ON THE HUMAN FACTORS IMPACT OF POLYGLOT PROGRAMMING ON PROGRAMMER PRODUCTIVITY

by

Phillip Merlin Uesbeck

Master of Science - Computer Science
University of Nevada, Las Vegas
2016
Bachelor of Science - Applied Computer Science
Universität Duisburg-Essen
2014

A dissertation submitted in partial fulfillment of
the requirements for the

Doctor of Philosophy – Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2019

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

November 15, 2019

This dissertation prepared by

Phillip Merlin Uesbeck

entitled

On The Human Factors Impact of Polyglot Programming on Programmer Productivity

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy – Computer Science
Department of Computer Science

Andreas Stefik, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Jan Pedersen, Ph.D.
Examination Committee Member

Evangelos Yfantis, Ph.D.
Examination Committee Member

Hal Berghel, Ph.D.
Examination Committee Member

Deborah Arteaga-Capen, Ph.D.
Graduate College Faculty Representative

Abstract

Polyglot programming is a common practice in modern software development. This practice is often considered useful to create software by allowing developers to use whichever language they consider most well suited for the different parts of their software. Despite this ubiquity of polyglot programming there is no empirical research into how this practice affects software developers and their productivity. In this dissertation, after reviewing the state of the art in programming language and linguistic research pertaining to the topic, this matter is investigated by way of two empirical studies with 109 and 171 participants solving programming tasks. Based on the findings, the design of a data management library, a common use-case for polyglot programming, is proposed broadly and then applied specifically to the language Quorum as a case study. The review of previous studies finds that there is a pattern of productivity gain that can be explained by the occurrence of type annotations in programming, which gives insight into how programmers comprehend code. Study results show that there is a significant improvement of programmer productivity when programmers are using polyglot programming in an embedded context ($\eta_p^2 = 0.039$) and that less experienced programmers do better in a group with more frequent, but less severe, switches, while more experienced developers perform better with less frequent but more complete switches between languages. A study on language switches on a file level shows that file level programming language switching has a clear negative impact on programmer productivity ($\eta_p^2 = 0.059$) and is most likely caused by the increased occurrence of errors when switching.

Acknowledgements

First and foremost, I want to thank my supervisor Dr. Andreas Stefik for his continued support of my efforts and for continuously pushing me to be the best researcher I can be. Without him and Dr. Stefan Hanenberg, who helped me get started on this journey into empirical programming language research, I would never have been in the situation to strive for a doctorate to begin with. I also have to extend special thanks to Dr. Deborah Arteaga, who helped me dive headlong into the topic of code switching and helped me understand linguistic research, without which this dissertation would look very different. I also have to thank the other members of my committee, Dr. Pedersen, Dr. Berghel, and Dr. Yfantis who have always shown confidence in my abilities and readily given advice.

Furthermore, I have to thank my colleagues of the Software Engineering and Media Lab at UNLV, especially William Allee, Patrick Daleiden, and Tim Rafalski, for all their support and collaboration. But most importantly, I want to thank my wife Emma for inspiration and support throughout the process, as well as the opportunity to make a life outside of research and work.

Lastly, I want to thank my family and friends, old and new, for supporting my ambitions and my friends for always having an open ear.

Phillip Merlin Uesbeck

University of Nevada, Las Vegas
December 2019

Table of Contents

List of Tables

List of Figures

List of Code Samples

Chapter 1

Introduction

In today's software development industry, there are over 8945 [Pig] choices available when it comes to choosing a programming language appropriate for a new project. While only about 24 general purpose languages are commonly used [MB15], to chose the right language, decision makers have to consider many different aspects of possible languages, such as security, reliability, performance, compatibility with existing software, scalability, available libraries, portability, maintainability, and developer productivity. With all these variables to consider, choosing the right language can be hard and in about 97% of open source projects more than one language is chosen, likely because there might be use-cases in the project that are hard to cover with a general purpose programming language [TT14]. This often forces developers to switch between programming languages within and across software development contexts. The practice of switching between languages within a development context is called polyglot programming [Fje08] or multi-language programming [KWDE98]. Among open source projects, the average number of computer languages (general purpose and domain specific languages) is about five [TT14, MB15] with 53% of commits to projects containing changes in at least two different computer languages at a time [VTTM12].

This suggests that software developers have to know a variety of computer languages at the same time and need to be able to mentally switch between the languages they know at will to complete their work. The process of learning programming is an ongoing area of research, although it is clear that learning programming must be challenging when considering the 20-50% dropout rates in college [WK14, Yue07, MAD+01]. Research on acquiring secondary computer languages shows that learning additional languages is at least still a considerable challenge [SW93]. An important consideration in the need for developers to switch between languages is the impact of the switch on developer productivity in up front development and continued maintenance and thus the overall cost of the software projects. According to the United States Bureau of Labor Statistics, the median salary for a software developer is $103,560 per year [BLS], even small changes to a developers productivity, such as a 3% reduction, there would be a waste of $3106 per year, per developer. Looking at the bigger picture in the United States, with its 1,256,200 software developers,

then the cost of the same productivity impact would be almost $4 billion per year in the United States alone.

The possible costs of productivity impacts on the economy merits an investigation into the state of polyglot programming to see if this practice is beneficial or detrimental to software developer productivity overall and if recommendations can be made to improve the way this practice is conducted. To guide the research within this thesis to a tangible goal, polyglot programming will be analyzed in the context of developing a data management API, with a focus on how to interact with a relational database from within a general purpose programming language. The design of the API will be influenced by empirical findings on the usability of programming languages and APIs from the literature and from experiments conducted for this specific purpose.

1.1 Polyglot Programming

To better establish the term polyglot programming, this section will give an explanation of what polyglot programming is and what common views of it are in the software engineering industry.

Polyglot programming, also known as mixed-language, or multi-language programming, was previously defined as:

"programming in more than one language within the same context, where the context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved" [Fje08]

For the purposes of this research proposal, there is a strong focus on polyglot programming as an activity in which a single individual has to switch between multiple computer languages, be that domain specific languages or general purpose programming languages, to solve a task. With this different focus and Fjeldberg's definition implying that change or creation of code have to be occurring while ignoring the possible need of code comprehension activities, a more fitting definition of polyglot programming can be proposed:

A software development activity requiring the knowledge of, and the switching between, two or more computer languages for the purposes of comprehending, changing, or creating computer code.

This definition incorporates the act of reading and comprehending code, as well as the act of programming itself. Further, the focus of this definition does not imply a necessary business context, as activities such as educational, recreational, and open source development are possibly polyglot programming, even if only one person is working in the same programming context.

From the view of the author, there are three different levels of switching between computer languages in a programming context:

1. **Project**: Switching between languages when switching programming contexts.
2. **File**: Switching between languages within a programming context but between different sections.
3. **Embedded**: Switching between languages within the same section of a program.

To keep the definitions general, programming context was chosen to describe what would typically called a project. Depending on the organization of programming projects, the term can be misleading for the purposes of this description, as multiple projects as used in common integrated development environments (IDEs) like Eclipse or NetBeans might belong to the same programming context for a developer, such as multiple modules that might be developed alongside each other but are held in separate "projects". The same goes for the choice "sections" in the listing. The most common "section" in this sense would be individual files, however some languages and project structures might have different levels of granularity.

Point one, **project** switching, would be switching between languages when switching to a completely different project, such as switching from an application project solely written in Java, to a data analytics project utilizing the language R. This requires a mental switch by the developer, and for the developer to have reasonable skill in both languages, but any possible switching cost might be negligible and insignificant. This is the most granular switch between languages and is not the focus of this proposal.

The second point, **file** level switches, concerns switching between languages between different files of a project. A project often consists of different files that might contain code in different languages and to complete a specified task, a programmer might have to switch between the files and therefore between the languages. An example could be adding a new button to a web application project. A programmer might have to add the button element to the correct HTML file, change the buttons style in a CSS file, write a click handler in a JavaScript file that makes a server call, and create a PHP file that handles the server call. There are different ways the programmer could go about this, such as going back and forth between files or by finishing one part in one file, and then moving on to the next. For this example, the second strategy might be the more likely one, but if more complex changes have to be made, a back-and-forth strategy might be more typical. Both strategies involve switches between languages, even though they happen in different ways and might have different effects of productivity.

The last level of polyglot language switching, the **embedded** level, is switching between two or more language within the same section or file. In this case, the code in a different language might be embedded in the code of the main language of the file. This is the most immediate type of switching and the main

focus of the research within this research proposal. A typical example of this would be SQL inside of a Java program like in the following code:

```
1          ResultSet rs = stmt.executeQuery("SELECT * FROM professors WHERE id < 32  ORDER BY salary DESC");
```

Code Sample 1: SQL in Java program

1.2 Views on Polyglot Programming

In the software engineering literature, programming solutions employing specialized computer languages to solve specific problems are popular. The view that specialized solutions have better qualities than more general solutions is especially found among proponents of domain specific language (DSL) design [KMB+96, vD97, VDK+98, VDKV00, Kru92, HB88].

Domain specific languages are a popular approach to solving software engineering problems that pertain to a specific domain. The introduction of such a language is claimed to empower domain experts to solve a problem instead of having to rely the use of a general purpose programming language (GPL) [Hud97], which the experts might either have to learn first or instruct another person to use which can lead to miscommunication. While there are "Literally hundreds of DSLs are in existence today" [VDKV00] and there are 32 domain specific languages commonly in open source projects on Github [MB15], examples for well-known DSLs are languages like HTML (web structure), CSS (web styling), SQL (database queries), and Make (software building). All four of these languages are typically found in many software development projects to provide specific solutions to reoccurring problems in software development. In a hypothetical web application project, HTML could be used to structure a website, CSS to improve the looks of that website, SQL to extract required data from a database, and Make to compile all of the files into one package. Such a project would likely also require the use of a general purpose programming language. A developer who is working on the project by themselves would have to learn all of the languages involved enough to be able to fulfill all requirements of the project sufficiently and would likely be forced to switch between languages throughout the project.[VDKV00]

Proponents of DSLs argue that the introduction of multiple languages brings advantages to the project that likely outweigh possible shortcomings such as cost to develop a DSL, cost to learn a DSL, and possible limitations in technical efficiency. One of the major advantages for having DSLs in projects is the possibility of letting domain experts solve a specific problem without requiring them to have knowledge of the entire project or even expertise in the general purpose language in use. While the example lists mostly relatively widely used DSLs, which many programmers might know well enough to develop on their own, expert knowledge can still be helpful in achieving best possible results. One example for this might be optimization of complex SQL queries. While software developers might have good overall knowledge of relational databases and the use of SQL, mistakes that reduce query performance can still occur. An expert in SQL might be

able to write SQL which is just as effective but also more efficient than what a general purpose programmer might write. This does not only facilitate specialized solutions to better solve problems, but also enables easier separation of work between team members. Other listed advantages for DSL usage are better maintainability, productivity, and portability as well as the claim that DSLs are typically self-documenting and preserving of the domain knowledge contained in a specific solution in a language that is close to the problem domain.[VDKV00]

While it appears plausible that solving problems with specific languages can lead to more maintainable, productive, and portable results, the limitation of knowledge seems to be a possible shortcoming of this approach. The use of highly adopted languages would likely be less of a problem, as most developers are expected to learn them to a sufficient degree. However, less popular languages might present a substantial hurdle for the health of projects, especially when in long-term maintenance. Use of low adoption languages can lead to projects having to be maintained by rare and highly paid specialists or might require expensive periods of instruction to non-specialists so that they can gain enough experience with the language to effectively maintain the software. The learning problem would likely become more severe if a new developer would have to learn multiple languages at the same time before being able to start working on a project.

Apart from the possible impacts that might be caused by a lack of expertise, the question remains what the effect on programmer productivity is when said programmer has the necessary expertise. When working with multiple languages at the same time, it stands to reason that developers will have to mentally switch between programming languages within the project context, which might increase cognitive load on the side of the programmer, which might reduce productivity in turn.

Studies using fMRIs suggest that the same areas of the brain are used during program comprehension as are used for natural language understanding [SKA+14, SPP+17]. Considering this connection to natural language processing, one can look to the field of linguistics. One area of linguistic research is the phenomenon of code switching and the related topic of language switching. Code switching is the behavior of natural language multi-linguals in which they switch languages within or between utterances inside the same context [HA01, Li96, YTF17] and investigations into the matter have shown a cost for switching between languages [Ols16, Ols17]. If a connection between the natural language switching behavior and switches between programming languages could be made, then linguistic theory could be applied to make predictions about switching behavior and to develop recommendations on how to structure programming projects and programming behavior to avoid productivity losses.

1.3 The Design of a Data Management Interface

To focus the research in the dissertation and to create an actionable result from the research conducted, this document will propose a specific design for a data management interface in chapter 6. The research on polyglot programming is directly related to the design of the library, as language switches are commonplace when connecting a program to a database through a database library. Be this in the case of switching from the general purpose language to the database's query language or, when setting up an object relational mapping framework, the case of switching between general purpose language and XML or other similar declarative languages.

Accessing, changing, and writing data is an essential part software development today and software developers often use structured files or database management systems of various kinds to this end. To easily be able to create, read, update, and delete (CRUD) data software developers use libraries and frameworks in their general purpose programming language of choice to interact with either files or databases. While alternative systems such as NoSQL databases have recently seen more use [HHLD11], professional projects typically rely on relational databases for their data management. This creates a need for programming languages to have professional-grade data management libraries.

Currently, there exist many different solutions to accessing a database from a programming language and many of the most commonly used programming languages, such as Java, C#, Python, Ruby, and PHP have more than one solution for this problem. The most basic approach to communication with a database from a programming language is creating a connection with a database and passing the database strings in the SQL query language to request data, or change in the data, from the database. Result data then has to be managed by the programmer. In the Java Database Connectivity API (JDBC), for example, programmers have to use the right methods to extract values from a *ResultSet*, which the query execution returns. The programmer has to know details of what the *ResultSet* is expected to contain and how to extract the content appropriately.

Other, more sophisticated, approaches to communicating with a database include Object-Relational Mapping (ORM) tools, in which the developers map tables to classes, making each object of that class one row of the table. Then, at runtime, the ORM tool takes over extracting the query results to individual objects. While this approach seems more comfortable and the ORM tools typically take over multiple important tasks such as caching, lazy loading, and query generation, issues with this approach have been reported in the literature. For example, the mapping between object graphs as might be typical in object-oriented programming languages and relational database tables can become cumbersome and badly configured ORMs can be prone to poor performance. [CJ10]

The complaints about different database access systems are at least as numerous as the systems themselves, one prominent blog post even calls the ORM approach the "Vietnam of Computer Science" [vie], stating that the approach is promising at first and for simple problems, but can easily produce more work when trying to cover all use-cases, creating a slippery slope similar to the involvement of the U.S. in the Vietnam war. The author, Ted Neward, states that this problem is stemming from the mismatch of the relational and object oriented data structures and recommends the use of other approaches, such as using object oriented databases, manual mapping, or the integration of relational concepts into general purpose programming languages. Criticism like this creates doubt on if this kind of tool is appropriate to use as a solution to accessing databases from programming languages. There have been a number of studies on how to make interaction with databases more easy for most users [JH16, BBE83, YS93, Sme93]. However, most of these studies have been focusing on what a stand-alone query language should look like and less on what the general purpose programming language to database interface should look like.

1.4 Scientific Contribution

The general goal of this document is to investigate the productivity impact of polyglot programming. The main driving question of this research is: *"Is there a measurable impact on productivity when developing software using multiple languages at once?"*. The software engineering literature suggests that there might be broad benefits to using specialized languages [KMB+96, vD97, VDK+98, VDKV00, Kru92, HB88], while linguistic theory implies that there might be a cost to switching between languages on the smaller scale. Therefore, this dissertation aims to investigate whether switching effects between computer languages exist and, if they do, test how big the effect is on programming productivity by testing computer language switching in empirical experiments.

Polyglot programming is a complex process and has effects on business and personal levels. On a business scale, interactions between developers and their knowledge about programming languages can be more important than the individual experience of developers [Fje08]. As this main driving question is too broad to answer conclusively in the span of the proposed research, and because this research is also driven by the attempt to inform library design, individual experience and productivity will be the focus of this research.

Chapter 2 will review existing knowledge on the topic of polyglot programming, programming language related productivity, linguistic research regarding code switching, and database access approaches, followed by a deep dive into type system research with a focus on productivity in combination with a study on enumerators to synthesize a principle to better understand how information in programming languages affects developer productivity in chapter 3.

With the focus being on the developer level productivity impact, a the first research question can be defined as (RQ1) *"Is there a measurable productivity impact when programmers switch between computer languages?"* This question is inspired by the observation in linguistics research that there is a switching time when speakers code-switch, both in understanding natural language [Ols17] and natural language production [Ols16]. If there are close ties between the way humans understand and produce code and the way humans understand and produce natural languages, then a similar effect might be found in computer languages. However, the nature of the similarities between natural and computer languages and the way they are processed in the brain is not clear in the literature and therefore is being tested in two of the experiments in this document. This question will be investigated in the experiments in chapters 4 and 5.

A part of this research question is also the investigation whether it is better for productivity to rarely switch or to switch back and forth repeatedly. Research suggests that both experience in switching languages [BMD17] and a setting in which language switching is common [Ols17] can reduce the cost of switching. If this can be confirmed for computer languages, then it might be beneficial to work in a quick back-and-forth switching pattern, while, if this is not confirmed, keeping switches rare might be preferable. From the results of the research conducted to answer these questions, recommendations are made on how to structure software projects to ensure the best productivity for software developers.

The second research question asks: (RQ2) *"Do programmers consciously experience switches between computer languages?"*. With investigation of this question, it is hoped that more insight can be gained into whether programmers realize that they are switching languages or if this is a mostly unconscious process. Similar to RQ1, this question will be addressed in the results of the experiments in chapters 4 and 5.

The third research question is: (RQ3) *"Is there a difference in productivity between participants who speak English natively and those who do not?"* This question is not directly related to the impact of programming language switching, but was inspired by the opportunity to collaborate with scholars from different countries. However, it might be possible that the answer to this question might uncover connections between how natural languages and computer languages are connected. Apart from that possibility, the answer to this question might have implications for computer science education around the world. If there is a difference to be found, it might be necessary to create curriculum with this finding in mind or make versions of programming languages that cater to different language needs. As the other two research questions, this question will be addressed as part of the experiments in chapters 4 and 5.

Further, in chapter 6 the design of a data management library will be proposed. Its design will be informed by the results of the research conducted to answer the main research question, as well as by the research literature. The aim of the library is to give users an easy-to-use and unified tool to manage data

from relational databases, as well as from structured data files. The library will be designed to manage typical CRUD operations and will be designed to be usable with data-reliant libraries, such as statistics packages.

Finally, the findings will be summarized in a conclusion chapter in chapter 7.

1.5 Methods

The first important step in this research endeavor is creating a good overview of the existing literature related with polyglot programming. For this, research in the areas of software engineering and linguistics has to be taken into account to inform the application of existing theories to the problem and to enable formation of informed hypotheses and explanations of results within existing models, as well as possibly adjusting existing models based on new information.

The main method of research for this project will be the use of human-centric randomized controlled trials to gather evidence on the effects of polyglot programming. To conduct the studies, the **E**mpirical **P**latform for the **I**nternet (EPI) was developed by Patrick Daleiden and myself and is continuing to be developed by Patrick Daleiden in the course of his dissertation. The platform enables the distribution of an experiment to a larger number of participants while recording many details about the participants' behavior, such as snapshots of the participants working on the tasks.

1.6 Structure

The rest of this dissertation will be structured as follows: Chapter 2 will lay out the relevant research on polyglot programming, domain specific languages, computer science education, programmer productivity, linguistics pertaining to polyglot programming, as well as database language research. Chapter 3 will focus on the impact of type annotations on productivity. In chapter 4 the first experiment on polyglot programming and programming language switching will be reported on, while chapter 5 reports on the second polyglot experiment regarding file level switching. Chapter 6 will focus on the the proposed design of a database access library. Finally, the dissertation will be concluded in chapter 7.

Chapter 2

Literature Review

2.1 Polyglot Programming

The term Polyglot programming is said to have been first used in a blog post by Neal Ford from 2006 [pol] in which the author explains that he is seeing a trend toward using more than one programming language in a project so that hard problems can be solved with tools deemed appropriate. Since then, polyglot programming has been repeatedly mentioned in the non-scientific software engineering community [EV12, For08], partly as a strategy for project design.

As mentioned by Neal Ford in his blog post, the use of multiple languages in a project is not new. The first mentions of the practice that were found in the context of this literature review are from the early 80's [DLGR81, EG84, Vou84] under the name mixed-language programming and has since also been called multi-language programming [KWDE98, CGHM06].

The seemingly first paper to address the issue was published by Darondeau et al. [DLGR81] in 1981, which is citing a paper from 1970 [Lan70], which in turn proposes the idea of using specialized languages for programming problems, because "Despite many attempts, no programming language has yet been devised which is suitable for all programming tasks." Darondeau et al. propose a system to enable static type checking across languages to help make mixed language programming possible. Another reason for language mixing is given by Einarsson and Gentleman [EG84] in 1984: To preserve and reuse already written code in other languages to avoid having to rewrite already implemented algorithms. The problem of duplication of effort is a problem that Stefik and Hanenberg state in their 2014 essay "The Programming Language Wars: Questions and Responsibilities for the Programming Language Community" [SH14c] is still too common in the current software development landscape, even though cross-language development is technically possible in many cases using technologies like Java Native Interface [jni].

There is no commonly accepted definition of polyglot programming, but it has been defined as *"programming in more than one language within the same context, where the context is either within one team, or several teams where the integration between the resulting applications require knowledge of the languages involved"* in a 2008 master's thesis by Hans-Christian Fjeldberg [Fje08]. This definition captures polyglot programming adequately on a business level. However, for the purposes of this dissertation proposal, the definition is too focused on team dynamics and not inclusive enough when it comes to other contexts. When considering the findings of a case study of a single large open source project that the majority of committed changes to the project contained code in two different languages or more, especially when adding new features [VTTM12], then one has to consider that polyglot programming also has a more personal, developer level side to it. This finding implies that one developer has to know at least two languages, and will have to be able to switch between said languages to finish their work. According to a survey of professional developers by Meyerovich [MR13], programmers self-report to know ten languages on average, six of which they report to know well. The pervasiveness of this can be seen in two repository analyses on GitHub projects, one of which [TT14], analyzing 15,216 different projects, found that 97% of projects used two or more computer languages. The other study [MB15] found in their analysis of 1150 projects that there were 24 general purpose languages and 31 domain specific languages in use. Between the two studies in [TT14] and [MB15], it appears that the average amount of languages in an open source project is about five.

The benefits and drawbacks of polyglot programming on a human-factors level are under-explored in the scientific literature. The main argument in favor of polyglot programming is about *"[...] choosing the right tool for the job"* [pol] as stated in the term-coining blog post by Neal Ford. Fjeldberg argues in his thesis [Fje08], investigating the business perspective of polyglot programming by conducting a literature review and case study, that the use of a more appropriate language for a task leads to better productivity and easier maintenance by reducing the lines of code of a project. The drawbacks of polyglot programming listed in his thesis pertain to the knowledge a developer needs to develop using multiple programming languages, the lack of tool support for polyglot programming, as well as possible impacts on maintainability by reducing the pool of developers who have the skills necessary for maintenance tasks in all the languages used. To further investigate the advantages and drawbacks listed, he conducted three case studies that qualitatively confirmed the improved productivity and maintainability. He notes that developers seemed more motivated by the switch of languages, but gives no concrete evidence for the claim.

The domain specific language community is embracing the concept of polyglot programming by developing DSLs for specific purposes, to make solutions more concise and their writing faster at the cost of possibly having to develop a domain specific language first [Hud97]. Hudak's summary of domain specific language development [Hud97] also asserts that domain specific language can allow domain experts to solve problems by themselves without being trained in general purpose programming because a DSL should be

understandable and usable by a domain expert. He mentions that DSLs are easier to reason about than general purpose languages and also lists easier maintainability as a possible advantage. These are claims that are shared with polyglot advocates. He also lists steps for a DSL software development method in which the domain is defined, then a DSL and tools to support the DSL are designed and developed and then the DSL and tools are used to create applications to solve problems. Further, Hudak recommends the use of embedded DSLs which rely on the infrastructure of a given host language to reduce development and learning effort when developing DSLs, since a given DSL would be close to the original language and so a programmer proficient in the given language could easily pick up learning the new language —a benefit that would be lost on the non-programmer domain expert.

A summary of the DSL literature from the year 2000 by van Deursen et al. [VDKV00], additionally to the previous claims, mentions that DSL programs are mostly self-documenting and enhance productivity, reliability, maintainability, and portability, enable the conservation of domain knowledge, allow for validation, and improve testability [KMB+96, vD97, VDK+98, VDKV00, Kru92, HB88]. A systematic review of DSL literature found that validation of DSL claims is rare, however, analyzing 390 studies between the years of 2006 and 2012, of which only 3.3% studied the effectiveness of DSL approaches [KBM16].

For the search of what impact polyglot programming has on productivity, the claims of enhanced productivity are the most interesting. They were tested in the form of a study by Kieburtz et al. testing four developers using a DSL for program generation and a tool that uses Ada program templates [KMB+96]. The experimenters trained the subjects in both technologies and gave each developer 34 different tasks, 17 in one technology and 17 in the other. In their data recording, both through self reports of the developers and recordings on their workstations, they found that there was a significant difference between technologies. All four subjects spent about twice as much time developing using the Ada template generation technology than they spent on using the DSL. Despite the low number of participants in this experiment, this experiment seems to show that the use of this specific DSL was useful. However, the design of the experiment does not take into account development time for the DSL, which already existed, or learning time because the developers received training before starting the experiment. The experiment also does not take into account possible switching between languages as a factor in productivity.

2.2 Education

Tied into the process of creating a better understanding of polyglot programming is the aspect of learning programming languages because, to be able to partake in polyglot programming, a person first has to learn to use computer languages [Fje08]. Research has found that novice programmers are impacted by aspects of programming such as the syntax choices of the languages they are learning [SSSS11, DLRTH11] and the

compiler errors they are encountering [Tra10, Bec16a] to the way programming is taught [NXK17]. This section will explore parts of the programming education literature to better understand the effort required to learn programming and to learn additional programming languages.

Universities find that there is a drop-out rate of 20-50% in their first computer science classes [Yue07]. Further, students that do finish the first programming class seem to be underperforming compared to the expectations of educators. A trial assessment of the programming skills of students in 4 universities in different nations found that students who finished their introductory programming class significantly underperformed compared to the expectations of the assessors, achieving only 20.81 % of the possible points in the assessment on average. The study conducted by McCracken et al. [MAD+01] attempted to create a standardized test in the form of a set of tasks to be finished in 90 minutes. The research was motivated by the concerns of professors who were noticing a lack of programming knowledge in their students. The researchers found that there seemed to be many students who could not finish the tasks successfully. The paper broke down the scores of students by 1) if the program executed, 2) if the program handled input correctly, 3) if the program solves the task correctly, and 4) the style. The highest average score by percentage was the the style score with 46.2%, while the execution score was 23.9% of the possible. The other two scores were below 5% on average. This indicates that students might have internalized the style of how their programs are supposed to look, but not to make their programs run reliably or how to correctly solve the problems given to them. The researchers assert that the apparent high number of students that did bot know how to proceed with the tasks indicates that they might have needed more time in their course to focus on the skill of abstraction of problems from a task description. They also found that low scoring students tended to blame their failure more on external factors, while higher scoring individuals tended to blame their own shortcomings. The researchers note, that there might be many variables that might have influenced the assessment, such as difficulty of the tasks and the given time constraint. There were also some difference in how the assessment was conducted in the different schools, which makes comparing the results more difficult. For example, one teacher gave students a similar task to prepare them, while others didn ot.

Understanding the task solving behavior of students can help with contextualizing qualitative analysis of programming task progress and give an impression of the issues students face when starting to learn programming concepts. To explore the problems with learning to program, Yuen [Yue07] interviewed 8 students in connection with them learning the three programming concepts functions, recursion, and arrays. The information from the interviews, the work of the students from working on the tasks, and the researcher's notes was analyzed from a cognitive learning theory perspective to uncover how students thought about the programming concepts and how knowledge was constructed. The researcher observes that the students have the most trouble with concepts that they have difficulty conceptualizing. When they rely on automatic knowledge, they have problems applying the learned information on unfamiliar knowledge and when they

rely on associated knowledge, they encounter difficulties separating the association when necessary, such as in the example of always associating arrays with loops, even when they have to use recursion to iterate through an array. Further, Yuen notes that students tended to start coding immediately, without first planning their approach, even when they stated that they usually do. From a language perspective, an interesting observation was that students also tended to want to express their code in syntactically correct Java —the language they were learning —even though it was not required by the experimenter, who was happy to let them express the algorithms in English descriptions or pseudo-code. One student responded to the question why they preferred the programming language approach that it was hard to put the program into words.

Another qualitative study with novice programmers investigated the task solving behavior of students that are confronted with increasingly harder programming tasks, showing with which aspects of programming they had trouble. In the study, Whalley and Kasto [WK14] conducted a think-aloud experiment with six programming students using Java to control a robot on a grid system. The tasks posed to the students started with a task analogous to a previously covered problem in their class and later tasks expanded on the idea, but required the students to combine concepts to be able to solve them. One of the students completed all problems without intervention and one other student only had to be helped once. Two other students had to rely on some redirection and scaffolding by the researcher. Of the last two students, one was able to solve two tasks without intervention but gave up in the third task when the researcher intervened. The last student was able to solve the first task with some help by the researcher and gave up on the other two tasks. The researchers characterize the students as tinkerers and stoppers and movers, as well as planners. The students giving up the tasks were showing random tinkering behavior right before stopping, showing that they did not reason about the programs behavior. All four other students were characterized as movers as they utilized the help offered to them to move along to finish the task. The top two students also exhibited planning behavior during the tasks, which helped them solve the tasks mostly without intervention. While the study presented in this document is not using a think-aloud protocol, the insights on student task progress in Whalley and Kasto's research might be applicable to qualitative analysis of code snapshot sequences.

Compiler error messages can be a significant problem when trying to learn programming [Tra10]. A study by Becker [Bec16a] argues that a majority of a novice programmer's time is spent repeatedly trying to fix similar errors. Becker empirically evaluates a tool giving improved compiler messages alongside normal Java compiler messages and shows that a group of students using this tool encounters significantly fewer errors throughout a four week period. This shows that errors in current compilers can be a significant hindrance for novices that could be alleviated with improved tools. It seems logical that this can also be an issue for programmers attempting to learn multiple languages. In a later publication [Bec16b] Becker proposes the use of a new metric, repeated error density (RED), to measure repetition of compiler error messages in sequences

of compilations as an indicator for programmers struggling with their tasks. This measure might also be a good indicator for programmers struggling with learning multiple languages and could be applicable for additional analysis of experiment outcomes.

The body of work on the acquisition of secondary languages is sparse. However, there is at string of studies on the issue by Scholtz and Wiedenbeck. One think-aloud study of theirs [SW90] focused on analyzing the process of 5 higher level university students who had been assessed to be highly skilled programmers. The participants' utterances were categorized to one of three concerns while the students were trying to program in an unfamiliar programming language: syntactic, semantic, or planning. The researchers found that the subjects were concerned with syntactic and semantic topics for about 40% of the time while the rest of the time was spent on planning activities that were divided into three types of planning. The first type of planning was big picture strategic planning concerned with solving the overall problem. The second type was tactical planning, which was concerned with solving local problems such as planning an algorithm. The last type was implementation planning, in which participants were mostly concerned with finding the right programming language structures. The researchers found that programmers seemed to handle syntactic and semantic issues easily. The biggest problem the researchers found was that the subjects would relate to the programming languages they know more than to the language they were using, neglecting to learn about the advantages of the new language and instead trying to use problem solving approaches from their primary programming language, ignoring the facilities of the new language. This lead to inefficient programs and the researchers note that the primary language focused thinking lead to the participant getting stuck with their formulated plans because they weren't executable in the current language. They suggest that materials for teaching a language to proficient programmers should list information on syntax and semantics, but especially focus on the strengths and weaknesses of a language using examples to show how to use the built in features of the language effectively.

A follow-up study on the same subject [SW93] studied 15 Pascal programmers as they attempted to solve a programming problem using either Ada, Icon, or Pascal within the span of two hours. Because all participants in the study were Pascal programmers, the Pascal group was considered a placebo group. The researchers recorded the developers and their screens and evaluated the resulting program. The analysis included the determination of the developers' performance by judging their solutions, as well as a protocol analysis of the think-aloud data to investigate the process the developers used. According to the researchers, the Ada group and the Pascal group were most alike in the strategy required to solve the problem, while the Icon solution was flexible and either allowed for a solution similar to the Pascal strategy or a solution unique to the language. The Pascal group was most successful, both in time and by fully finishing all subtasks, the Icon group came in second with only 10 more minutes of time needed to solve the task and almost all sub-tasks finished on average. None of the Ada group's programmers finished all sub-tasks and the average

amount of sub-tasks finished was about 60%. The researchers mainly attribute the unfamiliarity of syntax in Ada and the inflexibility of the language. In the protocol analysis, Scholtz and Wiedenbeck focused on how the subjects' plans developed throughout their process. They report that programmers spent about 65% of their time on planning, while 21% of the time was spent on syntax and 14% on the semantics. They found that the developers tended to try to use strategies they learned using Pascal. This worked well in the more high-level approaches to solving the problem in Ada, but the experimenters found that there were many changes in plans on the detail level in the Ada group. On the other hand, the developers using Icon were found to more often change their plans on a higher level when they changed from their initial, Pascal-inspired, plans to ones fitting the language they were currently using better. Overall the researchers found that there were differences in performance when using an unfamiliar language and that programmers tended to base their solution strategies mostly on the language they are most familiar with.

The investigation of programming language adoption by Meyerovich et al. [MR13] using data of open-source projects and developer surveys on popular developer forums also found relevant information on additional programming language learning. Their data shows that developers keep learning programming languages throughout their careers and that there is a 20% chance of learning a new language per project. Respondents reported that they learned ten languages on average and when asked to list languages that they know well, they listed an average of six languages. On the relationship between the kind of languages learned in the programmers' education and the languages they know at the time of the survey, it was found that a developer who learned languages from different language families during their education was more than twice as likely to know more diverse languages in their career. The survey also found that developers rate the learnability of languages differently. Learnability was measured on a 5 point scale, each point indicating a different time period. Rated the most easy-to-learn general purpose languages were Python, PHP, and Ruby with a time-span of one to three months. Meyerovich et al. remark that this is a curious finding since PHP is "notorious for its ad-hoc design". They speculate that developers rate the language as learned when they can use the subset that they need, even if they have not learned all parts of the language and that complexity in the eyes of researchers might be different from complexity in the eyes of developers.

2.3 Evidence on Programming Language Productivity

Since the focus of this research lies in finding information on programmer productivity using randomized controlled trials, a look at existing studies on the topic of programmer productivity is warranted. This section will therefore introduce the reader to some of the existing literature on the topic.

Type systems are the aspect of programming languages that have been studied most thoroughly using controlled experiments [Kai15a]. The first study on this topic is the controlled experiment by Gannon from 1977. Gannon investigated the difference between a statically typed language and a typeless language, both

designed for the experiment, and measured the errors that his 38 participants produced while solving the programming tasks. He found that participants using the statically typed language produced fewer errors than participants using the typeless language. He also found that these results especially apply for less experienced programmers. The author points out however, that the results are not surprising as the typeless group only had access to the data type *word* and connected operations, while the typed group had access to *integer* and *string* and the operations connected to those types. Thus, the experiment does not compare dynamically typed and statically typed languages, but rather a higher and a lower level programming language.[Gan77].

A study in 1998 run by Lutz Prechelt and Walter Tichy [PT98] compared ANSI C with K&R C. The difference between the two languages is that the ANSI C compiler does type checking for external function calls, while K&R C does not do that kind of checking. In this experiment with 40 participants, most of which were graduate students, the researchers found a significant difference in favor of the more type checked language for the measure of time to completion of the program for the second task. With this experiment being designed to be counter-balanced, the second task was a similar to the first task using the language the participants did not use in their first task. The researchers explain this finding by stating that participants likely had to get used to the concepts in the first task and that an effect was therefore not found until the participants were more familiar with the library in use, at which point then the difference between the type checked and the not type checked groups becomes detectable. The researchers also state that this explanation was confirmed by an investigation of the compiler inputs. Based on this explanation, the authors further reason that type checking itself does not seem to increase understanding of a program. Further, the experiment finds an advantage for ANSI C when it comes to productivity, number of defects introduced and number of defects in the final submissions.

To investigate the difference between statically and dynamically typed programming languages, Hanenberg ran an experiment in 2010 [Han10b] using two Smalltalk-based languages he developed for the purposes of the experiment. The 49 participants of the study came from a population of students that predominantly learned Java in their education. The experiment was designed as a long term project with multiple sessions. The results of the experiment show a significant negative impact for the statically typed language group when comparing development time to create a minimal scanner and no significant difference in the amount of test cases that were successfully passed by the final submissions. The author argues that the advantage in development time for the dynamically typed group likely comes from the fact that the project is rather small and that conventional wisdom about the benefits of statically typed programming languages states that it becomes a more effective tool in larger projects. This experiment started a series of follow-up studies on the topic. The next paper published in this series was authored by Andreas Stuchlik and Stefan Hanenberg [SH11a] in 2011. The researchers studied the difference in development time between a group

using Groovy (as the dynamically typed language) and Java in tasks with different amounts of type casts. The study found significant positive effects for the dynamically typed group, as long as the tasks remained somewhat short. Results for the longer tasks in the experiment did not show significant differences between the groups. These two experiments spawned a series of similar experiments on the topic of type systems and their relation to developer productivity, the results of which are quickly described in the next paragraphs.

A study by Mayer et al. [MHR+12a] compared the performance of a group using Groovy with the performance of a group using Java to investigate the differences between static and dynamic type systems. The 27 participants were to work with an API that was newly created for this experiment and that was not documented. The results show that the productivity was higher when using the static type system in tasks that were deemed harder by the researchers, while dynamic type stystem use seemed to give an advantage in easier tasks.

A follow-up of the topic studied in Mayer et al., Hanenberg et al. [HKR+14a] tested Groovy against Java again and found in a study with 33 participants, that static type systems made developers more productive when using previously unknown classes. Further, the study found that fixing type errors was made easier when using a static type system. When it comes to fixing semantic errors in code, there was no significant difference found between the two languages. The study also found that the number of switches between files that a participant made in the tasks in which static type systems were more effective, was a good indicator for the time results in the same tasks.

Petersen et al. [PHR14a] replicated the results of Hanenberg et al. [HKR+14a] while allowing participants to use a state-of-the-art IDE and providing typical JavaDoc documentation. The participants were to solve some of the same tasks as in Hanenberg et al. [HKR+14a]. This replication upholds the results from the previous experiment, showing a productivity advantage for developing using static type systems. Another study on the connection between tool support and static type systems was conducted by Fischer and Hanenberg [FH15a] and found code completion did not significantly impact productivity in an experiment comparing type systems and groups with and without code completions.

To further study the effect of static type systems on developer productivity, Hoppe and Hanenberg [HH13a] studied the effect of a different type system feature, generics, compared with the absence of this feature, raw types requiring type casting. In this experiment with 16 participants, the results show that there is a productivity advantage in using generics when using unknown, undocumented code. When investigating tasks involving fixing type errors, no difference was found, and developers were less productive using generics when trying to complete tasks involving changes to a strategy pattern.

As there seems to be a positive impact on productivity when using static type systems, an important question is what this impact is caused by. Spiza and Hanenberg [SH14a] studied the impact of explicit type names on developer productivity by conducting an experiment using the Dart programming language, which allows for optional type checking. The treatment in this experiment was the presence of type names in the code. Although type names were present in the code, the type checker was not enabled. The experiment found that participants with access to type annotations were significantly more productive than when the type annotations were not present. The experiment further found that subjects in the experiment would rely on the correctness of the type annotations and would perform significantly worse compared to subjects not using annotations when the experimenters deliberately introduced an incorrect type annotation in one task.

An investigation into the effect of documentation and type systems on developer productivity when using unfamiliar code conducted by Endrikat et al. [EHRS14a] found further evidence that static type systems have a positive impact on productivity. A general significant positive impact for having documentation could not be found (p=0.075), as the results did not meet the typical statistical threshold of $\alpha = 0.05$.

Other efforts in investigating ways to make programmers more productive include research into API usability. A study by Jeffrey Stylos and Brad A. Myers [SM08] shows an decrease of development times by factors from 2.4 to 11.2 when an action method necessary to complete a task was part of the object that was the main focus of the task, instead of being part of another object. The most prominent example given by Stylos and Myers was the use of a *send* method on a *mailMessage* object, of which the properties also had to be set, instead of on a *mailServer* object that was also involved in the transaction. Similar to this finding, a study by Ellis et al. [ESM07] found a significant increase in solution time when participants were forced to use the factory pattern to create objects instead of using constructors for the same task, suggesting that this pattern should be avoided unless absolutely necessary.

A case study of 26 million build errors at Google [SSE+14] by Seo et al. for Java and C++ builds investigated the productivity impact of the Google build work-flow. The investigation found that 37.4% of C++ and 29.7% of Java builds failed and that C++ programmers made about 10 builds per day, while Java programmers made about 7 builds per day. Seo et al. found that it took an average of 5 minutes to resolve C++ build errors, while it took an average of 12 minutes to solve Java build errors. When it comes to types of errors, the researchers found that 10% of build errors cause 90% of all build failures and the most common errors encountered were dependency errors. The researchers remark that these build errors seem to have a significant impact on the productivity of developers at their company and that tool engineers can use the information from this case study to improve tools to reduce the number of build errors in the hopes

of improving productivity.

2.4 Code Switching

Research into the time cost of switching between computer languages in polyglot programming as proposed in this document is inspired by the phenomenon of code switching and the mental cost of switching natural languages. As there are possible connections between how humans process natural and computer languages, this section will present some of the literature on the topic of code switching that might inform the research on polyglot programming and computer language switching. To frame the research of code switching, consider that 21% of people in the U.S. [Rya13] and up to 66% of the world population are multilingual [MS]. As a first part of this section, the general phenomenon will be explained in subsection 2.4.1, then, cognitive models for code switching will be explained in subsection 2.4.2, and lastly, subsection 2.4.3 will discuss the relation of code switches to polyglot programming.

2.4.1 Code Switching in Natural Languages

In the field of linguistics, code switching is defined as the meaningful juxtaposition of two language systems. This can mean a speaker switching between two or more languages in a conversational context. This phenomenon can appear in-between utterances, which is known as inter-sentential switching, such as a switch between languages between sentences, for example "I'm going to see Star Wars tomorrow. *Willst du mitkommen?* ('Do you want to come with?')". Code switching can also appear within an utterance, which is known as intra-sentential switching, i.e. a word or phrase in an English sentence is changed to German [HA01, Li96, YTF17]. An example for this could be "I went to *Gymnasium* ('a type of German high school') between fifth and 13th grade". Code switching is typically observed among multilingual people who are speaking an overlapping set of languages and typically only in the languages that all participants of the conversation understand [YTF17]. The switches are overwhelmingly grammatical, i.e. adhering to the grammatical rules of the languages in use, and lexical [YTF17] and can occur in a wide variety of contexts such as in the classroom while learning languages [LDO05] or in every-day contexts such as in conversations between friends similar to the example about going to see Star Wars above.

There are different reasons that might cause a speaker to code switch and one them is the use of a code switch to more accurately convey a meaning that could otherwise not be easily translated. For example in "I went to Gymnasium ('a type of German high school') between fifth and 13th grade", the speaker might have to elaborate to a listener who does not know the German language and culture on what the word means, because it does not directly translate to high school as it incorporates a wider range of years and is distinct

from the other three forms of secondary schools in the German school system. This would be an exception to the aforementioned pattern of code switches between speakers that can speak both languages. However, when speaking to a person with a good grasp on the German language and culture, then this would be a typical code switch, which would give the listener a very specific understanding of what kind of school the speaker attended in their childhood.

Other reasons for code switching can be based on the surrounding culture and the participants in the conversation. In some contexts, switching between languages fluently is a normal part of discourse such as in Spanish-English bilinguals in the United States and might be part of the identity of the speakers. For example, a mother might say to another: "I still need to bring a dress for her quinceanera [a party held for fifteen-year old girls in Hispanic countries] party". In other contexts, speakers might switch between languages to accommodate the language preferences of other participants in the conversation [YTF17], which can be based on the social circumstances of the conversation, such as the power relations between participants of the conversation. An example in [Wei05] illustrates this use with a mother who uses a code switch to assert her authority when speaking to her child about homework. Other researchers have found that speakers might use code switches to change roles in a conversation or to draw attention [YTF17]. An example in which a girl asks her mother for money in English while having spoken Chinese before [Wei05], shows the use of code switching possibly explained by attempts to distance oneself from a statement, as there is a different level of emotional connection to a statement in the switched-to language (see also [ASR94]). Code switches can also serve as a device to distinguish story from commentary, as seen in an example between children of similar age in which one of the children is trying to tell a story in one language while her friends interrupt in another language [Wei05].

Lastly, reasons for code switches can be instances in which a speaker cannot recall the word in the language they are currently speaking but can recall the right word in another language. For example, a German-English bilingual might say "I'm going to the *Kirmes* ('a fair, typically with rides and fun activities')". Cases in which this behavior had been observed shaped an early view in linguistics that code switching is a sign of poor language skill [HA01, YTF17]. This view has been increasingly challenged in the recent past. For example, in bilingual children, studies have found that children who code switch use the interjected words or utterances in a way that is consistent with the grammatical structure of the language. A study by Yow et al. [YTF17] on bilingual children found that frequent code switching improved their language skills by enabling them to "explore and use both languages (the weaker language with the stronger one) while keeping the intended meaning intact". Yow et al. [YTF17] further argue that this might make code switching a powerful tool for language learning in general, a hypothesis that was also tested in an advanced university-level language course where students were allowed to code switch. Students were found to engage more in course discussions, allowing them to express more complex thoughts utilizing code

switches when they felt they could not express the thought sufficiently in only the language the course was being taught in, while still developing their language skills [LDO05].

Testing code switching in a lab setting can be difficult as it is a phenomenon usually appearing in natural conversation. To try to induce code switching in participants of a study, Kroff and Fernández-Duque's [KFD17] tried to involve them in a natural appearing conversation in the form of a game in which they had to coordinate with an experimenter. By the way of the game, the researchers hoped to guide the conversation and introduce specific words. Both the researcher and the participant were bilingual Spanish-English speakers and the researchers encouraged code switching in the conversation by code switching themselves. The results of the experiment show that this type of conversation can be successful at inducing a participant to code switch as well, though it appears that the code-switching patterns resulting from the experiment depend on each participant's background of language usage and code switching experience. The researchers note that there seemed to be a component of non-verbal negotiation between speakers, i.e. a participant's readiness to code switch within the context of the game had an influence on the code switching behavior of the researcher. When a participant preferred to switch between languages less regularly, the researcher would adapt to that pattern of speech and mirror it in their own code switching behavior, even though they were supposed to switch often.

2.4.2 Cognitive Cost of Code Switching

It is assumed that language processing works on the basis of a conceptual thinking apparatus that is independent from, but connected to, different lexica for each language that a particular person speaks [KS94]. Research on this topic is still ongoing, and the exact nature of the connections between the conceptual system and the lexica and between the lexica themselves is not clear. It has been suggested that some words, like nouns for specific material objects, are connected in the lexicon, while others, like words for abstract concepts, are less likely to be, because they might not translate well between languages [HA01]. It is reasonable to assume that there is such a thing as a dominant language, often called L1, which a person is more proficient at than a second language (L2) and lexicon of which is more readily accessible to the speaker [KS94]. This was assumed to typically be the first language a person learned; however, some research suggests that the dominance of a language can change over the course of a person's life based on their frequency of use of that language [HA01].

There are a number of models that try to describe the mental process of switching between languages and make predictions about the cognitive cost of switching. The first model described the switching between languages in a way in which only one language could be active at a time [Ols16]; however, this model was revised because it was inconsistent with the abilities of bilinguals to translate speech into one language while

hearing it in another. The revised model proposes a two-switch model in which a bilingual has a switch for which language they produce and one for which language they understand [MK71]. In this model a bilingual person can use the switches independently from each other. The researchers proposing this model suggested that the trigger for switching between words when listening is typically on the phonetic level, i.e. triggered by patterns of sounds that typically belong to one language or another, while the switch for speaking is typically under conscious control of the speaker. Later, the Inhibitory Control Model (ICM) was developed, which considers the switches between languages to be based on inhibition of one language system [Gre98], the more inhibited, the longer a person takes to switch to that language. Recently, this model has been refined by considering the level of inhibition as a gradient that can be affected by the speaker's context [Ols16]. This model predicts that there has to be a time cost associated with the switch between languages by the way of the switch of inhibition. This is a phenomenon found and replicated by a number of different studies on the topic, many of which have been trying to identify the relationship between language dominance, context, and phonetic differences between the languages and the time to switch for participants.

2.4.3 The Relation of Polyglot Programming and Code Switching

The switching between natural languages can be considered a parallel to the switching between programming languages when programming. The closest match to the concept of intra-sentential code switching is the act of switching on an embedded level (see section 1.1), where a programmer might have to switch computer languages within a line of a program. The closest match to inter-sentential code switching in polyglot programming is likely the quick back-and-forth switching on the file level of computer language switching. The slower switching on the file level and switches on a project level are more comparable with the concept of language switching in linguistics [AG08].

In contrast to the different reasons of code switching in natural languages, in programming, switching between computer languages is mostly confined to a context in which it is necessary. The communication with a database for instance, unless other tools are creating an abstraction layer, requires the use of the database's query language. In that case, a programmer who does not have access to a library to abstract the query writing process away from the query language, has to write the query in the given language and is forced to switch. This difference between natural language code-switching and polyglot programming becomes less defined when the programmer has the option to use another language without the need to do so to achieve their goals. The programmer might choose to introduce a new language when they feel that a task might be better served using this new language, introducing new situations in which the developer will have to switch. However, this introduction of a new language requires more time on the side of the programmer than the use of a code-switch in a conversation. One important fact to keep in mind when comparing code-switches and polyglot programming is that in the case of programming, the conversation

partner is a compiler or interpreter and not a human being.

While designing a programming experiment as a conversation might be difficult due to programming's imperative nature, the induction of switching by example as seen in Kroff and Fernández-Duque's [KFD17] experiment could be replicated in a programming experiment by showing a participant in which situations the switch would be acceptable by giving code examples of similar situations.

The dominance of a natural language might have a parallel in the way programmers use programming languages. The observation in Scholtz's research [SW90] that developers would try to solve problems from the view of a programming language they are proficient in allows for the assumption that their most proficient language can be considered their dominant language. It might be called their computer language 1 (CL1). Which language is considered someone's CL1 is likely mostly determined by their work experience using that language and how recent their experience is. A person's dominant programming language might change more often than their dominant natural language. Another possible factor influencing a person's CL1 might be the first language they learned. [MR13] While a developer might consider one language as their dominant go-to language, it might be hard to rank the proficiency of other languages, especially when the languages serve very different purposes or are representatives of different language paradigms. A comparison of proficiency between Python and HTML would feel meaningless. Therefore, this proposal will distinguish between CL1, as a developer's primary language, and CL+, any further language a developer might know.

Further, it is important to remember that as of now, computer language learning is typically happens much later in a person's life than natural language learning among bilinguals, even though there are efforts to make programming education a staple of high school, middle school, or even grade school in the U.S., natural language learning occurs much earlier. Currently, proficiency over a programming language, while relative to interpretation of the word proficiency, is often only gained in university courses or later (see the results of [MAD+01] in section 2.2).

The aforementioned difference between a quick back-and-forth switching between languages during file level switches, i.e. switches between languages when switching between working on different files (see section 1.1), and the slower switches is informed by the finding that the importance of context is often stressed as a major factor determining the cost of switching languages. Yet, determining cognitive cost is not simple. One type of study that has been used are eye-tracking studies investigating cognitive delay when code switching by measuring how long it took participants to look at the right section of a computer screen found that while listening to a sentence in which code switching occurs, the time to comprehend the sentence was increased or decreased depending on the code-switching mode the participant was in, i.e. how much code switching had occurred shortly before [Ols17]. In a mostly monolingual mode, there were asymmet-

rical costs for participants, while there was virtually no cost in a bilingual mode. The asymmetrical cost in the monolingual mode shows higher cost for switching from L2 to L1 and lower cost while switching from L1 to L2, which is an observation that fits the predictions of the ICM model. According to the ICM model, the dominant language of a bilingual speaker has to be more cognitively inhibited to keep the speaker from speaking in it, which means that more effort has to be expended to reduce the amount of inhibition to switch to the dominant language. The observation that there was no switching cost in a bilingual mode was explained by there being a mostly balanced and low amount of inhibition while expecting either language.

A similar idea was tested when testing the differences in code switching behavior between two groups of Spanish-English bilinguals, one group with extensive code-switching experience and one with less experience. The researchers found that more experienced code switchers tended to switch more often and with less effort [BMD17], suggesting that experience with code switching plays a major role in code switching costs.

If a switching cost can be observed in computer language switching, a model similar to the ICM might explain the mechanics of the switch between computer languages. One important factor found in changing natural language switching cost was phonological structure when listening to speech, for example for English and Chinese speakers [Li96]. A possible close parallel to phonological structure might be the visual structure in source code, which could be impacted by choices made in syntax structure and syntax highlighting. Distinct visual representation for each language that is interacting might enable programmers to switch between languages more effortlessly. Possible ways to achieve that difference could be assigning a different font to each language, making all statements in a specific language the same color, or assigning different color patterns to each statement in a specific language.

2.5 Task Switching and Cognition

Other than the research in the field of linguistics regarding switching between natural languages, the field of task switching is highly relevant to the questions in this dissertation. As such, an overview of the state of the art of task switching research is presented in this section.

In task switching research, task switching frequency is presented on a continuum across several orders of magnitude of extremely common switching in the realm of seconds to switching between tasks in the span of hours [STB09]. Frequent and quick switching is called concurrent multitasking while less frequent switching is described as sequential multitasking [STB09].

The process of concurrent multitasking can be abstractly modeled using the ACT-R cognitive architecture [ABB+04, STB09], which is a model representing the human cognitive functions involved in the execution of tasks as a set of modules that interact with each other to successfully achieve a task. This

modular system uses a *procedural* module as a communications hub between the other modules, which have different functions such as storage of factual information, goal tracking, auditory or visual processing, and manual control. These modules represent resources that can be used independently, allowing for concurrent processing of different stimuli with the exception that the message hub, the *procedural* module, has to coordinate the communication between the other modules and creates a bottleneck, which can lead to delays in multiples of 50ms [STB09].

On the other hand, researchers of sequential multi-tasking employ the theory of *memory-for-goals*, which views different tasks as goals in memory. Only the strongest task in memory is the one an individual is working on [AT02, STB09]; therefore, to change between tasks, the goal of the new task has to get strengthened to be at the forefront, which is a process that takes time and creates delays between tasks. According to this theory, memories of task goals can also deteriorate, making it more difficult to switch back and forth. This theory of task switching is reminiscent of the inhibitory control model (ICM) found in linguistic research, in which a specific language has to be dominant in the cognitive system than other languages to be used, with the exception that in the memory-for-goals model, the task goals are strengthened and not inhibited and in the ICM, other languages do not deteriorate over time.

Salvucci et al. [STB09] attempt to marry both models, the threaded model of the ACT-R architecture and the *memory-for-goals* model by arguing that the higher level view of memory management in the *memory-for-goals* theory can be reflected in a reoccurring process of refreshing the memory of the goals of a first task while switching to and working on a second task. This reoccurring process then takes up a processing slot in the **procedural** module and in the memory modules and can cause resource conflicts which, in turn, leads to delays in task switching.

An interesting insight into these processes comes from a study by Trafton et al. [TABM03, STB09] in which participants either received a warning that they would have to switch between tasks or they had to switch immediately between tasks, then finish this secondary task and finally switch back to the previous task. Participants who could prepare for the switch were able to resume the first task significantly faster than participants who didn't have time to prepare for the switch This suggests that participants did exhibit preparatory behavior upon needing to switch, which helps to pick up the task more rapidly after the distraction. This behavior seems to be the process of finishing a subtask and memorizing the state of the task [STB09]. Immediately interrupted participants were found to sometimes utter reminders about the first task during the interrupting task concurrently with trying to solve the task [TABM03, STB09].

It appears as if software developers develop an intuitive understanding for this effect. In Meyer et al. [MFMZ14], in which the researchers evaluate the results of a survey on the topic of how developers

perceive productivity and the results of an observational study of 11 developers, developers express that they prefer to have fewer interruptions—or context switches—in their work days and that they prefer voluntary interruptions over involuntary ones. While the observational study shows that the developers actually switch contexts fairly often, an average of 13.3 times per hour, 72.7% of the observed developers were satisfied with their work productivity on the days of the observation. The researchers interviewed the participants on this issue and developers explained that self-imposed context switches are less expensive for productivity. Developers deciding to get up to get a coffee are likely to prepare for the task switch and might even use the coffee break to process their current problem. According to Meyer et al.'s observations, developers even often switch between tasks deliberately, for example when waiting for their build process to finish. In these cases, they typically choose short tasks such as quickly checking their email or reviewing code for 30 seconds.

When considering this finding from the perspective of programming language switching it is important to recognize that programmers who are needing to switch between two different languages would typically be able to take enough time to switch between primary and secondary task ("interruption lag") to prevent lengthy switching times from secondary to primary task ("resumption lag"), as the switching process is under their control, unless it is experimentally forced. According to a simulation using the ACT-R model by Salvucci et al. [STB09], having a very short —or non-existent —interruption lag can lead to loss of memory of the task and require lengthy reintroduction time while of more than one second, while recall of the task after having sufficient interruption lag takes about 116ms according to the simulation. The simulation also showed that remembering the first task during the interrupting task similar to the utterances that were observed in Trafton et al.'s study [TABM03], suggesting that, with practice, participants might adapt to immediate task switching to the extent that results become comparable to the group which has ample interruption lag, which is also consistent with the empirical results of Trafton et al.'s study. This is a remarkable finding which seems to match Olson's [Ols17] empirical finding that bilinguals seem to reduce switching times when they have adjusted to switch frequently in the given context and the finding by Beatty-Martínez and Dussias that trained code switchers exhibit less effort [BMD17] when switching.

2.6 Database Access Approaches

To give an overview of different approaches to access databases from general purpose programming languages, for the purposes of informing the design of the data management library, this section will some of the research in the field.

To see how easy SQL is to use and what users might have issues with, we can consult a number of empirical studies, such as a study on the differences between learning Query By Example (QBE), an alternative querying approach, and SQL [BBE83] showed that learning SQL didn't take as long as QBE, while just using the two techniques showed mixed results. One of the tasks in which QBE showed better results

involved joins between tables. In a survey after the experiments, most of the participants preferred SQL over QBE, participants praised the language for being well structured and "English-like", but criticized SQL for not having information about column names, how the 'IN' keyword works, a mismatch between logical and colloquial 'AND' and 'OR', and syntax issues such as quotation marks and parentheses.

A different course-long experiment on the differences between SQL and QBE [YS93] testing correctness found that there is only a significant difference between QBE and SQL in pen-and-paper tests (in favor of QBE), but not in an online setting. The same study also found that users who had already learned QBE did better with SQL, while users who first learned SQL did not show a significant difference in how well they used QBE. The paper also found that there is a set of operations in both query languages that make the queries significantly harder to work with, such as nested queries, grouping, set operations, and computed and correlation values.

In attempting to find what problems might be most pervasive while using SQL, a data mining investigation into Stack Overflow, a question and answer website for developers, by Nagy and Cleve [NC15] found that there are many questions about SQL on the platform, they especially find that one of the most commonly brought up SQL constructs is an INNER JOIN between two tables with 15,184 questions that contain that code. A study about what causes users to erroneously leave out join-clauses in SQL statements [Sme93] found that the two significant factors in this error are complexity in the query (more where clauses) and the absence of an explicit clue that a join clause might be required.

Some researchers investigated the technical differences between raw SQL systems and ORMs. A study comparing performance of ORM tools such as LINQ and NHibernate [CJ10], the .NET version of the popular Java-based ORM tool Hibernate, has found that the technical performance differences between ORM tools and SQL-based queries are not very large, but that ORM is slower than raw SQL. The author of that study suggests that the most likely reasons that ORMs are not more widespread are the complexities of integrating ORMs with legacy databases, the general distrust of ORMs because of experiences with earlier versions, and the significant investment of time and money to get the developers of a company trained [CJ10]. A similar experiment comparing the technical performance of the PHP-based Eloquent ORM and a similar SQL-string based tool [JH16] found that the SQL based tool was more performant than the ORM tool.

There is also some technical research on making ORM systems more performant and easier to use, such as Grust and Mayr's [GM12] proposal to change Ruby on Rails' Active record to send out fewer requests while using commands that are more native to the Ruby language.

There are multiple attempts to make the use of query languages themselves easier, for example by allowing users to have an amount of uncertainty with respect to specific names and relations between tables [LPJ14, TWCG12] or by creating visual interfaces for information retrieval [ACDLS02] and providing users with new tools such as debuggers [GR13]. One study found that using Visual SQL, a visualized version of SQL based on entity-relationship diagrams, leads to higher conceptual correctness [JT03]. Others attempt to replace query languages such as SQL by proposing systems that allow for more natural language queries [Har77].

Chapter 3

On the Effect of Type Information on Developer Productivity

To better understand the productivity impact of polyglot programming, considering the productivity implications of more well-researched aspects of programming languages can prove important. The programming language features most thoroughly researched under the aspect of programmer productivity are static type systems. There is a series of empirical studies that show a positive impact on productivity when programmers use statically typed programming languages. However, there is no clear theoretical framework to explain these results. This chapter will attempt to find a sound explanation for these findings in order to help advance our understanding of the mental process involved in programming tasks.

Additionally to adding to the understanding of productivity, an investigation into how developers benefit from certain programming language features can create a more detailed picture on how programmers perceive and process programming languages as they are using them, which is a vital part of understanding how switching programming languages might affect their productivity. Lastly, differences between languages, i.e. different features such as type systems and their textual representation, might also play a big role in switching processes, as the mental representation, in the light of different available information, handled code might differ between languages.

3.1 Introduction

There has been a long standing debate about the benefits of static type systems in the computer science community. Languages with static type systems are supposed to catch errors earlier, leading to more reliability. Opponents of statically typed languages argue that they lack flexibility or increase the amount of time to write a program. Partly because of this debate, evaluation of the impact of using different type

systems on development is one of the most commonly researched topics in human factors based empirical programming language research [Kai15b]. According to studies on the topic (see section 3.2 for details), static typing consistently shows a positive human factors impact compared to dynamic typing. Despite this consistency, while the effect has replicated across a wide variety of conditions, no reasonable explanation for the cause has emerged. We suspect this is the case, in part, because such research on students or other programmers is still quite rare in the literature [Kai15b].

In this chapter, the goal is to propose a theoretical explanation aimed at identifying why these observed differences may exist in the studies in literature. Considering previous findings and Spiza's research [SH14b], it seems reasonable that the cause is at least partially the annotations provided by static typing, which give the programmer more information to improve comprehension and thus improve performance. From the observations from the literature I construct a more general explanation and test it using a empirical experiment.

In the experiment, computer science students from the University of Nevada, Las Vegas are asked to solve a number of programming tasks using either constants or enumerated types. Since both conditions had a form of static typing, even though they might differ in how the compiler handles the type checking for both these systems and in how the type annotations are written, based on the theoretical explanation that most of the performance benefit should come from the type annotations rather than the type system itself, and because the same information is present in both types of annotations in this experiment, one would expect that the results between the groups would be very similar, while, if the performance results between the groups differ, one could expect that the explanation is wrong and that there must be another cause for the differences, for example the difference in type checking and error messages.

Ultimately, this chapter found that the results of the empirical study provide observations consistent with the theoretical explanation. This does not mean that the results prove that the explanation is correct, but the study with conditions not previously seen in the literature, was conducted in a good faith effort to disprove the explanation if possible.

The existing research on type systems and theories on the comprehension of code is presented in section 3.2. The literature on the topic will be reviewed and a comparison of the various studies to date will be provided. The existing evidence will be discussed and combined a first-cut explanatory theory in section 3.3. Finally, the type inference experiment will test the theoretical explanation in section 3.4.

3.2 Related Work

There exists a broader set of recent research investigating how the design of programming languages impacts, especially, students. As analyzed by Kaijanaho, the number of controlled trials in the area of programming

language usability between 1973 and 2012 is low: Kaijanaho found only 22 publications that tested programming language features [Kai15b].[1] Still, according to Kaijanaho *'the choice between static and dynamic typing is the second most studied design decision'* [Kai15b, p. 143]. While the focus of this work is on specifically studies that put their ideas about language design to the test in rigorous experiments, there are a variety of recent works on different kinds of debates in the literature, like whether or not visualization or blocks have an impact on learning. These studies all provide different kinds of information and this section is attempting to provide some general context on the topic.

3.2.1 Type Systems

Table 3.1 shows a summary of the literature that empirically tests the effect of type systems on users. Apart from the publication year, title and author names, a column was included indicating if the results are positive for the group or groups using type annotations and a column on the effect size if it was provided by the author. There are several studies in the table for which it was hard to make a strict determination. For example, Prechelt and Tichy's study [PT98] compared two groups with type annotations, so they were marked as "not covered" in the type annotation column. Stuchlik et al. [SH11b] focused on type casting and found negative impacts for the groups using annotations for short tasks, but not for longer ones, which is why the result was marked as mixed. Mayer et al. [MHR$^+$12b] also found benefits for the group without type annotations when it came to shorter, simple tasks, but found the opposite for longer, more complex tasks, making the results come out as mixed for type annotations. The last study marked as mixed is Okon et al.'s [OH16] study trying to force a positive impact for dynamically typed languages by deliberately designing artificial tasks to favor dynamic type systems. The study found a positive impact for the group without type annotations in two of the tasks, while the other two came out positive for the group with type annotations. In that case, however, the tasks were designed to make the annotations especially difficult.

The effect size in the table is shown as partial eta squared (η_p^2) which is the measure common between all the studies which reported effect sizes. It describes the percentage of variance in the measurements of the dependent variable that a factor in an ANOVA accounts for and is a number between 0.0 and 1.0. The effect size shown in this column represents the reported effect sizes between the groups using type annotations and the ones without. The first four of the studies in the table did not report an effect size and focused on finding significant differences between the groups and describing the difference of means without using ANOVAs. The fifth study in the table [MHR$^+$12b] did use ANOVAs to analyze the results, but did not report the effect size for the non-significant difference between groups. Some of the papers reported two effect sizes which were separated with commas. In these cases, the results come from counterbalanced experimental

[1] This statement is in line with a number of authors that argue that experimentation in general is not often applied (see for example [TLPH95, Tic98]).

Year	Study Name	Authors	Positive for Type Annotations	Effect Size
1977	An Experimental Evaluation of Data Conventions	J.D. Gannon	Yes	not reported
1998	A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking	Lutz Prechelt and Walter F. Tichy	not covered	not reported
2010	Doubts about the Positive Impact of Static Type Systems on Programming Tasks in Single Developer Projects - An Empirical Study	Stefan Hanenberg	No	not reported
2011	Static vs. Dynamic Type Systems: An Empirical Study About the Relationship between Type Casts and Development Time	Andreas Stuchlik and Stefan Hanenberg	Mixed	not reported
2012	An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software	Mayer et al.	Mixed	non-significant
2013	Do Developers Benefit from Generic Types?	Michael Hoppe and Stefan Hanenberg	Yes	0.24
2014	How Do API Documentation and Static Typing Affect API Usability?	Stefan Endrikat et al.	Yes	0.3
2014	Type Names without Static Type Checking already Improve the Usability of APIs (As Long as the Type Names are Correct): An Empirical Study	Samuel Spiza and Stefan Hanenberg	Yes	0.11, 0.33
2014	An empirical study on the impact of static typing on software maintainability	Stefan Hanenberg et al.	Yes	0.275, 0.246
2014	An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse	Pujan Petersen, Stefan Hanenberg, and Romain Robbes	Yes	0.56
2015	An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability using TypeScript and JavaScript in MS Visual Studio	Lars Fischer and Stefan Hanenberg	Yes	0.33, 0.44
2016	Can We Enforce a Benefit for Dynamically Typed Languages in Comparison to Statically Typed Ones? A Controlled Experiment	Sebastian Okon and Stefan Hanenberg	Mixed	0.039 (non-significant)

Table 3.1: Studies on the difference between static and dynamic typing with effect size (η_p^2) found

groups in which participants experience both treatments, first one and then the other. Thus, the first value represents the effect size in the first round and the second value represents the effect size in the second round.

The first controlled experiment that careful search could find was conducted by Gannon in the 70s and revealed a positive influence of a static type system in comparison to an untyped system in terms of errors and development time [Gan77]. Following that, Prechelt and Tichy ran a controlled experiment that compared static type checking versus no type checking on procedure arguments using two different versions of C, with the result that the type checked group was faster in solving programming tasks [PT98]. Later, Hanenberg conducted an experiment with students, asking them to write a parser in one of two specifically tailored programming languages. The experiment revealed a significant positive time benefit for a smaller task using a dynamically typed language (in comparison to a statically typed one), while no difference was measured for a larger task [Han10c, Han10a]. This experiment started a series of experiments on the topic. Stuchlik et al. analyzed the influence of type casts on development time (using a static or dynamic type system) for simple tasks [SH11b] with the result that type casts influence only the development time of very small tasks in a negative way, while no difference was measured for bigger programming tasks. Mayer et al. studied the effect of static types on API usability [MHR$^+$12b] with respect to developer performance. For most of the programming tasks the static types were beneficial. This experiment was later partially replicated by Kleinschmager et al. [KHR$^+$12, HKR$^+$14b] and in addition, the experiment by Kleinschmager et al. showed that static type systems improve developer performance for type error fixing tasks. Hoppe et al. investigated

the effects of generic types and showed in an experiment that additional type information provided by generic types have a positive effect on the usability of APIs [HH13b], while possibly reducing the extensibility of class hierarchies. Notably, Spiza et al. studied whether the findings from previous experiments could be reduced to the pure syntactic representation of type annotations using the Dart programming language and disabling type checking, showing a clear positive effect but with the caveat that single errors in these declarations can reverse any positive effects on development time [SH14b]. In an effort to check whether the previous findings could be replicated using modern IDE support, Petersen et al. ran an experiment allowing participants to use Eclipse and successfully confirming previous results [PHR14b]. Endrikat et al. [EHRS14b] performed an experiment in which the positive effect of the static type systems was larger than the effect of providing written documentation, hinting that type annotations might be a powerful tool to provide documentation in code. Interested in the effect that software development tools might have on the observed effects, Fischer et al. compared the effect of type systems with and without code completion, finding that the effect of static type systems remains large with and without code completion. Further, the effect of code completion on productivity was not significant [FH15b]. Finally, Okon et al. tried to enforce a benefit for dynamically typed programming languages using purely artificial programming tasks [OH16] to benefit dynamic typing. It turned out that only in two cases such benefit for dynamically typed languages was visible, in the other two cases the statically typed group still performed better. Most of these studies showed a benefit for statically typed programming languages with rare exceptions. Even an experiment trying to enforce a benefit for dynamically typed languages failed to do so consistently.

The only study on type inference that could be found, a qualitative study by Daly et al., checked, to what extent type inference, added to the programming language Ruby, helps developers. The authors argued that they were not able to detect differences in this study that was based on four subjects [DSF09].

Looking at empirical repository studies, a large scale study on 100,000 subjects, Brown and Altadmri compared the subjective impression of educators that teach programming languages with the actual data provided by students. With respect to type systems it turned out that the educators' opinion that types (e.g. in method calls) are often chosen in a wrong way was '*overrated in terms of frequency*' [BA14]. Additionally, a code repository study on projects written in a language with an optional type system conducted by Souza and Figueiredo detected that developers are much more likely to add type annotations to interface descriptions than to local variables [SF14]. Investigating programming languages by analyzing code commits on GitHub, Ray et al. [RPFD14] compared the amount of commits referring to bugs between languages. They found that statically typed languages seemed to produce fewer commits referring to errors than dynamically typed languages. Evaluating the ability of static type systems to detect bugs in JavaScript, Gao et al. [GBB17] tested whether the type systems of Flow 0.30 and TypeScript could detect bugs in publicly

available code. They found that both type systems could have prevented 15% of the bugs tested.

In three different studies Stefik and Siebert tested the effect of programming language syntax on different kinds of developers (from freshmen to professionals)[SS13]. Although type systems are not in the main focus of the study, it shows that especially novices have problems with type declarations. The study found a 92% error rate for type annotations.

3.2.2 Naming

Studies concerned with naming of identifiers when programming and its connection with comprehensibility of programs such as Lawrie et al. find that full word identifiers increase program comprehension compared to abbreviations and single letter identifiers [LMFB06]. Hofmeister finds supporting evidence for this by showing that programmers are able to find semantic defects in programs faster when working in environments with full word indentifiers instead of abbreviations and single letter identifiers [Hof15]. Binkley et al. found that users more accurately identify camel-cased identifiers compared to identifiers with underscores [BDLM09].

3.2.3 Program Comprehension

The process of understanding source code is the process of reading the code and additional pieces of information, be that textual documentation, graphs, or problem statements, about the code to create an understanding of what the program does or how it does it. Different models of the comprehension of code agree that this is a logical image of the code, instead of a literal one revolving around the syntax of the code [SM79, Bro78], i.e. it is an abstract representation of what the code does decoupled from the language it is written in. Based on this decoupling of understanding of code from its representation, Shneiderman states that programmers can often read a program in one language and reproduce it in another [SM79]. The view that there is a separation of logical representation and language is also held in linguistic research [KS94], supporting this interpretation of the cognitive mechanisms involved in code comprehension.

Top-Down Comprehension One of the most common models of code comprehension is the model of top-down comprehension. In this approach, program comprehension is mainly driven by the domain knowledge of the programmer. According to the model, a program cannot be fully understood unless the domain of the program is understood as well [Bro78]. The model states that a developer trying to understand a piece of code would use a number of cues from the code itself and from external documentation to reconstruct the domain knowledge and the connections between the different levels of domains that pertain to the program. Levels of domains in this sense span from low level machine instructions, over algorithms used to

solve the problem, to the real world use of the program. The code may have internal cues, often called beacons [SPP+17], include comments, formatting of code, structure of the code, method names, variable names, and type names. Using the different pieces of information available to the programmer, they may create and successively refine hypotheses about how the program might work until they are sufficiently confident in their knowledge of the program and the connected knowledge domains to complete their task. A recent fMRI study by Siegmund et al. [SPP+17] was unable to find evidence that the code comprehension of participants benefited from the beacons, but indicated that comprehension of code with beacons required less cognitive effort than comprehension with scrambled beacons. One interesting finding in their study is that beacons increased the use of an area of the brain typically related to natural language processing, likely due to more recognizable variable names. The author is unaware of a study on beacons related to type systems, but it is possible to make connections between this chapter is calling type annotations and what might be more generally classified as beacons.

Bottom-Up Comprehension Bottom-up comprehension views code comprehension as a process in which programmers construct knowledge about a program by reading pieces of code and mentally aggregating these code pieces into semantic chunks, building bigger chunks by combining chunks and therefore creating a semantic understanding of the program they are trying to understand [SM79, Bro78]. Siegmund et al. [SPP+17] state that the observed activity in the brain (BA 39) when programmers are forced to read code without semantic cues backs up the theoretical description of chunking as a semantic building process.

Work in the 80s and 90s especially challenged these simplistic models of top-down or bottom-up comprehension. For example, Pennington found that people in practice do seem to use such mechanisms, but they bounce between these kinds of processing [Pen87b, Pen87a]. Later work by Andrews (previously Mayrhauser) and others fleshed out these models [MV94, MV96, MV97]. To an extent, these models can be explanatory, but they are also generally not predictive i.e. they do not obviously predict whether something like static or dynamic typing would afford better human performance for some set of tasks.

Information Foraging Theory Apart from cognitive models of code comprehension, the field of human-computer interaction (HCI) considers the Information foraging theory (IFT) [PC95], which can be applied as a high-level view of information gathering when developers are trying to understand code. IFT is a set of ideas based on foraging theory from the field of biology [SK86], describing a person's search for information as similar to a predator's hunt for its prey. In this model, a person is searching for information in an environment made up of a topology of patches which contain information. Some of the pieces of information are cues (or scents) to lead the person or predator to the next patch. Conceptually, the idea is similar to beacons, although generally not discussed in the same way. People can process information, move to another

patch, or try to add another patch to the topology by enriching the environment. IFT predicts that people want to make the choice that generates the most value for the least amount of cost, but the amount of value and cost are often left ambiguous [FSP+13].

This theory is often applied to predict and describe the behavior of users on the web. In this case, patches can be websites and the connections between the patches are hyperlinks, hyperlink representation can be interpreted as cues. Research in this area has found that people tend to follow cues that are estimated to have more value [WM99, VLP09] and lower cost [PC98] as the theory predicts. A number of researchers have applied this theory to the behavior that software developers exhibit while navigating source code [KMCA06, LBB07, NMB11, FSP+13].

In this application of the theory, code files as well as additional materials such as documentation, e-mails, and bug tickets can be seen as the information patches that contain cues to other parts of the topology. For example, a bug ticket could mention the name of a class or method, which could lead a developer to check the entry in the systems documentation and the source code itself for the mentioned artifact. Including different types of material outside of the code to understand a program draws obvious connections to the top-down strategy of code comprehension. It seems possible that the IFT could function as a descriptive model of how developers might navigate between resources when applying a top-down strategy of code comprehension. IFT does not make predictions on the efficiency of programmers while programming, but describes their behavior in how they would move around an information environment and thus was not applicable to predict the outcome of either of the two experiments.

Cognitive Dimensions of Notations The last theory that will be mentioned is the well-known Cognitive dimensions of notations [Gre89, GP+96]. Cognitive Dimensions is a framework of thinking about design decisions when developing a user interface, typically a programming language or its surrounding components. The main focus of cognitive dimensions is to provide designers with vocabulary about the aspects of a design in question. As such, the framework is not a theory that can be used to predict the difference between programming languages with or without type annotations. In the terms of cognitive dimensions, the addition of type annotations into a language could be described as an increase in viscosity, the dimension concerned with how hard it is to make changes, as developers would have to type more to add a new variable or change the type of a parameter in more places when such changes are necessary. Diffuseness would also be increased, as more symbols would be required to create variables and define methods. There would likely be a decrease in error proneness if the types are checked, as well as a decrease in hidden dependencies since the type annotations would more directly spell out the dependencies and parameters passed into a method

are limited in their type.

However, it should be clear that Cognitive Dimensions lacks the ability to create specific predictions, and as such it is less general than it is speculative. Further, multiple authors have shown that the theory is non-evidence based [Kai15b, SHM+14]. In this context, it means that when scholars carefully investigate papers written using the theory, the finding is that those papers lack a foundation of evidence to come to theoretical conclusions.

3.3 Theoretical Synthesis

Given the results reviewed from previous studies, it is to be believed that it is reasonable to conclude that there is an overall positive impact on human productivity using statically typed programming languages with type annotations compared to languages that use dynamic typing. The results indicate that a substantial amount of the variance between groups in these experiments can be explained by the choice of type systems. Additionally, when looking at the error rates between statically and dynamically typed programming languages, studies have found that errors are less common for participants using statically typed languages.

Spiza's findings [SH14b] suggest that the performance effect does not come from the process of type checking itself in the compiler, but rather from the type annotations. Taking this interpretation further, it can be concluded that type annotations provide semantic information (serving as form of documentation) to programmers which aides code comprehension and thus performance. This conclusion seems reasonable when considering the results by Endrikat et al. [EHRS14b], which found that static type systems outperformed the effect of written documentation.

Looking beyond type systems, it can be seen that additional semantic information can increase comprehension and performance in other aspects of programming as well. For example, Becker et al. [Bec16a, BGI+16] studied the impact of improved contextual compiler error messages, especially for students, and found a positive impact of additional semantic information. While the specific messages in his tests are not always listed in his academic work, they were reviewed through private correspondence and compared to his published statistical results. Although this subject matter is very different than type systems, it was found that the results are consistent with the explanation of performance improvements from additional semantic information. It can be concluded, broadly speaking, that semantic annotations (effectively small differences in the structuring of words, the existence of words, or the specific words chosen) cause a statistically significant, identifiable and quantifiable performance improvement on similar tasks.

Stefik and Siebert [SS13] found that participant performance for the syntax choice "*for*" was worse than for the choice "*repeat*" as a loop keyword. The same finding was made by Weintrop [WW15] when comparing block- and text-based languages, finding a significant difference when evaluating student performance on understanding iterative logic that could be explained by the choice of keyword and syntax structure used for iteration. His results clearly demonstrated that the block based language using a simple "*repeat*" keyword and syntax structure outperformed the text based language keyword and syntax structure of a "*for*" loop. Showing that specific word choices and therefore the adequate presentation of semantic information can significantly influence the comprehension of contextual programming-related information. With regard to the preferred presentation of names, such as for identifiers of methods, variables, or types, Lawrie et al. [LMFB06] and Hofmeister [Hof15] show evidence that full name identifiers are beneficial and research by Binkley [BDL+13, BDLM09] shows that camel case is preferable over underscores.

Since the conclusion relies heavily on the semantic information that type names often carry, the idea of beacons in top-down code comprehension seems like a reasonable broad-based fit conceptually. Brooks [Bro78], in fact mentions type names as possible beacons that help programmers understand a given piece of code. This theory was not created based on evidence, however, and was called into question after its inception as a result. Nonetheless, it is intuitively easy to accept that type annotations could play an important role in how bottom-up code comprehension works. Type annotations could enhance a programmer's understanding of the semantics of a chunk of code since they could provide more contextual information in the limited scope of the chunk. Relative to the perspective of information foraging theory, this same effect could be stated by saying that annotations may be interpreted as scents that may lead programmers to investigate specific information patches, like type definitions in this case. It is important to note that these general conceptual theories are not specific enough to predict effect sizes or results and that if the empirical studies that were reviewed demonstrated the opposite results, these same theories could be used to provide arguments in the opposite way.

The next step is to interpret the empirical data in the literature and to aggregate the findings into a preliminary explanation that generalizes the findings in a bottom-up, evidence-first approach. It was attempted to strike a balance between generalizing the results to make it more broadly applicable and still keeping the explanation specific to what was observed in order to limit broad claims with weak backing. This and other explanations can and should be built on in independent studies in similar and dissimilar areas by others so that the programming language community can ultimately develop predictive, evidence-based, specific theories to guide design decisions for programming languages, software libraries and computer science education. The preliminary theoretical explanation in this context is:

Semantic Annotations Principle: Programming interfaces providing well-formed, explicit semantic information afford superior developer outcomes through enhanced code comprehension and contextual clues.

Intuitively, this principle seems obvious, but it is at odds to a degree with claims made by Green [GP+96] that language verbosity (or diffuseness in the cognitive dimensions of notations) has an adverse impact on comprehension and productivity. Ignoring the fact that Green's original claim was not based on evidence, it is a widely cited and accepted theory which the experimental results that were reviewed partially refute. The results clearly show in multiple trials and institutions that providing explicit additional information, especially in the form of type annotations, is not essential to the function of a program and makes the code more verbose than necessary, but clearly results in improved performance.

Considering that wrong type annotations severely increased time to solution in Spiza's study [SH14b], it seems that programmers utilize the information within type annotations with the notion that they are reliable, explicit, semantic information about the program. This semantic information enriches the code with more detailed clues on the data-flow of the program as well as the available methods to solve the task. This can aid code comprehension and reduce the time participants need to finish a task. The reduction in time needed to comprehend code seems to outweigh the negative time effect of adding the type annotations when writing code, although this effect may stem from the type of task typically employed in these comparative experiments. Most programming tasks in experiments about type systems are about debugging or extending existing code, requiring periods of time in which participants familiarize themselves with the presented code.

The results presented in this paper lead to the conclusion that one impactful way to improve productivity in debugging and extension tasks is reducing the time needed to comprehend the pre-existing code. This conclusion is shared by other scholars that claim that the most time intensive programming activity is code comprehension [Sta84, Fje83, CSW02]. The review shows that one effective way to increase the speed of code comprehension is to use type annotations in code to create additional, explicit, contextual semantic information. Static type checking in programming languages is also important for other reasons, including its role in enhancing runtime reliability in executable code through the elimination of runtime type errors and reliability of type annotations [SH14b].

3.4 Enumerated Types Experiment

To test the validity of the semantic annotations principle, an experiment was conducted comparing two groups with differences in type system and error handling while the type annotations between the groups still contain the same information. If a significant difference between the groups can be found, that would disprove the semantic annotations principle.

3.4.1 Experiment Description

The experiment was a randomized controlled trial conducted at the University of Nevada, Las Vegas investigating the impact of enumerated types on programming speed conducted in an online environment. Enumerated types are a feature in programming languages that allow for the definition of new types with a set of named values. As previous studies (see section 3.2) have shown benefits of static typing compared to dynamic typing, it was interesting whether this might be due to more specificity in type, such as that found in enumerated types compared to solutions with constants.

The experiment aims to investigate the implementation of enumerated types in Java. Java's enumerated types are compared to Java's constants (i.e. members with the keywords static and final), as programmers can solve the same problems with both programming language features and to keep the groups as similar as possible but for the specific feature used. From a perspective of a language designer considering whether to implement an enumerated type system in their new language, constants can be considered the baseline against which enumerated types are to be tested, as they are a common feature.

Because the broader experimental line is trying to find more information about a potential cause for differences in type systems, enumerated types were chosen because the annotations between constants and enums can be very similar, especially in the second and third task, where the enumerated types are compared with constant objects. In the experiment, if it is really the case that the annotations are contributing to developer productivity, we might observe next to no impact from enums. In contrast, if the previous ideas are incorrect, we would expect to observe some kind of difference with enumerators compared to constants.

The experiment was trying to answer the following research question:

RQ: What is the effect of using enumerated types compared to using constants for the same task?

Developer performance can be measured in terms of time in seconds required by developers to solve a given programming task. Thus, the general research question can be formulated as a null hypothesis H0 and a corresponding alternative hypothesis H1:

H0: $\mu_{enum} = \mu_{constants}$

H1: $\mu_{enum} \neq \mu_{constants}$

The means μ_{enum} and $\mu_{constants}$ represent the means for the development times for a given programming task for the enum and constants group respectively.

The experiment was designed as a two group repeated measures experiment. Participants were randomly assigned to one of the two groups by the online platform based on their self-reported level of education

(Freshman, Sophomore, Junior, Senior, etc.).

Participants

The programming experience was self-reported by the participants in a survey included in the study. The participants were recruited in different computer science classes at the University of Nevada, Las Vegas. Classes were selected, so that they were evenly distributed across different years in the students' college careers, so that participants came about equally from different programming skill levels. Level of education is a common measure for programming experience for university students [SKL+14a] and significant differences between levels have been found in respect to time to solution in a previous study[USH+16]. The participants were informed about the study during class time by a researcher reading the advertisement pamphlet while every student was given a copy. Then, students were able to go to the URL posted on the pamphlet and start the experiment whenever they had time. Students were offered extra credit for participation in the experiment and the amount of extra credit was based on what the professor was willing to give the students and ranged between 1-3% of total class points. Alternatively to participating in the study, students were able to achieve the same amount of extra credit by submitting an essay on a computer science topic.

Randomization

The process of assigning participants to the two groups happened entirely on the online platform. After entering their college year or professional status in to the survey at the beginning of the experiment, the participants were assigned to an experience category based on that information. The platform kept track which groups were already assigned to in each category until each group was assigned once, then all groups were free to be assigned again until each has been filled again. This mechanism was in place to keep the distribution to the groups as even as possible. Since this experiment only had two groups, that means that the platform determined which group to assign to a person using a pseudo-random number generator, and then noted which group was next in the database. The next person in the same category was assigned the noted group and the database reset the field of which group is going to be assigned next. The next person in the same category was then randomly assigned to one group and the other group was noted in the database again and so forth.

Measured Variables

In this experiment the main outcome variable was the time it took participants to complete a task. Additionally, the switches between the sample files in each task were measured as a possible explanation for

the differences in time. Also recorded was the participants' level of education as a measure of experience. For qualitative evaluation of the progress the participants made during each task and further study in the future, snapshots of the code inside the text box and the information inside the output box were recorded whenever the participants submitted their code to be checked, when the time ran out, and every 10 seconds.

Study Setting

After giving consent to participate in the experiment in the online platform, the participants were asked to fill out a survey to provide us with demographic information. When they were done filling out the survey, they could start the tasks of the experiment. First, participants were asked to read a code sample and were given five minutes until the programming part of the task would start automatically. Participants were also able to move on to the code section earlier by pressing a button on the web page. The code sample stayed accessible to the participants during the entire task, enabling them to refer back to it at any time.

The programming part of each task consisted of following instructions given to the participants in a text box in which they were also asked to program. The instructions were written in comments of the Java code they were asked to complete. In each of the tasks participants were provided with varying amounts of code scaffolding so that getting started with the tasks was not dependent on if participants knew how to write method syntax in Java and to make the tasks easily testable with automatic unit tests.

A participant could then compile and run their task by clicking a button and were then given the output of the Java compiler and program execution including information from the automatic test cases, if successfully compiled, in an additional textbox. If all unit tests passed successfully, the platform congratulated the participants and let them move on to the next task. If the task was not successfully completed, then participants were able to keep working on the task until they succeeded or until the time limit of 35 minutes was reached. If the time limit was reached, then participants would be moved on to the next task regardless of if they completed the task. The time limit was put in place to limit the maximum amount of time a participant would have to spend on the experiment and was negotiated with the local ethics board. While the experiment should provide students as long as they need to solve the tasks, their time commitment should also be ethically considered. Like before, the tasks were also piloted, although this does not always help ensure no ceiling is hit.

Programming Tasks

Participants in both groups were asked to solve the same tasks in a programming environment, but were encouraged to use a different feature by being provided a different code sample for their tasks. In the first

```
1  package code;
2  public class CellNetwork {
3
4    // First part of Solution
5    public enum Network
6    {
7      ATNT, VERIZON, SPRINT, TMOBILE, USCELLULAR
8    }
9    // end first part of solution
10
11   public String getNetworkCompanyName(Network network) {
12
13     String networkName = "";
14     // Second part of the solution
15     if (network == Network.ATNT) {
16         networkName = "AT&T";
17     } else if (network == Network.VERIZON) {
18           networkName = "Verizon Wireless";
19     } else if (network == Network.SPRINT  ) {
20           networkName = "Sprint Corporation";
21     } else if (network == Network.TMOBILE) {
22           networkName = "T-Mobile";
23     } else if(network == Network.USCELLULAR) {
24           networkName = "U.S. Cellular";
25       } else {
26       networkName = "UNKNOWN";
27       }
28     // end second part of solution
29
30     return networkName;
31   }
32 }
```

Code Sample 2: Enum group task 1 with solution marked with comments.

task, which can be seen in code sample 2 and code sample 3 respectively, participants were asked to create a number of "items", which was a neutral term used to refer to either enums or constants. The neutral term was used to keep the instructions neutral between the groups. Additional to creating "items", participants were asked to complete a method which returns a string based on what "item" was passed in as parameter. This could easily be solved using a switch-statement or using a number of if-statements. The samples for both groups used if-statements to solve the task to keep the syntax comparable to a wider range of programming languages.

For the second task (see code sample 4, and code sample 5), participants had to create a data structure, either a class or an enum declaration, containing a *toString*() method and a number of entries of the newly defined type.

The solutions of task 3 for the constants group can be seen in code sample 6 to code sample 9, and the solutions for the enum group can be seen in code sample 10 to code sample 13. Due to the length of the task code, the code samples had to be cut into different methods. The code needed to solve task 2 was available in task 3 in which participants had to fill out a number of methods to do specific tasks with the enums and constants, such as list all errors and react to specific errors in specific ways.

```
1  package code;
2  public class CellNetwork {
3
4    // First part of Solution
5    public static final int ATNT = 0;
6    public static final int VERIZON = 1;
7    public static final int SPRINT = 2;
8    public static final int TMOBILE = 3;
9    public static final int USCELLULAR = 4;
10   // end first part of solution
11
12   public String getNetworkCompanyName(int network) {
13     String provider = "";
14
15     // Second part of the solution
16     if (network == ATNT) {
17       provider = "AT&T";
18     } else if (network == VERIZON) {
19       provider = "Verizon Wireless";
20     } else if (network == SPRINT) {
21       provider = "Sprint Corporation";
22     } else if (network == TMOBILE) {
23       provider = "T-Mobile";
24     } else if (network == USCELLULAR) {
25       provider = "U.S. Cellular";
26     }
27     // end second part of solution
28
29     return provider;
30   }
31 }
```

Code Sample 3: Constant group task 1 with solution marked with comments.

3.4.2 Results

This section will discuss the results of the experiment. First, recruitment numbers are discussed in 3.4.2, then subsection 3.4.2 will present the descriptive statistics, finally followed by subsection 3.4.2, which discusses the details of the analysis of the data. The script used for analyzing the data can be found at `https://bitbucket.org/stefika/replication/src/master/2019UesbeckDissertation/enum`.

Recruitment

Sixty-seven participants across six levels of education (Freshman, Sophomore, Junior, Senior, Graduate, Post-Graduate) were recruited for this study to see if the impact could be found at all or if it was level dependent. Six of the participants identified themselves as female and 26 did not identify their gender, either by choice or because they never reached the end survey of the experiment. One of the participants identified themselves as a freshman, nine identified as Sophomores, 31 identified as juniors, 22 as seniors, 2 as graduate students, and 2 as post-graduates.

Of the 67 participants, 10 participants' data had to be removed because of either bugs in the testing environment (which made them lose their progress accidentally, making a fair time measurement impossible) or because they did not attempt to solve the tasks and waited out the maximum task time without changing the code to reach the section of the experiment in which they can enter their class information to receive their extra credit (although they still received extra credit). It was determined whether a participant was actively

```
1   public static class HTTPErrorCodes {
2     public int codeNum;
3     public String errName;
4     public String errDesc;
5     public HTTPErrorCodes(int num, String name, String desc) {
6       this.codeNum = num;
7       this.errName = name;
8       this.errDesc = desc;
9       }
10    @Override
11    public String toString () {
12      return codeNum + ":" + errName + ":" + errDesc;
13    }
14  }
15  public static final HTTPErrorCodes OK = new HTTPErrorCodes (200, "OK", "Action completed successfully."
        ↪ );
16  public static final HTTPErrorCodes CREATED = new HTTPErrorCodes (201, "Created", "Success following a
        ↪ post command.");
17  public static final HTTPErrorCodes NO_CONTENT = new HTTPErrorCodes (204, "No Content", "Server has
        ↪ received the request but there is no content to send back.");
18  public static final HTTPErrorCodes MOVED_PERMANENTLY = new HTTPErrorCodes (301, "Moved Permanently", "
        ↪ Requested a directory instead of a specific file. The web server added the filename index.html,
        ↪  index.htm, home.html, or home.htm to the URL.");
19  public static final HTTPErrorCodes BAD_REQUEST = new HTTPErrorCodes (400, "Bad Request", "The request
        ↪ had bad syntax or was impossible to be satisfied.");
20  public static final HTTPErrorCodes UNAUTHORIZED = new HTTPErrorCodes (401, "Unauthorized", "User failed
        ↪  to provide a valid user name / password required for access to file / directory.");
21  public static final HTTPErrorCodes FORBIDDEN = new HTTPErrorCodes (403, "Forbidden", "The request does
        ↪ not specify the file name. Or the directory or the file does not have the permission that
        ↪ allows the pages to be viewed from the web.");
22  public static final HTTPErrorCodes NOT_FOUND = new HTTPErrorCodes (404, "Not Found", "The requested
        ↪ file was not found.");
23  public static final HTTPErrorCodes SERVER_ERROR = new HTTPErrorCodes (500, "Server Error", "In most
        ↪ cases, this error is a result of a problem with the code or program you are calling rather than
        ↪  with the web server itself.");
24  public static final HTTPErrorCodes NOT_IMPLEMENTED = new HTTPErrorCodes (501, "Not Implemented", "The
        ↪ server does not support the facility required.");
25  public static final HTTPErrorCodes SERVICE_UNAVAILABLE = new HTTPErrorCodes (503, "Service Unavailable"
        ↪ , "The server cannot process the request due to a system overload. This should be a temporary
        ↪ condition.");
26  public static final HTTPErrorCodes GATEWAY_TIMEOUT = new HTTPErrorCodes (504, "Gateway Timeout", "The
        ↪ service did not respond within the time frame that the gateway was willing to wait.");
```

Code Sample 4: Constant group task 2 solution.

attempting to solve a task by inspecting the snapshots of their code by hand. If the snapshots didn't show progress in the code, the participants were ruled out. The bugs that some participants encountered reset their code to the starting point, costing them a lot of time as they would have to rewrite everything they had been working on and making the time measurements inaccurate. Thus, after exlusions, 57 participants were left.

Descriptive Statistics

Not all 57 participants completed all of the tasks and some participants reached the maximum time on some of the tasks, especially task three. Only three participants actually completed the last task in the given time frame.

Overall, participants in the constants group spent less time per task (M = 1525.36, SD = 648.92) than participants in the enum group (M = 1563.74, SD = 672.13). Participants in the constants group spent more time on average on task one (M = 939.07, SD = 615.95) than participants of the enums group (M =

```
1   public enum HTTPErrorCodes {
2     OK( 200, "OK", "Action completed successfully."),
3     CREATED( 201, "Created", "Success following a post command."),
4     NO_CONTENT( 204, "No Content", "Server has received the request but there is no content to send back.
          ↪ "),
5     MOVED_PERMANENTLY( 301, "Moved Permanently", "Requested a directory instead of a specific file. The
          ↪ web server added the filename index.html, index.htm, home.html, or home.htm to the URL."),
6     BAD_REQUEST( 400, "Bad Request", "The request had bad syntax or was impossible to be satisfied."),
7     UNAUTHORIZED( 401, "Unauthorized", "User failed to provide a valid user name / password required for
          ↪ access to file / directory."),
8     FORBIDDEN( 403, "Forbidden", "The request does not specify the file name. Or the directory or the
          ↪ file does not have the permission that allows the pages to be viewed from the web."),
9     NOT_FOUND( 404, "Not Found", "The requested file was not found."),
10    SERVER_ERROR( 500, "Server Error", "In most cases, this error is a result of a problem with the code
          ↪ or program you are calling rather than with the web server itself."),
11    NOT_IMPLEMENTED( 501, "Not Implemented", "The server does not support the facility required."),
12    SERVICE_UNAVAILABLE( 503, "Service Unavailable", "The server cannot process the request due to a
          ↪ system overload. This should be a temporary condition."),
13    GATEWAY_TIMEOUT( 504, "Gateway Timeout", "The service did not respond within the time frame that the
          ↪ gateway was willing to wait.");
14    int code;
15    String name;
16    String description;
17    private HTTPErrorCodes(int code, String name, String desc) {
18      this.code = code;
19      this.name = name;
20      this.description = desc;
21    }
22    @Override
23    public String toString() {
24      return code + ":" + name + ":" + description;
25    }
26  }
```

Code Sample 5: Enum group task 2 solution.

845.32.07, SD = 573.73). This is reversed in task two where the constants group spent less time on solving the task (M = 1557.69, SD = 474.27) compared to the enums group (M = 1808.96, SD = 418.78). The average times for task 3 are about the same, with a slightly longer average time for the constants group (M = 2079.31, SD = 118.92) compared to the enums group (M = 2036.93, SD = 233.26). Table 3.2 shows details on the average task times broken down by group. The boxplot in figure 3.1 gives an overview of the recorded times per task.

Analysis

Time: To analyze the results a mixed designs repeated measures ANOVA ws run using the R programming language with the outcome variable being the time to a correct solution, using group as a between-subjects variable and task as a within-subject variable. Sphericity was tested using Mauchly's test for Sphericity, which shows that the assumption of sphericity was violated for the variable task. As is standard, all reported numbers are shown with Greenhouse-Geisser corrections taken into account. The ANOVA shows a significant interaction effect at $p < 0.05$ between the between-subjects variables task and group $F(2, 110) = 3.73,\ p = 0.032,\ (\eta_p^2 = 0.030)$, as well as a significant effect for task $F(2, 110) = 152.49,\ p < 0.001,\ (\eta_p^2 = 0.555)$. There was no significant effect for group $F(1, 55) = 0.19,\ p = 0.661,\ (\eta_p^2 = 0.002)$. This indicates that the

```
1   /**
2    * TODO: Complete the method below.
3    * - This method should put all the HTTP
4    * Errors in the following format into the list.
5    * <CODE:NAME:DESCRIPTION>
6    *
7    * for example:
8    * 200:OK:Action completed successfully.
9    *
10   */
11  public List<String> getAllErrors() {
12
13    List<String> errors = new ArrayList<String>();
14    errors.add(""CODE:NAME:DESCRIPTION"");
15
16    errors.add(OK.toString());
17    errors.add(CREATED.toString());
18    errors.add(NO_CONTENT.toString());
19    errors.add(MOVED_PERMANENTLY.toString());
20    errors.add(BAD_REQUEST.toString());
21    errors.add(UNAUTHORIZED.toString());
22    errors.add(FORBIDDEN.toString());
23    errors.add(NOT_FOUND.toString());
24    errors.add(SERVER_ERROR.toString());
25    errors.add(NOT_IMPLEMENTED.toString());
26    errors.add(SERVICE_UNAVAILABLE.toString());
27    errors.add(GATEWAY_TIMEOUT.toString());
28
29    return errors;
30  }
```

Code Sample 6: Constants group task 3 solution. First Method.

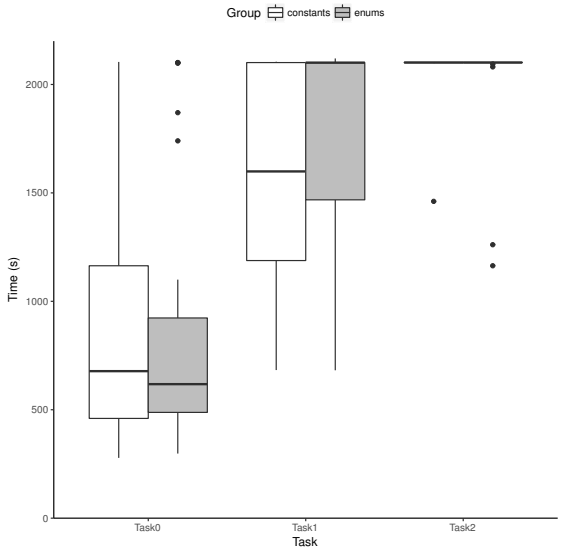

Figure 3.1: Boxplot showing time by task

hypothesis that enums make little difference might be correct. For example, even if the group effect would appear significant on a larger scale, the effect size is very small, so it would be unexpected if the impact would turn out to be very large.

To test the differences between the tasks, a post-hoc t-test with Bonferroni correction was run using the factor task. The test shows that there is a significant difference between task 1 and task 2 ($p < 0.001$), as

```
/**
 * TODO: Complete the method below.
 * - This method checks if requested URL is in the
 *    'webpages' list and returns a string result accordingly
 *    in the format <code>:<description>
 *
 * Consider using the method contains(String) for List<String>
 *
 * @param URL
 *    - requested URL
 * @return <code>:<description>
 *    - if URL is empty, error is BAD_REQUEST.
 *    - else if 'webpages' does not contain URL, error is NO_CONTENT
 *    - else, it is OK.
 */
public String get(String URL) {
  String response = """";
  if(isEmpty(URL)) {
    response = BAD_REQUEST.code+"":""+BAD_REQUEST.description;
  } else if(!webpages.contains(URL)) {
    response = NO_CONTENT.code+"":""+NO_CONTENT.description;
  } else {
    response = OK.code+"":""+OK.description;
  }

  return response;
}
```

Code Sample 7: Constants group task 3 solution. Second Method.

```
/**
 * TODO: Complete the method below.
 *  - This method adds the URL to 'webpages' list and returns
 *  a string result accordingly in the format <code>:<description>
 *
 * @param URL
 *    - URL to create
 * @return <code>:<description>
 *    - if URL is empty, error is SERVER_ERROR.
 *    - else, it is CREATED.
 */
public String post(String URL) {
  String response = """";
  if(isEmpty(URL)) {
    response = SERVER_ERROR.code+"":""+SERVER_ERROR.description;
  } else {
    response = CREATED.code+"":""+CREATED.description;
    webpages.add(URL);
  }
  return response;
}
```

Code Sample 8: Constants group task 3 solution. Third Method.

well as between task 1 and task 3 ($p < 0.001$) and task 2 and task 3 ($p < 0.001$).

File Switches: The number of switches between sample files detected in our experiment was also analyzed as an additional measure for effort, first a mixed designs repeated measures ANOVA with outcome file switches, group as between subjects variable and task as within-subject variable was run. According to Mauchly's Test for Sphericity, the sphericity assumption was violated. All reported numbers are corrected using the Greenhouse-Geisser corrections, as before. The ANOVA shows no significant interaction between group and task $F(2, 110) = 3.26,\ p = 0.056,\ (\eta_p^2 = 0.040)$, but there is a significant difference between the tasks $F(2, 110) = 14.32,\ p < 0.001,\ (\eta_p^2 = 0.143)$ and no significant difference for factor group $F(1, 55) = 1.84,\ p = 0.180,\ (\eta_p^2 = 0.010)$. Thus again, showing little impact of this feature in programming

```
1    /**
2     * TODO: Complete the method below.
3     * - this method replaces existing URL in 'webpages'
4     * with the newURL passed to the method.
5     * Consider using the methods indexOf(String) and set(int, String) for List<String>.
6     *
7     * @param URL
8     *    - existing url
9     * @param newURL
10    *    - url to replace with
11    * @return <code>:<description>
12    *    - if URL is empty, error is SERVER_ERROR.
13    *    - else if newURL is empty, error is BAD_REQUEST.
14    *    - else
15    *    - if URL is not found in 'webpages', error is NOT_FOUND
16    *    - else, replace URL with newURL and return OK
17    */
18   public String put(String URL, String newURL) {
19     String response = """";
20     if(isEmpty(URL)) {
21       response = SERVER_ERROR.code+"":""+SERVER_ERROR.description;
22     } else if (isEmpty(newURL)) {
23       response = BAD_REQUEST.code+"":""+BAD_REQUEST.description;
24     } else {
25       int i = webpages.indexOf(URL);
26       if(i == -1) {
27         response = NOT_FOUND.code+"":""+NOT_FOUND.description;
28       } else {
29         response = OK.code+"":""+OK.description;
30         webpages.set(i,newURL);
31       }
32     }
33
34     return response;
35   }
```

Code Sample 9: Constants group task 3 solution. Fourth Method.

languages.

Similar to the analysis for the time outcome, an addtional post-hoc t-test with Bonferroni correction was run on the factor task, which shows a significant difference between task 1 and task 2 ($p < 0.001$), and between task 1 and task 3 ($p = 0.001$), but not between task 2 and task 3 ($p = 0.334$). Addtionally, the Pearson correlation between file switches and task time was analyzed and found a correlation coefficient of $r = 0.46$ with $p < 0.001$. Which is a significant and fairly high.

3.4.3 Discussion of Results

The results of this experiment show a significant impact of the interaction between task and group, which seems to suggest that enumerated types might be more efficient for a programmer to use in some circumstances than constants, while they are worse in others. This can be seen in task 1, where participants have to define the enumerated types and use them in conditions. The task is simple compared to task 2 and task 3 and the participants in the enum group had a lower average time than participants in the constants group. On the other hand, the constant group had lower average times in task 2. Qualitative investigation into the difference suggests that this is because task 2 involves the creation of a type (either class or enum) with methods, and participants had trouble with how the *toString*() method in an enum would work, as

```
1       /**
2        * TODO: Complete the method below.
3        * - This method should put all the HTTP
4        * Errors in the following format into the list.
5        * <CODE:NAME:DESCRIPTION>
6        *
7        * for example:
8        * 200:OK:Action completed successfully.
9        *
10       */
11      public List<String> getAllErrors() {
12
13        List<String> errors = new ArrayList<String>();
14
15        errors.add(""CODE:NAME:DESCRIPTION"");
16
17        errors.add(HTTPErrorCodes.OK.toString());
18        errors.add(HTTPErrorCodes.CREATED.toString());
19        errors.add(HTTPErrorCodes.NO_CONTENT.toString());
20        errors.add(HTTPErrorCodes.MOVED_PERMANENTLY.toString());
21        errors.add(HTTPErrorCodes.BAD_REQUEST.toString());
22        errors.add(HTTPErrorCodes.UNAUTHORIZED.toString());
23        errors.add(HTTPErrorCodes.FORBIDDEN.toString());
24        errors.add(HTTPErrorCodes.NOT_FOUND.toString());
25        errors.add(HTTPErrorCodes.SERVER_ERROR.toString());
26        errors.add(HTTPErrorCodes.NOT_IMPLEMENTED.toString());
27        errors.add(HTTPErrorCodes.SERVICE_UNAVAILABLE.toString());
28        errors.add(HTTPErrorCodes.GATEWAY_TIMEOUT.toString());
29
30
31        return errors;
32      }
```

Code Sample 10: Enums group task 3 solution. First Method.

well as where the enum definition was supposed to be put (inside or outside the enum construct) and how to properly define an enum using a constructor. This effect accounts for 3.00% of the variance observed in this study. There was also a significant differences between the tasks, which account for 55.47% of the variance in the experiment. Our post-hoc test found that there are significant differences between all three tasks. This tells us that even if enums help, their impact is clearly small and task dependent.

Overall, it is important to be careful in the analysis of this experiment. The intent in running this study was never to show evidence that enums are good or bad. The goal was to run a test trying to **detect** an effect, but a non-significant p-value here does not signal a failure. Further, these results do not refute the stated principles. On the one hand, the measures were partially indirect. Enumerated types do have different syntax than constants, meaning the annotations are not identical. However, importantly they give very similar information to the reader. In essence, this study does not prove that annotations are the cause, but this test was designed with a direct intent to disprove the theory, but in this case the results came out looking more confirmatory.

Connecting these observations with the topic of this dissertation, we can see that comprehension of code is a major factor in programming productivity, which makes code comprehension a major factor in determining the impact of programmig language switching in polyglot programming. Based on this, from the viewpoint of linguistics, comprehension of language —opposed to production of language —is the most im-

```
1       /**
2        * TODO: Complete the method below.
3        * - This method checks if requested URL is in the
4        *  'webpages' list and returns a string result accordingly in the
5        * format <code>:<description>
6        *
7        * Consider using the method contains(String) for List<String>
8        *
9        * @param URL
10       *          - requested URL
11       * @return <code>:<description>
12       *          - if URL is empty, error is BAD_REQUEST. -
13       *          - else if 'webpages' does not contain URL, error is NO_CONTENT -
14       *          - else, it is OK.
15       */
16      public String get(String URL) {
17
18        String ans;
19
20        if (URL == """ || URL == null)
21          ans = (HTTPErrorCodes.BAD_REQUEST.code + "":"" + HTTPErrorCodes.BAD_REQUEST.description);
22        else if (webpages.contains(URL) )
23          ans = (HTTPErrorCodes.OK.code + "":"" + HTTPErrorCodes.OK.description);
24        else
25          ans = (HTTPErrorCodes.NO_CONTENT.code + "":"" + HTTPErrorCodes.NO_CONTENT.description);
26
27        return ans;
28      }
```

Code Sample 11: Enums group task 3 solution. Second Method.

portant aspect to investigate when looking at possible costs when switching between programming languages.

3.4.4 Threats to Validity

The web platform was not bug free during the course of this experiment. After the conclusion of the experiment, it became obvious that some participants encountered bugs which prevented them from moving forward in the experiment after finishing a task. All of the affected participants had to be removed from the data set, which has increased the drop-out rate of the experiment much more than was anticipated and it might have affected the outcome of the experiment. It is also possible that the results from the web platform can not be generalized to users with IDEs, because the text box did not have any special functions such as syntax highlighting and code completion.

This experiment also suffered from problems with the time limit being too short for many of the participants. This has an obvious ceiling effect on the results. The time limit was such a big problem in the third task of the experiment that the task had to be removed from analysis, as participants could not finish the task on time. Any replication of this experiment should increase the time limit. To be clear, even with careful piloting, it can be very difficult to guess the max values for the experiment ahead of time, which have to be reported to the ethics board before running a study.

The participants in this experiment were all trained to use the C++ programming language, however the experiment was conducted using Java. While there are many similarities in the two languages, they are

```
1       /**
2        * TODO: Complete the method below. - This method adds the URL to 'webpages'
3        * list and returns a string result accordingly in the format
4        * <code>:<description>
5        *
6        * @param URL
7        *       - URL to create
8        * @return <code>:<description>
9        *       - if URL is empty, error is SERVER_ERROR.
10       *       - else, it is CREATED.
11       */
12      public String post(String URL) {
13
14        String ans = """";
15
16        if (URL == """" || URL == null)
17          ans = (HTTPErrorCodes.SERVER_ERROR.code + "":"" + HTTPErrorCodes.SERVER_ERROR.description);
18        else {
19          webpages.add(URL);
20          ans = (HTTPErrorCodes.CREATED.code + "":"" + HTTPErrorCodes.CREATED.description);
21        }
22
23        return ans;
24      }
```

Code Sample 12: Enums group task 3 solution. Third Method.

distinct in a number of areas as well. The experiment samples were meant to provide participants with all parts of the Java syntax that were necessary to solve the tasks by giving them a number of code samples, but in a post-hoc qualitative analysis of the submitted code it was found that that there was some confusion for some of the participants. For example, some participants tried to debug their solutions using print statements, but used "*cout*" instead of "*System.out.println*". Others initially spelled "*String*" as "*string*". From looking at the data, it seems doubtful that these issues would change our conclusions of what the data is suggesting, but a variety of situations ultimately need to be tested to validate or refute any theory. For example, while the programming tasks in this experiment were mean to be fairly realistic, they are still tasks designed for an experiment and might not be fully generalizable to actual programming work in industry or, similarly, in the classroom.

3.5 Conclusion

In this chapter, a theoretical explanation for the superior performance of some static type systems over dynamic type systems that have been documented in the literature has been developed. To test the explanation, a new empirical experiment designed to attempt to disprove the explanation was conducted. The experiment found no significant performance difference between similarly annotated groups using either enums or constants to solve the same tasks.

Overall, these findings indicate that the explicit information from type annotations makes code easier to comprehend which leads to shorter task times. This effect can be found in many studies comparing statically typed and dynamically typed languages and accounts for cases where certain forms of static typing performs similarly to dynamic typing. It can be proposed that comprehension time is a major driver in task times,

```
1       /**
2        * TODO: Complete the method below.
3        * - this method replaces existing URL in 'webpages'
4        * with the newURL passed to the method.
5        * Consider using the methods indexOf(String) and set(int, String) for List<String>
6        *
7        * @param URL
8        *     -  existing url
9        * @param newURL
10       *     - url to replace with
11       * @return <code>:<description>
12       *     - if URL is empty, error is SERVER_ERROR.
13       *     - else if newURL is empty, error is BAD_REQUEST.
14       *     - else
15       *     - if URL is not found in 'webpages', error is NOT_FOUND
16       *     - else, replace URL with newURL and return OK
17       */
18      public String put(String URL, String newURL) {
19
20        String ans = """";
21
22        if (URL == """"  ||  URL == null)
23          ans = (HTTPErrorCodes.SERVER_ERROR.code + "":"" + HTTPErrorCodes.SERVER_ERROR.description);
24        else if (newURL == """"  ||  newURL == null)
25          ans = (HTTPErrorCodes.BAD_REQUEST.code + "":"" + HTTPErrorCodes.BAD_REQUEST.description);
26        else {
27          if (!webpages.contains(URL) ) {
28            ans = (HTTPErrorCodes.NOT_FOUND.code + "":"" + HTTPErrorCodes.NOT_FOUND.description);
29          } else {
30            int idx = webpages.indexOf(URL);
31            webpages.set(idx, newURL);
32            ans = (HTTPErrorCodes.OK.code + "":"" + HTTPErrorCodes.OK.description);
33          }
34
35        }
36
37        return ans;
38      }
```

Code Sample 13: Enums group task 3 solution. Fourth Method.

	Task 1					
Group	N	Min	Max	Mean	Median	Std. Dev.
Constants	29	278	2100	939.07	678.0	676.25
Enums	28	298	2100	845.32	618.0	573.73

	Task 2					
Group	N	Min.	Max.	Mean	Median	Std. Dev.
Constants	29	683	2100	1557.69	1599.0	474.27
Enums	28	682	2100	1808.96	2100.0	418.78

	Task 3					
Group	N	Min.	Max.	Mean	Median	Std. Dev.
Constants	29	1461	2100	2079.31	2100.00	118.92
Enums	28	1164	2100	2036.93	2100.00	233.26

Table 3.2: Times per task in seconds

	1	2
2	$3.0 * 10^{-15}$	-
3	$< 2 * 10^{-16}$	$1.4 * 10^{-7}$

Table 3.3: T-test with Bonferroni correction between tasks

with the implication that improvements in comprehension may be worthwhile in a trade-off to negative effects to other aspects like changeability or verbosity.

This chapter showed that there is a significant difference in productivity based on the context information provided in a given piece of code, which brings code comprehension to the forefront of topics to consider when investigating computer language switching.

Chapter 4

Randomized Controlled Trial on the Impact of Embedded Computer Language Switching

This chapter will present the design and results of pilot runs of an experiment which was created to evaluate the differences between the task times of three groups which require different amounts of language switching to complete. Each of the three groups is based on a different approach to integrating querying capabilities into Java.

4.1 Introduction

As discussed in the introduction in chapter 1 and in section 2.1, polyglot programming is a common occurrence in today's software development industry, so common, that ninety-seven percent of open source projects analyzed used two or more computer languages [TT14] and average number of computer languages in open source projects is about five [MB15, TT14]. Developers report that they know ten different computer languages [MR13].

With programming in different languages comes the need to switch between the languages a developer is using at a given time. Section 1.1 describes three different levels of switch, the project level switch, the file level switch, and the embedded switch. The project level switch occurs when a developer switches between projects that are using different languages. The file level switch occurs within a project or work context that uses different languages and can happen more or less rapidly, and the embedded level switch can occur within the same file when a secondary language is integrated into a host language. Quick versions of file level

switching and embedded level switching can be considered parallels to the concept of intra-sentential code switching[1] [HA01, Li96, YTF17], a rapid switch between natural languages in conversation, while slower file level switching can be considered a parallel to inter-sentential code switching[1] [HA01, Li96, YTF17], which is the process of switching languages between utterances. Project level switches are a possible parallel to natural language switching[1] [AG08]. Research of code switching in linguistics has shown that there is a cost to switching between natural languages [Ols16, Ols17] which elicits the hypothesis that a similar phenomenon could be found for switching between computer languages as well, considering findings that program comprehension and natural language comprehension seem to use the same areas of the brain [SKA+14, SPP+17].

If a switching cost between computer languages is found, recommendations could be made to avoid the process as much as possible to increase programmer productivity or to accommodate developer switches with better tools. Recommendation and research of tools to increase productivity is also common for concerns such as compiler errors [Bec16b]. Another possible solution might be the recommendation of work flows that reduce language switches, for example by making sure that developers work in a way that switching is more commonplace, which has been found to reduce switching cost in natural language code switching [Ols17]. The productivity of programmers has much importance since software is a $407.3 billion industry [gar] and when considering that the median salary for a software developer is $103,560 per year [BLS], small improvements can have large impacts on the cost of software development. Programmer productivity research is also conducted on programming language features [PT98, Han10b, HH13a] and API design [SM08, ESM07], which is strongly connected to this study, as code switching is studied in an API context.

A model explaining the switching cost in natural languages is the Inhibitory Control Model (ICM) [Gre98, Ols16], which describes switching between languages as a process of mentally inhibiting the languages not in use, instead of activating a single language. According to the model, the more a language is inhibited, the longer it takes to reduce the inhibition and switch back to this language. This model explains the findings that there are lower switching costs between switching from a dominant to a non-dominant language than when switching from a non-dominant language to a dominant language [Ols17], which might seem counter-intuitive as one would suspect that switching languages to one's dominant language should be easiest. According to the Inhibitory Control Model, to speak in the non-dominant language, one has to inhibit the dominant language far stronger and thus, switching back is more difficult. The model might also explain the finding that speakers in contexts in which they are required to switch back-and-forth between languages exhibit next to no switching costs [Ols17]. In the Inhibitory Control Model this finding is explained by the idea that in switching situations, the different languages are only inhibited slightly, which makes switching between languages easier. Other, more simple models like a model based on a two switch system [MK71], one for input and another for output of language, have difficulty explaining theses scenarios. This might serve

[1] see section 2.4.2 for a discussion.

as an explanatory model for programming language switching cost, if such a cost is found. If the results of the experiment show a time difference between no switching and switching, but less of a difference between no switching and frequent switching, then the results might be related to the second described scenario in which switching cost is low in contexts of regular switching activity.

This experiment focuses on the lowest level of computer language switching connected to polyglot programming, the embedded switch. A common example of such switching is the use of SQL to query databases from general purpose programming languages. An example of an established tool allowing for the use of SQL to query databases is the JDBC (Java Database Connectivity) API[2]. Code sample 14 shows an example of a simple JDBC query in which a connection to the database has already been established. While JDBC also allows for more secure prepared statements, in which a statement is created with placeholders that can then be filled with parameters that are cleaned from potentially harmful code (protecting from SQL injections[3]), this study will focus on the simpler mechanism of complete queries in one string as the basic form of database querying.

```
1  stmt = conn.createStatement();
2  ResultSet rs = stmt.executeQuery("SELECT id, first, last, age FROM People");
```

Code Sample 14: Example of a simple JDBC query

To investigate the possibility of a switching cost, a double-blind randomized controlled trial with a number of programming tasks was designed to be conducted with student and professional programmers. To model different amounts of language switches, three different versions of a querying API were created, in which switches between Java as the host language and SQL as a second language would be more or less frequent.

4.1.1 Objective

The experiment described in this chapter aims to compare if differing amounts of computer language switching impacts the development time of software by utilizing three APIs that vary in the amount of language switches needed to complete programming tasks. Specifically, an object-oriented query API will be compared to an API which integrates SQL as a string, and a hybrid between the two approaches. The object-oriented API will require no language switches and will be exclusively utilizing Java, while the string-based API requires language switches from Java to SQL and back when a query is being written. The hybrid approach requires more frequent language switches than the string-based API. The different API designs will be discussed in more detail in section 4.2.4. To keep the tasks simple for participants, this study will only focus

[2] `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`
[3] `https://en.wikipedia.org/wiki/SQL_injection`

on the implementation of simple queries. Further aspects of database programming using a general purpose language, such as complex queries (for example nested queries and aggregation functions), making database connections, and other operations are not relevant for the problem of polyglot programming.

The null-hypotheses are as follows:

$H_0$1: *There is no relationship between amount of computer language switching and productivity.*

$H_0$2: *Programmers do not notice consciously that they switch between computer languages.*

$H_0$3: *There is no difference between the productivity of native English speakers and programmers with a different native language.*

The alternative hypotheses can be derived from the null-hypotheses. The first hypothesis is connected to the first main research question, asking whether there is a cost to switching between two computer languages. In this experiment it was chosen to test this cost by checking for time difference between groups. It is possible to hypothesize that frequently switching between languages might increase mental strain and reduce productivity over time, which can be tested in future experiments.

The second hypothesis will only be analyzed qualitatively based on an exit survey in the experiment. The exit survey contains a question asking whether participants noticed that they switched between languages or not and whether they felt that this switching impacted their progress. Investigation of the experience of participants on the issue might help to determine whether participants consider switching between computer languages as an issue.

The aim of hypothesis three is to investigate the impact of natural languages on programming productivity. Recording participants' primary language will be necessary to be able to compare the performance of primary English speakers and other participants. The main measure for this hypothesis will be the time to a correct solution.

4.1.2 Design Progress

To make sure that the experiment is well designed, it was piloted extensively. That means, that the design of the experiment was iterated on and at in early piloting attempts the number of participants in the study was doubled each iteration according to the doubling rule. In five preliminary pilot studies, the experiment was tested with 2, 4, 12, 9, and 11 participants respectively. The results of each pilot study were analyzed to detect problems with the methodology. Specific focus of these iterations lay on general task design and on the design of the API. The first three iterations had only the string-based and the object-oriented group,

after these initial iterations were completed, a hybrid between the groups was conceived in pilot 4 and 5. The hybrid between the groups requires more frequent back-and-forth switching between the languages.

After the first three preliminary pilot studies, the experiment was ported onto a experimentation platform, whereas it had first been conducted using an unmodified Eclipse IDE, which produced less accurate measurements and was limited to only measuring time. After these modifications were made, two runs of the experiment were conducted. The first run of the experiment had 9 computer science majors in their senior year. In this attempt at running the study, participants encountered several errors that were caused by unexpected inputs into the query mechanism. The experiment code returned unhelpful error messages based on these unexpected inputs, which made solving the tasks extremely difficult to complete. The problems were fixed for the fifth run of the study by adding numerous error conditions to the software.

The fifth run of the study had 11 respondents, most of which were Sophomores. This pilot test showed a significant bug in the testing environment making the results unusable to make conclusions. Analysis of pilot tests 4 and 5 can be found in appendix A.

4.1.3 Structure

The layout of this write-up is loosely inspired by the CONSORT (Consolidated Standards of Reporting Trials) standard as it is used in the medical sciences [con], which defines what should be included in the publication of a research study to allow for "complete and transparent reporting." Hence, the rest of this chapter is going to be structured as follows: First, it will go over the design of the experiment in section 4.2. That section will include information about the design of the trial in subsection 4.2.1, the participants in subsection 4.2.2, the study setting in subsection 4.2.3, the intervention in subsection 4.2.4, the outcome variables in subsection 4.2.5, the sample size in subsection 4.2.6, the randomization in 4.2.7, and the blinding in subsection 4.2.8. Results will be shown and discussed in subsection 4.3, followed by qualitative results in section 4.4 and a discussion in section 4.5, finally the chapter will end in a conclusion in section 4.6.

4.2 Methods

This section will describe the details of the design of the experiment. Subsection 4.2.1 will describe the experiment on a high level, subsection 4.2.2 describes the recruitment process, subsection 4.2.3 describes the conditions of the experiment, subsection 4.2.4 focuses on the details of the intervention of the experiment, subsection 4.2.5 explains the measured outcomes, subsection 4.2.6 talks about the sample size of the experiment, subsection 4.2.7 explains the randomization mechanism used, and finally, subsection 4.2.8 focuses on blinding.

4.2.1 Trial Design

The described experiment is a repeated measures design in which participants were randomly assigned to one of three experimental groups. Participants were asked to solve 6 programming tasks in each group. The randomization was based on the participants' year in college, or alternatively their status as a professional developer. The experiment tested the effect of using an object oriented API against the established use of SQL strings and a hybrid approach.

4.2.2 Participants

Eligible to participate in the study were persons over the age of 18 years who had at least some programming experience. The programming experience was self-reported by the participants in a survey included in the study. The participants were recruited in computer science classes at the University of Nevada, Las Vegas, in the case of students, and on Twitter and Reddit in the case of professional developers. A factor used to distinguish participants was level of education, which is a common measure for programming experience for university students [SKL+14b] and significant differences between levels have been found in respect to time to solution [USH+16]. The participants were informed about the study during class time by a researcher reading the advertisement pamphlet while every student was given a copy. Then, students were able to go to the URL posted on the pamphlet and start the experiment whenever they had time. Students were offered extra credit for participation in the experiment and the amount of extra credit was based on what the professor was willing to give the students and ranged between 0-3% of total class points. Alternatively to participating in the study, students were able to achieve the same amount of extra credit by submitting an essay on a computer science topic.

4.2.3 Study Setting

The study was conducted using the EPI (Experiment Platform for the Internet) online platform which informed the participants about their rights and recorded their consent to participate. Then, the participants were asked to fill out a survey to classify them into one of the experience groups by college status (undergraduate year, graduate, post-graduate, non-degree seeking, or professional). The survey also recorded some additional information about the participant for analysis purposes, such as their total amount of programming experience and their primary natural language, as well as if they have any disabilities. When the survey was completed, participants were then informed about the details of the study by being shown the experiment protocol.

When the participants were done reading the study protocol, they were shown the screen of the first task on which they had 5 minutes to read the code sample and then move on to try to solve the coding task. The participants were able to refer back to the code sample during the coding phase of the task. The coding screen was a text box with some code already loaded into it, to give the participant some scaffolding for the solution and a framework on which automatic testing could be based. The coding screen also showed a timer with the remaining task time on it, as all tasks were limited to 45 minutes per task to prevent the experiment from taking too much time, as a maximum time commitment of four and a half hours was promised to the participants. When the participants felt that they had solved the task sufficiently, they could click the "Check Task" button. Then, the code they had written was sent to the server. On the server the code was compiled in combination with the other classes needed and then run against a test-case. The test-case determined if the task was solved successfully. If the code was sufficient, the output window on the page printed that the test was successful and then the website showed an overlay, telling the participants that they can move on to the next task with the click of a button. If the test was not successful, then the output of the test-case displayed in the output window. Participants were able to keep going with the same task until they either successfully solved it or until they run out of time for the specific task. Once all of the tasks had ended, participants asked to give some feedback on the experiment and then were thanked for their participation.

To test the difference between traditional SQL-like, string-based query building and the new design of query building using objects, as well as the hybrid approach, a small table library was built using the Java programming language. The table contains data in columns and the queries written in this experiment are all run using table objects, so that this table class takes over the role of the actual database. The table class design is part of the design idea for a new data management library, which is partly tested in this experiment. To keep everything as similar as possible in this experiment, the SQL-like queries were parsed and transformed to a similar kind of query object as the object-oriented API and hybrid API used.

4.2.4 Intervention

Three different groups were designed to represent different levels of language switching. The design of the experiment was additionally motivated by the desire to test different ways to design an API that allows to query a database and thus, the design is centered around different ideas of approaching the querying while also differing in the amounts of language switches that are necessary.

The first approach is the simple use of SQL strings for the API to send to the database for requests. The second approach was to create an API that allows users to create a query by building it from objects in a series of method calls, which adheres more closely to more typical object-oriented programming techniques. Then, a hybrid approach was devised. The hybrid approach uses a more object oriented approach, but

features the use of SQL-based strings for query conditions.

```
1  public Table query(Table table) throws Exception {
2
3      Query query = new Query();
4      q.Where("value1").LessThan(234).And("value2").GreaterThan(42)
5      query.Prepare("SELECT Field1, Field2 "
6          +" FROM table WHERE Field1 < 234 AND Field2 > 42 "
7          +" ORDER BY Field3 DESC");
8
9      Table result = table.Search(query);
10
11     return result;
12 }
```

Code Sample 15: Example of the String-based Design

The first design (see example in listing 15) using the string approach requires a user of the API to know the exact syntax of the desired SQL query they want to write and does not feature any amount of type checking support for the query. All error checking for this approach is done by the database and the programmer would be required to rely on the feedback of the database in locating any possible errors that might occur (for details on SQL errors see [TSV18]). Further, the SQL dialect in use on the side of the server must be considered and matched completely. On the other hand, any user with sufficient SQL expertise will be able to write a query using the full flexibility of the SQL dialect in use. This approach also enables the use of modularization in which the SQL queries are stored in external files instead of within the compiled source code, enabling change of the queries without recompilation of the source code. From the perspective of the polyglot programming levels (see 1.1), this approach would be considered embedded switching. The programmer would be generally programming in Java and switching to SQL when starting to write the query. When done writing the SQL query, the developer would then switch back to Java.

```
1  public Table query(Table table) throws Exception {
2
3    Query query = new Query();
4
5    query.AddField("Field1")
6      .AddField("Field2");
7
8    query.Filter(q.Where("Field1").LessThan(234).And("Field2").GreaterThan(42));
9    query.SortHighToLow("Field3");
10
11   Table result = table.Search(query);
12 }
```

Code Sample 16: Example of the Object-Oriented Design

The second design (see example in listing 16) requires a programmer to use a number of method calls to produce a query. This approach loses the flexibility that comes with using simple strings by removing the possibility to load the strings from a file or other source. However, the evaluation of the statement is first made in the programming language itself and type checking and runtime error checking can be leveraged to rule out a number of syntax errors that can be made in the string only design, which might improve productivity. Another difference in this approach is, that programmers do not have to switch between programming

languages to write a query. This might impact productivity because of the avoidance of switching costs.

```
1  public Table query(Table table) throws Exception {
2      Query query = new Query();
3
4      query.AddFields("Field1, Field2");
5
6      query.Filter("Field1 < 234 AND Field2 > 42");
7      query.SortHighToLow("Field3");
8
9      Table result = charts.Search(query);
10
11     return result;
12 }
```

Code Sample 17: Example of the Hybrid Design

The third design is a hybrid between the two approaches in which the query building process is separated into different method calls, but combines steps such as adding multiple fields by allowing the programmer to write a comma-separated list similar to what they would do in a SQL query. Notably, this approach avoids the use of objects to build a filter statement and requires the use of SQL syntax. This gives the programmer all the flexibility of a condition used in SQL, but also requires knowledge about the conditional syntax. This approach takes care of the general structure of a query but requires some use of the SQL syntax and requires more programming language switches from a programmer than the first approach did. However, programmers in this case might not realize that what they are writing is part SQL and might consider the syntax in the string as a local DSL.

One major factor in choosing the designs was the idea that there is a spectrum of how a language can be integrated into a host language. On one side of the spectrum, there is the integration of a DSL into a general purpose language unchanged, and without a direct connection between the two languages. This can be seen in the string-based approach, the general purpose programming language and the tools used by the programmer are unaware of the embedded language. All errors will be produced by the engine which executes the DSL code. In the case of a database, the instance trying to execute the code is only connected to during run-time and in a different process, possibly even connected over a network, which might make testing slow and requires a running database instance.

On the other side of the spectrum is full integration of the secondary language into the general purpose language. The DSL is abstracted into concepts used in the general purpose language, which then allows for automatic error checking based on the capabilities of the general purpose language. In some cases this may also allow for unit testing. In the case in this experiment, the DSL was abstracted into objects and method calls.

```
1  package library;
2
3  import library.*;
4
5  public class Task1 {
6
7      /**
8       * Please write this method to return a Table object containing all columns
9       * for all entries with an id smaller than 32 and sorted from high salary
10      * to low salary
11      *
12      * Table information:
13      *
14      * - prof -
15      *
16      * id (int) | firstname (String) | lastname (String) | salary (int)
17      *
18      *
19      *
20      * Use the technique shown to you in the samples given
21      *
22      */
23     public Table query(Table prof) throws Exception {
24
25         // Your Code here
26
27         return null;
28     }
29
30 }
```

Code Sample 18: Task 1 as presented to the participants.

The difference in intervention in this experiment is based on which kind of code sample the participants were exposed to. Each group got one compilable code sample demonstrating a number of different queries in the library. Each sample includes code equivalent to four different $SELECT$ statements in SQL, one $UPDATE$ statement and one $INSERT$ statement in SQL. The first select statement was a selection of all columns with ordering of entries from high to low, the second was a simple select of a number of columns with a where condition, the third included a join, and the last select statement was a simple selection of multiple columns with a where condition and a sorting statement. The $UPDATE$ and $INSERT$ statements were simple versions of their respective types. The difference between the samples is that each sample shows the code required to use the specific approach that was being tested in the respective group. See listings 72, 78, and 82 in appendix A to examine the entire samples the way they were presented to the participants.

```
1      public Table queryB(Table prof) throws Exception {
2          Query q = new Query();
3
4          q.Prepare("SELECT * FROM professors" +
5          " WHERE id < 32 ORDER BY salary DESC");
6
7          Table r = prof.Search(q);
8
9          return r;
```

Code Sample 19: Task 1 Solution for group SQL.

An example of what the tasks looked like to the participants can be seen in code sample 18, which shows the first task of both groups. The instructions are in the comment at the top and the code to solve the task has to be filled in where the command says “Your code here”. All other tasks had the same empty method

```
1
2       public Table queryC(Table prof) throws Exception {
3           Query q = new Query();
4
5           q.SortHighToLow("salary");
6           q.Filter(q.Where("id").LessThan(32));
7
8           Table r = prof.Search(q);
9
10          return r;
```

Code Sample 20: Task 1 Solution for the object-oriented group.

```
1
2       public Table queryA(Table prof) throws Exception {
3           Query q = new Query();
4
5           q.SortHighToLow("salary");
6           q.Filter("id < 32");
7
8           Table r = prof.Search(q);
9
10          return r;
```

Code Sample 21: Task 1 Solution for the hybrid group.

structure with an instructional comment at the top. The comments always also described the structure of the table object. Possible solutions to the first task as shown in listing 18 can be seen in listings 19, 20, and 21. Additional code that did not fit into this chapter can be found in appendix A. The comments for tasks 2-6 can be found in code samples 52 to 56. Possible solutions to the tasks are listed in code samples 57 to 71.

4.2.5 Outcomes

The first dependent variable of this experiment, time to a correct solution, was measured by taking a time stamp when the participant started a task and a time stamp when the correct solution was submitted. Alternatively, if the participant did not finish the task, the time stamp of the moment the time ran out was taken as the endpoint of the measurement. The difference between the two time stamps was then used as the time to correct solution measure. The experiment platform in use for the experiment automatically measured the task times and saved them to a database together with timestamped snapshots of the code each participant produced. As a random factor, the platform also recorded the participants' experience in using SQL and if they had taken a database management systems class, which was then combined into a binary measure representing whether a participant had database experience or not.

Additionally to the quantitative measures collected in this experiment, participants were also asked questions about their opinion on elements of the experiment. The pilot run of the experiment asked the following three questions:

- "Which of the concepts in the study did you find difficult to understand?"
- "Was there anything about the design of the programming language (not the study itself) that would have made these tasks easier?"
- "If you have any other comments or feedback, please type it here:"

The response to which will not be analyzed in detail in this chapter, but provide a way for the experimenter to check for bugs or inconsistencies in the experiment that are not readily apparent from the other recorded data.

Finally, a question about switching computer languages has been added to help gain insight about if participants realized that they were working in two different languages. The exact wording of the question was:

- "Did you feel like you had to switch between languages during the experiment and how do you think did this affect your progress while solving the tasks?"

4.2.6 Sample Size

The experiment has been conducted in five different versions before the current one, having 2, 4, 12, 9, and 11 participants respectively. The results from previous versions of the experiment can not be compared to the current version as changes affecting the results have been made. There are 5 viable levels of education to target for this experiment: sophomores, juniors, seniors, graduate students, and professionals.

4.2.7 Randomization

The process of assigning participants to the three groups happened entirely on the online platform and followed the covariate adaptive randomization approach [Sur11]. After entering their college year or professional status into the survey at the beginning of the experiment, the participants were assigned to a experience category based on that information. The platform kept track which groups were already assigned to in each category until each group was assigned once, then all groups were free to be assigned again until each has been filled again. This mechanism was in place to keep the distribution to the groups as even as possible.

4.2.8 Blinding

The experiment was designed to be double blind. The assignment of participants to their group was automatic. The researchers could not determine who was assigned to which group. It was possible for the researchers to tell who was assigned to which group after the participants finished the experiment, as this

was tracked in the collected data. Since the experiment was conducted online, there was no direct interaction with the participants and therefore the proctors had fewer avenues to accidentally or intentionally bias them.

The participants were not informed about which group they were assigned to or what the hypothesis of the study might be. They were only aware of the information right in front of them during the experiment. Information about the content of the experiment at time of recruitment was limited to the fact that the participants are being recruited to participate in a programming experiment, but no information about the topic was provided.

4.3 Quantitative Results

This section will present the results of the experiment. Analysis of data from two of the pilots is available in appendix A. For this experiment 149 participants were recruited within computer science classes at the University of Nevada, Las Vegas as well as professional software developers. The script used to analyze the data can be found at `https://bitbucket.org/stefika/replication/src/master/2019UesbeckDissertation/embedded`.

4.3.1 Recruitment

We recruited 149 participants for this study. Of the 149 participants, 40 had to be excluded from analysis for not having finished all 6 tasks or clearly not following the rules of the experiment (such as waiting out the experiment until the end without taking actions), leaving 109 participants. 12 of the participants were classified as freshmen, 23 as sophomores, 36 as juniors, and 29 as seniors. Additionally, 9 were professionals. Of the 109 participants, 36 identified as female. On average, the participants were 24 years old (M = 23.74, SD = 5.28). Of the participants in the experiment, 38 were in the hybrid group, 35 were in the polyglot group, and 36 were in the object group. Of the 109 participants, 14 had previous database experience. Eight of those experienced participants were professionals, 5 were seniors, and the last was a junior. Twenty-seven participants (32.92%) indicated that English was not their first language.

4.3.2 Baseline Data

An overview of the participants' average time per group and per task can be found in table 4.1. On average, it took participants 30 minutes (M = 1769.50s, SD = 931.50) to solve task 1, 26 minutes to solve task 2 (M = 1571.20s, SD = 1005.40), 32 minutes for task 3 (M = 1894.91s, SD = 932.56), 19 minutes for task 4 (M = 1122.88s, SD = 1083.42), 16 minutes for task 5 (M = 951.77s, SD = 1079.98), and 21 minutes for task 6 (M = 1244.41s, SD = 1035.28). Meaning that task 3 was the longest, while task 5 was the shortest task

Table 4.1: Times per task in seconds

	Object-Oriented			String-based			Hybrid			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 1	36	2016.11	881.60	35	1633.60	898.97	38	1661.05	982.94	1769.50	931.50
Task 2	36	1763.22	988.24	35	1268.11	1040.29	38	1651.87	955.42	1571.20	1005.40
Task 3	36	2080.78	833.97	35	1718.71	965.51	38	1881.11	980.18	1894.91	932.56
Task 4	36	1394.58	1134.82	35	1073.06	1146.47	38	911.36	938.12	1122.88	1083.42
Task 5	36	1228.86	1203.05	35	1073.60	1135.52	38	577.05	785.88	951.77	1079.98
Task 6	36	1457.44	1072.06	35	1360.17	1031.53	38	935.97	953.41	1244.41	1035.28

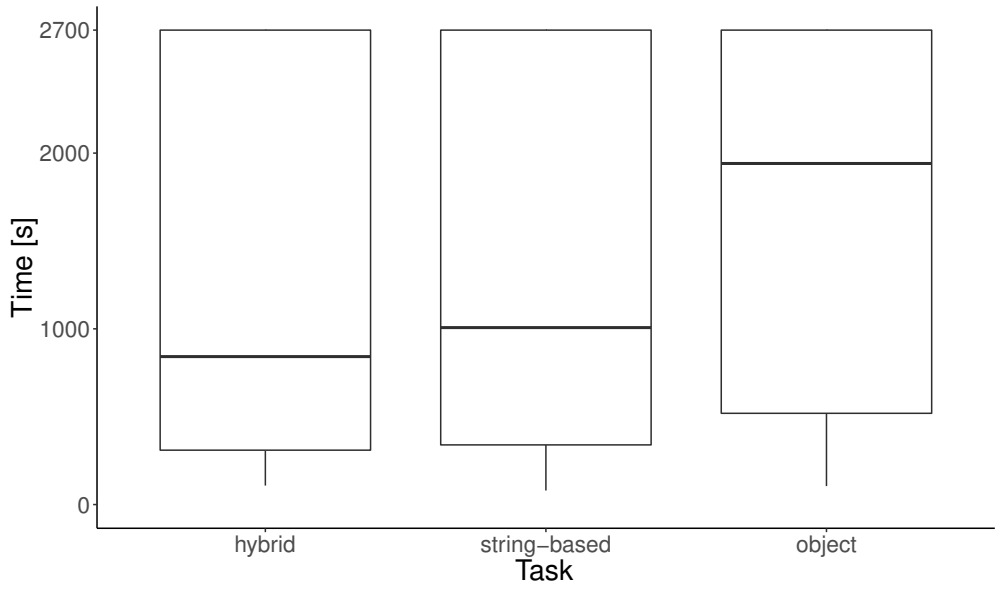

Figure 4.1: Boxplot of results between the groups

on average. When comparing the groups, the average time per task was the highest for the object-oriented group (M = 1656.83s, SD = 1043.20). The average task time was the second highest in the string-based group (M = 1357.54s, SD = 1057.53), while the hybrid group had the shortest average task time (M = 1269.74s, SD = 1043.20). Figure 4.1 shows the average task times between the three groups. Figure 4.2 shows the difference in task times broken down by level of education.

As the occurrence of participants not being able to finish the task in the limited time given to them is common in this experiment, taking a closer look at this phenomenon is warranted. Of the 109 participants, 60.55% encountered a task they couldn't finish completely. Of the 654 task instances worked on by the 109 participants (6 tasks and therefore 6 data points per participant), 35.62% weren't completed in time, making the average amount of tasks remaining uncompleted for each participant about 2 (M = 2.14, SD = 2.44). When broken down to which groups missed the most tasks, the object-oriented group failed to complete the most tasks with 44.91% of all task instances remaining uncompleted, the string-based group missed 33.33%

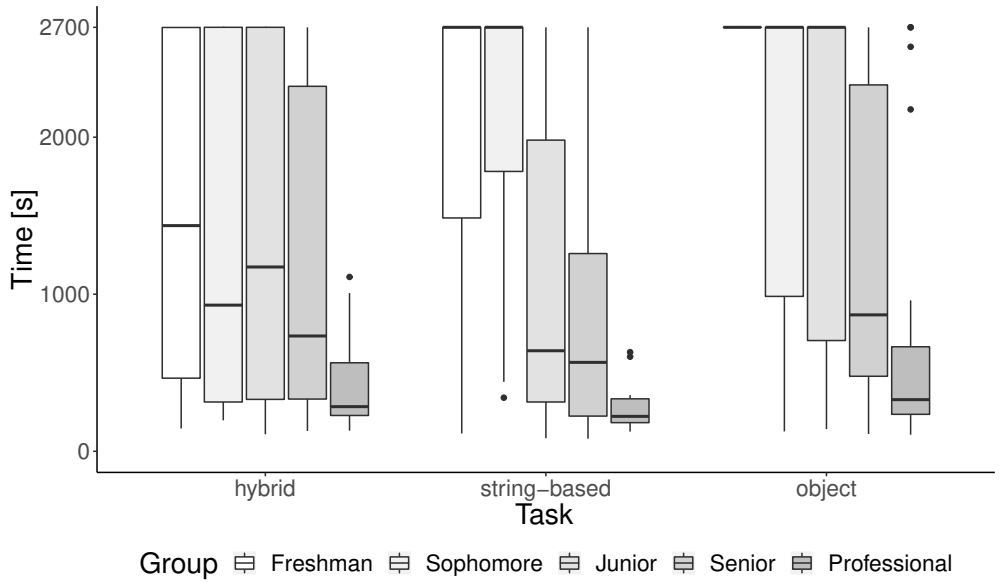

Figure 4.2: Boxplot of results between the groups based on their level of education

of tasks, and the hybrid group missed 28.95% of tasks. A breakdown of the percentages of failed tasks per group by level of education can be seen in figure 4.3.

4.3.3 Analysis

To analyze the results, a mixed designs repeated measures ANOVA was run using the R programming language with respect to time to solution, using task as a within-subjects variable and group and level of education as between-subjects variable. Sphericity was tested using Mauchly's test for sphericity, which shows that the assumption of sphericity was violated for the variable task, the interaction between group and task, level of education and task, and the three-way interaction between group, task, and level of education. Following reported numbers are reported with Greenhouse-Geisser corrections taken into account.

There are significant effects at $p < 0.05$ for the within-subjects variable task, $F(5, 470) = 28.83,\ p < 0.001$ ($\eta_p^2 = 0.064$), as well as for the between-subjects variable group, $F(2, 94) = 3.69,\ p = 0.029$ ($\eta_p^2 = 0.039$). There was also a significant effect for year $F(4, 94) = 8.05,\ p < 0.001$ ($\eta_p^2 = 0.175$) The interaction effect of group and task was significant, $F(10, 470) = 2.66,\ p = 0.008$ ($\eta_p^2 = 0.013$).

A t-test between participants that reported their primary language to be English and the participants who reported another language shows a significant effect $t(274.58) = 3.98, p < 0.001$. Primary English speakers had a lower average time during the experiment (M = 1331.62, SD = 1054.31) than non-primary English speakers (M = 1711.75, 1054.89). The effect size of the t-test was $r = 0.223$, or $r^2 = 0.0545$. A

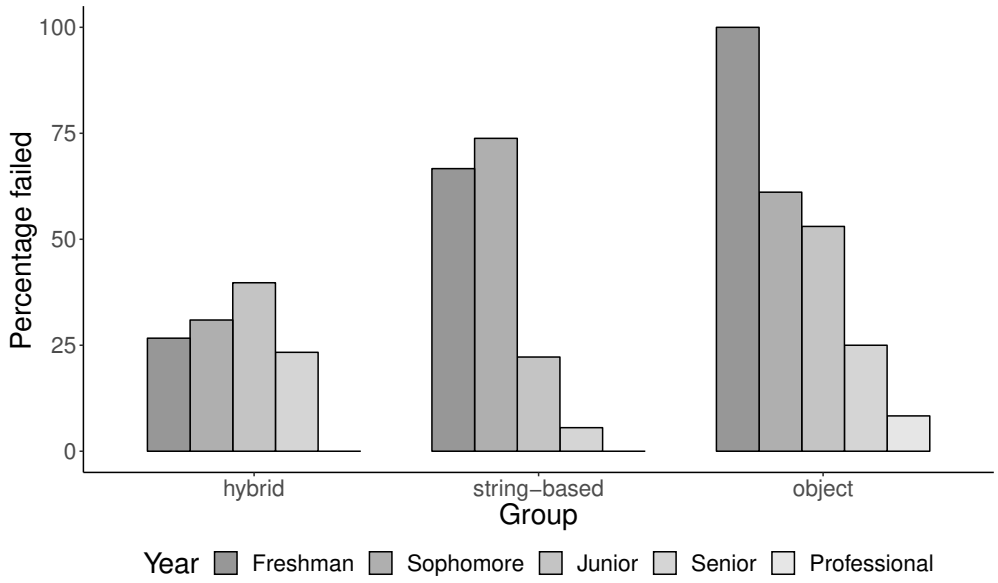

Figure 4.3: Barchart of results between the groups showing the percentage of failed tasks by level of education

Table 4.2: Bonferroni corrected t-test of average times by groups

	Hybrid	String-based
String-based	1.0000	-
Object	0.0037	0.01061

visual representation of the differences can be seen in figure 4.4. Testing only professional programmers for language differences showed a non-significant results $t(21.83) = 1.70,\ p = 0.104, r = 0.341$.

The differences between groups were tested using a t-test with Bonferroni correction. The t-test shows a significant difference between the average times of the hybrid and object oriented groups (p = 0.0037) and the average times of the polyglot and object-oriented groups (p = 0.0106).

To test the number of unfinished tasks in more detail a three-way log-linear analysis was conducted using the variables level of education, group, maxtime indicating whether a task time was maximum or not. A log-linear analysis is a test for more than two categorical variables in which at first, a saturated log-linear model is created which fits the data perfectly. Then the model is reduced through backward elimination, meaning that highest order interactions between variables are eliminated first. After each elimination of an interaction, the chi-squared statistic is recomputed to test whether the model still fits the data. Or in other words, both models are compared, and if there is a significant difference between the models, the model cannot be reduced in this way without resulting in a model that does not accurately fit the data [FMF12]. In this case likelihood ratio of the resulting model was $\chi^2(0) = 0, p = 1$, as the highest order interaction between level of education, group, and maxtime was significant, $\chi^2(12) = 51.08, p < 0.001$. This means the model could not be reduced from the saturated model. While reporting a model with 0 degrees of freedom

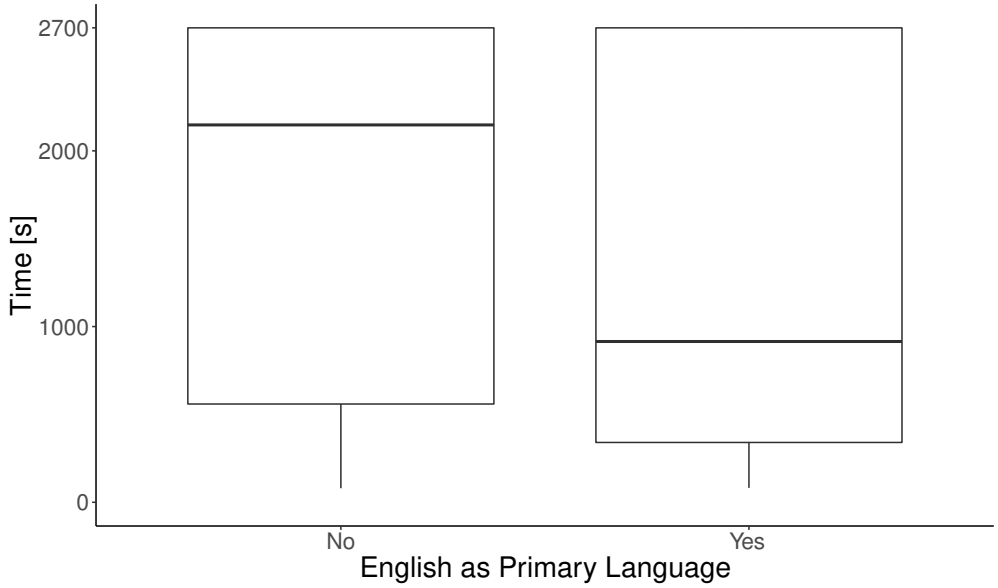

Figure 4.4: Boxplot of results between primary English speakers and non-primary English speakers

and p = 1 might look wrong, it is in the nature of a model fitting the data perfectly.

To further analyze this finding, a number of chi-square tests were performed on different combinations of two groups each and all levels of education. For freshmen, a significant association between the string based and the hybrid group was found regarding whether or not participants could finish the task in time $\chi^2(1) = 9.64, p = 0.002$, an effect that was also found for sophomores $\chi^2(1) = 15.46, p < 0.001$, juniors $\chi^2(1) = 5.34, p < 0.02$, and seniors $\chi^2(1) = 7.08, p < 0.008$. The odds ratios showed that freshmen and sophomores were more likely to fail to complete a task in the string based group compared to the hybrid group (5.327 times higher odds and 6.131 times higher odds respectively), while juniors and seniors were less likely to fail to complete a task in the string based group than in the hybrid group (odds ratios 0.436 and 0.196 respectively). Odds ratios below 1 indicate an inverse effect to odds ratios above 1, i.e. lower odds. Professionals overall only failed to complete two tasks in the object group, making this sort of comparative analysis infeasible.

Comparisons between the hybrid and object oriented group show significant effects for freshmen $\chi^2(1) = 18.48, p < 0.001$ and sophomores $\chi^2(1) = 8.61, p = 0.003$ with an infinitely high odd ratio for the freshmen due to the fact that no freshman finished object oriented tasks and an odds ratio of 3.46 for the sophomore comparison, indicating that they were more likely to fail object oriented tasks compared to hybrid tasks. Chi-square tests for juniors and seniors for the same comparison were not significant. Finally, looking at the comparison between the string based and object group, significant effects were found for freshmen $\chi^2(1) = 5.25, p = 0.022$, juniors $\chi^2(1) = 14.03, p < 0.001$, and seniors $\chi^2(1) = 8.08, p = 0.004$. The odds

ratios show that freshmen were once again infinitely more likely to fail a task in the object oriented group, while juniors had 3.91 higher odds to not complete a task in the object oriented group compared to the string-based group, and seniors' odd ratio was 5.59.

A chi-squared test between a binary measure of database experience and succeeding in a task was significant $\chi^2(1) = 43.18, p < 0.001$ with 18.21 higher odds to complete a task successfully when a participant had previous experience.

A log-linear analysis of the relationship between database experience, group, and notmaxtime (the inverse of maxtime for easier to understand results) resulted in a model with likelihood ratio $\chi^2(4) = 6.65, p = 0.16$. The interaction between group and notmaxtime was significant, $\chi^2(4) = 18.96, p < 0.001$, as was the interaction between database experience and notmaxtime, $\chi^2(3) = 63.15, p < 0.001$. This leaves us with a model with the three main effects database experience, group, and notmaxtime, as well as the two-way interaction effects between database experience and notmaxtime and group and notmaxtime.

Further analysis of the relationship between database experience and notmaxtime was conducted by running separate chi-squared tests of one group at a time in combination with the database experience measure. A significant effect was found in the object group $\chi^2(1) = 17.16, p < 0.001$, with 9.12 times higher odds that a participant might complete a task if they had previous database experience. In the polyglot group, there was also a significant effect with $\chi^2(1) = 13.55, p < 0.001$. In this case, no participants with previous database experience failed any task (out of 24 data points in that group), making the odds ratio infinite. Finally, in the hybrid group the effect is also significant with $\chi^2(1) = 14.07, p < 0.001$ and once again no participant with previous experience failed any of the tasks (n = 30).

4.4 Qualitative Results

In the current version of the experiment, the participants answered the question "Did you feel like you had to switch between languages during the experiment and how do you think did this affect your progress while solving the tasks?". This section will present some selected answers to this question.

4.4.1 Object-Oriented Group

The expectation for the responses in this group was that they would not notice a switch, as they exclusively wrote Java code with a few uses of strings to name fields of the table.

A common type of response to this question was that participants were not experiencing a switch, mostly with more experienced participants:

- *"no" - Professional*
- *"It felt to me as though I was using the same language throughout." - Professional*
- *"I do not think it affected my progress. I did not have to switch between languages." - Senior*
- *"I did not feel like that. I feel as though I was using similar enough concepts that there wasn't any real separation in my mind between how I would code previously." - Sophomore"*

A different type of answer to this question in this group was that participants considered the switch between their main language and the language used in the experiment on a larger code switching scale:

- *"No. Java is surprisingly similar to C++." - Sophomore*
- *"I am only familiar with the c++ language so I had a hard time understanding the java language." - Freshman*
- *"I never really felt this sense of "switching" between languages throughout the experiment. I am most familiar with C++ and I was familiar with object orientated programming so completing the the tasks was manageable. I feel that if someone has had experience with object oriented programming, then they shouldn't feel a huge shift of languages since each task involved calling methods to perform tasks. " - Junior*

And some participants in this group were confused by the question:

- *"I stuck with C++" - Junior*
- *"You could switch between languages!?!?!?!?!?" - Junior*
- *"No, I did not even know we had the option to switch between languages." - Junior*

From the responses, it seems that the participants do not report experiencing a change between languages within the experiment. Some of them report on the experience of having had to switch between languages to work on the experiment instead, as they are mostly familiar with another language, in those responses their main computer language (CL1) seems to be C++.

4.4.2 String-Based Group

This group was expected to experience a switch between languages, as they would have to switch between Java and SQL to complete the tasks.

Only 3 of the participants reported on experiencing the switch as intended:

- *"yes, java to SQL and back. Minimal." - Professional*
- *"I did feel like I had to switch between languages during the experiment. I used the "double-equals" (==) equality comparison operator from Java in an SQL query without thinking about it and that turned out to be the source of an erroneous result." - Professional*
- *"I felt like I had to think I was writing in C++ but picking from a database in SQL. It was easy since the SQL commands were just done inside query statements." - Senior*

Of the 36 participants in this group that left comments, 24 indicated that they did not switch languages. Some of their remarks focus on the overall switch between their CL1 and Java, instead of the switch within the tasks:

- *"I only used C++." - Junior*
- *"I did not feel like I had to switch between languages, so I think my progress while solving the tasks was helped. Once I understood what each line was doing, I could easily figure out what to do and which code samples would be beneficial." - Senior*
- *"No, I did not feel like I needed to switch languages during the experiment. I think this increased my velocity while solving the tasks." - Junior*
- *"I've only seen C++ before, so I didn't switch languages. If I did know more languages, I'm sure it would have helped." - Freshman*

Overall only 3 of the responses identified the switch between SQL and Java, and these participants reported that this did not impact their performance significantly. It seems all participants that identified the switch had a degree of familiarity with SQL.

Two thirds of the responses did not notice a change. It seemed that participants were mostly concerned with how their CL1 compared to the language used, or how they could use their previous knowledge of programming languages to help them solve the tasks. As the CL1 for most participants was C++ (since that is primarily taught at UNLV), with the occasional student who had experience with Java, there existed some prior knowledge amongst students regarding how the host language works as Java is syntactically similar to C++. However, the syntax and semantics of SQL are very different from Java and C++, as the language

follows the declarative computer language paradigm. Participants not noticing the difference is remarkable. It might suggest that participants take the languages they are not familiar with as one single language.

4.4.3 Hybrid Group

The hybrid group was also expected to notice a language switch, at least when it comes to writing conditions, as the syntax of the conditions in SQL and boolean statements in Java is different.

Some participants pointed out that they did notice a switch:

- *"I felt like it had a bit of SQL and Java." - Junior*
- *"Sure. I had to switch between an object oriented language and a set based query language. One was typed, the other "stringly" typed. It a switch, but after nearly a decade of experience and a dozen languages on under my belt, it's a minor switch. My day job has me routinely going from C# and UI work all the way down to C and embedded software in the same task though, so I may not be representative." - Professional*
- *"it did, since SQL code is not exaclty the same and the full code/ functions are not showing as well" - Sophomore*
- *"switch form different languages might need to change between different ways of thinking" - Graduate*

Most of the other participants only focused on the switch to Java as in the other two groups:

- *"No. I was familiar with Java before hand, and the API syntax was mostly intuitive to the point where for some of the problems I barely had to look at the sample code." - Senior*
- *"I did not feel like I had to switch between languages, and not having to switch allowed me to learn what to do throughout the tasks." - Junior*
- *"I thought it was all in the same language, so it was fine." - Junior*
- *"No, I do not feel like I had to switch between languages." - Freshman*

Similar to the string-based group, it seems as if most participants didn't consider this a significant switch between languages. Even the ones that did seem to notice a switch seemed unbothered by it. Most of the responses indicate that participants considered this the same language throughout.

4.5 Discussion

This section will first discuss the limitations of this study in subsection 4.5.1 and then interpret the results in subsection 4.5.2.

4.5.1 Limitations

While much care was taken in creating the study in a way to adhere to high standards of reporting [con], this experiment has limitations that might make the results less correct or generalizable.

First, the idea that the way programmers mentally manage programming languages similar to natural languages is not backed up by much evidence other than the findings that studies have found the use of similar brain regions being active during fMRI studies of code comprehension tasks [SKA+14, SPP+17]. This does not clearly imply a connection between the way computer and natural languages are comprehended or used, only that the same brain regions might be involved. As such, linguistic models might also not be applicable to the results of this study. The objective of finding if there are switching costs is in part hoping to find if such a connection between natural languages and computer languages exists. Even if initial evidence is found in this study, a statements about the relation between natural languages and computer languages still have to made cautiously and more research will be needed to confirm any findings.

The design of the tasks has been a central issue in the overall design of the experiment and was a focus of deliberation on trade-offs between ease-of-use, real world applicability, and representation of the problem of computer language switching. It is clear that the design of programming tasks in a programming experiment can have severe impacts on the results. Thus, despite frequent iteration of the task design, the design of the tasks might have unforeseen effects on the overall outcome of the experiment that might only become apparent in post-hoc analysis or might stay hidden entirely. Such problems could include parts of the tasks that appeared to be trivial to the experiment designer becoming major road blocks to the participants, such as unfamiliar syntax or semantics of the language used.

Further, the use of an automatic web platform to conduct the experiment allows for easier recruiting of larger samples of participants and less potential introduction of bias by experimenters, but it also requires the experimenters to rely on the participants' compliance with the rules without much oversight. As the experiment is not necessarily conducted in a lab setting, subjects are asked to create their own lab environment, but they might not be able to create such an environment due to external factors, or they might be unwilling to do so, be it because they don't recognize distractions as impact-full or because they do not care about the results of the experiment. Participants might be able to cheat the results by copying from other participants or by utilizing other resources to solve problems they are presented with during the tasks. Mitigation of some of these issues is possible by reviewing the progress of participants through the experiment by inspecting the saved code snapshots. This way, suspicious participants can be removed from the sample.

The experiment mostly includes students, which means that results of the experiment cannot be easily generalized to professional populations as significant differences between professional developers and computer science students have been found [USH+16].

When considering the results of the experiment, it is important to point out that most of the participants with database experience seemed to be professional developers that generally performed better than any other population in this experiment. The rest of the participants with database experience were seniors in a computer science program (except for one junior) and therefore also represented a more experienced population, which makes it necessary to point out that database experience is at least partly also a proxy for general programming experience and thus, conclusions made from the data concerning database experience should be considered very carefully. While it is likely that database experience would be more general in more experienced programmers in general due to the need to save and retrieve data is common in programming tasks and relational databases are a popular choice for this task, it is also important to point out that the rarity of database knowledge amongst the student participants stems from the choice to recruit at UNLV, which does not require students to take database courses to complete their computer science education. It is likely that choosing another university in which database classes are required would lead to a sample that might allow for more meaningful analysis of the database experience variable.

The analysis of primary natural language influence on development productivity has to be considered very carefully, as the measures used for this analysis might not be perfect representations of the language status of participants. The question asked to determine the participants' primary language is "What is your primary language (i.e., the language you speak at home)?". This question is not sufficient to determine when a participant learned their primary language, how much they use it, and when and how they learned English and how much they speak English as well as the English proficiency of participants. The measure for English proficiency used in the study is self-reported and may not be an accurate representation of their actual proficiency. Furthermore, even if primary language and proficiency are accurate, it is possible that these measures only serve as proxies for other factors such as prior education or socioeconomic differences et cetera. Thus, interpretation of the results regarding the relationship between natural language and programming productivity should be done with caution until a more comprehensive test of this relationship is conducted using appropriate linguistic measures for proficiency. *It has to be emphasized that these findings are not sufficient evidence to base hiring or academic decisions on.* The author hopes that these findings might lead to further research into the issue, which, in case that the difference is confirmed, can hopefully lead to solutions to bridge the differences for minorities in computer science in the United States as well as making learning programming easier for people across the globe who are not native English speakers.

4.5.2 Interpretation

Analysis of the results shows several interesting patterns through analysis of the amount of time spent to solve tasks in combination with analysis of the number of failed tasks.

Relationship between Language Switching and Productivity

The results of the ANOVA show that there was a significant difference between the productivity of the different groups at $p = 0.029$, which reflect different amount of code switching. Therefore we can reject $H_0 1$ that there is no relationship between switching and productivity. When looking at the differences between groups in figure 4.1 and at the post-hoc t-test, we can see that there are no significant differences between the hybrid and the string-based groups while there is a significant difference between each of those two groups and the object-oriented group. Both the string-based group and the hybrid group have lower average task times than the object-oriented group, indicating that switching to a second language while solving programming tasks might have a positive impact on programming productivity. The common assumption being that choosing the right language for a specific use-case might help express the instructions to the computer more appropriately, which in turn makes the task easier [pol, Fje08]. It is important to note that the overall effect size is relatively small with 3.9% of the variance accounted for.

While the object-oriented group appears to be the slowest group in the experiment, the other two groups are closer together in solution time. Figure 4.2 allows for a more detailed look at the differences between the two groups in terms of level of education as a representation of participant experience. The graph shows that participants in the string-based group that were freshmen or sophomores took longer on average to solve the tasks than participants which were juniors, seniors, or professionals. The medians for freshmen and sophomores being at the maximum time for tasks indicates that the subjects of those levels of education might have struggled to complete the tasks in time. A look at figure 4.3 shows a clearer picture of proportions of missed tasks per level of education and reveals that both freshmen and sophomores clearly failed more tasks in the string-based group than their more experienced counterparts. On the other hand, looking at the hybrid group's percentage of missed tasks reveals that participants in that group did not miss as many tasks. A fact that is also reflected in the lower average times for those two levels of education in the hybrid group compared to the string-based group (see figure 4.2), which results in this group having a slightly lower mean solution time (M = 1269.74s, SD = 1043.20), than the string-based group (M = 1357.54s, SD = 1057.53) even though more experienced participants (junior, senior, and professional) completed tasks faster in the string-based group than in the hybrid group.

This observation is confirmed in the loglinear analysis of task failures, which shows that there is an interaction between level of education, group, and the occurrence of maximum time task events. The chi-squared tests to further investigate this interaction show significant effects for all college levels between the string-based and the hybrid group with odds ratios indicating that freshmen were 5 times more likely to fail a task in the string-based groups and sophomores were 6 times more likely, while juniors and seniors were less likely to fail a task in the string-based group than in the hybrid group (odds ratios 0.4 and 0.2 respectively). This indicates that the code in the hybrid group was easier to write for less experienced participants while more experienced participants were faster in solving the tasks in the string-based group.

Two main explanations appear viable from these experimental results. For one, code switching in the string-based group was designed to be a more complete, less frequent, switch between languages within the context of the experiment. Meaning that participants switched from Java to SQL to write a complete SQL string and then switched back to Java. Code switching in the hybrid group, on the other hand, was a switch to a smaller section of SQL-like code that didn't fully require the understanding of a new language. This quicker switch might be easier to handle for inexperienced programmers, while a more experienced programmer might experience a more intense interruption in their programming concentration on one language. Inexperienced programmers might be more flexible to changes between languages because they think less about the constraints of the language they are currently using and might be thinking about both languages being part of a whole than more experienced developers as indicated by some of the comments made in the exit survey, such as *"No, I do not feel like I had to switch between languages."* in the hybrid group from a freshman, while a junior in the same group recognized the change: *"I felt like it had a bit of SQL and Java."*

The other explanation could be that more experienced programmers might be more familiar with SQL than the less experienced freshmen and sophomores and therefore feel more familiar with the straight integration of SQL into a Java instruction. Since freshmen and sophomores likely had no exposure to this practice and SQL before, confirmed by the fact that all participants that indicated experience with databases were juniors, seniors, and professionals, they had a harder time understanding the syntax and could not adapt as quickly as more experienced developers. A caveat to this explanation is the fact that of the 109 participants taken into account for the data analysis only 14 indicated they had any experience related to databases and most of those were professional developers, making the analysis of the relationship between database experience and success in solving tasks difficult, as it became more of a proxy for very experienced developers.

Overall, the results regarding the differences between the groups seem clear: The object-oriented group was the slowest group and had the most failed tasks across the board. Both the hybrid and the string-based groups are on par regarding their average time and a closer look shows that the hybrid group was easier than the string-based group for less experienced developers while the string-based group was easier for more

experienced developers. The evidence suggests that inexperienced developers either do better at switching rapidly because they have a less rigid understanding of the language they are using or that more experienced developers had more familiarity with SQL and string-based database programming and therefore solved those tasks more easily while inexperienced programmers were lost in the string-based group because they had difficulties picking up the rules of how to use SQL.

Participant Experience of Computer Language Switching

From the answers in the exit survey, the majority of the participants in the groups switching between languages did not notice that they were switching. Only 3 participants from the string-based group directly acknowledged the switching, while 24 out of the 36 participants (66.66%) denied that any switching was taking place. The other 9 participants gave unrelated answers. On top of that, most the participants of the hybrid group also did not remark that they noticed switching. In both of these groups participants that noticed switching also tended to minimize the effect they were thinking it was having on their performance: *"yes, java to SQL and back. Minimal."* However, some also noticed issues with switching: *"I did feel like I had to switch between languages during the experiment. I used the "double-equals" (==) equality comparison operator from Java in an SQL query without thinking about it and that turned out to be the source of an erroneous result."* This in itself is a interesting observation regarding confusion of syntax that might become more prominent in a file level switch.

Overall, $H_0 2$, the hypothesis that programmers do not consciously notice that they switch between computer languages, cannot be outright rejected with current evidence, even though a minority does recognize a switch. It appears that most participants do not notice the switch or don't feel like their switching actually affects their productivity. Especially when keeping in mind that the group switching the least, the object-oriented group, took the longest to complete the tasks, becomes reasonable to assume that the effect of computer language switching might be limited on the embedded language switching level.

Productivity Difference based on Native Language

The results of the study show that, overall, participants which stated their primary language is different from English solved tasks significantly slower than participants that stated that their primary language is English. However, testing experienced programmers, there was no significant result. While it could be hypothesized that bilingual programmers might be able to perform better than monolingual ones based on the observations that bilingual people have better language skills than monolingual people overall [YTF17], that would require that programming actually benefits from natural language skills, which is an unproven assumption seeming less likely after seeing the results of this experiment. Though it is possible that the participants in

this study weren't early childhood bilinguals and thus did not benefit from these improved language skills. The survey did not track the participants' exact language history and only asked participants to fill in their primary language and, if this language wasn't English, rate their English proficiency. An attempt to use the proficiency scores given by the participants to analyze the differences in performance failed because the scores were inconsistent with their performance. In fact, some primary English speakers rated their own proficiency as 8 (out of 10), which is unlikely to be accurate. A better assessment of language skill and language history of participants is necessary to make real conclusions from these data. In the future, a study investigating the relationship between English skill and programming productivity would need to ask participants more detailed questions and use a verified instrument to assess English language skill such as the TOEFL test [4].

An explanation for the differences between primary and non-primary English speakers that can be made from the data is that it is most likely that those that were not as comfortable with speaking English might have had more difficulty understanding the instructions than the primary English speakers or that non-primary English speakers had more trouble with the use of English keywords in languages and APIs they were unfamiliar with. Since professionals did not show a significant difference, it is likely that with increasing programming experience, differences between programmer productivity based on natural language disappear. In summary, overall there was reduced productivity for participants which were not primary English speakers, however this effect seems to disappear for more experienced programmers.

4.6 Conclusion

This chapter described an experiment on the impact of computer language switching on software development productivity motivated by findings in linguistic research suggesting that there is a time cost to switching between natural languages and the ubiquity of computer language switching in polyglot programming. To create tasks in which participants had to switch between languages at different rates, three groups were developed, ranging from no switches over some switches to a lot of switches. The groups were designed around three different ways to structure the API of a querying mechanism in a general purpose programming language. The design of the experiment has been improved iteratively, to ensure the tasks were as well designed as possible.

The chapter reports the design of the experiment in detail and shows the analysis of three datasets: two pilot studies and the final run of the experiment. Results show that the group with no switches took longer to solve tasks and had more participants not completing tasks than the two groups with switches. Both groups with switches were about even in performance with the hybrid group with a lot of switches being easier for less experienced programmers, while the string-based, more traditional, group with some switches

[4] https://www.ets.org/toefl

was easier for programmers with more experience (juniors, seniors, and professionals) to solve, while it was very difficult for participants with less experience to complete tasks. Further, the experiment revealed that the majority of participants do not recognize switches between languages in an embedded language switching context and that even the participants that recognize the switch minimize the the possible effects on their productivity.

Finally, the data show that the primary English speakers in this experiment performed better than the participants that noted that their primary natural language is different than English. It is likely that non-primary English speaker might either struggle understanding the task instructions or code samples as easily as primary English speakers or that English keywords in programming languages and English names of API functions might not be as easy to the understand.

Future research will have to replicate the differences between primary and non-primary English speakers when it comes to programming productivity and find more in-depth explanations for the differences than was possible with the current dataset. Regarding computer language switching, future studies will have to investigate if effects found in this study are reproducible and whether the same effects can be found in different computer language switching contexts such as file-level switching.

Chapter 5

Randomized Controlled Trial on the Impact of File Level Computer Language Switching

In this chapter, a study on file level computer language switching will be presented. This study is designed to test whether there is a measurable difference between groups working on a task in two separate files with both files either being written in the same language or in two different languages.

5.1 Introduction

There are considered to be three levels to computer language switching (see section 1.1): Project level switching when developers switch between projects and therefore between different sets of languages, file level switching when developers switch between files that contain different languages, and finally embedded level switching that happens when developers switch between languages withing the same file or other section of a program. Project level switching occurs when programmers switch between two projects that contain different sets of languages and is possibly similar to natural language switching [AG08]. File level switching is the process of switching between files that contain different programming languages. When working on one issue in one file using one language and then switching to another file written in another language to work on another part of the project, the process can be likened to inter-sentential code switching in natural languages, the process of code switching between utterances [HA01, Li96, YTF17]. A more rapid approach to switching on the file level and switching between languages within a file on the embedded level can be described as intra-sentential code switching [HA01, Li96, YTF17]. While the Randomized Controlled Trial on the Impact of Embedded Computer Language Switching (see chapter 4) investigates possible switching

effects on the embedded language level, this study aims to investigate effects on the file switching level.

Switching between languages on a file level is likely the most common occurrence of file switching in a project considering that the average number of languages being five was determined on a file-by-file basis, disregarding possible embedded languages in files [MB15, TT14]. A common type of project utilizing numerous languages are web projects as they often employ a number of languages for different parts of the functionality of a web page, such as using JavaScript as a language to define behavior on a web page defined in HTML and CSS, while receiving and sending data to the server written in PHP. And thus this study was designed to test whether we can measure any effect computer language switching might have by having participants solve two tasks that are similar to real world web development tasks.

Due to its ubiquity in current web development, JavaScript was a good choice for a programming language that would have to be included in an treatment for this experiment. The choice was made especially simple by the fact that it isn't uncommon to see it used both on the client and server side thanks to libraries such as Node.js [nod]. As the experimental treatment needs a second language to switch to, PHP was chosen as a typical server-side programming language that is used in 79% of server-side programs [php, HKV13]. Since PHP is not a front-end language, the tasks were constructed to make sense in a web programming environment without using strong front-end metaphors referring to website specifics and are more concerned about communication tasks between two systems that might be involved in a web programming environment.

Taking these considerations into account, a functional online double-blind randomized controlled trial was designed to the impact of switching between two programming languages while solving programming tasks. The main treatment in group one is the switch between JavaScript and PHP, while two control groups cover solving the same tasks in either just JavaScript or just PHP.

5.1.1 Objective

The experiment described herein is designed to measure whether switching between two programming languages on a file level might impact software development time. For this we utilize a group forced to switch between JavaScript and PHP to complete two tasks, while two other groups have to switch between different files of the same languages, JavaScript and PHP respectively.

The null-hypotheses are as follows:

$H_0$1: *Polyglot file switching has no impact on* ***productivity*** *compared to monoglot file switching.*
$H_0$2: *Polyglot file switching has no impact on* ***error rate*** *compared to monoglot file switching.*

$H_0$3: *There is no relationship between the amount of switches between files and productivity.*

$H_0$4: *Programmers do not consider switching between programming languages to be a factor in their productivity.*

$H_0$5: *There is no difference between the productivity of native English speakers and programmers with a different native language.*

The alternative hypotheses can be derived from the null-hypotheses.

Productivity is defined as the time to a correct solution given specific parameters, determined by whether a number of automatic test cases pass on the solution provided by the participant. Error rate is measured by the amount of unsuccessful submissions to the automatic testing system.

The first two hypotheses aim to answer the core question of this research, whether switching between languages might impact development productivity. While rejecting $H_0$1 might find that there is an overall effect, data that leads us to reject $H_0$2 might allow us to gain more insight into why there might be a difference between the group switching between languages and the groups that are not switching. If, for example, switching between languages results in more syntax errors made by participants, it would be reasonable to theorize that participants have difficulties switching between the syntaxes of the languages which results in more code & fix behavior, increasing the time needed to complete the task.

Hypothesis three is poised to answer whether the switching behavior of participants is a contributing factor in productivity differences. The experiment is designed to only allow a participant to look at one of the two files in the task at a time and requires the participant to switch between the files by clicking the tab of the file they are currently looking at. This allows to directly measure the amount of switches between the files they have made. It seems likely that a participant switching less would experience fewer difficulties from switching if there are any to report to begin with.

Hypothesis four is more qualitative in nature and is concerned with the perception participants have of their switching between languages and whether it impacted their work. To gain insight regarding this hypothesis, a question was included in the debriefing screen after the experiment ended.

Lastly, hypothesis number five aims to investigate whether participants that consider themselves primary English speakers perform better than participants that do not consider themselves primary English speakers. The primary language is recorded in a survey preceding the tasks, which asks which language the participants

consider their primary.

5.1.2 Design Process

This experiment was designed in a rapid iteration process. After the first version of tasks was complete, it was tested by a first experienced pilot participant. After this first test, the participant's recommendations were implemented where deemed sensible, after which two more participants completed the new version of the experiment. The second and third participants then provided more feedback which was incorporated in the design of the experiment to make sure the instructions are clear and the tasks are solvable in a reasonable time frame. A last pilot participant confirmed that the changes from the previous round of testing were successfully implemented. To make sure that the operation of the experiment would go smoothly, comments of early participants were screened for indications of possible bugs.

5.1.3 Structure

As with the previous chapter containing the embedded switching experiment (chapter 4), this chapter's structure is loosely inspired by the CONSORT standard [con], which gives guidelines on how to report on the proceedings of experiments in a well-structured, repeatable manner. The rest of the chapter will be structured as follows: In section 5.2, the methods used in the experiment will be described, including subsection 5.2.1 regarding the overall design of the experiment, subsection 5.2.2, which contains information on the participants, subsection 5.2.3, which contains information on the setting of the experiment, subsection 5.2.4 covering the intervention, subsection 5.2.5 covering the outcomes of the experiment, as well as the sample size in subsection 5.2.6, the randomization used in subsection 5.2.7, and the blinding in subsection 5.2.8. The results of the experiment are laid out in subsection 5.3 and discussed in section 5.5. The chapter will be concluded in section 5.6.

5.2 Methods

In this section, the high level design of the experiment will be described in subsection 5.2.1, while subsection 5.2.2 will describe how participants were recruited. Subsection 5.2.3 describes in what setting the experiment was conducted, while subsection 5.2.4 describes the groups and tasks that were used as the intervention in this experiment. Subsection 5.2.5 specifies which outcome measures were collected during the experiment and subsection 5.2.6 describes considerations regarding the sample size of the trial. Finally, subsection 5.2.7 describes the algorithm used to randomize group assignment and subsection 5.2.8 explains

Table 5.1: File Switching Experiment Design

		Group	(Warm-up) Task 1	(Warm-up) Task 2	Task 3	Task 4	
Survey	R	Polyglot	JavaScript	PHP	PHP/JavaScript	JavaScript/PHP	Debriefing
Survey	R	JavaScript	JavaScript	JavaScript	JavaScript	JavaScript	Debriefing
Survey	R	PHP	PHP	PHP	PHP	PHP	Debriefing

how the experimental design ensured blinding of both researchers and participants.

5.2.1 Trial Design

This is a double-blind randomized controlled trial with a repeated measures design. Participants were randomly assigned to one of three groups, each representing a set of languages to be used in the experiment. The sets being JavaScript, PHP, and JavaScript + PHP. The randomization was based on participants' year in college or status as a professional software developer to maintain equal assignments on different levels of software development experience. During the experiment, participants are asked to fill out a survey, complete four programming tasks, and fill out an exit survey about their experience during the experiment. The first two of the programming tasks were designed to be warm-up tasks for the participants, while the last two tasks represent the actual treatment.

An overview of the design of the experiment can be seen in table 5.1. Participants start with a survey, are randomly assigned to one of the three groups, then they complete the tasks in order until they have finished all of them and are then debriefed.

5.2.2 Participants

Participants were eligible to participate in this experiment if they were 18 years of age or older and had at least some programming experience. Programming experience evaluation was largely based on their year in a computer science program in college or their status as a professional software developer as, both of which was self-reported in the survey included in the experiment alongside other possible measures for programming experience. Participants' year in college is a common measure for programming experience and has been shown to impact times to complete solutions in programming experiments before [SKL+14b, USH+16]. Participants were recruited at the University of Nevada, Las Vegas in computer science classes and on Twitter. Students were recruited by a short speech and a pamphlet in their classes while professionals were solicited by a post on Twitter. Participation was entirely voluntary and student participants were given up to 3% of extra credit in the classes they were recruited from. An alternative for participation was available to receive the same credit.

5.2.3 Study Setting

The study was conducted using our Experiment Platform for the Internet (EPI). The platform walks participants through the consent process and if participants gave consent to participation, they were lead to a survey asking them to give information about their development experience, education, and optionally their specific class and name to be able to give them extra credit. After the completion of the survey, participants were given the experimental protocol and then directed to the first task.

The first and second task were designed to make sure participants had time to familiarize themselves with the environment and the languages. These tasks were kept simple and provided participants with information on how to use the specific language the task was written in. Participants needed to complete the questions successfully to reach the main tasks and they had as much time as they needed to solve those two warm-up tasks, enforcing a baseline of knowledge. The other two tasks were designed to be the main treatment and had a time limit of one hour with the ability to give up on a task by pressing a button that becomes visible after 45 minutes.

When starting a task, in contrast to the way it was handled in the embedded language switch experiment described in chapter 4, participants were directly given source code files to see without a sample phase since this isolates the switching effect between the source code files of concern without having participants switch between the code file and a sample as well. The source code files then contained scaffolding for the tasks and instructions on how to proceed, as well as some information on the syntax of the language and available functions. Participants could then run their code against a number of test cases by clicking the "Check Code" button. The code written by participants would then be sent to the server where it was run and any output of the interpreter and the test code would be displayed back to the participants in an output box. If the tests passed, participants were allowed to move on to the next task, otherwise they could use the information in the output box to solve any outstanding issues. After completing all tasks, the participants were lead to a quick survey about their experience during the experiment and, after completing this exit survey, thanked for participating.

5.2.4 Intervention

The main treatment in this experiment is the switch between two languages and they were chosen to be JavaScript and PHP since these are common choices for interactive websites. To be able to make sure that the effect isn't due to a specific programming language involved, two groups had to be added so that there was a pure JavaScript and a pure PHP group. Overall that means that this experiment needed three different groups: Polyglot, JavaScript, and PHP.

```
1
2  Requester = (function (helper) {
3    var my = {};
4
5  //
6  // This is a warmup task to help you familiarize yourself with the language.
7  // Please take your time to read the existing code and familiarize yourself with it.
8  // Once you feel comfortable, please try to solve the task. You can compile your
9  // code and have it automatically checked by clicking the compile button at the
10 // bottom. You have as much time and as many tries as you want.
11 //
12 // * Extend the createRequest function to check if the requestId is 3, and if so,
13 // create a delete request and send the request
14 //
15 // ## Available imported functions:
16 //
17 // helper.createGetRequest(requestId)
18 // Returns a GET request object
19 //
20 // helper.createPostRequest(requestId)
21 // Returns a POST request object
22 //
23 // helper.createDeleteRequest(requestId)
24 // Returns a DELETE request object
25 //
26 // helper.sendRequest(request)
27 // Sends out the request
28 //
29
30     my.createRequest = function(requestId) {
31
32       if (requestId === 1) {
33         var getRequest = helper.createGetRequest(requestId);
34         helper.sendRequest(getRequest);
35       } else if (requestId === 2) {
36         var postRequest = helper.createPostRequest(requestId);
37         helper.sendRequest(postRequest);
38       }
39       // Your code here
40       // $$$$$$$$$$$$$$$$$$$$
41       else if (requestId === 3) {
42         var deleteRequest = helper.createDeleteRequest(requestId);
43         helper.sendRequest(deleteRequest);
44       }
45
46       // $$$$$$$$$$$$$$$$$$$$
47     }
48
49     return my;
50 }(helper));
```

Code Sample 22: JavaScript version of the first warm-up task with solution.

The first warm-up task asked participants to write a small part of an almost finished function. The only part that needed to be filled out was the section between the commented lines with the dollar signs as can be seen in code sample 22. Participants were able to use the existing code to infer how to solve the task, especially pertaining to specifics of how to use the syntax. The solution required them to write an else-if statement checking whether the *requestId* was equal to 3, and if so, create a delete request using a preprepared method and send it using the preprepared *sendRequest* method. A plain if-statement instead of the else-if used here would work as well. The code between the comments was the correct solution and was not present in the actual file given to participants. There existed an equivalent PHP version of the same task.

The second warm-up tasks was similar to the first and required the participants to write another if-statement using chained together logical statements to test whether a parameter of the function was not valid and send an appropriate error message if that was the case. An example of the PHP version of the

```php
<?php
require_once('helper.php');

//
// This is a warmup task to help you familiarize yourself with the language.
// Please take your time to read the existing code and familiarize yourself with it.
// Once you feel comfortable, please try to solve the task. You can compile your code and have it
// automatically checked by clicking the compile button at the bottom. You have as much time and
// as many tries as you need.
//
// * Extend the checkErrors method to check if the id parameter is empty or not
// numeric. Send the appropriate error message if either case is true.
//
// ## Available imported functions:
// sendErrorMessageYear()
// Sends the appropriate error message in case the year parameter is wrong.
//
// sendErrorMessageCost()
// Sends the appropriate error message in case the cost parameter is wrong.
//
// sendErrorMessageId()
// Sends the appropriate error message in case the id parameter is wrong.
//
// empty(variable)
// Checks whether a variable is empty.
//
// is_numeric(variable)
// Function that checks whether a variable is numeric or not.
//

function checkErrors($year, $cost, $id) {
  if (empty($year) || !is_numeric($year) || $year < 0 || $year > 2050 ) {
    sendErrorMessageYear();
  }
  if (empty($cost) || !is_numeric($cost) || $cost < 0) {
    sendErrorMessageCost();
  }
  // Your code here
  // $$$$$$$$$$$$$$$$$$$
  if (empty($id) || !is_numeric($id)) {
    sendErrorMessageId();
  }
  // $$$$$$$$$$$$$$$$$$$
}


?>
```

Code Sample 23: PHP version of the second warm-up task with solution.

task can be seen in code sample 23. Once again, this sample already includes the solution to the problem for easier viewing of the sample and code between the dollar sign comments is solution code that wasn't available to the participants. Here participants needed to create a plain if-statement checking whether the variable *id* was empty or if it was not numeric and send an error message using a preprepared function if the condition was true. While the code sample shows the PHP version of the task, an equivalent JavaScript version existed as well for the pure JavaScript group.

The aim of both of these tasks is to make sure participants use variables, if-statements, function calls, and variable definitions in the languages needed for the tasks as well as use the "Check Code" button before starting the "real" tasks and get used to the style of the instructions given to them.

Task 3 was the first task in which participants had to switch between two files to complete the program. You can see the first file displayed to participants in code sample 24 in its PHP version and the second file

```php
<?php
require_once('database.php');
require_once('network.php');

// In this task you will have to program in both this and the other file. Use the
// tab at the top to switch between both files.
//
// The messenger sends a message to the presenter, which will then
// react to the message after which the messenger's receiveResponse function is called.
//
// ## Available imported functions:
//
// isValid($data)
// Returns true if data is valid.
//
// storeResponse($data, $source)
// Stores the received response and the source it came from
//
// encodeMessage($message)
// Returns a message that is encoded to the right format
//
// sendMessage($destination, $encodedmessage)
// Sends an encoded message to the destination
//

// * The function sendData should send data to the presenter after encoding the
// message.
function sendData($message, $destination) {

  // Your code here
  // $$$$$$$$$$$$$$$$$$$$
  $encodedMessage = encodeMessage($message);
  sendMessage($destination, $encodedMessage);
  // $$$$$$$$$$$$$$$$$$$$
}

// * The function receiveResponse should check whether the data is valid and store
// the data in case it is valid. Otherwise nothing should happen.
function receiveResponse($data, $source) {

  // Your code here
  // $$$$$$$$$$$$$$$$$$$$
  if (isValid($data)) {
    storeResponse($data, $source);
  }
  // $$$$$$$$$$$$$$$$$$$$
}


?>
```

Code Sample 24: PHP version of third task messenger file with solution.

that participants had to switch to in code sample 25. The code in this task sent a message from the messenger to the presenter, which had a method that reacted to incoming messages, which was then supposed to respond back to the messenger. The messenger then received the response in the *receiveResponse* method. Participants had to write the code to send the initial message, receive the message in the presenter, and to receive the presenter's response, which should encourage some switching between files.

The direct communication between the two files was handled using the libraries that the participants called to solve their tasks. In the case of the PHP to PHP and the JavaScript to JavaScript communication that was needed for the JavaScript and PHP group respectively, the communication between both parts of the system was easily achieved through importing both files in a test case and running their code, while the communication between PHP and JavaScript in task 3 was handled by the PHP program calling a bash script behind the scenes which ran the JavaScript program with data from the PHP program, the PHP

```
1  Presenter = (function (view, controller) {
2    var my = {};
3
4  // ## Available imported functions:
5  //
6  // controller.validateData(data)
7  // Returns true if data is valid
8  //
9  // view.update(data)
10 // Updates the view with new data
11 //
12 // view.getState()
13 // Returns the state of the view
14 //
15 // controller.respond(data, source)
16 // Sends a response message with data
17 //
18
19 // The onIncomingData function should update the view with new data if the data is
20 // valid, then, regardless of the validity of the data, the function should respond
21 // to the controller with the state of the view.
22   my.onIncomingData = function(data, source) {
23
24     // Your code here
25     // $$$$$$$$$$$$$$$$$$$$
26     if (controller.validateData(data)) {
27         view.update(data);
28     }
29     var state = view.getState();
30     controller.respond(state, source);
31     // $$$$$$$$$$$$$$$$$$$$
32   }
33
34   return my;
35 }(view, controller));
```

Code Sample 25: JavaScript version third task presenter file with solution.

program then ran its course until it called the PHP program again similarly by invoking a bash script so that the *receiveResponse* method could be called in response to the message. Either program threw errors if anything went wrong and recorded progress of the program in a hidden file that was later read by the test to make sure all tasks were completed satisfactorily. In task 4, the communication was from JavaScript to PHP and leveraged the fact that the experimentation software ran on a PHP server. The PHP file written by the participant is saved to the server and called from the JavaScript in a regular AJAX request, which also allowed for the JavaScript to handle the response.

In task 4, participants were told that the client receives an incoming message in its $onIncomingMessage$ file, which had to be handled. The message needed to be parsed and packaged and then sent off to the server for processing. The server then extracted the information it needs from the data package, checked if it is corrupt or not. If it was not corrupt, it saved the data to the database and sent back a positive results, and if the data was corrupt, the server did not save the data and sent a negative result back to the client. To check if data is corrupt the participants had to complete the $dataIsNotCorrupt$ function in the same file. The response of the server was handled in a callback that was implemented in the client file called $responseFunction$. There, the response message had to be parsed and checked for success. If the response indicated a success, the function needed to call the *respondSuccess* function to send a success message to the initial source of the message, and in the case of an error, the *respondError* function had to be called to

```
1  Messenger = function (helper, source, callback) {
2    var my = {};
3  // In this task you will have to program in both this and the other file. Use the
4  // tab at the top to switch between both files. While this file determines what
5  // code is being run on the client, the other determines the code on the server.
6  //
7  // When the client receives an incoming message, the onIncomingMessage function is
8  // is called, which then sends data to the server. The server then processes
9  // the message that was sent from the client to the server and sends back a
10 // response in the handleResponse method. The message back from the server is then
11 // handled by the response function.
12 // [..]
13 // Do not change this function////
14    var sendPackage = function (packagedData) {
15      helper.sendDataPackage(packagedData, responseFunction);
16    }
17 ////////////////////////////////
18 // * The responseFunction(response) should parse the message from the response,
19 // determine if the parsed message indicates a success and if true, send a success
20 // response to the source. If not, send an error response to the source.
21    var responseFunction = function (response) {
22        // Your code here
23        var parsedMessage = helper.parseMessage(response);
24        if (helper.isSuccess(parsedMessage)) {
25          helper.respondSuccess(source, parsedMessage);
26        } else {
27          helper.respondError(source, parsedMessage);
28        }
29
30        // $$$$$$$$$$$$$$$$$$$$
31         callback(source);
32    }
33 // * The onIncomingMessage(message) function should parse the incoming message,
34 // create a data package from parsed message, and send the data package.
35    my.onIncomingMessage = function(message) {
36     // Your code here
37     var parsedMessage = helper.parseMessage(message);
38     var packagedMessage = helper.makePackage(parsedMessage);
39     sendPackage(packagedMessage);
40     // $$$$$$$$$$$$$$$$$$$$
41   }
42   return my;
43 }
```

Code Sample 26: JavaScript version of the fourth task client file with solution.

indicate an error instead. The JavaScript version of the client file can be seen in code sample 26, while the PHP version of the server file can be seen in code sample 27.

5.2.5 Outcomes

The main outcome measure of this experiment is the time to a complete and a correct solution. It was measured automatically by the online experimentation platform by creating a time stamp when a task is first shown to a participant and the time stamp when the task was either completed successfully as measured by test cases, or when the maximum time ran out for the participant for this specific task. Additionally, the platform recorded each attempt at running the code the participants made and their outcome, which was used to calculate a number of failed attempts to run the code and a list of common errors made by the participants.

While some participants might have preferred to go step by step in running their programs after changing parts, which might have lead to inflated numbers of errors thrown by the testing system, the most interesting

```php
<?php
//[..]
require_once('functions.php');
// * To make sure you can easily check if the data is corrupt, the dataIsNotCorrupt
// function should return false if the data parameter is empty or not a number.
// Otherwise we can assume the data is not corrupt and return true.
function dataIsNotCorrupt($data) {
  // Your code here
  // $$$$$$$$$$$$$$$$$$$$
  if (empyt($data) || !is_numeric($data)) {
    return False;
  }
  return True;
  // $$$$$$$$$$$$$$$$$$$$
}
// * The handleResponse function needs to check if the favorite number contained in
// the incoming message is corrupt. If the number is not corrupt, save it into the
// database and return the success message. Otherwise return a failure message.
// Keep in mind that the $result variable has to be changed to return a failure
// message.
function handleResponse($incomingMessage) {
  $result = successMessage();
  // Your code here
  // $$$$$$$$$$$$$$$$$$$$
  $number = getFavoriteNumber($incomingMessage);
  if (dataIsNotCorrupt($number)) {
    saveDataIntoDB($number);
  } else {
    $result = failMessage();
  }
  // $$$$$$$$$$$$$$$$$$$$
  echo $result;
}
handleResponse($_POST);
?>
```

Code Sample 27: PHP version of the fourth task server file with solution.

part of unsuccessful run attempts by participants in this experiment were the attempts in which errors are present that might be caused by confusion between languages. A prime example for this might be syntax errors caused by participants not completely switching languages successfully and mistakenly writing JavaScript code in the PHP file or vice versa.

Another important measure in this experiment is the amount of switches made between the two files present in task 3 and 4. This is measured by how many time participants clicked the tabs to switch between files, which was recorded to the experiment database. This can help to identify whether more frequent switching caused more errors or increases in time to correct solution.

Finally, a qualitative measure of the participants' experience switching between languages was asked in the exit questionnaire to assess whether they felt like switching impacted their progress. The exact question was:

- "Did you feel like you had to switch between languages often and how do you think did this affect your progress while solving the tasks?"

5.2.6 Sample Size

It was unclear previous to running the experiment what number of participants would be needed to make significant findings regarding the hypotheses. The overall target was to aim for 30 participants per group.

5.2.7 Randomization

Assignment of groups was according to the covariate adaptive randomization approach [Sur11]. Based on the entered college year or status as professional, a random number between 1 and 3 was generated. The number was then assigned to the participant and noted in the database for that level of education. The next participant of the same level of education would then be assigned one of the other two groups based on another random number, and their assignment would then be noted again leaving the third participant of that level of education to be assigned the last remaining group. After this the memory of assigned groups for that level of education was reset and the fourth participant could be assigned to either of the three groups. This method helps to ensure even distribution across the different groups amongst each level of education, which is used as a measure for programmer experience.

5.2.8 Blinding

The experiment is double blind. Experimenters did not have a direct hand in the assignment of groups and could not influence the participants during the experiment due to its nature of being an online experiment. Likewise, participants were not made aware of the specific hypotheses, research direction, or their specific group and therefore were kept blinded to the intentions of the researchers.

5.3 Quantitative Results

This section will present the results of the experiment. The demographics will be discussed in section 5.3.1, the descriptive statistics are presented in section 5.3.2 and the analysis of the data is presented in section 5.3.3. For this experiment 195 students of the University of Nevada, Las Vegas were recruited from computer science classes. The script used to analyze the data can be found at `https://bitbucket.org/stefika/replication/src/master/2019UesbeckDissertation/embedded/fileswitching`.

5.3.1 Recruitment

Of the 195 students that participated, 18 had to be removed from the overall data set because they did not complete all tasks of the study. Of the 177 remaining students, 3 were freshmen, 40 were Sophomores, 64 were Juniors, and 67 were Seniors. Additionally, there were 2 graduate students and 1 professional. Since

Table 5.2: Times per task in seconds

	Polyglot			JavaScript			PHP			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 3	51	2254.53	1278.26	61	1542.10	1301.66	59	1522.53	1301.73	1748.18	1329.05
Task 4	51	3253.78	774.00	61	2633.08	1180.84	59	2563.07	1227.59	2794.05	1128.58

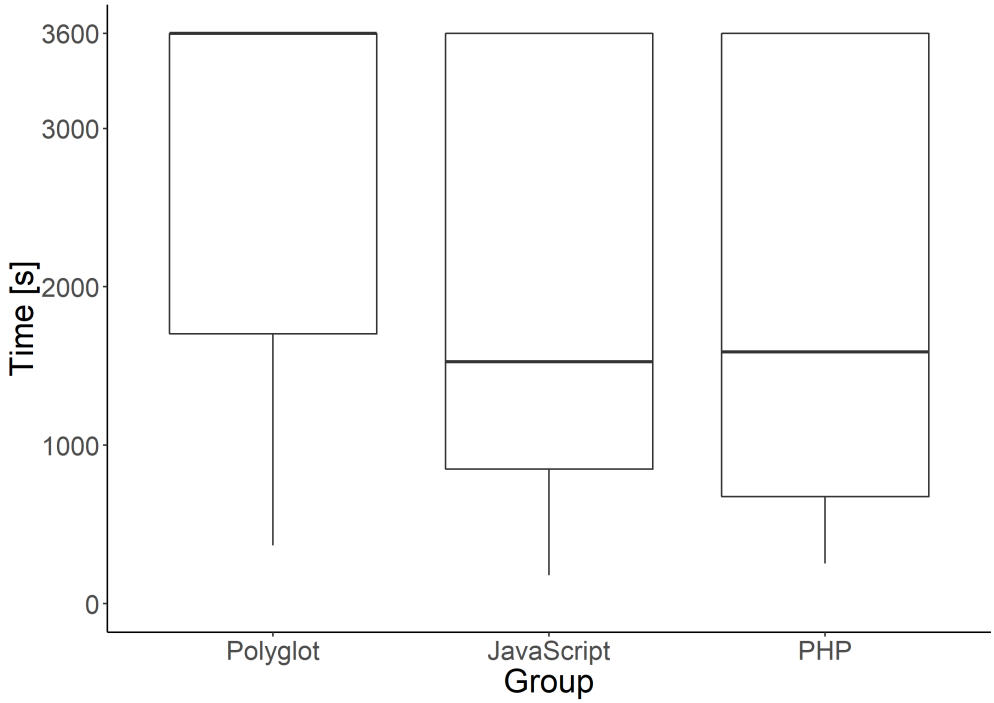

Figure 5.1: Boxplot of results between the groups

the populations of freshmen, graduates, and professionals were very small and since they would cause issues with some tests and create an unclear picture of the data in graphs, they were also removed from the data set, leaving 40 Sophomores, 64 Juniors, and 67 Seniors. Overall, that reduces the sample to 171 participants. Of the dataset, 51 participants were in the Polyglot group, while 61 were in the JavaScript group, and 59 were in the PHP group. Forty-seven of the participants identified themselves as female, while one participant selected "Other" for their gender. The average age of participants was 23 years (M = 23.13, SD = 4.00). Additionally, 31 participants selected a language other than English as their primary language (22.14%). As tasks 1 and 2 were unrestricted with regards for time and intended as warm-ups, all following analysis and graphs will only concern the results of tasks 3 and 4, making for 342 id-task combinations overall.

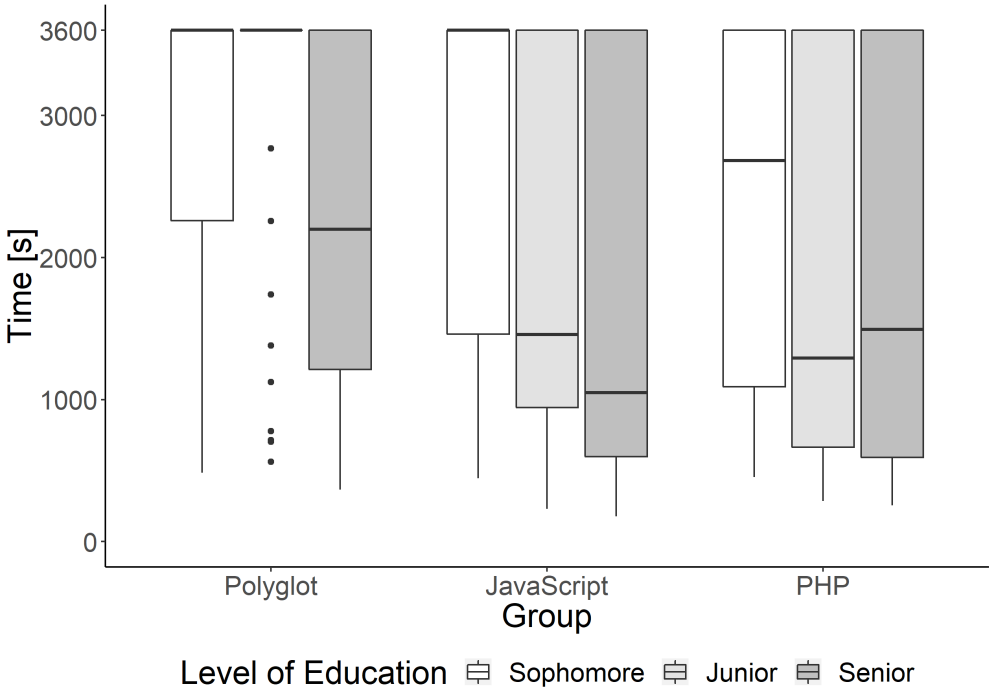

Figure 5.2: Boxplot of results between the groups based on their level of education

Table 5.3: Errors per task

	Polyglot			JavaScript			PHP			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 3	51	22.22	24.00	61	12.52	18.44	59	14.51	19.38	16.10	20.84
Task 4	51	18.69	16.17	61	13.18	14.52	59	16.24	22.89	15.88	18.31

5.3.2 Baseline Data

The Polyglot group was the slowest group regarding average task completion time with about 46 minutes (M = 2754.16s, SD = 1165.14), the JavaScript group was faster with about 35 minutes (M = 2088.09s, SD = 1353.17), and the fastest group was the PHP group with about 34 minutes (M = 2042.80s, SD = 1363.84). Task 4 had a longer average task time overall with 47 minutes (M = 2794.05s, SD = 1128.58), while task 3 took participants about 29 minutes (M = 1748.18s, SD = 1329.05). A detailed breakdown of the task times based on groups can be found in table 5.2. Further, figure 5.1 shows the differences between the groups in a box-plot, while figure 5.2 shows a similar boxplot, broken down by year of education. A graph with regard to the differences by group between the tasks can be seen in figure 5.3, which shows the mean times per tasks with 95% confidence intervals as error bars.

Aside from time, the platform also recorded the number of errors made by participants. These events were then counted up as the number of errors before achieving a complete task. The group with the highest

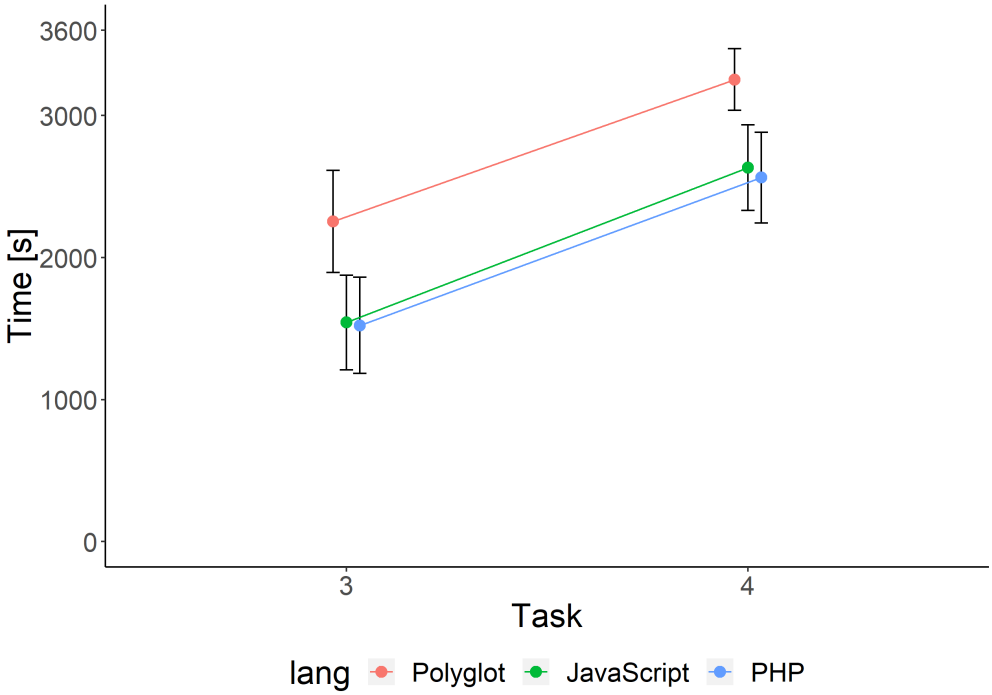

Figure 5.3: Graph of results between the groups

Table 5.4: File switches per task

	Polyglot			JavaScript			PHP			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 3	34	10.71	8.09	49	9.29	7.77	47	9.50	7.41	9.73	7.69
Task 4	13	7.38	4.37	32	6.19	4.41	35	6.66	5.55	6.59	4.89

average number of errors per task was the Polyglot group (M = 20.45, SD = 20.44), followed by the PHP group (M = 15.37, SD = 21.14), and lastly the JavaScript group (M = 12.85, SD = 16.53). A summary of the number of errors per task per group can be found in table 5.3. A comparison of the amounts of errors between groups can be seen in figure 5.4.

Lastly, the experimental platform recorded the number of file switches that occurred during the experiment. To properly analyze the occurrence of file switches between groups, the data need to be restricted to all participants that maintained working on the experiment until either the time ran out or the task was finished, as someone giving up would not accumulate any more file switches, which would bias the results. Thus, all id-task combinations in which a participant gave up were removed from the data, leaving 210 data entries.

On average, the Polyglot group switched the most (M = 9.79, SD = 7.36), while the PHP group switched less (M = 8.28, SD = 6.79), and the JavaScript group switched the least (M = 8.06, SD = 6.79). Most of the switching happened in task 3 (M = 9.73, SD = 7.69), while switching in task 4 was more rare (M =

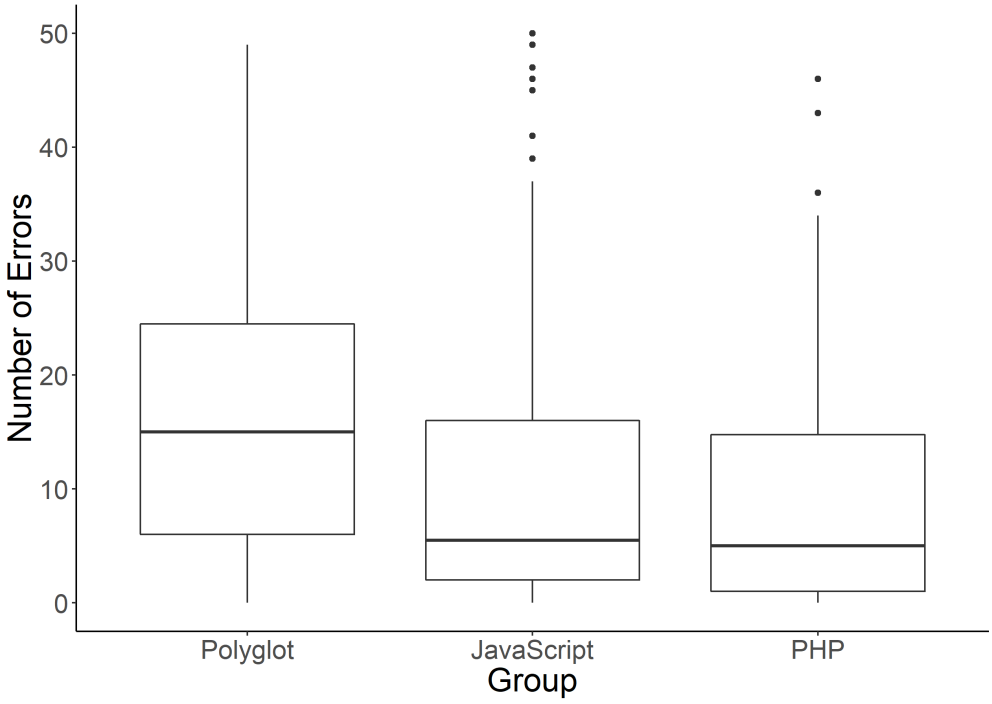

Figure 5.4: Errors boxplot of results between the groups

Table 5.5: Bonferroni corrected t-test of average times by groups

	Polyglot	JavaScript
JavaScript	< 0.001	-
PHP	< 0.001	1.000

6.59, SD = 4.89). A breakdown of average switches by group and task can be seen in table 5.4. A box-plot comparing the amounts of switches between the groups can be seen in figure 5.5.

5.3.3 Analysis

A mixed designs repeated measures ANOVA was run using the R programming language using time as the outcome, task as within-subjects variable, and group as the between-subjects treatment as well as level of education as a random variable between subjects. Sphericity was tested and where appropriate, the Greenhouse-Geisser corrections will be substituted in the reporting of the results that follows.

Using $p < 0.05$ as the threshold for significance, there was a significant interaction effect between group, level of education, and task $F(4, 162) = 2.64,\ p = 0.036$ ($\eta_p^2 = 0.014$). Further, there was a significant effect for the within-subjects factor task $F(1, 162) = 132.94,\ p < 0.001$ ($\eta_p^2 = 0.147$). The between-subjects factor level of education was also significant $F(2, 162) = 4.67,\ p = 0.011$ ($\eta_p^2 = 0.041$), as well as the between-subjects factor group $F(2, 162) = 7.14,\ p = 0.001$ ($\eta_p^2 = 0.059$). A post-hoc t-test using Bonferroni

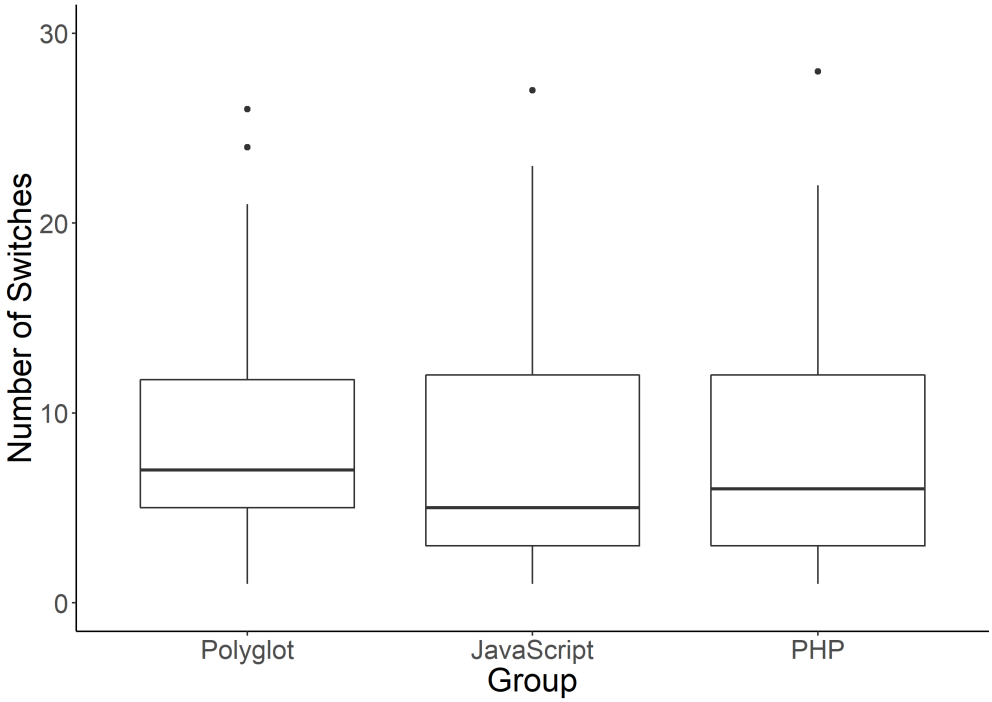

Figure 5.5: Switches box-plot of results between the groups

correction was performed, comparing the three groups of the experiment. The results of the test are reported in table 5.5 .

Further, a t-test of between participants reporting their primary language to be English and participants that reported a different language was performed and the results show a significant effect $t(103.28) = 4.23, p < 0.001, r = 0.384$. The results show that participants that reported that English is their primary language had a lower average time (M = 2144.89, SD = 1349.03), than participants that reported a different primary language (M = 2841.18, SD = 1130.62). A visual representation of the differences between the participants in each language group can be found in figure 5.6. An analysis of the impact of self-reported fluency using an ANOVA using English fluency as a between-subjects factor shows a significant impact of fluency $F(1, 169) = 6.87,\ p = 0.010$ ($\eta_p^2 = 0.039$). There is a significant negative correlation between fluency and time $\tau = -0.13, p = 0.006$.

Investigating error count as a metric, an ANOVA was run, using error count as the outcome and between-subjects variable group. The results of this ANOVA show a significant effect $F(2, 168) = 3.08, p = 0.049$ ($\eta_p^2 = 0.035$) between the groups. A post-hoc t-test with Bonferroni correction shows that there is a significant difference between the Polyglot and the JavaScript group $p = 0.011$ but not between Polyglot and PHP $p = 0.161$ and neither between JavaScript and PHP $p = 0.945$. Error count and time were significantly

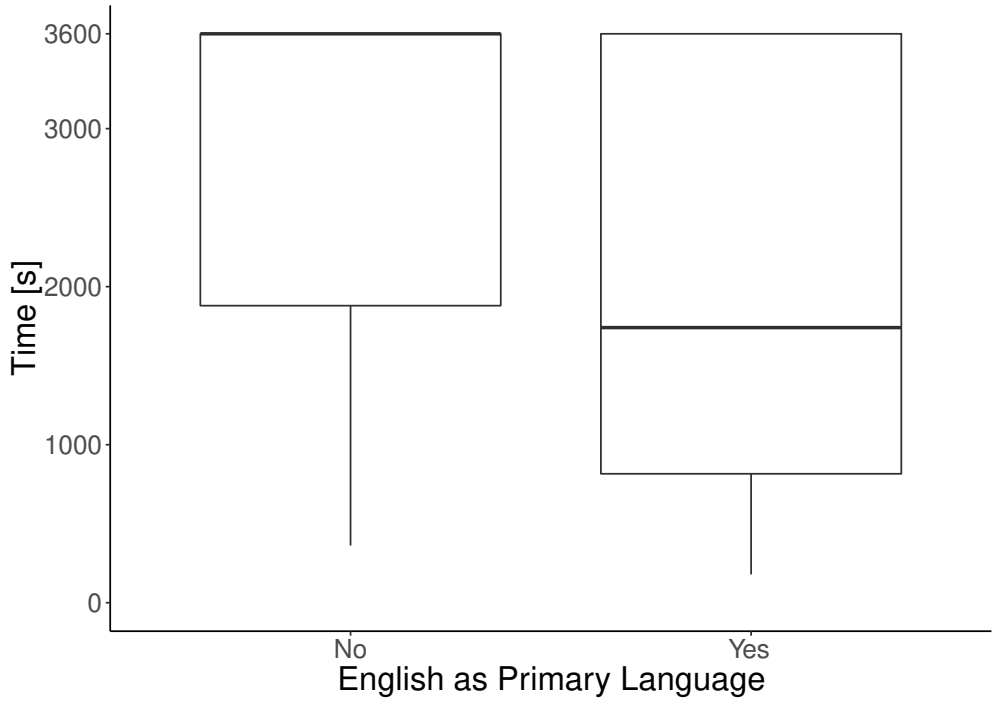

Figure 5.6: Boxplot of results between primary English speakers and non-primary English speakers

correlated $\tau = 0.39, p < 0.001$. This correlation is less pronounced if only the Polyglot group is tested $\tau = 0.25, p < 0.001$, while doing the same test with the two non-Polyglot groups yields a slightly higher correlation as the overall test $\tau = 0.40, p < 0.001$.

File switches were analyzed using an ANOVA using file switches as outcome and the between-subjects factor group. There is no significant effect found for group $F(2, 129) = 0.69, p = 0.50$ ($\eta_p^2 = 0.011$). There is a significant correlation between switches and time $\tau = 0.261, p < 0.001$.

5.4 Qualitative Results

In this experiment the same question as in the study on embedded programming language switching (chapter 4) was asked at the end of the experiment. The question was: "Did you feel like you had to switch between languages during the experiment and how do you think did this affect your progress while solving the tasks?". A positive answer for switching was only expected from participants in the Polyglot group, which mostly held true as most participants from the other groups either recognized there was no switch saying things such as "This was all in Javascript, so I didn't switch a single time." or they spoke about the switch between the language they feel familiar with and the language used in the study: "I only used C++ so it

did not affect my progress while solving the tasks", which might be considered more of a project-level switch.

Of the Polyglot group, 45 participants left a response to the question (88.24% of participants in the group). Of those 45 participants, 31 identified a switch between languages (68.89%). The other 14 participants seemed to be confused by the language. Some made statements that seemed to indicate that they might have thought that they worked in the same language throughout like this statement made by a Senior: "Since php felt like C++ it was very straight forward. Didn't feel like "switching" even occurred." The senior clearly recognized PHP, but did not recognize that he was using JavaScript as well. Another statement from a Sophomore brings this observation more to the point: "not sure. I am unfamiliar with the language." The Sophomore might have thought that they were using the same language in different modalities.

Further, 12 of the 45 participants said that their progress was impacted by the switch between languages (26.67% of the 45 participants, 38.71% of participants that identified the switch). It seems that participants noticed an impact when they had difficulties in switching between the syntaxes of the languages. As a Junior stated: "There were times where I confused syntaxes between the languages". A Sophomore also noted that "[...] I was confused between the language where we had to use language and the one where the syntaxes did not have a $-sign."

5.5 Discussion

This section will first discuss the limitations of this study in subsection 5.5.1 and then move on to interpret the results in subsection 5.5.2.

5.5.1 Limitations

It is important to recognize that no research design is perfect and that there are always limitations of research that make the results less generalizable. This is why this section will list these limitations to the study described in this chapter.

When looking at the differences between the tasks, it becomes clear that the fourth task was clearly more difficult than the third task. This task was iterated on multiple times to make it easier for participants to solve the task. It was tested with real programmers before running the experiment at scale, however most testers were more experienced programmers and thus the task might have been more difficult than intended, which might have influenced the results.

Importantly, it is necessary to acknowledge that using a web platform to run an experiment like this will possibly introduce unnecessary variance due to varying degrees of compliance with the rules of the ex-

periment. The experimenters can only ask participants to create an environment without distractions and not to cheat, while experimenters of a lab-based experiment could control the environment more specifically. Running a lab-based experiment comes at the cost of less flexibility for participants, requiring them to find time slots for the the experiment that work for the lab as well as themselves. This means that the potential amount of participants for the web-based experiment is much higher, likely making for a more generalizable sample. Obvious cases of not working on the experiment or cheating can easily be removed from the sample.

The analysis of primary natural language influence on development productivity has to be considered very carefully, as the measures used for this analysis might not be perfect representations of the language status of participants. The question asked to determine the participants' primary language is "What is your primary language (i.e., the language you speak at home)?". This question is not sufficient to determine when a participant learned their primary language, how much they use it, and when and how they learned English and how much they speak English as well as the English proficiency of participants. The measure for English proficiency used in the study is self-reported and may not be an accurate representation of their actual proficiency. Furthermore, even if primary language and proficiency are accurate, it is possible that these measures only serve as proxies for other factors such as prior education or socioeconomic differences et cetera. Thus, interpretation of the results regarding the relationship between natural language and programming productivity should be done with caution until a more comprehensive test of this relationship is conducted using appropriate linguistic measures for proficiency. *It has to be emphasized that these findings are not sufficient evidence to base hiring or academic decisions on.* The author hopes that these findings might lead to further research into the issue, which, in case that the difference is confirmed, can hopefully lead to solutions to bridge the differences for minorities in computer science in the United States as well as making learning programming easier for people across the globe who are not native English speakers.

5.5.2 Interpretation

The main question of this study was, whether there are differences in productivity between developers who are switching between programming languages and developers who are using only one language at a time. The data show a clear indication that there are differences. The following subsections will discuss the details of the results pertaining to the hypotheses and will be summarized in the last subsection.

Relationship Between Polyglot Programming and Task Time

$H_0 1$, which reads *"Polyglot file switching has no impact on* ***productivity*** *compared to monoglot file switching."*, relies on task time as the measure of productivity in this study. From the results we clearly have to reject the null-hypothesis, as the ANOVA shows a significant effect for the differences between groups ($p = 0.001$), with 5.9% of the variance of the model accounted for by the group variable. Figures 5.1 and 5.3

paint a clear picture that the Polyglot group stands apart from the other two groups in average task times, while both single language groups achieve close to the same results. This is also supported by the post-hoc t-test which shows that there are significant differences between Polyglot and both other groups, while there is no significant difference between the JavaScript and PHP group. It is interesting to note how similar the results for the two different languages are, which isolates the effect to the use of Polyglot. This difference between the Polyglot group and the similarity of task times indicate a clear cut difference in productivity between developers employing polyglot methods and developers working on with language at a time.

Relationship Between Polyglot Programming Error Rate

Null-hypothesis 2 ($H_0 2$) *"Polyglot file switching has no impact on* ***error rate*** *compared to monoglot file switching."* has to be rejected as false. The ANOVA results show a significant difference between the groups ($p = 0.049$) with an effect size of 3.5% VAF. The polyglot group produced more errors on average than the participants in the JavaScript and PHP groups, though the post-hoc test only shows the difference between the Polyglot and the JavaScript group ($p = 0.011$). Once again, there is no significant difference between the JavaScript and PHP group ($p = 0.945$). This indicates that even when considering this different measure, there is a negative impact of using polyglot programming on software development. Seeing that more errors are made in the Polyglot group also suggests that the occurrence of errors might contribute to the difference between the Polyglot group and the two non-polyglot groups. In fact, error rate was positively correlated with task time $\tau = 0.39$, however the correlation effect when just analyzing the Polyglot group shows a weaker correlation $\tau = 0.25$. A likely explanation for this effect is that participants in the Polyglot group confused both languages and might have created more, but less time intensive errors, such as syntax errors that are relatively easy to fix when realizing that they used the wrong language syntax.

Relationship Task Time and File Switches

The last hypothesis concerning different measurements to determine the differences between the three groups is $H_0 3$, stating *"There is no relationship between the amount of switches between files and productivity."* With the data collected in this experiment, this hypothesis can be rejected. Testing the correlation between file switches and time, there was a significant correlation between the two measures $\tau = 0.261, p < 0.001$, though the correlation was not very strong. Interestingly, the ANOVA results for file switches between groups show that there was no significant differences in the amount of file switches between groups ($p = 0.50$).

Programmer Attitudes and Observations Regarding Polyglot

The analysis of the responses given to the exit question regarding whether developers noticed switches between the languages and whether they felt like that these switches affected their productivity shows that in this experiment, approximately 69% of participants that were in the switching group actually recognized the switch, while participants that did not recognize the switch seemed to be thinking that they were using the same language in both files, perhaps using different features of the language. This is a large number of participants recognizing the differences compared to the results of chapter 4's study. However, this is not surprising as the switch was more obvious as it involved a clear action in switching between the files and the tabs that needed to be clicked to switch between files displayed their file names including file type endings (.js and .php respectively), which should be a good indication of the different language in addition to the difference in syntax

As seen in section 5.4, only about 27% of participants recognized that there was an impact on their performance when switching between programming languages. The ones that did, noted that they thought that the difference lay in the differences in syntax. While JavaScript and PHP have similar C-style syntax, they have many small differences, one of the most notable differences in this experiment amongst participants being the way variables are notated with a dollar symbol like this: $\$variableName$. From these results, it can be deduced that programmers do generally recognize the switch between languages on a file level, however they do not really consider this to impact their productivity while programming. The ones that do consider there to be an impact blame the effect on the differences in syntax between the languages confusing them.

Differences Based on Primary Language

Analysis of the primary language of participants shows a significant difference between primary English speakers and non-primary English speakers. According to the results, primary English speakers spent 36 minutes on average solving tasks, while non-primary English speakers spent about 47 minutes. The effect size was determined to be $r = 0.384$. An ANOVA analyzing the impact of fluency shows a significant effect for fluency on task time with 3.9% VAF effect size.

Non-primary English speakers more commonly did not complete tasks before the time limit or gave up more regularly. The easiest explanation for this observation is that people might have had difficulty understanding the instructions, which is supported by one non-primary English speaking Junior's comment:

> "This would affect my progress while solving the tasks because the way the instruction was written is a little bit difficult for me to understand. I have to read them multiple time, so the languages does affect the time I completed the tasks."

Other than difficulties understanding the instructions, it could also be that less proficient English speakers have difficulties understanding some syntax choices and function names that would possibly provide very proficient English speakers with more information.

While this is the second finding of a difference in programming productivity based on English proficiency (see chapter 4, it is important to remind the reader of the limitations stated in section 5.5.1. Assessing primary language and proficiency as done in this experiment would not be regarded as sufficient by linguistic researchers and is therefore at most an approximation. Future research needs to investigate this issue with more sound assessments. If the true cause of reduction in productivity can be found, it can hopefully be remedied to make learning programming easier for everyone who does not speak English natively.

Summary

Overall, this experiment shows that there are significant differences between a group using two languages compared to groups using only one. Based on the observation that there are no significant differences between both one-language groups it seems likely that the observed effect can solely be ascribed to the use of more than one language in the Polyglot group. The exact cause of the effect is not clear, but the data shows that the Polyglot produced more errors on average than the other groups, which suggests that programmers switching between languages might become more prone to making mistakes that cost time to fix. Looking at the switches between files, there are no significant differences regarding the occurrence of switches between and the groups, but there is a correlation between switches and task times. This could indicate that the delay caused by file switches might be stronger in the polyglot group while the number is larger. However, the strongest explanation based on the presented data is that participants in the Polyglot group took longer because they produced more errors due to the switch between languages.

When considering the connections of these findings with linguistic research on switching between natural languages, it is likely that participants were in a mostly monolingual switching mode (see [Ols17] and explanation in section 2.4.3) when being forced to switch between languages for the first time. Many participants stated in the comments regarding computer language switching that they only know C++ and that they have never had to switch between languages before. Based on their likely monolingual switching mode and the ICM model, that means the first language they were using was their dominant language, i.e. they would have to switch to the second language with great effort, or with great switching cost, taking time to adjust to the second language. There is no clear indication in the results of the study that there was a greater delay between when a participant switched between languages until they started typing than in the groups

where they were merely switching between files of the same language. This indicates that the switching delay that might be observed in linguistics between natural languages is not found in computer language switching, however the task design might not have encouraged enough switching to make out a difference between the groups. Opposed to natural language switching that typically produces correct utterances in the language that was switched to by the speaker [YTF17], computer language switching seems to result in more programming errors compared to groups not switching between languages.

5.6 Conclusion

In this chapter, an experiment testing the effect of file level computer language switching was described and the results were presented and discussed. The goal was to determine whether file level computer language switching might influence the productivity of software developers. The experiment had three groups, one that had to switch between JavaScript and PHP, one using just JavaScript, and one using just PHP. The tasks in this experiment were designed to represent generic web development tasks in which the interactions between two files of code were important.

The results of the experiment show a clear difference in task times between the polyglot group and the two monolingual groups. The polyglot group needed significantly more time to complete tasks. Analysis of the error rate of the three groups shows that the polyglot group also encountered more errors on the way to completing tasks, showing a similar pattern to the increased task times, suggesting that the switch between languages might lead to more errors, which decreases productivity. No clear patterns could be found in the amount of switches between files needed to complete the tasks, indicating that the cognitive switch between languages might not be a major factor in increased task times. This suggests that findings in the linguistic literature regarding code switching in natural languages, which suggests that switching between languages typically produces few errors but has a cognition based delay when speakers or listeners have to switch between languages [YTF17, Gre98, Ols16] might not apply to computer language switching and that there are differences between the cognitive mechanisms involved in switching between computer languages and natural language code switching.

The majority of participants switching between programming languages recognized the switch, but most expressed that they did not feel the switch impacted their productivity. This shows a disconnect between perceived impact of polyglot programming and the actual impact on productivity. The few participants that did note a decrease in productivity suggested that their main issue was switching to the right syntax and that syntax errors cost them the most time in connection with switching. This study also found a significant differences in performance between speakers of different primary languages, suggesting that there is a significant influence of natural languages on the programming experience, a topic that will have to be

investigated in more detail in future research.

Chapter 6

Design of a Data Management Library

In this chapter, a design for a data management library is described. The design is based on the insights gathered from the experiments in chapters 4 and chapter 5, as well as the previous research reviewed in chapter 2. Section 6.1 will be the introduction, section 6.2 will describe the design goals of this library, while section 6.3 will describe the different considerations to be made in the design of the library, and section 6.4 will show the design of the query system based on examples. Section 6.5 will show the class diagram and the grammar needed to parse conditional syntax and will be followed by section 6.6, which discusses alternative design ideas. Finally, the chapter will be concluded in section 6.7.

6.1 Introduction

Data are an integral part of programming and past trivial amounts of data, these data have to be stored, retrieved, and changed when needed. Common approaches to saving data are plain files as well as databases such as relational and more recently NoSQL [HHLD11] databases. When it comes to storing complex information, relational databases are the most common approach to saving data. They allow for the definition of multiple data tables that can optionally be connected by key values in the tables, which allows for complex models of data. While there were different approaches to querying data from database management systems [BBE83, YS93], SQL (Structured Query Language) prevailed as the main approach to querying data from relational databases. While the language was first standardized in 1986 by ANSI, numerous dialects of the language exist and are continuously developed further [RKS+09]. The dialects used often depend on the specific product that is being used, such as MySQL, postgreSQL, SQL server, et cetera.

Other popular data sources are comma separated value (CSV) files or other similar delimited plain text data formats (e.g. using tabs or spaces as separators between values) that represent data in a table format. These formats are easy to inspect with software products such as Microsoft's Excel, Google's Sheets, as well as LibreOffice Calc, but are also easy to load into table formats in various programming languages

such as popular languages for statistical analysis, e.g. R and Python. In programming languages such as R and Python, data from CSV files and databases are often represented as data frames, which keep the table representation of data intact. These data frames are also the central to the use of many statistical analyses.

Analyzing data in programming languages therefore requires the ability to extract data from databases or structured plain text files to a in-memory representation of the data. To further analyze the data based on the in-memory representation might further require filtering columns and rows of data, changing data, and aggregating data. For extraction of data from databases, SQL is commonly used in industry [RLMW14], however, often programming languages take different approaches to integrating SQL calls into the language. One approach is the direct integration of strings of plain SQL of whichever dialect is needed by a programmer using a library. Other approaches, such as Object-Relational-Mapping (ORMs), attempt to bind classes defined in a programming language to tables in the database, each object of the class then represents one row of the table. In ORM systems, fetching, updating, and creating new entries in the database is often automated for simple cases. Critics of the approach note that this is an elegant solution for these more simple cases but that there can be issues for more complex queries due to a problem called "impedance mismatch" [IBNW09], which originates from the differences between object and relational storage. Research shows that configuration issues can make interaction with data cumbersome and lead to performance issues [CJ10].

Seeing as managing data is an important aspect of programming, developing an easy-to-use data management library appears imperative. This chapter will propose the design of a data management library based on insights of existing research, as well as the research conducted in chapters 3, 4, and 5. The library will first be described abstractly, and then specific implementation considerations for a prototype in the Quorum programming language will be made.

6.2 Goals

When developing a library, the first step is to define the goals the library is supposed to fulfill. The library designed in this chapter is aiming to fulfill the following goals:

- **G1:** Being able to read data from relational databases and structured data files.
- **G2:** Supporting standard create, read, update, and delete functionality.
- **G3:** Storing data automatically in a ready-to-use data format for statistics.
- **G4:** Having a querying interface that is easy-to-use for inexperienced programmers.

While many of these goals are commonly accepted functionality of data management libraries, the main contribution of this work to the scientific literature will be the evidence-based design of the querying interface. Further, the design should be applicable to the Quorum programming language[1], for which no data management library exists at the moment.

6.3 Design Considerations

This section will discuss the considerations made when making the different design decisions that together make up the proposed design.

6.3.1 Data Representation

The easiest way to represent data from table based databases and table based structured data files in memory is in a table format, or data frame as it is called in R and Python's pandas. This eliminates the need to try to match the table format to the object structure of object-oriented programming languages and keeps the data in a format that can easily be used to compute statistics. A data frame is a collection of columns that contain data of the same type, much like a database table, and have a name identifying the data in the column (e.g. "id" or "full name"). This format is still useful when only a single entry is needed, as the table can just contain one row of data that can be used further if the programmer wishes.

6.3.2 Querying

Extracting data from a structured plain text file is relatively straight forward, as the common assumption in this case is that the programmer is intending to load the whole file (unless the file is extremely large). Thus the system only needs to know the location of the file to be able to load it. Loading data from a database however is more complicated. The system needs to establish a connection to said database, authenticate, and send a query to request data. As previously discussed, using queries in form of strings is the most basic approach to sending requests for data to a database. The Java Database Connectivity API (JDBC) is a well known example of this. However, this technique does not provide any checking before the SQL is sent to the database and therefore requires a good understanding of SQL. From the experiment in chapter 5 it is apparent that polyglot programming causes reduced programming productivity and thus shying away from querying solutions that require the direct use of a secondary computer languages is advisable. The experiment in chapter 4 compared three different approaches to querying data, one of which clearly failed to provide any advantages. This naive implementation of the structure of SQL into the object-based structure

[1] https://quorumlanguage.com/

of Java produced the worst results and should not be considered a good solution to query databases. While more experienced developers might have performed better using the polyglot approach, based on the results from the experiment in chapter 5, this solution has to be ignored in favor of the hybrid approach, which proved easier-to-use for less experienced developers without a negative impact on those with more experience, which fits goal **G4**. However, the approach used in the experiment was preliminary and might be improved upon in this design, while still capturing the spirit of it.

The idea behind the hybrid query design was to enable the use of easy-to-understand method calls to create a query object, which represents the desired constraints on the data. While the same idea was behind the approach represented in the object-oriented group, the code used in the object-oriented group took this idea to the extreme and required users to use chains of method calls to generate logic trees for the conditions used in filtering the data (the `WHERE` clauses), while the design of the hybrid group conceded that this approach was difficult to use and opted to maintain the syntax used in SQL to represent the filtering conditions. This lead to a tool that forced a quick language switch to a limited different syntax.

After completing additional research into the effects of polyglot programming, it seems advisable to follow an approach that attempts to emulate the hybrid approach while also trying to make the secondary syntax used for conditions as similar as possible to the host language. With the goal to avoid the use of SQL-like syntax to avoid enforcing polyglot programming. Lambda functions could potentially be used to create similar conditions for in-language querying but there is evidence suggesting that lambda functions are be difficult to use [USH+16].

The query syntax should be applicable both to filtering in-memory data held in table objects (and possibly other data structures) and for sending requests to databases. This requires the possibility for a query object to be converted to a SQL query before being sent to a database, which would be harder to do using lambda functions as syntax for conditionals.

While the design of this library is based on the best available evidence, it needs to be said that not all choices made while creating the design were based on rigorous data. There is no comprehensive analysis of naming choices and other smaller decisions when designing libraries, which makes it necessary to use a best guess at what might be easy to use and understand. The design needs to be tested against older designs to see if it does improve interaction with a database and additional studies to revisit naming choices might be necessary.

6.4 Query Design

This section aims to give insight into more specific design choices in the design of the library. The section will have an explanation of the design question or choice, one or more code samples regarding that design choice, and a discussion of relevant evidence, if any. As the specifics are modeled for the Quorum programming language, the code samples will be in Quorum-based pseudo code. If not shown specifically, the scaffolding code in code sample 28 can be assumed to be included in the code samples.

```
1  text location = "mysql://databaselocation"
2  text user = "greatuser"
3  text password = "bestpassword"
4  Database db
5  db:Connect(location, user, password)
6  SelectQuery q
7  q:SetTarget("users")
8
9  [building the query]
10
11 Table t = db:Search(q)
```

Code Sample 28: Scaffolding to following examples

6.4.1 Design of Default Options

The library is aiming to contain sensible defaults to keep simple interactions with the database quick to write. As an example, this is reflected in the default settings of a fresh Query object, which retrieves an entire table in its most basic setting, being equivalent to the SQL star operator. This is meant to allow users to easily retrieve all data if they so wish. They can then further constrain the query to limit the data they might want to retrieve. Additionally, there is a further method to retrieve the data of an entire table by calling a convenience method on the database object, making it possible to skip the step of creating a query object for this simple purpose.

The example in code sample 29 shows the user of the defaults of the `Query` object. A connection with the database is established, a `Query` object is created, the query target, in this case the name of a table in the database but optionally a query instead, and then the `search` function is called on the database object using the query as the parameter for the search. Since the query was not further specified, the defaults of the query are applied in the search. This includes the projection[2] of all columns and the selection of all rows in the table. The convenience method `Database:GetTable(text)` allows for the retrieval of an entire table in just one step. An example of this is available in code sample 30.

[2]Column choice

```
1  SelectQuery q
2  q:SetTarget("users")
3  Table t = db:Search(q)
```

Code Sample 29: Full table query

```
1  Database db
2  db:Connect("mysql://databaselocation", "greatuser", "bestpassword")
3  Table t = db:GetTable("users")
```

Code Sample 30: Full table without defining query object

6.4.2 Selection of Fields

In this library, projection is handled by adding field names to the `Query` object one-by-one while defining their expected type by choosing the appropriate method to add the field. The field names need to completely match the column names in the database table the way a SQL query would address them. Alternatively to this, listing all fields in the request in a comma-separated list was discussed. While that was the way the hybrid group in the embedded computer language switching experiment (see chapter 4) was designed, the introduction of a custom syntax for listing fields would require novices to learn additional syntax that is only available in one place of the language and there is no evidence that this approach to handle projection is superior to using language features that already exist and most programmers might be already familiar with. This additionally removes and instance of switching languages (from the host language to a tiny DSL). A concrete example of projection in this library can be found in code sample 31. The basic methods to add fields are `AddIntegerField(text)`, `AddNumberField(text)`, and `AddTextField(text)`. An additional consideration has to be made regarding the use of renaming of columns in projection. For this purpose an additional method was introduced that allows users to rename previously added column names. A concrete example of this can be found in code sample 32. The first parameter of the `RenameField(text,text)` function names the existing column, while the second establishes the new name for the column.

```
1  /*
2          SELECT id, time, comments
3          FROM ...
4  */
5  q:AddIntegerField("id")
6  q:AddNumberField("time")
7  q:AddTextField("comments")
```

Code Sample 31: Basic field selection

```
1  /*
2          SELECT id, time AS Tasktime, comments
3          FROM ...
4  */
5  q:AddIntegerField("id")
6  q:AddNumberField("time")
7  q:AddTextField("comments")
8  q:RenameField("time", "Tasktime")
```

Code Sample 32: Basic field selection with renaming

6.4.3 Aggregation Functions

Similar to the projection of columns, queries also need to be able to apply aggregation functions. For this purpose a number of aggregation field functions were added to the `Query` class. These include `AddAverageField(text)`, `AddCountField(text)`, `AddConcatenateField(text, text)`, `AddMaximumField(text)`, `AddMinimumField(text)`, `AddStandardDeviationField(text)`, `AddSumField(text)`, and `AddVarianceField(text)`. These come together with the ability to add fields to the list of fields by which an answer is grouped for the use of the aggregation functions. For this purpose the `AddGroupBy(text)` function was added. As is typical in SQL, all non-aggregated columns in the list of columns to be included in the result set have to be part of the list of columns that the results are grouped by. The library should check for this constraint at the time the query is supposed to be used for a search and throw an error if it is violated. An example of an aggregated column is presented in code sample 33. An example using the `AddGroupBy(text)` method is given in code sample 34. This example also shows the naming convention for using aggregates of columns. The default naming convention for MySQL would be to name the average column "AVG(time)". As this naming convention would be confusing to users who are not aware of the name of the AVG function in MySQL since the library method is not using the shortened "AVG" and spells out the name to "Average" instead, the library automatically renames the field internally to a name according to the standard naming convention used in Quorum, and thus renaming the field in the library requires the use of that name instead.

```
1  /*
2          SELECT AVG(time)
3          FROM ...
4  */
5  q:AddAverageField("time")
```

Code Sample 33: Basic aggregated column

```
1  /*
2          SELECT id, AVG(time) as "time"
3          FROM ...
4          GROUP BY id
5  */
6  q:AddAverageField("time")
7  q:RenameField("averageTime", "time")
8  q:AddGroupBy("id")
```

Code Sample 34: Aggregated column with group by and renaming

6.4.4 Where Clauses

`WHERE` clauses represent the most difficult part of designing this library, as they are boolean expressions comparing columns with values or other columns and there is no syntax in the Quorum programming language that allows for easily expressing these comparisons. The Java programming language in version 8 and up and other popular languages such as Python and JavaScript allow for the use of lambda functions, which can be used for this purpose when comparing data within the context of the language. While this might seem like

a good argument for the implementation of lambda functions, there is evidence that lambda functions can be more difficult to use for programmers [USH+16], lambda functions can be substituted with short classes, and translating lambda functions to SQL code is non-trivial.

The approach taken in chapter 4's hybrid group was to embed a small domain specific language into the library, which allowed for the use of SQL syntax for `WHERE` clauses by the way of string parameters to functions. The strings were then parsed and checked for errors by the library before being applied to tables. This approached worked best for less experienced programmers compared to full SQL strings as queries and especially when compared to the approach of building these clauses by chains of method calls. Based on the evidence that the hybrid approach worked best for novices, it was selected for the design of this library. However, based on the results from chapter 5 it seems important to consider that language switching can cause delays, even when the syntax between languages is close. Thus, the design of the hybrid group `WHERE` clauses was modified for this library to allow for a closer syntax for the host language or even a full integration into the language as a special construct. This section will, but for a quick example of the integrated version, primarily consider the string version, as implementing the integrated version might provide extra checking at compile time but would also come with significant implementation cost without concrete evidence that it is worthwhile. The integrated version would completely assimilate the DSL into the language itself, and eliminate the barrier of handling the additional syntax as strings.

The syntax for the search expressions is heavily based on the syntax of the hybrid group but some changes were made to make the syntax closer to the one of Quorum, reducing the difference between the Quorum host language and the search DSL. For example, as databases need to be able to disambiguate duplicate column names between tables included in the query, users need to be able to identify columns of specific named tables by using the `tablename:columnname` syntax. Here, the typical period used by SQL for this purpose is substituted by a colon to match Quorum's syntax for referencing fields of objects. A departure from the Quorum syntax is that the search DSL syntax allows for the use of predefined aggregation functions such as `Average` for the use in `HAVING` clauses as well as the use of SQL set notation. Overall, the grammar defining well-formed search strings largely resembles the grammar of Quorum expressions.

In the integrated version, the library would be designed to rely on a structure called "search", which would be declared similarly to a variable. The definition would start with the reserved keyword "search", followed by the desired name for the instance of search and an equals sign with an expression on the right-hand side of the equals sign making up the actual content of the `WHERE` clause. The syntax of the search expression would be similar overall to the Quorum expression syntax but would have some differences in syntax and semantics to be able to function as conditions for database transactions. First, all identifiers named in the expression that would not previously be defined in the program context would be considered

to be column names and would not act like host language variables. Program variables and method calls in the expression would be evaluated and their values would be used in the query. The language needs to check whether this definition is a valid reference to a field of an object, otherwise it needs to be handled as a column name.

```
1  /*
2          SELECT *
3          FROM ...
4          WHERE id > 120
5  */
6  q:Filter("id > 120")
```

Code Sample 35: Basic filtering

A basic example of using this syntax can be seen in code sample 35. There, a small search string is added to a `Query` object called `q` by calling the method `Filter`. At run time, the search expression is parsed and turned into a tree structure to represent the condition by the Query class. The `Query` object will retain this tree structure as an object and the tree can either be used to generate a part of a SQL-query in string format, or to operate on other data structures at run time. An example of how variables from context can be integrated in these searches can be seen in code sample 36. The variable `timeConstraint` is defined before the filter defined and is added to the string by concatenation. Code sample 37 shows how a variant with language integration could achieve the same query as seen in code sample 36 by integrating the value automatically. A word of caution: As SQL-injection is a common security threat, adding user defined values to SQL strings is not recommended. For this purpose, filtering unexpected SQL syntax from user provided strings is recommended.

```
1  /*
2      SELECT *
3      FROM ...
4      WHERE id > 120 and time > 23.1
5  */
6  number timeConstraint = 23.1
7  q:Filter("id > 120 and time > " + timeConstraint)
```

Code Sample 36: Basic filtering with variable

```
1  /*
2          SELECT *
3          FROM ...
4          WHERE id > 120 and time > 23.1
5  */
6  number timeConstraint = 23.1
7  search condition = id > 120 and time > timeConstraint
8  q:Filter(condition)
```

Code Sample 37: Basic filtering with variable using language integration

6.4.5 Joins

Joining together two or more tables for select queries is an important aspect of retrieving data from databases. To enable the library to perform joins, the `Query` class has a number of different methods: `AddNaturalJoin(text, text)`, `AddCrossJoin(text, text)`, `AddLeftJoin(text, text, text)`, `AddRightJoin(text, text, text)`, `AddInnerJoin(text, text, text)`, and `AddFullJoin(text, text, text)`. It should be noted here, that researching whether these names a meaningful to users might be worthwhile to make the library easier to use for inexperienced programmers. While the functions for natural joins and cross joins just need the names of the tables involved in the join operation, the other joins also include a search string to define the join condition. The joins are added to a list of joins necessary to perform the query as a chain of joins is possible in a SQL query. An example of a natural join can be found in code sample 38. The example of a cross join in code sample 39 also shows that it is possible to join tables in the database with other queries. Code sample 40 shows an example of a left join, representing the group of joins that require a search argument. The `LeftJoin(text, text, search)` method is called using a search string which stipulates that the ids of table events and table users have to be equal to add the join to the `Query` object. This is similar to the implementation in the hybrid group in the embedded computer language switching experiment (see chapter 4). There, the syntax did not allow for other join conditions than joins on equal values in two columns, which was sufficient for the tasks given in the experiment. The change toward using search strings to define the condition allows for a larger variety of conditions such as `"events.id > users.id"`, or `"events.id = users.id + 23"`.

```
1  /*
2          SELECT *
3          FROM
4            users NATURAL JOIN comments
5  */
6  q:AddNaturalJoin("users", "comments")
```

Code Sample 38: Natural join

```
1  /*
2          SELECT *
3          FROM
4            users CROSS JOIN comments
5  */
6  SelectQuery r
7  r:SetTarget("comments")
8
9  q:AddCrossJoin("users", r)
```

Code Sample 39: Cross join with query argument

A more complex example of a join using the proposed library can be seen in code sample 41. In this example, a self join of table events is performed by joining on the condition that the `eventid` is in a set of values provided by a second select query. This example shows the ability to use queries as part of a search condition, allowing for extensive combinations of search strings and queries to create flexibility in how queries are written. It is achieved by converting the search string with a placeholder (`:min`) into a `Predicate` and

```
/*
        SELECT *
        FROM
          events
          LEFT JOIN
          users
          ON events.id = users.id
*/
q:AddLeftJoin("events", "users", "events.id = users.id")
```

Code Sample 40: Left join

then matching the placeholder with the query object using `ResolvePlaceholder(text, query)`. Another example of using nested queries can be found in code sample 42, in which table `events` is joined with the result of a nested query.

```
/*
        SELECT *
        FROM
          events AS eventsA
          LEFT JOIN
          events AS eventsB
          ON eventsB.eventid IN
             (SELECT MIN(eventid) AS id
              FROM
                events
              WHERE eventid > eventsA.eventid AND eventsA.time < time)
*/

SelectQuery bquery
bquery:Filter("eventsA:eventid > eventid and eventsA:time < time")
bquery:MininumField("eventid")
bquery:RenameField("minimumEventid", "id")
bquery:SetTarget("events")
bquery:SetTableName("min")

Table event1
event1:SetName("events", "eventsA")
Table event2
event2:SetName("events", "eventsB")

Predicate p
p = p:FromSearch("eventsB:id IN :min")
p:ResolvePlaceholder(":min", bquery)

q:AddLeftJoin(event1, event2, p)
```

Code Sample 41: Self-join

6.4.6 Aggregate Conditions

As mentioned in subsection 6.4.4, `HAVING` clauses are also supported by leveraging search strings. A concrete example of this use can be found in code sample 43. The use of the search syntax for `HAVING` clauses allows the user to apply aggregation functions to queries in the same way they would use for `WHERE` clauses allowing to chain together conditions.

```
/*
        SELECT b.id AS id, age, yearofeducation, task
        FROM
          events
          LEFT JOIN
            (SELECT id, age, year AS yearofeducation
             FROM
               user
               INNER JOIN
               survey
               ON user.id = survey.id
             WHERE age > 18 AND year = "Sophomore")
             AS b
          ON
            events.id < b.id
*/

//Preparing subtable
SelectQuery joinedUserSurvey

joinedUserSurvey:AddIntegerField("id")
joinedUserSurvey:AddIntegerField("age")
joinedUserSurvey:AddTextField("year")
joinedUserSurvey:RenameField("year", "yearofeducation")
joinedUserSurvey:InnerJoin("user", "survey", "user:id = survey:id")
joinedUserSurvey:Filter("age > 18 and year = 'Sophomore'")
joinedUserSurvey:SetQueryName("b")

/* Creating main query */
SelectQuery q
q:AddIntegerField("b:id")
q:AddIntegerField("age")
q:AddTextField("yearofeducation")
q:AddIntegerField("task")

q:LeftJoin("events", joinedUserSurvey, "events:id < b:id")

Table t = db:Search(q)
```

Code Sample 42: Join with Nested Query

```
/*
        SELECT COUNT(id), group
        FROM
          events
        GROUP BY
          group
        HAVING
          AVG(time) < 3600 and COUNT(id) > 10
*/
q:AddCountField("id")
q:AddTextField("group")
q:SetTarget("events")
q:AddGroupBy("group")
q:AddHaving("Average(time) < 3600 and Count(id) > 10")
```

Code Sample 43: Having clause

6.4.7 Sorting Values

Another important aspect to querying values from databases is being able to sort the results based on the preferences of users. For this purpose, users can sort using the `AddSortHighestToLowest(text)` and `AddSortLowestToHighest(text)` methods provided by the library. During the piloting phase of the experiment, some participants stated to the proctor that the SQL choices of "ASC" and "DESC" were confusing and as such, the choice was made to spell out the direction in this way to be as clear as possible at the cost of making the method name more wordy.

```
/*
        SELECT *
        FROM
          events
        ORDER BY
          time DESC, id ASC
*/

q:SetTarget("events")
q:AddSortHighestToLowest("time")
q:AddSortLowestToHighest("id")
```

Code Sample 44: Sorting

6.4.8 Inserting, Updating, and Deleting Entries in a Database

While the previous sections have focused on the interactions possible with `SELECT` queries, it is important to remember that database interaction is not solely based on retrieving data from a database. Other interactions include inserting new entries, updating existing entries with new data, and deleting entries from a database table. Both inserting and updating values were tested in the experiment in chapter 4 and designs for these interactions are based on the techniques used in the hybrid group of the experiment and an attempt to keep the design consistent with the rest of the library. As can be seen in code sample 45, values in `INSERT` queries are added alongside their field names. In code sample 46, similar to the `INSERT` query, in `UPDATE` queries, updated values are added together with the names of their fields. Consistent with `WHERE` clauses in `SELECT` queries, the `WHERE` clause in `UPDATE` queries can be written using search strings. Since all of these types of queries do not return values, the `Run(query)` method is used to execute these queries.

```
/*
        INSERT INTO users
          (id, name, age )
        VALUES
          (66, "Sam Smith", 90)
*/
InsertQuery q
q:SetTarget("users")
q:AddValueInteger("id", 23)
q:AddValueText("name", "Sam Smith")
q:AddValueInteger("age", 90)

db:Run(q)
```

Code Sample 45: Inserting Entries

```
/*
        UPDATE users
        SET name = "Samuel Smith", age = 120
        WHERE name = "Sam Smith"
*/
UpdateQuery q
q:SetTarget("users")
q:AddUpdateText("name", "Samuel Smith")
q:AddUpdateInteger("age", "120")
q:Filter("name = 'Sam Smith'")

db:Run(q)
```

Code Sample 46: Updating Entries

Deleting entries from tables works using a combination of previously mentioned methods. The `DeleteQuery` (code sample 47) class requires the setting of a target using `SetTarget(text)`, naming a table, and optionally allows to set a filter using the `Filter(text)` method to limit the deletion of to specific entries. The `CreateQuery` class uses the target as the name of a new table, allowing users to add column names for new columns the same way `SELECT` queries project column names. This can be seen in code sample 48. Code sample 49 also shows the possibility to create tables from `SelectQuery` results by using the `From(SelectQuery)` method.

```
1  /*
2          DELETE FROM users
3          WHERE name = "Samuel Smith"
4  */
5  DeleteQuery q
6  q:SetTarget("users")
7  q:Filter("name = 'Samuel Smith'")
8
9  db:Run(q)
```

Code Sample 47: Deleting Entries

```
1  /*
2          CREATE TABLE users2 (
3              id int,
4              name varchar(255),
5              age int
6          )
7  */
8  CreateQuery q
9  q:SetTarget("users2")
10 q:AddIntegerField("id")
11 q:AddTextField("name")
12 q:AddIntegerField("age")
13
14 db:Run(q)
```

Code Sample 48: Creating Database Tables

```
1  /*
2          CREATE TABLE users3 AS
3          SELECT id, age
4          FROM users
5          WHERE id < 23
6  */
7  SelectQuery q
8  q:AddIntegerField("id")
9  q:AddIntegerField("age")
10 q:SetTarget("users")
11 q:Filter("id < 23")
12
13 CreateQuery r
14 r:SetTarget("users3")
15 r:From(q)
16
17 db:Run(r)
```

Code Sample 49: Creating Database Tables from queries

6.4.9 Data Without Databases

As stated in goal **G1**, the library should also be able to use data from other data sources, be that inserting data into a `Table` class by hand, or by reading structured plain text files such as comma separated value (CSV) files. Most of the functionality that was presented to be possible with the `Query` class is also possible to be used on the `Table` class. With the exception of `CreateQuery` objects, since their functionality is not needed on `Table` objects. When used with a table, the `Query` object would not be translated into a SQL string but the same information defining the queries would be used to affect the Table. For this purpose, the Table class has the methods `Search(Query):Table` and `Run(Query):Table`. While the search method acts the same as the database search method, returning a new table object after applying the query instructions, the run method, opposed to the run method of the `Database` class, also returns a new table object. It is desirable to create changed copies of tables instead of altering the tables by default to maintain data integrity. If the user wishes to overwrite the table, they can overwrite initial variable as can be seen in code sample 50. In this sample, data are read from a CSV file into the table, then a `SelectQuery` is defined and applied to the table. The return value from the `Search(Query)` method is then reassigned to table `t`.

```
1  Table t
2  t:ReadCSV("data.csv")
3
4  SelectQuery q
5  q:AddIntegerField("id")
6  q:Filter("ordernumber = 66")
7
8  t = t:Search(q)
```

Code Sample 50: Reading from CSV Files and Applying Queries to Tables

6.5 Design Artifacts

To give an overview of the classes making up the library, a class diagram was created. It can be seen in figure 6.1. Essential to the library is the the abstract class `Query` and its subclasses `InsertQuery`, `CreateQuery` and the abstract class `ConditionQuery` with its subclasses `DeleteQuery`, `UpdateQuery`, and `InsertQuery`. The `Query` class maintains a type field representing its specific kind of query for identification based on the constants of the `QueryType` class, as well as the query name and a list of fields. The `Field` class is a simple tuple to maintain the field name with a string representation of the type to make sure that the right operations are used during execution of the query. The representation of the type could also be managed differently with the use of enumerated constants or similar. The `Query` class also manages the query target, the representation of the data source the query is based on, such as a database table or another query. As `INSERT` and `CREATE` queries do not require conditions, their corresponding classes are not subclasses of the `ConditionQuery` class, but direct subclasses of the `Query` class. As `SELECT` queries can be sources to `CREATE` queries, the `CreateQuery` class has a method `From(SelectQuery)` which stores the `SelectQuery` object until the query is executed. The `CreateQuery` class is the only not accepted by table objects when

executing queries.

Since the `DeleteQuery` class does not need extra functionality, given that its functionality is captured in the code available through extending the `ConditionQuery` class, it only holds its query type. Since it also does not need the array of fields needed for other queries, it overrides the `GetFields()` method throwing an exception. The `UpdateQuery` uses an array of objects to contain the assigned values in the same order the fields were added. The `SelectQuery` allows for a wide variety of method calls to build select queries. Due to trying to retain relative readability of the class diagram, some methods have been omitted from the list of methods the `SelectQuery` would implement. Especially pertaining to the variety of join methods using different parameter combinations such as just table names and a string search, queries and a string search, both combinations but with a predicate search for using placeholders et cetera. Otherwise notable is that the class maintains a list of joins that have been added, as well as an additional field for the `HAVING` clause and a field for the `GROUP BY` clause.

The `Predicate` class represents the nodes of the binary tree structure generated from parsing the search strings to represent the conditions as objects. The structure can be traversed to perform the filtering by the `Table` class. The class maintains a `PredicateType` for each node to help interpret the information withing nodes such as the content of the values and the further traversal of the tree. The class is used mainly in the `ConditionQuery` and the `SelectQuery` class but can also substitute text based search strings in the `Table` class filter method.

The main data structure of the library, the `Table` class, contains a list of `Column`s that in turn contain the information within the tables columns. The `Table` class is either the result of a request to the database through the `Database` class, or a `Table` class can be instantiated by itself and filled with information manually, or by reading a CSV file. The `Table` class also offers the opportunity to use text-based or predicate-based searches on the content, or even allow for the use of complete `Query` objects to query the data inside the table. Searches and other queries on the database always return the results as a new table without altering the table. The `Database` class is a representation of a connection to a database, and after completing a connection, allows for the use of the search and run methods to query the database.

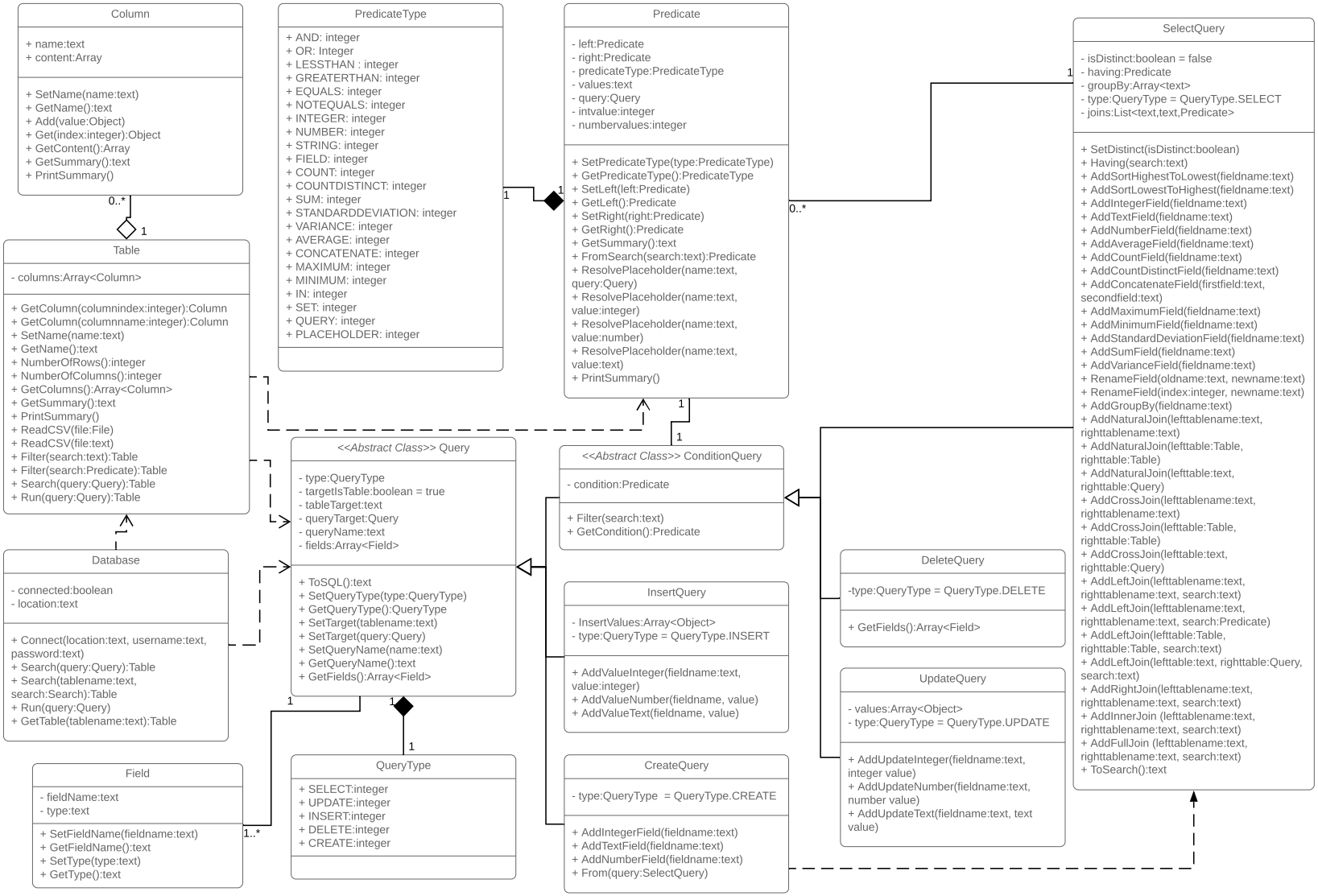

Figure 6.1: Class diagram of the Library

The grammar rules to parse the search strings can be seen in code sample 51. The grammar is expressed in ANTLR 4 [PHF14] syntax for ease of implementation. The main part of the grammar is the `search_expression` rule, which is similar to the normal `expression` syntax that exists in Quorum and shares many of the same rules. The `search_expression`, however, adds a couple of features necessary for database conditions and does not allow for the inclusion of variable names and function calls. It adds the ability to name columns in the format `ID(:ID)?` which allows naming simple column names (e.g. `id`) as well as column names based on tables or sub-queries (e.g. `users.id`). It also allows for the use of aggregation functions with predefined aggregation function names as used in the `function` rule and the comparison to NULL. The grammar was also switched to using single quotes for strings within the search as the search strings themselves will be defined using Quorum's typical double quote for strings. Further, it allows for the definition of placeholders for values added to the condition later. Placeholders take the form `:name`. Lastly, it adds the ability to use set notation for comparisons whether column values are in a set which comes with the keywords `in set` and comma separated lists of items in a set of the format `(1, 2, 3)`.

6.6 Alternative Designs

As mentioned in section 6.4.4, the alternative design of integrating the search syntax was considered as an option, hoping that this design might give meaningful error messages at compile time. After further deliberation, it quickly became clear that adding the search syntax to the language would involve significant changes to Quorum's type checking system and therefore significant investment into implementing the feature without knowing how strong the benefits of the direct integration into the language would really be.

In a similar vein, it was considered whether adding more extensive type safety to the system would be worthwhile as chapter 3 suggests that type information might help programmer productivity. Two systems were suggested, one in which programmers would add additional type information in the search expressions such as `integer id < 23`, in which case the annotations would provide readers of the code with additional information and the system could perform some additional type checking against what the programmer expected the type to be. The second solution involved writing classes representing the tables from the database, which could then be used for type checking for queries at compile time and make assigning column types for table columns easier. It was then determined that this would add significant overhead while programming, which might be too difficult for beginning programmers to perform for simple querying, while not being able to provide that types in the database would stay the same between compilation and execution of the program and not having evidence that this extra effort on the side of the user would add to the programming experience. Technical solutions to the development overhead were discussed, such as using a program that would generate code to represent the tables of the database, however it was decided that while this would make this approach more convenient, the development effort would be significant for a system of only po-

tential benefit. Validating this sort of system would likely require multiple additional studies to determine whether the approach would be useful.

6.7 Conclusion

This chapter described the design of a data management library based on the results of previous experiments. Decisions are based on scientific results where available. The design of the library attempts to strike a balance between keeping the ability to express SQL-like conditions concisely, while also reducing the need for programming language switching and its negative effects to achieve the goal of writing queries. It handles data using a table first approach, in which holding data in a table format is the default. To filter and query data in tables and databases, the library uses several query classes that allow users to use normal object-oriented programming features to define queries with the exception of introducing a small domain specific language based on the syntax of the host language to allow users to easily define expressions to use for conditions in database queries. Using a syntax similar to the host language should prevent users from getting confused about language syntax when using the DSL, reducing negative impacts of polyglot programming.

```
grammar SearchKeyword;

start  :
     search_expression EOF
  ;
search_expression :
    LEFT_PAREN search_expression RIGHT_PAREN
    |   INT
    |   BOOLEAN
    |   DECIMAL
    |   STRING
    |   NULL
    |   col=column_name
    |   placeholder=placeholder_rule
    |   MINUS search_expression
    |   NOT search_expression
    |   aggregationfunction = function
    |   search_expression IN search_expression
    |   comma_separated_set
    |   search_expression (MULTIPLY | DIVIDE |MODULO) search_expression
    |   search_expression (PLUS | MINUS) search_expression
    |   search_expression (GREATER | GREATER_EQUAL | LESS | LESS_EQUAL) search_expression
    |   search_expression (EQUALITY | NOTEQUALS) search_expression
    |   search_expression (AND) search_expression
    |   search_expression (OR) search_expression
    ;
column_name :
 ID (COLON ID)?
;
placeholder_rule:
    COLON ID
;
comma_separated_set:
    LEFT_PAREN (INT | DECIMAL)(COMMA (INT | DECIMAL))* RIGHT_PAREN
;
function :
   COUNT LEFT_PAREN col=column_name RIGHT_PAREN
 | COUNTDISTINCT LEFT_PAREN col=column_name RIGHT_PAREN
 | AVG LEFT_PAREN col=column_name RIGHT_PAREN
 | SUM LEFT_PAREN col=column_name RIGHT_PAREN
 | STDEV LEFT_PAREN col=column_name RIGHT_PAREN
 | VARIANCE LEFT_PAREN col=column_name RIGHT_PAREN
 | CONCATENATE LEFT_PAREN col=column_name COMMA col2=column_name RIGHT_PAREN
;
SEARCH        : 'search';
COUNT         : 'Count';
COUNTDISTINCT : 'CountDistinct';
AVG           : 'Average';
SUM           : 'Sum';
STDEV         : 'StandardDeviation';
VARIANCE      : 'Variance';
CONCATENATE   : 'Concatenate';
NULL          : 'undefined';
IN            : 'in set';
LEFT_PAREN    : '(';
RIGHT_PAREN   : ')';
INT           : '0'..'9'+;
DECIMAL       : '0'..'9'+ (PERIOD ('0'..'9')*)?;
GREATER       : '>';
GREATER_EQUAL : '>=';
LESS          : '<';
LESS_EQUAL    :  '<=';
EQUALITY      : '=';
PLUS          : '+';
MINUS         : '-';
MULTIPLY      : '*';
DIVIDE        : '/';
MODULO        : 'mod';
PERIOD        : '.';
COMMA         :  ',';
AND           :  'and';
OR            : 'or';
NOT           : 'not' | 'Not';
NOTEQUALS     : ('n' | 'N' ) 'ot=';
COLON         :  ':';
DOUBLE_QUOTE  :  '"';
SINGLE_QUOTE  :  '\'';
BOOLEAN       :  'true' | 'false';
STRING        : SINGLE_QUOTE .*? SINGLE_QUOTE;
ID            : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9' | '_')*;
NEWLINE       : '\r'?'\n' -> skip;
WS            : (' '|'\t'|'\n'|'\r')+ ->skip;
```

Code Sample 51: Search Grammar

Chapter 7

Conclusion

This chapter will summarize the contribution made by this dissertation by first summarizing the content of the chapters in section 7.1, then addressing the research questions posed in the introduction in section 7.2 and finally making recommendations for future research in section 7.3.

7.1 Summary

The introduction of this dissertation in chapter 1 introduced the reader to the topic of Polyglot programming and its pervasive nature in software development, citing that 97% of open source projects use more than one programming language [TT14]. The introduction also defines polyglot programming as the activity of using more than one programming language requiring switching between languages. Three different types of computer language switches are defined: Project level switches, file level switches, and embedded level switches. While project level switches concern switching between sets of languages when switching between programming projects, file level switches are switches between source code files written in different languages, and embedded level switches are switches between languages within one file, such as using domain specific languages embedded in the host language. Additionally, the contribution of this dissertation is introduced: Answering questions regarding whether polyglot practices, especially computer language switching, have an effect on programmer productivity using randomized controlled trials [con]. While the domain specific language literature suggests that using the right language for the job has positive impacts on software development [VDKV00], linguistics research suggests that there might be costs to switching between natural languages as a speaker [Ols16, Ols17], a finding that might be significant because similar brain regions are used for natural language understanding and code comprehension [SKA+14, SPP+17]. As an additional contribution, a database management library is designed based on the findings of the polyglot research.

The literature review in chapter 2 covers subjects such as the origin of polyglot programming and opinions about its use, computer science education, evidence on programming language productivity, research

on natural language code switching, and literature regarding database access.

In "On the Effect of Type Information on Developer Productivity" (chapter 3), the empirical literature on type systems, the programming language feature most researched with human factors methods, is reviewed in depth together with related existing theories. The aim is to gain a better understanding of why programmers seem to program more productively using statically typed languages. As a result, the chapter suggests the "Semantic Annotations Principle" stating that providing additional information in programming interfaces improves productivity, even if it increases verbosity of code. Additionally, the review of the existing literature suggests that the time it takes developers to comprehend code is an important determinant of productivity in programming. After establishing the principle, an experiment is described comparing two groups, one using enumerators and the other using constants to solve the same tasks. While both approaches are valid ways to solve the tasks and they are both similarly annotated, the results show that there is no significant difference between the groups due to the differences in the type mechanics, maintaining that improvements in comprehension from type annotations might be the main driver behind the productivity improvements from statically typed languages.

Chapter 4, "Randomized Controlled Trial on the Impact of Embedded Computer Language Switching", describes an experiment investigating the differences between different amounts of computer language switching on an embedded switching level: No switching, complete switching, and quick and shallow switching. For this purpose, the study is designed in a database programming context. Participants had to use an object-oriented solution to switching leveraging exclusively method calls to create queries, an approach where SQL was completely embedded as a domain specific language in Java using strings, or a hybrid approach where participant mostly used the host language to create queries, but conditions and field lists were defined using parts of the SQL language. The results show that participants with less experience did best with the hybrid approach, while more experienced developers achieved better times with the string-based approach. Across the board, the full object-oriented approach performed the worst. Overall, the quantitative and qualitative results suggest that less experienced developers were not impacted by switching as much. It seems plausible that they did not have a good understanding of the boundaries between languages because they were unfamiliar with both. Experienced developers, on the other hand, had experience with both languages and were comfortable with switching but were conversely delayed by the switching in a more unfamiliar language combination.

The experiment on file level computer language switching in chapter 5 investigates whether there is a measurable impact on productivity when developers have to switch between computer languages on a file level. The experiment has three groups: one group completely using JavaScript, another using PHP, and a third, polyglot, group switching between JavaScript and PHP. Participants in these groups had to solve

two programming tasks that required them to switch between source code files while programming. While the monoglot groups used the same language in both files, the polyglot group's tasks always switched between both languages. The results show a clear negative impact for having to switch between programming languages, while both monoglot groups had roughly equal task times. Further investigation shows that the negative impact is mostly caused by increased error rates in the polyglot group.

Finally, chapter 6 describes the design of a data management library based on the findings of the preceding chapters. As the design aims to be friendly for beginning programmers, the library is largely based on the design used for the hybrid group from the embedded language switching experiment, but attempts to improve on the design to make it more fully functional and better matched to the host language, Quorum, to avoid switching costs. To that end, a Quorum-like domain specific language is is proposed that allows for the definition of expressions to be used for conditions in queries. The chapter describes the possible use of the library using a number of example code snippets and provides a class diagram and a grammar for the integration of the new syntax into Quorum.

7.2 Research Questions

In this section, the three initial research questions will be addressed and new evidence provided towards answering the questions will be discussed.

7.2.1 RQ1: *"Is there a measurable productivity impact when programmers switch between computer languages?"*

While the results from the embedded level switching experiment are unclear, but might suggest that at least on the embedded level, experienced programmers might have a positive impact when they are using two familiar languages and less experienced programmers might do well with more rapid switching as long as the differences between languages are not clearly noticeable, the results of the file level switching experiment seem to clearly indicate that there is a negative impact to file level computer language switching. It appears that file level computer language switching leads to increased error rates, which may be the cause of productivity loss. While linguistic research suggested a switching time between programming language, none of the currently available results can confirm there being a concrete switching time, not further supporting the link between natural language understanding and computer language comprehension. However, the non-existence of results in the studies presented does not disprove the existence of a possible link.

To improve productivity, one could recommend to decrease the amount of programming language switching in a developers working time. This can be done by eliminating computer languages from projects or not introducing new ones when not absolutely necessary, or by structuring the work flow in a way that as much work as possible is performed in one language at a time before switching, reducing the switching impact. However, the results of the embedded language switching study suggest that the benefit of

7.2.2 RQ2: *"Do programmers consciously experience switches between computer languages?"*

The debriefing questions in both the embedded level switching experiment and the file level switching experiment suggest that not all programmers notice the switch consciously. In the embedded level switching experiment the minority of participants, mostly more experienced programmers, recognized the switch, while in the file switching experiment, approximately 68.89% of participants noted that there was a switch. Across both experiments, most participants stated that they did not think that switching languages impacted their performance, which might show that developers are not intuitively aware of the impact computer language switching has on their performance.

7.2.3 RQ3: *"Is there a difference in productivity between participants who speak English natively and those who do not?"*

Lastly, both studies tracked the participants' primary language in a survey question and a self-reported number rating their fluency if their primary language was not English. This information was used when evaluating the results of both experiments and in both a statistically significant result of lower performance was found. While this seems to be a clear indication for lower performance amongst non-native English speakers, closer scrutiny and consulting an expert revealed that the self-reported fluency number is a subpar indication of actual English fluency and that instead, a standardized test should be used, as well as a more detailed survey on natural language learning history for the participants. Other aspects which should also be taken into account, is whether participants code switch between their L1s and L2s, and whether English is an L2 or an L3. Investigating the effect for more experienced programmers failed to show a significant difference. There might be other hidden variables influencing the results as such the results of this dissertation regarding this research question might not be easily generalizable. It stands to reason that more research could create a clearer picture of the difference and might uncover the specific reasons for the effect. As it stands, the best explanation for the differences is non-primary English speakers taking longer to read the English language instructions.

7.3 Future Research

With many questions with regard to polyglot programming remaining unanswered and needing further research, this dissertation should be seen as more of a starting point for future research into the topic. For one, all experiments in this dissertation need to be replicated with more diverse populations, as they were mostly performed with U.S. participants. For another, the results of the embedded computer language switching experiment are not showing clear cut results regarding the impact of embedded language switching. Maybe, a redesign of the experiment, focusing on just two groups in a different context could lead to more clear results with respect to the impact of embedded level switches.

While the results of the file level switch experiment appear clear, the experiment only tested two specific languages that were relatively close in syntax and semantics. A more thorough exploration into the effects of different language combinations would add to the depth of understanding the observed effect. Questions such as whether the effect would be larger when two languages of different paradigms are used or whether the effect changes in size when the difference between syntaxes between languages becomes larger. And further, a more detailed measurement of switching time between languages might reveal whether there is a concrete mental switching time rather than just an increase in errors.

Diving deeper into the differences or similarities between what natural language theory suggests and what can be observed in computer language switching would be fascinating. Linguistic research suggests that people switching between languages might experience less impact when they have experience switching or when they are currently in a switching mindset [Ols17]. Study of these factors might create a rich understanding of what one could do to counteract negative impacts.

Looking at the debriefing comments some of the participants left when answering whether they noticed a switch between languages, it appears as if more inexperienced developers might have been less likely to realize they were switching than more experienced developers. Based on this observation, an investigation into whether inexperienced programmers have a less clear understanding of syntax restrictions and differences might give a better understanding how program comprehension develops through programming experience.

While the design of the library is evidence-based, there are a myriad of decisions to make when designing a library that have not been empirically tested yet and there is a chance that the amount of options is so vast that the evidence gathering process will never be completed. The library could be empirically validated against a similar library to evaluate whether the sum of design decisions did actually make for a better data management library than currently available designs. Further, doing a focused naming study on the library to improve the choice of names for functions and classes might improve the usability.

Lastly, the described studies focus on task completion time and error rates, while repeated switching could also impact the mental state of developers. Studies investigating whether switching languages might impact mood or tiredness of programmers might give a better indication of what the real effects of computer language switching might be.

Appendix A

Data Analysis of Pilot Runs for Chapter 4

A.0.1 Pilot 1

These are the results from the first pilot, conducted in December 2017. The pilot was similar to the current version of the experiment but after its conclusion, some adjustments had to be made, making the results not directly comparable. However, to give an overview of the direction of the results, this subsection will present the pilot data.

Recruitment

We recruited 9 participants for the study. Of the 9 participants, 2 identified themselves as female and 7 as male. On average, the participants were about 25 years old (M = 25.11, SD = 4.46). All 9 participants were Seniors in computer science. Three of the participants reported that English is not their primary language. The string-based group had 2 participants, the object-oriented group had 4, and the hybrid group had 3 participants. The demographics can also be found in table A.1.

Table A.1: Demographics

Metric	String-Based	Object-Oriented	Hybrid
N	2	4	3
Database Experience	0.00%	75%	66.66%
Female	0.00%	50.00%	0.00%
Age	21.5 (SD = 0.71)	24.25 (SD = 3.20)	28.67 (SD = 5.51)
Native	100.00%	75.00%	33.33%

Table A.2: Times per task in seconds

	Object-Oriented			String-based			Hybrid			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 1	4	1371.67	545.68	2	648.57	346.72	3	1356.33	756.73	1418.89	605.28
Task 2	4	676.33	143.39	2	209.29	82.00	3	783.00	764.44	792.56	651.66
Task 3	4	1006.67	533.35	2	295.29	112.10	3	2101.67	0.58	1699.22	654.75
Task 4	4	438.67	220.18	2	272.00	95.54	3	324.00	250.51	485.11	621.33
Task 5	4	154.33	39.95	2	510.86	166.97	3	156.00	57.94	408.78	639.54
Task 6	4	663.00	412.94	2	510.86	166.97	3	372.67	186.27	725.67	787.17

Baseline Data

All 9 of the participants finished all tasks, however not all were successfully completed. The data (all times in seconds) can be found in table form in table A.2. The average time for task 3 for all participants was the highest of all tasks (M = 1699.22, SD = 654.75), followed by task 1 (M = 1418.89, SD = 605.28), task 2 (M = 792.56, SD = 651.66), and task 6 (M = 725.67, SD = 787.17). The task with the lowest average time was task 4 (M = 408.78, SD = 639.54) and the second lowest was task 4 (M = 485.11, SD = 621.33). Overall the average task time for the object oriented group (M = 1077.25, SD = 818.36) was higher than the average task time for the hybrid group (M = 848.94, SD = 802.82). The string-based group had the lowest average task time (M = 719.75, SD = 725.82).

Figure A.1 shows the average task times between the three groups. Figure A.2 shows the differences in task times broken down based on whether participants answered that they had previous database experience. Figure A.3 shows the task times by group in a boxplot.

Analysis

To analyze the results, we ran a mixed designs repeated measures ANOVA using the R programming language with respect to time to solution, using task as a within-subjects variable and group as between-subjects variable. Sphericity was tested using Mauchly's test for sphericity, which shows that the assumption of sphericity was violated for the variable task. Following reported numbers are reported with Greenhouse-Geisser corrections taken into account.

There are significant effects at $p < 0.5$ for the within-subjects variable task, $F(5, 30) = 16.229,\ p < 0.001$ ($\eta_p^2 = 0.462$), but no significant effect for group, $F(2, 6) = 0.328,\ p = 0.732$ ($\eta_p^2 = 0.069$). The interaction effect of group and task was significant, $F(10, 30) = 2.597,\ p = 0.021$ ($\eta_p^2 = 0.216$).

To test the differences between the tasks, we ran a Bonferroni test. As can be seen in table A.3, there are significant differences between task 1 and 2 (p = 0.046), 1 and 4 (p = 0.021), 1 and 5 (p = 0.031), 3 and 4 (p = 0.022), and 3 and 5 (p = 0.018).

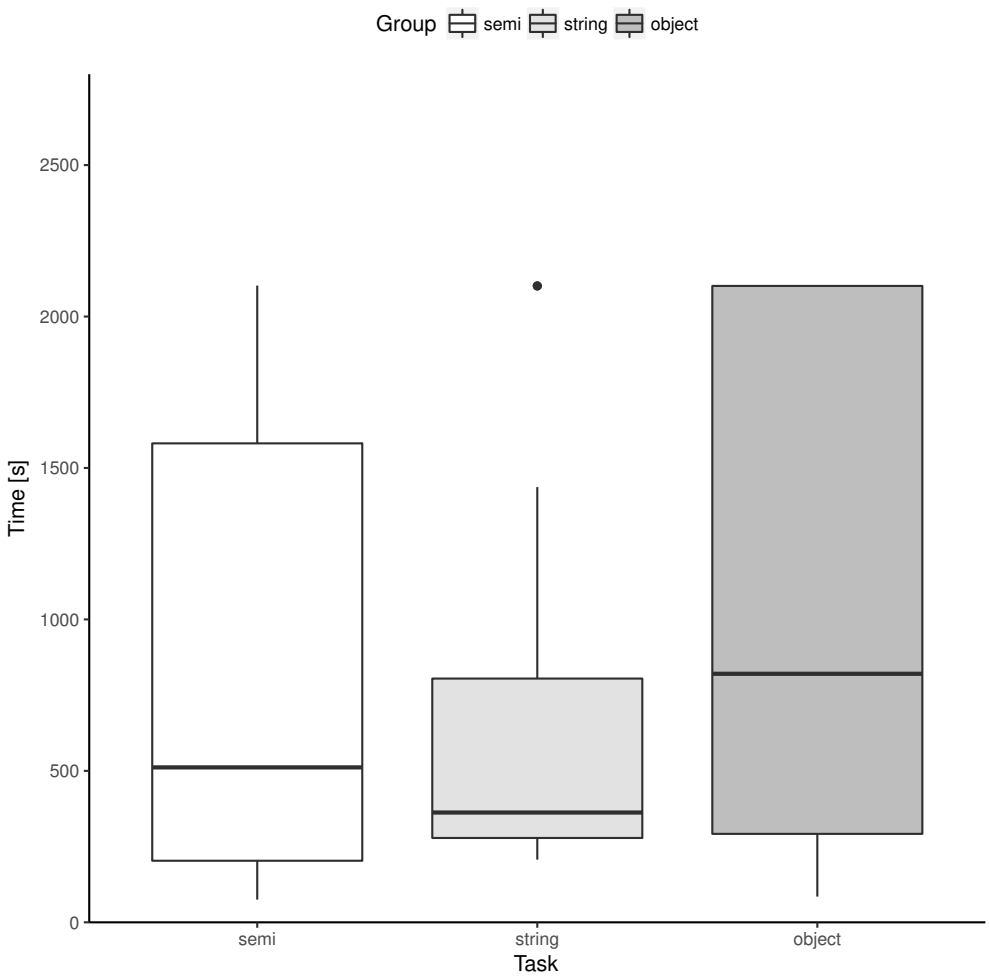

Figure A.1: Boxplot of results between the groups

Table A.3: Bonferroni test of task times

	1	2	3	4	5
2	0.046	-	-	-	-
3	1.000	0.207	-	-	-
4	0.021	0.465	0.022	-	-
5	0.031	0.619	0.018	1.000	-
6	0.452	1.000	0.253	1.000	1.000

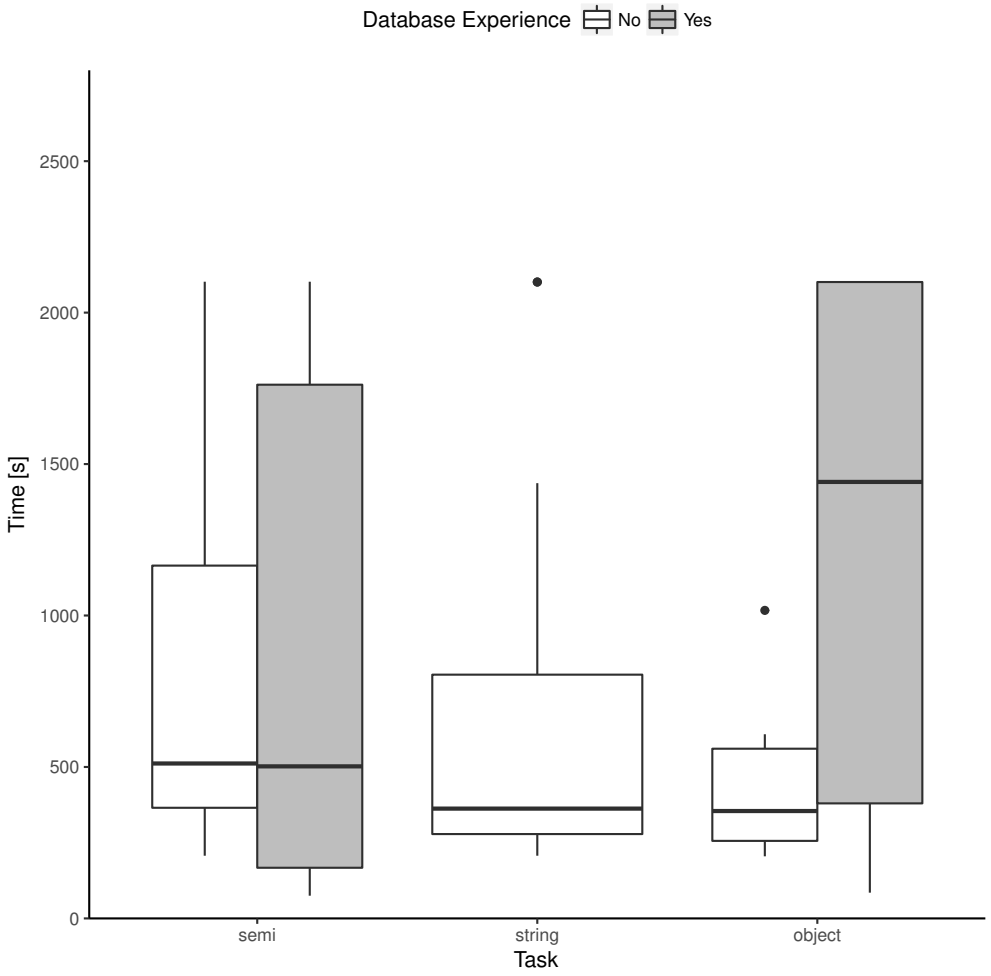

Figure A.2: Boxplot of differences between participants with and without database experience

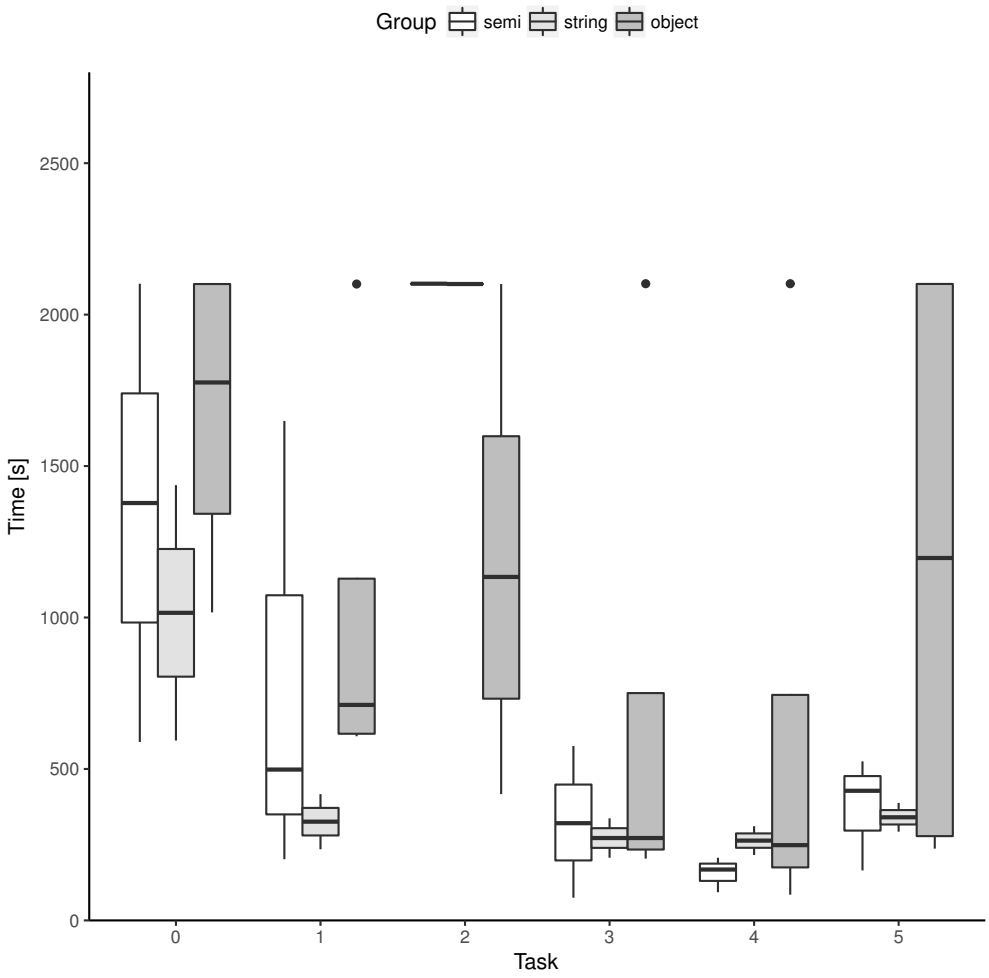

Figure A.3: Boxplot of differences in time by task

Table A.4: Demographics.

Metric	String-Based	Object-Oriented	Hybrid
N	4	3	4
Database Experience	25.00%	33.33%	25.00%
Female	50.00%	66.66%	0.00%
Age	23.50 (SD = 5.74)	20.33 (SD = 1.53)	26.00 (SD = 10.74)
Native	75.00%	66.66%	100.00%

A.0.2 Pilot 2

This section presents the data recorded in a second pilot in summer and fall 2018. This run of the experiment ended up being canceled as the wrong version of the experimental software was in use.

Recruitment

We recruited 11 participants for the pilot study. Of the 11 participants, 5 identified themselves as female and 6 as male. On average, the participants were about 24 years old (M = 23.55, SD = 7.10). Six of the participants were sophomores, two were juniors, two were seniors, and one was a graduate student. Four of the participants reported that English is not their primary language. The string-based group had 4 participants, the object-oriented group had 3, and the hybrid group had 4. The demographics can also be found in table A.4.

Baseline Data

All 11 of the participants completed all tasks. The data (all times in seconds) can be found in table form in table A.5. The average time for task 3 for all participants was the highest of all tasks (M = 1450.45, SD = 722.35), followed by task 1 (M = 1239.45, SD = 614.05), task 2 (M = 777.36, SD = 416.42), and task 6 (M = 589.64, SD = 367.10). The task with the lowest average time was task 4 (M = 347.09, SD = 184.75) and the second lowest was task 5 (M = 391.64, SD = 578.24). Overall the average task time for the string-based group (M = 965.04, SD = 751.49) was the highest, followed by the object-oriented group (M = 718.44, SD = 507.60) and the hybrid group (M = 694.12, SD = 615.21).

Figure A.4 shows the average tasks times between the two groups. Figure A.5 shows the differences in task times broken down based on whether participants answered that they had previous database experience. Figure A.6 shows the task times by group in a boxplot.

Analysis

To analyze the results, we ran a mixed designs repeated measures ANOVA using the R programming language with respect to time to solution, using task as a within-subjects variable and group and database

Table A.5: Times per task in seconds.

	Object-Oriented			String-based			Hybrid			Total	
Task	N	mean	SD	N	mean	SD	N	mean	SD	mean	SD
Task 1	3	1371.67	545.68	4	1380.50	700.38	4	999.25	668.48	1239.45	614.05
Task 2	3	676.33	143.39	4	782.00	641.35	4	848.50	368.77	777.36	416.42
Task 3	3	1006.67	533.35	4	1870.25	419.55	4	1363.50	965.22	1450.45	722.35
Task 4	3	438.67	220.18	4	268.00	134.39	4	357.50	215.69	347.09	184.75
Task 5	3	154.33	39.95	4	702.50	945.13	4	258.75	107.21	391.64	578.24
Task 6	3	663.00	412.94	4	787.00	437.08	4	337.25	47.33	589.64	367.10

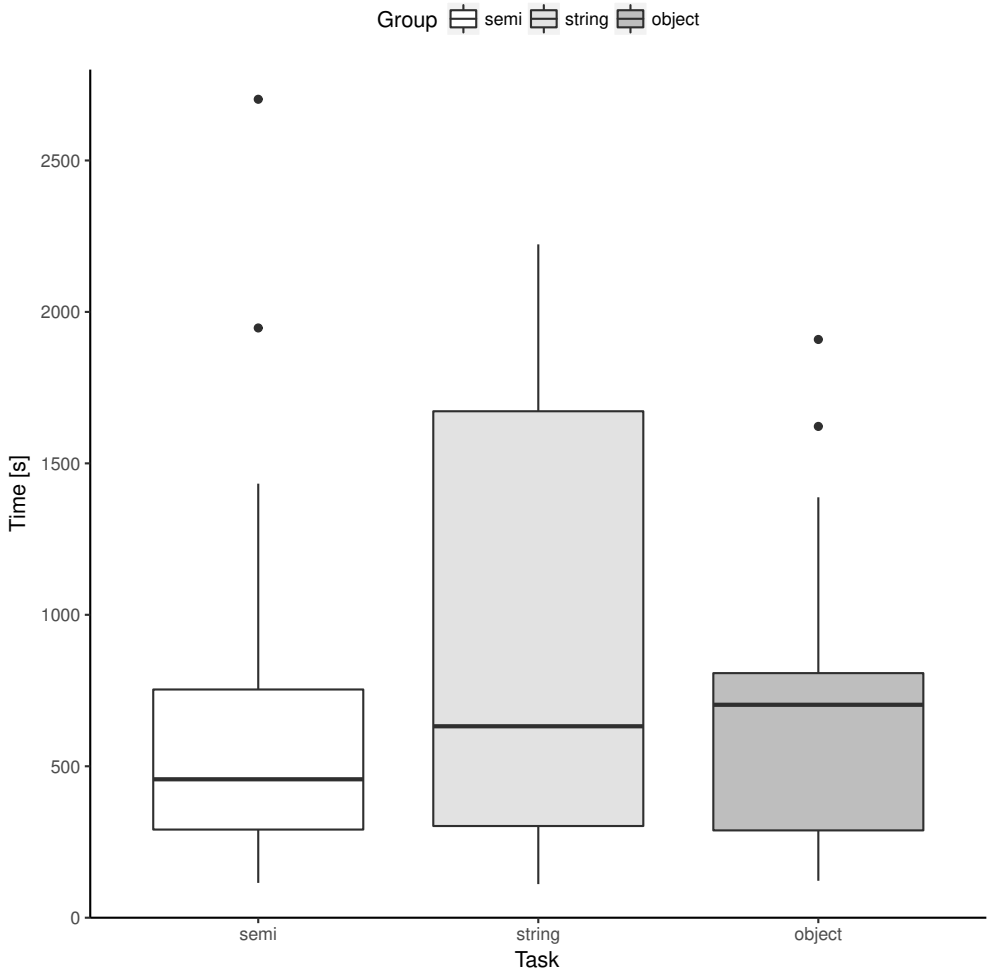

Figure A.4: Boxplot of results between the groups.

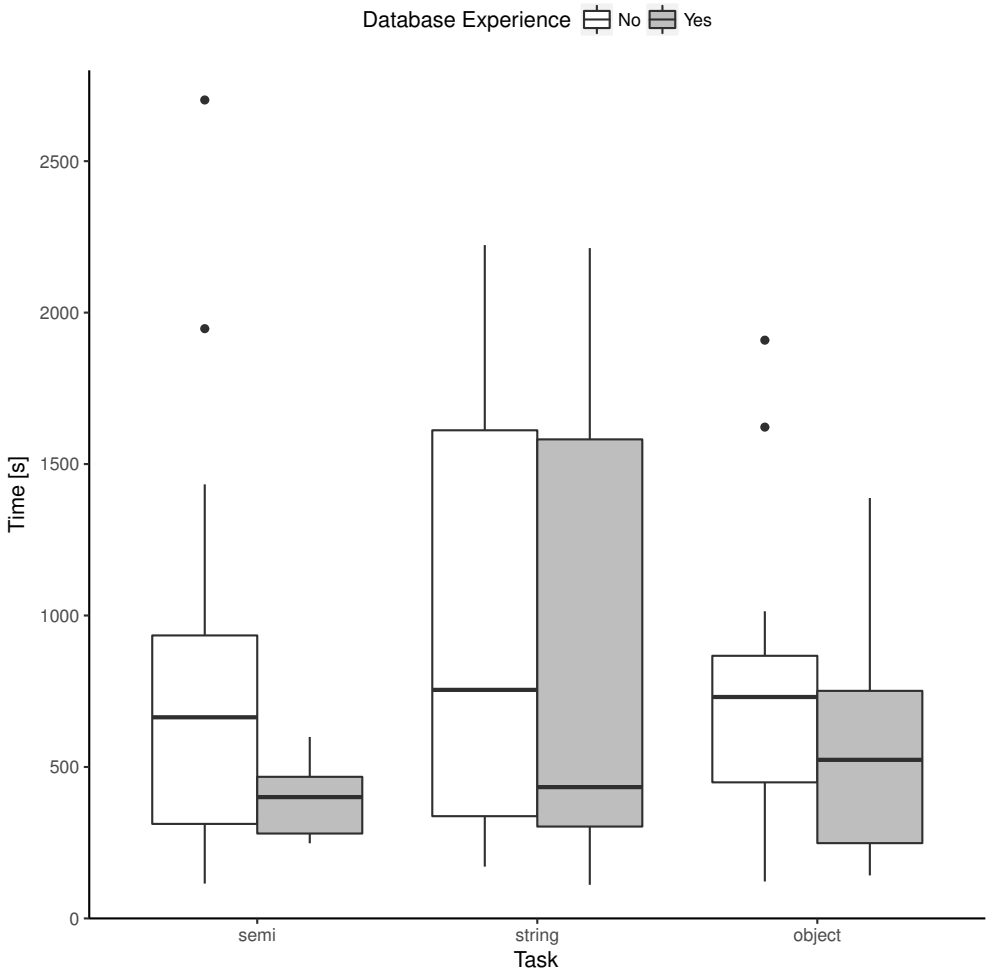

Figure A.5: Boxplot of differences between participants with and without database experience.

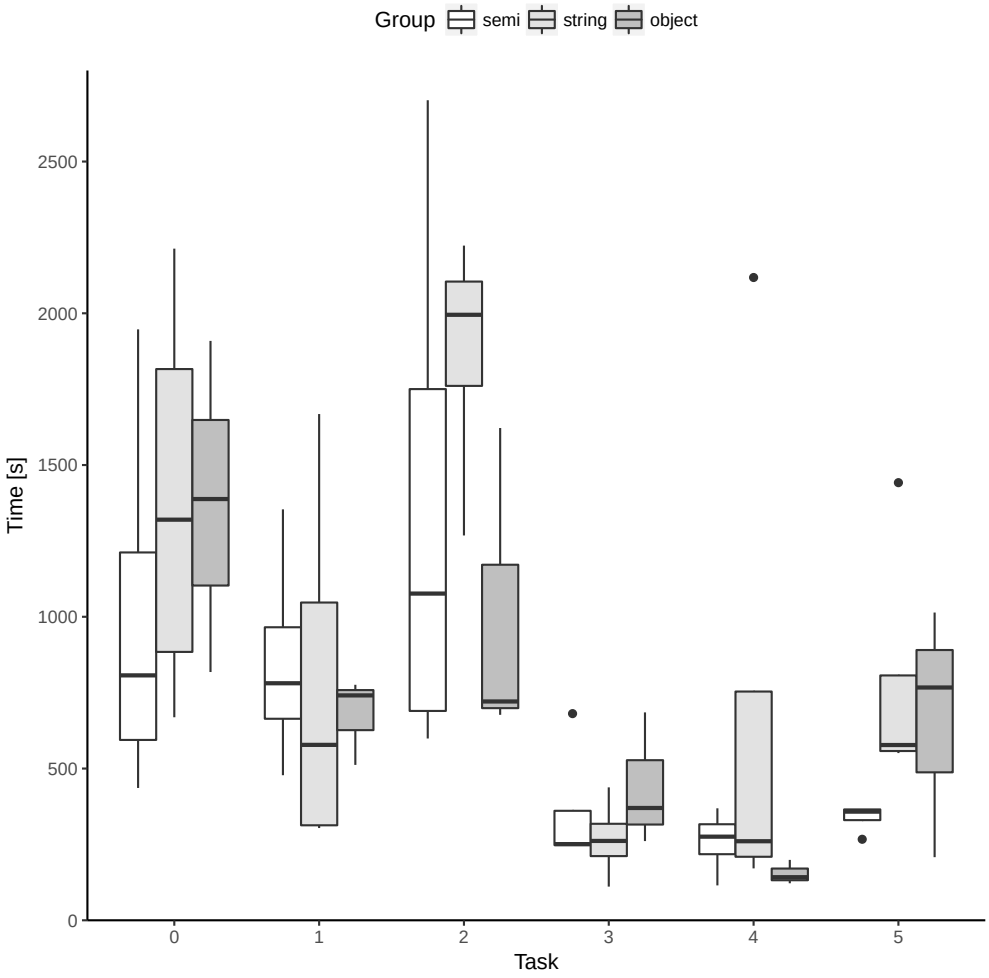

Figure A.6: Boxplot of differences in time by task.

Table A.6: Bonferroni test of task times.

	1	2	3	4	5
2	1.000	-	-	-	-
3	1.000	0.476	-	-	-
4	0.011	0.152	0.009	-	-
5	0.041	1.000	0.008	1.000	-
6	0.108	1.000	0.035	0.960	1.000

experience as between-subjects variables. Sphericity was tested using Mauchly's test for Sphericity, which shows that the assumption of sphericity was violated for the variable task. Following reported numbers are reported with Greenhouse-Geisser corrections taken into account.

There is a significant effect at $p < 0.5$ for the within-subjects variable task, $F(5, 25) = 7.065,\ p < 0.001$ ($\eta_p^2 = 0.479$), but no significant effect for group, $F(2, 5) = 0.889,\ p = 0.467$ ($\eta_p^2 = 0.110$) or database experience, $F(1, 5) = 0.973,\ p = 0.369$ ($\eta_p^2 = 0.064$). None of the interaction effects are significant.

To test the differences between the tasks, we ran a Bonferroni test. As can be seen in table A.3, there are significant differences between task 1 and 4 (p = 0.011), 1 and 5 (p = 0.041), 3 and 4 (p = 0.009), 3 and 5 (p = 0.008), and 3 and 6 (p = 0.035).

Appendix B

Code Samples For Chapter 4

This appendix lists additional code samples belonging to the experiment described in chapter 4.

```
1       /**
2        * Please write this method to return a Table object containing the columns
3        * id, year, and grade for all entries with lastname "Schmidt" who do not
4        * have a grade "F"
5        *
6        * Table information:
7        *
8        * - student -
9        *
10       * id (int) | firstname (String) | lastname (String) | grade (String) | year
11       * (int) | class (String)
12       *
13       * Use the technique shown to you in the samples given
14       *
15       */
```

Code Sample 52: Task 2 task description

```
/**
 * Please write this method to return a Table object containing the columns
 * songname, artist, and timesplayed for all entries which have a rating above 3,
 * were written after the year 2009, and belong to the genre pop. Also include
 * all songs by the artist "Dude". The result should be sorted from high timesplayed
 * to low timesplayed.
 *
 *
 * Table information:
 *
 * - charts -
 *
 * id (int) | songname (String) | artist (String) | rating
 * (int) | year (int) | genre (string) | timesplayed (int)
 *
 * Use the technique shown to you in the samples given
 *
 */
```

Code Sample 53: Task 3 task description

```
/**
 * Please write this method to return the Table object student with an entry
 * changed. The entry for the student with the first name "Herbert" and the
 * last name "Hauser" should be changed so that the grade is now "A".
 *
 * Table information:
 *
 * - students -
 *
 * id (int) | firstname (String) | lastname (String) | grade
 * (String) | year (int)
 *
 * Use the technique shown to you in the samples given
 *
 */
```

Code Sample 54: Task 4 task description

```
/**
 * Please write this method to return the Table object student with an entry
 * added. The new entry should have the id "23", the first name "Tom", the
 * last name "Young", the year "3", and the grade "C"
 *
 * Table information:
 *
 * - students -
 *
 * id (int) | firstname (String) | lastname (String) | year
 * (int) | grade (String)
 *
 * Use the technique shown to you in the samples given
 *
 */
```

Code Sample 55: Task 5 task description

```
/**
 * Please write this method to return a Table object containing the columns
 * id, firstname, lastname, clubname of all students.
 * You will have to use data from clubmap to complete these requirements.
 *
 * Table information:
 *
 * - students -
 *
 * id (int) | firstname (String) | lastname (String) | grade (String) | year
 * (int) | birthyear (int)
 *
 * - clubmap -
 *
 * cid (int) | studentid (int) | clubname (String)
 *
 *
 * Use the technique shown to you in the samples given
 *
 */
```

Code Sample 56: Task 6 task description

```
public Table queryB(Table students) throws Exception {
    Query q = new Query();

    q.Prepare("SELECT id, year, grade "+
        "FROM students " +
        "WHERE lastname = 'Schmidt' and grade != 'F'");

    Table r = students.Search(q);
```

Code Sample 57: Task 2 solution for group SQL

```
}

public Table queryC(Table students) throws Exception {
    Query q = new Query();

    q.AddField("id")
        .AddField("year")
        .AddField("grade");

    q.Filter(q.Where("lastname").Equals("Schmidt").And("grade").NotEquals("F"));

    Table r = students.Search(q);
```

Code Sample 58: Task 2 solution for the object-oriented group

```
    }

    public Table queryA(Table students) throws Exception {
        Query q = new Query();

        q.AddFields("id, year, grade");

        q.Filter("lastname = 'Schmidt' and grade != 'F'");

        Table r = students.Search(q);
```

Code Sample 59: Task 2 solution for the hybrid group

```
    public Table queryB(Table charts) throws Exception {
        Query q = new Query();

        q.Prepare("SELECT songname, artist, timesplayed "+
            "FROM charts "+
            "WHERE rating > 3 AND year > 2009 "+
                "AND genre = 'pop' OR artist = 'Dude' "+
            "ORDER BY timesplayed DESC");
```

Code Sample 60: Task 3 solution for group SQL

```
        Table r = charts.Search(q);

        return r;
    }

    public Table queryC(Table charts) throws Exception {
        Query q = new Query();

        q.AddField("songname")
            .AddField("artist")
            .AddField("timesplayed");

        q.Filter(q.Where("rating").GreaterThan(3)
            .And("year").GreaterThan(2009)
```

Code Sample 61: Task 3 solution for the object-oriented group

```

        return r;
    }

    public Table queryA(Table charts) throws Exception {
        Query q = new Query();

        q.AddFields("songname, artist, timesplayed");

        q.Filter("rating > 3 and year > 2009 "+
            " and genre = 'pop' or artist = 'Dude'");
        q.SortHighToLow("timesplayed");
```

Code Sample 62: Task 3 solution for the hybrid group

```
1       public Table queryB(Table students) throws Exception {
2           Query q = new Query();
3
4           q.Prepare("UPDATE students SET grade = 'A' "+
5               "WHERE firstname = 'Herbert' AND lastname = 'Hauser'");
6
7           Table r = students.Search(q);
8
9           return r;
```

Code Sample 63: Task 4 solution for group SQL

```
1
2       public Table queryC(Table students) throws Exception {
3           Query q = new Query();
4
5           q.Replace("grade", "A");
6           q.Filter(q.Where("firstname").Equals("Herbert")
7               .And("lastname").Equals("Hauser"));
8
9           Table r = students.Update(q);
```

Code Sample 64: Task 4 solution for the object-oriented group

```
1
2       public Table queryC(Table students) throws Exception {
3           Query q = new Query();
4
5           q.Replace("grade", "A");
6           q.Filter(q.Where("firstname").Equals("Herbert")
7               .And("lastname").Equals("Hauser"));
8
9           Table r = students.Update(q);
```

Code Sample 65: Task 4 solution for the hybrid group

```
1       public Table queryB(Table students) throws Exception {
2           Query q = new Query();
3
4           q.Prepare("INSERT INTO students (id, firstname, lastname, year, grade) "+
5               "VALUES ('23', 'Tom', 'Young', 3, C)");
6
7           Table r = students.Search(q);
8
9           return r;
```

Code Sample 66: Task 5 solution for group SQL

```
1
2       public Table queryC(Table students) throws Exception {
3           Query q = new Query();
4
5           q.AddValue("id", 23)
6               .AddValue("firstname", "Tom")
7               .AddValue("lastname", "Young")
8               .AddValue("year", 3)
9               .AddValue("grade", "C");
10
11          Table r = students.Insert(q);
12
13          return r;
```

Code Sample 67: Task 5 solution for the object-oriented group

```
1
2       public Table queryA(Table students) throws Exception {
3           Query q = new Query();
4
5           q.IntoFields("id, firstname, lastname, year, grade");
6           q.SetValues("23, 'Tom', 'Young', 3, 'C'");
7
8           Table r = students.Insert(q);
9
10          return r;
```

Code Sample 68: Task 5 solution for the hybrid group

```
1       public Table query(Table students, Table clubmap) throws Exception {
2           Query q = new Query();
3
4           q.Prepare("SELECT id, firstname, lastname, clubname "+
5               "FROM students JOIN clubmap ON students.id = clubmap.studentid");
6
7           Table r = students.Search(q, clubmap);
8
9           return r;
```

Code Sample 69: Task 6 solution for group SQL

```
1
2       public Table queryC(Table students, Table clubmap) throws Exception {
3           Query q = new Query();
4
5           q.AddField("id")
6               .AddField("firstname")
7               .AddField("lastname")
8               .AddField("clubname");
9
10          q.Combine(students, "id", clubmap, "studentid");
11
12          Table r = students.Search(q);
13
14          return r;
```

Code Sample 70: Task 6 solution for the object-oriented group

```
1
2       public Table queryA(Table students, Table clubmap) throws Exception {
3           Query q = new Query();
4
5           q.AddFields("id, firstname, lastname, clubname");
6
7           q.Combine(students, "id", clubmap, "studentid");
8
9           Table r = students.Search(q);
10
11          return r;
```

Code Sample 71: Task 6 solution for the hybrid group

```
1  package sample;
2
3  import library.*;
4
5  public class SampleB {
6
7      /**
8       * Results of print commands are in the comments and
9       * formatted for better readability.
10      *
11      * The format of the tables in this sample:
12      *
13      * - cars -
14      *
15      * id (int) | make (String) | model (String) | year (int) |
16      * licenseplatenumber (String) | registrationyear (int)
17      *
18      * - ticketlist -
19      *
20      * tid (int) | carid (int) | tickettype (String) | description (String)
21      *
22      *
23      */
24     public void sample(Table cars, Table ticketlist) throws Exception {
25
26         cars.outputTable();
27         /*
28             Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
29             Renault,  Laguna,  322 SKO,  80,  2014,  2015,
30             Mercedes,  C250,  660 OLK,  56,  2015,  2016,
31             Ford,  Fusion,  910 GRE,  12,  2017,  2017,
32          */
33
34         ticketlist.outputTable();
35         /*
36             Parking,  3,  41,  no time on parking meter,
37             Parking,  5,  56,  student in staff space,
38             Speeding,  8,  56,  20mph too fast in 30mph zone,
39             Parking,  10,  41,  double parked,
40             Speeding,  11,  56,  10mph too fast in 65mph zone,
41             Speeding,  23,  41,  15mph too fast in 45mph zone,
42             Speeding,  43,  12,  drove faster than police car,
43             Parking,  44,  41,  parked in street,
44             Red light,  51,  56,  Ran red light 10 sec after red,
45             Speeding,  56,  12,  10mph too fast in 35mph construction zone,
46          */
```

Code Sample 72: Sample of the string-based group

```
Query allCarInfoQuery = new Query();

allCarInfoQuery.Prepare("SELECT * FROM cars");

Table allCarInfo = cars.Search(allCarInfoQuery);

allCarInfo.outputTable();
/*
    Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
    Renault,  Laguna,  322 SKO,  80,  2014,  2015,
    Mercedes,  C250,  660 OLK,  56,  2015,  2016,
    Ford,  Fusion,  910 GRE,  12,  2017,  2017,
 */


Query insertCarsQuery = new Query();

insertCarsQuery.Prepare(
        "INSERT INTO cars "
        + "(id, make, model, year, licenseplatenumber, registrationyear) "
        + "VALUES (2345, 'Toyota', 'Verso', 2016, '326 NEB', 2017)");

Table carsInserted = cars.Search(insertCarsQuery);

carsInserted.outputTable();
/*
    Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
    Renault,  Laguna,  322 SKO,  80,  2014,  2015,
    Mercedes,  C250,  660 OLK,  56,  2015,  2016,
    Ford,  Fusion,  910 GRE,  12,  2017,  2017,
    Toyota,  Verso,  326NEB,  2345,  2016,  2017,
 */

Query carsNotMercedesQuery = new Query();

carsNotMercedesQuery.Prepare("SELECT make, model, year"
        + "FROM cars WHERE make != 'Mercedes' ORDER BY year DESC");

Table carsNotMercedes = cars.Search(carsNotMercedesQuery);

carsNotMercedes.outputTable();
/*
    Ford,  Fusion,  2017,
    Renault,  Laguna,  2014,
    Citroen,  Berlingo,  2003,
 */
```

Code Sample 73: Sample of the string-based group (cont.)

```
        Query carTicketQuery = new Query();

        carTicketQuery.Prepare("SELECT id, make, model, description "
                + "FROM cars JOIN ticketlist ON cars.id = ticketlist.carid");

        Table ticketsForCars = cars.Search(carTicketQuery, ticketlist);

        ticketsForCars.outputTable();
        /*
            41,  Citroen,  Berlingo,  no time on parking meter,
            41,  Citroen,  Berlingo,  double parked,
            41,  Citroen,  Berlingo,  15mph too fast in 45mph zone,
            41,  Citroen,  Berlingo,  parked in street,
            56,  Mercedes,  C250,  student in staff space,
            56,  Mercedes,  C250,  20mph too fast in 30mph zone,
            56,  Mercedes,  C250,  10mph too fast in 65mph zone,
            56,  Mercedes,  C250,  Ran red light 10 sec after red,
            12,  Ford,  Fusion,  drove faster than police car,
            12,  Ford,  Fusion,  10mph too fast in 35mph construction zone,
        */

        Query carsOlderQuery = new Query();

        carsOlderQuery.Prepare("SELECT make, model, year, licenseplatenumber, registrationyear "
                + "FROM cars WHERE year < 2015 ORDER BY year ASC");

        Table carsOlderThan2015 = cars.Search(carsOlderQuery);

        carsOlderThan2015.outputTable();
        /*
            Citroen,  Berlingo,  2003,  255 FAD,  2005,
            Renault,  Laguna,  2014,  322 SKO,  2015,
         */


        Query updateQuery = new Query();

        updateQuery.Prepare("UPDATE cars SET licenseplatenumber = 'none' "
                + "WHERE make = 'Mercedes' and model = 'CS250'");

        Table updatedYear = cars.Search(updateQuery);

        updatedYear.outputTable();
        /*
            Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
            Renault,  Laguna,  322 SKO,  80,  2014,  2015,
            Mercedes,  C250,  none,  56,  2015,  2016,
            Ford,  Fusion,  910 GRE,  12,  2017,  2017,
         */
    }

}
```

Code Sample 74: Sample of the string-based group (cont. 2)

```
1  package sample;
2  import library.*;
3
4  public class SampleC {
5
6      /**
7       * Results of print commands are in the comments and
8       * formatted for better readability.
9       *
10      * The format of the tables in this sample:
11      *
12      * - cars -
13      *
14      * id (int) | make (String) | model (String) | year (int) |
15      * licenseplatenumber (String) | registrationyear (int)
16      *
17      * - ticketlist -
18      *
19      * tid (int) | carid (int) | tickettype (String) | description (String)
20      *
21      *
22      */
23 ,    public void sample(Table cars, Table ticketlist) throws Exception {
24
25          cars.outputTable();
26          /*
27              Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
28              Renault,  Laguna,  322 SKO,  80,  2014,  2015,
29              Mercedes,  C250,  660 OLK,  56,  2015,  2016,
30              Ford,  Fusion,  910 GRE,  12,  2017,  2017,
31           */
32
33          ticketlist.outputTable();
34          /*
35              Parking,  3,  41,  no time on parking meter,
36              Parking,  5,  56,  student in staff space,
37              Speeding,  8,  56,  20mph too fast in 30mph zone,
38              Parking,  10,  41,  double parked,
39              Speeding,  11,  56,  10mph too fast in 65mph zone,
40              Speeding,  23,  41,  15mph too fast in 45mph zone,
41              Speeding,  43,  12,  drove faster than police car,
42              Parking,  44,  41,  parked in street,
43              Red light,  51,  56,  Ran red light 10 sec after red,
44              Speeding,  56,  12,  10mph too fast in 35mph construction zone,
45           */
```

Code Sample 75: Sample of the Object-Oriented Group

```
        Query allCarInfoQuery = new Query();

        Table allCarInfo = cars.Search(allCarInfoQuery);

        allCarInfo.outputTable();
        /*
            Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
            Renault,  Laguna,  322 SKO,  80,  2014,  2015,
            Mercedes,  C250,  660 OLK,  56,  2015,  2016,
            Ford,  Fusion,  910 GRE,  12,  2017,  2017,
         */


        Query insertCarsQuery = new Query();

        insertCarsQuery.AddValue("id", 2345);
        insertCarsQuery.AddValue("make", "Toyota");
        insertCarsQuery.AddValue("model", "Verso");
        insertCarsQuery.AddValue("year", 2016);
        insertCarsQuery.AddValue("licenseplatenumber", "326 NEB");
        insertCarsQuery.AddValue("registrationyear", 2017);

        Table carsInserted = cars.Insert(insertCarsQuery);

        carsInserted.outputTable();
,       /*
            Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
            Renault,  Laguna,  322 SKO,  80,  2014,  2015,
            Mercedes,  C250,  660 OLK,  56,  2015,  2016,
            Ford,  Fusion,  910 GRE,  12,  2017,  2017,
            Toyota,  Verso,  326NEB,  2345,  2016,  2017,
        */

        Query carsNotMercedesQuery = new Query();

        carsNotMercedesQuery.AddField("make");
        carsNotMercedesQuery.AddField("model");
        carsNotMercedesQuery.AddField("year");

        carsNotMercedesQuery.Filter(carsNotMercedesQuery.Where("make").NotEquals("Mercedes"));
        carsNotMercedesQuery.SortHighToLow("year");

        Table carsNotMercedes = cars.Search(carsNotMercedesQuery);

        carsNotMercedes.outputTable();
        /*
            Ford,  Fusion,  2017,
            Renault,  Laguna,  2014,
            Citroen,  Berlingo,  2003,
         */
```

Code Sample 76: Sample of the Object-Oriented Group (cont.)

```
1
2           Query carTicketQuery = new Query();
3
4           carTicketQuery.Combine(cars, "id", ticketlist, "carid");
5
6           carTicketQuery.AddField("id");
7           carTicketQuery.AddField("make");
8           carTicketQuery.AddField("model");
9           carTicketQuery.AddField("description");
10
11          Table ticketsForCars = cars.Search(carTicketQuery);
12
13          ticketsForCars.outputTable();
14          /*
15              41,  Citroen,  Berlingo,  no time on parking meter,
16              41,  Citroen,  Berlingo,  double parked,
17              41,  Citroen,  Berlingo,  15mph too fast in 45mph zone,
18              41,  Citroen,  Berlingo,  parked in street,
19              56,  Mercedes,  C250,  student in staff space,
20              56,  Mercedes,  C250,  20mph too fast in 30mph zone,
21              56,  Mercedes,  C250,  10mph too fast in 65mph zone,
22 ,            56,  Mercedes,  C250,  Ran red light 10 sec after red,
23              12,  Ford,  Fusion,  drove faster than police car,
24              12,  Ford,  Fusion,  10mph too fast in 35mph construction zone,
25           */
26
27          Query carsOlderQuery = new Query();
28
29          carsOlderQuery.AddField("make");
30          carsOlderQuery.AddField("model");
31          carsOlderQuery.AddField("year");
32          carsOlderQuery.AddField("licenseplatenumber");
33          carsOlderQuery.AddField("registrationyear");
34
35          carsOlderQuery.Filter(carsOlderQuery.Where("year").LessThan(2015));
36          carsOlderQuery.SortLowToHigh("year");
37
38          Table carsOlderThan2015 = cars.Search(carsOlderQuery);
39
40          carsOlderThan2015.outputTable();
41          /*
42              Citroen,  Berlingo,  2003,  255 FAD,  2005,
43              Renault,  Laguna,  2014,  322 SKO,  2015,
```

Code Sample 77: Sample of the Object-Oriented Group (cont. 2)

```
,
1
2           Query updateQuery = new Query();
3
4           updateQuery.Replace("licenseplatenumber", "none");
5           updateQuery.Filter(updateQuery.Where("make").Equals("Mercedes").Or("model").Equals("CS250"));
6
7           Table updatedYear = cars.Update(updateQuery);
8
9           updatedYear.outputTable();
10          /*
11              Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
12              Renault,  Laguna,  322 SKO,  80,  2014,  2015,
13              Mercedes,  C250,  none,  56,  2015,  2016,
14              Ford,  Fusion,  910 GRE,  12,  2017,  2017,
15          */
16
17      }
```

Code Sample 78: Sample of the Object-Oriented Group (cont. 3)

```
1  package sample;
2  import library.*;
3
4  public class SampleA {
5
6      /**
7       * Results of print commands are in the comments and
8       * formatted for better readability.
9       *
10      * The format of the tables in this sample:
11      *
12      * - cars -
13      *
14      * id (int) | make (String) | model (String) | year (int) |
15      * licenseplatenumber (String) | registrationyear (int)
16      *
17      * - ticketlist -
18      *
19      * tid (int) | carid (int) | tickettype (String) | description (String)
20      *
21      *
22      */
23     public void sample(Table cars, Table ticketlist) throws Exception {
24
25         cars.outputTable();
26         /*
27             Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
28             Renault,  Laguna,  322 SKO,  80,  2014,  2015,
29             Mercedes,  C250,  660 OLK,  56,  2015,  2016,
30             Ford,  Fusion,  910 GRE,  12,  2017,  2017,
31         */
32
33         ticketlist.outputTable();
34         /*
35             Parking,  3,  41,  no time on parking meter,
36             Parking,  5,  56,  student in staff space,
37             Speeding,  8,  56,  20mph too fast in 30mph zone,
38             Parking,  10,  41,  double parked,
39             Speeding,  11,  56,  10mph too fast in 65mph zone,
40             Speeding,  23,  41,  15mph too fast in 45mph zone,
41             Speeding,  43,  12,  drove faster than police car,
42             Parking,  44,  41,  parked in street,
43             Red light,  51,  56,  Ran red light 10 sec after red,
44             Speeding,  56,  12,  10mph too fast in 35mph construction zone,
45          */
```

Code Sample 79: Sample of the Hybrid Group

```
Query allCarInfoQuery = new Query();

Table allCarInfo = cars.Search(allCarInfoQuery);

allCarInfo.outputTable();
/*
    Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
    Renault,  Laguna,  322 SKO,  80,  2014,  2015,
    Mercedes,  C250,  660 OLK,  56,  2015,  2016,
    Ford,  Fusion,  910 GRE,  12,  2017,  2017,
 */


Query insertCarsQuery = new Query();

insertCarsQuery.IntoFields("id, make, model, year, licenseplatenumber, registrationyear");
insertCarsQuery.SetValues("2345, 'Toyota', 'Verso', 2016, '326 NEB', 2017");

Table carsInserted = cars.Insert(insertCarsQuery);

carsInserted.outputTable();
/*
    Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
    Renault,  Laguna,  322 SKO,  80,  2014,  2015,
    Mercedes,  C250,  660 OLK,  56,  2015,  2016,
    Ford,  Fusion,  910 GRE,  12,  2017,  2017,
    Toyota,  Verso,  326NEB,  2345,  2016,  2017,
 */

Query carsNotMercedesQuery = new Query();

carsNotMercedesQuery.AddFields("make, model, year");

carsNotMercedesQuery.Filter("make != 'Mercedes'");
carsNotMercedesQuery.SortHighToLow("year");

Table carsNotMercedes = cars.Search(carsNotMercedesQuery);

carsNotMercedes.outputTable();
/*
    Ford,  Fusion,  2017,
    Renault,  Laguna,  2014,
    Citroen,  Berlingo,  2003,
 */
```

Code Sample 80: Sample of the Hybrid Group (cont.)

```
Query carTicketQuery = new Query();

carTicketQuery.Combine(cars, "id", ticketlist, "carid");

carTicketQuery.AddFields("id, make, model, description");

Table ticketsForCars = cars.Search(carTicketQuery);

ticketsForCars.outputTable();
/*
    41,  Citroen,  Berlingo,  no time on parking meter,
    41,  Citroen,  Berlingo,  double parked,
    41,  Citroen,  Berlingo,  15mph too fast in 45mph zone,
    41,  Citroen,  Berlingo,  parked in street,
    56,  Mercedes,  C250,  student in staff space,
    56,  Mercedes,  C250,  20mph too fast in 30mph zone,
    56,  Mercedes,  C250,  10mph too fast in 65mph zone,
    56,  Mercedes,  C250,  Ran red light 10 sec after red,
    12,  Ford,  Fusion,  drove faster than police car,
    12,  Ford,  Fusion,  10mph too fast in 35mph construction zone,
 */

Query carsOlderQuery = new Query();

carsOlderQuery.AddFields("make, model, year, licenseplatenumber, registrationyear");

carsOlderQuery.Filter("year < 2015");
carsOlderQuery.SortLowToHigh("year");

Table carsOlderThan2015 = cars.Search(carsOlderQuery);

carsOlderThan2015.outputTable();
/*
    Citroen,  Berlingo,  2003,  255 FAD,  2005,
    Renault,  Laguna,  2014,  322 SKO,  2015,
 */
```

Code Sample 81: Sample of the Hybrid Group (cont. 2)

```
        Query updateQuery = new Query();

        updateQuery.Replace("licenseplatenumber", "none");
        updateQuery.Filter("make = 'Mercedes' and model = 'CS250'");

        Table updatedYear = cars.Update(updateQuery);

        updatedYear.outputTable();
        /*
            Citroen,  Berlingo,  255 FAD,  41,  2003,  2005,
            Renault,  Laguna,  322 SKO,  80,  2014,  2015,
            Mercedes,  C250,  none,  56,  2015,  2016,
            Ford,  Fusion,  910 GRE,  12,  2017,  2017,
         */

    }
}
```

Code Sample 82: Sample of the Hybrid Group (cont. 3)

Bibliography

[ABB+04] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004.

[ACDLS02] Lerina Aversano, Gerardo Canfora, Andrea De Lucia, and Silvio Stefanucci. Understanding sql through iconic interfaces. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 703–708. IEEE, 2002.

[AG08] Jubin Abutalebi and David W Green. Control mechanisms in bilingual language production: Neural evidence from language switching studies. *Language and cognitive processes*, 23(4):557–582, 2008.

[ASR94] Jeanette Altarriba and Azara L Santiago-Rivera. Current perspectives on using linguistic and cultural factors in counseling the hispanic client. *Professional Psychology: Research and Practice*, 25(4):388, 1994.

[AT02] Erik M Altmann and J Gregory Trafton. Memory for goals: An activation-based model. *Cognitive science*, 26(1):39–83, 2002.

[BA14] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[BBE83] James M Boyle, Kevin F Bury, and R James Evey. Two studies evaluating learning and use of qbe and sql. In *Proceedings of the Human Factors Society Annual Meeting*, volume 27, pages 663–667. SAGE Publications Sage CA: Los Angeles, CA, 1983.

[BDL+13] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[BDLM09] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. To camelcase or under_score. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, pages 158–167, 2009.

[Bec16a] Brett A Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131. ACM, 2016.

[Bec16b] Brett A Becker. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 296–301. ACM, 2016.

[BGI+16] Brett A Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175, 2016.

[BLS] BLS. Software developers - summary. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. Accessed: 2018-04-20.

[BMD17] Anne L Beatty-Martínez and Paola E Dussias. Bilingual experience shapes language processing: Evidence from codeswitching. *Journal of Memory and Language*, 95:173–189, 2017.

[Bro78] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering*, pages 196–201. IEEE Press, 1978.

[CGHM06] Cyrille Comar, Matthew Gingell, Olivier Hainque, and Javier Miranda. Multi-language programming. In *GCC Developers' Summit*, page 59. Citeseer, 2006.

[CJ10] Stevica Cvetković and Dragan Janković. A comparative study of the features and performance of orm tools in a. net environment. In *International Conference on Object and Databases*, pages 147–158. Springer, 2010.

[con] Consort - Welcome to the CONSORT Website.

[CSW02] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[DLGR81] Ph Darondeau, Paul Le Guernic, and Michel Raynal. Types in a mixed language system. *BIT Numerical Mathematics*, 21(3):245–254, 1981.

[DLRTH11] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 208–212, New York, NY, USA, 2011. ACM.

[DSF09] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. Work in progress: an empirical study of static typing in ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU),Orlando, October 2009*, 2009.

[EG84] Bo Einarsson and W Morven Gentleman. Mixed language programming. *Software: Practice and Experience*, 14(4):383–395, 1984.

[EHRS14a] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API documentation and static typing affect API usability? In *Proceedings of the ICSE 2014 (accepted for publication)*, ICSE '14, 2014.

[EHRS14b] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA, 2014. ACM.

[ESM07] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.

[EV12] Benjamin J Evans and Martijn Verburg. *The well-grounded Java developer: Vital techniques of Java 7 and polyglot programming*. Manning Publications Co., 2012.

[FH15a] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. In *ACM SIGPLAN Notices*, volume 51, pages 154–167. ACM, 2015.

[FH15b] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 154–167, New York, NY, USA, 2015. ACM.

[Fje83] Richard K Fjeldstad. Application program maintenance study. *Report to Our Respondents, Proceedings GUIDE*, 48, 1983.

[Fje08] Hans-Christian Fjeldberg. *Polyglot programming. a business perspective.* PhD thesis, Master thesis, Norwegian University of Science and Technology, 2008.

[FMF12] Andy Field, Jeremy Miles, and Zoë Field. *Discovering statistics using R.* Sage publications, 2012.

[For08] Neal Ford. *The productive programmer.* " O'Reilly Media, Inc.", 2008.

[FSP+13] Scott D Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2):14, 2013.

[Gan77] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.

[gar] Gartner says worldwide software market grew 4.8 percent in 2013. `https://www.gartner.com/newsroom/id/2696317`. Accessed: 2018-06-02.

[GBB17] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, pages 758–769. IEEE Press, 2017.

[GM12] Torsten Grust and Manuel Mayr. A deep embedding of queries into ruby. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1257–1260. IEEE, IEEE, 2012.

[GP+96] Thomas R. G. Green, Marian Petre, et al. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of visual languages and computing*, 7(2):131–174, 1996.

[GR13] Torsten Grust and Jan Rittinger. Observing sql queries in their natural habitat. *ACM Transactions on Database Systems (TODS)*, 38(1):3, 2013.

[Gre89] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[Gre98] David W Green. Mental control of the bilingual lexico-semantic system. *Bilingualism: Language and cognition*, 1(2):67–81, 1998.

[HA01] Roberto R Heredia and Jeanette Altarriba. Bilingual language mixing: Why do bilinguals code-switch? *Current Directions in Psychological Science*, 10(5):164–168, 2001.

[Han10a] Stefan Hanenberg. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 300–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[Han10b] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, 2010. ACM.

[Han10c] Stefan Hanenberg. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 22–35, 2010.

[Har77] L.R. Harris. User oriented data base query with the ROBOT natural language query system. *International Journal of Man-Machine Studies*, 9(6):697–713, November 1977.

[HB88] Robert M Herndon and Valdis A Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.

[HH13a] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 457–474. ACM, 2013.

[HH13b] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: an empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 457–474, 2013.

[HHLD11] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[HKR+14a] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[HKR+14b] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[HKV13] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php feature usage: a static analysis perspective. In *Proceedings of the 2013 international symposium on software testing and analysis*, pages 325–335. ACM, 2013.

[Hof15] Johannes C. N. Hofmeister. *Influence of identifier length and semantics on the comprehensibility of source code*. Department of Psychology, University of Heidelberg, Germany, 2015.

[Hud97] Paul Hudak. Domain-specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[IBNW09] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.

[JH16] Ishaq Jound and Hamed Halimi. Comparison of performance between raw sql and eloquent orm in laravel, 2016.

[jni] Java native interface specification–contents. `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`. Accessed: 2018-04-26.

[JT03] Hannu Jaakkola and Bernhard Thalheim. Visual sql–high-quality er-based query treatment. *Conceptual modeling for novel application domains*, pages 129–139, 2003.

[Kai15a] Antti-Juhani Kaijanaho. The extent of empirical evidence that could inform evidence-based design of programming languages: A systematic mapping study. *Jyväskylä Licentiate Theses in Computing, University of Jyväskylä*, 2015.

[Kai15b] Kaijanaho Kaijanaho. *Evidence-based programming language design : a philosophical and methodological exploration.* University of Jyväskylä, Finnland, 2015.

[KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: a systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

[KFD17] Jorge R Valdés Kroff and Matías Fernández-Duque. Experimentally inducing spanish-english code-switching. *Multidisciplinary Approaches to Bilingualism in the Hispanic and Lusophone World*, 13:211, 2017.

[KHR+12] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 153–162, 2012.

[KMB+96] Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.

[KMCA06] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12), 2006.

[Kru92] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[KS94] Judith F Kroll and Erika Stewart. Category interference in translation and picture naming: Evidence for asymmetric connections between bilingual memory representations. *Journal of memory and language*, 33(2):149–174, 1994.

[KWDE98] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. Program comprehension in multi-language systems. In *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, pages 135–143. IEEE, 1998.

[Lan70] C. A. Lang. Languages for writing systems programs. In *Software Engineering Techniques, Buxton and Randell Ed., Nato Conference report*, pages 101–106, 1970.

[LBB07] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. Scents in programs: Does information foraging theory apply to program maintenance? In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 15–22. IEEE, 2007.

[LDO05] Grit Liebscher and JENNIFER DAILEY-O'CAIN. Learner code-switching in the content-based foreign language classroom. *The Modern Language Journal*, 89(2):234–247, 2005.

[Li96] Ping Li. Spoken word recognition of code-switched words by chinese–english bilinguals. *Journal of memory and language*, 35(6):757–774, 1996.

[LMFB06] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[LPJ14] Fei Li, Tianyin Pan, and Hosagrahar V Jagadish. Schema-free sql. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1051–1062. ACM, ACM, 2014.

[MAD+01] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. ACM, 2001.

[MB15] Philip Mayer and Alexander Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 4. ACM, 2015.

[MFMZ14] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 19–29. ACM, 2014.

[MHR+12a] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tucson, AZ, USA, October 21-25, 2012*, OOPSLA'12, pages 683–702. ACM, 2012.

[MHR+12b] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[MK71] John Macnamara and Seymour L Kushnir. Linguistic independence of bilinguals: The input switch. *Journal of Verbal Learning and Verbal Behavior*, 10(5):480–487, 1971.

[MR13] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18, October 2013.

[MS] Victoria Marian and Anthony Shook. The cognitive benefits of being bilingual. `http://dana.org/Cerebrum/2012/The_Cognitive_Benefits_of_Being_Bilingual/`. Accessed: 2018-06-04.

[MV94] Anneliese von Mayrhauser and A. Marie Vans. Comprehension processes during large scale maintenance. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 39–48, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[MV96] Anneliese von Mayrhauser and A. Marie Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[MV97] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the Seventh Workshop on Empirical Studies of Programmers*, pages 157–179, New York, NY, 1997. ACM Press.

[NC15] C. Nagy and A. Cleve. Mining Stack Overflow for discovering error patterns in SQL queries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 516–520, September 2015.

[NMB11] Nan Niu, Anas Mahmoud, and Gary Bradshaw. Information foraging as a foundation for code navigation (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 816–819. ACM, 2011.

[nod] Node.js.

[NXK17] Greg L Nelson, Benjamin Xie, and Andrew J Ko. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 2–11. ACM, 2017.

[OH16] Sebastian Okon and Stefan Hanenberg. Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? a controlled experiment. In *Internation Conference on Program Comprehension*, pages to appear, preprint made available by the authors, 2016.

[Ols16] Daniel J Olson. The gradient effect of context on language switching and lexical access in bilingual production. *Applied Psycholinguistics*, 37(3):725–756, 2016.

[Ols17] Daniel J Olson. Bilingual language switching costs in auditory comprehension. *Language, Cognition and Neuroscience*, 32(4):494–513, 2017.

[PC95] Peter Pirolli and Stuart Card. Information foraging in information access environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 51–58. ACM Press/Addison-Wesley Publishing Co., 1995.

[PC98] Peter Pirolli and Stuart K Card. Information foraging models of browsers for very large document spaces. In *Proceedings of the working conference on Advanced visual interfaces*, pages 83–93. ACM, 1998.

[Pen87a] Nancy Pennington. Comprehension strategies in programming. In Gary M. Olson, Sylvia Sheppard, Elliot Soloway, and Ben Shneiderman, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, Westport, CT, 1987. Greenwood Publishing Group Inc.

[Pen87b] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014.

[php] W3techs - extensive and reliable web technology surveys.

[PHR14a] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 212–222. ACM, 2014.

[PHR14b] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 212–222, New York, NY, USA, 2014. ACM.

[Pig] D Piggott. Hopl: an interactive roster of programming languages. `http://hopl.info/`. Accessed: 2018-04-19.

[pol] Polyglot programming. `http://nealford.com/memeagora/2006/12/05/Polyglot_Programming.html`. Accessed: 2018-04-25.

[PT98] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, 1998.

[RKS+09] Marko Rosenmüller, Christian Kästner, Norbert Siegmund, Sagar Sunkle, Sven Apel, Thomas Leich, and Gunter Saake. Sql á la carte–toward tailor-made data management. *Datenbanksysteme in Business, Technologie und Web (BTW)–13. Fachtagung des GI-Fachbereichs" Datenbanken und Informationssysteme"(DBIS)*, 2009.

[RLMW14] Julian Rith, Philipp S Lehmayr, and Klaus Meyer-Wegener. Speaking in tongues: Sql access to nosql systems. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 855–857. ACM, 2014.

[RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[Rya13] Camille Ryan. Language use in the united states: 2011. *American community survey reports*, 22:1–16, 2013.

[SF14] Carlos Souza and Eduardo Figueiredo. How do programmers use optional typing?: An empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 109–120, New York, NY, USA, 2014. ACM.

[SH11a] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, Portland, Oregon, USA, 2011. ACM.

[SH11b] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, Portland, Oregon, USA, 2011. ACM.

[SH14a] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of APIs - as long as the type names are correct: An empirical study. In *Proceedings of the Modularity 2014 (accepted for publication)*, AOSD '14, 2014.

[SH14b] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of apis (as long as the type names are correct): an empirical study. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 99–108, 2014.

[SH14c] Andreas Stefik and Stefan Hanenberg. The programming language wars: Questions and responsibilities for the programming language community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 283–299, New York, NY, USA, 2014. ACM.

[SHM+14] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Amschler Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In *Proceedings of the 2014 IEEE 20th International Conference on Program Comprehension*, ICPC '14, pages 223–231. IEEE Computer Society, 2014.

[SK86] David W Stephens and John R Krebs. *Foraging theory*. Princeton University Press, 1986.

[SKA+14] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389. ACM, 2014.

[SKL+14a] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SKL+14b] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[SM79] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[SM08] Jeffrey Stylos and Brad A Myers. The implications of method placement on api learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

[Sme93] John B Smelcer. User errors in the use of the structured query language (sql). In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 37, pages 382–386. SAGE Publications Sage CA: Los Angeles, CA, 1993.

[SPP+17] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150. ACM, 2017.

[SS13] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, November 2013.

[SSE+14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 724–734, New York, NY, USA, 2014. ACM.

[SSSS11] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. An empirical comparison of the accuracy rates of novices using the quorum, perl, and randomo programming languages. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, PLATEAU '11, pages 3–8, New York, NY, USA, 2011. ACM.

[Sta84] Thomas A Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, (5):494–497, 1984.

[STB09] Dario D Salvucci, Niels A Taatgen, and Jelmer P Borst. Toward a unified theory of the multitasking continuum: From concurrent performance to task switching, interruption, and resumption. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1819–1828. ACM, 2009.

[Sur11] KP Suresh. An overview of randomization techniques: an unbiased assessment of outcome in clinical research. *Journal of human reproductive sciences*, 4(1):8, 2011.

[SW90] Jean Scholtz and Susan Wiedenbeck. Learning second and subsequent programming languages: A problem of transfer. *Int. J. Hum. Comput. Interaction*, 2(1):51–72, 1990.

[SW93] Jean Scholtz and Susan Wiedenbeck. Using unfamiliar programming languages: the effects on expertise. *Interacting with Computers*, 5(1):13–30, 1993.

[TABM03] J Gregory Trafton, Erik M Altmann, Derek P Brock, and Farilee E Mintz. Preparing to resume an interrupted task: Effects of prospective goal encoding and retrospective rehearsal. *International Journal of Human-Computer Studies*, 58(5):583–603, 2003.

[Tic98] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31:32–40, 1998.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[Tra10] V Javier Traver. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[TSV18] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. Errors and complications in sql query formulation. *ACM Transactions on Computing Education (TOCE)*, 18(3):15, 2018.

[TT14] Federico Tomassetti and Marco Torchiano. An empirical assessment of polyglot-ism in github. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 17. ACM, 2014.

[TWCG12] Arash Termehchy, Marianne Winslett, Yodsawalai Chodpathumwan, and Austin Gibbons. Design independent query interfaces. *IEEE Transactions on Knowledge and Data Engineering*, 24(10):1819–1832, 2012.

[USH+16] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, pages 760–771. ACM, 2016.

[vD97] Arie van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. *Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39, 1997.

[VDK+98] Arie Van Deursen, Paul Klint, et al. Little languages: Little maintenance? *Journal of software maintenance*, 10(2):75–92, 1998.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[vie] The vietnam of computer science. `http://blogs.tedneward.com/post/the-vietnam-of-computer-science/`. Accessed: 2018-04-29.

[VLP09] Markel Vigo, Barbara Leporini, and Fabio Paternò. Enriching web information scent for blind users. In *Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility*, pages 123–130. ACM, 2009.

[Vou84] Mladen A Vouk. On the cost of mixed language programming. *ACM SIGPLAN Notices*, 19(12):54–60, 1984.

[VTTM12] Antonio Vetro, Federico Tomassetti, Marco Torchiano, and Maurizio Morisio. Language interaction and quality issues: an exploratory study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 319–322. ACM, 2012.

[Wei05] Li Wei. How can you tell?: towards a common sense explanation of conversational code-switching. *Journal of Pragmatics*, 37(3):375–389, 2005.

[WK14] Jacqueline Whalley and Nadia Kasto. A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 279–284. ACM, 2014.

[WM99] Alan Wexelblat and Pattie Maes. Footprints: history-rich tools for information foraging. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 270–277. ACM, 1999.

[WW15] David Weintrop and Uri Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *ICER*, volume 15, pages 101–110, 2015.

[YS93] MY-M Yen and Richard W. Scamell. A human factors experimental comparison of sql and qbe. *IEEE Transactions on Software Engineering*, 19(4):390–409, 1993.

[YTF17] W QUIN YOW, JESSICA SH TAN, and SUZANNE FLYNN. Code-switching as a marker of linguistic competence in bilingual children. *Bilingualism: Language and Cognition*, pages 1–16, 2017.

[Yue07] Timothy T Yuen. Novices' knowledge construction of difficult concepts in cs1. *ACM SIGCSE Bulletin*, 39(4):49–53, 2007.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Phillip Merlin Uesbeck

pmerlindrews@gmail.de

Degrees:

Master of Science in Computer Science 2016
University of Nevada, Las Vegas

Bachelor of Science in Applied Computer Science - Software Engineering 2014
Universität Duisburg-Essen

Publications:

Rafalski, T, Uesbeck, P.M., Panks-Meloney, C., Daleiden, P., Allee, W., Mcnamara, A., and Stefik, A. "*A Randomized Controlled Trial on the Wild Wild West of Scientific Computing with Student Learners.*" Proceedings of the 2019 ACM Conference on International Computing Education Research, pp. 239-247. ACM, 2019.

Uesbeck, P. M., Stefik, A., "*A Randomized Controlled Trial on the Impact of Polyglot Programming in a Database Context.*" 9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018).

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. "*An Empirical Study on the Impact of C++ Lambdas and Programmer Experience*" Software Engineering (ICSE), 2016 IEEE/ACM 38th IEEE International Conference on. Vol. 1. IEEE, 2016.

Dominguez, M., Bernacki, M. L., and Uesbeck, P. M. "*Predicting STEM Achievement with Learning Management System Data: Prediction Modeling and a Test of an Early Warning System.*" EDM 2016 (pp. 589-590).

Thesis Title: Towards a Better Understanding of the Impact of Polyglot Programming on Programmer Productivity

Thesis Examination Committee:

Chairperson, Dr. Andreas Stefik, Ph.D.

Committee Member, Dr. Jan "Matt" Pedersen, Ph.D.

Committee Member, Dr. Evangelos Yfantis, Ph.D.

Committee Member, Dr. Hal Berghel, Ph.D.

Graduate Faculty Representative, Dr. Deborah Arteaga, Ph.D.