Code Comprehension in a Multi-Paradigm Environment: Background and Experimental Setup

Daniël Floor[1,2], Rinse van Hees[2] and Vadim Zaytsev[1,3]

[1]Computer Science, University of Twente, Enschede, The Netherlands
[2]Info Support, Veenendaal, The Netherlands
[3]Formal Methods & Tools, University of Twente, Enschede, The Netherlands

Abstract

Code comprehension, a fundamental asset in software development and its maintenance, is influenced by the programming paradigms employed. Comprehending code takes up a major part of the maintenance process. This study forms a basis in discovering the relation between code comprehension and multi-paradigm usage. The different paradigms covered here are Object-oriented programming and functional programming. To measure the possible impact an experimental setup is designed that will help capturing quantitative and qualitative data. The decision of using interviews as study experiment allows for the capturing of the qualitative data necessary for an in-depth exploration of comprehension strategies and participants' cognitive reasoning. The interviews will use Kotlin code snippets, this choice harmonises with participants' familiarity with Java, which serves as a foundation, and the design of interview questions, which prioritise the comprehension of code and the unravelling of its underlying purpose. This paper provides the background and experimental setup that allows to investigate the relationship between code comprehension and multi-paradigm usage.

Keywords

code comprehension, multi-paradigm languages, code smells

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 realised 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 [1]. These paradigms in its 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 clearly 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 [2, 3, 4]. The code comprehension side of multi-paradigms is yet to be researched in depth. Code comprehension focuses on the process of understanding the behaviour of the code.

Code comprehension is an important part of ensuring code quality and maintainability. Code comprehension is a process of understanding the behaviour 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 [5]. 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 will give a first insight on the impact of multi-paradigm usage on code comprehension. The focus will lie on object-oriented programming languages that have incorporated functional programming concepts and constructs. In a multi-paradigm perspective, there are two

SATToSE'23: Post-proceedings of the 15th Seminar on Advanced Techniques and Tools for Software Evolution, June 2023, Fisciano, Italy
dan.floor@gmail.com (D. Floor);
Rinse.vanHees@InfoSupport.com (R. v. Hees);
vadim@grammarware.net (V. Zaytsev)
https://grammarware.net/ (V. Zaytsev)
0000-0001-7764-4224 (V. Zaytsev)

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 [1]. The modularity of a language has been described as key to successful programming [6]. With the high usage and similarities of C# and Java, it is interesting to examine the code comprehension of the mixed multi-paradigm usage of these languages. To gain a broader picture and more diverse results, Scala and Kotlin are two other languages that will be examined. Before meaningful conclusions can be drawn on code comprehension, mixed multi-paradigm constructs must be defined. After these constructs are defined, an empirical study will be performed to examine the connection between code comprehension and mixed multi-paradigm usage. In cases of similarities of poor code comprehension, a set of new code smells will be defined. Code smells stem from the 1990s and became popular after a publication by Beck and Fowler in 1999 [7]. A recent study from 2018 has defined and listed a set of previously and newly defined code smells [8]. These smells do still not include code smells that cover multi-paradigm code.

To provide structure to the research, the following **research questions** have been established.

RQ1: Which multi-paradigm constructs can be identified when combining object-oriented programming and functional programming?

This document covers the different concepts and constructs that identify functional programming and object-oriented programming. These constructs already have plenty of metrics and ways to measure their complexity and thus their impact on maintainability. Four multi-paradigm languages have been selected for our analysis: C#, Java, Scala, and Kotlin. In previous research [9], it has already been concluded what support for functional programming constructs these languages have. We want to discover the impact of mixing multi-paradigm code on the code comprehension of practitioners. To achieve this, we must define which multi-paradigm constructs can be identified when OOP and FP code are combined. With the outcome of this question, we can compare the constructs with the other languages and look for similarities but also differences in functionality. These answers will help us construct the answers for the next research question.

RQ2: How can we study the code comprehension of functional programming usage in multi-paradigm languages?

In section 3 we will elaborate on the importance of software quality and the impact of code comprehension on software quality. Yet we have not covered how we could set up a study that will accurately measure the impact of code comprehension on software quality. To answer this question we will go over other code comprehension studies, from the ICPC, and identify how they designed their studies. These papers go through multiple selection phases and end up with the relevant code comprehension studies. When having studied all relevant code comprehension studies we can assess what the best structure will be to study code comprehension of functional programming usage in multi-paradigm languages.

RQ3: Are there similarities in code comprehension of multi-paradigm languages?

With the answers to previous research questions, we can define and build a study that makes it possible to analyse this. The study will use practitioners from the field in any of the four analysed languages, as well as students that are familiar with any of those languages. With the results gained from the study, we can analyse code comprehension. In areas where code comprehension scores poorly, we will look at similar constructs that cause this issue. Additionally, we will also look at possible positive impacts of constructs on code comprehension, if there are any.

RQ4: How can we define code smells that reflect the code comprehension of these languages?

Lastly, we want to turn the analysis of the previous research question into something more conspicuous. We want to achieve this by defining code smells that are introduced in a multi-paradigm environment. We do this by going over other sources that defined code smells and understand what makes a code smell a code smell. With the knowledge gained from this and the analysis of the study we performed, we will be able to define code smells, in case they are present.

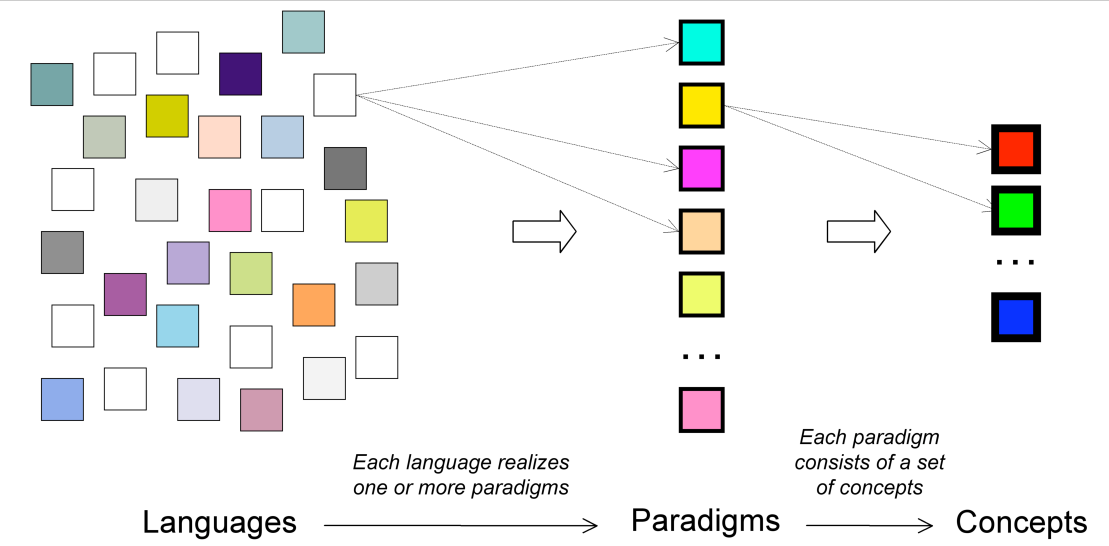

Figure 1: Languages, paradigms & concepts [1]

2. Background

2.1. Programming paradigms

Programming paradigms, e.g. object-oriented programming, can be viewed as a categorisation 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 realised through a programming language. Such a language can in its turn realise more than one paradigm. This creates a hierarchy with endless possibilities. Van Roy's visualisation [1], seen in Figure 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 [1]. This taxonomy can be seen in Figure 2.

A programming language is not restricted to realising only one paradigm and can realise 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#, 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 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 [1]. 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 subsection 2.3.

2.1.1. 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.

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

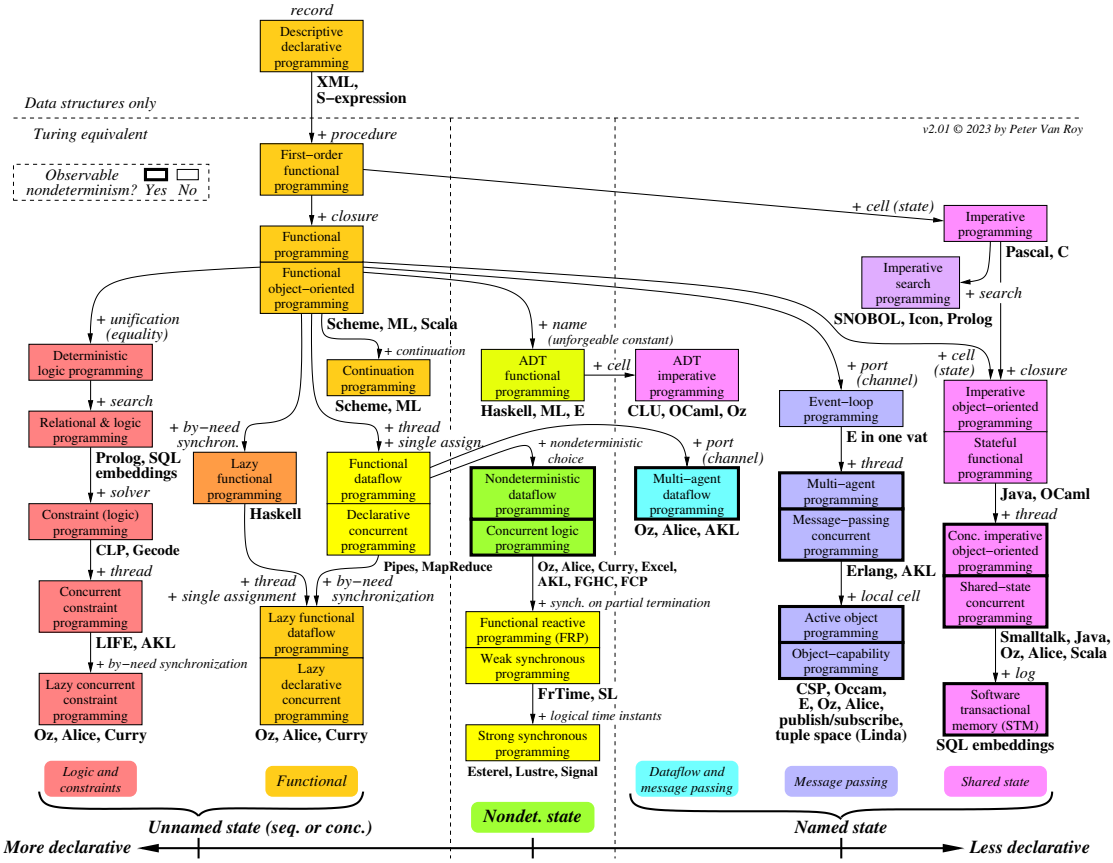

Figure 2: Programming paradigms taxonomy [1]

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), which emphasises the creation of objects that encapsulate data and behaviour. Objects interact with each other by sending messages and invoking methods, which modify the state of the objects.

2.1.2. Procedural programming

Procedural programming is a programming paradigm that revolves around the concept of procedures, which are also known as subroutines or functions. Procedure are small sections of a program that performs 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 popularised the use of procedural programming and is widely regarded as one of the most influential programming languages of all time [10]. In the figure procedural programming is not necessarily listed, but instead, it falls under just the imperative programming block.

2.1.3. 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 as "only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things" [11]. Later and somewhat more formally, OOP was described as "a method of implementation in which programs are organised 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" [12].

It is evident that the view on OOP has shifted through time, but objects will be the centre 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 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.

2.1.4. 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 optimised requiring lots of resources.

2.1.5. 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 [13]. 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 categorised 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 [6]. 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 subsection 2.3. The completely different approach of functional programming complements object-oriented programming enabling different approaches and implementations. More on these differences in subsection 2.3.

2.2. 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 [1]. 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 limited to are: Python, Java, C#, Kotlin, and Scala. We chose to research the languages Java, C#, Kotlin, and Scala. This is due to the fact that 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 a hybrid language in such a way functional programming and object-oriented programming are possible. They do not only support functional programming constructs but it is also able to write code that is completely functional. This contrast would be interesting to see whether the impact on code comprehension differs for Scala + Kotlin and Java + C#.

2.3. 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.

2.3.1. 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 [12]:

- `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()`.
- `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 overrides the implementation of `Animal`.

2.3.2. Functional programming

The origin of functional programming lies in lambda calculus [13]. 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[6]:

- `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 1.

Listing 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 not altering 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 2 a small Haskell program has been given that calculates the sum of all elements in the list.

Listing 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 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: 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.

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

- `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 allows the transformation of 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 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 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
```

- `Pattern matching`: With pattern matching, it is possible to distinguish the behaviour 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 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.

2.3.3. Multi-paradigm constructs

All of the previously described concepts and constructs are clear with singular usage, but the expected behaviour 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 three languages that we will use for our research: C#, Java, and Scala. The current support of functional programming concepts is listed in Table 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 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 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 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 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 three 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. The aim of our research is 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 is able to be analysed using single paradigm metrics [14, 15]. 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.

3. 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 [16]. 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 [17, 18], software testing [19], 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 [20]. While many aspects are covered, we are interested in software quality. The ideology regarding software quality is described in a standard [5]. 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.

3.1. Maintainability

Maintainability is defined as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainer"[5]. So the focus heavily lies on the degree a maintainer is affected by the quality of code. The standard describes five subcharacteristics 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 analyse code it becomes impossible to identify

errors in the code or even reason about its behaviour.
- Modifiability: Modifiability is the degree to which a product can be changed without introducing defects or decreasing the quality of the existing product. In order to modify a program a maintainer must be able to reason about the code and understand its behavior.
- auto: 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 behaviour. 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.

3.2. Code Comprehension

- top-down strategy
- bottom-up item strategy
- knowledge-based strategy
- Systematic/As-Needed Strategy
- Latent Semantic Analysis Strategy

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 behaviour by primarily referencing the source code [21]. 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 recognised as a substantial component of maintenance costs, accounting for a considerable portion ranging from 50% to 90% of these expenses [22] 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.

3.3. 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.

3.3.1. Bottom-up

The bottom-up comprehension strategy, initially proposed by Mayer et al [23], 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 memorising the syntax of the code. As additional layers of comprehension are added, this internal representation is expanded and refined.

3.3.2. 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 [24]. 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.

3.3.3. Knowledge-based

A bottom-up strategy and a top-down strategy are not mutually exclusive and are generally used both while comprehending programs [25]. Letovsky describes a mental model that contains the programmer's knowledge base which represents the current understanding of the code, this model is then actually built using an assimilation process. This process shows how the evolution of the mental model uses both bottom-up and top-down comprehension strategies. The bottom-up and

top-down strategies for program comprehension are not mutually exclusive and are commonly employed together. According to Letovsky, 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.

3.3.4. Systematic

The systematic comprehension strategy is a methodical approach to program comprehension described by Littman et al [26], 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 behaviour of the entire program. This is therefore, a more useful strategy for larger projects and much less for projects with a smaller codebase.

3.3.5. As-needed

The as-needed comprehension strategy, described by Littman et al [26], 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, prioritising 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.

3.4. Code smells and anti-patterns

All projects strive to deliver high-quality software. This high quality is achieved through set design patterns that help structure the code. While the goal is to always write high-quality software, in practice this is not feasible. Bad code gets injected decreasing its quality and the ability for a maintainer to understand it. The decrease originates from bad designs and patterns. These poorly designed patterns are also called anti-patterns and Brown defined 40 different of these anti-patterns [27]. For each of the anti-patterns, its consequences and the solution to resolve the patterns are proposed. Two well-known anti-patterns are:

- God Object: A large class or module that contains too much functionality becomes very complex. These classes are much harder to maintain due to their high complexity and size.
- Spaghetti code: This is the type of code that has its functionality and behaviour tangled into multiple classes without keeping any structure. This is the type of code that is difficult to understand as it is not always clear what its actual functionality is.

Besides the structured anti-patterns, there are a more minor group of indicators of poor code quality. This smaller group is called code smells and was first introduced by Beck & Fowler [7]. They describe these poor code constructs as pieces of code that start to smell due to a higher likeliness of containing faults. Code smells describe certain characteristics and smaller patterns in code that could lead to bugs. A few examples of code smells as defined by Beck & Fowler are:

- Duplicate code: Identical code or almost identical code snippets that appear in multiple places. These pieces could be extracted to a single method to centralise their functionality.
- Long method: These are methods that are large in size. The large size is usually an indication that it carries a lot of functionality and inherently becomes more complex and harder to maintain. It is suggested to refrain from having methods that contain more than one purpose and separate functionality amongst the methods.
- Large parameter list: A method containing a lot of parameters becomes difficult to use and might become inconsistent. Using fewer parameters makes a method more readable and easier to understand.

Where anti-patterns are known bad practices with clear indications on how to solve them, code smells are more subjective and indicators of potential problems. Within just a single paradigm, a large amount of both anti-patterns and code smells are defined and recognised. This is not necessarily the case for the combination of multiple paradigms. There is some research focusing on multi-language anti-patterns and code smells, but as it highlights this is more focused on bad practices concentrated on the communication between two language components of a system [28]. For a multi-paradigm environment within one language, there is still a lot to explore. Therefore, this research aims to identify code smells based on program comprehension. Poor program comprehension can be the cause of the presence of one or more code smells/anti-patterns [29].

4. Experimental Setup

The primary objective of our project 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 customisation 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.

4.1. 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 subsection 3.3. But, it's 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's 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.

4.1.1. 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 there are some measurements that 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 requires 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 in order 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.

4.1.2. Qualitative analysis

Unlike quantitative data, which is expressed in numbers, qualitative data is descriptive in 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, behaviours, 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 behaviour, attitudes, and cultural contexts. When contextualising this within the scope of our research objectives, investigating behavioural 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.

4.2. Literature research

To conduct a proper and representative study on the effects of multi-paradigm programming on code comprehension, it's 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 own study. It's also of the essence to consider the goals and scope of those past studies in our analysis.

4.2.1. 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.

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. 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 53 papers. 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, 14 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.

4.2.2. Paper Categorisation

In order to gain useful information from the relevant 38 selected papers, it is important to establish factors we can use in order to analyse usefulness per study type. We have established a few aspects that are written down and summarised 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 gives an indication of 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?). In order 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 Table 2.

	amount
Small	7
Medium	13
Large	18

Table 2
Distribution study sizes

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 there are multiple studies that try to combine the backgrounds. By considering both it creates a heterogeneous pool of participants. The distribution of this can be found in Table 3.

	amount
Students	18
Industry	10
Hybrid	10

Table 3
Distribution participant demography

Study type While the overarching goal of various studies centres 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 categorised 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' behaviours.

The categorization of the papers can be found in Table 4

	amount
Survey	10
Experiment	16
Observation	12

Table 4
Distribution study types

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 in a physical setting. It is relevant to see what kinds of studies are able to 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 Table 5

	amount
Online	11
Physical	26
Hybrid	1

Table 5
Distribution execution style

4.3. Findings

In investigating the impact of multi-paradigm usage on code comprehension, various study methods come into play, 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 emphasising 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's 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 minimise outside influences, preventing potential confusion caused by unrelated factors and ensuring the clarity of the study's outcomes.

4.4. 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.

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.

4.4.1. 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 individualised 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 analysing each interview. Furthermore, the gathered interview data requires careful processing, in adherence to GDPR regulations.

4.4.2. Survey

Contrarily, surveys chart an entirely distinct trajectory. Emphasising the number of participants over in-depth engagement, surveys aim to glean insights from a broad spectrum of respondents while minimising 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 behaviours 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's essential to underscore the care required in survey setup, encompassing a thorough examination of questions and choices, weighing their respective advantages and disadvantages. Once finalised, 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.

4.5. 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 utilisation [10]. This could potentially be attributed to its steep learning curve. It is pertinent to underscore that within the organisational 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.

4.6. 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 fulfil this requirement, they carry substantial risks of yielding unsatisfactory or insufficient responses, hindering meaningful analyses.

Given the pivotal significance attributed to capturing participants' cognitive behaviour 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's noteworthy that participants may still be requested to complete a brief survey prior to 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 favours 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.

5. Related Work

Landkroon researched the fault prediction of the multi-paradigm language Scala, it was specifically focused lied on the combination of functional programming and object-oriented programming [2]. He combined the OOP metric suite proposed by Chidamber et al. [14] and the FP metric suite for Haskell [15]. He used three popular Scala projects to check whether code quality metrics could predict whether faults labelled in the issue tracker could predict bugs. In his research, he developed a new validation method that extends the validation method from Briand et al. [30]. He showed that his new validation method could outperform Briand's method, especially in projects with a longer life cycle.

Zuilhof adopted the validation method proposed by Landkroon [4]. He developed a metric suite for the language C# that is tailored for functional programming-inspired constructs. This suite was then used to investigate the effectiveness of using the suite to predict error-proneness. The results showed an improvement in projects with active usage of FP-inspired constructs against the baseline model. Konings had a similar approach as Zuilhof but adapted the metric suite to fit the language Scala [3]. These candidate metrics did not show any significant improvement to the baseline model. Additionally, Konings used the programming paradigm score as a single metric to predict fault-proneness, this did not perform well. It was noted that mixed multi-paradigm code had a significantly higher percentage of faults than non-mixed code.

Jacobs defined a purity metric to predict fault-proneness [31]. It was concluded that the purity metric performed slightly better than functional and object-oriented metric suites. Additionally, the purity metric would perform at its best when applying it to methods and training each specific model for each function type.

Arend saw the shortcomings and that previous research kept doing the same, but slightly different [9]. Therefore, he designed a language-agnostic code quality assurance framework (LAMP) for multi-paradigm languages. The goal of this framework is to create one framework on which all metrics can be run. To achieve this, transformations from the source language to the LAMP framework need to happen. Arend has already defined transformation from Java to the framework.

Pankratius et al. performed an empirical study [32] on the impact of functional programming combined with imperative programming on the process of parallel programming. While the imperative code in Java was easier to understand a combination of functional programming and imperative programming in Scala proved to perform the best.

Abbes et al. studied the impact of two well-known anti-patterns and their impact on code comprehension [29]. In their findings, they concluded that the presence of one anti-pattern doesn't impact code comprehension that much, but the presence of multiple anti-patterns decreased code comprehension a lot. These findings will be useful to keep in mind when defining our empirical study. Another study by Khomh et al. found that classes containing code smells are more likely to be fault-prone [33].

Abidi et al. detected that there is a lack of research regarding multi-language systems. They defined 12 code smells for these multi-language systems [28]. These multi-language systems do differ from multi-paradigm languages. The smells reported mostly focus on the interaction between the two(or more) languages of the system. Whereas our aim goes more towards bad practices within one language using multiple paradigms to define their system. In an additional study, the impact of such multi-language design smells has been studied [34]. Their findings were that multi-language design smells have negative impacts on software quality.

6. Conclusion

In the dynamic landscape of programming languages, which has seen the continual evolution of both languages themselves and the complexity of problems they address, a notable shift has occurred. Programming languages now incorporate constructs from multiple paradigms, particularly those of object-oriented and functional programming. This blending of paradigms has introduced a challenge to what was once a clear understanding of these individual constructs. As we strive to comprehend code, a pivotal task in its maintenance, the significance of investigating the potential impact of multi-paradigm usage on code comprehension becomes evident.

To evaluate this influence, a thoughtfully formulated human study assumes a prominent position. Our research journey led us to the formulation of a study framework that encompasses both quantitative and qualitative data acquisition. The implementation of interviews, facilitated by a set of thoughtfully crafted questions, was concluded to be the optimal approach to capture the cognitive processes at play. In these interviews, participants are neither tasked with coding exercises nor programming challenges. Instead, they are assigned with the task of comprehending code snippets written in Kotlin, a language that participants need not be extensively experienced in, rather, a familiarity with navigating a semi-unfamiliar linguistic landscape suffices.

The structure of this study forms a foundational base for delving into how multi-paradigm programming might influence code comprehension. By combining quantitative and qualitative data, we open up a window into the complex web of cognitive behaviours entangled with the understanding of multi-paradigm code. The implications drawn from incorporating these insights together hold immense significance. They not only refine our grasp of comprehending multi-paradigm code but also set the course for future work. This study acts as a guiding torch, showing the way for more investigations into untangling the complexities of multi-paradigm code. In doing so, it adds to our understanding of programming and the ever-changing world of software development.

References

[1] P. Van Roy, et al., Programming paradigms for dummies: What every programmer should know, New computational paradigms for computer music 104 (2009) 616–621.

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

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

[4] B. Zuilhof, R. van Hees, C. Grelck, Code quality metrics for the functional side of the object-oriented language c#, in: A. Etien (Ed.), Post-proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019), volume 2510 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2019, pp. 99–108. URL: https://ceur-ws.org/Vol-2510/sattose2019_paper_12.pdf.

[5] ISO/IEC 25010, ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, Technical Report, ISO, 2011.

[6] J. Hughes, Why functional programming matters, The computer journal 32 (1989) 98–107.

[7] K. Beck, M. Fowler, Bad smells in code, Refactoring: Improving the design of existing code 1 (1999) 75–88.

[8] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158–173.

[9] M. van der Arend, The LAMP Framework — A Language-Agnostic Code Quality Assurance Framework for Multi-Paradigm Languages, Master's thesis, Universiteit Twente, Enschede, The Netherlands, 2023. URL: http://purl.utwente.nl/essays/94619.

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

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

[12] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, K. A. Houston, Object-oriented analysis and design with applications, ACM SIGSOFT software engineering notes 33 (2008) 29–29.

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

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

[15] C. Ryder, Software metrics: measuring haskell., Trends in Functional Programming (2005) 31–46.

[16] F. J. Buckley, R. Poston, Software quality assurance, IEEE Transactions on Software Engineering SE-10 (1984) 36–41. doi:10.1109/TSE.1984.5010196.

[17] T. Gilb, D. Graham, Software inspections, Addison-Wesley Reading, Masachusetts, 1993.

[18] M. 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, Springer, 2001, pp. 335–360.

[19] C. Kaner, J. Falk, H. Q. Nguyen, Testing computer software, John Wiley & Sons, 1999.

[20] G. G. Schulmeyer, Handbook of software quality assurance, Artech House, Inc., 2007.

[21] K. H. Bennett, V. T. Rajlich, N. Wilde, Software evolution and the staged model of the software lifecycle, in: Advances in Computers, volume 56, Elsevier, 2002, pp. 1–54.

[22] D. J. Robson, K. H. Bennett, B. J. Cornelius, M. Munro, Approaches to program comprehension, Journal of Systems and Software 14 (1991) 79–84.

[23] B. Shneiderman, R. Mayer, Syntactic/semantic interactions in programmer behavior: A model and experimental results, International Journal of Computer & Information Sciences 8 (1979) 219–238.

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

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

[26] D. C. Littman, J. Pinto, S. Letovsky, E. Soloway, Mental models and software maintenance, Journal of Systems and Software 7 (1987) 341–355.

[27] W. H. Brown, R. C. Malveau, H. W. S. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[28] M. Abidi, M. Grichi, F. Khomh, Y.-G. Guéhéneuc, Code smells for multi-language systems, in: Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLop '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–13. URL: https://doi.org/10.1145/3361149.3361161. doi:10.1145/3361149.3361161.

[29] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 181–190. doi:10.1109/CSMR.2011.24.

[30] L. Briand, W. Melo, J. Wust, Assessing the applicability of fault-proneness models across object-oriented software projects, IEEE Transactions on Software Engineering 28 (2002) 706–720. doi:10.1109/TSE.2002.1019484.

[31] B. Jacobs, 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.

[32] V. Pankratius, F. Schmidt, G. Garretón, Combining functional and imperative programming for multi-core software: An empirical study evaluating Scala and Java, in: 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 123–133.

doi:10.1109/ICSE.2012.6227200, iSSN: 1558-1225.

[33] F. Khomh, M. Di Penta, Y.-G. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conference on Reverse Engineering, 2009, pp. 75–84. doi:10.1109/WCRE.2009.28.

[34] M. Abidi, M. Openja, F. Khomh, Multi-language design smells: A backstage perspective, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 615–618. URL: https://doi.org/10.1145/3379597.3387508. doi:10.1145/3379597.3387508.