Functional and Imperative Object-Oriented Programming in Theory and Practice

A Study of Online Discussions in the Programming Community

Per Jernlund & Martin Stenberg
Abstract

Functional programming (FP) has progressively become more prevalent and techniques from the FP paradigm has been implemented in many different Imperative object-oriented programming (OOP) languages. However, there is no indication that OOP is going out of style. Nevertheless the increased popularity in FP has sparked new discussions across the Internet between the FP and OOP communities regarding a multitude of related aspects. These discussions could provide insights into the questions and challenges faced by programmers today. This thesis investigates these online discussions in a small and contemporary scale in order to identify the most discussed aspect of FP and OOP. Once identified the statements and claims made by various discussion participants were selected and compared to literature relating to the aspects and the theory behind the paradigms in order to determine whether there was any discrepancies between practitioners and theory. It was done in order to investigate whether the practitioners had different ideas in the form of best practices that could influence theories. The most discussed aspect within FP and OOP was immutability and state relating primarily to the aspects of concurrency and performance. This thesis presents a selection of representative quotes that illustrate the different points of view held by groups in the community and then addresses those claims by investigating what is said in literature. It was shown that there were no direct discrepancies between the practitioners and the theory.

Keywords

Programming paradigms, Object oriented programming, Functional programming, Online discussions, Theory and Practice, Concurrency, Mutability, Immutability, Performance.
Index

Abstract 1
Keywords 1
Index 2

1 Introduction 4
  1.1 Background 4
  1.2 Previous research 5
    1.2.1 Object-Oriented and Functional programming 5
    1.2.2 Comparing programming paradigms 6
    1.2.3 Theory and Practice 6
    1.2.4 Analysis of Topics and Trends in Online Communities 7
  1.3 Problem domain 8
  1.4 Research question 9
  1.5 Delimitation 9

2 Method 10
  2.1 Research Strategy 10
  2.2 Research Approach 11
  2.3 Internet Research 12
    2.3.1 Online vs. offline existence and demographics 12
    2.3.2 Observation and participation 13
    2.3.3 Anonymity 13
    2.3.4 Privacy 14
    2.3.5 Scope 14
    2.3.6 Representativeness 15
  2.4 Method for collection of discussion data 15
    2.4.1 Selected platforms 15
    2.4.2 Sample Size 16
    2.4.3 Selection Criteria for Posts and Comments 17
    2.4.4 Methods for categorizing discussion aspects 17

3 The Paradigms of Functional and Object-Oriented Programming 19
  3.1 Functional Programming 19
    3.1.1 Declarative Programming 19
    3.1.2 Background 19
    3.1.3 Immutability and Side Effects 19
    3.1.4 Higher Order Functions 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.5 Pure Functions</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Object-Oriented Programming</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1 Imperative Programming</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2 Procedural roots</td>
<td>21</td>
</tr>
<tr>
<td>3.2.3 Object-Oriented programming style</td>
<td>22</td>
</tr>
<tr>
<td>3.2.4 The concept of objects</td>
<td>22</td>
</tr>
<tr>
<td>3.2.5 Access modifiers</td>
<td>22</td>
</tr>
<tr>
<td>3.2.6 Encapsulation and Data hiding</td>
<td>22</td>
</tr>
<tr>
<td><strong>4. Theory</strong></td>
<td><strong>23</strong></td>
</tr>
<tr>
<td>4.1 Concurrency</td>
<td>23</td>
</tr>
<tr>
<td>4.2 State</td>
<td>24</td>
</tr>
<tr>
<td>4.2.1 Mutable state</td>
<td>24</td>
</tr>
<tr>
<td>Concurrency with Mutable State</td>
<td>24</td>
</tr>
<tr>
<td>Performance with Mutable State</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2 Immutable State</td>
<td>25</td>
</tr>
<tr>
<td>Concurrency with Immutable State</td>
<td>25</td>
</tr>
<tr>
<td>Performance with Immutable State</td>
<td>26</td>
</tr>
<tr>
<td><strong>5. Analysis and results</strong></td>
<td><strong>27</strong></td>
</tr>
<tr>
<td>5.1 Mutable state</td>
<td>30</td>
</tr>
<tr>
<td>5.2 Immutable state</td>
<td>34</td>
</tr>
<tr>
<td><strong>6. Conclusion</strong></td>
<td><strong>35</strong></td>
</tr>
<tr>
<td><strong>7. Discussion</strong></td>
<td><strong>35</strong></td>
</tr>
<tr>
<td><strong>8. References</strong></td>
<td><strong>38</strong></td>
</tr>
</tbody>
</table>
1 Introduction

In this section the subject background of this thesis is presented, followed by previous research which gives a basis for the problem domain and the research questions. Lastly, the delimitations for this study are presented.

1.1 Background

In 1989, Hughes wrote the article “Why functional programming matters” which became one of the most widely cited works within the domain of functional programming (Hu, Hughes & Wang 2015). Hu et al. (2015) wrote a follow-up article called “How functional programming mattered” that further described the development of the functional programming paradigm. Hu et al. (2015, p 349, 367) claim that FP no longer can be considered a niche and that it has had a major impact in the programming field during the past 25 years. New technologies and issues have arisen and therefore it is possible that FP could have taken a new direction.

Functional programming (FP) defines programs in terms of functions. Functional programs consist of a main function which itself is made up of several smaller functions that are evaluated when the program is executed. There is no way of assigning values to variables, variables are immutable after they have been defined. Furthermore there are no side-effects in strict FP, this means that when a function is evaluated it has no effect whatsoever on the state of other parts of the program. This in turn means that functionally written programs does not need to take the flow control into consideration, functions can be evaluated in any order. This prevents many types of errors. (Hughes 1989, p 98)

Object-oriented programming is designed so that it mimics the way people handle concepts. This means that each concept has a unique name and a unique content. In OOP this is implemented as classes which fills the role as blueprints from which objects can be instantiated. Objects consists of attributes which describes the object's constitution and methods which are the actions that the object can perform. The structure of the objects are identical but its attributes can be assigned different values, i.e. an object of the class car can be instantiated with different colors or brands. Not only that these objects can even be instantiated with identical values and still exist as separate objects in the same manner identical twins are similar but different persons. (Kindler & Krivy 2011, ss 314–315)

Today, object-oriented languages are the most used languages according to GitHub’s annual Octoverse report (GitHub 2018). However, it should be taken into account that many languages implements concepts from different paradigms. A concrete example of that is the introduction of lambda expressions in C# version 3.0 (Hejlsberg & Torgersen 2007). Today we see many giants within IT like Facebook, Whatsapp and LinkedIn using functional languages and concepts to different extent (Hu, Hughes & Wang 2015, s 349). Even Google has reworked and implemented a functional concept, MapReduce, with great success. It is
being used among other things for production of data for Google’s search service, sorting, data mining, machine learning and for a number of other systems (Dean & Ghemawat 2008). There’s a large likelihood that developers and businesses are not going to abandon object-oriented principles but there are indications that a change is occurring.

1.2 Previous research

This section presents previous research which acts as background for the problem domain. The first paper presented describes the history and benefits of the functional and object-oriented programming paradigms. The next paper deals with previous research regarding comparisons of programming paradigms which shows that this is a valid research approach. The following paper deals with the relationship between theory and practice which is relevant for the research question of this thesis. While the last paper is about a previous study that analyzed topics that were being discussed among programmers on Stack Overflow.

1.2.1 Object-Oriented and Functional programming

Dan Clark (2011) published a book on the fundamentals of C# in which he aptly describes object-oriented programming, its history, why it became so widespread and the many concepts often associated with OOP. The first time that object-oriented concepts came to be was during the 1960s in a language called Simula which was used for computer simulations (ibid.). It grew and developed into various branches with languages like Smalltalk and C++ but it was not until the 1990s that it became mainstream due to Java, .Net and C# (ibid.). In 1989 John Hughes published an article called Why Functional Programming Matters that became highly cited due to his arguments regarding the benefits that functional programming can provide (Hughes 1989). 25 years later, Hughes together with Hu and Wang reviewed the impact of functional programming in the article How Functional Programming Mattered (Hu, Hughes & Wang 2015). A number of examples are presented that shows increased adoption of the functional style within industry and thereby illustrating the relevance of functional programming in the present-day software development industry (ibid.). The article goes to great lengths to distinguish functional programming from other paradigms and what makes functional programming a valuable tool for programmers, illustrated by code examples written in Haskell (ibid.). The authors then go on to describe the influence that functional programming has had on education, industry and other programming languages from various paradigms (ibid.). Many Universities such as MIT, Oxford and Cambridge all teach functional programming first as it lets students focus on more fundamental programming issues and think more abstractly (ibid.). Imperative object-oriented programming languages such as C++ and Java both have some support for functional features (A Dictionary of Computer Science, 2016; Clark 2011; Hu, Hughes & Wang 2015). Giants in the social media industry and the financial sector have all adopted functional programming for various purposes (Hu, Hughes & Wang 2015). In summary, the article clearly shows that functional programming is no longer limited to academic use and that it has had a significant effect in society (ibid.).
1.2.2 Comparing programming paradigms

In 1996 Harrison, Samaraweera, Dobie and Lewis published an article called *Comparing programming paradigms: an evaluation of functional and object-oriented programs*. The article investigates whether there is a significant difference in code quality for twelve algorithms written in both a functional and an object-oriented language. The functional language selected was SML and the object-oriented language was C++. The research method is quantitative and in order to measure code for potential differences they developed two sets of metrics. The application domain that was selected was image analysis and it was chosen because the languages would be faced with a reasonably complex and wide selection of tasks involving varied data types, file handling, I/O and user-interfaces. Although it was not an expressed goal in this article, the authors did record some performance metrics which was presented. It showed that SML compilation times were greater than the C++ compilation times. The conclusion showed that there were a few significant differences but also some interesting similarities. SML code took longer to test and had more bugs per thousand lines of uncommented source code. Nearly four times the number of library function calls were made in the SML code, which indicated a higher rate of code reuse. No significant difference was found regarding the number of known errors, modification requests during development and the time taken to implement those changes. At the end of the project, the developers were prompted to submit an assessment of their experience related to programming in the two languages. SML was considered useful for prototyping and designing image processing algorithms but C++ was considered a more pragmatic choice when actually implementing them. In summary, the subjective complexity measurement provided by the developers implied that personal preference of developers may be a deciding factor when choosing between languages. (Harrison, Samaraweera, Dobie & Lewis 1996)

1.2.3 Theory and Practice

Robert Glass (1996) wrote an article called *The Relationship Between Theory and Practice in Software Engineering*. In this article, Glass argues that it is not a given that theory precedes practice but that it is in fact many times the other way around.

Glass gives famous examples of when practice preceded theory in other fields: such as when the airfoil was invented and gave rise to the field of aerodynamics and how the invention of the steam engine preceded the field of thermodynamics. While regarding computer science, time-sharing systems were created and sold before there was any established theory about them. But Glass does mention that it most often is a matter of a combination between theory and practice. That they go hand in hand and incite each other’s development. Computers and their development was in the beginning limited to only take place in academic laboratories, however this eventually enabled others than those in academia to create software which in turn influenced the creation of new areas of research. With that Glass concludes that it is unwise to believe that either theory or practice necessarily precedes the other. (Glass, 1996)
“The importance is profound because there are times when theory has answers to questions practice has not yet asked, but there are also times when practice has answers to questions theory has not yet raised...”
- R. Glass

Glass means that “best-of-practice” is a concept that is a good example of practice leading theory. But that many people still have a strong intuition of theory always leading practice and how it is an assumption that can be very hard to change. To address this Glass mentions a few areas within software development where he believes practice has come further and should be held above the theory in those areas. The areas are: software design, software maintenance, user interface, programming in the large (software of large scale), modeling and simulation and lastly metrics. The reasons Glass gives to why practice has come further than theory varies from the industry having larger budgets to the different interests and incentives of industry and academia. Glass concludes that if he ever wanted to learn about any of these areas as a programmer during the 90s, he would undoubtedly look to the practice. (Glass, 1996)

1.2.4 Analysis of Topics and Trends in Online Communities

Barua, Thomas and Hassan (2012) produced a paper called What are developers talking about? An analysis of topics and trends in Stack Overflow. Stack Overflow is a Q&A website that is focused on programming and software development. In this paper it is stated that as Stack Overflow has accumulated millions of posts consisting of questions and answers it has become a repository with a lot of valuable information. As an analysis of the posts can answer such things as what is trending in the developer community and explain how certain technologies are used. The study is quantitative and a very large set of data is analyzed. To analyze all the posts a modelling technique called latent Dirichlet allocation (LDA) is used. The LDA technique is used to automatically find what is discussed in the Q&A posts. The topics were then analyzed to find out how they relate to each other and to discover trends in the online discussions. Among other things the study found out how certain discussed topics had a tendency to start discussions in certain other topics. Furthermore it was discovered how discussions regarding some topics such as web development and MySQL had been increasing over time. (Barua, Thomas & Hassan 2014)
1.3 Problem domain

As FP has made its way into the mainstream of programming it has become a real alternative to OOP but rather than one paradigm replacing the other they offer alternative ways to solve the same problems. While one paradigm might be better suited for some problems there might be other problems where there is much more a matter of preference. If the paradigms are better at solving different things it means that they can complement each other, they are better off together than alone. But whenever there are alternatives to do things we humans tend to hold different opinions which often leads to discussion. Luckily as of today it has become easier to observe discussions, it is no longer necessary to attend gatherings with the advent of the information age and the Internet. According to previous research, there is valuable information that can be gained from studying online discourse (Barua, Thomas & Hassan 2014). Modern websites such as Stack Overflow and Quora has provided people with the ability to vote on the answers that they consider to be the best and most accurate. Thereby, making it much easier for someone who is seeking out information to identify quality answers in a discussion. A voting system is of course not a guarantee for correct answers, while many of the upvoted answers may be correct it is not clear to what extent. Therefore it would be of interest to investigate to what extent up-voted answers relates to established theory. But as stated by Glass (1996) sometimes practice precedes theory, therefore it would also be of interest to look into those answers that contradicts the theory, as they might provide valuable insights not yet realized. The potential information that can be gained from comparing statements made by practitioners and established theory may result in new and interesting findings relating to best practices in software development.
1.4 Research question

This thesis will focus on studying the discussions relating to FP and OOP. As both FP and OOP are well established programming paradigms they have a lot of different programming aspects that could be taken into account. For clarity, an aspect in the context of this study is an area within the subject domain of computer programming, that is being discussed online. Furthermore, some aspects are in themselves very broad and would naturally be intertwined and related with other aspects within the same discussions and should therefore also be taken into account. The study’s focal point will be to explore the relationship between the practitioners and the theory relating to a discussed aspect. Thus it is not of primary concern of what exactly is being discussed. The aspect is therefore chosen based on how much it is being discussed relative to other aspects to ensure an abundance of data and to analyze an aspect that is interesting to a broader audience. Furthermore, if something is being frequently discussed, there is a higher likelihood of there being a varied set of opinions that can yield interesting results when analysed.

What FP and OOP programming aspect with its related aspects is the most discussed in the online programmer community and are there any discrepancies between the claims made therein and what is presented as facts in literature?

1.5 Delimitation

When analysing the empirical material. The focal point will be on general concepts associated with the FP and OOP paradigm. Because of this, mentions of algorithms or language specific constructs will not be taken into account because they are often too specific or contextual for it to provide any real value in the analysis and its results.
2 Method

This section describes the methodology that was used in order to produce this thesis. It begins with a description of the general research strategy. It is followed by an explanation of the research approach which is a general description of how the research was conducted in practice. Specific methods that are mentioned there are described in more detail later in the method chapter. The research approach is followed by a description of the internet research method followed by detailed descriptions of the methods used for selecting and coding data.

2.1 Research Strategy

This study was conducted as an internet research study. The reason for conducting research in the manner of an internet research study is because the relevant aspects are studied with a focus on width rather than depth. Instead of trying to exhaustively capture every single aspect of the discussion, the focal point was rather to try and identify the aspects with the highest rate of recurrence within several discussions. The study is based on the practical use of the paradigms instead of their use in an artificial environment. Information was collected from several sources. The empirical data comes from a selection of online forums where programming and the two paradigms are discussed by users who are knowledgeable and involved in programming to varying degrees. We chose the discussions as a starting point since knowledge of what aspects are frequently discussed was a prerequisite for knowing where to direct attention when comparing the discussion to the theory.

The study is both qualitative and quantitative since the purpose is to both determine the frequency of discussed aspects and to understand and compare the discussed aspects to the theory (Oates 2006, pp 245, 266-267).

The comparisons between the literature and the discussion has been done using predefined categories relating to the different concepts of each programing paradigm (Goldkuhl, s 30). Unlike that of a new research area where many phenomena are potentially undiscovered and therefore uncategorized.
2.2 Research Approach

The method for this thesis was established based on previous research on methodologies for gathering research material from internet forums. The previous research on methods was mainly found by searching through Google Scholar with terms such as “Internet forum research”. This led to the discovery of an article called Analyzing Internet Forums: A Practical Guide (Holtz, Kronberger & Wagner 2012). The discussion material from the internet forums that was gathered during the internet research phase was coded according to methodologies presented in The Coding Manual for Qualitative Researchers (Saldaña 2013). These methodologies were complemented by Researching Information Systems and Computing (Oates 2006) provided through the exam course.

The online forums that were decided to be included in the study were selected based on the criteria of forum topic and size. The websites that were included are: Stackoverflow, Stackexchange, Reddit and Quora. To find the posts of interest within the websites both Google and the local search functions within the websites were used. Search terms such as “Functional programming vs object-oriented programming Stack Overflow” were used. Each term was searched for four times by changing the website name each time. The forum posts that were selected for analysis were based on a number of selection criteria described further down in the methodology chapter.

All of the thirty selected posts were put into a Google Drive spreadsheet. MAXQDA\(^1\) is the software that has been used to code all material. First of all, the forum posts were collected by using the webcollector browser plugin provided by MAXQDA which made it possible to scrape the selected posts and import them into the MAXQDA software (MAXQDA 2016). The data was then coded in MAXQDA using the descriptive coding method. It is a method that codes nouns and memos to identify topics in a text (Saldaña 2013, pp 88-91). It was used by coding segments in the text for what was identified as being part of a certain aspect within programming. Each time a new aspect was found within the segments it was extracted and placed as a column in a separate spreadsheet. Thereafter the most discussed aspect was identified by applying magnitude coding. A method for coding data by different magnitudes (Saldaña 2013, pp 72-77). The identified aspects where categorized with either 1 or 2, where 1 represented that the aspect was discussed by the participants and that they agreed. While 2 also represented that the aspect was discussed but at least two of the participants disagreed. Each forum post (row) could only receive a single number per aspect (column) which meant that several mentions or discussions still resulted in a single mark of 1 or 2 per aspect and post. The final selection for a programming aspect was based on how high the frequency of occurrence was for a given aspect by evaluating the total score of 1’s and 2’s. By summing up the total value for each column a frequency of occurrence could be discerned and by summing up the 2’s in each column it could also be discerned which aspects generated the

\(^{1}\) *MAXQDA is a qualitative research framework, Version: MAXQDA Analytics Pro 2018 (Release 18.2.0)*
most contention. This frequency data answered the first part of the research question what FP and OOP programming aspect with its related aspects is the most discussed in the online programmer community? and laid the foundation for the second part of the research question. Once the aspect with the highest frequency had been identified it became possible to look for patterns within the comments of the post that discuss the aspect. This was done by adapting the pattern coding method. A coding method that is used to find similarities in different data to make it into manageable units (Saldaña, 2013, pp 209-213). The method was applied to all the segments pertaining to the aspect by looking for phrases or words that the segment revolves around. These words and phrases were thereafter synthesized to form patterns. This was done in two steps, first the segments were categorized by attitude towards the aspect and thereafter coded for what programming aspects it focused on. An example of one such pattern could be if a user claims that mutable state within a program was a bad trait for concurrency but necessary for other tasks then this user would be categorized as being neutral. When the different attitudes towards mutability were identified the quotes were read and coded with words that later could be synthesized into a relating aspect. An example of that would be how different quotes could talk about the same aspect with different words such as concurrency, parallelization and multi-threading, that later could be categorized as part of the same.

The comments of the selected aspects that were discussed were set against each other by comparing the different arguments that were made for each paradigm. The more elaborate comments that summarized what many others thought were chosen to be included in the thesis under analysis and results for the purpose of illustrating the actual discussion to the reader. When both the aspects and arguments that were made were identified the search for theories to independently address these discussions was started. To do this both theory about the specific paradigms and the aspects being discussed had to be identified. The theory was found in both scientific articles from Google Scholar and in popular well cited programming text books. The text books were found through recommendations and through searching Amazon.com for books with good reviews for a given term such as Functional Programming. To ensure credibility the publisher and authors were looked up and the title was searched for in Google Scholar to confirm that they had citations from scientists.

2.3 Internet Research

Internet Research (IR) is conducted in this thesis in order to gather an adequate set of data regarding the discussion about the two programming paradigms FP and OOP. According to Holtz, Kronberger & Wagner (2012, p 3) internet forums contain an abundance of data which is already in digital form and publicly accessible to anyone with an internet connection. Therefore it is suitable for the purposes of this thesis.

2.3.1 Online vs. offline existence and demographics

According to Oates (2006, p 49), Internet research is good for studying an online community however it is limited in the regard that the members offline activities can not be studied. This
problem is further compounded by the fact that many users identify themselves through fictitious usernames and rarely publish personally identifiable information (Holtz et al. 2012, pp 4-5). The problem with lacking information about the demographics of a user-base can be mitigated by looking at the forum guidelines, mission statements or introductory pages as well as statements made by the moderators detailing the purpose of the forum (Ibid.). This can allow the researcher to make qualified guesses as to the general demographics of a user base (Ibid.).

Regardless, it does impose a limitation on the study’s empirical material in that it is not possible to actually observe the programmers and their actual coding but the lack of demographic data is not a big problem provided one is aware of the risks. In order to handle this problem when collecting data, careful attention was paid to what topics and types of content that were allowed on the forum and what the stated intended purpose of the forum was. This allowed for a more focused scope when collecting data and reduced the amount of irrelevant material found when searching for discussion data.

### 2.3.2 Observation and participation

When online communities are being studied it is possible to either partake in their discussions or to only observe them (Oates 2006, p 50). Comparisons have been made that equates forums to virtual focus groups whose interaction can be studied without researchers interjecting and affecting the discussion. This in turn creates relatively authentic, natural data (Holtz et al. 2012, p 4). It has not been in this thesis interest to affect the online discussion or to in any way influence the outcome of whatever aspect that is discussed. Therefore this thesis has only gathered discussion material strictly by observation governed by defined selection criteria without any form of interactions with any community members. Note that observation in this context does not mean that people partaking in online discussions were observed in person, what it means is that the discussions on the online forums were observed.

### 2.3.3 Anonymity

If one compares Online discussions to face-to-face interactions, the perceived anonymity on the internet promotes a higher degree of openness since there is less fear of social repercussions when expressing an unpopular opinion (Holtz et al. 2012, pp 4-5). The drawback is that anonymity can lead to a higher level of aggression in the discourse (Ibid.). In contrast to this, Oates (2006, p 239) claims that anonymity leads to people stating opinions that they do not actually believe in and that all internet documents (such as posts on a forum) is produced for a particular audience and with some ulterior purpose. Both these statements regarding anonymity are valid. For this thesis, anonymity is considered beneficial since it promotes openness and the topics discussed on these forums are of a practical nature and therefore it is rarely a question of sensitive topics for example relating to politics, race, sexuality or religion.
2.3.4 Privacy

Privacy is also a concern on the internet, some users may not consent to having their comments harvested and used, even for research purposes (Holtz, Kronberger & Wagner 2012, pp 5-6). However, The general consensus within the academic community seems to be that there are forums that are meant to be private or semi-private such as support groups for abuse victims or recovering addicts and there are those that are meant for more general discussions of less sensitive nature (Ibid.). Regardless, It is recommended that all data gathered for research purposes is sufficiently anonymised in order to protect the users privacy (Ibid.). This can be done by not publishing usernames and avoiding the use of verbatim quotes that can be found using search engines (Ibid.). Oates (2006, p 50) also lifts the ethical dilemma of watching peoples’ online activity without them knowing, stating that it may anger users if they were to find out that their interactions have been covertly observed by someone who infiltrated their group.

However, the forums selected for this thesis are not of a sensitive nature as described by Holtz nor are they protected by any form of access control mechanism, such as mandatory login. The texts submitted by users are completely accessible by anyone regardless if they have a registered account or not. That would imply that the users are aware of the fact that anyone on the internet can read their posts. In order to not compromise the accessibility and transparency of the research and data for this thesis the usernames will not be included but verbatim quotes are. This does not completely anonymize the users as it is possible to find their usernames by entering the respective quotes into a search engine. However, by not providing their usernames it is possible for the owner of the quote to become completely anonymous by deleting their online post. This would not be possible if their usernames were provided. This is deemed acceptable as these forums are public and the topics are not sensitive.

2.3.5 Scope

Furthermore, Oates mentions the problem with IR regarding how it is all interlinked and how that creates a need to put a limit on how far one should go. Should a link only be followed when it is directly recommended or should they all be disregarded? There is no perfect answer to this, therefore it is something the researcher must decide taking into account the nature of the research. (Oates 2006, p 149)

This thesis will only follow links to a maximum depth of one hop from the actual discussion forum i.e. someone links an article or blog post. However, it was only taken into account when it was needed in order to clarify a user’s statement or when it was the focal point of the discussion. No further links has been followed as that would have resulted in too much data. Links to videos in comments have been ignored as they require too much time to analyse. To conduct an analysis of the discourse between developers on a video blogging platform such
as YouTube would be outside the scope of this thesis. However it could be an interesting area for future research since there are plenty of videos that are directed towards developers.

2.3.6 Representativeness

Regarding the representativeness of the data. One can not always assume that an online discourse is representative to a whole demographic group since not everybody has access to the internet and those that do may or may not participate in online discussions. (Holtz, Kronberger & Wagner 2012). For this thesis it is a safe assumption to make that a very large majority of developers have some form of access to the internet since it is an invaluable tool in their daily work. However, it must be said that not all of them partake in online discussions and therefore it is hard to make any accurate claim to the representativeness of the data. There is information presented regarding the number of users that are active on the various platforms under section 2.4 which shows that there is a very large population of programmers present there to show the representativeness of the selected platforms.

2.4 Method for collection of discussion data

This section describes the selected platforms with some information related to their size and general purpose. This is followed by sample size and all criteria that was adhered to when selecting forums, posts and comments for collection and analysis. Finally, there is a description of the method used for categorizing the collected posts.

2.4.1 Selected platforms

The platforms that have been selected share an important feature. In each post the answers that is considered the most correct or relevant by the largest number of users gets pushed to the top or bottom through user voting. This kind of system is both important and useful. It is important since it is well known that the majority of registered users on social media platforms rarely contribute by commenting. However, the option to promote good content without having to make the effort of writing a message mitigates this problem. This makes it easier for users to contribute where they may otherwise have abstained from doing so. It is also useful since it displays the opinion of a group of users without having to read a comment from each one that states variations of the same thing. It gives weight to the different contributions to a discussion and highlights where there is consensus and where there is not. In terms of how representative the selected platforms are to the programming community an account of website ranking and platform participants follows. According to Alexa, the selected platforms are some of the most visited websites in the world (2019a, 2019b, 2019c, 2019d). Regarding programmer communities on Reddit, The largest programming subreddit has just under 2.1 million subscribers (Reddit 2019b). Stack Overflow reported that they have over 50 million monthly visitors and 11 million registered users (Stack Exchange 2019). The Stack Exchange Q&A site Software Engineering has 284 thousand users (ibid.). The topic computer programming on Quora has 3 million followers (Quora 2019b).
- **Stack Exchange** and **Stack Overflow**: A large network of Q&A style forums where many topics of a technical nature are discussed. Both websites contain a large population of developers that ask and answer questions. There is often no simple answer which leads to discussions that can be drawn from. Both websites are part of the same network but they exist on different domains. Stack Overflow is the largest of the two in relation to programming as it is completely specialized towards programming while Stack Exchange consists of many different Q&A communities with a wide, varying range of topics, including software engineering. (Stack Exchange 2019a)

- **Reddit**: One of the larger social media platforms on the internet that contain several subforums (subreddits) dedicated to programming, Computer Science and different programming languages. Most of the larger subreddits are rigorously moderated and enforce rules for what kinds of topics and behaviours are permitted. These rules and intended subject domain are often clearly stated and were useful when deciding if a subreddit was a good source or not. (Reddit 2019a)

- **Quora**: A Q&A style forum where users ask and answer questions. Quora uses several techniques to promote high-quality answers and push them to the top. Apart from user-voting they also take into account if a user submitting an answer has a high reputation, is a verified expert on the subject or if the content contains indicators of a good answer such as pictures and phrasing. Votes are also weighted based on a voter's reputation which prevents gaming of the voting system in the interest of fairness and promotion of good content. (Quora 2019a)

### 2.4.2 Sample Size

An adequate sample size is required for the sample to be considered representative and for qualitative research methods to be applicable. Holtz, Kronberger & Wagner (2012, pp 10-12) states that data from internet forums can be used for almost all kinds of qualitative analysis and that for qualitative methods such as content analysis researchers use a comparatively smaller amount of textual data. Oates (2006, p 100) specified that a sample of 30 is adequate for smaller studies, the final sample size of this thesis is 30 posts.
2.4.3 Selection Criteria for Posts and Comments

In order to ensure relevance and usefulness of collected data, the following criteria for selection of posts and comment sections were devised:

Forums

- Forums, Subforums and posts must be directed towards programmers in order to ensure that the comments come from the relevant groups that have experience in programming.
- The posts must contain at least five comments or more so that there is some discussion taking place that can be studied.
- The post must not be older than ten years since this is a contemporary study.

Comments

- Limit the search within posts to the top 3 comments in order to limit the collecting to only encompass the most relevant content and to make sure that the essence of the discussion is captured.
- Comments containing irrelevant content (i.e jokes, memes, off-topic banter, ad-hominem etc.) are ignored since they contribute nothing to this thesis. Jokes can contain information that can be inferred if one is well versed in the subject matter. But, the ambiguousness of humor on the internet makes this problematic and therefore this kind of content is disregarded in favor of serious contributions to a discussion.
- Links found within comments referencing potentially relevant material will be followed to a maximum depth of one hop. This was done in order to understand the meaning behind a comment or if the link was a focal point of a discussion post.

2.4.4 Methods for categorizing discussion aspects

In order to first gain a grasp on what aspects are being discussed, the method of Descriptive Coding was used in order to create a categorized inventory of the discussion data’s content. The idea behind descriptive coding is to use short nouns and memos to identify the aspect discussed within for example a corpus of textual data. Descriptive coding is suitable for almost all types of qualitative research. (Saldaña, 2013, pp 88-91)

As many aspects were discussed regarding FP and OOP it was necessary to limit them as it would not be possible within the thesis’s scope to address all the categories, this was solved by using magnitude coding. It is a method that Saldaña recommends to follow up descriptive coding with (2013, p 90). It is used to describe the intensity of already categorized material and it is a fitting method for studies that are qualitative and descriptive and deals with elemental statistical data (Saldaña 2013, pp 72–73). According to the magnitude coding
method one way to code the intensity is to do it with different numbers and then summarize their total value to reach a conclusive result (Saldaña 2013, pp 73, 75).

When the most discussed aspect had been identified the pattern coding method was applied to all of the segments of the chosen aspect. The reason for using this method was to identify patterns in aspects which was discussed so that the different programming challenges related to the aspect could be discovered. Pattern coding is a method that is appropriate for a second cycle of coding (Saldaña 2013, pp 209). It is meant to take a lot of data and transform it into units that can be worked with in a more meaningful way (Ibid.). The patterns are identified not by finding the exact same occurrences of something but by finding things that would appear similar (Ibid, pp 209-213). For example, words in different reviews of a restaurant such as “richness in taste”, “great food” and “a competent chef” could be summarized into a pattern described as “highly regarded restaurant” (Ibid.).
3 The Paradigms of Functional and Object-Oriented Programming

This section describes concepts of FP and OOP that are chosen because of their relevance to understand the theory and analysis that follows. This means that these descriptions are not complete descriptions of the paradigms but adapted to the context of this thesis.

3.1 Functional Programming

This section describes some relevant concepts that are useful for understanding the characteristics of functional programming in the context of this thesis.

3.1.1 Declarative Programming

The paradigm of declarative programming (DP) is focused on what the desired result should be and not how it should be achieved. As long as the result fulfills the criteria of the expected result any implementation of the code is deemed acceptable. Compared to imperative programming, DP does not provide any control of execution flow nor any assignment statements. The computation model of von Neumann is not connected to DP (A Dictionary of Computer Science, 2016).

3.1.2 Background

Functional programming has its roots in lambda calculus invented by A. Church in the 1930s. At the time of its invention it was not considered a programming language as computers that could have run it did not exist at the time. Church’s intention behind developing lambda calculus was to use functions in a way that they could be intuitively understood in order to take advantage of functions computational property. In order to do that a divergent step was taken from the theory were functions are seen as sets and recursion was introduced, which was a contradiction to the set theory as it is not possible for a set to contain itself. However recursion is the central aspect of lambda calculus and it is that which makes it potent. Modern FP languages as of today can be seen as lambda calculus with additional features. (Hudak 1989, p 363)

3.1.3 Immutability and Side Effects

A main characteristic of strict FP is that neither mutability nor side effects are allowed. In FP data is immutable, once data is created its value cannot be reassigned (Hughes 1988, p 98). A side effect is what happens when a function does something outside of its scope other than returning a result (Chiusano & Bjarnason 2015, p 3). A few examples of side effects would be a function that changes a variable value, changes a data structure, writes or reads data or
shows something on a monitor (Ibid.). One of the disadvantages of side effects is that they make it harder to understand a program as it is necessary to take into account the side effects and how they affect other parts of the programs (Chiusano & Bjarnason 2015, p 12). While a function without side effects is simple to understand as it only has a local impact in the form of the returned result (Ibid.).

However, it should be noted that many FP languages allow mutability and side effects in a limited fashion to make some aspects more effective and easier to work with. Naturally, this deviation comes with its cost by decreasing the benefits that the counterparts: immutability and the absence of side effects provide. (Sutter & Laurus 2005, p 60)

### 3.1.4 Higher Order Functions

In FP functions are treated as any other value, that means that functions can be passed as arguments to other functions, they can be returned as output from a function and they can be stored in data structures (Narbel 2009, p 183). When a function can take other functions as arguments they are called higher order functions (Chiusano & Bjarnason 2015, p 19).

### 3.1.5 Pure Functions

A pure function is a function that does not have any side effects. It takes input and gives output. An example of this would be a length function that takes a string as input and returns an integer based on the length of the string. The integer return value is only dependent on the string input value and nothing else. It will always return the same integer value for a given string and nothing more, which makes it a pure function. (Chiusano & Bjarnason 2015, pp 10–11)
3.2 Object-Oriented Programming

This section describes some relevant concepts that are useful for understanding the characteristics of object-oriented programming in the context of this thesis.

3.2.1 Imperative Programming

Imperative programming, as its name implies, is done by giving a set of explicit instructions on how a task is supposed to be performed. Whatever result is expected, it is not the focal point of imperative programming. The programmer controls the flow of the overall execution through the use of conditional statements which can lead to different sets of instructions depending on the evaluation of the conditional statements. Another characteristic of imperative programming is the assignment statement that is implemented destructively. This means that an assigned value can be replaced by another if assigned to the same memory address. Imperative languages such as Cobol, Fortran, C++ and Java are closely related to the von Neumann model of computation. (A Dictionary of Computer Science, 2016)

3.2.2 Procedural roots

In the 1970s and 80s there was a wide use of procedural and structured languages such as C, Pascal and Fortran when developing business-oriented software systems. Procedural style code is structured in a top-down linear manner meaning the program execution flow follow a series of consecutive steps from start to finish. Structured style code functions in much the same way but with some additions like flow control functionality (for and while loops) and the ability to have returning branches in the execution flow of the program. This was sufficient for smaller and less complex systems but became cumbersome as projects grew in size. The work of maintaining and debugging these programs became harder and when there was additions or changes to existing functionality this could often adversely affect the rest of the system. Furthermore there was a change where systems evolved into distributed models where the database, business logic and user interface were loosely coupled and accessed over networks rather than in a centralized system which made this style of programming even more difficult. In the 90s object-oriented languages like C++ and Java became more widely adopted. C++ was arguably difficult to work with and imperfect in terms of object orientation since the developer had to manage things like memory and security manually. Then Java gained momentum since it was a managed language that freed the developer from these tediums and woes so that they could instead focus on perfecting the business logic. (Clark, 2011)

Note that the Procedural style and the object-oriented style are not opposites but rather complements to each other. Modern object-oriented languages are still written as a set of consecutive instructions that are run from start to finish but with the added benefits of objects, encapsulation, abstraction, composition and other features. (Weisfeld, 2013)
3.2.3 Object-Oriented programming style
The idea behind the object-oriented style of programming is that instead of thinking about for example business logic as a series of steps that one would take to achieve a result, it is a way to model the way people and organizations actually think and behave in the real world. The OOP style of encapsulating attributes and behaviour into logically grouped objects that interact with each other is meant to reflect the way people behave, communicate and interact with one another. (Weisfeld, 2013)

3.2.4 The concept of objects
According to Weisfeld (2013, p 7) an object is a fundamental building block of object-oriented programs and are comprised of both attributes and behaviours. This is one of the main differences between structured and object-oriented programming since the data and the functions traditionally were kept separate. Objects communicate with each other through sending messages or invoking methods (Clark, 2011, p 1). Concretely an object is an instance of a class and contains data in the form of attributes and defined behaviour in the form of methods that can be used to manipulate the data and to perform other tasks.

3.2.5 Access modifiers
In order to control state changes and ensure that attributes and methods are only available to the objects that need them. Object-oriented languages use what is called access modifiers to provide controlled access to select elements of a class. Public attributes and methods are available to any object while private methods and attributes are only available to objects of the class associated with them. It is this feature that form the basis of encapsulation and data hiding. (Weisfeld, 2013, pp 21-22)

3.2.6 Encapsulation and Data hiding
When logically related attributes and behaviour is bundled up into objects then that is called encapsulation. This is one of the big features of object-oriented programming since it groups together the data related to the object with the functionality meant to be applied to that data. Another term associated with encapsulation is data hiding. In good object-oriented design, only the methods and attributes needed to perform a certain task is exposed while the internal implementation remains private and hidden from the rest of the program. The public method should then be designed like an interface so that users can interact with the objects without needing a deeper understanding of the internal workings of the class. This provides benefits in security, testing and maintenance since it allows for controlled access and change of state to occur while still strictly regulating it. (Weisfeld, 2013, pp 21-24)
4. Theory

This section begins with a description of the concept of concurrency which lays the foundation for the next section 4.2 that handles the concept of state and its relation to concurrency and performance. Section 4.2 is used for analysis of the empirical material and it is done in accordance with the section’s structure.

4.1 Concurrency

Concurrency in computing refers to computing different computations in parallel (Sutter & Laurus 2005, pp 54–58, 62). As the increase in computer power is being realized progressively by increasing the number of CPU cores rather than making transistors smaller it has also put a demand on software design to adapt (ibid.). To make use of multiple cores the programs themselves have to be split up into threads so that they can run concurrently (ibid.). Threads are computing processes that run in parallel while sharing memory (Lee 2006, p 33). In most applications threads interact with each other except in simpler applications (Hwu, Keutzer & Mattson 2008, p 315). Applications like web servers, can easily make use of threads, but the reason for this is that the threads can run independently from each other (Lee 2006, p 33). They do share data but it is done through a database (ibid.). But, usually it is difficult to design concurrency as the different threads needs to interact with each other and it puts higher demands on the skill of the programmer (Sutter & Laurus 2005, pp 54–58, 62).

There are two dimensions that should be taken into account when dealing with concurrency: the size of the split threads and the level of their coupling. The size is important as it has an overhead cost to make computations concurrent. Coupling in this case relates to how much communication and synchronisation there is between threads of the program, where the optimal level of coupling is zero. Such a program is easy to code and it means that the tasks can run independently and in any order without costs for communication or synchronization. Nonetheless, such programs are rare. (Sutter & Laurus 2005, p 57)

Programs are made up of execution threads which are sequences of related instructions that are processed by the CPU. By handling multiple threads in a single core CPU the processor could achieve a performance increase but this is limited due to the use of shared memory resources. In order to mitigate this, multiple cores were implemented within the same CPU chip so that each thread could have its own set of execution resources by the cores having separate caches. Multi-core processors are MIMD (Multiple Instructions Multiple Data) because different cores execute different threads. Furthermore, multiple CPU chips on the same motherboard resulted in higher power consumption and temperature increases which was undesirable, this also contributed to the development of multi-core CPUs. Multi-core CPUs are becoming more widespread due to increased demand for higher levels of
performance. However, software still needs to be specifically programmed to take advantage of multi-core CPUs. Few programs are designed to take full advantage of this type of hardware which means that the power is not being utilized fully. (Chhibber & Garg, 2014, pp 31-32)

4.2 State

State in a program is defined as the set of values of all the variables at a certain time during execution (Prata 2004). Therefore shared state is when different objects or threads in a program have read or write access to the same set of variables. There are some cases when programming can be written completely without state, but this is a radical approach (Hinsen 2012). It can be done in FP, as programs can be composed solely of functions that transform a value (ibid.). However this is only viable to solve certain types of problems (ibid.).

4.2.1 Mutable state

Mutable state is state that can be altered without replacing the variable containing the value.

Concurrent with Mutable State

The most prevalent variant of concurrency in multi threaded software is unstructured concurrency. It means that the threads that run in parallel are different to each other which makes it hard to share state between them as it is not predictable. It requires that the threads are synchronized to avoid data races. This requires explicit synchronization of the threads otherwise the program will not operate deterministically. (Sutter & Laurus 2005, p 58)

Concurrent programs that use threads are generally difficult to reason about from a programmer perspective (Hovemeyer & Pugh 2004, p. 80). This is due to the many possible behaviours that a program can exhibit due to nondeterminism and the complexities that come with multithreaded execution (ibid.). This is especially true for mutable state, it makes programs complex and harder to comprehend which leads to errors commonly associated with state. (Moseley & Marks 2006, p 6).

Mutable state is problematic when it is shared between parts of a program or different execution threads running in a concurrent program or in other forms of distributed parallel systems. The problem occurs when one part modifies the shared state in a manner that is not expected, this is a frequent source of bugs when dealing with concurrent programming and mutable state. (Hinsen, 2012)

Data races are a type of bug that can occur when threads share mutable state. The problem appears when two threads can access the same state which they can both read and write to. If these threads are not coordinated there is a risk that either thread write or reads a value that is not correct because of interference from the other thread. (Sutter & Laurus 2005, p 58)
Races can be prevented by implementing locks. A lock can hinder another thread from accessing the same data at the same time. Locks work but they bring their own problems, one is that they can end up in deadlocks which happens when two threads have locked up some data while trying to access each other’s locked data. This can be a hard thing to prevent from a design perspective as it is up to the programmer to be careful and know what lock to pick and at what time when dealing with shared mutable data. (Sutter & Laurus 2005, p 58)

**Performance with Mutable State**
As mutable state allows modification of data structures instead of copying and recreating them it is inherently more efficient than immutable data structures. However, there are workarounds that can make immutable data structures reuse its earlier state making their efficiency comparable with their mutable counterparts. But there are some data structures like arrays that cannot be implemented in other ways than with mutable state without losing efficiency. (Hinsen 2012, p 82)

**4.2.2 Immutable State**
Immutable state is state that cannot be altered without replacing the variable containing the value with a new variable containing the new state.

**Concurrency with Immutable State**
FP languages like Haskell and Scheme have a natural advantage towards parallelism as they use immutable data which is suitable for concurrency (Sutter & Laurus 2005, p 60). Furthermore, as side effects are less common in these languages it puts less importance on the order of execution (ibid.). However, much of the concurrency in FP languages is situated in procedure calls, which is a size that is a bit too small to be optimal for splitting up on multiple CPU cores (ibid.). That being said, programmers that implemented concurrency in both functional and imperative languages considered it to be easier to do so using a functional language (Pankratius, Schmidt & Garretón, 2012, p 131).

Immutability provided by FP provides a significant benefit when dealing with concurrency since using immutable data with multiple threads is possible without running the risk of encountering race conditions and deadlocks (Manning Publications, 2017). Because of immutable data some operations that run in parallel does not have to be coordinated such as when a search engine queries its database (Sutter & Laurus 2005, p 57).

Race conditions are present in both FP and OOP languages and can be considered a major issue when developing concurrent software due to the difficulty in locating the source of the race condition. However, the reason for there being occurrences of race conditions in FP is due to a language permitting mutable data to be used (Pankratius, Schmidt & Garretón 2012, p 124, 131)
Performance with Immutable State

Immutability provides thread safety at the cost of worse performance (Pankratius, Schmidt & Garretón 2012, p 131). One reason for the decrease in performance is that updating some immutable data-structure can trigger expensive copy processes (ibid.). Performance is a problem when working with immutable data-structures (Hinsen 2012, p 82). For example, updating the last element of an immutable list can only be done by copying all the elements and creating a new list. This process is very inefficient since it means a completely new list is created for each element added (ibid.). This problem can be solved in large by using persistent data structures which is a common data structure throughout functional programming (ibid.). Such a data structure reuses parts of the data structure when it is updated and thereby bypass the need to perform copying (ibid.). However, all data structures do not have a corresponding persistent version, there are some data structures that have to remain mutable to maintain their efficiency, such as arrays (ibid.).
5. Analysis and results

In this section, the empirical material is presented in the form of verbatim quotes. The analysis is arranged in accordance with the structure of section 4.2. Therefore, the quotes are grouped into two sections; quotes that are in favour of mutable state and quotes that are in favour of immutable state. Each quote is summarized and then followed by an analysis where the claims made in the quotes are compared with the corresponding theory regarding concurrency and performance in section 4.2.

The original dataset that was collected consists of 30 forum threads. The distribution of threads across forums is as follows: Reddit: 15, Quora: 8, Stack Overflow: 6, Stack Exchange: 6. The aspects identified by going through all the threads in accordance with the selection criterion for comments resulted in a multitude of different aspects. The top 3 recurring programming aspects were in descending order of frequency: mutability & state, multiparadigm and design. Of these three mutability and state and multiparadigm were very close in frequency but the aspect of mutability and state was generally more debated and therefore selected as the primary aspect for further analysis in relation to the second part of the research question.

Out of the 30 threads, the threads relating to the selected programming aspect were 18 in total and they are distributed as follows: Reddit: 6, Quora: 2, Stack Exchange: 5, Stack Overflow: 5. The coded segments collected from the 18 threads that addressed the aspect amounted to 62 segments. Within these 62 segments we found 23 occurrences that explicitly mentions the concept of mutability together with the concept of state, additionally there were more comments that discussed these two concepts implicitly. This is the reason for combining mutability and state into one category. It is also worth mentioning that several if not most of the statements favoring immutability were also in favor of FP and the same goes for mutability that was more favored by advocates of OOP. The segments that represented the different viewpoints with the greatest accuracy were extracted and presented in the thesis.
5.1 Mutable state

This section contains quotes that were categorized as in favor of mutable state. Selected quotes are presented followed by a summary and an analysis, first from the perspective of concurrency and then from the perspective of performance. These quotes were chosen to be presented as they represent a broader point of view found in the discussion material with different opinions for why mutable state is a beneficial property.

Quote 1

“\(^{7}\)I implemented the Myers Diff longest common subsequence algorithm. The core of this algorithm is an array that is read and written to with great frequency as you traverse the buffers. As a divide and conquer algorithm, it's also very easy to parallelize. My implementation (as in the paper) relied heavily on mutable state. An immutable functional approach would transform simple load/stores into tree traversal / allocation, and be orders of magnitude slower. Indeed the Haskell LCS is implemented via MArray, and you can guess what the M stands for. “

- Thread 13, Person D

Person D discusses a type of algorithm which is dependent on mutating state frequently and argues against an immutable approach by stating that it would be much slower than the mutable implementation of the algorithm.

When dealing with concurrency on the procedural level which is at the level of the algorithm that person D writes about. On this level, immutable state through the use of FP is actually beneficial according to the theory (Sutter & Laurus 2005, p 60). But it does not necessarily mean that mutable state will result in worse thread safety. It depends on the algorithm itself, if it is not necessary to have complicated locking mechanisms then there is not any real drawback and it does become a question of performance. According to the theory regarding performance, more operations would be required for updating an immutable data structure which implies that the mutable implementation by person D is more efficient from a performance perspective (ibid.).

Finally, Without taking the specifics of the described algorithm into account, the claims made in this comment are in alignment with established theory.
Quote 2

“[…] As I now see, FP practices can also be applied to many a->a' problem domains, where the transformation yields an updated version of its starting state, a process that can be re-applied recursively. However, one begins to incur an escalating cost (particularly in performance) as 'a's state structure becomes increasingly complex, unpredictable, and interwoven with the various functions that perform the transformation. Imagine, for example, an incremental, generational garbage collector that is implemented with FP, only able to work with immutable data. Ironically (at least to me), FP technology depends on mutable data infrastructures: RAM, file systems, RDBMS, etc. It uses clever algorithms like HAMT and RRB-Trees to reduce the performance penalty for manipulating alterable (but immutable) containers. But if the problem domain requires you to change the value of x inside 'a' so all functions now see and work with the new value, there may be ways FP can accomplish this, but with some situations there will be a performance and perhaps code complexity cost that likely outweigh the testability and parallelization benefits (if any in that situation) of using FP's immutability discipline. [...] “
- Thread 9, Person A

Person A states that there are many situations where FP can be used to update a data structure based on its original state but that this incurs costs in terms of performance and complexity which would outweigh the benefits of testability and easy parallelization.

In accordance to the theory, person A mentions that immutable state does bring benefits to parallelization (Sutter & Laurus 2005, p 60). But, person A also states that it could lead to increased costs in the form of code complexity. Person A mentions that FP can change a value within a data structure without a performance penalty but that there will be a penalty in some cases. This is comparable to what Hinsen (2012, p 82) mentions; while some immutable data structures have workarounds to maintain performance, not all of them do. Overall, the comments is somewhat vague, none of the claims contradict the theory but it cannot be concluded if the statement regarding code complexity costs is in a concurrent or non-concurrent context.
Quote 3

“Another example, F#. You could say it's mostly FP, but there are many OOP concepts and constructs available. You can define classes, abstract classes, interfaces, deal with inheritance. You can even use mutability when it makes your code clearer or when it dramatically increases performance. Many modern languages are multi-paradigm. “
- Thread: 1, Person: A

Person A states that a benefit of the language F# is that it supports the use of mutable data even though it is mainly considered a functional language. Person A thinks that mutability can increase readability and performance.

Nothing in this comment refers to concurrency. However, as in accordance with the theory person A acknowledges that mutability can in some cases increase performance (Hinsen 2012, p 82). In this case it is illustrated how it can be used in tandem with immutability as part of a multiparadigm language. Person A’s statement is in alignment with the theory.

Quote 4

“ ‘Concurrency; Plays extremely nice with the rising multi-core technology.’ The problem is just pushed around. With immutable data structures you have cheap thread safety at the cost of possibly working with stale data. With mutable data structures you have the benefit of always working on fresh data at the cost of having to write complicated logic to keep the data consistent. It's not like one of those is obviously better than the other. “
-Thread 15, Person B

Person B neither argues for nor against mutable state but he does argue against another commenter who argues for immutable state. Person B mentions pros and cons with both mutability and immutability. B thinks that immutable data makes it easier to ensure thread safety but it makes your data less flexible. While mutability lets you work with fresh data but that it comes with increased code complexity to handle the mutable state.

Person B states that it is easier to implement thread safety into an application with immutable state. That is in alignment with the theory (Pankratius, Schmidt & Garretón 2012, p 131). Furthermore, person B claims that there is a cost associated with mutable data because of the need for complicated logic in order to ensure that data remains consistent. Which is in accordance with the theory that concurrent programming is generally considered difficult due to the complexities that comes with managing state (Hovemeyer & Pugh 2004, p. 80). It is easier to reason about immutable state and harder to make mistakes as some bugs such as race conditions are neutralized (Manning Publications, 2017). As mutable data indeed requires the implementation of additional logic to maintain thread safety (Sutter & Laurus...
2005, p 58). When it comes to performance, it is not discussed. All the statements made are aligned to the theory.

5.2 Immutable state

This section is structured in the same way as the previous section 5.1. These quotes represent the other side of the discussion where favour is towards immutability.

quote 5

“Avoiding mutable state is especially important for easy parallelization of your programs. If a function changes its arguments instead of returning newly created values, it's very difficult to reason about the state of the program, and parallel processes can't run independently, because they need to know (or check) what's already been done. A disadvantage of avoiding mutable state is that your program may need a lot more memory, since it's always creating new objects instead of modifying existing ones.”
-Thread 11, Person B

Person B claims that mutable state should be avoided when working with concurrency since it makes code harder to reason about regarding its various states and hampers parallelization with the trade-off of increased memory costs since immutable state requires that objects are re-created instead of modified.

Person B is correct in that mutable state makes it harder to parallelize programs as it becomes more complex to reason about (Moseley & Marks 2006, p 6). But Person B states that parallel processes cannot run independently, this is not in accordance with the theory, which states that threads do share memory but they can still run independently from each other (Lee 2006, p 33). If person B means that they are not running independently because of shared memory then it is expressed lously. Person B claims that there is a performance cost when working with immutable data due the way objects are re-created instead of modified which is in alignment with the theory (Hinsen 2012, p 82). The comment person B made is in accordance with the theory but it is vague, it implies preconditions that are not unreasonable but since the arguments hinges upon those preconditions they should have been expressed explicitly.
“As CPUs can't really be minimized anymore (at least not for long as we're approaching the atomic sizes of the components) the manufacturers have switched from building faster CPUs to adding more cores. However, none of the mainstream OOP languages takes advantage by default of multi core architectures- nor do OOP languages make it easy to do concurrent programming. The main culprit is the mutable state inherent in objects. FP deals with immutable data and apply functions to this data in a way very similar to mathematics. In functional programming f(x) will always yield the same result for the value "x". That's why it's very easy to run such functions in a concurrent manner.”
-Thread 11, Person C

Person C discusses the physical limitations of modern CPUs in terms of the size of its components and how it relates to the proliferation of multicore architecture. After that, he makes a statement about OOP languages not utilizing this architecture by default and that this is due to the presence of mutable state in objects. Finally, he argues for FP with its determinism and the way it simplifies concurrent implementations.

From a concurrency perspective, person C is not clear in the reasoning behind the statement that OOP languages do not take advantage of multicore architecture by default. What Person C probably means is that OOP has no inherent advantage because of mutable state. Performance wise it is correct that manufacturers have shifted focus to creating more CPU cores to increase computing capability (Chhibber & Garg 2014). But it is wrong to argue that mainstream OOP languages does not take advantage of this. As stated by Sutter and Laurus, already in 2005 computing power was increasingly realized by increasing the number of CPU cores (Sutter & Laurus 2005). Therefore it is safe to assume that all mainstream programming languages of today takes advantage of multi core architectures. While FP has attributes that help with concurrency, the concurrency is often located at the procedural call level, these smaller chunks are not always advantageous to split up on different cores as there is an overhead computing cost to do so (Sutter & Laurus 2005, p 57, 60). Person C makes statements that by themselves are in alignment with the theory. But, the statement that OOP does not take advantage of multi core architectures would contradict the trend within CPU technology development. If that is not what is meant it is a vaguely formulated statement.
“I will say that race conditions in Erlang do genuinely seem to be more manageable and less pernicious than shared-state imperative programming. However, it's not race-free either, as commonly purported.“
-Thread 13, Person B

Person B claims that the bug known as a race-condition is easier to manage in a non-imperative way but that FP (Erlang) does not completely eliminate them.

Regarding concurrency, it is correct that it is easier to handle race conditions in FP. If the data that is shared between the threads is immutable it cannot be altered and there is no risk of reading an incorrect value (Manning Publications, 2017). While pure FP would remove all race conditions, FP languages in general do not. The reason why is because many FP languages allows the use of mutable data, while immutable data is the default (Sutter & Laurus 2005, p 60). Performance wise nothing is mentioned. Person B’s statements are in alignment with the theory.

“While your post has many points that merit complex responses, I'll start with this one. Functional programming is actually a very natural fit for traditional video games, because they are discretized. Each state can be seen as a function from the previous state plus new input to a new state, producing a new rendering frame. That does not have to mean you need to use a functional programming language - John Carmack wrote about functional programming in C++, because functional languages are probably too slow for him. That doesn't mean the style isn't still a useful way to manage complexities of state that would otherwise be unwieldy”
-Thread 9, Person B

Person B, claims that since FP has higher-order functions it makes state in video-games easier to handle but that FP is not necessarily perfect in spite of this. B goes on to claim that the FP style makes it easier to handle state complexities that would be unwieldy to implement through an imperative style. But that these two styles could be combined with benefit.

Regarding concurrency Person B mentions that FP is good for managing state complexities which is in accordance with Sutter & Laurus (2005, p 60). FP implies the use of immutable data. Performance wise, person B states that it would be slower than imperative programming, but it is not possible to determine from which context this statement is made without asking person B to go into the specifics behind the reasoning. While mutable state is efficient in more instances there are many cases where equivalent implementations of immutable data structures can match the efficiency of mutable data structures (Hinsen 2012,
But as an overall statement it is fair to say that person B is in alignment with the theory. Functional style can be used in an imperative language like C++ to gain benefits from both paradigms (Hu, Hughes & Wang 2015). In the same manner FP can use OOP concepts (Sutter & Laurus 2005, p 60). Person B’s statements are in alignment with the theory.

quote 9

“On the metaphorical side, one could say that functional programming is about growing your dictionary or glossary; functional programming is very linguistic (or poetic, or mathematic, as you please); it is about defining words, and then composing those words in larger expressions to obtain new meanings. Perhaps hearing about things like monad transformers or monoids in the category of endofunctors doesn’t make feel like it, but functional programming is simpler than OOP: lack of state will always be easier than having to manage state.”
-Thread 10, Person A

Person A claims that FP is simpler than OOP even though it does not seem like it because the absence of state will always be an easier alternative than to manage state. Person A equates programming in a functional language to forming sentences with more complex meaning by combining single words.

Regarding concurrency, it is in alignment with the theory that it is easier to deal with FP than OOP in the sense that state does not have to be managed to the same extent (Moseley & Marks 2006, p 6). However, while FP programs can be implemented completely stateless it is only useful for certain types of problems (Hinsen 2012, p 82). Therefore it is not very generalizable to say that FP will always be easier because of it lacking state, as that is only the case for certain FP programs. Person A makes no mention of anything relating to performance. While person A does not directly contradict the theory, the statement is vague, but if person A implies that FP programs always lack state then it is in contradiction to the theory.
6. Conclusion

In this section the research question posed in the beginning of this thesis is answered.

The FP and OOP programming aspect that was the most discussed was *state and mutability* and the aspects that were most frequently related to the discussions were *concurrency* and *performance*. Regarding the nine quotes that were analyzed, five of them are in alignment with the theory while four of the quotes are too vaguely phrased to determine what the individual person meant. This does not mean that those four quotes contain contradictions to the theory but their claims are ambiguous and therefore cannot be validated. In conclusion, there were no obvious discrepancies between the discussion and the theory.

7. Discussion

This section contains a discussion regarding the results together with interesting observations that were made during the course of this project.

The results of the analysis concluded that there were no obvious discrepancies between the discussion and the theory. There is probably more than one reason for this, but one could be the fact that this project focused on the comments with the highest amount of votes, since this indicates that the content of the comment is agreed upon by many other people. However, consensus does not correlate directly to factual correctness, which is the reason behind this choice. It can be argued that if discrepancies are to be found then looking at the comments where most people agree is not the place to look. However, it would have been trivial to find inaccuracies in comments with low votes and probably much of the content would have been of poor quality. If one were to analyse such data then the results would probably not be very interesting. Additionally, every vote cast represents one persons opinion, so a comment with many votes is more representative of the community as a whole.

As Glass (1996) wrote, there are instances when practice precedes theory. In this study the people discussing the aspect of mutability and state could be seen as the practitioners of the field. By putting their statements in comparison with the theory it could have potentially shown arguments or ideas that were different or contradicted the theory in the field. However this was not the case as the quotes from the practitioners either went in alignment with the theory or were too ambiguous to categorize. It could depend on the theory of the thesis being focused on too many high level concepts, a level where concepts already are well established. Maybe the results would have been more diverse by instead going into a lower and more technical level of mutability and state, a level which is less explored. But considering how small the final data set was it could of course have shown different results by expanding the
empirical material gathering. In hindsight, the problem is that mutability and state is a subject that is too broad. The discussion between FP and OOP was certainly a broad entry but it was okay as a first step to find out what was actually discussed in this context. A study that would focus on mutability and state from the beginning would have a bigger pool of relevant material and could probably have given a more diverse conclusion.

An interesting thing that became apparent during the course of this project was that the comments often contained some small error or omission. Comments made were not exhaustive in their attempts to answer the question posed in the post. It could be assumed that this is because people generally do not have the time or the motivation to write an essay which fully explains a subject in order to answer a question regarding a programming aspect that may be considered basic by experienced developers. This does not mean that there were many contradictions or direct errors but rather that there were instances where claims were made in a way that assumed that readers had a firm grasp of the subject at hand. People were generally less concerned about the risk of being misunderstood or about expressing themselves in a clear manner in order to properly make a point.

The problem with ambiguous comments and careless phrasing is that they may imply something incorrect when in fact they mean something else entirely. Those with experience and an understanding of the subject at hand can probably see past careless omissions and fill in the blanks without active cognitive effort. They simply understand what the comment is meaning to say in context to the subject. The problem is more substantial for readers who lack experience or an understanding of the subject. They may look to the most voted answers and assume that the literal statement is true when in fact there is more to a subject than what appears at first glance. The insight that can be drawn from this is that if one is new to a subject then it is certainly beneficial to complement any knowledge found on public forums with additional sources that take greater care regarding phrasing and that goes into sufficient detail when explaining complex subjects. This is not to say that the forums are not a good source of information, on the contrary, forums can provide assistance in finding a direction when trying to learn more about a subject. It is advised that even highly up-voted comments should not be fully relied upon and taken with a healthy dose of salt.

Another problem with carelessly ambiguous phrasing is that it often muddies the point that is being made which leads to (sometimes heated) discussions about a misunderstanding that had occurred because of this. Regardless, the discussions were generally held in a factual and helpful manner and it was rare to find any form of hostility or ad-hominem type arguments. Some people are perhaps frustrated by inaccuracies but instead of venting that frustration, it seems like some become motivated to make an effort to post correcting or complementary remarks which is beneficial for all readers. However, it was very rare to find any form of sources that support claims made in the discussion. This is probably because finding such resources takes time and effort. But if a person is highly motivated to display their knowledge
of a subject or highly invested in a discussion then they can sometimes go to that length in order to make a point, which further improves the quality of the discussion.

There was one particular misconception that was identified that clearly illustrates the problem of careless phrasing. During the study it became evident that there is a common misconception regarding object-oriented programming which is that imperative programming features are an inherent part of OOP. The reason for this is probably because many popular OOP languages such as C++, C# and Java are imperative as well as object-oriented. However, OOP and IP are two different paradigms that although commonly combined can be completely separated. For this project, we simply chose to refer to imperative object-oriented programming as OOP in order to prevent confusion when analysing the discussion material.

During the analysis it became clear that performance is a hard aspect to analyze without going into the specifics of the algorithm or the context of the software implementation. While there are some basic aspects that can be taken into consideration, such as how immutable data structures cannot be updated in the same straightforward manner as mutable data structures. It is hard to draw any interesting conclusion without getting into the details. It would therefore be interesting to see studies that looked into mutable and immutable data in certain implementation contexts and with certain algorithms. Moreover, for anyone who wants to do further research about the paradigms of FP and OOP could look into any of the aspects that was found out to be discussed the most online. The top three aspects that were discussed in descending order are; mutability and state, multiparadigm, and software design.
8. References


