences [191, 192]. It has now become popular in second language learning [193, 194]. Implicit instruction is a method that allows learners to deduce grammar rules without necessarily being aware [185]. Therefore, they internalize the underlying grammar rule without the teacher drawing their attention to it. This instruction makes use of the learners' implicit learning process. The teacher allows the learners to create their own conceptual structures and assimilate new information upon presenting the learning material to them.

The implicit instruction can be applied when learners are taught the TCC concepts during PL2 learning. This is because the similarities in PL1 and PL2 can help the learners learn PL2 implicitly due to the positive effects of semantic transfer. For example, a student's prior knowledge of a Python *if-statement* may effortlessly help them learn the Java *if-statement* because it looks and works the same way in both languages.

Explicit Instruction

Unlike implicit instruction, explicit instruction is a method of teaching that usually focuses on developing the learners' metalinguistic awareness [185]. The learners are conscious of the learning of the language, its semantics, and concepts. This instruction can be done inductively by assisting the learners in making discoveries about the language grammar rule [185]. It can also be done deductively by giving the learners descriptions, and examples of the grammar rule during teaching [185]. In this teaching method, the teachers also provide learners with metalinguistic corrective feedback of the target language when they encounter errors during teaching.

There are various teaching activities a teacher can adopt, such as scaffolding, and corrective feedback, activating learners' prior knowledge, and making lesson objectives that target building on this prior knowledge [185, 196]. This method is suitable for teaching the FCC concepts because negative semantic transfer from PL1 may cause misconceptions in PL2. Applying these activities may help learners restructure and correct their mental models to accommodate PL2.

The Bridging Technique

The Bridging intervention proposed by Perkins and colleagues [45] is a form of explicit instruction that assists the learners to transfer semantic/conceptual knowledge that they failed to transfer implicitly. The Bridging intervention encourages the making of abstract conceptual associations between the initial learning, and its target domain [45]. In this intervention, the teacher uses

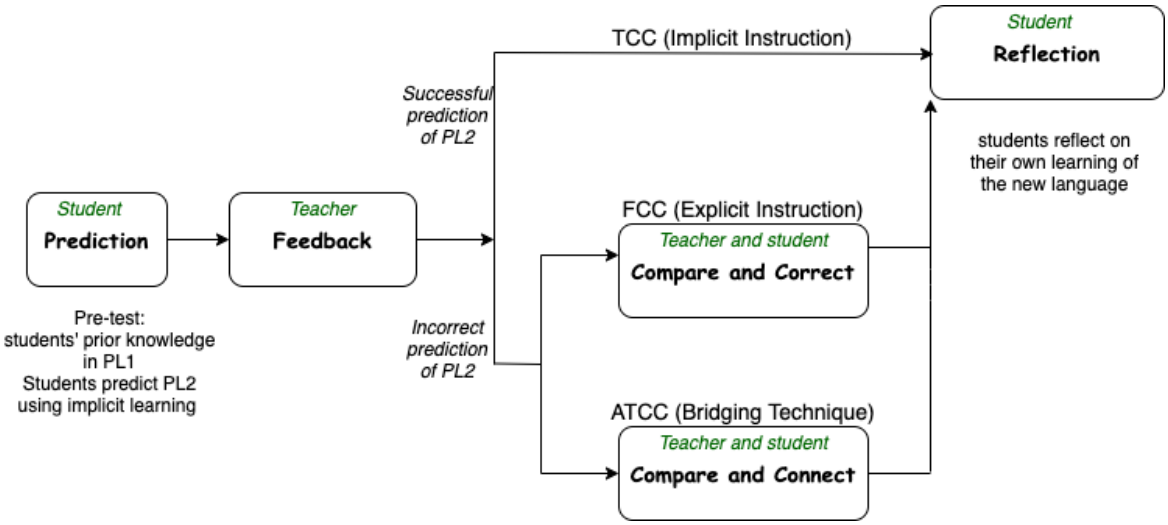


Figure 9.1: Transfer Pedagogy Framework aligning to the MPLT categories

scaffolding to break up new concepts and build on students' prior conceptual knowledge to ease their learning. Analogies that connect the learner's initial and target conceptual knowledge are used to help foster transfer. These interventions are appropriate for teaching the ATCC concepts because the students usually fail to connect these concepts to their prior language knowledge because of failure to notice syntax similarities between PL1 and PL2.

9.2.2 The Pedagogy for Second Programming Language Learning

The pedagogy for second programming language learning that promotes conceptual transfer and understanding is presented in this subsection as shown in Figure 9.1. The activities involved in the transfer pedagogy are presented from the perspective of the different MPLT concept categories, TCC, FCC, ATCC, in alignment with the pedagogical approaches in Section 9.2.1. The first (prediction) and the second (Feedback) steps are carried out at the initial learning of PL2 and engage the students explicitly on second language learning issues. The third step is the intervention step and varies depending on the MPLT concept categories being taught. Students then actively reflect on the concepts they have learned in the final step.

The description of the transfer intervention activities in Figure 9.1 is as follows:

1. **Prediction**: The teacher initially creates and categorizes some code-comprehension questions according to the Model's TCC, FCC, and ATCC concepts in the form of a quiz; see Appendix B. At the beginning of the PL2 lesson, the students are given the code comprehension task of the quiz in PL1 and PL2. There are two reasons for giving the students a code comprehension quiz in both languages. The first reason for the comprehension task in PL1 is to understand the students' prior knowledge. This activity is important to create a learning objective/activity to build on their prior knowledge. The second reason for the prediction task/guess quiz in PL2 (before students learn PL2) is to understand their implicit transfer of knowledge and their initial thoughts on PL2. Engaging students in guess quizzes (before learning) activate consciousness and is also reported to be beneficial to students [163, 164] it gives them room to fail hence providing the opportunity for corrective feedback in PL2.

2. **Feedback**: Students are given the quiz results on how they performed in PL1 and PL2. This feedback is given to improve their consciousness and awareness of the transfer mechanisms when learning PL2. Furthermore, students are given explanations about their performance and meaningful examples. Interventions vary per concept category and are carried out as follows depending on the results of the quiz:

    - **TCC-Implicit:** The implicit learning of PL2 based on syntax and semantic similarities between PL1 and PL2 usually makes the students successful at the prediction stage in the TCC concept category. Therefore the suitable instruction for the TCC concepts is Implicit instruction. The teacher takes advantage of the implicit learning and just teaches students PL2 concepts without referring to PL1. Students are then immediately given problem-solving exercises. The teacher does not need to provide corrective feedback because students' learning is usually positive and not affected by the negative semantic transfer. However, the teacher can still reactively give corrective feedback if students experience errors in problem-solving in a new language. Some minimal explicit instruction can also be applied at this stage, where necessary.
    - **FCC-Explicit (Compare and Correct):** Students are usually unsuccessful in predicting the FCC concept category because of negative semantic transfer. Therefore, explicit instruction is the best transfer intervention for this concept. The teacher gives students corrective feedback on their errors and refers back to the first language. This is done by comparison exercises/examples between PL1 and PL2. These exercises/examples allow the teacher to explain deeper programming concepts. The teacher makes students aware and conscious of the PL2 language elements (syntax,

semantics, and concepts) before giving them problem-solving activities. Students' group/pair discussions are also encouraged.

- **ATCC-Bridging (Compare and Connect):** Students are usually unsuccessful in predicting the ATCC concept because of failure to transfer from PL1 to PL2 due to different syntax between PL1 and PL2. Therefore, it is suitable to use the Bridging technique for the transfer intervention in this concept. The activities include using analogies to help connect common concepts between PL1 and PL2 hence fostering transfer. The teacher does explicit interventions by linking programming concepts in PL1 and PL2 and using examples of these same concepts in both languages to assist the students. Students are given more examples in different contexts after the initial bridging activity. The teacher allows the students to ask questions for clarifications during the intervention.

3. **Reflection**: At this last stage, students reflect on their learning. This reflection also allows them to identify what caused their success or failure to transfer well and what needs to change. The activities around reflection may include discussions (groups or pairs), individual reflections, or even journal entries.

This pedagogical framework can be flexible and adopted to suit the teacher's lesson plans. Teachers can follow the pedagogy by moving back and forth between the pedagogy parts. For example, prediction quizzes can be given in chunks at different stages of the lesson, or reflections can be done throughout the lesson.

9.3 Participants

In order to investigate the transfer pedagogy and answer the research questions on the effectiveness of transfer interventions on second programming language learning, the study consisted of participants who had just completed their first year of procedural Python and were now registered for the object-oriented Java in their second year at The University of Glasgow. The study was a between-subject study design, therefore involving the Control group and the Intervention group. The Control group involved 120 participants who were registered for the object-oriented Java course in the year 2019, and the Intervention group involved 84 participants who had registered for the **same** course in the year 2020. Just like in the previous chapters, to reduce noise, the only data included for analysis was for participants with only one programming language knowledge (Python), less than one year of programming experience, and had no knowledge of the Java programming language. It reduced the number of Control group participants to 30 and the intervention group to 32.

The reason for getting participants from different academic years is because the study was carried out in a real-life classroom hence it was a challenge to carry out controlled experiments

Table 9.1: A Python baseline assessment analysis using Mann-Whitney U test comparison: Mean scores (out of total 4) grouped by MPLT concept category.

Concept Category	Intervention group	Control group	P-Value
TCC	3.50	3.53	0.822
FCC	3.37	3.09	0.107
ATCC	2.83	3.06	0.542

that involve dividing a cohort and exposing only half to a potentially beneficial intervention. To ensure that the two groups are homogeneous, they were given a Python baseline pre-quiz that covered the same concepts as aligned with the MPLT concept categories (TCC, FCC, and ATCC). An analysis using Mann-Whitney U test comparisons revealed that there were no significant differences in the scores between the groups with regards to their proficiency in the Python language; see Table 9.1. Furthermore, the same lecturer delivered the course for both groups. The groups were also at the same level of study (year 2) and had the same age range (18-23 years).

The threat to the validity of this study is that the delivery of the lectures was different. In the year 2020, the course was delivered online because of the Covid-19 pandemic. Research has, however, reported that there is no significant difference in the students' scores between online and face-to-face instruction [198, 199]. Also, even though the delivery of instruction was different, the course covered the same learning material. Participants were assured that their participation in the study is anonymous and voluntary and that taking part in the transfer quizzes does not affect their grades. Before the study, participants had to sign a consent form.

9.4 The Research Design

The study adopted a between-subject study design to investigate how transfer pedagogy will improve second PL learning. The participants from both groups were offered the Java course for two and a half weeks, which totaled five one-hour lectures. In addition to the lectures, participants were offered a 2-hour lab session for Java programming exercises each week. For the two and a half weeks, the course covered concepts on identifiers, primitive and non-primitive types, composite types (arrays and objects and classes), scope, loops, methods. The concepts were introduced in the same order from Lessons 1-5 for both groups. There were also live coding sessions for both groups.

9.4.1 The Control Group Course Delivery

The Control group attended the Java course face-to-face and did not use transfer interventions. The Java lecturer did not teach referring back to the Python language, which the students already know. The course did not adopt the Explicit and Bridging transfer interventions pro-

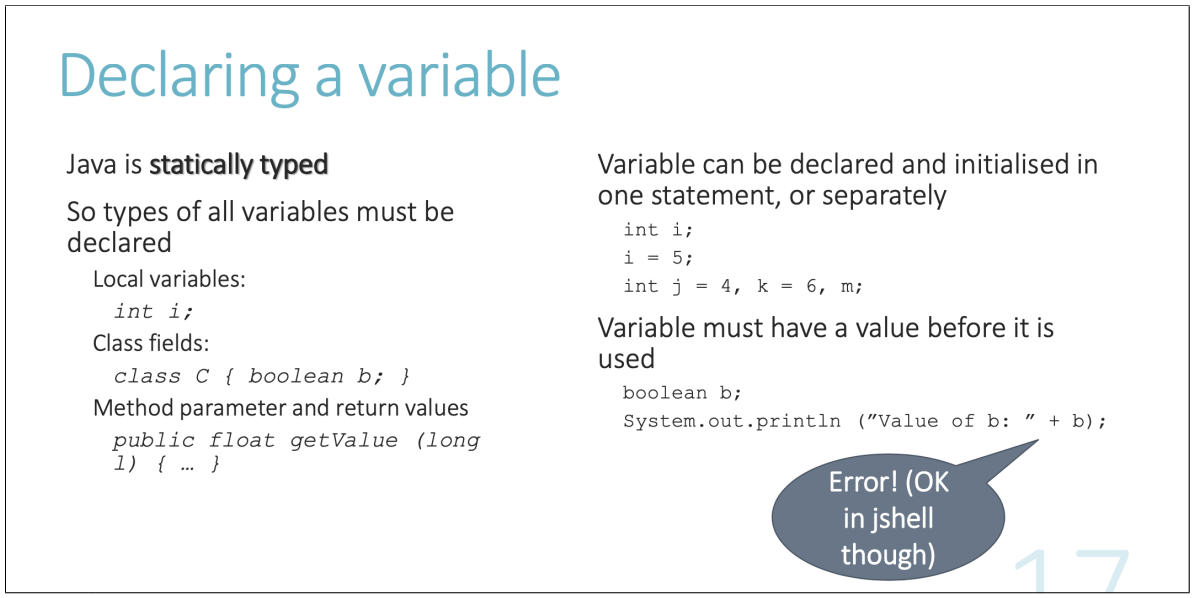


Figure 9.2: A screenshot with an example of an FCC concept variable declaration in Lesson 2 as it was taught in the control group (2019) without explicitly comparing with Python

posed in the transfer pedagogy. For example, when the lecturer introduced variables and declarations to students, they did not compare Java as a statically typed language and Python as a dynamically typed language, see Figure 9.2.

9.4.2 The Intervention Group Course Delivery

The Intervention group attended the Java course online and were taught following the transfer pedagogy in Section 9.2. The pedagogy was embedded into the Java course as follows:

Prediction

At the prediction phase (Lesson 1), participants were given code-comprehension quizzes in Python and Java. They were given the Python quiz to get their baseline knowledge. The Java guess quiz was given to make students actively think about the PL semantics and engage with prior knowledge. The guess quiz was also given to investigate the students' implicit transfer from Python based on syntax similarities, as proposed in the MPLT.

Feedback

In Lesson 2, participants were given group feedback in the form of a bar graph showing how they performed in the Java quiz as compared to the Python quiz. As expected in the MPLT, participants performed better on TCC than the FCC and the ATCC, see Chapter 6 for similar results in guess quizzes and explanation. The lecturer made students aware of how the transfer from Python to Java happened, including explaining both negative and positive transfer. Some

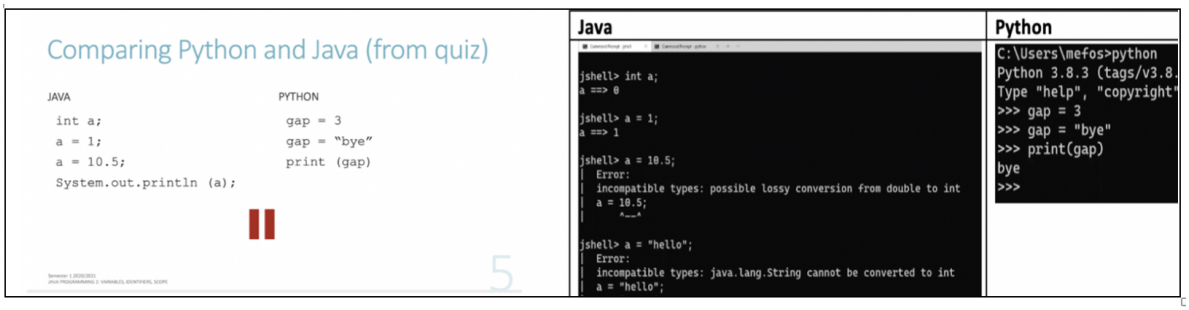


Figure 9.3: Lesson 2 notes and live coding screenshot of FCC concept: The explicit instruction of dynamic vs static concept during a live coding session in class

of the concepts covered in the quiz were explicitly taught in this lesson, referring back to prior Python language. The rest of the concepts were introduced in subsequent Java lessons.

FCC-Explicit (Compare and correct)

The Explicit instruction using the **compare and correct** interventions was adopted whenever the FCC concepts were taught in any of the five lessons. Figure 9.3 shows how the FCC concept of *dynamic* (Python) versus *static* (Java) was introduced to students in-class notes and live coding sessions. The lecturer started by making students aware that they incorrectly guessed this concept in the guess quiz. The lecturer made explicit comparisons of the Python and Java construct and corrected students' misconceptions that were caused by negative semantic transfer, see Figure 9.3. The lecturer also engaged students with live-coding activities in the same concept using the two languages to explicitly show when the code will compile or when it will give errors in each language. This allowed for students to review their conceptual knowledge and restructure it hence deepening conceptual understanding.

ATCC-Bridging (Compare and connect)

The Bridging technique using **compare and connect** was adopted when the lecturer introduced objects in Lessons 4 and 5. An analogy for Java objects used was Python dictionaries, because dictionaries and objects have similar underlying semantics even though the syntax looks different. A dictionary is a value with identity and has a collection of name/value pairs that can be updated just like Java objects have a value with an identity that has attributes/fields that can be updated. The difference is that the Java objects and classes are abstract types with hidden implementation details. A Java object has methods defined in the class definition. While these are not part of a dictionary, a Python user-defined data type can be created using a dictionary and associated functions. Figure 9.4 shows how the bridging intervention was implemented during a live coding session. Students were given more Java objects examples after the intervention.

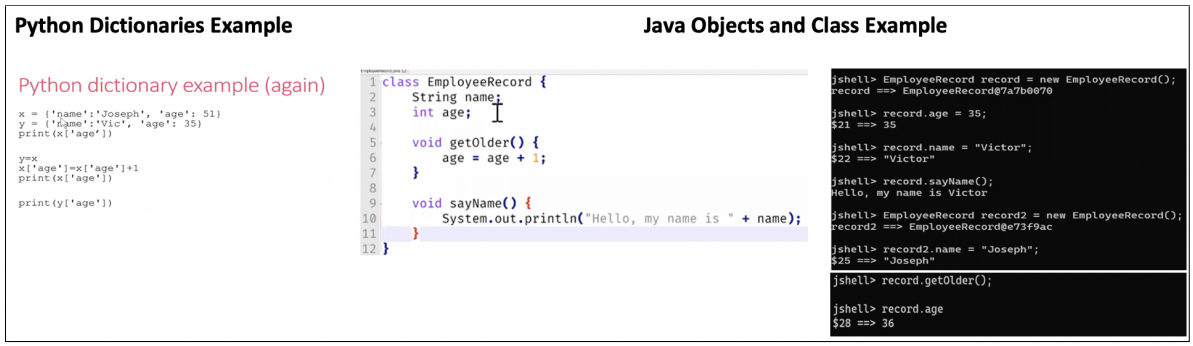


Figure 9.4: Lesson 5 screenshot: The bridging intervention technique the lecturer used for Python dictionaries and Java objects during live coding session in class

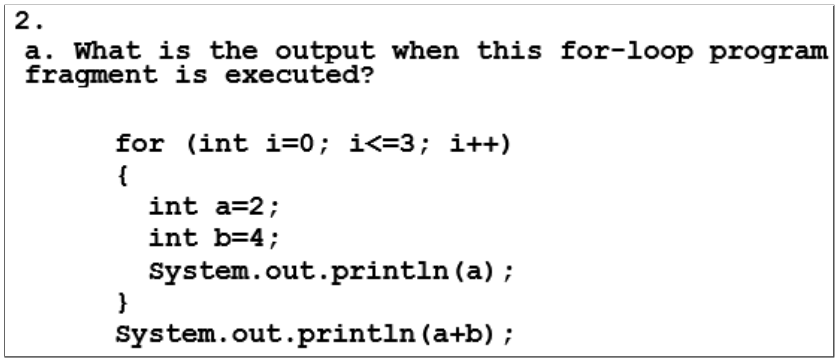


Figure 9.5: Sample screenshot for the post test question-FCC category

Reflection

At the end of the interventions students were given open-ended surveys to write their experiences and reflect on their learning.

9.5 Instrument

Python and Java code comprehension quizzes that cover the TCC, FCC, and ATCC constructs were given to the participants to obtain, measure, and analyze data, see Appendix E. The details about designing this instrument are already elaborated in Section 6.2. An example of the categorised concepts for the quizzes is in Figure 9.6. Each quiz consisted of twelve questions for this study consisting of four questions per construct category (TCC, FCC, and ATCC). Both quizzes covered concepts that the participants were taught in both the Python and Java course. These include expressions, operators, scoping, control structures (indefinite and definite loops), functions, and data structure. An example of a FCC question in the Java quiz for a *for-loop* construct is shown in Figure 9.5.

Java	Python
int[] len={4,4,5}; int[] can={4,4,5}; System.out.println(len==can);	len=[4,4,5] can=[4,4,5] print (len == can)
Answer: False	Answer: True

Figure 9.6: An example of a FCC concept of array equality in Java and Python

Assume the Java code below, to declare variables, has been executed.

What would be the value after executing each of the following expressions of Java code?

```
int[] len={4,4,5};
int[] can={4,4,5};

System.out.println(len==can);
```

Figure 9.7: Week 3 sample Java quiz on array equality

.

9.6 Data Collection

The Control and the Intervention group were given a code-comprehension quiz in Python at the beginning of the course to assess their baseline knowledge; see Appendix E. They were also given a code-comprehension quiz in the Java language in the third week of learning Java to investigate the effect of the pedagogy on their conceptual and semantic knowledge of PL2. The Python and the Java quiz took 20-25 minutes to complete. All participants were able to finish the quizzes before or on the given allocated time without reporting any difficulty.

.

Participants in the intervention group were given a Likert scale survey and open-ended questions to report on their experiences of being taught using the pedagogy. In addition, a few participants who volunteered were interviewed to gain in-depth knowledge of their experiences. The questions are described in Section 9.8.2. The lecturer was also interviewed for one hour to get their experiences with using the pedagogy.

9.7 Data Analysis

The quantitative data and qualitative data was analysed as follows:

9.7.1 Quantitative Analysis

The quantitative data analyses followed the steps outlined in Section 6.5. An Excel spreadsheet was used to record scores. A score of 0 was given for an incorrect answer, and a score of 1 was given for a correct answer for each of the individual constructs. That meant that the maximum scores a participant could get were 12, with each concept category (TCC, FCC, and ATCC) containing four questions. The mean score in each category was calculated using the R package. The non-parametric Mann-Whitney U test was used to compare the significant difference in the scores between the tests because the data was not normally distributed. The generally accepted p=.05 was regarded as the cut-off point; therefore, statistical significance that is accepted is when *p-value* is less than or equal to .05 [115]. P-values alone do not indicate the extent of the difference (if any exist) therefore the effect size was calculated. A recommendation from Cohen's classification of effect sizes which is small (d = 0.2), medium (d = 0.5), and large (d $\geq$ 0.8) was used.

9.7.2 Qualitative Analysis

The qualitative data analyses using thematic analysis followed the steps outlined in Chapter 7. In this approach, the researcher closely examined the interview and survey data to identify common themes, ideas, and patterns of meaning that came up repeatedly. The researcher then discussed the initial codes with the researcher's supervisor, and conflicts were discussed and resolved. The initial codes were generated inductively, which were then merged and collapsed into related themes through an iterative process.

9.8 Results

9.8.1 Week 3 Java Quiz Results Comparisons

The data was analysed using a Mann-Whitney U test comparison and the results revealed significant differences in the Java mean score grouped by concept categories between the groups on the FCC and ATCC scores. There were no significant differences in the TCC score; see Table 9.2.

A Mann-Whitney U test was computed for each construct in the quiz for further analysis as shown in Table 9.3. The results revealed that there was no significant difference between the groups in each of the TCC constructs. This may be because of the ceiling effects. The participants had scored high in the TCC concepts leaving not much room for performance to improve in these constructs.

Overall the Intervention group performed better than the Control group on FCC constructs with a significant positive difference on three concepts: *string coercion* (p <0.001), *array equality* (p <0.001), and *string multiplication* (p=0.013). The effect size for the *string coercion* and

Table 9.2: A Java post-quiz analysis using Mann-Whitney U test comparison: Mean scores (out of total 4) grouped by concept category

Category	**Control group (2019) Mean**	**Intervention group (2020) Mean**	**P-Value**
TCC	3.47	3.55	0.525
FCC	1.1	2.75	0.001
ATCC	1.7	2.41	0.020

Table 9.3: Mean scores and effect size of individual constructs tested in Control and Intervention groups in Week 3

Categ	**Construct**	**Control-2019 (n=30) Mean**	**Interven-2020 (n=32) Mean**	**P-Value**	**Effect size**
TCC	String concatenation	0.90	1.00	0.149	0.26
TCC	Operator precedence	0.87	0.91	0.765	0.05
TCC	While loop	0.80	0.78	1.00	0
TCC	Functions	0.90	0.86	0.538	0.11
FCC	Array equality	0.13	0.66	<0.001	0.68
FCC	String coercion	0.13	0.78	<0.001	0.75
FCC	String multiply	0.37	0.69	0.013	0.45
FCC	Int division	0.47	0.63	0.208	0.23
ATCC	Object retrieval	0.55	0.77	0.017	0.43
ATCC	Object update	0.55	0.64	0.291	0.19
ATCC	Object assignment	0.53	0.64	0.202	0.23
ATCC	Object aliasing	0.07	0.36	0.009	0.47

the *array equality* was medium. The Intervention group also performed significantly better than the Control group on ATCC constructs on two concepts: objects aliasing (p=0.009) and objects retrieval (p=0.017).

9.8.2 Students' Feedback

Students in the intervention group were asked to answer the following questions on a 5 point Likert Scale: "Was it helpful when the teacher showed you the differences between the languages". Out of the 32 responses, 44% of the students *strongly agreed* to this statement, and 41% reported to *agree*. Only 3% reported to *disagree*; see Table 9.4. The students were also asked: "Was it helpful when the teacher showed you the similarities between the languages"; 25% of the students *strongly agreed* to this statement, and 53% reported to *agree*.Students were lastly asked: "Was it helpful when the teacher showed the connection between Python dictionaries and Java objects"; 9% of the students *strongly agreed* to this statement, and 38% reported to *agree* while the majority (50%) were *undecided*.

Students were asked the following open-ended question to get in-depth knowledge of their

Table 9.4: Week 3: Intervention group (2020) feedback on the transfer interventions per concept category on a Likert Scale, n=32.

Category	**FCC interventions helpful**	**TCC interventions helpful**	**ATCC interventions helpful**
Strongly Agree	44%	25%	9%
Agree	41%	53%	38%
Undecided	12%	16%	50%
Disagree	3%	6%	3%
Strongly Disagree	0%	0%	0%
Total	100%	100%	100%

given answers in Table 9.4: "Share any thoughts you might have on your learning experiences of the programming concepts covered in this survey during the past two weeks. e.g., You can further explain in more detail why you chose the level of agreements in the above questions". Twenty-five students out of the thirty-two answered this open-ended survey. They are represented as *student 001-student 025.* The analysed results using thematic analysis method produced the following themes:

The teaching method was helpful: Most of the students (22 out of 25, 88%) reported that the transfer interventions were helpful. Their answers were further divided into the following sub-themes:

1. ***Helped me understand concepts better:*** Five students reported that the interventions assisted them in understanding the programming concepts better. For example, student 013 explained about their experience of learning the block scoping concept *"If you create a variable inside a Java for-loop that only exists inside a for-loop. I was introduced to scoping in my Python course but I did not really have a strong grasp of it. The Java lecturer wrote sample code and common pitfalls using the Java-Python survey examples and showing global variables and local ones which made it easier to avoid mistakes."* This student went on to explain their experience with learning referencing concept *"We did not get exposure to the concept of referencing in Python, we heard about it but did not see much of it in practice [.....]. The Java lecturer showed us how to use .equals() when it comes to comparing objects not ==. In the String the way Java references is not like Python, in Java it checks if the addresses are equal while in Python it checks the values."* Student 003 also mentioned that *"I had no idea what static and dynamic typed languages meant before starting the Java course. I learnt that Python is dynamic while Java is static"*.

2. ***Helped me recognise the difference between the two languages:*** Nine students expressed that it helped them understand both languages better. For example, student 006 expressed that *" It helped me recognize the differences between Python and Java and hopefully be competent in both of them."* and student 017 mentioned that *"In Python you don't have to declare the type of a variable, I thought in Java I could just say a="abc" but*

in Java I have to be specific and say String a="abc."

3. ***Helped me aware of the habits not to continue in the new language:*** Four students described that the transfer interventions helped them avoid negative transfer from their Python knowledge to Java knowledge. For example, student 032 mentioned that *"I think it's more useful to point out the differences since Python is something most of us are already familiar with and often use this experience as a "fallback" when we are not sure about how something works. Therefore, it is more useful to know when this will not produce the expected results rather than when it will."* while student 003 said *"If I am coding in the lab and it does not work, I go back to the slides and realize the lecturer warned me."*

4. ***Helped me connect to my prior knowledge:*** Four students expressed that it helped them to connect to their Python knowledge. Student 001 said *"By doing comparisons it helped me understand as I had previously done Python"* while student 007 said *"It was very helpful to use a language we learned last year to help us in transitioning from Python to Java and it's definitely better than starting a programming language from scratch:)"* Student 017 also said *"The quiz at the beginning was quite interesting to see how well I still remember Python and how well I think Java would look like."*

The teaching method was not helpful: Three students reported that they did not find the teaching method helpful. Their answers were divided into sub-themes as follows:

1. ***It makes learning harder:*** One student felt that the ATCC interventions made their learning harder, saying *"I feel Java objects and classes are a bit hard to understand because we never worked with objects and classes with Python, so trying to compare them to dictionaries actually makes the learning process a bit harder"*(student 014).

2. ***It was confusing:*** Two students felt that the lecturer using two languages to teach confused them. For example, student 009 said *"It gets more confusing when two languages are taught at the same time."*

9.8.3 Lecturer's Feedback

At the end of the interventions, a one-hour semi-structured interview was given to the lecturer to get insights into their experience of adopting the pedagogy. The following lead questions were asked to the lecturer: "Why did you agree to take part in modifying your pedagogy?", "Do you believe the new teaching is beneficial or not?" and "Did you experience any challenges?" This interview was just for one person, and the highlights of the narrative are as follows:

I was not aware of transfer challenges in the classroom: *"The 2019 pre-quiz showed that the students were affected by False Carryover concepts in Java, I was initially not aware they will have misconceptions but it was clear they did, so I thought let me explicitly explain these concepts instead of hoping they will figure it out at some point".*

I enjoyed using the pedagogy: The lecturer reported that they enjoyed using some parts of the transfer interventions and they found them beneficial. Their response is as follows:

1. ***The prediction stage was helpful:*** *"I liked that I gave out a Python quiz and a Java prediction quiz/survey at the beginning of the course. I would advise other lecturers to know what prior knowledge students are bringing into the class, rather than assume you actually know how they performed"*

2. ***The feedback stage was helpful:*** *"I liked that I was giving the students the feedback to show them how they transferred knowledge from Python to Java. I enjoyed explaining the concepts based on this quiz survey results."*

3. ***The compare and correct stage was helpful and provided opportunity to deepen conceptual understanding:*** *"I explained to them basing on their Python performance and explaining why they got things wrong in Java. I hope they get over the confusions about how the languages behave differently sooner. That way it helps us to start learning deeper concepts of Java and avoid them getting stuck. I also think that explaining that a different language does it in a different way kind of gets the students to understand the underlying model/ways of concepts a bit more. For example, showing how variables are expressed differently in both languages may help you understand them more clearly."*

I had challenges adopting the pedagogy

The lecturer also reported that they faced some challenges because they lacked proficiency in PL1 (Python). the lecturer also reported the students' lack of participation. Their response is as follows:

1. ***I had limited knowledge of Python:****"One of the challenges was not knowing Python enough to explain it in a way that it makes sense. I probably could have done a better job in this intervention. I tried to do a live-coding example of Python in class and failed miserably. I had to learn Python and understand how it works before doing this pedagogy."*

2. ***I had challenges implementing the compare and connect stage:*** *"I had challenges with ATCC interventions, I tried to connect Java objects to dictionaries when I introduced objects. I was a bit skeptical because I was not sure if they mean the same thing. I don't really know much about dictionaries."*

3. ***I had challenges with a diverse group of students:*** *I made sure I gave examples of the two languages during class using live coding, however, often those quickly spun off in the wrong directions, which did not follow the transfer plan for the lectures due to experienced students getting curious and asking questions about code. Inexperienced students were shy to participate.*

9.9 Discussion

This chapter answers the following specific research questions:

- (RQ4c): Can students' learning of TCC, FCC and ATCC be improved by the transfer pedagogy?
- (RQ4d): What are students' views on learning the PL concepts through the transfer pedagogy?
- (RQ4e): What are the teacher's views on teaching using the transfer pedagogy?

The discussions of this research will be presented as a reflection to each research question:

RQ4c: Effect of the pedagogy on learning of the TCC, FCC and ATCC

Just like in the previous chapter, the results of this study revealed that the implicit instruction based on the transfer pedagogy did not improve participants' learning of TCC concepts. However, as expected (from Chapter 4 and Chapter 6) the participants still performed better in these concepts than other concepts because they were assisted by implicitly mapping syntax and semantics of their prior knowledge (Python) and new knowledge (Java) concepts resulting in subsequent positive semantic transfer. It, therefore, suggests that these concepts can be implicitly taught as proposed in the pedagogy. Teachers do not need to put much focus on explaining and making comparisons in the TCC concepts as the implicit learning mechanisms automatically assist in the learning of these concepts.

The results of this study reveal that the explicit [185] and bridging [45] transfer interventions were mostly effective when transitioning students from procedural Python to object-oriented Java for most concepts in the FCC and ATCC concepts. The rationale for this outcome is in the activities implemented during these transfer interventions (e.g., corrective feedback and analogies). They raised the students' consciousness and awareness of the nature of the programming languages (concepts, semantics, syntax), therefore, assisting them in modifying and correcting their existing conceptual knowledge as proposed by the MPLT [30] and other cognitive studies [45, 185]. Students, however, did not significantly improve in their learning of the FCC *(integer division)* concept. The rationale for this may have to do with students intuitively transferring their mathematical knowledge of this concept, which works like Python. This implies that more intense transfer interventions may be beneficial for this concept or other concepts similar.

RQ4d: Students' views on learning the concepts through the transfer pedagogy

The results revealed that most students found the transfer interventions that make the PL1 and PL2 comparisons most beneficial. This may mean that these interventions raised the students'

awareness and consciousness when learning PL2. In other words, their metalinguistic ability was engaged [185] during the interventions. These results also imply that the students learn better from their own failed attempts and corrective feedback. A few students (3) reported that it was challenging for them when the teacher used two languages. The possible explanation for this could be that some experienced students were interrupting the interventions because of their curiosity in learning the second language. However, interventions that cater to both experts and novice programmers should be further researched.

Half of the participants had neutral feelings about the ATCC concepts experience. While 47% of them found the ATCC interventions helpful. This could be because the lecturer showed a lack of confidence in teaching the ATCC concepts using bridging techniques, especially on Python dictionaries. The lecturer's beliefs and motivations of teaching concepts may affect how the students learn a concept [200]. Another possible explanation could be that the bridging interventions and analogy interventions may be a challenge for students [39, 147, 148]. Such interventions may need careful step-by-step scaffolding activities that connect abstract concepts by showing structural similarities, as shown in Chapter 4. Despite these challenges, the effectiveness of the ATCC concepts is promising.

The students also reported the benefit of a deeper conceptual understanding of some programming language concepts. Making comparisons of multiple conceptions has been reported to deepen conceptual understanding in other studies [142]. Further studies on how teaching second languages can deepen conceptual understanding should be investigated.

RQ4e: Teacher views on teaching using the pedagogy

The lecturer expressed the benefits and the challenges they experienced when using the pedagogy. They reported that getting students' prior language knowledge was beneficial for teaching the second programming language. They also reported that they used transfer interventions to assist the students in learning deeper programming language concepts. Despite the excitement for adopting the pedagogy, the lecture faced some challenges too. The main challenge was the lack of PL1 knowledge. This may have affected their confidence when they had to implement the bridging techniques. This may explain why the students may have been neutral about this intervention. The other challenge the lecturer had was teaching a class with mixed backgrounds. This is already explained in the previous subsection. Possibly, the experienced students intimidated the relative novice programmers by asking advanced programming language concepts during the interventions. These interruptions may have affected some students for whom this pedagogy is designed, affecting their learning process.

The next chapter summarises the thesis findings into a discussion and relates them to prior research, generalise the findings, point out important contributions, implications, and limitations.

Chapter 10

Discussion and Conclusion

Prior research has reported that novice programmers experience transition challenges when moving to new programming languages. Existing research that explains the process of programming language transfer has been conducted on experienced programmers writing programs in a new programming language, giving little attention to how relative novices move from their first or second programming language. To contribute to the existing research on transfer, this thesis investigates how a relative novice programmer's knowledge of concepts in their first PL transfers when learning a second PL. Understanding how relative novice programmers transfer knowledge between programming languages can help account for the success and failure of transfer during second language learning.

Drawing from second natural language acquisition theories, this thesis aims to investigate if semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages. Furthermore, the thesis examines how the transfer interventions in second programming language learning can improve conceptual transfer and understanding. The thesis adopts a mixed-methods design [75] that is conducted in four main stages. The first stage is an exploratory qualitative study to investigate how relative novice programmers transfer knowledge from PL1 to PL2 deductively using natural language theories. Secondly, a Model of Programming Language Transfer (MPLT) based on the preliminary study is developed. Thirdly, the MPLT is validated quantitatively with four experiments. Lastly, transfer interventions and pedagogy for transfer are developed and investigated.

This chapter elaborates on and evaluates the summary of the research findings thoroughly and coherently and discusses the thesis contributions. The chapter begins by presenting the research questions and the experiments that were designed to answer them. Next, the essential findings and the interpretation of results guided by the four high-level research questions are presented. This also includes discussions on how these findings relate to previous studies on PL transfer. These will be followed by a discussion on the thesis contribution to CSE knowledge, the implications, and limitations. Lastly, the conclusions and areas for future research will be presented.

Table 10.1: A summary table of the a total of nine experiments conducted to answer the research questions. These experiments are presented in Chapter 5-9.

Experiment	Chapter	Methods	Research Question
Exploratory Study of Semantic transfer	4	Qualitative	RQ1
Designing Model of Programming Language Transfer (MPLT)	5	Model design	RQ1
Validating MPLT	6	Quantitative	RQ2
Teachers' views on Transfer	7	Qualitative	RQ3
Explicit Interventions using a student-centred Approach	8	Quantitative	RQ4
Pedagogy for Transfer	9	Qualitative and Quantitative	RQ4

10.1 Research Questions and Key Findings

This section discusses the research questions and a summary of the research findings. The detailed discussions are already presented in each individual chapter. These findings will be presented at a high-level and in a coherent manner here.

The thesis statement is that **Semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages; the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.**

This thesis has four high level research questions. Table 10.1 provides a summary of the nine experiments conducted to investigate these research questions. The first set of research questions (RQ1 and RQ2) aim to examine the first part of the thesis statement *"semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages"*.

The first research question is aimed at understanding how semantic transfer in natural languages is relevant in PL transfer.

- **RQ1: How are principles of semantic transfer in natural languages applicable to patterns of transfer in the context of relative novices transferring from first to subsequent PLs?**

An exploratory qualitative study with five second-year undergraduate students transitioning from procedural Python to object-oriented Java was conducted over ten weeks to answer RQ1. The findings in Chapter 4 suggest that semantic transfer did take place, in line with Jiang's model of second language acquisition based on the notion of semantic transfer [25, 26]. The perceived similarities between PL1 and PL2 appeared to catalyze semantic transfer as suggested by Ringbom's cross-linguistic similarity relations [28]. Therefore, if the syntax and semantics of PL1 and PL2 were similar, this accelerated the learning of PL2. However, PL2 learning was affected

negatively if syntax was similar but semantics different. The findings also revealed that students struggled to connect common concepts that look syntactically different between PL1 and PL2. The results also suggest that inappropriate transfer issues remain persistent without explicit instruction. Negative transfer or interference from the first programming language (Python) remained persistent, as observed in week 4 of students transitioning to Java. Furthermore, the students' struggle with connecting common and abstract concepts that are represented differently between the two languages remained persistent, as seen with these students' experiences halfway through learning Java.

Two other transfer issues inductively emerged from this first study findings. Firstly, the transfer does not only occur at the semantic level but can also occur at a conceptual level during code comprehension. That is, these students started by recognizing similarities in the syntax between PL1 and PL2 and then subsequently transferring the semantic (behavior) and conceptual (concept name) knowledge from PL1. Secondly, the transfer can happen bi-directionally (PL1 to PL2 and from PL2 to PL1) as learning progresses, showing students' fragile knowledge of PL1 and PL2 concepts.

The qualitative study yielded results that led to the development of a Model of Programming Language Transfer tailor-made for relative-novices transfer in programming languages. This model is used to investigate transfer in programming languages quantitatively. Therefore the second research question is:

- **RQ2): Do relative-novices transfer their programming language conceptual and semantic knowledge to a new programming language during code comprehension according to the MPLT?**

RQ2 is answered through validating the MPLT hypotheses as follows:

1. If a learner encounters a True Carryover Construct (TCC), they will perceive it as a TCC because of syntax match, hence semantic transfer will be appropriately effected from PL1 to PL2 and the learning will be positively impacted.

2. If a learner encounters a False Carryover Construct (FCC), they will perceive it as a TCC because of syntax match, hence semantic transfer will be inappropriately effected from PL1 to PL2 and the learning will be negatively impacted.

3. If a learner encounters an Abstract True Carryover Construct (ATCC), they will perceive it as novel since there is little or no syntax match, hence there will be little or no semantic transfer from PL1 to PL2.

Four experiments on transfer between procedural Python and object-oriented Java were conducted, see Chapter 6. The first two experiments assessed semantic transfer from procedural Python to object-oriented Java **before** the students learned the Java programming language. The

third experiment evaluated semantic transfer from procedural Python to object-oriented Java **after** the Java programming language instruction. The last experiment assessed semantic transfer from object-oriented Java to procedural Python **after** the Java programming language instruction.

The results show that students encounter three types of semantic transfer based on syntax similarities between PL1 and PL2 as proposed by the MPLT's three construct categories (TCC, FCC, and ATCC). Furthermore, the results show that students who have learned PL2 and those who have not yet learned PL2 have similar transfer patterns as proposed by the MPLT's hypotheses, suggesting that the transfer process happens automatically and unconsciously without explicit instruction. This automatic transfer becomes beneficial when students learn the TCC concepts in PL2. Therefore, as the four studies reveal, the similarity between the two languages can catalyze positive semantic transfer; hence, understanding these concepts becomes faster and easier. These results support the MPLT in Chapter 5 which suggests that students have one mental representation of these concepts in both PL1 and PL2; therefore, the assumption is that if students know a TCC concept well in PL1, they will also automatically know it well in PL2.

The data in all four experiments suggest that the FCC is the most challenging concept to learn in PL2 when students transition from Python to Java. The similarity between the two languages can catalyze negative semantic transfer between PL1 and PL2. In this case, constructs may have similar syntax, but the semantics work differently in PL1 and PL2; hence the automatic semantic transfer triggered upon recognition of similar syntax deceives the students into thinking that the semantics/behavior between the two languages is also the same. The findings support the MPLT hypothesis and show that students hold one mental representation of PL1 and PL2 even when the semantics have changed. This was evidenced in the results when the students indicated that they could transfer knowledge from PL1 to PL2 and bidirectionally from PL2 to PL1. This transfer depended on which language they feel they are more confident within understanding the concept.

Lastly, the results suggest minimal/no semantic transfer on ATCC concepts between PL1 and PL2. Students struggled with understanding ATCC concepts semantics in PL2 because of their unrecognised representation even though they already know them in PL1. Furthermore, if the teacher did not help the students connect common concepts represented differently in PL1 and PL2, the students formed two different mental models of the same concept, as was evidenced in the qualitative study in Chapter 4.

Despite these promising results, it is crucial to note that the categorizations of the three constructs (TCC, FCC, and ATCC) depend on how the researcher perceives similarities between PL1 and PL2; therefore, this might be different for how the students perceive the similarities. Nonetheless, the overall results of the four experiments provided convincing evidence that students from different CS classes, years of study, and different programming language backgrounds followed similar patterns of learning PL2 as predicted by the MPLT hypothesis.

Therefore, the findings show how similarity between programming languages and subsequent semantic transfer plays a significant role in learning new programming languages.

The few researchers who address transfer in programming languages have mainly focused on experienced programmers problem-solving. The studies that validated the MPLT are based on code comprehension activities of PL1 and PL2 by relative novice programmers. This does not mean that similar transfer problems will not be prevalent in code production. Scholtz and Wiedenbeck [19, 20, 22] discovered that when experienced programmers solve problems in new languages, they transfer similar/familiar known algorithms in their previous languages to new languages. They repeatedly change their plans to suit the new language constructs when the languages behave differently, which results in delays in problem-solving. Using familiar algorithms is only helpful to experienced programmers when the two languages are similar, that is, the algorithms can be expressed in the new language as reported in [21].

This thesis is the first attempt at addressing transfer from a code comprehension perspective. Transfer in comprehension is mainly concerned with how a learner perceives syntax similarities between PL1 and PL2 during code comprehension. This type of transfer makes it easier to assess the relative strength of the positive versus the negative effects of transfer [70] in programming semantics and concepts as proposed in the MPLT. Furthermore, this research brings insights into how relative-novice programmers transfer their understanding of conceptual/semantic knowledge as they learn new languages considering that they usually struggle with the surface details/syntax [130] of a programming language as compared to experienced programmers [3].

The first set of research questions' findings supported the thesis claim that semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages. The second set of research questions (RQ3 and RQ4) aims to examine the second part of the thesis statement *"the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language."*

RQ3 aims at investigating transfer from a teacher's perspective. The teachers can give insights into whether high-school students experience the same semantic transfer issues as identified by the MPLT given that this model was developed from observations from university students. If that is the case, the teacher findings can provide insights into whether they adopt transfer strategies that cater to matters associated with the transfer. If the teachers don't use transfer strategies, it is a motivation for this thesis to design transfer strategies that can be implemented in second language learning. Hence the following research question:

- **RQ3: How do teachers typically approach PL transfer for relative novices in the classroom?**

A qualitative study was conducted with 23 K12 teachers in the Netherlands and Scotland to answer this research question. The results revealed that most teachers first introduce students to

simpler visual languages (e.g., Scratch) when they teach them programming. They believe this allows the students to have more confidence and engage in programming. Weintrop's research findings have supported this rationale [112] that reported that students had higher levels of confidence, enjoyment, and interest when using isomorphic blocks-based, text-based, and hybrid blocks/text tools. Other researchers in agreement with these findings are [150, 151] while other studies found that starting with simpler languages reduces students' syntax errors [130].

The findings in this thesis revealed that when teachers transition students from simpler languages to more formal languages like Java/Python, they experience both benefits and problems of transfer from PL1 to PL2. These include positive semantic transfer, negative semantic transfer, and lack of transfer as proposed by the MPLT. These findings suggest that transfer issues are a big problem for both k12 and university students.

Despite the reported transfer challenges, most interviewed teachers did not believe they had to implement transfer strategies in the classroom. They gave reasons such as they do not think students have a strong understanding of concepts in the first language to transfer to a new language. It shows that when teachers teach first programming languages, they do not focus on developing students' conceptual knowledge but rather on engaging students to use the first programming language to create games and be comfortable using programming languages. However, this kind of teaching may defeat one of the most important educational goals, which is the transfer of learning and the incremental development and deepening of understanding. Transfer of learning allows students to use previously acquired knowledge and skills in new learning. This transfer is also essential in learning new programming languages since programming languages share similar underlying concepts. The results of this study also indicate that teachers usually think that transfer occurs naturally to learners, but in reality, without explicit instruction, learners struggle with transferring knowledge from one context to the other [149].

The findings, however, also reveal that some teachers use transfer strategies in the classroom when teaching second programming languages, although some of them use them simplistically and spontaneously. These teachers believe that multiple programming languages help develop students' conceptual knowledge and cognitive skills. Some teachers use well-known transfer strategies such as bridging and explicit instructions analogies to help students build new knowledge through activating previous languages and helping students connect similar concepts between previous languages and new languages.

Although the CS curriculum guidelines recommend more than one programming languages, there is not much research on the teacher's perspective on how students transfer to new languages, hence this study is filling in this gap. Understanding how relative novices transfer semantic and conceptual knowledge and how teachers approach transfer in the classroom can help account for the success and failure of PL transfer in the second PL classroom. The last question, therefore, considers how such an understanding could help in building transfer interventions that can help improve second language learning in the classroom.

- **RQ4: How can transfer teaching interventions based on our understanding of semantic transfer improve second PL learning?**

The thesis presented two quantitative intervention experiments in Norway and Scotland to answer this research question. In Norway, the study implemented transfer interventions with first-year undergraduate students transitioning from object-oriented Python to object-oriented Java. In Scotland, a more embedded pedagogy with three interventions catering for each MPLT category (TCC, FCC, and ATCC) was implemented on second-year undergraduate students transitioning from procedural Python to object-oriented Java.

The results from the two studies revealed that when students are learning TCCs, both implicit and explicit interventions are not very beneficial. Students in the Control group performed similarly to the transfer Intervention groups in both studies in these categories. The interpretation of these results supports the proposal by the MPLT. According to the MPLT, relative novice programmers hold one mental representation (similar conceptual, semantic and syntactic representation) of the TCC concepts in PL1 and PL2. PL2 is acquired automatically by matching similar syntax of PL1 and PL2 followed by subsequent semantic transfer. It, therefore, means that when students know a concept in PL1, they will also know it through implicit learning in PL2. In this case, the findings from these studies confirm that teachers can use this implicit learning of the TCC and use implicit teaching methods [185] to teach TCCs in PL2. Students are expected to learn the PL2 semantics and concepts in implicit teaching unconsciously. Teachers do not need to make the students consciously focus on the PL2 form and explain and make comparisons of PL1 and PL2 in the TCC concepts. Implicit learning mechanisms automatically assist in learning these concepts. The pedagogy, however, does not address the TCC misconceptions that students already have in PL1; therefore, should consider careful steps in addressing these.

The results showed that the explicit interventions were considerably more effective on the FCC categories than other MPLT categories. Students in the Intervention group outperformed those in the Control group in both studies after explicit instruction. Furthermore, the results suggest that explicit student-focused or instructor-focused interventions are beneficial to students' learning of the FCC concepts. The lecturer helped the students become conscious of the PL2 differences during explicit instruction by providing explicit descriptions of the underlying semantics and concepts to be learned in PL2. It could imply that the students' meta-cognition abilities were engaged in the learning process when explicit instruction was applied. Meta-cognition can be defined as thinking about thinking [83]. It is becoming a common topic in computing education research [152]. Some studies have reported the benefits of meta-cognition scaffolding. For example, they discovered that students can misinterpret a problem statement and form an incorrect mental model of the problem which they may find difficult to correct. After meta-cognition scaffolding, students then experience significantly fewer errors that are related to the formation of an incorrect mental model when working on programming problems

after meta-cognition scaffolding [153].

Furthermore, some students who used the instructor-led interventions also reported a deeper conceptual understanding of programming language concepts after the interventions. This could be because the lecturer helped students learn by comparing PL1 and PL2 and giving students feedback on their misconceptions caused by the transfer. Recent CSE research by Margulieux and colleagues [142] proposed that multiple conceptions develop better conceptual knowledge when the students are guided to compare the multiple conceptions of a concept during instruction (in this case, comparing PL1 and PL2). The explicit instructions also assisted the students to modify and correct their existing conceptual knowledge as proposed by the MPLT [30] and other cognitive studies [45, 185].

Lastly, the results revealed that the students in the Intervention group scored higher marks in all the ATCC concepts. Still, only half of the ATCC concepts scores differed significantly between the Control and Intervention groups. It implies that the bridging [45] transfer interventions were most helpful in helping students connect concepts between PL1 and PL2. However, they were not the easiest to implement in the classroom for both the lecturer and the students. There are several possible explanations for this result. The lecturer was not confident in connecting PL1 and PL2 because they had limited PL1 knowledge. This could mean that this affected students' learning of these concepts. Prior research reported that the teacher's self-efficacy has an impact on students' behavior, learning, and achievement [154, 155, 190]. Therefore, these results imply that for second programming language teachers to help their students transition effectively, they should have knowledge of PL1 and PL2.

Furthermore, the challenge in implementing the bridging interventions for abstract concepts is also reported in prior computing education research [39, 147, 148]. For example, Dann et al. [10] used bridging techniques to help students transition from Alice to Java. This was done by allowing students to view equivalent Java code with greater syntax details on the same screen in both Alice and Java. The results showed improvement in the overall scores of the Intervention group as compared to the Control group. The ATCC interventions need careful step-by-step scaffolding activities that also show structural similarities in connecting abstract concepts [149], such as in the ones implemented in the exploratory study in Chapter 4 .

10.2 Emerging Findings and Contributions to the Larger Field of CSE Research

The previous section presented the summary findings from the four research questions of this thesis. This section discusses the broader findings that emerged and extended beyond the research questions. It discusses these findings relating to the larger field of research. The research on transfer was conducted over three years with different teachers and student cohorts at different universities and countries. Some findings emerged related to the broader previous and future

CSE research. This section focuses on contributions of this thesis on three important topics of CSEd: Relative Novices' Fragile Knowledge, improving conceptual understanding through multiple conceptions theory , and contributions to modern CS curriculum.

10.2.1 Relative Novices' Fragile Knowledge

This work gives insights into how fragile knowledge affects the mental models of novice programmers when they transfer to new programming languages. According to Perkins and Martin, fragile knowledge is "knowledge that is partial, hard to access, and often misused" [156]. Students' fragile knowledge has been reported in other computing education research that agrees that relative novices have delay in the development of viable mental models of program execution [3, 49–52]. Perkins and Martin [156] describe four types of fragile knowledge of basic programming language commands as follows: Partial knowledge, Inert knowledge, Misplaced knowledge and Conglomerated knowledge. In this section, three types (Partial, Inert, and Misplaced) of the fragile knowledge will be discussed in relation to how they relate to this thesis findings on how students transfer between programming languages and how this understanding can advance the knowledge about fragile knowledge in computer science education.

Perkins and Martin describe *Partial knowledge* as the type of fragile knowledge that the student has not been able to retain or has never learned [156]. This thesis discovered that students have *Partial knowledge* of programming language concepts in their PL1 that was not very firm and that can be easily changed when they transitioned to PL2. It was mainly observed in the studies in Chapter 6, when students were transitioning from Java (PL1) to Python (PL2). The findings revealed that students seemed to have a bi-directional transfer in the FCC categories. They transferred their knowledge of Java to Python and Python to Java in different concepts. Regardless of where the transfer came from, students held one mental representation (concepts, semantics and syntax) of the two languages even when the semantics had changed, but the syntax was similar; see the MPLT in Chapter 5.

When the transfer happens from PL2 to PL1, this could imply that the students were not able to retain this knowledge in their PL1, or it could mean that they learned it properly first in PL2. For example, in experiment 4 in Chapter 6, participants had the correct Python (PL2) execution model for the variable reassignment, with 91% of them getting it right. Still, most of them carried the suitable Python variable reassignment model across to the Java code showing that they base their interpretation on syntax similarities between Python and Java. It could imply that the students never learned properly about static typing and variable declaration in their Java language (PL1) in such a way that they could retain the knowledge. It made their knowledge susceptible to change when they learned PL2.

When students know two programming languages, their understanding of programming language concepts is heavily based on the language that they understand better. They effect semantic and conceptual transfer to the language they partially know based on syntax similarities. In

Chapter 9, some students also expressed that they were learning or getting exposed to some concepts in their PL2 and they did not have a firm grasp of them in PL1; for example, one student said *"We did not get exposure to the concept of referencing in Python (PL1), we heard about it but did not see much of it in practice until we learned Java (PL2)"*. It should be noted that these are the concepts that the students had covered in their PL1 though.

The thesis also contributes by showing that fragile knowledge in the first programming language may be caused by the teacher not being explicit about concepts when teaching the first programming language. Most teachers interviewed in this research revealed that when they teach PL1, they focus on students' engagement and enjoyment rather than teaching them deep fundamental programming language concepts.

Perkins and Martin describe *Inert knowledge* as the knowledge that the student has but fails to retrieve [156]. Belmont and colleagues describe this knowledge as the knowledge that has been acquired in one context but fails to bridge to another context or application [157]. In the context of programming languages, an example of inert knowledge would be when a student cannot solve a programming problem, but the teacher gives them a hint, they solve it successfully which shows that initially they failed to access the already existing knowledge [156].

This type of knowledge is evidenced in this thesis when students learn the ATCC concepts in PL2. A great example is the students' failure to transfer their knowledge of *object aliasing* from Python (PL1) to Java (PL2), see Experiment 3 in Chapter 6. The 63% students who knew the concept of aliasing in the context of Python dictionaries failed to transfer that knowledge in Java object aliasing. It was because the syntax of Java objects and Python dictionaries look different such that the students cannot connect the two concepts as similar concepts. This thesis contributes significantly by showing how inert knowledge inter-plays with transfer in learning new programming languages. It shows the role that similarity between PL1 and PL2 plays in transferring otherwise inert knowledge.

Usually, teachers expect students to transfer knowledge from one context to the other naturally [149,158]. This view was also expressed by teachers interviewed in this thesis in Chapter 7, where they expressed that they expect students to transfer quite naturally to new languages. This kind of thinking can be true for the TCC concepts. However, usually, for abstract concepts, the transfer may not be easy, as has been demonstrated in the ATCC category in this research.

To deal with this issue of inert knowledge, the thesis proposed a pedagogy that uses bridging techniques to help students transfer this type of knowledge in the ATCC category. When using this strategy in the exploratory study, students were seen to have *Aha!* moments as they retrieved their inert knowledge about data structures in PL2. An example is after the researcher gave students a hint to access their already existing knowledge by associating Python dictionaries to Java objects, one student said *"When you say the word object, it seems like it's this abstract thing. I think now I see an object as an advanced way to store data and give you options to pass it around like lists and dictionaries with techniques to manipulate the data"*. It shows that the

student managed to access the knowledge that was initially not accessible to help them to help them comprehend a the Java language.

Misplaced knowledge is the kind of knowledge about commands that may be applicable in other contexts, but the students use it in lines of code it doesn't belong [156]. In the context of this thesis findings, this occurs when students are learning FCC concepts in PL2. Students were reported to know certain concepts in PL1 but inappropriately transferred the semantics of these concepts to PL2. No research focuses on how negative transfer occurs during code comprehension in programming languages. The assumption is that it's a trivial problem in programming languages as most CS instruction focuses on computational thinking and problem-solving. Instructors fail to understand that the persistence of negative semantic transfer exposes the students' fragile knowledge of programming language concepts and semantics. This thesis highlights some of the causes of misplaced knowledge in programming languages. Firstly, students automatically transfer semantic knowledge from PL1 to PL2 based on syntax similarities. Secondly, when they learn PL2 and the instructor is not explicit about the changed behavior of a concept, students' misplaced knowledge remains persistent. This thesis contributes by showing how explicit interventions that refer back to PL1 are beneficial to help students correct their mental models and deepen conceptual understanding.

Overall, what can be said about the implications of the discovery about the interplay between fragile knowledge and semantic transfer in learning new programming languages? As already mentioned, the traditional CS classroom emphasizes early computational thinking and problem-solving exercises with an assumption that students have sufficient prior semantic and conceptual knowledge to generate new learning constructively [115,145,146]. This approach can lead a student to believe that the most important achievement in a first programming course is to problem solve; hence they use trial and error methods to get the program working [159]. Furthermore, teachers and students may think that passing problem-solving assessments signify that the students have acquired sufficient conceptual knowledge. The research, however, has shown that even though students can build programs, they do not understand them sufficiently [53].

Furthermore, other researchers [159] have argued that writing programs too early in teaching programming languages may delay students' development of viable mental models of program execution, given that their understanding of the programming language (e.g., syntax, semantics/notional machine, concepts) is still very fragile [3, 49–52]. This argument supports the thesis findings on the challenges students face when transferring to second programming languages. First the thesis has explicitly demonstrated that upon completion of their PL1 course, students still have fragile knowledge. The students' fragile knowledge results in it being misplaced (FCC), not used (ATCC), and susceptible to change as they transfer to new languages. This thesis offers contributions by showing how different kinds of transfer interventions can assist in resolving various types of fragile knowledge in a novice programmer. Furthermore, it shows how second language learning can be an opportunity to fix fragile knowledge and deepen

conceptual understanding as shall be explained next.

10.2.2 Deepening Conceptual Understanding through Second Language Learning

The previous section has looked into how CS students have fragile knowledge of programming language concepts. This section follows on to discuss how second language learning can be used as an opportunity to build and deepen students' conceptual knowledge.

Conceptual knowledge can be explained as the understanding of the principles and relationships that underlie a domain [160, 161] or knowing why [162]. In the context of programming languages, Margulieux and co-authors describe an example of conceptual understanding as [142] *"understanding why different **data types** are used and how to use them appropriately to solve a problem"*. Conceptual knowledge can be referred to as deep learning and can result in better retention and transfer of knowledge [142]. Margulieux and colleagues [142] proposed a theory of multiple conceptions that describes that learners develop better conceptual knowledge by comparing multiple conceptions of a concept during instruction. Guided by the literature, they propose five teaching design principles for creating activities that guide students to compare multiple conceptions and develop conceptual knowledge. These are vicarious failure, self-explanation, inductive reasoning, conceptual growth and conceptual change. Some of these principles will be discussed in relation to how they relate to this thesis's findings and contributions.

- **Vicarious failure:** Vicarious failure deals with exposing students to common non-correct conceptions so that they can be helped to form correct conceptions and help them restructure their conceptual network correctly. This method has been proven to improve students' conceptual understanding [163]. This principle was applied in this thesis in the pedagogy for transfer, see Chapter 9. The implementation of the principle was at the first stage of the pedagogy, the **predictive stage**. Students were given guess quizzes in Java (PL2) even before learning the Java language. It resulted in them producing failed attempts at the Java code comprehension activities of the FCC and the ATCC constructs due to the negative transfer and lack of transfer, respectively. Some studies have empirical evidence to show that engaging students on tasks they have not yet learned first (e.g. guess quiz in this context) can be beneficial [163, 164]. Usually, students fail, just like in this guess quiz. Then this provides an opportunity for instruction to provide corrective feedback, which becomes beneficial for the students' conceptual learning. After the intervention, the results suggest that students learn better from their own failed attempts or solutions as one student said in Chapter 9, *"I think it's more useful to point out the differences since Python is something most of us are already familiar with and often use this experience as a "fallback" when we are not sure about how something works. Therefore, it is more useful to know when*

this will not produce the expected results rather than when it will". This means that vicarious failure is a principle useful for second language learning and it is well suited in the pedagogy for transfer and its effort to improve conceptual understanding.

- **Self-explanation**: This principle concerns learners being presented with multiple examples that allow them to compare and reason about the key concepts and reconcile the difference between them. This technique then helps the students to develop their conceptual knowledge's mental connections. This thesis shows how this principle applies to learning a second programming language. The principle was used throughout the pedagogy of transfer specifically in two interventions. The students were given both the Python and Java equivalent code fragments during the interventions for FCC and ATCC. The first FCC interventions included explicit transfer interventions of **compare and correct**. Giving students these interventions can help them correct their *misplaced fragile knowledge*. It means that the students were given an intervention to correct their mental models that are caused by the negative semantic transfer. It was done by comparing Python and Java and being given corrective feedback—in this study, comparing and correcting allowed students to hold two mental representations of the semantics in two languages for the same concept. This research contributes by showing that this principle, if applied in pedagogy, it helps the students to *restructure* the mental model. For example, one student said *"In Python, you do not have to declare the type of a variable, I thought in Java I could just say a="abc" but in Java, I have to be specific and say String a="abc"*. It suggests that initially, the students viewed PL1 and PL2 behaving the same way, and later, after corrective feedback, the student now has two different semantic representations for the concept of *declaration* in their mental model.

  The second intervention that adopted this principle included Bridging transfer interventions of **compare and connect** on the ATCC concepts. The Bridging transfer techniques are very similar to the analogical transfer interventions [45, 142, 165]. Giving students analogies can help them transfer the *inert knowledge* that has failed to transfer. During the Bridging techniques for this study, students were given Python dictionaries and Java objects and shown that they have the similar underlying meaning of a data structure even if they are represented with very different syntax in two languages. This approach helped the students have a deeper conceptual understanding of data structures. For example, in the exploratory study, one student initially said *"I think objects its the most confusing part. I'm still not a 100 percent sure what they are"*. After the Bridging technique, the student appears to restructure their conceptual mental model and connect it to their existing knowledge of data structures. Their thought pattern changed, and they said *"I think now I see an object as an advanced way to store data and give you options to pass it around like lists and dictionaries with techniques to manipulate the data"*. This intervention seems to have been more successfully implemented in a small exploratory study with

five students in Chapter 4 as compared to a large size study in Chapter 9. It could be that Bridging techniques for a larger classroom size were a challenge. Also, the lecturer was not competent in one of the languages to make it a fully successful intervention. However, the results indicate that Bridging techniques can be used in second language learning to deepen conceptual understanding.

- **Conceptual growth**: This principle is concerned with deepening students' conceptual knowledge by explicitly identifying gaps in their knowledge. Margulieux and colleagues suggest that this can be done by interventions that include testing, giving students examples they can not yet explain to show their knowledge limitations. This principle was adopted in the pedagogy for transfer by giving students pre-quiz in both languages at the beginning on the PL2 lesson to identify missing or gaps in their existing knowledge. Where, gaps were identified, the pedagogy adopted Explicit (for FCC) and Bridging (for ATCC) techniques to help fill the gaps of missing knowledge or *incomplete knowledge* as proposed by Perkins and Martin. This approach improved the students' conceptual understanding as one student said in the intervention study in Chapter 9, *" I was introduced to scoping in my Python course but I did not really have a strong grasp of it. The Java lecturer wrote sample code and common pitfalls using the Java-Python survey examples and showing global variables and local ones which made it easier to avoid mistakes."*

- **Conceptual change**: The last principle from Margulieux and colleagues that is applicable in this thesis is the Conceptual change principle. They propose that identifying misconceptions from prior knowledge can help in conceptual change. It can tackle the problem of students' fragile knowledge (*misplaced knowledge*). In the context of this research, as already explained in previous points, the pedagogy for transfer adopted *compare and correct* techniques when students encountered FCC in the new language. The findings from the studies in Chapter 4, Chapter 8, and Chapter 9 showed that this approach was effective at prompting conceptual change for students who initially were affected by the negative semantic transfer.

This section presented arguments about how second language learning can deepen or improve conceptual understanding. This was done using Margulieux and colleagues' principles of multiple conceptions theory, which proposes that students develop better conceptual knowledge when guided to compare multiple conceptions of a concept during instruction. The thesis contributed to understanding the use of second programming language learning to improve conceptual understanding. Furthermore, the pedagogy for transfer offers intervention strategies aligned with the MPLT and shows the effectiveness of adopting the principles of multiple conceptions. Three chapters in this thesis empirically validated the interventions.

.

10.2.3 Multiple Programming Languages in the Curriculum

This section shows how the thesis contributes to the modern CS curriculum. The professional societies in computing (ACM and IEEE-Computer Society) have provided guidelines for undergraduate CS programs for the past decades, starting with the curriculum developed in 1968 [80]. The early 1968 curriculum had appeared to have more focus on the understanding of programming languages (e.g., implementation, language design, etc.) [166] as compared to computational thinking and algorithms. As the evolution of programming languages progressed and found its way into the curriculum [1], there was a need to create space for new languages and paradigms in the curriculum [131]. The 1978 curriculum introduced the CS1 (Computer Programming 1) and CS2 (Computer Programming 2), with the aims of CS2 building on the students' knowledge in CS1. The curriculum designers proposed that it may be necessary to introduce students to a second programming language [167]. Furthermore, the curriculum emphasized students acquiring knowledge of problem-solving or algorithm development and programming languages knowledge. The subsequent CS curriculum guidelines for CS undergrad followed a similar pattern [81] with the 2013 curriculum guideline acknowledging the use of different paradigms in the university curriculum (object-oriented programming, others functional programming, others platform-based development). These guidelines are flexible, and any university can customize them to what works best for them.

The modern CS curricula guidelines result in CS students rapidly and constantly switching between different programming languages in their education. At the University level, the programming language courses have sequences such as CS0-CS1-CS2. CS0 is the introductory course with no programming language background knowledge, while CS1 and CS2 are the first and second required courses in the sequence, respectively. After completing CS0, students are usually later on required to take CS1 or/and CS2, which mainly teach programming with different object-oriented languages like Java and C++. [168]. Some CS0s use visual programming language while some use text-based languages to transition students to CS1 and later to CS0 [169–171]. The curriculum at primary and secondary schools (K12) also follows this approach of teaching different programming languages, for example, in the UK [172].

As much as different programming languages introduced in the curriculum look different, oftentimes the underlying principles are the same, and the difference is usually the superficial syntax [5, 173]. For example, the approach of procedural languages may differ from object-oriented languages, or the approach for logic programming may differ from functional programming languages; however, seeing the underlying similarity between them may be helpful [173] to educators and students. The early comparative languages courses, were introduced in CS to find common ground between programming languages and the underlying principles [173]. Although teaching comparative languages could have had the right reasons for deepening conceptual understanding in mind [131, 173], there is no empirical evidence to show that the approach was successful.

Here, the focus of the discussion is shifted back into this thesis and how it can be situated in this big broad topic. As already discussed in the previous section, the introduction of multiple programming languages in the curriculum can be seen as an opportunity to deepen the students' conceptual knowledge. The current curriculum offers the opportunity to gradually teach programming concepts using different programming languages, hence provides several advantages. First, unlike in the early comparative programming languages, where students were taught two or more languages simultaneously, the modern curricula allow for the gradual introduction of programming languages. For example, one programming language can be taught (CS0/1) for at least a semester before shifting to the following programming language (CS2) for another semester. This gradual introduction allows teachers to adopt different transfer strategies per course. This thesis has revealed that students have fragile knowledge of the first programming language; this is evidenced in a lot of prior work. The thesis has shown that this fragile knowledge affects successful transfer to a second programming language. Therefore, it would be beneficial for first programming language teachers to start thinking about ways to help students gain the conceptual understanding that can be helpful for transfer.

Secondly, this thesis has revealed different types of knowledge affected by transfer that the students bring into the second programming language course. Therefore, second language teachers can build on this prior knowledge and correct students' negative transfer as well as improve conceptual understanding. It contributes to CSE by developing a pedagogical guideline for teaching second programming languages that the teachers can adopt for their classrooms.

Overall, the thesis brings the educators and the CSE community to shift the focus back to the important knowledge structures of underlying programming language concepts that they should build on the students. The focus should be less on debating which programming languages/-paradigms are better than others. The issues raised here are discussed further in the Implications section.

10.2.4 MPLT in other Programming Languages Contexts

This research developed a MPLT that has been validated in the context of relative novice programmers transferring from Python to Java and bidirectionally from Java to Python. Although Python and Java programming languages have some differences, they are closely related as imperative languages and share similar syntax, semantic and conceptual features hence they may share a lot of TCC and FCC and less ATCC.

In the transition from a block-based programming language, like Scratch, to a text-based language, like Python, the two languages share core underlying computational concepts if one looks past the blocks. The Scratch IDE comes with a library of pre-built sprites, which may appear as a library in another language like Python (e.g., a Python Turtle library). The MPLT has been used in different PL contexts, like in a recent study by Espinal et al. [210] which used the MPLT to analyze their studies of transfer between block-based PL and text-based PL and from

Spanish block-based PL to English text-based PL. They concluded that transfer from Makecode (block-based) to Scratch (block-based) occurred mainly as TCC. They also concluded that transfer from a Spanish translation of Makecode to an English Python programming language was ATCC. Interestingly, as programming languages like Scratch are being translated into other languages (e.g., Norwegian), further exploration of transfer using the MPLT becomes an area of interest.

Evidence of support for the MPLT has also been reported in prior work discussed in Chapter 2. For example, Santos et al. [39] reported experiences with their students transitioning from Java to Racket. Students struggled with Java concepts that do not work the same way (e.g., string concatenation, type systems), which may be FCCs, and struggled with concepts that look different (e.g., Racket struct to Java class), which may be ATCCs.

Exploring the MPLT in programming language paradigms that are very different may present opportunities to explore different scenarios that are currently not in the MPLT, as explained in section 5.2. For example, these languages may have concepts that have similar syntax but different semantics and different conceptual roots or concepts with different syntax and different semantics. The current research is limited to Python and Java as imperative programming languages. However, functional languages like Haskell may present the unknown in terms of conceptual transfer and may require further research.

It should be noted that the MPLT was studied in the context of relative novice programmers who only knew one programming language before learning a second one. Therefore the results of this thesis cannot be generalizable to the whole population of people learning new programming languages from different knowledge backgrounds and experiences. This opens room for further research to explore the MPLT in other contexts of the student population.

The next section discusses the implications and the future work.

10.3 Implications and Future Work

This thesis developed and validated the first theoretical model (MPLT) about relative novices' transfer based on code comprehension which has implications to CSE community. Secondly, this thesis developed the first pedagogical guideline of second programming language learning based on this theory (MPLT) which had pedagogical implications to the CSE community. Lastly, the thesis explored teachers and their experiences in second language learning which can have pedagogical implications too. This section presents specific recommendations to teachers, researchers, and curriculum designers in CSE.

Implications for Teaching

It is recommended that the programming language teachers should always consider their students' prior programming language knowledge and design classroom activities that engage their

initial understanding to help them evolve their conceptual knowledge as they learn new programming languages. The thesis provides guidelines that can help teachers develop assessments that aim at collecting students' prior knowledge in PL1 using the MPLT three categories, see Chapter 6. The PL2 guess quizzes that have matching concepts as the PL1 assessments are also necessary to implement the pedagogy of transfer. The recommendations to second programming language teachers based on the MPLT and the pedagogy are as follows:

- The studies that validated the MPLT have revealed that TCC is the most straightforward and easy concept to transfer as reported in Chapter 4, Chapter 8 and Chapter 9. It is because they have similar syntax and semantics; hence syntax matching and semantic transfer positively affect the students' learning in a new language. The teachers can take advantage of the knowledge students already have on these concepts to accelerate teaching a PL2. In other words, the teachers can use the implicit methods of teaching a second language in this category, as students are helped by their implicit learning to automatically learn this concept in the second programming language.

- The studies in this thesis revealed that the FCC concepts are the most difficult concepts to transfer between programming languages because they have similar syntax and different semantics; hence, semantic transfer negatively affects the learning of these concepts. The FCC usually leads the students to have misplaced knowledge used in the wrong contexts in both languages, primarily PL2. Teachers are encouraged to use explicit interventions of transfer that **compare** the two programming languages and **correct** student's misconceptions caused by the negative semantic transfer. Teachers should explain to the students that the languages behave differently; they also need to explain why this is the case. Giving further explanations helps students understand concepts better and restructure their mental models. An example of such an intervention can be when a teacher explains the concepts of a for-loop in Java vs. Python. In comparing these concepts, the teacher can teach deeper concepts such as iterator-based vs. index-based for-loops, deepening students' conceptual understanding. Enough practice exercises must be given to ensure that both concepts are deeply embedded. The work has also shown that these interventions are successful in both student-centered and teacher-centered approaches. However, the teacher-centered approaches seem to help deepen conceptual understanding better. These interventions are supported by the principles of multiple conceptions as well.

- ATCC was also reported to be more challenging for students shifting from PL1 to PL2. This lack of transfer caused by the lack of similarities between PL1 and PL2 on the same concepts leads to inert knowledge. In the context of this thesis, students experience difficulties when they move from a concrete representation (e.g., Python dictionaries) to an abstract representation (e.g., Java objects). To assist semantic and conceptual transfer in these categories, instructors can adopt Bridging techniques that use **compare** and **connect**

and show students how common concepts that are represented differently have similar underlying semantics. These strategies require the teacher to have a deeper conceptual understanding of concepts. They were reported to be the most challenging to implement by the lecturer who used them. Therefore, careful scaffolding from abstract to concrete representation must be carefully designed. These interventions are supported by the principles of multiple conceptions as well.

Implications to Curriculum Design

The modern CS curriculum encourages more than one programming language for universities and high schools. The aims of any curricula should be to facilitate transfer and learning progression and development of conceptual understanding. The introductory to programming languages sequences such as (CS0-CS1-CS2) should be used as an opportunity to deepen students' conceptual knowledge. As revealed in the teacher interviews, some teachers teach multiple programming languages to follow what the curriculum recommends. It means that teachers will aim to implement whatever is included in the curriculum. This suggests that the CS curriculum designers should use these thesis findings and transfer pedagogical guidelines as guidance to design teaching material that promotes transfer as students transition between programming languages. This thesis has also revealed that if the second language teacher does not teach the first programming language course, they may struggle with transfer interventions because of a lack of knowledge. The teachers don't necessarily have to teach the first programming course (using PL1), but they need to know the first programming language and possibly how it was taught. Therefore, teachers should participate in a professional development program on programming languages and transfer interventions that cover the value of teaching multiple PLs to develop conceptual knowledge, the benefits and challenges of second language learning, and teaching techniques for transfer.

Implications to Research

This research has opened several avenues for further research in computing education. The thesis has shown that initial investigations into the MPLT and transfer pedagogy can help improve second language learning. First, the PL designers, educators, and researchers have to explore categorizing other programming languages and paradigms into the MPLT categories of TCC, FCC, and ATCC. Such research outcomes can help guide educators who teach different programming languages. Secondly, the pedagogy for transfer has not been explored using other programming language contexts. Also, it has not been studied in the context of high schools, which generally would need interventions to transfer block-based to text-based languages. Furthermore, research can be carried out on how students with a broader range of experiences such as age, from different disciplines, and multiple PL knowledge learn subsequent programming languages. In-depth experiments for these suggestions above would be a valuable next step.

Furthermore, further research on using second language learning to deepen conceptual understanding would be useful. Researchers can carry out future work on transfer with teachers who already consider transfer strategies essential and see the benefit of teaching second programming languages. Suppose the recommended research outcomes prove to be a success. In that case, the findings can be used to design the CS curriculum that already includes multiple programming languages and can also underpin broader teacher professional development programs on transfer. Analyzing insights and views collected from researchers is also an important step toward improving second and subsequent programming language teaching.

10.4 Study Limitations

This thesis provides valuable insights into the process of programming language transfer for relative novices. It also gives insights into the effectiveness of transfer interventions tailor-made for types of transfer as proposed by the MPLT. Although it reports on promising results, it has limitations in the context of the claims made. This section reviews the limitations and discusses how they can be addressed in the future.

The first limitation concerns categorisations of the MPLT (TCC, FCC and ATCC). These categorisations depend on perceptions of language similarities by the researcher and may not always generalise to how students perceive similarities. There is currently no formal way to measure similarity in PLs hence follow-up studies should try to provide a formal similarity metrics that can form as a guideline for such categorisations. Furthermore, the categorisations were limited to two programming languages (Python and Java) and did not address all known programming language concepts. Further research on how these categorisations is applicable to other programming language contexts would be very beneficial for this research.

The second limitation was that the research was carried out for the first few weeks of a second programming language course. Therefore, this made it difficult to assess the long-term effects of transfer. All the transfer interventions for this study were carried out in real-life classrooms. Real-life classrooms come with several restrictions on the researcher on how many times they can be allowed inside the same classroom to collect data. Therefore, to have as much minimal disturbance as possible on the regular running of the course, this research was limited to only the first few weeks of students learning the second programming language course. There were no long-term transfer assessments in this research. Therefore, this question remains open and needs to be included in further study.

The third limitation was that the lecturer did not fully enact the intended transfer pedagogy. It, again, is because of the limitations that come with real-life classroom experiments: First, the lecturer may have considered the experiences of their students in their course; hence, they choose only elements they are confident with from the intended pedagogy. Second, the diverse students also caused a disturbance and distracted the class during some transfer interventions. Last, The

teacher had limited knowledge of the students' first programming language, which may have negatively affected how she adapted the pedagogy, especially for the Bridging interventions. Future research should consider how this pedagogy can be adapted to suit a diverse group of students with different PL backgrounds. Future work should develop and train teachers on programming languages and transfer interventions.

The fourth limitation of the research was that the teachers who participated in the study were the teachers who were mostly very experienced and who were already interested in CS research. It means that the teacher's transfer experiences in the classroom may not give a clear picture of diverse teachers, e.g., those with limited experience. Future studies would consider a more varied representation of teachers.

10.5 Conclusions

The claim of this thesis is twofold, as reflected in the thesis statement. The first claim is that *semantic transfer based on syntax similarities plays a role in relative novices' conceptual transfer between programming languages*. The second claim of the thesis statement proposes that *the implementation of deliberate semantic transfer interventions during relative novices' second language learning can lead to improved conceptual transfer and understanding in learning a second programming language.*

This thesis explores this claim by investigating how students transfer between programming languages (Python and Java) during code comprehension for a period of three years that involved nine experiments. This work is the first that designed a Model of Programming Language Transfer for relative novice programmers that is based on code comprehension. Through validating the model, the thesis concludes that similarities between programming languages play a significant role in semantic and conceptual transfer between programming languages. This thesis also shows how the MPLT was used to shape the design of a transfer pedagogy in the classroom. It revealed how the transfer interventions proposed by the pedagogy of transfer can lead to improved conceptual transfer and understanding.

This thesis, therefore, contributes to the understanding of relative novices' transfer mechanisms and offers a systematic approach or guideline to investigate programming language transfer for relative novices that can be adopted in different programming languages. Furthermore, it offers a pedagogical guideline of how to teach a second programming language to improve the transfer and understanding of programming language concepts. The discoveries in this work have come at the right time where it can provide the much-needed solution to the current dilemma of teaching of multiple programming languages in schools. Furthermore, teaching and learning models have become more important as CS moves from university-industry teaching to mainstream. The mainstream model means that CS education caters for more teachers and learners of different abilities.

Bibliography

[1] Mark Guzdial and Benedict du Boulay. The history of computing education research. *The Cambridge handbook of computing education research*, 2019:11–39, 2019.

[2] Amjad Altadmri and Neil Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. *SIGCSE 2015 - Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.

[3] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[4] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[5] Shriram Krishnamurthi and Kathi Fisler. Programming paradigms and beyond. *The Cambridge Handbook of Computing Education Research*, 37, 2019.

[6] Mark Guzdial. Teaching two programming languages in the first cs course. *On the Internet at https://cacm. acm. org/blogs/blog-cacm/228006-teaching-two-programming-languages-in-the-first-cs-course/fulltext (visited August 2019)*, 2018.

[7] Michael Kölling, Neil CC Brown, and Amjad Altadmri. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, pages 29–38. ACM, 2015.

[8] David Weintrop and Uri Wilensky. How block-based, text-based, and hybrid block/-text modalities shape novice programming practices. *International Journal of Child-Computer Interaction*, 17:83–92, 2018.

[9] Deborah J Armstrong and Bill C Hardgrave. Understanding mindshift learning: the transition to object-oriented development. *MIS Quarterly*, pages 453–474, 2007.

[10] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. Mediated transfer: Alice 3 to java. In *SIGCSE*, volume 12, pages 141–146. Citeseer, 2012.

[11] H James Nelson, Gretchen Irwin, and David E Monarchi. Journeys up the mountain: Different paths to learning object-oriented programming. *Accounting, Management and Information Technologies*, 7(2):53–85, 1997.

[12] Karen P Walker and Stephen R Schach. Obstacles to learning a second programming language: An empirical study. *Computer Science Education*, 7(1):1–20, 1996.

[13] Johannes Holvitie, Teemu Rajala, Riku Haavisto, Erkki Kaila, Mikko-Jussi Laakso, and Tapio Salakoski. Breaking the programming language barrier: Using program visualizations to transfer programming knowledge in one programming language to another. In *2012 IEEE 12th International Conference on Advanced Learning Technologies*, pages 116–120. IEEE, 2012.

[14] Vikki Fix and Susan Wiedenbeck. An intelligent tool to aid students in learning second and subsequent programming languages. *Computers & Education*, 27(2):71–83, 1996.

[15] Divna Krpan, Saša Mladenović, and Goran Zaharija. Mediated transfer from visual to high-level programming language. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 800–805. IEEE, 2017.

[16] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. From scratch to "real" programming. *ACM Transactions on Computing Education (TOCE)*, 14(4):25, 2015.

[17] David Weintrop and Uri Wilensky. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education*, 142:103646, 2019.

[18] Jean Scholtz and Susan Wiedenbeck. Adaptation of programming plans in transfer between programming languages: a developmental approach. *Empirical studies of programmers: Sixth workshop*, 233, 1996.

[19] Jean Scholtz and Susan Wiedenbeck. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, 2(1):51–72, 1990.

[20] Jean Scholtz and Susan Wiedenbeck. Using unfamiliar programming languages: the effects on expertise. *Interacting with Computers*, 5(1):13–30, 1993.

[21] Quanfeng Wu and John R Anderson. Problem-solving transfer among programming languages. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY . . . , 1990.

[22] Jean Scholtz and Susan Wiedenbeck. Learning a new programming language: a model of the planning process. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume 2, pages 3–12. IEEE, 1991.

[23] Jean Scholtz and Susan Wiedenbeck. An analysis of novice programmers learning a second language. In *PPIG (1)*, page 9, 1992.

[24] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[25] Nan Jiang. Lexical representation and development in a second language. *Applied linguistics*, 21(1):47–77, 2000.

[26] Nan Jiang. Semantic transfer and its implications for vocabulary teaching in a second language. *The modern language journal*, 88(3):416–432, 2004.

[27] Nan Jiang. Form–meaning mapping in vocabulary acquisition in a second language. *Studies in Second Language Acquisition*, 24(4):617–637, 2002.

[28] Håkan Ringbom. *Cross-linguistic similarity in foreign language learning*, volume 21. Multilingual Matters, 2007.

[29] Ethel Tshukudu and Quintin Cutts. Semantic transfer in programming languages: Exploratory study of relative novices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 307–313, 2020.

[30] Ethel Tshukudu and Quintin Cutts. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 227–237, 2020.

[31] Ethel Tshukudu and Quintin Cutts. Understanding conceptual transfer in second and subsequent programming languages. In *Cambridge Computing Education Research Symposium*, page 18.

[32] Ethel Tshukudu. Towards a model of conceptual transfer for students learning new programming languages. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 355–356, 2019.

[33] Ethel Tshukudu and Siri Annethe Moe Jensen. The role of explicit instruction on students learning their second programming language. In *United Kingdom & Ireland Computing Education Research conference.*, pages 10–16, 2020.

[34] Kris Powers, Stacey Ecott, and Leanne M Hirshfield. Through the looking glass: teaching cs0 with alice. In *SIGCSE*, volume 7, pages 213–217. Citeseer, 2007.

[35] David Weintrop and Uri Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *ICER*, volume 15, pages 101–110, 2015.

[36] David Weintrop and Nathan Holbert. From blocks to text and back: Programming patterns in a dual-modality environment. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*, pages 633–638, 2017.

[37] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–25, 2017.

[38] Luke Moors, Andrew Luxton-Reilly, and Paul Denny. Transitioning from block-based to text-based programming languages. In *2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 57–64. IEEE, 2018.

[39] Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. Experiences in bridging from functional to object-oriented programming. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pages 36–40, 2019.

[40] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: why is it difficult for developers to learn another programming language? In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 691–701. IEEE, 2020.

[41] James P Byrnes. *Cognitive development and learning in instructional contexts*. Allyn & Bacon, 2001.

[42] Bransford, J., Brown, A. & Cocking, R. How people learn: Brain, mind, experience, and school.. (National Academy Press,1999)

[43] Robert S Woodworth and EL Thorndike. The influence of improvement in one mental function upon the efficiency of other functions.(i). *Psychological review*, 8(3):247, 1901.

[44] Edward L Thorndike and Robert S Woodworth. The influence of improvement in one mental function upon the efficiency of other functions. ii. the estimation of magnitudes. *Psychological Review*, 8(4):384, 1901.

[45] David N Perkins, Gavriel Salomon, et al. Transfer of learning. *International encyclopedia of education*, 2:6452–6457, 1992.

[46] Gavriel Salomon and David N Perkins. Rocky roads to transfer: Rethinking mechanism of a neglected phenomenon. *Educational psychologist*, 24(2):113–142, 1989.

[47] Roy D Pea and D Midian Kurland. On the cognitive effects of learning computer programming. *New ideas in psychology*, 2(2):137–168, 1984.

[48] Meryl Reis Louis and Robert I Sutton. Switching cognitive gears: From habits of mind to active thinking. *Human relations*, 44(1):55–76, 1991.

[49] Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[50] Mordechai Ben-Ari. Constructivism in computer science education. *Acm sigcse bulletin*, 30(1):257–261, 1998.

[51] Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[52] Dale Parsons and Patricia Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163, 2006.

[53] Jean Salac and Diana Franklin. If they build it, will they understand it? exploring the relationship between student code and performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 473–479, 2020.

[54] Carsten Schulte. Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research*, pages 149–160. ACM, 2008.

[55] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[56] Ben Shneiderman. Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, 5(2):123–143, 1976.

[57] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. *The nature of expertise*, 129:152, 1988.

[58] David A Watt. *Programming language design concepts*. John Wiley & Sons, 2004.

[59] Aneta Pavlenko. *6. Conceptual Representation in the Bilingual Lexicon and Second Language Vocabulary Learning*. Multilingual Matters, 2009.

[60] Aneta Pavlenko. *The bilingual mental lexicon: Interdisciplinary approaches*, volume 70. Multilingual Matters, 2009.

[61] Scott Jarvis. *5. Lexical Transfer*. Multilingual Matters, 2009.

[62] Zhang Ying. The representation of bilingual mental lexicon and english vocabulary acquisition. *English Language Teaching*, 10(12):24–27, 2017.

[63] Willem JM Levelt. *Speaking: From intention to articulation*, volume 1. MIT press, 1993.

[64] Mick Randall. *Memory, psychology and second language learning*, volume 19. John Benjamins Publishing, 2007.

[65] Mark Aronoff and Kirsten Fudeman. *What is morphology?*, volume 8. John Wiley & Sons, 2011.

[66] Charles A Perfetti, Laura C Bell, and Suzanne M Delaney. Automatic (prelexical) phonetic activation in silent word reading: Evidence from backward masking. *Journal of Memory and Language*, 27(1):59–70, 1988.

[67] Muhammad Raji Zughoul. Lexical choice: Towards writing problematic word lists. *International Review of Applied Linguistics*, 29(1):45–60, 1991.

[68] Anne Castles, Kathleen Rastle, and Kate Nation. Ending the reading wars: Reading acquisition from novice to expert. *Psychological Science in the Public Interest*, 19(1):5–51, 2018.

[69] Agnieszka Otwinowska. *Cognate vocabulary in language acquisition and use*. Multilingual Matters, 2015.

[70] Håkan Ringbom. On l1 transfer in l2 comprehension and l2 production. *Language learning*, 42(1):85–112, 1992.

[71] John R Anderson. Acquisition of cognitive skill. *Psychological review*, 89(4):369, 1982.

[72] R Mitchell and F Myles. Second language learning theories (2nd edn), London: Arnold, 2004.

[73] Rosamond Mitchell, Florence Myles, and Emma Marsden. *Second language learning theories*. Routledge, 2019.

[74] Michael Homer and James Noble. Combining tiled and textual views of code. In *2014 Second IEEE Working Conference on Software Visualization*, pages 1–10. IEEE, 2014.

[75] John W Creswell and Vicki L Plano Clark. *Designing and conducting mixed methods research*. Sage publications, 2017.

[76] R Burke Johnson, Anthony J Onwuegbuzie, and Lisa A Turner. Toward a definition of mixed methods research. *Journal of mixed methods research*, 1(2):112–133, 2007.

[77] Victoria Clarke, Virginia Braun, and Nikki Hayfield. Thematic analysis. *Qualitative psychology: A practical guide to research methods*, pages 222–248, 2015.

[78] Helen Noble and Joanna Smith. Issues of validity and reliability in qualitative research. *Evidence-based nursing*, 18(2):34–35, 2015.

[79] Ali Delice. The sampling issues in quantitative research. *Educational Sciences: Theory and Practice*, 10(4):2001–2018, 2010.

[80] Mehran Sahami, Steve Roach, Ernesto Cuadros-Vargas, and Richard LeBlanc. Acm/ieee-cs computer science curriculum 2013: reviewing the ironman report. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 13–14, 2013.

[81] Strawman Draft. Computer science curricula 2013. *ACM and IEEE Computer Society, Incorporated: New York, NY, USA*, 2013.

[82] Rod Ellis. 1. implicit and explicit learning, knowledge and instruction. In *Implicit and explicit knowledge in second language learning, testing and teaching*, pages 3–26. Multilingual Matters, 2009.

[83] Jennifer A Livingston. Metacognition: An overview. 2003.

[84] Anne McKeough, Judy Lee Lupart, and Anthony Marini. *Teaching for transfer: Fostering generalization in learning*. Routledge, 2013.

[85] John D Bransford, Ann L Brown, Rodney R Cocking, et al. *How people learn*, volume 11. Washington, DC: National academy press, 2000.

[86] Miranda C Parker, Mark Guzdial, and Shelly Engleman. Replication, validation, and use of a language independent cs1 knowledge assessment. In *Proceedings of the 2016 ACM conference on international computing education research*, pages 93–101, 2016.

[87] Dedre Gentner. The development of relational category knowledge. In *Building object categories in developmental time*, pages 263–294. Psychology Press, 2005.

[88] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006.

[89] Antony Bryant and Kathy Charmaz. *The SAGE handbook of current developments in grounded theory*. Sage, 2019.

[90] Ilker Etikan, Sulaiman Abubakar Musa, and Rukayya Sunusi Alkassim. Comparison of convenience sampling and purposive sampling. *American journal of theoretical and applied statistics*, 5(1):1–4, 2016.

[91] Justus J Randolph, George Julnes, Erkki Sutinen, and Steve Lehman. A methodological review of computer science education research. *Journal of Information Technology Education: Research*, 7(1):135–162, 2008.

[92] David Stokes and Richard Bergin. Methodology or "methodolatry"? an evaluation of focus groups and depth interviews. *Qualitative market research: An international Journal*, 2006.

[93] Paul Gill, Kate Stewart, Elizabeth Treasure, and Barbara Chadwick. Methods of data collection in qualitative research: interviews and focus groups. *British dental journal*, 204(6):291–295, 2008.

[94] Elizabeth Charters. The use of think-aloud methods in qualitative research an introduction to think-aloud methods. *Brock Education Journal*, 12(2), 2003.

[95] Robert S Siegler and Kevin Crowley. The microgenetic method: A direct means for studying cognitive development. *American psychologist*, 46(6):606, 1991.

[96] Donna Teague and Raymond Lister. Longitudinal think aloud study of a novice programmer. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*, pages 41–50. Australian Computer Society, Inc., 2014.

[97] Victoria Clarke and Virginia Braun. Teaching thematic analysis: Overcoming challenges and developing strategies for effective learning. *The psychologist*, 26(2), 2013.

[98] Jennifer Fereday and Eimear Muir-Cochrane. Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development. *International journal of qualitative methods*, 5(1):80–92, 2006.

[99] Kristin A Searle and Yasmin B Kafai. Boys' needlework: Understanding gendered and indigenous perspectives on computing and crafting with electronic textiles. In *ICER*, pages 31–39, 2015.

[100] Colleen M Lewis. The importance of students' attention to program state: a case study of debugging behavior. In *Proceedings of the ninth annual international conference on International computing education research*, pages 127–134. ACM, 2012.

[101] William R Cook. On understanding data abstraction, revisited. In *ACM SIGPLAN Notices*, volume 44, pages 557–572. ACM, 2009.

[102] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, 1977.

[103] Anna Eckerdal and Michael Thuné. Novice java programmers' conceptions of object and class, and variation theory. In *ACM SIGCSE Bulletin*, volume 37, pages 89–93. ACM, 2005.

[104] Juha Sorva. Misconceptions and the beginner programmer. *Computer Science Education: Perspectives on Teaching and Learning in School*, page 171, 2018.

[105] Willem JM Levelt, Ardi Roelofs, and Antje S Meyer. A theory of lexical access in speech production. *Behavioral and brain sciences*, 22(1):1–38, 1999.

[106] Steven Davis. *Connectionism: Theory and practice*. Number 3. Oxford University Press on Demand, 1992.

[107] Joachim Diederich. *Knowledge-intensive recruitment learning*. International Computer Science Institute Berkeley, CA, 1988.

[108] American Educational Research Association et al. *Standards for educational and psychological testing*. American Educational Research Association, 2018.

[109] Allison Elliott Tew and Mark Guzdial. Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 97–101, 2010.

[110] Allison Elliott Tew and Mark Guzdial. The fcs1: a language independent assessment of cs1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 111–116, 2011.

[111] Shuchi Grover, Stephen Cooper, and Roy Pea. Assessing computational learning in k-12. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 57–62, 2014.

[112] David Weintrop. *Modality matters: Understanding the effects of programming language representation in high school computer science classrooms*. PhD thesis, Northwestern University, 2016.

[113] Ravi Sethi. *Programming languages concepts and constructs*. Addison Wesley Longman Publishing Co., Inc., 1996.

[114] Ravi Sethi. *Programming languages: concepts and constructs*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[115] Sally A Fincher and Anthony V Robins. *The Cambridge handbook of computing education research*. Cambridge University Press, 2019.

[116] Norbert Schmitt. Tracking the incremental acquisition of second language vocabulary: A longitudinal study. *Language learning*, 48(2):281–317, 1998.

[117] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 213–218. ACM, 2017.

[118] Tamar Degani, Anat Prior, and Natasha Tokowicz. Bidirectional transfer: The effect of sharing a translation. *Journal of Cognitive Psychology*, 23(1):18–28, 2011.

[119] Joel E Dworin. Insights into biliteracy development: Toward a bidirectional theory of bilingual pedagogy. *Journal of Hispanic Higher Education*, 2(2):171–186, 2003.

[120] Aneta Pavlenko and Scott Jarvis. Bidirectional transfer. *Applied linguistics*, 23(2):190–214, 2002.

[121] Ethel Tshukudu, Quintin Cutts, Olivier Goletti, Alaaeddin Swidan, and Felienne Hermans. Teachers' views and experiences on teaching second and subsequent programming languages. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, pages 294–305, 2021.

[122] Bert Zwaneveld, Jacob Perrenet, and Roel Bloo. Discussion of methods for threshold research and an application in computer science. In *Threshold concepts in practice*, pages 269–284. Brill Sense, 2016.

[123] Nino Pataraia, Isobel Falconer, Anoush Margaryan, Allison Littlejohn, and Sally Fincher. Who do you talk to about your teaching?'networking activities among university teachers. *Frontline Learning Research*, 2(2):4–14, 2014.

[124] Daniel I Lee, Gwendolyn Gardiner, Erica Baranski, International Situations Project, and David C Funder. Situational experience around the world: A replication and extension in 62 countries. *Journal of personality*, 88(6):1091–1110, 2020.

[125] Judith Bell and Stephen Waters. *EBOOK: DOING YOUR RESEARCH PROJECT: A GUIDE FOR FIRST-TIME RESEARCHERS*. McGraw-Hill Education (UK), 2018.

[126] Yan Zhang and Barbara M Wildemuth. Unstructured interviews. *Applications of social research methods to questions in information and library science*, pages 222–231, 2009.

[127] Jane Ritchie, Jane Lewis, Carol McNaughton Nicholls, Rachel Ormston, et al. *Qualitative research practice: A guide for social science students and researchers*. sage, 2013.

[128] Virginia Braun and Victoria Clarke. Thematic analysis. 2012.

[129] Mostafa Javadi, Koroush Zarea, et al. Understanding thematic analysis and its pitfall. *Demo*, 1(1):33–39, 2016.

[130] Felienne Hermans. Hedy: A gradual language for programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 259–270, 2020.

[131] KN King. The evolution of the programming languages course. In *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 213–219, 1992.

[132] Dale Parsons and Patricia Haden. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Procedings of The Twentieth Annual NACCQ Conference*, pages 209–215, 2007.

[133] Julie Stern, Krista Ferraro, and Juliet Mohnkern. *Tools for teaching conceptual understanding, secondary: Designing lessons and assessments for deep learning*. Corwin Press, 2017.

[134] Julie Stern, Krista Ferraro, Kayla Duncan, and Trevor Aleo. *Learning That Transfers: Designing Curriculum for a Changing World*. Corwin Press, 2021.

[135] Nour Tabet, Huda Gedawy, Hanan Alshikhabobakr, and Saquib Razak. From alice to python. introducing text-based programming in middle schools. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 124–129, 2016.

[136] Monika Mladenović, Žana Žanko, and Andrina Granić. Mediated transfer: From text to blocks and back. *International Journal of Child-Computer Interaction*, page 100279, 2021.

[137] Katrina Falkner and Judy Sheard. 15 pedagogic approaches. *The Cambridge Handbook of Computing Education Research*, page 445, 2019.

[138] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of economic behavior & organization*, 81(1):1–8, 2012.

[139] PC Price, RS Jhangiani, IA Chiang, DC Leighton, and C Cuttler. Research methods in psychology (3rd american edition). *Washington: PressBooksPublications*, 2017.

[140] Mahmoud Kaddoura. Think pair share: A teaching learning strategy to enhance students' critical thinking. *Educational Research Quarterly*, 36(4):3–24, 2013.

[141] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[142] Lauren Margulieux, Paul Denny, Kathryn Cunningham, Michael Deutsch, and Benjamin R Shapiro. When wrong is right: The instructional power of multiple conceptions. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, pages 184–197, 2021.

[143] Nick C Ellis. *Memory for language*. na, 2001.

[144] Ethel Tshukudu, Quintin Cutts, and Mary Ellen Foster. Evaluating a pedagogy for improving conceptual transfer and understanding in a second programming language learning context. In *21st Koli Calling International Conference on Computing Education Research*, pages 1–10, 2021.

[145] Kevin M Oliver. Methods for developing constructivist learning on the web. *Educational technology*, 40(6):5–18, 2000.

[146] Ken Rowe. Effective teaching practices for students with and without learning difficulties: Constructivism as a legitimate theory of learning and of teaching? *Student Learning Processes*, page 10, 2006.

[147] Keith J Holyoak and Kyunghee Koh. Surface and structural similarity in analogical transfer. *Memory & cognition*, 15(4):332–340, 1987.

[148] Mary L Gick and Keith J Holyoak. The cognitive basis of knowledge transfer. In *Transfer of learning*, pages 9–46. Elsevier, 1987.

[149] Anthony V Robins, Lauren E Margulieux, and Briana B Morrison. Cognitive sciences for computing education. *Handbook of Computing Education Research*, pages 231–275, 2019.

[150] Wallace Feurzeig and George Lukas. Logo—a programming language for teaching mathematics. *Educational Technology*, 12(3):39–46, 1972.

[151] Varvara Garneli, Michail N Giannakos, and Konstantinos Chorianopoulos. Computing education in k-12 schools: A review of the literature. In *2015 IEEE Global Engineering Education Conference (EDUCON)*, pages 543–551. IEEE, 2015.

[152] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. What do we think we think we are doing? metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 2–13, 2020.

[153] Paul Denny, James Prather, Brett A Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. A closer look at metacognitive scaffolding: Solving test cases before programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, pages 1–10, 2019.

[154] Goudarz Alibakhshi, Fariborz Nikdel, and Akram Labbafi. Exploring the consequences of teachers' self-efficacy: a case of teachers of english as a foreign language. *Asian-Pacific Journal of Second and Foreign Language Education*, 5(1):1–19, 2020.

[155] Muhammad Gulistan, Athar Hussain, et al. Relationship between mathematics teachers' self efficacy and students' academic achievement at secondary level. *Bulletin of Education and Research*, 39(3):171–182, 2017.

[156] David Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. ir85-22. 1985.

[157] John M Belmont, Earl C Butterfield, Ralph P Ferretti, et al. To secure transfer of training instruct self-management skills. *How and how much can intelligence be increased*, 147:154, 1982.

[158] Mark Guzdial. Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, 2015.

[159] Quintin Cutts, Matthew Barr, Mireilla Bikanga Ada, Peter Donaldson, Steve Draper, Jack Parkinson, Jeremy Singer, and Lovisa Sundin. Experience report: Thinkathon–countering an" i got it working" mentality with pencil-and-paper exercises. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 203–209, 2019.

[160] SK Johnson. Conceptual and procedural knowledge in mathematics: An introductory analysis. *Conceptual and procedural knowledge: The case of mathematics*, 91(1):175–189, 2005.

[161] Camilla Gilmore and Lucy Cragg. The role of executive function skills in the development of children's mathematical competencies. In *Heterogeneity of function in numerical cognition*, pages 263–286. Elsevier, 2018.

[162] Arthur J Baroody. The development of adaptive expertise and flexibility: The integration of conceptual and procedural knowledge. *The development of arithmetic concepts and skills: Constructive adaptive expertise*, pages 1–33, 2003.

[163] Manu Kapur. Examining productive failure, productive success, unproductive failure, and unproductive success in learning. *Educational Psychologist*, 51(2):289–299, 2016.

[164] Manu Kapur and Charles K Kinzer. Productive failure in cscl groups. *International Journal of Computer-Supported Collaborative Learning*, 4(1):21–46, 2009.

[165] Mary L Gick and Keith J Holyoak. Schema induction and analogical transfer. *Cognitive psychology*, 15(1):1–38, 1983.

[166] William F Atchison, Samuel D Conte, John W Hamblen, Thomas E Hull, Thomas A Keenan, William B Kehl, Edward J McCluskey, Silvio O Navarro, Werner C Rheinboldt, Earl J Schweppe, et al. Curriculum 68: Recommendations for academic programs in computer science: a report of the acm curriculum committee on computer science. *Communications of the ACM*, 11(3):151–197, 1968.

[167] Richard H Austing, Bruce H Barnes, Della T Bonnette, Gerald L Engel, and Gordon Stokes. Curriculum'78: recommendations for the undergraduate program in computer science—a report of the acm curriculum committee on computer science. *Communications of the ACM*, 22(3):147–166, 1979.

[168] Stephen Davies, Jennifer A Polack-Wahl, and Karen Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 625–630, 2011.

[169] Ursula Wolz, Henry H Leitner, David J Malan, and John Maloney. Starting with scratch in cs 1. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pages 2–3, 2009.

[170] David J Malan and Henry H Leitner. Scratch for budding computer scientists. *ACM Sigcse Bulletin*, 39(1):223–227, 2007.

[171] Cindy Marling and David Juedes. Cs0 for computer science majors at ohio university. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 138–143, 2016.

[172] Neil CC Brown, Sue Sentance, Tom Crick, and Simon Humphreys. Restart: The resurgence of computer science in uk schools. *ACM Transactions on Computing Education (TOCE)*, 14(2):1–22, 2014.

[173] LB Wilson and RG Clark. Comparative programming languages addison-wesley. *Reading Mass*, 1988.

[174] Jeffrey Bonar and Elliot Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2):133–161, 1985.

[175] Roy D Pea. Language-independent conceptual "bugs" in novice programming. *Journal of educational computing research*, 2(1):25–36, 1986.

[176] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–24, 2017.

[177] Dimitrios Doukakis, Maria Grigoriadou, and Grammatiki Tsaganou. Understanding the programming variable concept with animated interactive analogies. In *Proceedings of the The 8th Hellenic European Research on Computer Mathematics & Its Applications Conference (HERCMA'07)*, 2007.

[178] Michael Kölling, Neil CC Brown, and Amjad Altadmri. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, pages 29–38. ACM, 2015.

[179] Kintsch, Walter and Van Dijk, Teun A. Toward a model of text comprehension and production. In *Psychological review*. American Psychological Association, 85(5):363, 1978.

[180] Schulte, Carsten and Clear, Tony and Taherkhani, Ahmad and Busjahn, Teresa and Paterson, James H An introduction to program comprehension for computer science educators In *Proceedings of the 2010 ITiCSE working group reports*, pages 65–86, 2010.

[181] Prat, Chantel S and Madhyastha, Tara M and Mottarella, Malayka J and Kuo, Chu-Hsuan. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports, Nature Publishing Group*, 10(1):1–10, 2020.

[182] Fodor, Jerry A and Pylyshyn, Zenon W. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988

[183] Rumelhart, D. The architecture of mind: A connectionist approach. *Mind Readings*. pp. 207-238 (1998)

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

[185] Ellis, R. Implicit and explicit learning, knowledge and instruction. *Implicit And Explicit Knowledge In Second Language Learning, Testing And Teaching*. **42** pp. 3-25 (2009)

[186] Yilmaz, K. Constructivism: Its theoretical underpinnings, variations, and implications for classroom instruction. *Educational Horizons*. **86**, 161-172 (2008)

[187] Kim, J. The effects of a constructivist teaching approach on student academic achievement, self-concept, and learning strategies. *Asia Pacific Education Review*. **6**, 7-19 (2005)

[188] Ebert, M. & Ring, M. A presentation framework for programming in programing lectures. *2016 IEEE Global Engineering Education Conference (EDUCON)*. pp. 369-374 (2016)

[189] Kurtz, B., Fenwick, J., Tashakkori, R., Esmail, A. & Tate, S. Active learning during lecture using tablets. *Proceedings Of The 45th ACM Technical Symposium On Computer Science Education*. pp. 121-126 (2014)

[190] Rich, P., Larsen, R. & Mason, S. Measuring teacher beliefs about coding and computational thinking. *Journal Of Research On Technology In Education*. pp. 1-21 (2020)

[191] Reber, A. Implicit learning of artificial grammars. *Journal Of Verbal Learning And Verbal Behavior*. **6**, 855-863 (1967)

[192] Reber, A. Transfer of syntactic structure in synthetic languages.. *Journal Of Experimental Psychology*. **81**, 115 (1969)

[193] Leung, J. & Williams, J. Prior linguistic knowledge influences implicit language learning. *Proceedings Of The Annual Meeting Of The Cognitive Science Society*. **35** (2013)

[194] Spada, N. & Tomita, Y. Interactions between type of instruction and type of language feature: A meta-analysis. *Language Learning*. **60**, 263-308 (2010)

[195] Qvortrup, A., Wiberg, M., Christensen, G. & Hansbøl, M. On the definition of learning. (University Press of Southern Denmark,2016)

[196] Talley, P. & Hui-Ling, T. Implicit and explicit teaching of English speaking in the EFL classroom. *International Journal Of Humanities And Social Science*. **4**, 38-45 (2014)

[197] Exton, C. Constructivism and program comprehension strategies. *Proceedings 10th International Workshop On Program Comprehension*. pp. 281-284 (2002)

[198] Neuhauser, C. Learning style and effectiveness of online and face-to-face instruction. *The American Journal Of Distance Education*. **16**, 99-113 (2002)

[199] McCutcheon, K., Lohan, M., Traynor, M. & Martin, D. A systematic review evaluating the impact of online or blended learning vs. face-to-face learning of clinical skills in undergraduate nurse education. *Journal Of Advanced Nursing*. **71**, 255-270 (2015)

[200] Ryan, R. & Deci, E. Intrinsic and extrinsic motivation from a self-determination theory perspective: Definitions, theory, practices, and future directions. *Contemporary Educational Psychology*. **61** pp. 101860 (2020,4), https://linkinghub.elsevier.com/retrieve/pii/S0361476X20300254, 00000

[201] Kroeger, P. Analyzing grammar: An introduction. (Cambridge University Press,2005)

[202] Ringbom, H. Actual, perceived and assumed cross-linguistic similarities in foreign language learning. *AFinLAn Vuosikirja*. (2007)

[203] Gooskens, C. The North Germanic Dialect Continuum. *The Cambridge Handbook Of Germanic Linguistics*. pp. 761-782 (2020)

[204] Baker, C. The effects of the Norman conquest on the English language. *Tenor Of Our Times*. **5**, 41 (2016)

[205] Tabet, N., Gedawy, H., Alshikhabobakr, H. & Razak, S. From alice to python. Introducing text-based programming in middle schools. *Proceedings Of The 2016 ACM Conference On Innovation And Technology In Computer Science Education*. pp. 124-129 (2016)

[206] Mladenović, M., Žanko, Ž. & Granić, A. Mediated transfer: From text to blocks and back. *International Journal Of Child-Computer Interaction*. **29** pp. 100279 (2021)

[207] Kao, Y., Matlen, B. & Weintrop, D. From One Language to the Next: Applications of Analogical Transfer for Programming Education. *ACM Transactions On Computing Education (TOCE)*. (2022)

[208] Kotovsky, L. & Gentner, D. Comparison and categorization in the development of relational similarity. *Child Development*. **67**, 2797-2822 (1996)

[209] Gentner, D. Bootstrapping the mind: Analogical processes and symbol systems. *Cognitive Science*. **34**, 752-775 (2010)

[210] Espinal, A., Vieira, C. & Guerrero-Bequis, V. Student ability and difficulties with transfer from a block-based programming language into other programming languages: a case study in Colombia. *Computer Science Education*. pp. 1-33 (2022,6)

Appendix A

Chapter 4: Exploratory Study Materials

This appendix includes the instruments that were used in Chapter 4 (The exploratory study). It includes the quizzes, surveys and consent form administered to the students. The quizzes include both the Python and the Java version of code comprehension questions.

The following interview scripts were used as a guidance in interviewing the participants:

A.0.1 Week 0 interview activities: Before Java

Python Program Interview Questions: The following are further Questions if the student's explanation is not enough to deduce their comprehension of the program

- From you understanding of the Python programming language, talk me through, what you see in this whole program
- What do you think this line of code does?
- What do you think this whole program prints after execution?

Java Program Interview Questions:

- I obviously do not expect you to know everything in this java code as I assume you have not been educated on it yet. I know you have experience of python (and maybe other programming languages). Looking at this java code take me through what you are seeing?
- Does it look completely new to you or if it's more or less something you already know from python or elsewhere, if so what?
- What do you think this line of code does?
- What do you think this whole program prints after execution?

Listing A.1: Python Program used in session 1

```
total = 0;

def sum( arg1, arg2, arg3 ):
    total = arg1 + arg2;

    if arg1 > arg3:
     print "Total is equal to : ", total+1
    elif arg1 < arg3:
     print "Total is equal to : ", total-1
    return total;

x=10
y=20
z=30
sum( x, y, z );
print "Total is equal to : ", total
```

Listing A.2: Java Program used in session 1

```
public class Rectangle
  int length=3;
  int width=4;
  public static void main(String args[]) {
    Rectangle r1=new Rectangle();
    r1.insert(12,4);
  }
  public int insert(int l, int w) {
    int result=l*w;
    if (l > w)
      System.out.println("The answer is " + l*w);
    else
      System.out.println("no result");
    return result;
  }
}
```

A.0.2 Week 2 interview activities: After two weeks of Java

The purpose of this question is to observe how the students map their knowledge of the Python code to Java code comprehension on Carryover and Changed Concepts: Do they use Semantic mapping or Syntax mapping or both.

- Q1) You have been given a python code on the left side of the Table, write down the results when it executes. Identify the similar code that has the same logic statement and result on

the Java code given on the right side of the table.

Python		Java
mylist = [1,2,3] for x in range(len(mylist)): print (mylist[x])	a.	int array []= {1,2,3}; for (int x in range(len(array)) { System.out.println(x); }
	b.	int array []= {1,2,3}; for (int x=0; x<array.length; x++){ { System.out.println(array[x]);}
	c.	No equivalent

Figure A.1: **program 1 mapping**

Python			Java

a=3
b=9
if a<b:
print"a is less than b"
elif a>b:
print"a is greater than b"
else:
print"Bye"
		a.	int a=3;
			int b=9;
			if (a<b){
			System.out.println ("a less than b");}
			else if(a>b) {
			System.out.println ("a greater than b");}
			else {
			System.out.println ("Bye");}

		b.	int a=3;
			int b=9;
			if (a<b){
			System.out.println ("a less than b");}
			else (a>b) {
			System.out.println ("a greater than b");}
			else {
			System.out.println ("Bye");}

		c.	No equivalent

Figure A.2: **program 2 mapping**

Python		Java

x=4
y=6
y, x = x, y
 	
print (x,y)
	a.	int x=4;
		int y=6;
		int z=x;
		x=y;
		y=z;
		System.out.println (x+ " " + y);

	b.	int x=4;
		int y=6;
		y,x=x,y;
		System.out.println (x+ " " + y);

	c.	No equivalent

Figure A.3: **program 3 mapping**

Python		Java

for i in range(5):
  if i==3:
  i = 10
  print(i)
	a.	for (int i=0; i<5; i++;)
		  {
		  if (i==3)
		  { i=10;}
		  System.out.print(i);}

	b.	for (int i=0; i<5;)
		  {
		  if (i==3)
		  { i=10;}
		  System.out.print(i);}

	c.	No equivalent

Figure A.4: **program 4 mapping**

Question 5: (Same Syntax, Same Semantics)

Python		Java

for i in range(1, 10, 2):
  print (i)
	a.	for (int i = 1; i < 10; i += 2)
		  { System.out.print(i)}

	b.	for (int i = 1; i < 10;){
		  i+=2;
		  System.out.print(i); }

	c.	No equivalent
Answer		

Figure A.5: **program 5 mapping**

Python		Java
a=3 a="Hello" print (a)	a.	int a=3; String a="Hello"; System.out.println(a);
	b.	int a=3; int a="Hello"; System.out.println(a);
	c.	No equivalent

Figure A.6: **program 6 mapping**

A.0.3 Week 4 interview activities: After four weeks of Java

- Please indicate and explain your experiences (how easy or difficult) it was for you to learn each concept by circling the number that indicates the level of learning difficulty. (Variables, Conditional Statements Iteration, Arrays, Methods, Objects, Classes and Encapsulation)
- How confident are you in your understanding of each of the Java Programming Constructs in Table a? You can uses a scale of 1-5; 1 being the least Confident and 5 being the most Confident
- Execute the below Python and Java code line by line, talking me through what is happening as each line is executed.

Listing A.3: Python Program used in session 3

```
for i in range(5):
    if i==3:
        i = 10
    print(i)
```

Listing A.4: Java Program used in session 3

```
for ( int i=0; i<5; i++;)
{
    if (i==3)
        {
        i=10;
        }
    System.out.print( i);
}
```

A.0.4 Week 6 interview activities: After six weeks of Java

- You are in your sixth week of learning Java Programming; what is your experience now with learning the following Java programming constructs (Variables, Conditional Statements, Iteration, Arrays, Methods, Objects, Classes, Encapsulation, Inheritance, Polymorphism)? You can share about your confusions, blocks, breakthroughs e.t.c
- How confident are you in your understanding of each of the Java Programming Constructs in Table a? You can uses a scale of 1-5; 1 being the least Confident and 5 being the most Confident
- The purpose of this exercise is to trigger student's prior knowledge in Python Dictionaries that they can use to map to Java objects. Students will find similarities between Python dictionaries and java objects and observe if it can help in their understanding of Java Objects:

Listing A.5: Python Program used in session 4 (week 6)

```
def p(n,a):
    return {'name':n, 'age':a}

def getName(obj):
    return obj['name']

def incAge(obj):
    a = obj["age"]+1
    return a

me=p("Joseph", 51)
you=p("Vic", 35)

print("your name is:", getName(me),", your age is:", incAge(me))
print("your name is:", getName(you),", your age is:", incAge(you))
```

Listing A.6: Java Program used in session 4 (week 6)

```
public class person{

    public String name;
    public int age;

    public person (String n, int a)
    {
        this.name=n;
        this.age=a;
    }
    public String getName()
    {
    return name;
    }

    public int incAge()
    {
        int a=age+1;
        return a;
    }

    public static void main(String []args)
    {
        person n1=new person("Joseph",51);
        person n2=new person("Vic",35);

        System.out.println( "Your␣name␣is:␣" + n1.getName() + "␣Age:"
        System.out.println( "Your␣name␣is:␣" + n2.getName() + "␣Age:"
    }
}
```

University of Glasgow | College of Science & Engineering

Consent Form

Title of Project: Understanding Conceptual Transfer for students learning new programming languages

I understand that my data is being collected in the form of written questionnaire and quizzes, for use in an academic research project at the University of Glasgow. The data collected will later be transcribed and analysed, and the data may be used in published reports and/or research papers. I understand that I will remain anonymous in any paper or report that is published.

I give my consent to the use of data for the purpose of academic research on the understanding that:

- The material I provide will be handled in the manner outlined above.
- The material will be kept in secure storage at all times.
- The material will be retained in secure storage for use in future academic research.
- The material may be used in future publications, both print and online.
- I have read and understood the attached Participant Information Sheet.

I wish to be kept informed of any publications resulting from this work: ☐

Signed by the participant: _____ Date: _____

[OPTIONAL] Preferred email address (if you have indicated that you wish to be kept informed of any resulting publications):

Figure A.7: **Consent Form for the first exploratory study**

Appendix B

Chapter 6: Validation of the MPLT Study Materials

This appendix includes the instruments that were used in Chapter 6. The appendix starts with the construct categorisations instrument that is explained in Section 6.2 in Chapter 6. Then it is followed by the consent form and the quizzes as they were presented to the students. The quizzes include both the Python and the Java version of code comprehension questions.

Categorisation of Java and Python constructs according to MPLT (TCC, FCC and ATCC)

Control Structures				
Construct	**Construct Category**		**Python**	**Java**

While loop	TCC	Syntax	`i = 0;`
			`while (i < 3):`
			`  a=1`
			`  print(a)`
			`  i+=1`
				`int i = 0;`
				`while (i < 3) {`
				`  int a=1;`
				`  System.out.println(a);`
				`  i++;`
				`}`

		Semantics	*1* *1* *1*	*1* *1* *1*

If statement	TCC	Syntax	`a=1;`
			`b=2;`
			`if (a>b):`
			`  print("hello")`
			`else:`
			`  print("bye")`
				`int a=1;`
				`int b=2;`
				`if (a>b){`
				`  System.out.println("hello");}`
				`else{`
				`  System.out.println("bye");}`

		Semantics	*bye*	*bye*

For loop	TCC	Syntax	`for i in range(2):`
			`  print ("hello")`
				`for (int i=0; i<2; i++){`
				`  System.out.println("hello");`
				`}`

		Semantics	*hello* *hello*	*hello* *hello*

For loop	FCC	Syntax	`for i in range(2):`
			`  print ("hello")`
			`  i=3`
				`for (int i=0; i<2; i++){`
				`  System.out.println("hello");`
				`  i=3;`
				`}`

		Semantics	*hello* *hello*	*hello*

For loop Scope	FCC	Syntax	`for i in range(2):`
			`  a=2`
			`print (a)`
				`for (int i=0; i<2; i++){`
				`  int b=2;`
				`}`
				`  System.out.println(b);`

		Semantics	*2*	*error can't find b*

While loop Scope	FCC	Syntax	`i = 0;`
			`while (i < 2):`
			`  a=1`
			`  print(a)`
			`  i+=1`
			`print(a)`
				`int i = 0;`
				`while (i < 2) {`
				`  int a=1;`
				`  i++;`
				`}`
				`System.out.println(a);`

		Semantics	*1* *1* *1*	*error can't find symbol a*

Functions (Scope, arguments and parameters, function calls)				

Functions	TCC	Syntax	def cap(z,y):
			 pa=z+y
			 zap()
			 return pa
			
			def zap():
			 pa=3
			 print(pa)
			
			print (cap(7,3))
				public class MyClass {
				
				 public static int gen(int g, int s)
				 {
				 int a=g+s;
				 cen();
				 return a;
				 }
				 public static void cen()
				 {
				 int a=3;
				 System.out.println(a);
				 }
				 public static void main(String[] args)
				 {
				 System.out.println(gen(7,3));
				 }
				
				}

		Semantics	*3* *10*	*3* *10*
Data Types and Type Checking (Static vs Dynamic)				

Primitive data types	FCC	Syntax	a=1
			a=10.5
			print (a)
				int a=1;
				a=10.5;
				System.out.println (a);

		Semantics	*10.5*	*error, incompatible types*

	FCC	Syntax	a=1
			a=2
			print (a)
				int a=1;
				int a=2;
				print (a)

		Semantics	*2*	*variable a already defined*

	TCC	Syntax	a=1
			a=2
			print (a)
				int a=1;
				a=2;
				print (a)

		Semantics	*2*	*2*
Data Types and Copy and Reference Semantics				

Copy semantics for primitive types in Java and Python reference semantics (e.g., it creates a new '***a' variable***)	TCC	Syntax	a=1
			b=a
			a=2
			print (b)
				int a=1;
				int b=a;
				a=2;
				System.out.println (b);

		Semantics	1	1

Reference semantics for composite types	ATCC	Syntax	cars = ['Toyota']
			cars2=cars
			cars.append("Mercedes")
			print(cars2)
				import java.util.ArrayList;
				public class Main {
				 public static void main(String[] args)
				 {
				 ArrayList<String> cars = new ArrayList<String>();
				 cars.add("Toyota");
				 ArrayList<String> cars2=cars;
				 cars2.add("Mercedes");
				 System.out.println(cars2);
				 }}

		Semantics	[Toyota, Mercedes]	[Toyota, Mercedes]

Expressions and operators				

Equality of composite data types	FCC	Syntax	`e = [1, 2, 3]`
			`f = [1, 2, 3]`
			
			`print(e==f)`
				`int[] e = {1, 2, 3};`
				`int[] f = {1, 2, 3};`
				
				`System.out.println(e==f)`

		Semantics	`True`	`False`

Array addition	TCC	Syntax	`e = [1, 2, 3]`
			`f = [1, 2, 3]`
			`print(e[0]+e[2])`
				`int[] e = {1, 2, 3};`
				`int[] f = {1, 2, 3};`
				
				`System.out.println(e[0]+e[2]);`

		Semantics	`4`	`4`
String concatenation	TCC	Syntax	`print("hello" + "ss")`	`System.out.println("hello" + "ss");`
		Semantics	`helloss`	`helloss`
String coercion	FCC	Syntax	`print( "hello" + 1)`	`System.out.println( "hello" + 1)`
		Semantics	`TypeError: can only concatenate str (not "int") to str`	`hello1`
Integer division	FCC	Syntax	`print(3/2)`	`System.out.println (3/2)`
		Semantics	`1.5`	`1`
String multiplication	FCC	Syntax	`print("hello"*2);`	`System.out.println("hello"*2);`
		Semantics	`hello hello`	`error`
Operator precedence	TCC	Syntax	`print(3*(1+2));`	`System.out.println(3*(1+2));`
		Semantics	`9`	`9`

Objects aliasing	ATCC	Syntax	`n1 = {'name':'Joseph', 'age': 51}`
			`n2=n1`
			`n1['age']= n1['age']+1`
			`print(n2['age'])`
			`print(n1['age'])`
				`public class Robot {`
				`String name;`
				`int age;`
				
				`public Robot(String n, int w)`
				`{`
				`this.name=n;`
				`this.age=w; }`
				
				`public void agga() {`
				`age=age+1;`
				`}`
				
				`public static void main(String []args)`
				`{`
				`Robot n1=new Robot("Joseph", 51);`
				`Robot n2=n1;`
				`n1.agga();`
				`System.out.println (n1.age+ ""`
				`+n2.age);}}`

		Semantics	`52` `52`	`52` `52`

Consent Form

Title of Project: Understanding Conceptual Transfer for students learning new programming languages

Thank you for your interest in participating in this research. Before you agree to take part, please read the participant information sheet [provided separately].

The research project aims to better understand how students transfer from one programming language to the other at the University of Glasgow. Your participation in this research is voluntary. You may refuse to take part in the research or exit the online interview or withdraw from the study at any time without penalty.

You will take the quiz online using Microsoft Forms, where data will be stored in a password protected electronic format. Microsoft Forms does not collect identifying information such as your name, email address, or student-ID. Therefore, your responses will remain anonymous. No one **(including the lecturer)** will be able to identify you or your answers, and no one will know whether or not you participated in the study. Microsoft forms is not a platforms within course structures (e.g., Moodle), where the lecturer can see if you participated or not. Note that participation in this quiz will not in any way affect your course grades.

The data collected will later be analysed, and the data may be used in published reports and/or research papers.

NB:

Electronic consent: Clicking on the "Agree" button indicates that

- You have read the above information
- You voluntarily agree to participate

Whether you tick Agree or Disagree below, you can still take the quiz, which you may find interesting and helpful, only we will know not to include your responses in our research.

☐ Agree

☐ Disagree

Figure B.1: **Consent form for the experiments that validated the model**

Java Questions

I would like to understand your initial expectations about how Java works. For each question, please give your first guess what the code does.

What will be the output of this Java program fragment?

```
int a;
a=1;
a=10.5;
System.out.println (a);
```

What will be the output of this Java program fragment?

```
for (int i=0; i<2; i++)
{
  String var="Hello";
  System.out.println(var);
}
System.out.println(var);
```

What will be the output of this Java program fragment?

```
int i=0;
while (i<2){
  System.out.println(i);
  i++;
}
```

What will be the output of this Java program fragment?

```
String a=new String ("lab");
String b=new String ("lab");
System.out.println(a==b);
```

What will be the output of this Java program fragment?

```
public class Main{
public static int gen(int g, int s) {
int a=g*2;
return a;
}

public static void main(String[] args) {
System.out.println(gen(7,3));
}
}
```

What will be the output of this Java program fragment?

```
public class Robot {
  String label;
  int num;

public Robot(String n, int w){
  this.label=n;
  this.num=w;
}
```

```
public void agga(){
  num=num+2;
  System.out.println(num);
}

public static void main(String []args){
  Robot n1=new Robot("Nori", 51);
  Robot n2=new Robot("Alen", 22);
  System.out.println(n1.num);
  n2=n1;
  n1.agga();
  System.out.println(n2.num);
}
}
```

What will be the output of the following Java program fragments?

```
System.out.println("hello" + "there");
System.out.println(5/2);
System.out.println("run" * 2);
System.out.println("exec" + 3);
```

Python Questions

I would also like to understand what you know about Python. Again, please give your initial guess as to what each code fragment will do.

What will be the output of this Python program fragment?

```
x=0
while x<2:
  print("word")
  x+=1
```

What will be the output of this Python program fragment?

```
gap=3
gap="bye"
print(gap)
```

What will be the output of this Python program fragment?

```
for i in range(0,2):
  a="uk"
  print (a)
print (a)
```

What will be the output of this Python program fragment?

```
a="class"
b="class"
print(a==b)
```

What will be the output of this Python program fragment?

```
def cap(z,y):
  pa=z*y
  return pa
print (cap(3,4))
```

What will be the output of this Python program fragment? *

```
x = {'name':'Joseph', 'age': 51}
y = {'name':'Vic', 'age': 35}
print(x['age'])
y=x
x['age']=x['age']+1
print(x['age'])
print(y['age'])
```

What will be the output of the following Python program fragments? *

```
print("Good" + "morning")
print(5/2)
print("Good" * 3)
print("Good" + str(3))
```

Internet Technology
Programming concepts Quiz (Python and Java)

Prepared by Ethel Tshukudu
Email:
Centre for Computer Science Education
University of Glasgow

1. Write down precisely what would be printed when each of the following **Python code** snippets is executed. You may assume the code immediately below, to declare variables, has been executed.

```
a="Java"
b="Python"
c=5
d=2
x1=[9,4,1]
x2=[9,4,1]
```

Code snippet	Your answer here:
`print(x1==x2)`	`True`
`print(x2[0]+x2[2])`	`10`
`print(a + b)`	`JavaPython`
`print (a*2)`	`JavaJava`
`print (b+c)`	`error`
`print (c/d)`	`2.5`
`print (d*(c+d))`	`14`

2. What will be the output of the following **Python program** fragments?

Program Fragment	Your answer here:

a.
`a=1`
`a= "jump"`
`print (a)`
	`jump`

b.
`Sc1 = ["Axe"]`
`Sc2 = Sc1`
`Sc1.append("Wood")`
`print(Sc1)`
`print(Sc2)`
	`['Axe', 'Wood']`
	`['Axe', 'Wood']`

c. `for i in range(2):`
`print ("pen")`
`i=3`
	`pen`
	`pen`

a. `for i in range(2):`
`a=2`
`print (a)`
	`2`

b.
`i = 0`
`while (i < 2):`
`a="post"`
`print(a)`
`i+=1`
	`Post`
	`post`

Internet Technology
Programming concepts Quiz (Python and Java)

Prepared by Ethel Tshukudu
Email:
Centre for Computer Science Education
University of Glasgow

c.
`def pin(e,f):`
`  ga=e+f`
`  api()`
`  return ga`
 	
`def api():`
`  ga=3`
`  print(ga)`
 	
`print (pin(3,3))`
	3
	6

d.
`class Student:`
`  def __init__(self, name, mark):`
`  self._name = name`
`  self._mark = mark`
 	
`  def update(self):`
`  self._mark += 1`
`  print(self._mark)`
 	
`f1 = Student("James", 59)`
`f1.update()`
	60

e.
`n1 = {'name':'Joseph', 'age': 51}`
`n2 = {'name':'Vic', 'age': 35}`
`n1=n2`
`n2['age']=36`
 	
`print(n2['age'])`
`print(n1['age'])`
	36
	36

f.
`a=1;`
`b=2;`
`if (a>b):`
`  print("hello")`
`else:`
`  print("bye")`
	bye

Internet Technology
Programming concepts Quiz (Python and Java)

Prepared by Ethel Tshukudu
Email:
Centre for Computer Science Education
University of Glasgow

1. Write down precisely what would be printed when each of the following **Java code** snippets is executed. You may assume the code immediately below, to declare variables, has been executed.

```
String a="public";
String b="void";
int c=5;
int d=2;
int[] x1={3,2,1};
int[] x2={3,2,1};
```

Code snippet	**Your answer here:**
`System.out.println (x1==x2);`	`False`
`System.out.println (x2[0]+x2[2]);`	`4`
`System.out.println (a + b);`	`publicvoid`
`System.out.println (a*2);`	`error`
`System.out.println (b+c);`	`Void5`
`System.out.println (c/d);`	`2`
`System.out.println (d*(c+d));`	`14`

2. What will be the output of the following **Java program** fragments?

Program Fragment	**Your answer here:**

a.
`int a=1;`
`String a="hello";`
`System.out.println(a);`
	`error`

b.
`for (int i=0; i<2; i++){`
`System.out.println("hello");`
`i=3;}`
	`hello`

c.
`public static void main(String[] args){`
`for (int i=0; i<2; i++){`
`int b=2;}`
`System.out.println(b);}`
	`error`

d.
`int i = 0;`
`while (i < 3) {`
`int a=1;`
`System.out.println(a);`
`i++;}`
	`1`
	`1`
	`1`

e.
`int a=4;`
`int b=6;`
`if (a>b){`
`System.out.println("axe");}`
`else{`
`System.out.println("wood");}`
	`wood`

Internet Technology
Programming concepts Quiz (Python and Java)

Prepared by Ethel Tshukudu
Email:
Centre for Computer Science Education
University of Glasgow

f.
`public class Main {`
`public static int gen(int g, int s)`
`{`
`int a=g+s;`
`cen();`
`return a;`
`}`
`public static void cen()`
`{`
`int a=2;`
`System.out.println(a);`
`}`
`public static void main(String[] args)`
`{`
`System.out.println(gen(4,3));`
`}}`
	2
	7

g.
`public class Robot {`
`String name;`
`int age;`
 	
`public Robot(String n, int w)`
`{ this.name=n;`
`this.age=w;`
`}`
`public void agga()`
`{ age=age+1;`
`System.out.println(age);`
`}`
`public static void main(String []args)`
`{`
`Robot n1=new Robot("Joseph", 51);`
`n1.agga();`
`}}`
	52

h.
`import java.util.ArrayList;`
`public class Main {`
`public static void main(String[] args)`
`{`
`ArrayList<String> cars = new`
`ArrayList<String>();`
`cars.add("Audi");`
`ArrayList<String> cars2=cars;`
`cars2.add("Corolla");`
`System.out.println(cars);`
`System.out.println(cars2);`
`} }`
	[Audi, Corolla]
	[Audi, Corolla]

Internet Technology
Programming concepts Quiz (Python and Java)

Prepared by Ethel Tshukudu
Email:
Centre for Computer Science Education
University of Glasgow

Appendix C

Chapter 7 Appendix: Teachers and their Experiences Study Materials

This appendix includes the instruments that were used in Chapter 7 (Teachers' views on transfer). The appendix presents the details of the interview script used to interview the teachers.

Teacher's views on teaching second and subsequent programming Languages

This study investigates teacher's perspectives on programming language transfer, especially in their own experiences when teaching second and subsequent programming languages in the classroom. These findings can be used to propose teaching methods that help teachers interpret and improve their own classroom practices when teaching second/subsequent programming languages.23 years

Background questions:

- How many years of experience do you have teaching programming?

- What are your educational qualifications?
- What is the school level of students being taught (secondary/K12 or university)?
- How many programming languages do you teach? Which one is the current one?

Questions on transfer and teaching methods

Why do you teach a second/subsequent programming language course?

Do you teach both the first and second/subsequent programming language introduced to students? *(Please list them in order of first to current)*

What are the course outcomes? What knowledge do you wish your students would have acquired at the end of their first and second programming course? e.g. *(programming language comprehension, problem solving, Code writing, Code debugging, programming concepts knowledge)*

- If you want the course outcomes to be all of the above examples, you can arrange them by order of preference.

Take me through your teaching methods of both first and second programming language (where applicable)? E.g.

- Do you start teaching problem solving/ algorithms first then introduce the language later or vice versa
- lab activities focus on code comprehension/ problem solving

- discussions
- feedback
- homework (e.t.c)

What are your views on programming language transfer as students move to their second/subsequent programming languages? E.g.

- Do you think there are benefits of the first programming language knowledge when students learn the second/subsequent programming language?

- Do you experience any transfer problems in the classroom when teaching second/subsequent programming languages?

What measures or techniques do you currently use to help students transfer knowledge from first programming language to second programming language (if any)? E.g

- Do you usually teach the second programming language concepts from scratch assuming students don't know much from their first programming language?

- Do you see connections between programming concepts between the two languages you teach? Do you think students see them?

- Do teach students the second programming language referring back to the first language they learnt to make connections? If so how? If not why?

- Do you have any other strategies you use in the classroom when teaching a second programming language to connect students to their prior knowledge?

Appendix D

Chapter 8 Appendix: Exploring Transfer Interventions Study Materials

This appendix includes the instruments that were used in Chapter 8 (Exploring the transfer interventions). The appendix starts with the the transfer interventions activities in the classroom and then the quizzes. The quizzes include both the Python and the Java version of code comprehension questions.

Java
Pre Quiz

1. Write down precisely what would be printed when each of the following Java code snippets is executed. If you think that any expression would result in an error, explain clearly the cause of the error.

You may assume the code immediately below, to declare variables, has been executed.

```
int a = 3;
String b = "fi";
double c = 10.2;
int[] e = {1, 2, 3};
int[] f = {1, 2, 3};
```

```
a. System.out.println(e==f);

b. System.out.println(e[0]+e[2]);

c. System.out.println(a + b);

d. System.out.println(b + b);

e. System.out.println(a/2);

f. int sum = a + c;
   System.out.println(sum);

g. System.out.println(a*(a+2));
```

2.

a. What is the output when this **for-loop** program fragments are executed?

```
for (int i=0; i<2; i++){
  System.out.println("ola");
  i=3;}
```

b.

```
for (int i=0; i<2; i++){
   int a=3;}
System.out.println(a);
```

3. What will be the value of **sum** when the following code fragment is executed?

```
int sum =
0;
int i =
0; while
(i < 3)
{
      sum = sum + i;
       i++;
}
System.out.println(sum);
```

4. What will be the output when the following code fragment is executed?

```
public class HelloWorld
{
  public static int map(int g, String s)
  {
    int a=g+2;
    can();
    return a;
  }

  public static void can()
  {
    int a=3;
    System.out.println(a);
  }
  public static void main(String []args){
    System.out.println (map (7,"hello"));}
}
```

Java
Post Quiz

1. Write down precisely what would be printed when each of the following Java code snippets is executed. If you think that any expression would result in an error, explain clearly the cause of the error.

You may assume the code immediately below, to declare variables, has been executed.

```
String[] x = {"ab", "cd", "ef"};
String[] y = x;
```

h.
```
System.out.println(x==y);
```

i.
```
System.out.println(y[1]+y[2]);
```

j.
```
int a;
a=1;
a=10.5;
System.out.println (a);
```

k.
```
System.out.println("hello" + str(3));
```

l.
```
System.out.println("1" + "1");
```

m.
```
System.out.println(5/2*1);
```

n.
```
System.out.println(3-(1+2));
```

2.

a. What is the output when this **for-loop** program fragments are executed?

```
for (int i=0; i<2; i++){
  System.out.println("hello");
  i=3;}
```

b.
```
for (int i=0; i<=3; i++)
{
  int a=2;
  int b=4;
  System.out.println(a);
}
System.out.println(a+b);
```

3. What will be the value of **sum** when the following code fragment is executed?

```
int i =
0; while
(i < 2)
{
System.out.println("sum");
      i++;
}
```

4. What will be the output when the following code fragment is executed?

```
public class flo {
  public static int x=10;
  public static int rate(int p, int q)
  {
    int z=p+x;
    return z;
  }

  public static int flow(int p)
  {
    int deli=p*2;
    int prime=rate(p, deli);
    return prime;
  }
public static void main(String []args){
System.out.println(flow(2));
}}
```

Answer sheet for transfer interventions

Java	Answer	Python	Answer	Do they execute the same or different? Why?

public class HelloWorld
{
  public static void main(String []args){
  System.out.println (map (7,"hello"));
  }
 	
  public static int map(int g, String s)
  {
  int a=g+2;
  can();
  return a;
  }
 	
  public static void can()
  {
  int a=3;
  System.out.println(a);
  }
}
	3 9	def cap(z,y):
		 pa=z+2
		 zap()
		 return pa
		
		def zap():
		  pa=3
		  print(pa)
		
		print(cap(7,"hello"))		

Java	Answer	Python	Answer	Does the Java code snippet mean the same as the Python code? Why?

int a = 3;
String b = "fi";
double c = 10.2;
int[] e = {1, 2, 3};
int[] f = {1, 2, 3};
		a = 3
		b = "fi";
		c = 10.2;
		e = [1, 2, 3]
		f = [1, 2, 3]		

System.out.println(e==f);	**False**	print(e==f)		
System.out.println(e[0]+e[2]);	**4**	print(e[0]+e[2])		
System.out.println(a + b);	**3fi**	print(a + b)		
System.out.println(b + b);	**fifi**	print(b + b)		
System.out.println(a/2);	**1**	print(a/2)		
int sum = a + c; System.out.println(sum);	**error**	sum = a + c print(sum)		
System.out.println(a*(a+2));	**15**	print(a*(a+2))		

Java	Answer	Python	Answer	Do they execute the same or different? Why?

for (int i=0; i<2; i++){
System.out.println("ola");
i=3;
}
	ola	for i in range(2):
		print ("ola")
		i=3		

for (int i=0; i<2; i++){
int a=2;}
System.out.println(a);
}
	error	for i in range(2):
		a=2
		print (a)		

Java	Answer	Python	Answer	Do they execute the same or different? Why?

int sum = 0;
int i = 0;
while (i < 3){
  sum = sum + i;
  i++;
}
System.out.println(sum);
	3	total=0
		i = 0
		while (i < 3):
		  total +=i
		  i+=1
		print(total)		

Appendix E

Chapter 9 Appendix: Pedagogy for Transfer Study Materials

This appendix includes the instruments that were used in Chapter 9 (The transfer pedagogy). The appendix the quizzes. The quizzes include both the Python and the Java version of code comprehension questions.

Java Questions

I would like to understand your initial expectations about how Java works. For each question, please give your first guess what the code does.

What will be the output of this Java program fragment?

```
int a;
a=1;
a=10.5;
System.out.println (a);
```

What will be the output of this Java program fragment?

```
for (int i=0; i<2; i++)
{
  String var="Hello";
  System.out.println(var);
}
System.out.println(var);
```

What will be the output of this Java program fragment?

```
int i=0;
while (i<2){
  System.out.println(i);
  i++;
}
```

What will be the output of this Java program fragment?

```
String a=new String ("lab");
String b=new String ("lab");
System.out.println(a==b);
```

What will be the output of this Java program fragment?

```
public class Main{
public static int gen(int g, int s) {
int a=g*2;
return a;
}

public static void main(String[] args) {
System.out.println(gen(7,3));
}
}
```

What will be the output of this Java program fragment?

```
public class Robot {
  String label;
  int num;

public Robot(String n, int w){
  this.label=n;
  this.num=w;
}
```

```
public void agga(){
  num=num+2;
  System.out.println(num);
}

public static void main(String []args){
  Robot n1=new Robot("Nori", 51);
  Robot n2=new Robot("Alen", 22);
  System.out.println(n1.num);
  n2=n1;
  n1.agga();
  System.out.println(n2.num);
}
}
```

What will be the output of the following Java program fragments?

```
System.out.println("hello" + "there");
System.out.println(5/2);
System.out.println("run" * 2);
System.out.println("exec" + 3);
```

Python Questions

I would also like to understand what you know about Python. Again, please give your initial guess as to what each code fragment will do.

What will be the output of this Python program fragment?

```
x=0
while x<2:
  print("word")
  x+=1
```

What will be the output of this Python program fragment?

```
gap=3
gap="bye"
print(gap)
```

What will be the output of this Python program fragment?

```
for i in range(0,2):
  a="uk"
  print (a)
print (a)
```

What will be the output of this Python program fragment?

```
a="class"
b="class"
print(a==b)
```

What will be the output of this Python program fragment?

```
def cap(z,y):
  pa=z*y
  return pa
print (cap(3,4))
```

What will be the output of this Python program fragment? *

```
x = {'name':'Joseph', 'age': 51}
y = {'name':'Vic', 'age': 35}
print(x['age'])
y=x
x['age']=x['age']+1
print(x['age'])
print(y['age'])
```

What will be the output of the following Python program fragments? *

```
print("Good" + "morning")
print(5/2)
print("Good" * 3)
print("Good" + str(3))
```

Appendix F

Chapter 9: Ethics Documents

This appendix includes the Ethics application documents used for the experiments in this thesis.

University of Glasgow | College of Science & Engineering

Dr. Christoph Scheepers
Senior Lecturer

School of Psychology
University of Glasgow
62 Hillhead Street
Glasgow G12 8QB
Tel.: +44 141 330 3606
Christoph.Scheepers@glasgow.ac.uk

Ethical approval for:

Glasgow, October 11, 2021

Application Number: 300210041

Project Title: Understanding conceptual transfer in students learning new programming languages

Lead Researcher: Professor Quintin Cutts

This is to confirm that the College of Science and Engineering Ethics Committee has reviewed the above application and **approved** it. Please keep this letter for your records.

Please note that if your proposal involves **face-to-face research**, approval to carry out this research is only granted when one of the following two conditions has been met:

(a) You have performed a risk assessment of your research protocol in your research facility, had it approved by your Head of School / Director of Institute, and received permission to proceed with this specific research project, or

(b) The University has generally lifted its social distancing restrictions on face-to-face interaction, including research.

If any of the above is true, or if your research collects data in a format that does not require social contact (e.g., online research), you may begin data collection now. Approval for this project lasts for 6 months from the date you are allowed to proceed with data collection.

Also please download and read the Collated Comments associated with your proposal. This document contains all the reviews of your application and can be found below the approval letter on the Research Ethics System. These reviews may contain useful suggestions and observations about your research protocol for improving it. Good luck with your research.

Sincerely,

Dr Christoph Scheepers
Ethics Officer
College of Science and Engineering
University of Glasgow

Figure F.1: **Ethics approval**

University of Glasgow | College of Science & Engineering

Dr. Christoph Scheepers
Senior Lecturer

School of Psychology
University of Glasgow
62 Hillhead Street
Glasgow G12 8QB
Tel.: +44 141 330 3606
Christoph.Scheepers@glasgow.ac.uk

Ethical approval for:

Application Number: 300200021 Glasgow, October 9, 2020

Project Title: Understanding Conceptual Transfer for students learning new programming languages

Lead Researcher: Professor Quintin Cutts

This is to confirm that the College of Science and Engineering Ethics Committee has reviewed the above application and **approved** it. Please download the approval letter from the Research Ethics System for your records.

Please note that if your proposal involves **face-to-face research**, approval to carry out this research is only granted when one of the following two conditions has been met:

(a) You have a performed a risk assessment of your research protocol in your research facility, had it approved by your Head of School / Director of Institute, and received permission to proceed with this specific research project, or

(b) The University has generally lifted its social distancing restrictions on face-to-face interaction, including research.

If any of the above is true, or if your research collects data in a format that does not require social contact (e.g., online research), you may begin data collection now. Approval for this project lasts for 6 months from the date you are allowed to proceed with data collection.

Also please download and read the Collated Comments associated your application. This document contains all the reviews of your application and can be found below your approval letter on the Research Ethics System. These reviews may contain useful suggestions and observations about your research protocol for strengthening it. Good luck with your research.

Sincerely,

Dr Christoph Scheepers
Ethics Officer
College of Science and Engineering
University of Glasgow

Figure F.2: **Ethics approval**