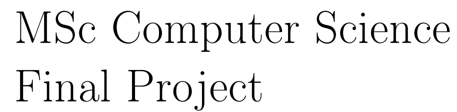


MSc Computer Science
Final Project

Code comprehension in the multi-paradigm environment Kotlin

Daniël Floor

Supervisor: Vadim Zaytsev, Rinse van Hees, Dennis Reidsma

February, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction **3**
- 1.1 Research questions . . . 4
- 1.2 Outline . . . 4

2 Programming paradigms **5**
- 2.1 Imperative programming . . . 5
  - 2.1.1 Procedural programming . . . 7
  - 2.1.2 Object oriented programming . . . 7
- 2.2 Declarative programming . . . 8
  - 2.2.1 Functional programming . . . 8
- 2.3 Multi-paradigm languages . . . 8

3 Programming constructs **10**
- 3.1 Object-oriented programming . . . 10
- 3.2 Functional programming . . . 11
- 3.3 FP support in OO languages . . . 13

4 Software quality assurance **15**
- 4.1 Maintainability . . . 15
- 4.2 Code Comprehension . . . 16
- 4.3 Comprehension Strategies . . . 16
  - 4.3.1 Bottom-up . . . 17
  - 4.3.2 Top-down . . . 17
  - 4.3.3 Knowledge-based . . . 17
  - 4.3.4 Systematic . . . 17
  - 4.3.5 As-needed . . . 17

5 Human study **19**
- 5.1 Study requirements . . . 19
  - 5.1.1 Quantitative analysis . . . 19
  - 5.1.2 Qualitative analysis . . . 20
- 5.2 Literature research . . . 20
  - 5.2.1 Selection procedure . . . 20
  - 5.2.2 Paper overview . . . 22
  - 5.2.3 Paper Categorization . . . 24
- 5.3 Findings . . . 26
- 5.4 Study design . . . 26
  - 5.4.1 Study Type . . . 26
  - 5.4.2 Language . . . 27

5.4.3 Decision . . . 28

6 Study setup 29
6.1 Participants . . . 29
6.1.1 Sample size . . . 29
6.1.2 Recruiting process . . . 29
6.2 Interview structure . . . 30
6.2.1 Equipment setup . . . 30
6.2.2 Interview start . . . 31
6.2.3 Interview tasks . . . 31
6.2.4 Post interview . . . 33
6.3 Methodology . . . 33

7 Interview questions & constructs 35
7.1 Multi-paradigm constructs . . . 35
7.1.1 Considered functional programming constructs . . . 35
7.1.2 Impure lambda functions . . . 36
7.1.3 Imperative lambda functions . . . 36
7.1.4 Encapsulated higher-order functions . . . 37
7.1.5 Branched pattern matching . . . 37
7.2 Interview Questions . . . 38
7.2.1 Question 1 . . . 38
7.2.2 Question 2 . . . 39
7.2.3 Question 3 . . . 39
7.2.4 Question 4 . . . 40
7.2.5 Question 5 . . . 40
7.2.6 Question 6 . . . 41
7.2.7 Question 7 . . . 42
7.2.8 Question construct usages . . . 42

8 Experiment results 44
8.1 Data collection . . . 44
8.1.1 Quantification comprehension strategies . . . 45
8.2 Quantitative results . . . 47
8.2.1 Demographic results . . . 47
8.2.2 Questions . . . 48
8.2.3 Versions . . . 51
8.2.4 Comprehension strategies . . . 52
8.3 Qualitative results . . . 53
8.3.1 Observations made . . . 53
8.3.2 Theory building . . . 55
8.3.3 Theory refinement . . . 56

9 Discussion 61
9.1 Implications . . . 61
9.1.1 Leveraging Long-Term Memory for Code Understanding . . . 61
9.1.2 Multi-paradigm code boundary threshold . . . 62
9.1.3 Combined cognitive processes trigger for successful code comprehension 62
9.2 Limitations . . . 62
9.2.1 Approach . . . 62

9.2.2 Language . . . 63
9.2.3 Threats to validity . . . 63
9.3 Future work . . . 64
9.3.1 More focus on multi-paradigm constructs . . . 64
9.3.2 Different kind of context . . . 64
9.3.3 Validate current findings . . . 64
9.3.4 More programming languages . . . 65
9.3.5 Bigger comprehension questions . . . 65

10 Conclusion **66**
10.1 RQ 1 . . . 66
10.2 RQ 2 . . . 66
10.3 RQ 3 . . . 67

A Information Letter **73**

B Consent form **75**

C Interview Questions **77**
C.1 Object-oriented questions . . . 77
C.1.1 Question 1 . . . 77
C.1.2 Question 2 . . . 77
C.1.3 Question 3 . . . 78
C.1.4 Question 4 . . . 79
C.1.5 Question 5 . . . 80
C.1.6 Question 6 . . . 81
C.1.7 Question 7 . . . 82
C.2 Multi-paradigm questions . . . 84
C.2.1 Question 1 . . . 84
C.2.2 Question 2 . . . 84
C.2.3 Question 3 . . . 85
C.2.4 Question 4 . . . 85
C.2.5 Question 5 . . . 86
C.2.6 Question 6 . . . 87
C.2.7 Question 7 . . . 88

Acknowledgements

I wish to express my sincere gratitude to Info Support and all individuals who played a pivotal role in the realization of this thesis, even with a slight extension. A special acknowledgment goes to Rinse van Hees for his invaluable guidance and unwavering supervision, which significantly contributed to the successful completion of this research. His expertise and commitment have been truly instrumental, and I extend my heartfelt thanks for his dedicated input.

I also want to express my gratitude to my fellow graduates at Info Support for contributing to a conducive work environment that fostered collaboration and productivity. Especially to Jochem and Jeffrey for allowing me to drive with them back and forth between Veenendaal and Enschede. This allowed me to have two extra free evenings throughout the week.

I am deeply grateful to my parents for everything they have done for me, their unwavering support and assistance on weekdays played a crucial role in sustaining my motivation throughout this academic journey. Their constant encouragement and belief in my abilities have served as an endless source of inspiration.

To my loving girlfriend, Anissa, I extend my deepest appreciation. Her unwavering support, characterized by attentive listening, motivational words, and the creation of delectable meals, has been an anchor during challenging times. I am genuinely thankful for her unwavering presence and encouragement, particularly during the summer when my motivation was at an all-time low.

I extend my gratitude to my academic supervisors at the University of Twente, Vadim Zaytsev, and Dennis Reidsma. Their insights and perspectives on my research were invaluable, guiding me to relevant literature that significantly enriched my work. I offer special thanks to Vadim for his support in presenting my early ideas at the SATToSE conference in Italy, a valuable experience that deepened my understanding of academic discourse and made me grow as a person.

A sincere thank you is extended to the members of my badminton association, DIOK. The positive and engaging environment provided a sanctuary during the year-long journey of working on this thesis, contributing significantly to maintaining my sanity and overall well-being.

In conclusion, I extend my heartfelt appreciation to everyone mentioned above and to the collective support that has shaped this thesis. Your contributions, whether academic or personal, have left a mark on this research. Thank you for being part of my academic roller coaster.

yours truly,
Daniël Floor

Chapter 1

Introduction

Each software problem has its own needs and requirements. These needs can be satisfied by one of many programming paradigms, which in turn can be realized by one of many programming languages. With the increasing need for more complex systems, the demands can be satisfied by the use of multiple programming paradigms, allowing for a fitting solution to the problem. A programming paradigm can be viewed as a set of concepts [60]. These paradigms in turn can then once again be implemented by multiple programming languages. Languages that implement multiple paradigms, these languages are also called multi-paradigm languages. For the usage of multi-paradigm programming, there is a distinction between two types of usages.

1. Parallel usage: This entails the usage of multiple paradigms in one program, where there is a clear separation between the usage of the paradigms.

2. Mixed usage: This entails multi-paradigm code blocks where the different paradigms are mixed. A piece of code thus contains code written in multiple paradigms.

For parallel multi-paradigm usage, the program's responsibilities are separated. It is still possible to evaluate the single paradigm code blocks separately. With mixed programming usage there no longer is a clear separation, possibly making the process of understanding the code harder and more time-consuming. One prominent multi-paradigm combination is object-oriented programming and functional programming. Previous research on this multi-paradigm combination has focused on fault-proneness and defining new code metrics for mixed usage of programming paradigms [30, 37, 57, 66] or creating a language agnostic code quality framework for multiparadigm languages. The code comprehension side of multi-paradigms is yet to be researched in depth. Code comprehension focuses on the process of understanding the behavior of the code.
Code comprehension is an important part of ensuring code quality and maintainability. Code comprehension is a process of understanding the behavior and functionality of the source code. Poor code comprehension can lead to a maintainer not being able to work efficiently as a result of poor code quality. To establish code quality there is a standard that explains how code quality is measured [29]. One of the key aspects of this standard is maintainability. While the standard does not directly mention code comprehension, code comprehension still has a big impact on maintainability. Without understanding the code, a codebase becomes more difficult to maintain properly.

We give a first insight into the implications of using multi-paradigm constructs on code comprehension. Additionally, we give insights into the different comprehension strategies that are used while solving comprehension tasks. The focus lies on object-oriented programming languages that have incorporated functional programming concepts and con-

structs. In a multi-paradigm perspective, there are two different kinds of paradigms, the first paradigm is chosen for the problem most frequently targeted by the language. Whereas, the second paradigm is chosen to support abstraction and modularity which fills the gaps the first paradigm leaves open [60]. The modularity of a language has been described as key to successful programming [28]. There are abundant programming languages that allow for object-oriented programming combined with functional programming constructs, but the focus of this study will be on Kotlin. The language combines the two paradigms naturally and harmoniously. To conduct a meaningful experiment first a set of multi-paradigm constructs is defined. Additionally, a literature study is performed to gain insight into what type of study is suitable for capturing comprehension strategies. An interview that captures both quantitative and qualitative data is performed on 30 participants who were challenged with 7 comprehension questions.

1.1 Research questions

To provide structure to the research, the following research questions are established. These questions are answered in the remainder of the thesis.

- **RQ1: Which multi-paradigm constructs can be identified when combining object-oriented programming and functional programming to study code comprehension?**
- **RQ2: How can we study the impact on code comprehension in multi-paradigm programs?**
- **RQ3: What is the impact of multi-paradigm programming on code comprehension in Kotlin?**

1.2 Outline

In this section, we describe the way the document is set up. The first few chapters describe the background. In chapter 2 we highlight and discover the different kinds of paradigms and the concept of multi-paradigm languages. In chapter 3 the different kinds of programming constructs for object-oriented programming and functional programming are highlighted. Chapter 4 covers software quality and the different kinds of comprehension strategies. Chapter 5 describes the literature study performed, describing what kind of study setup is required for looking into the impact of multi-paradigm programming on code comprehension. Chapter 6 describes the setup and structure of the performed interviews. Chapter 7 describes our defined multi-paradigm constructs and highlights the code questions used in the interviews. Chapter 8 describes the quantitative results and the qualitative results of the interviews. Chapter 9 describes the discussion, containing the implications of the results, the limitations of the research, and potential future work. Lastly Chapter 10 concludes the findings of each of the research questions.

Chapter 2

Programming paradigms

Programming paradigms, e.g. object-oriented programming, can be viewed as a categorization and grouping of a set of concepts that guide the development of software. Each paradigm is associated with a distinct set of principles and techniques that can be realized through a programming language. Such a language can in its turn realize more than one paradigm. This creates a hierarchy with endless possibilities. Van Roy's visualization [60], seen in Figure 2.1, highlights the wide range of combinations that are possible.

Despite the distinct set of concepts of a paradigm, there are often common grounds between paradigms. A taxonomy, a way to classify the different paradigms, can be constructed of the programming paradigms that display the relations between the paradigms [60]. This taxonomy can be seen in Figure 2.2.

A programming language is not restricted to realizing only one paradigm and can realize two or even more. These kinds of languages are called multi-paradigm languages, think of most object-oriented programming languages that support functional programming constructs(Java, C#, Kotlin or Python). As demand for increasingly complex systems grows, the need for multi-paradigm programming languages has become more prevalent. These languages enable developers to select the best paradigm for a given task, resulting in greater flexibility and expressiveness in code.

Within the taxonomy of figure 2.2 two primary categories of paradigms can be distinguished: declarative programming and imperative programming. While these are not the only paradigms, most other paradigms are based on either one of the two paradigms. Understanding the strengths and weaknesses of each paradigm, and how they can be combined, can lead to the development of powerful languages with a multitude of possibilities [60]. To understand why the combination of two paradigms, object-oriented programming, and functional programming, creates a powerful combination. The next sections will delve more into understanding the differences between imperative programming and declarative programming. For the languages the different kinds of constructs are not yet discussed, this will happen in chapter 3.

2.1 Imperative programming

Imperative programming is a programming paradigm that focuses on statements that modify the state of a program. Programs in this paradigm are constructed using a sequence of statements executed in a specific order, with each statement altering the state of the program. These alterations can either change a variable or affect the program's environment. Two well-known imperative programming paradigms are procedural programming and object-oriented programming.

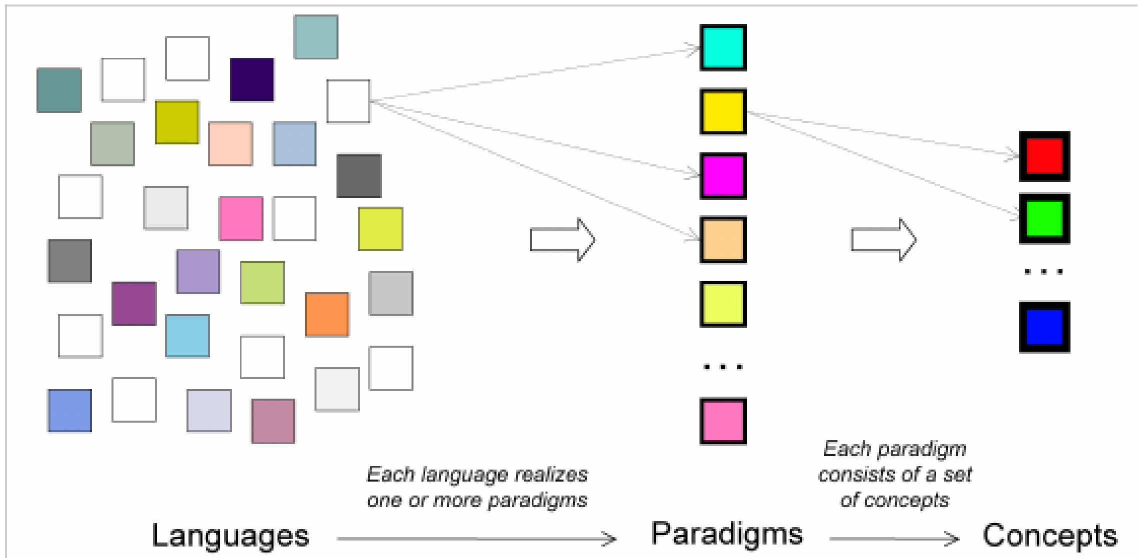


FIGURE 2.1: Languages, paradigms & concepts [60]

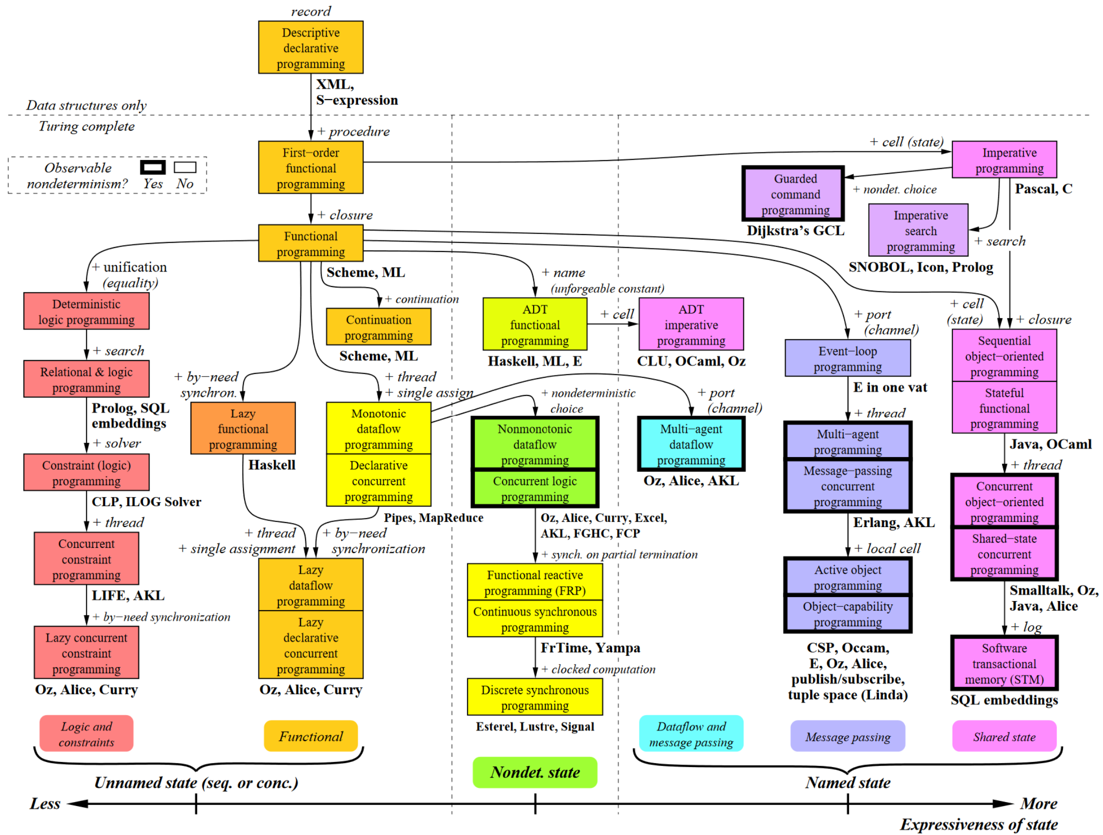


FIGURE 2.2: Programming paradigms taxonomy [60]

Imperative programming has been widely used in the development of systems that require precise control of program flow and state. However, this paradigm can often result in code that is difficult to read and maintain, especially as programs grow larger and more complex. Despite its limitations, imperative programming remains an important programming paradigm due to its wide usage in real-world systems. Understanding the principles and techniques of this paradigm can provide developers with valuable insights into designing and implementing effective software.

Procedural programming is one example of an imperative programming paradigm, where programs are constructed using procedures or subroutines that perform a specific task. The procedure is executed in a step-by-step manner, with each statement modifying the state of the program until the desired output is achieved.

Another well-known imperative programming paradigm is object-oriented programming (OOP), more on this in Section 2.1.2, which emphasizes the creation of objects that encapsulate data and behavior. Objects interact with each other by sending messages and invoking methods, which modify the state of the objects.

2.1.1 Procedural programming

Procedural programming is a programming paradigm that revolves around the concept of procedures, which are also known as subroutines or functions. Procedures are small sections of a program that perform a specific task. Procedural programming supports features that alter the control flow such as if-statements and loops (for and while). Any kind of procedure may be called by another procedure at any time, giving it a wide variety of possible applications.
One of the first languages to adopt the procedural programming paradigm was ALGOL, which introduced the concept of block structure and the use of subroutines to make programs modular. The C programming language, developed in the 1970s, also popularized the use of procedural programming and is widely regarded as one of the most influential programming languages of all time [46]. In the figure procedural programming is not necessarily listed, but instead, it falls under just the imperative programming block.

2.1.2 Object oriented programming

Object-oriented programming is a paradigm in software development that revolves around the concept of objects, which are instances of classes. The term was first introduced by Kay in 1967 [56]. OOP or how he described it was:

> "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."

If we look at a more recent and better definition of OOP, which is defined in the book "Object-oriented Analysis and Design with Applications" it is defined as follows [9]:

> "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships"

It is evident that the view on OOP has shifted through time, but objects will be the center of the paradigm. OOP facilitates the use of abstraction, which involves defining the essential characteristics of an object while hiding unnecessary implementation details. Additionally, OOP makes heavy use of designing maintainable code using loose coupling

and having high modularity. In figure 2.2 we can see that OOP is on the right side of the spectrum, meaning that no paradigm expresses the state of the program more than OOP, this is combined with named states and closures.

2.2 Declarative programming

Where imperative programming focuses on state changes, declarative programming focuses on specifying the problem that has to be solved. This can either be expressed as a set of logical or mathematical rules that describe what the desired outcome is. The result of this is an implementation that has a higher level of abstraction. The benefit of this is that code is much more readable and easier to understand. This improves the time one needs to write a program. The drawback of this is that declarative programming usually is not properly optimized requiring lots of resources.

2.2.1 Functional programming

Functional programming (FP) is a paradigm that uses functions to make computations. This approach for programming is based on lambda calculus, which is a mathematical theory about functions developed in the 1930s [15]. A program is defined as a function call, where each function in its turn also calls other functions. One of the most significant characteristics of functional programming is that the functions avoid altering the state of the program and do not contain side effects. This can also be categorized as functional purity. A function is only pure if it does not alter the state of a program.

One of the strengths of FP is the high modularity of the programs [28]. Due to the high modularity, it is easy to define new components (functions in this case), without changing the functionality. This high modularity is possible with the introduction of higher-order functions and lazy evaluation, but more on this in chapter 3. The completely different approach of functional programming complements object-oriented programming enabling different approaches and implementations. More on these differences in Chapter 3.

2.3 Multi-paradigm languages

With the various wildly different programming paradigms explained, we can also support more than one paradigm in one language, and these languages are called multi-paradigm languages. Within a multi-paradigm language, the first paradigm is considered to be the paradigm that is most frequently targeted by the language to solve a problem. The second paradigm is chosen to support abstraction and add modularity to the language [60]. The combination of paradigms that we will cover and focus on in our research will be object-oriented programming with functional programming. The adaptation of using functional programming constructs in object-oriented programming continues to grow and more and more languages start supporting the usage of these constructs, think of languages as but not limited to are: Python, Java, C#, Kotlin, and Scala. We chose to research the support for functional programming in the languages Java, C#, Kotlin, and Scala. This is because Java and C# are rather similar in OOP style and Java, Scala, and Kotlin are JVM languages. It is good to note that Java and C# have a similar approach to the functional programming constructs, where they are additions to the already existing object-oriented language features. Scala and Kotlin are slightly different, these languages were designed as hybrid languages in such a way functional programming and object-oriented programming

are possible. They do not only support functional programming constructs but are also able to support code that is completely written in a functional style.

Chapter 3

Programming constructs

Within the different kinds of programming paradigms, there exist different kinds of programming constructs. Both object-oriented programming and functional programming are based on a set of concepts. We will list the different concepts for both object-oriented programming and functional programming.

3.1 Object-oriented programming

In object-oriented programming, there are a few core concepts that are inherently OOP. These core concepts of object-oriented programming are as follows [9]:

- `Encapsulation`: In OOP classes are used to encapsulate data and methods that function on this data. This is then used to protect private information and only expose the things that should be available publicly.
- `Inheritance`: Classes can inherit, partially, the functionality of other classes. The depth of inheritance is limitless. Inheritance enables code reuse, as the subclass can reuse the code of the superclass, and also provides a way to extend and modify existing classes without having to rewrite them from scratch. With this, a hierarchy of classes can be established where subclasses can extend the functionality of a superclass. This makes code more modular and better maintainable. A simple example of inheritance can be explained as follows. The class `Dog` extends the class `Animal`. The class `Animal` has a method `eat()` which the class `Dog` inherits and can also call this function. The function for `Dog` has the same behavior as with an object of class `Animal`. Additionally, `Dog` contains a method that `Animal` does not have namely `bark()`.

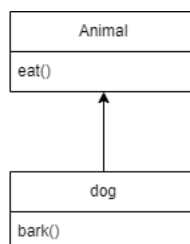


FIGURE 3.1: Inheritance of a Dog

- `Polymorphism`: This describes the concept that allows objects of a different type to be treated as if they are the same. Think of a class `Dog` and a class `Cat` that extend a

class Animal. Animal contains a method getName() with a standard implementation. Both Dog and Cat class override the implementation of Animal.

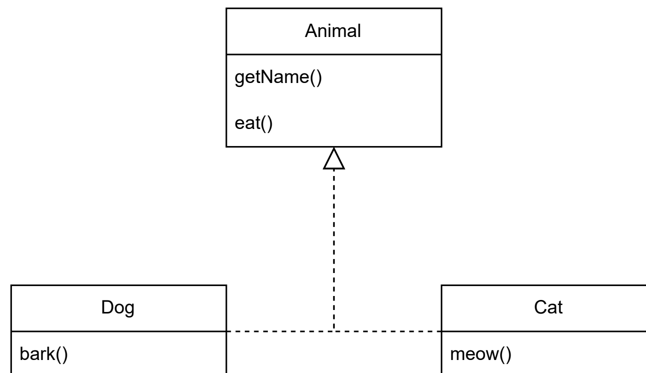


FIGURE 3.2: Polymorphism of a dog and cat

3.2 Functional programming

The origin of functional programming lies in lambda calculus [15]. In this paradigm, programs are constructed by the application and composition of functions. All functional programming examples are written in Haskell to display what pure functional programming looks like. The main concepts of functional programming are defined as follows [28]:

- `First class & Higher-order functions`: Within functional programming functions serve as a first-class citizen, meaning that they can serve as a variable, be passed on as arguments to other functions, or be the return value of a function. These functions that take functions as arguments are called higher-order functions. An example of a higher-order function is the function map. `map` takes a function and applies this function to each element in a list. An example usage of a higher order function can be seen in Listing 3.1.

  LISTING 3.1: Map example in Haskell

  ```
  addOne :: (Num a) => [a] -> [a]
  addOne xs = map (+1) xs
  addOne [2,3,4,5] == [3,4,5,6]
  ```

  The function `addOne` takes as argument a list and results in a list where each item of the list has been incremented by one. The second line of the Listing shows what the result will be. By using higher-order functions, code becomes more compact and its generality increases the possible functional applications of the program.

- `Referential transparency`: Referential transparency is the property that allows the replacement of an expression with the computed value of the expression, or the other way around, and does not alter the outcome of the program.

- `Recursion`: Recursion is a programming technique in which a function keeps calling itself. When a function calls itself, a new instance of the function is created and the

process continues until a certain condition is met, the base case. In Listing 3.2 a small Haskell program has been given that calculates the sum of all elements in the list.

LISTING 3.2: Recursion example in Haskell

```
mySum :: (Num a) => [a] -> a
mySum [] = 0
mySum (x:xs) = x + mySum xs
```

- `Lazy evaluation`: With lazy evaluation, expressions are not evaluated until their results are required. Instead of evaluating/calculating the entire expression a program only evaluates the necessary expressions. This makes it possible to construct infinite data structures. Listing Listing 3.3 shows the power of lazy evaluation, it creates an infinite list that contains only ones. With the power of lazy evaluation, it is possible to keep generating a list containing only ones and it will never end. It is important to mention that not all functional programming languages have lazy evaluation, an example of this is Isabelle[1]. Isabelle is a generic proof assistant that proves termination rules.

LISTING 3.3: Lazy evaluation example in Haskell

```
ones :: [Int]
ones = 1:ones
```

While these are the core concepts for functional programming, there are still other concepts that are used in functional programming.

- `Anonymous functions`: an anonymous function is a function that does not carry a name. Such functions can not be referenced by other parts of the code. The functions serve as a way to write compact code that is only required for parts of the code.

- `Currying`: Currying is a technique in functional programming that transforms a function that takes multiple arguments into a sequence of functions that each take a single argument. Currying is named after mathematician Haskell Curry, who used the concept extensively in the 20th century. With the introduction of currying, another functional programming concept becomes available namely `Partial application`.

- `Partial application`: Partial application refers to fixing a number of arguments of a function resulting in another function that takes fewer arguments. This becomes possible when combining both currying and higher-order functions, where functions can also serve as variables. In the Listing 3.4 function add takes two arguments, namely the two numbers that need to be added. The function addOne on the other hand returns a function that only takes 1 argument. The function returns the function add where its first argument is already fixed to one.

LISTING 3.4: Lazy evaluation example in Haskell

```
ones :: (Num a) => a -> a -> a
add x y = x + y

addOne => (Num a) => (a -> a)
addOne = add 1
```

[1] https://isabelle.in.tum.de/

- `Pattern matching`: With pattern matching, it is possible to distinguish the behavior of a function based on which patterns the input matches. This allows comparing values against certain patterns which then influence the outcome of the function call. Listing 3.2 shows an example of pattern matching. It distinguishes two patterns, namely the empty list or a list with at least one element. Using this structure it is possible to clearly distinguish cases and allow for readable recursive code to be developed.

3.3 FP support in OO languages

All of the previously described concepts and constructs are clear with singular usage, but the expected behavior becomes more blurry once combining multiple constructs. This is something that will be researched in the final project. Still, it is important to know which concepts are supported by the four languages considered for this research: C#, Java, Scala, and Kotlin. The current support of functional programming concepts is listed in Table 3.1.

Language support				
	Java	Scala	C#	Kotlin
Recursion	**1**	**2**	**1.0**	**1.0**
Referential transparency	**1**	**2**	**1.0**	**1.0**
Higher-order functions	**8**	**2**	**1.0**	**1.0**
First-class functions	**8**	**2**	**1.0**	**1.0**
Anonymous functions	**8**	**2**	**3.0**	**1.0**
Currying	**8**	**2**	**3.0**	**1.0**
Lazy evaluation	**8**	**2**	**3.0**	**1.0**
Pattern matching	**7**	**2**	**7.0**	**1.0**
Partial application	**8**	**2**	**7.0**	**1.0**

TABLE 3.1: Functional programming support OOP languages

While all of the languages do support referential transparency this is heavily dependent on the methods/functions. As we defined it before for something to be referentially transparent, you should be able to interchange a method for the value it returns without altering the outcome of the program. This is still possible in all three of the languages but is only partially supported since none of the languages is pure. So they do support it, but only in a limited fashion. Therefore, a maintainer must be very careful when writing/altering code and check for purity and immutability. All the other constructs are supported, where things such as currying for C# and Java need to be very explicit while Scala is much closer to languages like Haskell, where it is the standard. But this is because Scala differs from C# and Java in how functions are used. In both Java and C# they have to be explicitly encapsulated by a Function construct, while in Scala they are completely regarded as just a variable, without needing such a construct. The following Listings display the difference in how functions work for the languages and how currying looks.

LISTING 3.5: Functions in Scala

```
val sum: (Int, Int) => Int = (x, y) => x + y
val curriedSum: Int => Int => Int = x => y => x + y
val curriedSum2: Int => Int => Int = sum.curried
val addOne: Int => x => x + 1
```

Listing 3.6: Functions in Java

```
Function<Integer, Integer> Add = (u,v) -> u + v;
Function<Integer, Function<Integer, Integer>>
                  curryAdd = u -> v -> u + v;
Function<Integer, Integer> > curryAddOne = curryAdd.apply(1);
```

Listing 3.7: Functions in C#

```
Func<int, int, int> add = (a, b) => a + b;
Func<int, int, Func<int, int> addCurr = a => b => a + b;
Func<int,int> addCurrOne = addCurr(1);
```

Lastly, pattern matching for all four of the languages is possible, but not in the way pure functional programming languages use it. In all four instances, it can be achieved through a switch or case statement that describes the different kinds of patterns possible. Now combining these functional programming constructs into an object-oriented environment increases the versatility of solutions. While on the surface this does look like a good addition, it is important to ask the question of whether the code remains maintainable. Combing multiple paradigms into one piece of code could reduce the ability to understand the code. Less understandable code leads to higher maintenance costs since it takes up more time.

As mentioned in the introduction we distinguish two different cases of multi-paradigm usages namely: parallel usage and mixed usage. Our research aims to focus on the mixed usage of multi-paradigm code since we expect the highest change in code comprehension here. Code that separates the usage of OOP and FP can be analyzed using single paradigm metrics [14, 49]. This gives a better understanding of the quality of the code. When mixing the two paradigms in the code there no longer is a clear separation of paradigms and we expect that it requires additional reasoning of the maintainer to try to comprehend the code, and is therefore something to research.

Chapter 4

Software quality assurance

We have now discussed the different kinds of paradigms and the constructs that they use. While the chosen language influences the effectiveness of a solution other factors determine whether the produced software is of a good quality. Assessing the quality of software is therefore an important part of a development cycle. To assess the quality of software there is the process of Software Quality Assurance (SQA) that ensures that software products meet the specified quality standards and requirements. SQA stems from early ideas in the '50s and has since undergone subsequent extensive exploration and research [11]. During this time it became more apparent that there was a need for quality assurance. In the years thereafter more research on quality assurance was performed. Some areas that were explored were software inspections [24, 21], software testing [32], and many other factors that are more aimed at software processes than just the code itself. Later on, a handbook describing all aspects of SQA came out with extensive descriptions [51]. While many aspects are covered, we are interested in software quality. The ideology regarding software quality is described in a standard [29]. It describes eight characteristics that influence the quality of a software product: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. When looking specifically at the influence of quality on maintenance tasks, compatibility, maintainability, and portability remain. We will take a closer look into maintainability and what it is influenced by.

4.1 Maintainability

Maintainability is defined as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainer" [29]. So the focus heavily lies on the degree a maintainer is affected by the quality of code. The standard describes five sub-characteristics that fall under maintainability.

- **Modularity**: Modularity is the degree to which distinct components impact other components. Higher modularity means that components are less dependent on the functionality of other components. Higher modularity makes it easier to maintain and replace single components and makes it easier to oversee the project.

- **Reusability**: Reusability is the degree to which can be used in more than one system. By making code as general as possible it becomes possible to reuse code in other systems or other parts of the code. By doing so similar functionality is all in one place making it easier to maintain.

- **Analysability**: Analysability is the degree of effectiveness and efficiency in the assessment of the impact of a system of intended changes, or diagnose deficiencies in a system. This is a very important part for maintainers, when unable to analyze code it becomes impossible to identify errors in the code or even reason about its behavior.
- **Modifiability**: Modifiability is the degree to which a product can be changed without introducing defects or decreasing the quality of the existing product. To modify a program a maintainer must be able to reason about the code and understand its behavior.
- **Testability**: Testability is the degree to which test criteria can be established for a system and tests can establish whether the criteria are met. Without testing, it is harder to assess the correctness of a program. Therefore, is an important aspect that influences the ability of a maintainer to perform its tasks.

While each characteristic focuses on different aspects and impacts on the maintainer there is a common ground for most of them. We identify an underlying and recurring pattern that is required for a maintainer. A maintainer needs to be able to reason about the code and understand its behavior. This is especially prominent in the Modularity, Analysability, and Modifiability. Without an understanding of a program, a maintainer is unable to perform its tasks and is therefore an important and noteworthy aspect of quality assurance.

4.2 Code Comprehension

In the previous section, the significance of comprehending code was highlighted as an essential aspect of a maintainer's responsibilities. This comprehension, referred to as program comprehension, entails the process through which software engineers gain an understanding of a software system's behavior by primarily referencing the source code [5]. While program comprehension encompasses a broader scope, code comprehension specifically concentrates on comprehending smaller components of the software system rather than the system as a whole. Code comprehension can therefore be seen as a part of the entire program comprehension process.

Furthermore, program comprehension has been recognized as a substantial component of maintenance costs, accounting for a considerable portion ranging from 50% to 90% of these expenses [47] For this reason, it is clear that code comprehension plays a central role in maintaining code and thus code quality. While there is a dedicated conference regarding program comprehension, there is still little research focusing on the program comprehension side of multi-paradigm languages. Part of this is due to little research aimed at understanding the impacts of combining multiple paradigms.

4.3 Comprehension Strategies

Comprehending code has one goal and that is to understand the purpose of the code. Although it may appear obvious and straightforward, the process of comprehending code varies among individuals, as each person employs their own unique comprehension strategy. Multiple comprehension strategies exist, each approaching the task of comprehension in distinct ways.

4.3.1 Bottom-up

The bottom-up comprehension strategy, initially proposed by Mayer et al [53], encourages a step-by-step approach to comprehension. This strategy involves reading the source code and mentally grouping the low-level software components into higher-level abstractions. These abstractions serve as "chunks" to construct a comprehensive understanding of the program. The primary objective of this strategy is for programmers to develop an internal representation of the program, focusing on grasping its underlying concepts rather than memorizing the syntax of the code. As additional layers of comprehension are added, this internal representation is expanded and refined.

4.3.2 Top-down

The top-down strategy as the name suggests is the complement of the bottom-up strategy. The top-down strategy starts with gaining a high-level understanding of the program [10]. Brooks describes the top-down strategy as a hypothesis-driven strategy. General hypotheses keep being refined as more information is extracted from the source code and its documentation. Once the high-level understanding is established, maintainers narrow their attention to specific sections of the code that are relevant to their comprehension goals. They proceed by delving into lower-level details, such as individual functions or code blocks, to understand the implementation specifics and how they contribute to the overall behavior.

4.3.3 Knowledge-based

The bottom-up and top-down strategies for program comprehension are not mutually exclusive and are commonly employed together. According to Letovsky [39], a knowledge-based comprehension strategy involves the creation of a mental model that represents the programmer's current understanding of the code. This mental model is constructed through an assimilation process that incorporates elements of both bottom-up and top-down comprehension strategies. As the mental model evolves, both strategies contribute to its development, allowing for a comprehensive understanding of the code.

4.3.4 Systematic

The systematic comprehension strategy is a methodical approach to program comprehension described by Littman et al [40], where maintainers follow a predefined and structured process to understand the software system. By tracing the flow of data through the program, maintainers gain insights into the sequence of steps the program takes and how these steps are interconnected. This systematic tracing allows maintainers to map the behavior of the entire program. This is, therefore, a more useful strategy for larger projects and much less for projects with a smaller codebase.

4.3.5 As-needed

The as-needed comprehension strategy, described by Littman et al [40], presents a dynamic approach to program comprehension, where maintainers purposely concentrate on particular code segments and details while performing maintenance tasks. This strategy is guided by the direct need to understand specific aspects of the code, prioritizing relevance and significance. Rather than adhering to a predetermined top-down or bottom-up

sequence, maintainers adjust their comprehension efforts according to the code's context and complexity, addressing specific requirements as they are encountered.

Chapter 5

Human study

The primary objective of this thesis is to provide valuable insights into the influence of multi-paradigm usage on code comprehension by performing a user study. This chapter covers the process of designing the human study. We begin by discussing the established requirements and essential components necessary for the study. Additionally, we explore existing research on the design and customization of human studies for code comprehension through a thorough literature review, which serves as the foundation for informed decision-making throughout the study design process. Following the literature review, we present the design of the human study, articulating the chosen methodology and the rationale behind these design decisions.

5.1 Study requirements

The design of the study needs to meet specific conditions to ensure the results hold meaningful insights. This is essential because we want the outcomes of the study to shed light on how using multi-paradigms might affect a programmer's code comprehension. Achieving this involves gathering data that lets us compare the experiences of different participants, and that's where quantitative data comes into play. However, merely comparing numbers doesn't provide enough to make meaningful conclusions. The heart of the matter lies in grasping the cognitive processes of participants. While we're not just concerned about where participants initially focus their attention, we're more intrigued by how they reason and progress in their thinking. This aligns well with the diverse comprehension techniques discussed in Chapter 4.3. However, it is crucial to note that these cognitive dimensions, while intriguing, rely on quantitative data to give weight to our findings from observations that are made. That's why it is crucial for the study, in whatever form it takes, to allow for both quantitative and qualitative analyses. This double approach doesn't only enhance the credibility of our findings but also helps us dig deeper into our understanding. In the upcoming sections, we'll delve into the areas that warrant measurement within the study. This covers both the quantitative and qualitative data elements.

5.1.1 Quantitative analysis

Quantitative data is a type of information that can be expressed in numerical terms and is something that can be measured may it be on a scale or not. Quantitative data relates to quantities, amounts, and objective measurements, making it ideal for mathematical and statistical analyses. In research and analysis, it is essential to provide empirical evidence, allowing researchers to draw objective conclusions based on measurable facts. This data

is often acquired through means such as surveys, experiments, observations, and measurements. Examples of quantitative data include age, height, weight, test scores, sales figures, and occurrence counts. Relating quantitative data to research on code comprehension some measurements should be taken into account. When comparing different things and their effect on comprehension, standard things to measure are, correctness and time to complete.

Correctness refers to the correctness of an answer given by a participant, may it be an open answer or a closed one. Correctness is measured using the two options correct or incorrect, in analyses this is usually described with a 0 for an incorrect answer and a 1 for a correct answer.

Time to complete refers to the time a participant is required to answer a question. This gives an insight into what possible factors are that could influence, positively or negatively, the time to comprehend relevant code pieces to answer a question.

Besides these two measurements, relevant demographic information of participants will be captured. This information can, later on, be used to distinguish different groups and make more specific conclusions.

5.1.2 Qualitative analysis

Unlike quantitative data, which is expressed in numbers, qualitative data is descriptive by nature. It deals with qualities, characteristics, attributes, and subjective observations. This type of data is usually captured in textual or narrative form, which allows researchers to better understand the complexities of human experiences, behaviors, and perceptions.

Qualitative data is particularly useful for exploring nuances, contexts, and underlying motivations that quantitative data may not fully capture. It helps researchers gain a deeper understanding of human behavior, attitudes, and cultural contexts. When contextualizing this within the scope of our research objectives, investigating behavioral patterns concerning comprehension emerges as a compelling area of exploration. Measurements described in the previous section can give an insight into whether certain hypotheses are correct.

Concentrating on the subjective observations gathered during the conducted study facilitates the analysis of participants' cognitive thinking. Therefore, the design of the study should allow for the gathering of qualitative data that entails the cognitive thinking process endured by the participant. It would be interesting to see whether these observations can be linked to previously identified potential comprehension techniques.

5.2 Literature research

To conduct a proper and representative study on the effects of multi-paradigm programming on code comprehension, it is important to understand how previous human studies(studies involving human participants) on program comprehension have been carried out. By examining how other studies approached comprehension, we can make informed decisions when designing our study. it is also of the essence to consider the goals and scope of those past studies in our analysis.

5.2.1 Selection procedure

We collected relevant papers on this topic to review human studies on program comprehension. We focused on papers presented at the International Conference on Program Comprehension, from its 2nd to 30th editions. Going through this substantial literature required a structured approach to ensure we didn't miss any relevant papers. Our selection procedure consisted of three phases:

1. Initial Selection: We first considered all papers based on their titles to identify potentially relevant ones. If the abstract's initial sentences weren't clear, we read further to decide.
2. Abstract Analysis: In this phase, we carefully read the abstracts to check if the papers indeed involved human studies. If there was uncertainty, we looked for any mention of human studies in the papers themselves.
3. Inclusion Criteria Refinement: The remaining papers were examined more closely to ensure they met our specific inclusion criteria for human studies.

Initial selection

Each phase had its own set of inclusion and exclusion criteria. In the initial phase, we focused on inclusive factors derived from the papers' titles, without any specific exclusion criteria. The inclusion criteria consisted of the following points:

- title contains words: empirical/case/exploratory/human/quantitative/qualitative study
- title indicates an impact on comprehension or understanding
- indicating a difference between two or more perspectives

After the first selection phase, we had a remainder of 166 papers.

Abstract refinement

As mentioned before, the inclusion criteria for the second phase is the inclusion of a human study. Additionally, there was one exclusion criterion namely: it should not be a paper regarding a tool. Papers regarding their build tool are not regarded to be the relevant types of papers we are looking for, so these types of papers were excluded after this. This left us with 54 papers.

Inclusion criteria refinement

In the last phase, there were only exclusion criteria that were there to ensure the papers were within the scope.

- papers that involved human studies spanning an extended observation period
- papers that relied on eye tracking for comprehension analysis
- papers whose human studies served purposes other than measuring comprehension

In total, 16 papers were filtered out during this phase: 3 due to prolonged observation periods, 3 due to eye tracking involvement, 5 due to relevance issues, and 5 that initially seemed to contain a human study but did not meet our criteria. Consequently, we were left with 38 relevant papers that conducted a human study, forming the basis of our analysis.

5.2.2 Paper overview

This section highlights the findings of each of the papers that went through the selection procedure. We go through the papers in order from the latest ICPC procedure to the earliest procedure. For the study types, in case it was performed online (o) is added, and in case it was performed hybrid (h) is added. Additionally, in the participants column, the total amount of participants is stated including whether they were professionals(p) or students(s), or both(p&s).

year	author(s)	study type	participants	research area
2022	Wyrich et al. [64]	survey(o)	256 p&s	Test whether certain setups affect subjective assessment on code comprehension.
2021	Langhout & Aniche [38]	experiment(o)	132 s	Check whether atoms of confusion indeed result in less interpretable code.
2021	Cates et al. [13]	survey(o)	113 p&s	Does the structure of code (compound vs intermediate state) affect comprehension.
2021	Wiese et al. [63]	survey(o)	125 s	Examines intermediate students' understanding of code execution involving multiple boolean expressions
2020	Stapleton et al. [55]	survey(o)	45 p&s	Evaluates the quality of machine-generated summaries compared to human written summaries
2020	Dias et al. [19]	observation	16 p	Does a visual comprehension tool improve understanding of Javascript.
2020	Bai et al. [3]	experiment	18 s	How do graduates search when using an unfamiliar programming language
2020	Shargabi et al. [52]	experiment	178 s	Studies the effects of tasks on program comprehension mental model.
2019	Bauer et al. [4]	observation	22 p&s	Studies the impact of indentation on program comprehension within Java.
2018	Dos et al. [20]	survey(o)	62 p&s	Study the impact of coding standards on readability in Java.
2017	Avidan & Feitelson [2]	observation	9 p	Studies what the effects of variable names on code comprehension are.
2017	Ajami et al. [1]	experiment(o)	220 p	Which different kinds of code structures matter regarding code comprehension.
2017	Valdecantos et al. [59]	experiment	28 p&s	Studies whether Data Context Interaction improves code comprehension against OO, compares Trygve with Java.
2015	Roehm [48]	observation	21 p	Studies why developers put themselves in a user perspective during program comprehension.
2014	Jbara & Feitelson [31]	experiment	103 s	Studies whether code with higher regularity results in higher comprehension.
2012	Katzmarski & Koschke [33]	survey(o)	206 p	How does program complexity compare to the perceived complexity of programmers, and how methods and statistics can be adapted to program-understanding contexts.
2012	Feigenspan et al. [22]	experiment	128 s	How does experience compare to proficiency in programming.

2012	Nunez & Kiczales [43]	observation	50 s	How do registration-based abstractions affect the comprehension of students.
2012	Kleinschmager et al. [34]	experiment	33 p&s	Studies whether static type systems are better for maintainability than dynamic systems. Java(static) is compared with Groove (dynamic).
2011	Samaraweera et al. [50]	survey(o)	62 p&s	How does a reader interpret the intention of the main programmer. They look at meaning-preserving program refactorings.
2009	Cornelissen et al. [16]	experiment	24 p%s	How does visualization contribute to the comprehension process. Using their tool Extravis.
2009	Binkley et al. [6]	survey(o)	135p&s	Does camelCase improve understanding of code compared to under_scores.
2008	de Lucia et al. [18]	experiment	70 s	Compares ER(entity relation) and UML class diagram during comprehension tasks on data models.
2008	Fleming et al. [23]	observation	15 s	How do maintainers comprehend concurrent systems during system maintenance.
2005	Hogganvik & Stolen [27]	experiment	56 s	How well does CORAS support comprehension on risk analysis. Graphical icons measured against nongraphical icons (slight improvement in speed but not in correctness)
2005	O'Brien & Buckley [45]	observation	2 p	This paper reviews, merges, and adapts existing information-seeking models for different domains to propose a non-linear information-seeking model for programmers involved in software maintenance.
2004	Kuzniarz et al. [36]	survey	44 s	Studies whether using stereotypes improves understanding of UML models.
2003	Ko & Uttl [35]	observation(h)	75 s	Studies which comprehension strategies are used when learning an unknown domain. Stata was used as an example domain.
2002	Binkley [7]	experiment	63 s	Studies how semantic differences affect comprehension in the C language.
2001	Mosemann & Wiedenbeck [42]	experiment	76 s	Studies the effects of different navigation methods on novice programmers comprehension.
2001	O'Brien & Buckley [44]	experiment	8 p	Studies which comprehension processes are employed by participants.
2000	Corritore & Wiedenbeck [17]	observation	30 p	Studies the direction and scope of comprehension-related activities. OO programmers tend to use top-down approaches and procedural programmers a bottom-up approach
1999	Von Mayrhauser & Lang [62]	observation	25 p	Studies the impact of static analysis tools on comprehension during software maintenance. Compares the Lemma environment with the standard Unix environment.
1998	Burkhardt et al. [12]	observations	49 p&s	Studies the difference in comprehension of OO programs of experts and novices.

1998	Von Mayrhauser & Vans [62]	observations	2 p	Studies what comprehension process programmers use to perform their adaptive maintenance and which actions they perform.
1998	Sim et al. [54]	survey(o)	69 p	The study is aimed to characterize the source code searching behavior of programmers to construct a tool.
1997	Visaggio [61]	experiment	30 s	Studies how the quality of maintenance processes are affected by the ease of program comprehension. It compares the quick fix and iterative enhancement maintenance process.
1994	Tapp & Kazman [58]	experiment	39 s	Studies whether color and fonts help while performing programming tasks.

5.2.3 Paper Categorization

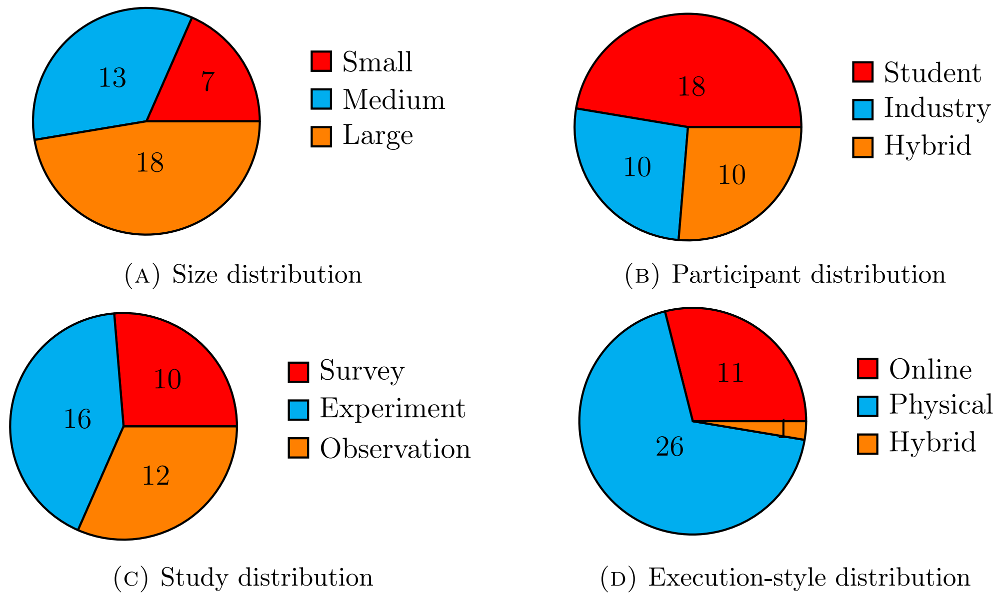
To gain useful information from the relevant 38 selected papers, it is important to establish factors we can use to analyze usefulness per study type. We have established a few aspects that are written down and summarized per paper. The information that we have extracted from the papers contains the following:

- The kind of human study that was performed.
- Whether the study was performed online or in a physical session.
- The number of human participants and whether they were professionals or students (or both)
- Amount of different participant groups.
- The essence and conclusion of the paper.

With just this information there is nothing to compare the papers with each other from. From the extracted information, we deduced four different categories that help put the performed human studies into perspective. Additionally, it has been decided not to put the purpose of the papers into a category as this required too much of a subjective analysis, but the goal of the papers is not completely disregarded in the process of identifying the most suitable study form to study the impact on code comprehension in a multi-paradigm environment. Each category is briefly explained including our view on the importance of the category.

Study size

The first characteristic to be considered in designing a meaningful human study for code comprehension is the size of the participant pool. This indicates what acceptable sizes are for program comprehension studies. The sizes of participant pools ranged from, only a couple of participants that were closely monitored by the researchers to wide-scale surveys that amassed more than 250 respondents(are 2 references necessary?). To distribute the papers into different categories, we established three size categories: small studies (1-20), medium studies(21-50), and large studies(50+). The distribution of this can be found in Figure 5.1a.



(A) Size distribution

(B) Participant distribution

(C) Study distribution

(D) Execution-style distribution

FIGURE 5.1: Characteristics distributions

Participant demography

Another interesting characteristic to consider is the background of the participants. The main distinction most researchers make is between industry professionals and students. While some studies are focused on students, may it be due to availability or interest multiple studies try to combine the backgrounds. By considering both it creates a heterogeneous pool of participants. The distribution of this can be found in Figure 5.1b.

Study type

While the overarching goal of various studies centers around examining different aspects of program comprehension, there exists a diversity of requirements that drive distinct execution methodologies. These variances in necessities give rise to various modes of study execution. Classifying the different forms of human studies is, therefore, an additional dimension to be considered. As we delve into the relevant papers, it becomes evident that the spectrum of human studies can be boiled down to three primary categories.

The initial category is the survey/questionnaire study, which, as the name implies, gathers data through surveys or questionnaires. This kind of study doesn't require direct interaction between researchers and participants.

The second category is the experiment study. This type of study requires some degree of interaction between participants and researchers, ranging from interviews involving in-depth queries to real-time tasks administered in the participant's presence. Studies categorized as experiments can incorporate surveys, but the presence of human interaction classifies them as an experiment.

Lastly, the observation study concentrates on observing participants as they undertake a predetermined set of tasks. These observations are typically coupled with a "thinking-aloud" approach to document sessions. While human interaction exists, the emphasis is not on active engagement but rather on passive observation of participants' behaviors.

The categorization of the papers can be found in Figure 5.1c.

Execution style

The last category that is being considered is the way the study is performed. This entails whether it was performed in an online environment or a physical setting. It is relevant to see what kinds of studies can be performed online and which should be performed in a physical setting. Beforehand it should be noted, that some papers have been published during corona, during these times it is expected that studies prefer an online approach. The division can be found in Figure 5.1d.

5.3 Findings

When trying to investigate the impact of multi-paradigm usage on code comprehension, various study methods can be employed, namely surveys, and experiments. Surveys can be conducted online, while other methods are typically carried out in a physical setting whereas an online application is rare. Surveys offer the advantage of gathering both quantitative and qualitative data, although sometimes yield results that may not align with initial expectations. Interviews, as an alternative to surveys, share similar objectives but allow for more guided and insightful responses.

Survey-based studies tend to lean towards emphasizing quantitative data collection, mainly due to the inherent nature of surveys allowing relatively unguided data gathering. This focus on quantitative aspects is partly a result of the survey's structure, which can lead to results that are more easily quantifiable.

For methodological feasibility and simplicity, it is recommended not to introduce unnecessary complexities, such as involving multiple programming languages or numerous variables. Instead, a practical approach involves carefully selecting a limited number of variables or factors. This focused strategy aims to minimize outside influences, preventing potential confusion caused by unrelated factors and ensuring the clarity of the study's outcomes.

5.4 Study design

Several parts complete the design of the human study. Each of these is described in the subsequent sections. All factors and possibilities are laid out and explained in depth. The different factors and characteristics of the study setup are the type of study, the participants, and the study language.

5.4.1 Study Type

As outlined earlier, two distinct study types align particularly well with the requisites of our study: interviews and surveys. Each of these methods carries its own set of advantages and limitations, some of which have been previously highlighted and are reiterated here for comprehensive coverage.

Interviews

Interviews offer a robust means of guiding conversations and creating opportunities for seeking clarifications where necessary. This inherent flexibility aligns adeptly with our objective of capturing qualitative data of a desired standard. Interviews allow participants to elaborate more extensively on specific findings, a dimension that might be constrained within the boundaries of a survey, which typically demands brevity.

Nonetheless, these advantages of the interview methodology are not without trade-offs. The requirement for individualized interview sessions, as opposed to group formats, renders this approach time-intensive. A statistically significant study, necessitating more than a mere handful of interviewees, demands a substantial investment of time in conducting, transcribing, and subsequently analyzing each interview. Furthermore, the gathered interview data requires careful processing, in adherence to GDPR.

Survey

Contrarily, surveys chart an entirely distinct trajectory. Emphasizing the number of participants over in-depth engagement, surveys aim to glean insights from a broad spectrum of respondents while minimizing their time commitment. This approach yields many results underscored by versatility. Although surveys excel in collecting quantitative data, they possess the capacity to capture qualitative insights through succinct questions. Nonetheless, soliciting qualitative data within a survey is less straightforward than collecting quantitative information. The degree of participant guidance is constrained, even when questions are deliberately framed to encourage qualitative responses. The resultant behaviors might not conform to expectations, contributing to the unpredictability of outcomes.

Survey implementation demands a reasonable time investment, primarily in the survey design and validation. Once this foundational step is accomplished, the only thing that remains is finding participants and motivating them to partake in the study. It is essential to underscore the care required in survey setup, encompassing a thorough examination of questions and choices, weighing their respective advantages and disadvantages. Once finalized, it no longer is feasible to alter the survey as this renders previous results useless, hence the careful and thorough approach in the survey design.

5.4.2 Language

Besides the study type, the selection of the programming language significantly influences the study's trajectory. The language choice is a multifaceted decision containing two primary considerations: the number of programming languages under consideration and the specific language(s) to be employed.

As previously indicated, the introduction of multiple programming languages introduces an increase in confounding factors. These variables demand careful management to preclude the potential invalidation of results. Simultaneously, embracing multiple languages affords the potential for enriched analyses. However, for the sake of maintaining study feasibility, the pursuit of using multiple languages, as elaborated in the Findings section, will not be pursued further.

Equally vital is the selection of the programming language, besides the employed study type. Considerations extend to factors such as the level of expertise required for effective engagement with the chosen language. The languages considered are Java, Scala, C#, and Kotlin. Notably, Scala and Kotlin emerge as languages expressly designed with multi-paradigms in mind, accommodating both programming paradigms of functional programming and object-oriented programming. Conversely, Java and C# encompass functional programming constructs, albeit being specially used for augmenting object-oriented code, particularly pronounced in Java.

Scala, while historically significant, has witnessed a decline in its utilization [46]. This could potentially be attributed to its steep learning curve. It is pertinent to underscore that within the organizational framework of Info Support, the research's host institution, and the University of Twente, Scala's popularity remains minimal.

Java and C#, on the other hand, are in development practices within Info Support, presenting a substantial participant pool. Moreover, Java holds prominence as the educational programming language at the University of Twente, rendering it an avenue for student recruitment.

Finally, the inclusion of Kotlin deserves discussion. Despite its limited usage at the University of Twente and Info Support, its alignment with Java's syntax and concepts renders it accessible. This facilitates the comprehension of Kotlin by those acquainted with Java. Kotlin may serve as an exploratory tool for measuring participants' cognitive processes in a semi-unfamiliar environment. Such exploration enables the observation of participants' reasoning and thought processes in an environment that remains unprejudiced by established programming norms. This dimension is particularly relevant for assessing the intersection of functional and object-oriented programming concepts without the possible bias introduced by active development in a specific programming style.

5.4.3 Decision

In light of a thorough evaluation of the advantages and drawbacks associated with both interview and survey methodologies, coupled with the requisites of the study, the chosen study type is interviews. Despite the time-intensive nature of interviews, we believe they present the optimal avenue for capturing relevant insights into code comprehension within this context. Gathering qualitative data relevant to the tracing of comprehension strategies and the cognitive reasoning processes of participants. Although surveys might potentially fulfill this requirement, they carry substantial risks of yielding unsatisfactory or insufficient responses, hindering meaningful analyses.

Given the pivotal significance attributed to capturing participants' cognitive behavior and considering it a foundational aspect of the research, interviews emerge as the more fitting choice. This alignment aligns with the study's objectives and interests.

it is noteworthy that participants may still be requested to complete a brief survey before their interview, facilitating the capture of relevant demographic information.

Turning to the characteristic of the language within the study, Scala, owing to its limited practical usage, emerges as less feasible for participant recruitment. The decision, therefore, hinges on the selection between Java, C#, and Kotlin. Rather than opting for the language most extensively employed by participants in their active development, the decision favors Kotlin. The language's close alignment with Java, while preserving a distinct identity, renders it an ideal candidate for this study. This choice enables the genuine capture of participants' cognitive processes in code comprehension, as reliance on prevalent programming practices becomes unavailable. While familiarity with Kotlin is expected among participants, proficiency in writing active code in it is not mandatory. A basic understanding of Java serves as a sufficient foundation, with the provision for a brief introduction to Kotlin's syntax and features, if necessary.

Crucially, the interview questions are formulated in such a way they can be answered without requiring code alteration. This allows for an equal task for each participant and excludes individual coding capabilities. The primary focus of these questions is rooted in comprehending code and unearthing its underlying purpose.

Regarding participant demographics, a balance of students and professional developers is chosen. This mix not only ensures diversity but also affords a comprehensive exploration of code comprehension in a multi-paradigm landscape. It also offers the opportunity to examine whether programming experience carries relevance as a contributing factor. Although participants will predominantly be drawn from Info Support and the University of Twente due to their availability, recruitment is not restricted to these entities.

Chapter 6

Study setup

This chapter will describe the setup of the study, including the structure of the interviews and the reasons behind the setup. In Chapter 5 it is displayed that the form of the human study will be that of an interview.

6.1 Participants

In the section concerning the human study, it is elucidated that participants will be recruited from Info Support and the University of Twente, but the recruitment is not limited to these two institutions. Nevertheless, there exist several other aspects concerning the study's participants. Within this context, a distinction is established, differentiating between the number of participants and the recruitment process. Both sections describe the decision-making when establishing the criteria.

6.1.1 Sample size

Following our study design, we have established a specific target number of participants, which stands at 30. This number aligns with our research objectives and the balance between quantitative data collection and the in-depth exploration of comprehension strategies, more on this in section 6.3.

6.1.2 Recruiting process

The recruitment of participants is an important aspect of the execution of this study. A clear differentiation is drawn between the processes involved in recruiting professionals and students. For three weeks, interviews were conducted, and their scheduling was facilitated through the utilization of Datumprikker. Two separate Datumprikker schedules were used for scheduling interviews: one for on-location sessions at Info Support in Veenendaal and another for sessions at the University of Twente. This approach streamlined the scheduling of multiple interviews, minimizing the need for extensive planning and communication to establish suitable meeting times.

Professionals

Within our framework, professional participants are defined as individuals engaged in full-time computer science positions, preferably software development positions. Those who are working part-time as software developers and studying do not fall within this category. The pool of professional participants was predominantly drawn from Info Support, the company

closely associated with this research. Various communication channels within the platform Slack within Info Support were used to solicit potential participants. Recognizing the requisite knowledge of Java or Kotlin for participation, the message was thoughtfully shared within the Java community. Additionally, offline recruitment strategies were employed, involving in-person discussions with individuals, and gauging their interest in participating. Upon expressing interest, further information was subsequently given to them.

Eligibility for professional participants requires them to fulfill positions as software developers and have experience with Java or Kotlin. While the majority of these professionals are from Info Support, several professionals outside of Info Support were also approached and accepted into the study.

Students

The criteria for selecting computer science students are more strict in comparison to professionals. Unlike professionals, there's a greater uncertainty regarding the foundational programming skills of students who are just starting their academic careers. To mitigate this uncertainty, the selection process exclusively includes students who have progressed to at least the third year of their bachelor's program or are pursuing a master's degree.

For these students, a direct outreach was employed to gauge their potential interest in participating. Furthermore, for larger groups, comprehensive messages that encapsulated all the pertinent research information were circulated. This systematic approach has proven to be notably effective, resulting in a high response rate from the students.

6.2 Interview structure

The interview consists of three different parts. The first part is general, secondly, we have a set of tasks the participant is asked to perform, and lastly, we have a post-interview session. The interview will be a so-called semi-structured interview. A semi-structured interview is a research method that enables the capturing of qualitative data, using a pre-determined set of questions with the possibility to divert from the questions according to the reaction of the participants. This approach ensures that the required data to be captured will indeed be captured.

6.2.1 Equipment setup

To effectively capture the interviews and facilitate comprehensive post-interview analysis, it is required to employ audio recording. Notably, the emphasis of the qualitative data collection within these interviews resides in capturing the cognitive processes of the participants. To achieve this, it becomes necessary to record not only what participants say but also their on-screen actions, including cursor movements. This allows the tracking of the comprehension steps taken by the participant to pinpoint precise identification of the lines that are discussed.

To minimize potential external influences that could interfere with participants' engagement in the comprehension tasks during the interviews, a standardized approach is adopted. All interviews are conducted using a singular laptop configuration, specifically, the Dell Latitude 5531. This laptop is equipped with a 12th Gen Intel(R) Core(TM) i7-12800H processor and features an NVIDIA GeForce MX 550 18 GB GPU. This uniformity in hardware ensures consistency across the data collection process.

6.2.2 Interview start

At the beginning of the interview, participants are greeted with an introduction regarding the interview topic, and code comprehension in Kotlin. In advance of the interview, participants have been provided with a detailed information letter, available in Appendix A, containing all details. As the interview begins, participants are briefed on what they can expect, ensuring alignment with the contents of the aforementioned information letter.

Importantly, participants are not provided with the entirety of the study's information; they are solely informed that it relates to code comprehension in Kotlin. More specifically, they are kept unaware of the broader subject of multi-paradigm programming. This strategic choice is made to mitigate the risk of introducing a learning curve and to ensure that participants do not carry expectations into the interview, thereby safeguarding the impartiality of the results.

Participants are informed about the explanation behind conducting the interviews. It is explicitly explained that these interview settings are designed to capture the comprehension strategies employed by them. Although questions posed during the interview may be answered correctly, incorrectly, or partially correctly, it is emphasized that the primary focus of the interview is not centered on the correctness of these responses. Instead, the main objective lies in capturing the comprehension process, as it is the main contributor to the results.

Following this briefing, participants are kindly requested to provide their consent by signing a form that addresses the recording of the interview and data processing, the consent form can be found in Appendix B. Lastly, participants will be presented with a series of general demographic questions capturing the following things:

- Are they working professionally or still studying?

  If working professionally: How many years of work experience?

  If studying: What year of study Bachelor/Master?

- Years of experience in Java.

- Years of experience in Kotlin.

6.2.3 Interview tasks

The majority of the time will be spent on the interview tasks. The participant will be given a certain comprehension task to perform. Such a task does not require the participant to write any code. The task only requires one to read, interpret, reason, and comprehend the piece of code. During this time the participant is allowed to do anything, besides running the code. This entire process is accompanied by the use of the think-aloud protocol. The task will continue as long as there is no definitive answer. Once a final answer has been presented by the participant, we continue to the next task on one condition. It has to be clear what the thought process has been, in case this is not clear, the participant is asked to elaborate on how they came to this conclusion. Based on what information that is missing the following questions can be asked:

- How did you come to this conclusion?
  This is a general question that will be asked when the think-aloud process proves too limited. Asking this question and any additional follow-up questions will be asked until a satisfying and complete answer is given by the participant.

- Why do you think that is the answer?
  Another general question, that helps to extract the reasoning of why the answer has been given. This helps with connecting the thought process during the task.

- In case the participant fails to finalize an answer, the following question will be asked: Where do you get stuck, that results in you not being able to give a definitive answer?
  Asking this question allows, for a better understanding of why a participant was unable to complete the task. In case it was not clear to the participant, it will be commented that failing to complete a task has no consequences and we will carry on with the remaining tasks.

Task types

The tasks presented to the participants will range in form, but all are comprehension questions. In this subsection, the different types of task types are elaborated, by stating the purpose of the task and what we try to gain out of it.

- What is the functionality of the code? The goal of this task is for the participant to scan the code and figure out what the functionality is of the code. It requires the participant to look for points of recognition and understand what happens in the majority of the code. Such a comprehension task should reflect properly what kind of comprehension technique, as described in Chapter 4, is being employed. The task does not require the participant to read all code if it becomes apparent what the functionality of the code is.

- If the input of this 'X' then what is the output? This task, although similar to the first kind of task, does not require a full picture of the overall code, but the participant is forced to follow the flow of the program. This allows us to see, whether the flow of the program follows logically from the code written.

- The output is 'X', but it should be 'Y', where is the fault in the code located and explain why? This task combines the comprehension of the code and spitting through the flow of the program. First, the functionality of the code needs the be understood, followed by a debugging task where the bug in the code needs to be located. This task contains multiple comprehension stages and allows us to analyze the employed comprehension strategies for each stage.

Satisfying answer

As previously described each task continues to the next task when no answer is found and this is indicated, or when a satisfying answer is given by the participant. So what is our definition of a satisfying answer? First of all, a satisfying answer does not mean that the answer has to be correct. It could very well be that an incorrect answer has been given to said tasks but remains a satisfying answer. A satisfying answer is the kind of answer that allows us to have a complete overview of the thought process of the participant. This includes the answer to the task combined with the reasoning that is based on the think-aloud method or possibly complemented with answers to additional questions asked. This complete answer allows for a complete qualitative analysis of the thinking process, linking them to different comprehension strategies.

6.2.4 Post interview

After the completion of all the tasks and gathering all required information for this, we will continue to the post-interview. This part of the interview consists of two different parts: additional demographic questions and paradigm-related questions regarding the tasks.

Demographic questions

The additional questions asked at the end of the interview are all paradigm-related. The reason for not disclosing these questions at the start of the interview is to reduce any possible learning. By asking the questions afterward, there is no connection between multi-paradigm usage and code comprehension. Until this point, they are only aware of the code comprehension aspect of the study. This set of questions will ask questions regarding the experience with functional-, object-oriented, and multi-paradigm programming.

Paradigm related questions

After the additional demographic questions, the last part of the interview will take place. Additional information regarding the research will be explained as well as the objective of the study, which is to study the impact of multi-paradigm usage on code comprehension. Afterward, some questions regarding functional programming and multi-paradigm programming will be asked. By asking these questions a bigger picture of the participant's understanding of the topics is captured. This information can later be used in looking for reasons for certain comprehension behaviors. If there were any interesting observations made while performing the tasks, there is room here to ask in-depth questions. This is asked afterward as these questions require the participant to know the full objective of the study.

6.3 Methodology

This section will describe the methodology used for the analysis of the qualitative data. This section describes which qualitative analysis methodology will be used. As stated in the introduction of the thesis, there is still little research on the impacts of multi-paradigm programming on code comprehension. That is also why this exploratory study is performed to gain insights into what the consequences are of this type of programming. The study is performed out of curiosity and does not have a strong hypothesis when going into it. This leads us to a qualitative analysis methodology of grounded theorem. Grounded theorem uses inductive reasoning to derive theories It was first introduced by Strauss and Glaser [25].
This method is chosen for the qualitative analysis as the qualitative part of our interviews is regarding the cognitive processes and comprehension strategies employed by the participants. Since we are purely interested in what possible consequences of the usage of multi-paradigm programming, we hope to form multiple theories regarding this. The type of answers we are looking for neatly connects with the Grounded Theory methodology.

Sample size

In diving into a qualitative approach to the principles of Grounded Theory, figuring out the right sample size becomes a bit of a puzzle. Within several research regarding the sample sizes of qualitative methodology, it becomes clear that a sample size of 25-30 participants is deemed necessary to reach saturation [41]. Since we want to use quantitative data to

support our findings in the analysis of Grounded Theory it was decided to go for a sample size of 30 participants.

Analysis steps

The methodology of Grounded Theory consists of 3 parts.

1. Code text and theorizing: Code the text of the interview. Useful concepts are identified and named. This is called open coding. In our case each question is summarized explaining the thought process of the participant, and interesting quotes are written down as well. Since the research is aimed to discover the cognitive reasoning repetitive non descriptive things are omitted from this. We argue that this does not impact the text coding.

2. Memoing and theorizing: Memoing is the process by which a researcher writes running notes bearing on each identified concept. The running notes constitute an intermediate step between coding and the first draft of the completed analysis. Memos are field notes about the concepts and insights that emerge from the observations. Memoing starts with the first concept identified and continues right through the processing of all the concepts. Memoing contributes to theory building.

3. Integrating, refining, and writing up theories: Once coding categories emerge, the next step is to link them in a theoretical model constructed around a central category that holds the concepts together. The constant comparative method comes into play, along with negative case analysis. Negative case analysis refers to the researcher looking for cases that are inconsistent with the theoretical model.

Chapter 7

Interview questions & constructs

To execute the interviews as described in Chapter 6, the initial step involves the construction of multi-paradigm constructs. Once these constructs are in place, the subsequent phase entails crafting the comprehension questions and tasks that will be utilized during the interview.

7.1 Multi-paradigm constructs

While the research initially considered four distinct programming languages, it ultimately led to the decision to proceed with Kotlin, more on this in Chapter 5. This section will delve into the multi-paradigm constructs that we have defined. It is worth noting that, to the best of our knowledge, there exist no formally established multi-paradigm constructs that combine the principles of object-oriented programming and functional programming. Thus, we defined constructs that encapsulate both the declarative aspects of functional programming and the imperative aspects of object-oriented programming.

7.1.1 Considered functional programming constructs

In the preceding sections of this thesis, various functional programming constructs were outlined and explained. However, it is relevant to emphasize that not all of these constructs were employed in our research. Some functional constructs, by their nature, do not seamlessly align when combined with object-oriented programming.

Consequently, we have identified the following functional constructs that have been integrated into our research in defining multi-paradigm constructs in Table 7.1.

It is worth noting that our selection of functional programming constructs is not complete, as specific considerations were made regarding the relevance of certain constructs. Recursion, for instance, was not included, as it is now well-integrated into both functional

Included constructs	**Excluded constructs**
Higher-order functions	Currying
First-class functions	Partial application
Anonymous functions	Lazy evaluation
Pattern matching	Recursion
Referential transparency	-
Functional purity	-

Table 7.1: Included and excluded functional constructs

and object-oriented paradigms, making it less distinctive in the context of multi-paradigm constructs. This does not mean it is used in defining multi-paradigm constructs, it's not seen as a functional programming construct in the context of this thesis.

Additionally, constructs such as Currying, Partial Application, and Lazy Evaluation, although supported in the programming languages under discussion, do not find a seamless fit within an object-oriented context. Given our primary objective of identifying constructs that harmonize with the object-oriented environment, these constructs were intentionally omitted from the definitions of the multi-paradigm constructs.

From here on several defined multi-paradigm constructs are discussed per section and are given examples of how they can be used.

7.1.2 Impure lambda functions

The multi-paradigm construct Impure lambda functions possesses a notably global character, as it merges the utilization of lambdas from functional programming while abandoning the purity status associated with functional programming. This fusion is achieved by integrating the inherently impure nature of object-oriented programming. It is essential to acknowledge that, in the context of the other defined constructs, there is potential for the presence of impure lambda functions as well. However, it is not a strict rule that every lambda containing object-oriented programming elements must be impure. An instance of an impure lambda function is exemplified in Listing 7.1.

Listing 7.1: Impure lambda

```kotlin
    var counter = 0
    val impureLambda = { value: Int ->
        counter += value
        println("Counter updated to: $counter")
    }
```

In this listing, you can see that the variable counter gets incremented within the lambda, rending the lambda to become impure. Additionally, it also prints the new value of the counter, making the lambda once again impure. This example is purely there to demonstrate how easily within Kotlin a lambda function can be made that is no longer pure.

7.1.3 Imperative lambda functions

In the domain of functional programming, functions can be chained this is commonly done through the use of function applications. However, these functions are primarily oriented towards a purely declarative description of what needs to be accomplished. When harmonizing this with object-oriented programming, a synthesis emerges, fusing the declarative attributes of functional programming with the imperative characteristics of object-oriented programming.

We distinguish two different kinds of imperative usages namely a linear imperative path and a non-linear path. This distinction hinges on whether the lambda functions contain branching elements, such as loops or conditional branching (e.g., if-else statements), or if it does not. Both non-branching and branching functions fall under the category of imperative lambda functions.

Within Kotlin, a wealth of support exists for well-established higher-order functions, including functions like map, fold, and filter. In pure functional programming, the functions

to be implemented are often concise and straightforward. In contrast, within the realm of object-oriented programming, this constraint is more flexible, allowing for the creation of larger imperative implementations. These larger implementations seamlessly interweave with the declarative nature of lambda functions within a function application context. In Listing 7.2 an example implementation of an imperative lambda function can be seen.

Listing 7.2: imperative lambda function

```
fun f(x: List<Int>): Int {
    var y = 0
    x.forEach { z ->
        if (z + y < 10)
            y += z
        else
            y -= z
    }
    return y
}
```

7.1.4 Encapsulated higher-order functions

While higher-order functions are foundational in the realm of functional programming, encapsulation holds a similarly vital role in object-oriented programming. In addition to employing pre-implemented higher-order functions, an alternative approach involves writing custom higher-order functions. The encapsulation of these higher-order functions involves integrating their functionality directly into the class structure. This architectural choice permits the creation of more abstract object functions, enhancing the generality of the classes. This, in turn, facilitates the reuse of class implementations. A practical illustration of this concept can be observed in Listing 7.3.

Listing 7.3: encapsulated higher-order function

```
class TextProcessor {
    private var textTransformation: (String) -> String = { it }

    fun setTransformation(transformation: (String) -> String) {
        textTransformation = transformation
    }

    fun processText(input: String): String {
        return textTransformation(input)
    }
}
```

7.1.5 Branched pattern matching

Pattern matching, or case distinction, is a core part of the descriptive nature of functional programming. Due to its useful nature, object-oriented languages have also adopted the use of it. But instead of using this as clear case distinctions, it can be used to branch in a function based on certain inputs. It allows for checking properties, not limited to the base cases of a function. A simple program adding all elements in a list and returning the sum

of it can be found in Listing 7.4.

LISTING 7.4: Branched pattern matching

```
fun sumList(list: List<Int>): Int {
    return when {
        list.isEmpty() -> 0
        else -> {
            val head = list[0]
            list.first() + sumList(list.drop(1))
        }
    }
}
```

While this construct does not need to have any recursion, it does increase the complexity of such a function. Instead of using traditional looping functionalities, pattern matching can also be used. So this construct can replace an imperative loop.

7.2 Interview Questions

Chapter 6 describes what the procedures of the interview are, but does not discuss what the actual interview questions are. This section describes each question, what the goals of the questions are, and how they relate to multi-paradigm constructs. As mentioned before, for each of the questions there are two versions, one object-oriented version and one multi-paradigm version that incorporates functional programming constructs that are defined in the previous section. Questions 1-4 are smaller than the last three questions and participants were given roughly 5 minutes to come to an answer. In the case of question 3, since it contains two sub-questions participants were allowed slightly more time if one of the sub-questions took more time. Part of the questions will be displayed in the following subsections, but the complete code questions can be found in Appendix C. Otherwise, you can find the question here online as well.

7.2.1 Question 1

In the first question, the participants are tasked with calculating the output of a function that adds items from a list balancing under the value 10. It was decided that the first question should not ask too much of the participants and allow them to get familiar with Kotlin and what is expected from them in this study. The object-oriented variant uses a *for-loop* to loop through all elements in a given list whereas the multi-paradigm variant uses the *forEach* function. Thus, the differences are that the OO version uses a *for-loop* and the multi-paradigm variant uses a *forEach*. The only difference is that the *for-loop* contains a body whereas a *forEach* call is given a Lambda function that takes as input an item from the list. In Listing 7.5 you can see the differences between OO and MP.

Listing 7.5: Body logic Question 1

```
fun g(x: List<Int>): Int {      |   fun f(x: List<Int>): Int {
    var y = 0                   |       var y = 0
    for (z in x) {              |       x.forEach { z ->
        if (z + y < 10)         |           if (z + y < 10)
            y += z              |               y += z
        else                    |           else
            y -= z              |               y -= z
    }                           |       }
    return y                    |   return y
}                               |   }
```

The body is identical for both versions. The only difference is that the MP variant is not in a code block but in a Lambda, more specifically an Imperative Lambda function which is also impure.

7.2.2 Question 2

The second question requires the participants to identify the functionality of the code, which is an implementation of merge sort[1]. Both versions contain the structure of merge sort, meaning it splits up the list into two parts and recursively calls itself again to merge the outputs of those recursive function calls. Where the OO version uses a straightforward implementation with while loops in the merge function, the MP variant does this differently, it is built using the Branched pattern matching construct, making distinctions between the different cases that need to be checked. The specified function can be found in Listing 7.6.

Listing 7.6: Multi-Paradigm merge function

```
fun funcB(listA: List<Int>, listB: List<Int>): List<Int> {
    return when {
        listA.isEmpty() -> listB
        listB.isEmpty() -> listA
        listA.first() < listB.first() -> {
            listOf(listA.first()) + funcB(listA.drop(1), listB)
        }
        else -> {
            listOf(listB.first()) + funcB(listA, listB.drop(1))
        }
    }
}
```

7.2.3 Question 3

Question 3 consists of 2 parts in which the participants are tasked with calculating the output of 2 function calls (of 2 functions) and determining the functionality of these functions. Function A checks whether a given number is prime, and Function B calculates the greatest common divisor according to the Euclidean algorithm[2].

[1] https://en.wikipedia.org/wiki/Merge_sort
[2] https://en.wikipedia.org/wiki/Euclidean_algorithm

Question 3a

In the calculation of whether a number is prime, the OO version uses a for-loop to check for possible divisors, whether the MP version first creates a range of numbers and then applies the higher-order function *none* with the divisor predicate given. The OO version uses a more imperative approach whereas the MP version uses a more declarative approach. By doing so this gives us an insight into whether in small functions a separation of OO and FP has any impact.

Question 3b

For Question 3b the Euclidean algorithm is implemented. While both do the same, the OO version uses explicit states and assigns explicit values to variables, whereas the MP version does this implicitly by recursively calling the function while swapping the arguments and applying the modulo operation. Similar to 3a there is still a distinction between OO and FP which allows us to check whether implicit assignments differ from explicit assignments.

7.2.4 Question 4

Question 4 is the first question that includes the implementation of a Class, the questions up to this point only used functions. In this question, the participants are tasked with calculating what the output is of the function call. Additionally, they are asked if they know what the class represents. The class in this case contains a list of doubles and represents a polynomial where for each item in the list its index corresponds to the exponent in the polynomial. The class contains a method, which is the evaluate function with a certain value x. The style is similar to that in question 1, as the differences in versions only differ in the logic, which in this case is the calculation. The OO version uses a for loop to go through each element in the list, whereas the MP version first performs a map operation on the list and then sums up the list.

7.2.5 Question 5

For Question 5 some context is already given. The participants are already provided with two data classes namely Edge and Node and know they need to identify a function within a class Graph. So they are aware that they are working in a graph context. The function that they need to identify is an implementation of Dijkstra's shortest path algorithm[3] that returns a pair that contains the cost of the path and the path itself. Within the class Graph, there is a variable called *AdjacencyList*. This variable contains a Map for each Node in the graph as the key that maps to all edges where the Node is the source of. The participants are briefed beforehand on what this variable is and what it means. This creates additional context that the participants are allowed to use.
The OO and MP versions differ a little bit in structure. The OO version only has one function within the class which calculates the shortest path, whereas the MP version contains two functions. The first function still calculates the shortest path, but to reduce the complexity of the function, the code responsible for updating the intermediate paths + costs has been extracted to a function on its own. Participants are made aware that this function exists and is called within the main function. The extracted function can be seen in Listing 7.7.

[3] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Listing 7.7: Extracted function question 5 (MP)

```
private fun funcB(
    listEdge: List<Edge>,
    nodeA: Node,
    mapA: Map<Node, Pair<Int, List<Node>>>,
    a: Int,
    listNode: List<Node>
): Map<Node, Pair<Int, List<Node>>> {
    return listEdge.fold(mapA.toMutableMap()) { mapB, edge ->
        val varA = a + edge.weight
        val (varB, _) = mapB[edge.destination]
            ?: Pair(Int.MAX_VALUE, listOf(nodeA))
        if (varA < varB) {
            mapB[edge.destination] = varA
                to (listNode + edge.destination)
        }
        mapB
    }
}
```

The construct that you can see in this listing is also used within the other function. A fold is used to iterate over a list. After each iteration the initial value, which in the case of Listing 7.7 is "mapA.toMutableMap()" keeps being updated. With the previously defined multi-paradigm constructs it is clear that this is no longer a pure and declarative lambda function. The map keeps getting changed, rendering it no longer pure, and additionally containing additional if statements creating an imperative lambda. This same construction is also used in the first function, whereas the OO version does this by using an outer while-loop and an inner for-loop for updating the paths.
The idea of this question and the reasoning behind creating such big imperative lambda functions is to check what the impacts are. In question 1 the size and complexity remained limited, whereas that is the opposite within this question.

7.2.6 Question 6

Question 6 is slightly different in terms of tasks that need to be performed. They are given a class that has a matrix as a class property. The participants are tasked with looking for a bug in the program. They are given the current output of the main function where it is also mentioned that there is a bug in the system that influences the behaviour of the program. To give them some directions they are first tasked with figuring out what exactly is missing, once they do this they can deduce where the fault in the system takes place.
The class contains 2 functions that transform a matrix, the first one transposes the matrix and the other one filters numbers out of the matrix based on a certain filter condition, in the case of this question that is filtering out numbers that are not even.
The OO version has a filter function built within the class where it is already specified. The MP version uses several Higher-Order functions that provide a general implementation where the argument functions specify what precisely is done. This creates a dependency on lambdas that are specified here and there. The issue within the program is that by filtering the values out of the matrix, they are completely deleted instead of being replaced creating an unbalanced matrix, where not each row has the same amount of columns. This is then not taken into account in the transpose function stripping of a value at the end.

Debugging code requires multiple cognitive levels [65]. Bloom describes 6 levels of cognitive learning: knowledge, comprehension, application, analysis, synthesis, and evaluation [8]. These 6 levels were then used to classify the cognitive activities used during debugging [65]. While the debugging processes here also describe fixing the issue, for this question the participants are only required to spot what goes wrong. This requires the participants to first gain an understanding of the program and after this reason about its behavior. While it remains a comprehension task, it does ask slightly different things from the participant allowing for a more complete view into the different comprehension strategies employed.

7.2.7 Question 7

The last question of the interview gives the participants the task of figuring out what the functionality is of the class and what the result would be within the main. The class implements the KPM algorithm[4] that calculates the occurrences of a sub-string in a string. Similarly to exercise 2 the two versions differentiate between using loops for the OO version and using branched pattern matching for the MP version. Additionally, the main that is given to the participants might indicate what the use is of the program. In Listing 7.8 you can find the main function.

Listing 7.8: Main function of question 7

```
fun main() {
    val varB = D().funcA("ABABDABACDABABCABAB", "ABA")
    if (varB.isNotEmpty()) {
        println(varB)
    } else {
        println("Empty")
    }
}
```

It has deliberately been chosen to use expressive arguments, allowing the participants to reason about the program. To not make things too obvious the return type of the function is not an Integer counting the amount of occurrences but a List keeping track of all starting indices. Additionally, the algorithm uses a smart thing to transform the sub-string into a small state machine allowing for efficient searching. To make this function useful, it was decided to give a string that produces more and in this case results in [0,0,1].
The MP version contains additional functions, that are required to set up the branched pattern matching correctly as these functions are invoked recursively. Apart from that the versions remain the same.

7.2.8 Question construct usages

For all questions and their respective versions, the differences and similarities have been discussed. Now we provide an overview of which multi-paradigm or functional programming constructs were used for the multi-paradigm versions.

[4] https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm

Question	Used MP/FP constructs
Question 1	Impure lambda, Imperative lambda
Question 2	Branched pattern matching
Question 3a	Higher-Order function
Question 3b	Functional purity, Referential transparency
Question 4	Method chaining, Higher-Order function
Question 5	Impure lambda, Imperative Lambda
Question 6	Encapsulated Higher-Order function
Question 7	Branched pattern matching

Table 7.2: Used MP and FP constructs per question

Chapter 8

Experiment results

This chapter discusses the results of the executed experiment described in Chapter 6. The Setup also describes that the interviews performed contain the capture of quantitative and qualitative data. Therefore the results will be separately discussed.

8.1 Data collection

Throughout the interviews, a comprehensive set of data has been gathered. It can be categorized into four distinct parts: demographic data, quantitative data, qualitative data, and data in need of quantification. The quantitative data gathered is as follows:

- Profession: Either professional, Bachelor student, or Master student
- Years of experience: In the case of a student this amounts to the current year they are in, for professionals this is the number of years active as a professional
- Experience in Java in terms of years (rounded to half years)
- Experience in Kotlin in terms of years(rounded to half years)
- Functional programming rating on a scale of 1-5, as described in Chapter 6, will be asked at the end of the interview.

The quantitative data collected per question per participant consists of the following:

- Version of the question: OO or MP.
- Time spent on the question.
- The answer of the participant in the form of correct or incorrect.

The qualitative data encompasses the entirety of the interview. Participants express their thoughts, not limited to their thought process of moving through the code. This includes comments on constructs and reflections on non-descriptive variable names. While the original comprehension process and strategy are inherently qualitative, they can be interpreted and quantified. In Chapter 4, five distinct comprehension strategies are outlined: Bottom-up, Top-Down, Systematic, Knowledge-Based, and As-needed. The latter is considered an unfit strategy due to the general descriptive nature and the small nature of the code examples, raising the question of whether the as-needed strategy is similar to a Systematic or a Knowledge-Based strategy. Henceforth, discussions about comprehension strategies will refer to the four strategies: Bottom-up, Top-Down, Systematic, and Knowledge-Based.

8.1.1 Quantification comprehension strategies

To quantify the employed comprehension strategies by the participants, there needs to be structure and regularity when it comes to interpreting the data. Therefore this subsection is dedicated to explaining what criteria are used to assign a strategy to a comprehension process of a participant of a question. For each comprehension question, all thinking steps have been written down, tracking how they flow through the program and possible notes that are relevant to evaluate what kind of strategy is being employed.

Bottom-Up

The Bottom-up strategy as previously defined is a strategy that involves building low-level components into higher-level abstractions, allowing to slowly build a representation of the program from the bottom up until a goal is visible. While this strategy does not necessarily require the programmer to understand the syntax, it does require the programmer to slowly build their representation of the code. In practice when trying to label the process of the programmer we look to understand sequential code blocks separately and place it in the representation that. This means that for comprehension tasks they have to calculate what the output of a program is, they first go through the code that contains the logic, try to understand that, and then perform the calculation. This is especially relevant for questions 1 3 and 4, but also partly for question 7 in the interview. When a participant goes through the code usually line by line, look at what is being called and figure out the functionality by starting at the small code blocks and building this up.
We took a random participant,which is number 15, who with previous knowledge in mind performed a bottom-up strategy for the OO variant of question 1. The following things have been written down on what the thought process was for this participant.

> *Call g with the list provided, then a loop for z in x, loop over the list. Variable names are bad. If < 10 add else subtract. Interesting. I guess I have to run it in my head. Quickly gets to 5, as that is when the condition is triggered. Sees z + y so revises and gets to 7.*

In this regard, it becomes apparent that at first, the participant looks for the end goal of the question. After this goal is clear, they dive into the function g. They see a variable assignment and look at the if-else structure and reason about what this does. Once it's clear that if $<$ 10 a value in the list is added and otherwise subtracted they run the function. With a clear representation that was built up from the bottom they first calculate the result as 5, but recall that z+y should be smaller than 10 the calculate again.
Even though question 1 is small, it is evident that a bottom-up approach has been taken and similar reasoning is used to deduce that for bigger code questions the strategy remains the same. They first look at what happens with the function so they are aware of the context and then dive into the code, usually starting at the start of the function call chain, after understanding what this function does, after grouping the smaller groups s.a. the conditions in a while or an if-else statement, they can capture some essence of a function on at least a functionality level, if not they continue with the possible other functions that are still there.

Top-Down

The top-down strategy is the complete opposite of the bottom-up strategy as this requires the programmer to state a hypothesis and down the road polish the hypothesis as more

information is present. In practice, the strategy entails exactly what is described before. The participant looks through the information provided and uses the context to create an initial guess of what the use of the program might be. This can range from using the function specification(parameters and return type) to glancing through the structure of the program recognizing it and using that as a starting point. In the interviews conducted, we encountered three different ways that the participants used to make an initial guess/hypothesis.

1. Use the function specification and the provided context to guess what the use of a function might be.
2. Recognizing the structure of the function, or part of a function, to guess what the use of it might be.
3. Using the input of a function in a main body to make a guess what the use of it might be.

The first type is in almost all cases when observed for question 5, the implementation of Dijkstra's algorithm. A couple of people used this same type to make an initial guess in question 2 of the merge sort.
The second type when observed has predominately been seen in questions 2 and 3b. In the merge sort exercise people use the deduction of a recursive divide and conquer pattern that it might be a sorting or directly already be a merge sort algorithm. Likewise with the Euclidean algorithm, the structure sets of things that they recognize the distinct swapping of variables and the modulo counting operation.
Lastly, the last type of hypothesis forming is mostly seen in question 7, which contains a big input that allows the participant to reason about its possible usage. Although A majority of people did comment on what the usage might be, only a handful adjusted their strategy on it. This brings us to the following, multiple participants did make initial guesses at the start of a question but did not use this knowledge to try and comprehend things. They simply continued in a Bottom-Up or Systematic way. These executions are therefore not viewed as Top-Down strategies and thus not labeled as such.

Knowledge-Based

A Knowledge-Based comprehension strategy combines elements from the Bottom-Up, Top-Down, and Systematic comprehension strategies. In practice, we observed that this usually entails a Top-Down approach combined with either a Systematic approach or a Bottom-Up approach. It is important to note that those who at one point quickly suggest what it might be and not use this information to adapt their strategy do not use a Top-Down approach. A combined approach means a combined strategy that alternates between strategies and utilizes the information gained from both. We observed that a Knowledge-Based strategy is usually the result of uncertainties at one point during the comprehension process. They might have made an initial guess of what it might be and get stuck down the way such that they change strategy and once a better representation has been built they go back to their initial guess and try to tie it all together from there. An example of such a Knowledge-Based strategy is from participant 24 in question 7.

> *checks the main first, create D with funcA with 2 strings. Something with substrings maybe? A gets 2 strings and list of integers. Confused about what it might be doing. Goes over the lines, Sees the funcB call, and goes into this. Gets a string and outputs an IntArray. goes over the lines. Check if the first*

characters are the same, if not the same, if not the first time running reset varC to the value of one before. Tries to reason about the functionality, counting the amount of the same character? Skip it for now and goes back to funcA. But we do need to know what it does, so it goes back. It does not check for the whole string? hmm, look for the functionality. Doesn't bring me any further so, back to A and go into the whileloop and check the different cases here. sees main input wants to make an assumption stops himself. if chars are the same increase both counters. if at the end add the difference in indexes. When at the varE assignment. Think of it as a representation of a state machine. So it returns [0,5,10,15]. The algorithm does smart things to construct a state machine to optimize the search.

During the process of answering question 7, the participant ranged from using a Top-Down strategy looking for possibilities to see how it checks for a substring, but dove into function B which felt abstract and required a Bottom-Up strategy. This continued until it connected that the intArray could be some state machine diagram and this led to the recognition that the earlier hypothesis was indeed correct allowing them to finalize an answer.

Systematic

Lastly, we have the Systematic comprehension strategy, in this strategy, the programmer goes with the flow of the program to figure out what the goal is. From our observation, a systematic strategy is usually employed in rather calculation-heavy exercises such as question 3 and question 1. Additionally, this strategy has also been used multiple times within question 6. They are tasked with figuring out what the purpose of the code is and what is missing. The participants use a Systematic strategy to try and understand what the flow of the program is like and where possible issues arise. So if a participant uses the flow of a program and the information they used beforehand to create a representation of the program, we assigned the Systematic comprehension strategy.
We clearly distinguish between Bottom-Up and Systematic, as smaller programs could look similar to each other, as both go through the code. We only assign a systematic approach if they only read the parts of the code that are required at that specific time, if they don't a Bottom-Up approach is assigned.

8.2 Quantitative results

Now that the comprehension strategies are quantified we can use them within the quantitative results to deduce useful results that allow us to say something about code comprehension between multi-paradigm and object-oriented programs. We first present the results of the demographic information after this we report the results based on 3 different perspectives namely: question-based, version-based, and comprehension strategy-based.

8.2.1 Demographic results

Table 8.1 displays the demographic data retrieved from the 30 participants. Both Java experience and Kotlin experience are measured in years and where applicable rounded to half years. Additionally, the years of experience for professionals is the number of years they are working professionally, whereas for students it depicts the year they are currently in study progress-wise.

participant	Java experience	Kotlin experience	profession	experience	FP rating(1-5)
1	19	3	professional	15	4
2	4.5	5	professional	11	4
3	1	0	professional	0	2
4	1	1	professional	3	3
5	20	0	professional	25	2.5
6	5	0	master	1	3
7	5	0	master	2	4
8	7	0	master	2	2.5
9	1	0	master	2	4
10	2.5	3	master	2	2
11	2	0	master	2	3
12	8	2	professional	10	4
13	1	0	professional	5	4
14	18	2.5	professional	18	2
15	3	0	master	2	4
16	1	0	professional	0	1
17	4	0	bachelor	3	4
18	5	4	master	2	4
19	4	0	master	2	4
20	2.5	0	master	2	2
21	2	0.5	master	2	2
22	5	0	master	1	2
23	2	0	professional	1	2
24	5	1	professional	1.5	3.5
25	5	0	master	2	4
26	10	0.5	master	2	3
27	6	0	master	2	4
28	5	0.5	master	2	4
29	12	3	master	2	4
30	2	0	master	2	2.5

Table 8.1: Overview of the gathered demographic data

As the table displays 11 professionals were recruited and 19 students. Of these professionals, five of them graduated recently, whereas the other six have been working for a longer time. Of the students, 18 are master students and one is a bachelor student close to completing the bachelor's in their third year.

All participants have had some experience in Java, whereas the people with limited experience(1-2 years) encountered it within their studies but did not use it besides the study. Out of the 30 participants 12 have done some programming in Kotlin before, and the remaining 17 participants have never done anything with Kotlin before.

8.2.2 Questions

We first report the results on a question basis. In this section, it describes how the questions are answered and how they are performed. First, no distinction is made between students and professionals, in the end, the results of them separately are plotted.
In Figure 8.1 the correctness percentage of each question is displayed. Additionally, the

correctness percentage for the Object-Oriented version and Multi-paradigm version are displayed as well. Here we see that for most questions both versions performed similarly. But for question 3b the multi-paradigm version scored slightly better, whereas questions 5 and 6 were significantly better answered for the object-oriented versions.

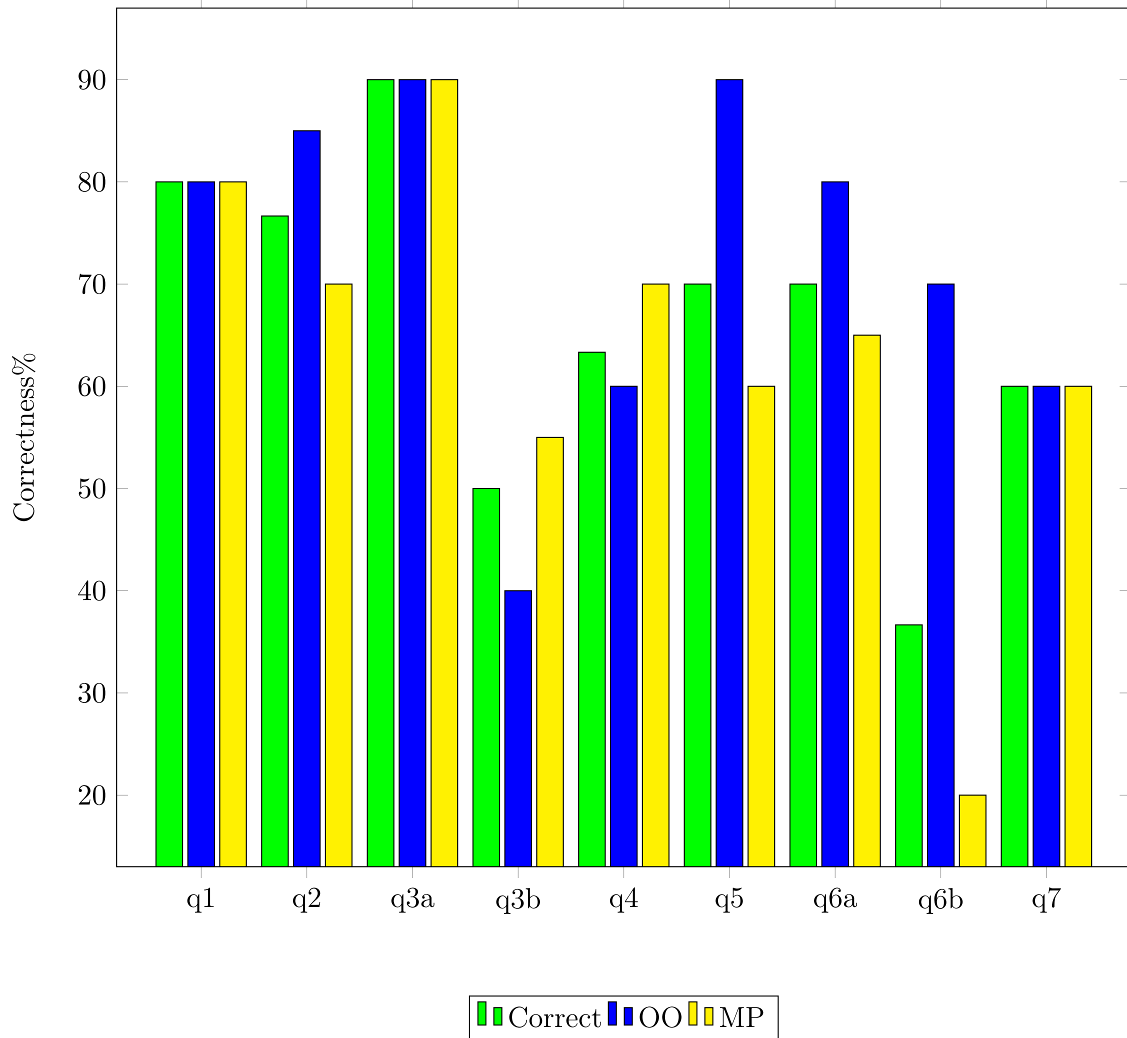


FIGURE 8.1: Correctness of question answers

Many participants answered the questions correctly with confidence, but not all did and made an educated guess. We captured whether a participant guessed their final answer correctly. This means that the participant is not yet sure whether their answer is correct or not. This was determined at the end of the question by asking whether they were certain that it was their final answer. In Table 8.2 you can see for which questions the amount of guessed answers, similar to previous figures the OO and MP versions are separated to allow for a better overview of the distribution. We can see that especially for questions 5 and 7 multiple participants had to make an educated guess to answer the question.

Next up we have the time distribution of the questions without the distinction of version type in Figure 8.2. For each of the questions, the average time is denoted with a cross. We see that questions 1 to 4, look similar besides question 2 which took a bit more time for most, as this was also the slightly bigger question. The last 3 questions, which had a time limit of around 10 minutes, have a similar distribution as well. In Figure 8.3 the boxplots

Question	OO	MP
1	0	0
2	2	0
3a	0	0
3b	0	1
4	0	0
5	0	5
6	0	0
7	4	2

TABLE 8.2: Educated guesses per question

of each question separately are displayed comparing the results of the OO versions and the MP versions.

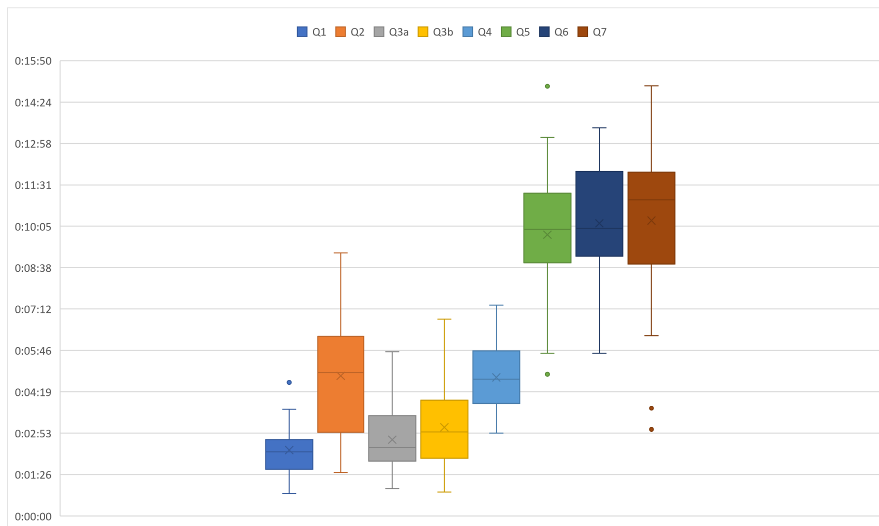


FIGURE 8.2: Time distribution question 1 to question 7

As previously explained, 12 participants already had Kotlin experience in Table 8.3 the distribution of questions answered correctly displayed of those with and those without Kotlin experience. We can see that the participants with Kotlin experience performed

experience in Kotlin	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total	average correct
>0 years	11	9	11	11	6	10	9	4	7	78	6.5
0 years	13	14	16	13	9	11	12	7	11	106	5.9

TABLE 8.3: Distribution of correct answers based on Kotlin experience

slightly better compared to those who had no experience. It seems that they performed slightly better especially in the smaller questions when compared to those that have no experience and perform similarly in the other questions. Additionally, 19 participants were students and 11 were professionals. Table 8.4 displays the number of correct answers for the students and the professionals. As you can see the students performed better compared to the professionals, more about this in Section 8.3.3.

	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total	average correct
student	15	15	18	14	14	13	14	10	11	124	6.9
professional	9	8	9	1	5	8	7	1	7	55	5.8

TABLE 8.4: Distribution of correct answers students and professionals

8.2.3 Versions

For each question, a multi-paradigm version and an object-oriented version were created. To evaluate these different question versions, three different question compositions were created. Each of these compositions is used for 10 participants. The question distribution is depicted in Table 8.5. With this kind of distribution, it means that for each question version either 20 or 10 participants have answered it.

	composition 1	composition 2	composition 3
1	OO	MP	MP
2	MP	OO	OO
3	MP	MP	OO
4	OO	OO	MP
5	MP	MP	OO
6	MP	OO	MP
7	OO	MP	OO

TABLE 8.5: Question distribution

Additionally, it is important to check whether the versions have big differences or if they perform similarly. Table 8.6 describes the amount of correctly answered questions for each composition. It can be noted that all compositions perform similarly except for composition 2 which performs slightly better, as it seems to have performed better for question 2. This might have to do with the fact that composition 2 was the only composition that got the OO variant for question 6.

composition	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total	average correct
c1	8	7	10	6	5	6	8	2	6	58	5.8
c2	8	9	8	5	7	6	8	7	6	64	6.4
c3	8	7	9	4	7	9	5	2	6	57	5.7

TABLE 8.6: Correctly answered questions per version

Besides seeing the performance of each composition it might also be interesting how the OO and MP versions score when looking at the professionals and the students. In Table 8.7 it is displayed what the distribution was for the students and professionals on the OO and MP questions.

	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total answered
professionals MP	8	3	6	6	5	6	8	8	3	53
professionals OO	3	8	5	5	6	5	3	3	8	46
students MP	12	7	14	14	5	14	12	12	7	97
students OO	7	12	5	5	14	5	7	7	12	74

TABLE 8.7: Question distribution for each question version

As you can see both the students and professionals have answered slightly more multi-paradigm questions compared to object-oriented questions. Table 8.8 displays the amount of correctly answered questions and displays the success rate for the OO and MP questions. It can be noted that the students perform generally better than the professionals for both multi-paradigm questions and object-oriented questions, which is also displayed in Table 8.4.

	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total correctly	success rate(%)
professionals MP	7	2	5	1	2	3	5	0	1	26	49.1
professionals OO	2	4	4	0	3	5	2	1	6	27	58.7
students MP	9	5	13	10	5	9	8	4	5	68	70.1
students OO	6	12	5	4	9	4	6	6	6	58	78.4

Table 8.8: Correctly answered questions for each question version

8.2.4 Comprehension strategies

Besides the results on a question basis, the results regarding the comprehension strategies are also interesting to look at. This section describes the results of the different comprehension strategies: Bottom-up, Systematic, Top-down, and Knowledge-based. Firstly we display the distribution of the strategies in general and make a distinction between students and professionals. In Table 8.9 you can see the distribution per strategy. In the table,

Strategy	total times employed	Employed by students	Employed by professionals
Bottom-up	130 (54.1%)	75 (49%)	55 (62.5%)
Systematic	41 (17.1%)	23 (15%)	18 (20.45%)
Top-down	48 (20%)	41 (27%)	7 (7.95%)
Knowledge-based	21 (8.8%)	13 (9%)	8 (9.1%)

Table 8.9: Distribution of strategies employed

you can see that the majority of the questions have been answered using a bottom-up or systematic strategy, while a much smaller percentage has been answered using a top-down or knowledge-based strategy.

Secondly, the distribution of employed comprehension strategies can be found in Figure 8.4.

Figure 8.4 already illustrates the distribution of the strategies per question, but not yet the success rate of the strategy. This distribution can be seen in Table 8.10. It should be noted that the comprehension strategy was assigned to the entirety of question 6, so when checking for correct answers for question 6 the number is counted twice when calculating the success rate.

Strategy	q1	q2	q3a	q3b	q4	q5	q6a	q6b	q7	total correct	success rate
BU	20	7	13	2	14	5	10	5	10	86 out of 148	58.1%
SY	4	0	9	4	1	0	8	5	0	31 out of 50	62%
TD	0	15	2	6	3	12	1	0	6	45 out of 49	91.8%
KB	0	0	3	3	1	4	2	1	2	17 out of 23	73.9%

Table 8.10: Success rate per comprehension strategy

8.3 Qualitative results

While the quantitative results do cover multiple interesting parts, they have little impact when using them on their own when reasoning about what they mean. To give those results more meaning the qualitative results are used to describe the different observations made that can be linked to the quantitative results. Additionally, using the Grounded Theorem analysis method as described in Chapter 6.3.
We first describe which interesting observations were made during the interviews. The first step of coding the text has already been done allowing us to quantify the comprehension strategies of the participants. The data that was used to deduce these things combined with quotes made by the participants will form the basis of our analysis.
Secondly, we describe the theories that follow from the observations made and explain what the theories would entail.
Lastly, we refine the theories and analyze the results to either confirm or negate the theory.

8.3.1 Observations made

the different observations made can be grouped into certain categories. In this section, the different categories including which observations were made are described. This does not yet say anything regarding building any theories and merely reports which things stood out including quotes where needed. These observations are made based on the entire process but also include observations that dive into specific questions or types of questions.

Recognizing patterns

One of the first observations made is that some participants use the context of the questions, which could be: code structure, function specifications, or for question 5 the graph context. While going through the code to answer the questions, some stop to try and reason with the current information at hand. Within these moments hypotheses are formed that dictate the direction the participants take for solving the questions. While it's seen with both professional participants and student participants, from the observations made students are more likely to use this approach in questions that allow for such reasoning. The questions respectively that are observed to allow for this style are 2, 3, 5, and 7.

Not looking at the bigger picture

This observation is opposite from the previously made observation. Some participants take a completely different approach and choose(or not) to not use any previously acquired information to make premature conclusions. These participants rather seem to first want to read all the code in the question and reason about them, by understanding small chunks of code. After these smaller chunks are understood towards the end of answering a question, is where these participants make their first guesses on what the program does. This way of working is mostly seen with professionals, students are also observed using this approach but seem to change strategy here and there earlier if it seems like a good fit.

Multi-paradigm for professionals

The participants are given questions either from an object-oriented perspective or those of a multi-paradigm perspective. It seems that students in general don't have much difference in being able to solve the questions, the professionals seem to have a much harder time, especially the multi-paradigm questions. This group, especially the professionals who work

for a longer time, was very vocal in their opinions on functional programming constructs. They recognized that they were dealing with functional programming but expressed negative feelings toward it. An example of this is the participant 5. One of the things this participant said was:

> *I much more prefer Imperative programming this is much clearer to understand for the reader.*

It looks like Participant 5 is not the only participant who struggles with the combination of imperative programming and declarative programming. The exercises, like 1 and 4 seem to have a clear enough separation between declarative and imperative that allows them to answer the questions correctly. The other bigger questions where there no longer is this clear separation seem to be the questions the more experienced participants struggle more. Of course, some questions are more difficult than others, but the students seem to be able to adapt better to the mix of programming paradigms. The exercises containing purely object-oriented programming are much better answered by the professionals compared to the multi-paradigm questions. From the observations, they seem less likely to get stuck in figuring out what to do next if the question is purely imperative and does not contain declarative statements.

Working as a compiler

This observation, while similar to other observations made still deserves to be mentioned separately. It was noted that some participants are more extreme than others when it comes to their comprehension strategy. Of those participants who do not look at the bigger picture, some try to comprehend the code like a compiler would. Seeing everything line for line and trying to understand the intermediate representation of the program. Most of these thinking processes can be labeled as a systematic strategy.

Multiple lambda declarations

The multi-paradigm version of question 6, seems to be the most difficult question, especially the second part of the question. It was seen that the participants openly struggled with understanding the newly defined higher-order functions. The fact that the higher-order lambda is specified into two places and only becomes more concrete when having everything puzzled together gives the participants a hard time. In the end, most participants can figure out some things go wrong in the matrix calculation but usually try to pinpoint the issue of the usage of the higher-order function usage, as it is unclear what the exact functionality is of this.

Hard to follow variables without meaning

One of the most recurring comments that were given during the interviews, is the lack of proper variable names. The variable names in the questions have been stripped of any meaning, besides possibly saying something about their type. This is something that is mentioned as annoying with pretty much all of the participants. Only a small portion of the participants actively tries to tackle this issue. This small portion is renaming variables and making small alterations to the code to make it more readable for them, such that it helps them in the comprehension process. Even though the participants were specifically told at the start of the interview that they were allowed to do anything with the code, apart from running it, close to all simply did not touch the code and decided that reading was the way to go.

8.3.2 Theory building

Many observations were made, with the most noteworthy ones concretely documented. These observations revealed recurring themes connecting various observations. A prominent theme emerged concerning the employed comprehension strategies, particularly highlighting a distinction between professionals and students in their strategy application. Students exhibited greater flexibility when tackling multi-paradigm questions, whereas professionals tended to adhere more steadfastly to their chosen strategy, which is either a systematic or a bottom-up approach. This observation leads us to formulate our first theory:

Theory 1:

Students are more adaptable in employing different comprehension strategies for multi-paradigm questions than professionals.

Another notable observation is that, for the majority of the multi-paradigm questions, there is not much difference in participants' approaches, except for questions 5 and 6. It seems that when the multi-paradigm code is not deeply intertwined within the code, participants can adequately reason and comprehend it. However, when the declarative aspects become increasingly mixed with the imperative side, comprehension becomes less clear and more challenging. This leads us to the following theory:

Theory 2:

Multi-paradigm programs become less comprehensible only when the boundary between declarative and imperative sides is no longer clear to the reader.

Examining the observation in question 6, where multiple lambda declarations cause confusion, reinforces this theory. Multiple lambda declarations in different places make it more challenging to follow the program's flow, as one must consider multiple points in the program. This complexity is also evident in question 5, where the multi-paradigm version loops over the list of nodes using a fold. While this caused only minor confusion, the large specified lambda, part of a significant imperative sequence, confuses participants regarding its functionality. Although participants may understand segments of the program, the inherent uncertainties, caused by the programming style, often result in educated guesses. Table 8.2 shows that 5 out of 12 correct answers were based on educated guesses, often linked to the graph nature and participants discerning manipulations involving edges and weights.

The last takeaway from the observations concerns a specific type of comprehension strategy. It appears that those who can form hypotheses about the possible functionality of the code are more likely to answer the question correctly. Individuals who recognize patterns and adjust their comprehension process accordingly—employing the "top-down" strategy and the "Knowledge-based" strategy—are better able to answer the questions correctly. This leads to the formation of the following theory:

Theory 3:

Employing a top-down or knowledge-based comprehension strategy is more successful than a bottom-up or systematic approach.

8.3.3 Theory refinement

In the previous section, 3 different theories are proposed based on the observations that are made. This Section will go into more detail about the theories and go through the results to verify the theories that were created based on the observations. Chapter 9 will cover what the implications are of each theory and give possible interpretations of what the theories could mean.

Theory 1

The first theory is regarding the differences in strategies employed between professionals and students. If we look at Table 8.9 we can see that in more than 80% of the cases, a professional used either a bottom-up or a systematic comprehension strategy to answer whereas for the students this is 64%. The main difference in this regard is that students used a top-down approach in 27% of the cases whereas professionals did this in only 8% of the cases. From the 7 times professionals used a top-down strategy, they were employed by 4 of the 11 professionals. For the students on the other hand, of the 19 students, only 2 students have not used a top-down strategy. If we look at the distribution of strategies employed for each of the questions in Figure 8.4, we can see that the top-down strategy is mostly employed in questions 2, 3b, 5, and 7.

As discussed before these are questions that contain plenty of context for the participants to work with and allow them to employ a hypothesis-driven strategy. The other questions contain much less context, making it a lot harder to be able to make an initial hypothesis regarding the functionality of the program. All in all, when coming back to the theory, it seems that it is indeed the case that students can switch their strategies since 4/11 professionals used a top-down strategy whereas 17/19 students used a top-down strategy for one of the questions.
While this indeed seems to be the case, aren't there possible factors that are currently not taken into account that might have influenced the participants when trying to answer the questions? One of these factors is that questions 2,3 and 5 all contain elements that are taught during the Bachelor program of Technical Computer Science in Twente, where the majority of the student population is from. They are taught early about searching algorithms and later on they get into contact with different algorithms, including the Euclidean algorithm and algorithms performed on graphs. Only question 7 is regarding a topic not taught in the program. The fact that this knowledge is still rather close in time when taught, might be a reason as to why students can recognize patterns easier and thus be able to form hypotheses earlier than the professionals. In the book The Programmer's Brain [26] Hermans speaks about different cognitive processes programmers use when performing a cognitive task. These different processes can be seen in Figure 8.5. With these different cognitive processes, describing which processes are used for each comprehension strategy is possible.

- Bottom-up: In a bottom-up strategy the programmer predominately uses short-term memory(STM) within their working memory to build up an understanding of the program from the bottom up.
- Systematic: Similarly to the bottom-up strategy the programmer mostly relies on the short-term memory to gain an understanding of the program, the only difference is the way the short-term memory is utilized. With a systematic approach, only the relevant lines within the flow of the program are processed.
- Top-down: For the top-down strategy the programmer utilizes long-term memory to retrieve previous knowledge about certain structures and the short-term memory to process what is being written down.
- Knowledge-based: With this strategy, the participant mainly focuses on short-term memory until something is recognized. From that moment on, the long-term memory combined with the information saved in the short-term memory a hypothesis can be formed.

So instead of saying that students are more adaptable in employing different comprehension strategies, it becomes more clear that students are rather better at utilizing the long-term memory when comparing this with the professionals. Once again, it does help that the questions that allowed for a top-down approach contained information that all students have learned before, whereas this cannot be guaranteed for the professionals. So the more refined theory is as follows:

Theory 1:

Students utilize their long-term memory knowledge gained during their studies, better than professionals when performing code comprehension questions.

Theory 2

The second theory says something in general about multi-paradigm programming. It states that by using multi-paradigm programming constructs while retaining a clear boundary between the imperative side and declarative side there are no issues. Once this boundary no longer is separated, the multi-paradigm programs become much harder to comprehend. If we look at the boxplots for the questions 8.2, we see that for question 5 and for question 6 the majority of the participants spend the full 10 minutes answering the questions for the multi-paradigm part, whereas the object-oriented part for both of the questions is answered much faster, where plenty of people did not need the complete 10 minutes to answer the question. Additionally, when looking at the distribution of questions answered correctly in Figure 8.1 we see that for questions 1, 3a, 4, and 7 there are little differences between the object-oriented and the multi-paradigm questions.

For question 3b the multi-paradigm variant performed better than the object-oriented variant. A possible reason for this is similar to the object-oriented version of question 2. The format of these two questions is similar to how they are presented went taught in school. Which makes this easier for the participants to read and possibly easier to recognize in an earlier stage. This leaves us with a big discrepancy for questions 5 and 6

respectively where the object-oriented versions scored significantly better than the multi-paradigm versions, for question 5 this difference is 30%, for question 6a this is 15% and for question 6b this is 50%.

Question 6a is rather close in performance, the reason for this might be, that participants mention that based on the things they read, such as checking whether a number is even, seeing a filter in function C, and lastly the output, most participants were still able to deduce what the expected output would be. When further explaining and figuring out where the bug in the program is located, participants mention that they don't know due to the dense mess of lambda declarations. They struggle with understanding what the implications are of the defined lambdas and either run out of time, or they cannot give a correct answer.

In question 5 the participants struggled with understanding the construction of the fold combined with its big lambda, whereas there were little complications with the object-oriented variant. With all these results and observations on the participants, it becomes apparent that multi-paradigm constructs which no longer contain a clear flow, due to the lack of separation of the imperative side and declarative side negatively impact the comprehension of the participants. As long as it is still clear to the programmer what the use is of the multi-paradigm constructs, by following its flow when reading the code, there are no differences with the object-oriented questions. This all confirms the theory that was previously described and is as follows:

Theory 2:

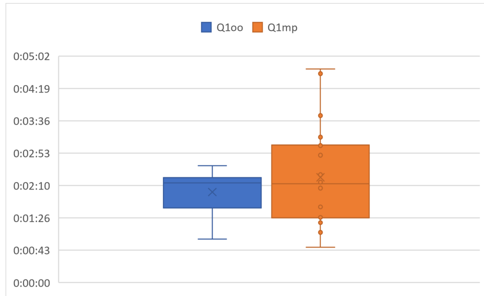
Multi-paradigm programs become less comprehensible only when the boundary between the declarative and imperative sides is no longer clear to the reader.

Theory 3

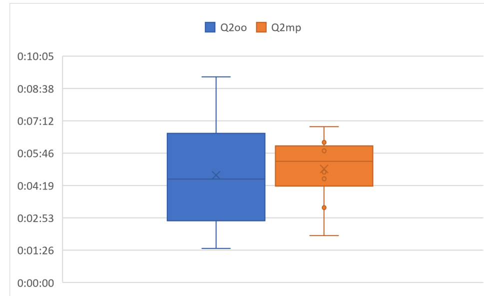
The third and last defined theory is regarding the different employed comprehension strategies. From the results, it becomes clear that a top-down or knowledge-based strategy is much more successful than a bottom-up strategy or a systematic strategy. Table 8.10 shows the distribution. While a bottom-up strategy and systematic strategy have success rates of roughly 60%, whereas the success rate of the top-down strategy is almost 92%. Additionally, the success rate of the knowledge-based strategy is roughly 74%. Both approaches that involve forming a hypothesis perform better than the strategies that do not involve this. When once again looking at the cognitive processes described by Hermans [26], we can see that the strategies that utilize long-term memory have a better performance, than the strategies that utilize long-term memory much less. With this information, we can make the previously defined theory more specific and relate it to long-term and short-term memory usage. This gives us the following theory:

Theory 3:

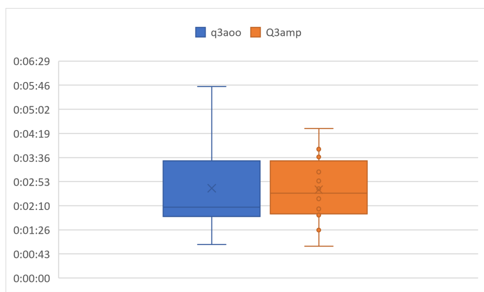
Forming hypotheses during the code comprehension process helps with successfully identifying the functionality of the code.



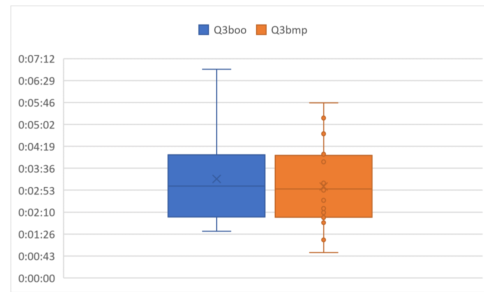
(A) Boxplot question 1



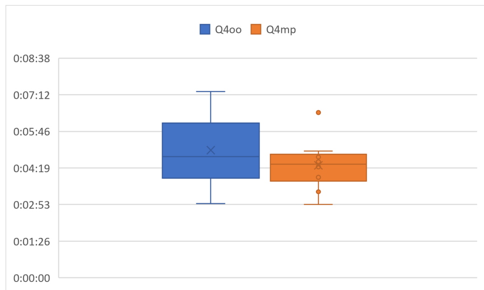
(B) Boxplot question 2



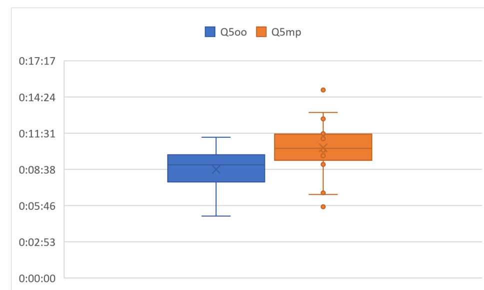
(C) Boxplot question 3a



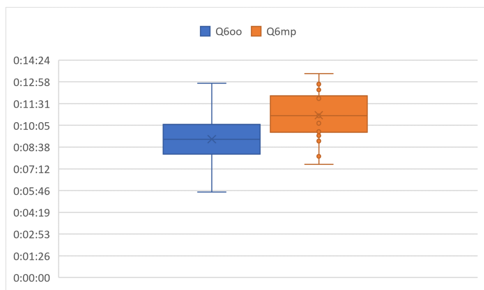
(D) Boxplot question 3b



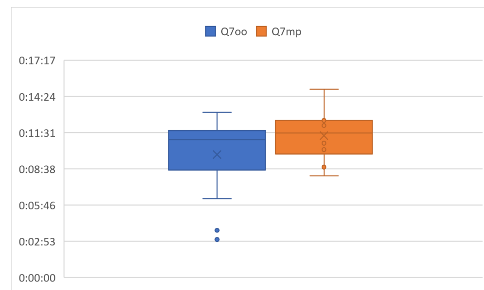
(E) Boxplot question 4



(F) Boxplot question 5



(G) Boxplot question 6



(H) Boxplot question 7

FIGURE 8.3: Boxplots question 1-7 OO and MP versions

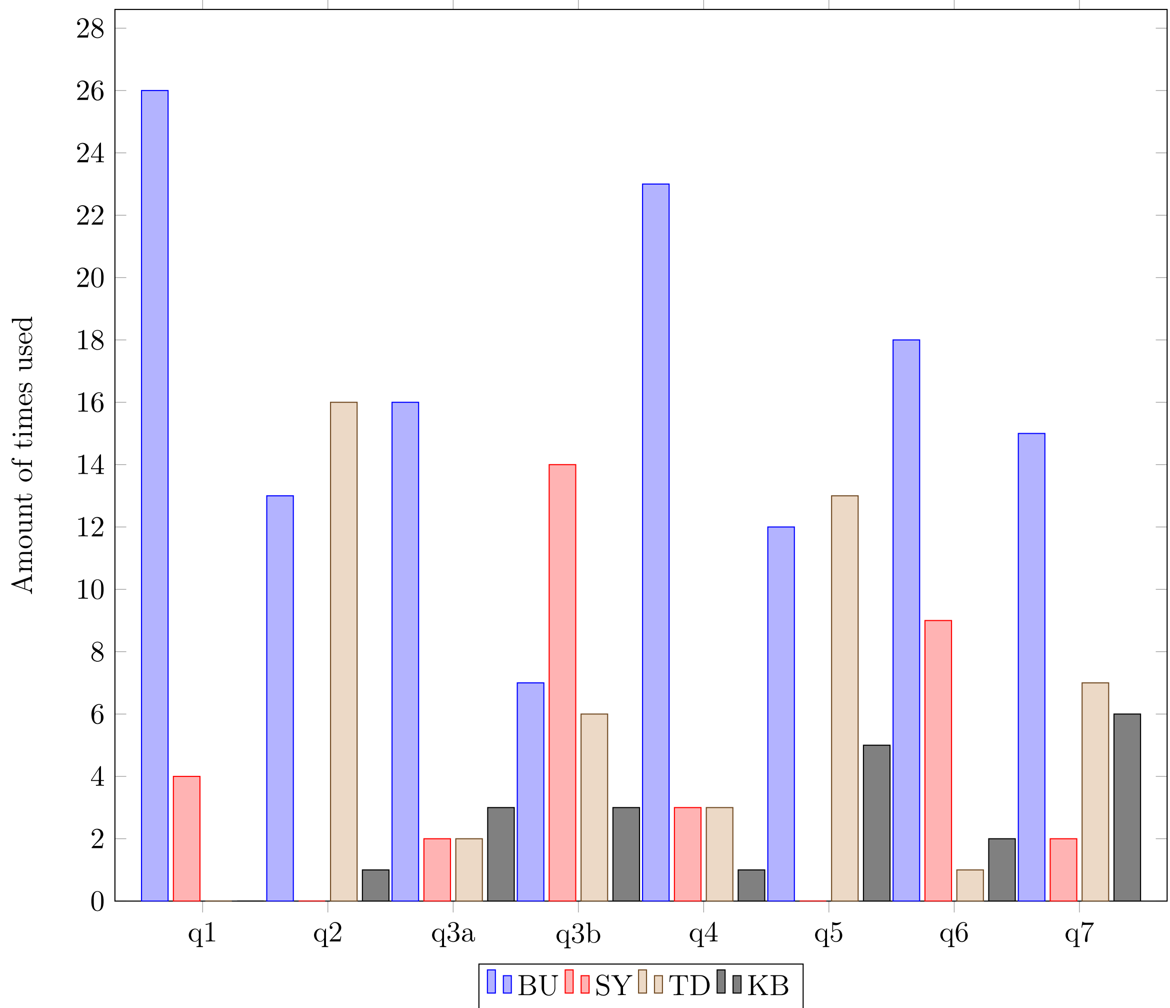


FIGURE 8.4: Distribution of comprehension strategies per question

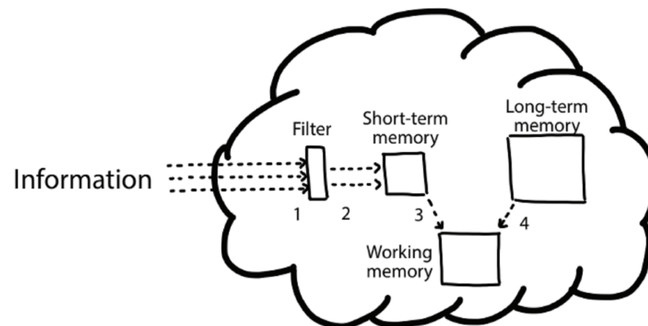


FIGURE 8.5: Cognitive processes used when programming by Hermans [26]

Chapter 9

Discussion

9.1 Implications

Chapter 8 concluded with the formation of three theories that emerged from the observations and are confirmed using the quantitative data. This Section describes what the implications of each of these theories are and what they mean in the grand scheme of things. Additionally, we give possible reasons for these specific outcomes.

9.1.1 Leveraging Long-Term Memory for Code Understanding

"Students utilize their long-term memory knowledge gained during their studies, better than professionals when performing code comprehension questions."

From the results, it becomes clear that students performed better on the code comprehension questions because they utilized their long-term memory better than the professionals did. Most professionals (7/12) did not use a top-down strategy to answer the comprehension questions. This does seem odd since the professionals also studied. The only difference is that they are used to a certain way of working which might imply that the professionals try to stick to the way of working, whereas the students have more freedom when it comes to solving these comprehension questions. Additionally, the context questions are mostly regarding material the University of Twente students covered in their recent years, which is not something that can be guaranteed for the professionals. After the interviews when going over the correct answers to each question, most professionals did indicate they knew the different algorithms. To further test this theory, it would be interesting to perform a similar research containing code comprehension questions with context, where the context is not mainly regarding material taught in the university. This would mean more questions similar to question 7, where once again the students were able to differentiate with comprehension strategy. This question did have a higher usage of the knowledge-based strategy, this was mostly due to confusion at the start where the participants starting with a top-down approach changed towards a bottom-up/systematic approach to figure out what part of the code does.

9.1.2 Multi-paradigm code boundary threshold

"Multi-paradigm programs become less comprehensible only when the boundary between the declarative and imperative sides is no longer clear to the reader."

Besides for questions 5 and 6, the boundary between the declarative side of the multi-paradigm constructs and the imperative sides of the constructs is clear. In these questions, all comprehension strategies were successfully able to answer the questions, whereas, for questions 5 and 6, the mostly bottom-up and systematic approach was much less successful for the multi-paradigm variants than the object-oriented variant. A reason for this would be that with these approaches the programmer mostly relies on its short-term memory when solving the questions. When the flow of the program is clear, it becomes easier for the programmer to create a cognitive representation of pieces of the code. The multi-paradigm variants no longer contain this clear flow which makes it more difficult for the programmer to create this cognitive representation of pieces of the code. They try to puzzle things together but have too many pieces that do not make sense making the comprehension process increasingly difficult. It is therefore important when wanting to apply multi-paradigm programming constructs to your code to carefully consider the flow of the program and the declarative and imperative aspects of the constructs. Once this becomes blurry it would be better to stick to a pure object-oriented solution or a fitting functional programming solution. Multi-paradigm programming can be used within your program, but one must be careful and try to put in perspective how others should interpret it. Only apply this within a project if those that have to work with it are familiar with the style of programming, otherwise, stick to the style the team is used to.

9.1.3 Combined cognitive processes trigger for successful code comprehension

"Forming hypotheses during the code comprehension process helps with successfully identifying the functionality of the code."

When forming hypotheses, one utilizes their long-term memory within the code comprehension context, we talk about using a top-down or a knowledge-based approach. This in turn means that the participant is more or less hypothesis-driven. Following from the results this kind of approach performs better the bottom-up and systematic strategy. A possible, and plausible reason for this is that once something is recognized a participant can form a hypothesis on the possible functionality of the code. When the participant can continue to confirm or refine a hypothesis the chances of answering the question increases, since an early feeling keeps getting confirmed. Whereas, simply relying on reading the code and trying to build a mental model from nothing has much more potential to fail, as the participant is required to understand previously read code. So this theory is in line with expectations when looking at the success rate of the different comprehension strategies.

9.2 Limitations

9.2.1 Approach

The process and approach to this research have been carefully considered. However, the main researcher performing and setting up this interview is not specialized in researching

cognitive behaviors and hasn't conducted interviews in such a setting before. With this inexperience, the approach and execution are not optimal and might contain inaccuracies which could lead to slight deviations in the results. While this is the case, most processes are carefully written down and have been discussed with those more experienced. But without experience, it might be the case that certain interesting observations have gone unnoticed.

9.2.2 Language

The results of our study are promising regarding the effects of using multi-paradigm programming when programming. However, it should be noted that these results cannot be generalized yet. The participants performed the comprehension questions written in Kotlin. A majority of the participants haven't used or written Kotlin before the start of the study. While the results do give a good impression of the different comprehension strategies that are employed, these results cannot be generalized. They must remain within the context of Kotlin. Section 9.3 describes that more similar research is needed applying this research to other programming languages to allow for examining the results from the different studies and see whether the observations align with the expectations. Even with these possible limitations, we are confident that the results from the interviews do serve as a good basis for future research on multi-paradigm programming and its impact on code comprehension.

9.2.3 Threats to validity

Internal validity

- Change in leading the interview: The researcher who led the interviews has never done anything like this before. That means that by performing each different interview the researcher learns more about performing the interviews better. This could potentially lead to a discrepancy in the conduction of the interviews. To mitigate this risk as much as possible, a large procedure was written down, which can be read in Chapter 6. Parts of the interviews were more easily conducted and directly asked more relevant questions after performing a few interviews. With the procedure, it was ensured that for each of the participants, the same information was distributed, and similar (follow-up) questions were asked. Hence, we believe that issues regarding inexperience in conducting the interviews are more than enough mitigated.

- Paper selection bias: For the second research question a literature research has been conducted. For this, the papers of the ICPC were evaluated. While many program comprehension papers were read and judged, a large portion of program comprehension papers was not considered. This might create a bias in the selection of the papers. As described all research needs to be scoped, and a conference dedicated to program comprehension is the best way to have the most relevant papers in place. Therefore, we do not think there is a threat to the validity of this selection. The selection procedure has also been clearly described allowing others to try and confirm the results themselves.

External validity

- Sampling Bias: With the current selection of participants, which are students from the University of Twente and mostly professionals working at Info Support, there

might be a bias with the population. Now this could be the fact that this demographic is not properly representing the broader population of programmers. in Chapter 6 the inclusion criteria for participating in the study for both professionals and students are described. These criteria are there in place to ensure that the expected knowledge for the questions is met. Therefore, while having a limited variety in the population and mainly drawing participants from two institutions, we believe that the sampling population is still representative of the broader population.

- Applicability study environment: The study environment encompassed a 1 on 1 interview where the participant had to answer 7 code comprehension questions where the variables were, mostly, stripped from any meaning. This potentially endangers the applicability of the study in the real world. As most participants mentioned they struggled with the meaningless variable names, which differs from a real-world application. While this might be true, the goal of the study is to explore the possible impacts of multi-paradigm programming on code comprehension. It was specifically chosen to work with smaller code questions. The implications of the results of this study are still required to be validated. With these things in mind, it is expected that the study environment still simulates the real-world application enough to gather relevant information about the impact of multi-paradigm programming on code comprehension.

9.3 Future work

The research conducted for this thesis contains interesting results but mostly serves as an exploratory study to see what the implications of multi-paradigm usage is. To come up with more concrete and more precise results more research needs to be done in the field of multi-paradigm programming.

9.3.1 More focus on multi-paradigm constructs

We proposed four multi-paradigm constructs. We do not claim that these are the only multi-paradigm constructs and possibly more constructs can and should be defined. We mostly focused on the combination of functional programming constructs and object-oriented constructs. Additional research needs to be performed to create more multi-paradigm constructs that not only focus on the structure of constructs.

9.3.2 Different kind of context

The current questions used for the code comprehension that contained context, are mainly in an algorithm context. To gain more insight, additional research with other types of context in the code comprehension questions. This allows for additional evaluation to check whether the results are in line with expectations.

9.3.3 Validate current findings

As mentioned before, the research performed in this study was exploratory. This means that the findings of this study still need to be validated through other research that validates the results and findings from this study. This can be done by recreating this study according to the information given in Chapter 6 and Chapter 7. Additionally, more questions should be defined using our defined multi-paradigm constructs allowing better comparison of the performance of the constructs.

9.3.4 More programming languages

Within our study, we focused on defining multi-paradigm constructs in Kotlin and studying the impact on comprehension within this programming language. For most of the participants, Kotlin has been a language they haven't used much. Future research should expand by possibly comparing the performance between programming languages. But first, a similar study should be performed with the comprehension questions written in a different programming language. Once this is done, there are possibilities to compare the results from the different languages and see whether there are implications that are language agnostic.

9.3.5 Bigger comprehension questions

Our study has focused more on the small code questions, of a maximum of 60 lines of code per question. This does give us some great insights into the different comprehension strategies employed by programmers, but this behavior can't be compared with the behavior when programmers are working on bigger projects. Future research could focus on performing a similar study, but rather using separate smaller comprehension questions to use a bigger, possibly real-time code base. This code base should of course contain multi-paradigm programming segments.

Chapter 10

Conclusion

This Chapter concludes and answers the research questions, of our exploratory study, as formulated in Chapter 1.1.

- **RQ1: Which multi-paradigm constructs can be identified when combining object-oriented programming and functional programming to study code comprehension?**
- **RQ2: How can we study the impact on code comprehension in multi-paradigm programs?**
- **RQ3: What is the impact of multi-paradigm programming on code comprehension in Kotlin?**

10.1 RQ 1

The first research question asks about which multi-paradigm constructs can be identified when combining object-oriented programming and functional programming. We have identified and specified 4 different multi-paradigm constructs based on functional programming constructs and object-oriented constructs. As Chapter 7 describes we constructed the following constructs: impure lambda functions, imperative lambda functions, encapsulated higher-order functions, and lastly branched pattern matching.

10.2 RQ 2

To accurately research the possible impacts of multi-paradigm programming on how we can study the impact on code comprehension in multi-paradigm programs. To answer this question, we performed a literature study where we went through the papers of all procedures of the ICPC, and performed 3 selection rounds. After these selection rounds, we were left with 38 papers that are researching similar setups to our research. From these papers, we concluded that to study the impact of something, both qualitative data and quantitative data needed to be gathered. The most fitting type of study for this would be that of semi-structured interviews in which the participants would answer code comprehension questions. Surveys were deemed unsuitable since the researcher was unable to steer what qualitative data needed to be collected. The qualitative data would be analyzed using grounded theory due to the exploratory nature of the study.

10.3 RQ 3

The third and last research questions researches the impact of multi-paradigm programming on code comprehension in the language Kotlin. From the results, we constructed three different theories regarding multi-paradigm programming and code comprehension. Multi-paradigm programs do not impact code comprehension when the boundary between the declarative side of the construct and the imperative side of the construct is clear. When this boundary fades and the flow within the program becomes unclear, the programmers have a harder time trying to comprehend the code which results in more in an incorrect or missing answer. Additionally, we conclude that hypothesis-driven comprehension strategies are more successful than the bottom-up or systematic comprehension strategies. This is due to the utilization of the long-term memory.

Bibliography

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering*, 24:287–328, 2019.

[2] Eran Avidan and Dror G Feitelson. Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 55–65. IEEE, 2017.

[3] Gina R Bai, Joshua Kayani, and Kathryn T Stolee. How graduate computing students search when using an unfamiliar programming language. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 160–171, 2020.

[4] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C Hofmeister, and Sven Apel. Indentation: simply a matter of style or support for program comprehension? In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 154–164. IEEE, 2019.

[5] Keith H Bennett, Václav T Rajlich, and Norman Wilde. Software evolution and the staged model of the software lifecycle. In *Advances in Computers*, volume 56, pages 1–54. Elsevier, 2002.

[6] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 158–167. IEEE, 2009.

[7] David Binkley. An empirical study of the effect of semantic differences on programmer comprehension. In *Proceedings 10th International Workshop on Program Comprehension*, pages 97–106. IEEE, 2002.

[8] Benjamin S Bloom, Max D Engelhart, Edward J Furst, Walker H Hill, and David R Krathwohl. *Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive domain.* McKay New York, 1956.

[9] Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Conallen, and Kelli A Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983. doi:https://doi.org/10.1016/S0020-7373(83)80031-5.

[11] Fletcher J. Buckley and Robert Poston. Software quality assurance. *IEEE Transactions on Software Engineering*, SE-10(1):36–41, 1984. doi:10.1109/TSE.1984.5010196.

[12] J-M Burkhardt, Françoise Détienne, and Susan Wiedenbeck. The effect of object-oriented programming expertise in several dimensions of comprehension strategies. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 82–89. IEEE, 1998.

[13] Roee Cates, Nadav Yunik, and Dror G Feitelson. Does code structure affect comprehension? on using and naming intermediate variables. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 118–126. IEEE, 2021.

[14] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. doi:10.1109/32.295895.

[15] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 33(2):346–366, 1932. doi:10.2307/1968337.

[16] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, and Bart Van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *2009 IEEE 17th international conference on program comprehension*, pages 100–109. IEEE, 2009.

[17] Cynthia L Corritore and Susan Wiedenbeck. Direction and scope of comprehension-related activities by procedural and object-oriented programmers: An empirical study. In *Proceedings IWPC 2000. 8th International Workshop on Program Comprehension*, pages 139–148. IEEE, 2000.

[18] Andrea De Lucia, Carmine Gravino, Rocco Oliveto, and Genoveffa Tortora. Data model comprehension: an empirical comparison of er and uml class diagrams. In *2008 16th IEEE International Conference on Program Comprehension*, pages 93–102. IEEE, 2008.

[19] Martin Dias, Diego Orellana, Santiago Vidal, Leonel Merino, and Alexandre Bergel. Evaluating a visual approach for understanding javascript source code. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 128–138, 2020.

[20] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension*, pages 277–285, 2018.

[21] Michael E Fagan. Advances in software inspections. In *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*, pages 335–360. Springer, 2001.

[22] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2012.

[23] Scott D Fleming, Eileen Kraemer, RE Kurt Stirewalt, Laura K Dillon, and Shaohua Xie. Refining existing theories of program comprehension during maintenance for concurrent software. In *2008 16th IEEE International Conference on Program Comprehension*, pages 23–32. IEEE, 2008.

[24] Tom Gilb and Dorothy Graham. *Software inspections.* Addison-Wesley Reading, Masachusetts, 1993.

[25] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. The discovery of grounded theory; strategies for qualitative research. *Nursing research*, 17(4):364, 1968.

[26] Felienne Hermans. *The Programmer's Brain: What every programmer needs to know about cognition*. Simon and Schuster, 2021.

[27] Ida Hogganvik and Ketil Stolen. On the comprehension of security risk scenarios. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 115–124. IEEE, 2005.

[28] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[29] ISO/IEC 25010. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.

[30] Bjorn Jacobs and c.l.m. Kop. Functional purity as a code quality metric in multi-paradigm languages. Master's thesis, Radboud University Nijmegen, 2022. URL: https://research.infosupport.com/wp-content/uploads/Master_thesis_bjorn_jacobs_1.6.1.pdf.

[31] Ahmad Jbara and Dror G Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22nd international conference on program comprehension*, pages 189–200, 2014.

[32] Cem Kaner, Jack Falk, and Hung Q Nguyen. *Testing computer software*. John Wiley & Sons, 1999.

[33] Bernhard Katzmarski and Rainer Koschke. Program complexity metrics and programmer opinions. In *2012 20th IEEE international conference on program comprehension (ICPC)*, pages 17–26. IEEE, 2012.

[34] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. Do static type systems improve the maintainability of software systems? an empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162. IEEE, 2012.

[35] Amy J Ko and Bob Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 175–184. IEEE, 2003.

[36] Ludwik Kuzniarz, Miroslaw Staron, and Claes Wohlin. An empirical study on using stereotypes to improve understanding of uml models. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 14–23. IEEE, 2004.

[37] Erik Landkroon. *Code quality evaluation for the multi-paradigm programming language scala*. Master's thesis, Universiteit van Amsterdam, 2017.

[38] Chris Langhout and Maurício Aniche. Atoms of confusion in java. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 25–35. IEEE, 2021.

[39] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and Software*, 7(4):325–339, 1987. URL: https://www.sciencedirect.com/science/article/pii/016412128790032X, doi:https://doi.org/10.1016/0164-1212(87)90032-X.

[40] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

[41] Bryan Marshall, Peter Cardon, Amit Poddar, and Renee Fontenot. Does sample size matter in qualitative research?: A review of qualitative interviews in is research. *Journal of computer information systems*, 54(1):11–22, 2013.

[42] Russell Mosemann and Susan Wiedenbeck. Navigation and comprehension of programs by novice programmers. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 79–88. IEEE, 2001.

[43] John-Jose Nunez and Gregor Kiczales. Understanding registration-based abstractions: A quantitative user study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 93–102. IEEE, 2012.

[44] Michael P O'Brien and Jim Buckley. Inference-based and expectation-based processing in program comprehension. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 71–78. IEEE, 2001.

[45] Michael P O'Brien and Jim Buckley. Modelling the information-seeking behaviour of programmers-an empirical approach. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 125–134. IEEE, 2005.

[46] Pypl, 2022. URL: https://pypl.github.io/PYPL.html.

[47] Dave J Robson, Keith H Bennett, Barry J Cornelius, and Malcolm Munro. Approaches to program comprehension. *Journal of Systems and Software*, 14(2):79–84, 1991.

[48] Tobias Roehm. Two user perspectives in program comprehension: end users and developer users. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 129–139. IEEE, 2015.

[49] Chris Ryder. Software metrics: measuring haskell. *Trends in Functional Programming*, pages 31–46, 2005.

[50] Gayani Samaraweera, Macneil Shonle, and John Quarles. Programming from the reader's perspective: Toward an expectations approach. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 211–212. IEEE, 2011.

[51] G Gordon Schulmeyer. *Handbook of software quality assurance*. Artech House, Inc., 2007.

[52] Amal A Shargabi, Syed Ahmad Aljunid, Muthukkaruppan Annamalai, and Abdullah Mohd Zin. Performing tasks can improve program comprehension mental model of novice developers: An empirical approach. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 263–273, 2020.

[53] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8:219–238, 1979.

[54] Susan Elliott Sim, Charles LA Clarke, and Richard C Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, pages 180–187. IEEE, 1998.

[55] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13, 2020.

[56] Berlin Stefan L. Ram. Dr.alan kay on the meaning of "object-oriented programming", July 2003. URL: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en.

[57] Sven Konings. *Source code metrics for combined functional and Object-Oriented Programming in Scala*. Master's thesis, November 2020. Publisher: University of Twente. URL: http://essay.utwente.nl/85223/.

[58] Riston Tapp and Rick Kazman. Determining the usefulness of colour and fonts in a programming task. In *Proceedings 1994 IEEE 3rd Workshop on Program Comprehension-WPC'94*, pages 154–161. IEEE, 1994.

[59] Héctor Adrián Valdecantos, Katy Tarrit, Mehdi Mirakhorli, and James O Coplien. An empirical study on code comprehension: data context interaction compared to classical object oriented. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 275–285. IEEE, 2017.

[60] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621, 2009.

[61] Giuseppe Visaggio. Relationships between documentation and maintenance activities. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 4–16. IEEE, 1997.

[62] Anneliese Von Mayrhauser and Stephen Lang. On the role of static analysis during software maintenance. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 170–177. IEEE, 1999.

[63] Eliane S Wiese, Anna N Rafferty, and Garrett Moseke. Students' misunderstanding of the order of evaluation in conjoined conditions. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 476–484. IEEE, 2021.

[64] Marvin Wyrich, Lasse Merz, and Daniel Graziotin. Anchoring code understandability evaluations through task descriptions. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 133–140, 2022.

[65] Shaochun Xu and V. Rajlich. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, pages 176–182, 2004. doi:10.1109/COGINF.2004.1327473.

[66] Bart Zuilhof, Rinse van Hees, and Clemens Grelck. Code quality metrics for the functional side of the object-oriented language c#. In *SATToSE*, 2019.

Appendix A

Information Letter

Briefing: Code Comprehension interview

Dear participant,
Thank you for agreeing to participate in my research by participating in an interview. I am Daniël Floor, a second-year master's computer science student at the University of Twente. For my master's thesis at Info Support, I am investigating code comprehension in Kotlin.

Purpose of the Interview:
During the interview, you will be given a series of comprehension tasks in Kotlin. Your goal is to complete the comprehension task. You will be asked to do this using the think-aloud protocol, which means that you will tell all the things you are thinking. This is used for in-depth analysis for comprehension. Beforehand, you will be given the comprehension tasks, you will be asked a couple of general demographic questions. Only relevant information about you is needed, and this does not contain any personally identifiable information. After the tasks have been completed, a small debrief will take place where you can ask some questions regarding the tasks and you may answer some additional questions asked to you. The interview in total should cost around 45-60 minutes of your time.

Voluntary Participation:
Participating in this study is entirely voluntary. If you wish to stop at any point, you are free to do so without providing a reason.
Recordings & consent:
During this session, you will be given a set of tasks to perform, during this time an audio and screen recording will take place. The recorded footage will be used to help analyze the results. All personally identifiable information, will not be used in the analysis. All recorded footage will be destroyed when the research has been completed, which should be in November 2023. At the beginning of the interview, you will be asked to sign a consent letter to ensure compliance with the relevant regulations.

Research purpose:
The goal of this research is to gain more insight into the language Kotlin. Maintaining code is a big part of the software development process of which code comprehension is a major factor that influences the time required for maintenance. The goal of the study is to gain insight into the code comprehension in Kotlin.

Contact Information:
If you have any questions or comments regarding the research, please feel free to contact me via email at d.floor@student.utwente.nl or Daniel.floor@infosupport.com.

Appendix B

Consent form

Consent Form for code comprehension Thesis Daniël Floor

YOU WILL BE GIVEN A COPY OF THIS INFORMED CONSENT FORM

Please tick the appropriate boxes	**Yes**	**No**
Taking part in the study		
I have read and understood the study information dated [17/10/2023], or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.	○	○
I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.	○	○
I understand that taking part in the study involves an audio- and screen-recorded, which will be destroyed after completing the research, which will be around November 2023.	○	○
Use of the information in the study		
I understand that information I provide will be used for a master thesis study regarding code comprehension.	○	○
I understand that personal information collected about me that can identify me, such as video footage will not be shared beyond the study team.	○	○
I agree that my information can be quoted in research outputs.	○	○
I agree to be audio- and screen-recorded.	○	○

Signatures

______________________ ______________________ ________

Name of participant Signature Date

I have accurately read out the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.

_________________________ ___________________ ________

Researcher name Signature Date

Study contact details for further information: [*Daniël Floor, d.floor@student.utwente.nl]*

Contact Information for Questions about Your Rights as a Research Participant

If you have questions about your rights as a research participant, or wish to obtain information, ask questions, or discuss any concerns about this study with someone other than the researcher(s), please contact the Secretary of the Ethics Committee Information & Computer Science: ethicscommittee-CIS@utwente.nl

UNIVERSITY OF TWENTE.

Appendix C

Interview Questions

This Appendix entry contains all object-oriented and multi-paradigm questions defined for the code comprehension interviews.

C.1 Object-oriented questions

C.1.1 Question 1

LISTING C.1: Question 1 OO

```
/**
 * What is the result of the function call on line 17?
 */
fun g(x: List<Int>): Int {
    var y = 0
    for (z in x) {
        if (z + y < 10)
            y += z
        else
            y -= z
    }
    return y
}

fun main() {
    val a = listOf(1, 2, 3, 4, 5)
    println(g(a))
}
```

C.1.2 Question 2

LISTING C.2: Question 2 OO

```
/**
 * What is the functionality of funcA, and what does funcB do?
 */
fun funcA(listA: List<Int>): List<Int> {
```

```
    if (listA.size <= 1) {
        return listA
    }

    val varA = listA.size / 2
    val listB = listA.subList(0, varA)
    val listC = listA.subList(varA, listA.size)

    val listD = funcA(listB)
    val listE = funcA(listC)

    return funcB(listD, listE)
}

private fun funcB(listA: List<Int>, listB: List<Int>): List<Int> {
    var i = 0
    var j = 0
    val listC = mutableListOf<Int>()

    while (i < listA.size && j < listB.size) {
        if (listA[i] < listB[j]) {
            listC.add(listA[i])
            i++
        } else {
            listC.add(listB[j])
            j++
        }
    }

    while (i < listA.size) {
        listC.add(listA[i])
        i++
    }

    while (j < listB.size) {
        listC.add(listB[j])
        j++
    }

    return listC
}
```

C.1.3 Question 3

Listing C.3: Question 3 OO

```
/**
 * What is the output of line 14 and 15
 * and what does funcA and funcB do
```

```
 */
fun funcA(valA: Int): Boolean {
    if(valA<2) return false
    for (i in 2..valA/2) {
        if (valA % i == 0) {
            return false
        }
    }
    return true
}
fun funcB(valA: Int, valB: Int): Int {
    var varA = valA
    var varB = valB
    while (varB != 0) {
        val varC = varB
        varB = varA % varB
        varA = varC
    }
    return varA
}

fun main() {
    println(funcB(12,16))
    println(funcA(15))
}
```

C.1.4 Question 4

Listing C.4: Question 4 OO

```
/**
 * What does this class represent and what is printed on line 16?
 */
class A(private val listA: List<Double>) {
    fun funcA(a: Double): Double {
        var varA = 0.0
        for (i in listA.indices) {
            varA += listA[i] * Math.pow(a, i.toDouble())
        }
        return varA
    }
}

fun main() {
    val varA = A(listOf(3.0, 0.0, -2.0, 1.5))
    println(varA.funcA(2.0))
}
```

C.1.5 Question 5

Listing C.5: Question 5 OO

```
/**
 * you have been given a data class edge and node
 * and within the class graph there is a to be
 * identified function.
 * What does funcA do and what does it return?
 */
data class Edge(val source: Node, val dest: Node, val weight: Int)

data class Node(val name: String)
class Graph(val nodes: List<Node>, val edges: List<Edge>) {
    // this is a list with all neighbours of each Node
    private val adjacencyList: Map<Node, List<Edge>>
        get() {
            val adjacencyMap = mutableMapOf<Node, List<Edge>>()

            for (node in nodes) {
                val adjacentEdges = mutableListOf<Edge>()
                for (edge in edges) {
                    if (edge.source == node) {
                        adjacentEdges.add(edge)
                    }
                }
                adjacencyMap[node] = adjacentEdges
            }

            return adjacencyMap
        }


    fun funA(nodeA: Node, nodeB: Node): Pair<Int, List<Node>>? {
        val mapA = mutableMapOf<Node, Pair<Int, List<Node>>>()
        mapA[nodeA] = 0 to listOf(nodeA)

        while (mapA.isNotEmpty()) {
            val currentNode = mapA.minByOrNull { it.value.first }
            if (currentNode == null || currentNode.key == nodeB) {
                return mapA[nodeB]
            }
            val varA = currentNode.value.first
            val listB = currentNode.value.second
            mapA.remove(currentNode.key)
            val listEdge = adjacencyList[currentNode.key] ?:
                                emptyList()

            for (edge in listEdge) {
                val nodeC = edge.dest
```

```
                    val varB = varA + edge.weight
                    val listC = listB + nodeC


                if (varB < (mapA[nodeC]?.first ?:
                    Int.MAX_VALUE)) {
                        mapA[nodeC] = varB to listC
                    }
                }
            }

            return null
        }
}
```

C.1.6 Question 6

LISTING C.6: Question 6 OO

```
/**
 * This class has a matrix as its variable
 * The result on line 53 is [2,4,8] but this is not entirely
 * correct, why and what is missing?
 */
class C(val valA: List<List<Int>>) {
    fun funcA(): C {
        if (valA.isEmpty()) {
            return C(emptyList())
        }

        val varA = valA.size
        val varB = if (varA > 0) valA[0].size else 0

        val listA = mutableListOf<MutableList<Int>>()
        for (i in 0 until varB) {
            val listB = mutableListOf<Int>()
            for (j in 0 until varA) {
                listB.add(valA[j][i])
            }
            listA.add(listB)
        }

        return C(listA)
    }

    fun funcB(): C {
        val listA = mutableListOf<List<Int>>()
        for (varA in valA) {
            val listB = mutableListOf<Int>()
```

```
            for (varB in varA) {
                if (funcC(varB)) {
                    listB.add(varB)
                }
            }
            listA.add(listB)
        }
        return C(listA)
    }
    private fun funcC(varA: Int) : Boolean {
        return varA % 2 == 0
    }
}

fun main() {
    val varA = listOf(
        listOf(1, 2, 3),
        listOf(4, 5, 6),
        listOf(7, 8, 9)
    )
    val varB = C(varA)
    val result = varB.funcB()
    result.funcA().valA.forEach { a -> println(a) }
}
```

C.1.7 Question 7

Listing C.7: Question 7 OO

```
/**
 * What does funcA do and what is the result of the function
 * call on line 64.
 */
class D {
    fun funcA(a: String, b: String): List<Int> {
        val varA = a.length
        val varB = b.length
        val result = mutableListOf<Int>()

        val varC = funcB(b)

        var i = 0
        var varE = 0

        while (i < varA) {
            if (b[varE] == a[i]) {
                i++
                varE++
            }
```

```
            if (varE == varB) {
                result.add(i - varE)
                varE = varC[varE - 1]
            } else if (i < varA && b[varE] != a[i]) {
                if (varE != 0) {
                    varE = varC[varE - 1]
                } else {
                    i++
                }
            }
        }

        return result
    }

    private fun funcB(a: String): IntArray {
        val varA = a.length
        val result = IntArray(varA)
        var varC = 0
        var varD = 1

        while (varD < varA) {
            if (a[varD] == a[varC]) {
                varC++
                result[varD] = varC
                varD++
            } else {
                if (varC != 0) {
                    varC = result[varC - 1]
                } else {
                    result[varD] = 0
                    varD++
                }
            }
        }

        return result
    }
}

fun main() {
    val varB = D().funcA("ABABDABACDABABCABAB", "ABA")
    if (varB.isNotEmpty()) {
        println(varB)
    } else {
        println("Empty")
    }
}
```

C.2 Multi-paradigm questions

C.2.1 Question 1

LISTING C.8: Question 1 MP

```
/**
 * What is the result of the function call on line 17?
 */
fun f(x: List<Int>): Int {
    var y = 0
    x.forEach { z ->
        if (z + y < 10)
            y += z
        else
            y -= z
    }
    return y
}

fun main() {
    val a = listOf(1, 2, 3, 4, 5)
    println(f(a))
}
```

C.2.2 Question 2

LISTING C.9: Question 2 MP

```
/**
 * What is the functionality of funcA, and what does funcB do?
 */
fun funcA(listA: List<Int>): List<Int> {
    if (listA.size <= 1) {
        return listA
    }
    val varA = listA.size / 2
    val listB = listA.subList(0, varA)
    val listC = listA.subList(varA, listA.size)
    return funcB(funcA(listB), funcA(listC))
}

fun funcB(listA: List<Int>, listB: List<Int>): List<Int> {
    return when {
        listA.isEmpty() -> listB
        listB.isEmpty() -> listA
        listA.first() < listB.first() -> {
            listOf(listA.first()) + funcB(listA.drop(1), listB)
        }
```

```
            else -> {
                listOf(listB.first()) + funcB(listA, listB.drop(1))

            }
      }
}
```

C.2.3 Question 3

LISTING C.10: Question 3 MP

```
/**
 * What is the output of line 14 and 15
 * and what does funcA and funcB do
 */
fun funcA(a: Int): Boolean {
    return (a > 1) && (2 until a).none { a % it == 0 }
}

fun funcB(a: Int, b: Int): Int {
    return if (b == 0) a else funcB(b, a % b)
}

fun main() {
    println(funcB(12,16))
    println(funcA(15))
}

fun main() {
    val a = listOf(1, 2, 3, 4, 5)
    println(g(a))
}
```

C.2.4 Question 4

LISTING C.11: Question 4 MP

```
import kotlin.math.pow

/**
 * What does this class represent and what is printed on line 16?
 */
class A(private val listA: List<Double>) {

    fun funcA(a: Double): Double {
        return listA
            .mapIndexed { i, c -> c * a.pow(i.toDouble()) }
            .sum()
    }
```

```
}

fun main() {
    val varA = A(listOf(3.0, 0.0, -2.0, 1.5))
    println(varA.funcA(2.0))
}
```

C.2.5 Question 5

Listing C.12: Question 5 MP

```
/**
 * you have been given a data class edge and node
 * and within the class graph there are 2 to be
 * identified functions.
 * What does funcA do and what does it return?
 */
data class Edge(val source: Node, val dest: Node, val weight: Int)
data class Node(val name: String)
class Graph(val nodes: List<Node>, val edges: List<Edge>) {
    // this is a list with all neighbours of each Node
    private val adjacencyList: Map<Node, List<Edge>> =
        nodes.associateWith { node ->
        edges.filter { it.source == node }
    }

    fun funcA(nodeA: Node, nodeB: Node): Pair<Int, List<Node>>? {
        val (listA, mapA) = nodes.fold(
            emptyList<Node>() to
                mapOf(nodeA to Pair(0, listOf(nodeA)))
        ) { (listB, mapB), _ ->
            if (listB.isEmpty() || listB.last() != nodeB) {
                val currentNode = mapB
                    .filterKeys { it !in listB }
                    .minByOrNull { it.value.first }

                val pairBnodeB = mapB[nodeB]
                if (pairBnodeB != null &&
                    (currentNode == null ||
                    currentNode.key == nodeB)) {
                    pairBnodeB.second to mapB
                } else {
                    val varA = checkNotNull(
                               mapB[currentNode?.key]?.first)
                    val listC = mapB[currentNode?.key]?.second ?:
                                emptyList()
                  val listD = adjacencyList[currentNode?.key] ?:
                                emptyList()
                  val mapB = funcB(listD, nodeA, mapB, varA, listC)
```

```
                    val listE = if (currentNode?.key != nodeA)
                    listB + currentNode!!.key else
                    listOf(currentNode.key)
                      listE to mapB
                  }
              } else {
                  listB to mapB
              }
          }
          return if (listA.isNotEmpty() && listA.last() == nodeB)
                      mapA[nodeB] else null
      }

      private fun funcB(
          listEdge: List<Edge>,
          nodeA: Node,
          mapA: Map<Node, Pair<Int, List<Node>>>,
          a: Int,
          listNode: List<Node>
      ): Map<Node, Pair<Int, List<Node>>> {
          return listEdge.fold(mapA.toMutableMap()) { mapB, edge ->
              val varA = a + edge.weight

              val (varB, _) = mapB[edge.dest] ?:
                                  Pair(Int.MAX_VALUE, listOf(nodeA))

              if (varA < varB) {
                  mapB[edge.dest] = varA to (listNode + edge.dest)
              }
              mapB
          }
      }

}
```

C.2.6 Question 6

Listing C.13: Question 6 MP

```
/**
 * This class has a matrix as its variable
 * The result on line 53 is [2,4,8] but this is not entirely
 * correct, why and what is missing?
 */
class C(val valA: List<List<Int>>) {
    fun funcA(): C {
        val valB = valA.size
        val valC = if (valB > 0) valA[0].size else 0
```

```
        return C(List(valC) { a ->
            List(valB) { b -> valA[b][a] }
        })
    }

    fun funcB(
        lambA: (List<List<Int>>, (List<Int>)
            -> List<Int>) -> List<List<Int>>,
        lambB: (Int) -> Boolean
    ): C {
        return C(lambA(valA) { a -> funcC(a, lambB) })
    }

    private fun funcC(a: List<Int>,
        lambA: (Int) -> Boolean): List<Int> {
        return a.filter { lambA(it) }
    }
}

fun main() {
    val varA = listOf(
        listOf(1, 2, 3),
        listOf(4, 5, 6),
        listOf(7, 8, 9)
    )
    val varB = C(varA)
    val lambA: (List<List<Int>>, (List<Int>) -> List<Int>)
        -> List<List<Int>> = { a, lambB ->
        a.map { b -> lambB(b) }
    }
    val result = varB.funcB(lambA) { a -> a % 2 == 0 }
    result.funcA().valA.forEach { a -> println(a) }
}
```

C.2.7 Question 7

Listing C.14: Question 7 MP

```
/**
 * What does funcA do and what is the result of the function call
 * on line 52.
 */
class E {
    fun funcA(a: String, b: String): List<Int> {
        val varA = funcB(b)
        return funcD(a, b, 0, 0, varA, mutableListOf())
    }

    private fun funcB(a: String): IntArray {
```

```
        fun funcC(b: Int, c: Int, d: IntArray): IntArray {
            return when {
                b == a.length -> d
                a[b] == a[c] -> {
                    d[b] = c + 1
                    funcC(b + 1, c + 1, d)
                }

                c != 0 -> funcC(b, d[c - 1], d)
                else -> {
                    d[b] = 0
                    funcC(b + 1, 0, d)
                }
            }
        }

        return funcC(1, 0, IntArray(a.length))
    }

    private fun funcD(a: String, b: String, c: Int, d: Int,
    e: IntArray, result: MutableList<Int>): List<Int> {
        return when {
            c == a.length -> result
            a[c] == b[d] -> {
                if (d == b.length - 1) {
                    result.add(c - d)
                    funcD(a, b, c + 1, e[d], e, result)
                } else {
                    funcD(a, b, c + 1, d + 1, e, result)
                }
            }

            d != 0 -> funcD(a, b, c, e[d - 1], e, result)
            else -> funcD(a, b, c + 1, d, e, result)
        }
    }
}

fun main() {
    val varA = E().funcA("ABABDABACDABABCABAB", "ABA")

    if (varA.isNotEmpty()) {
        println(varA)
    } else {
        println("Empty")
    }
}
```