A Randomized Controlled Trial on the Effects of Embedded Computer Language Switching

???

ABSTRACT

Polyglot programming, the use of multiple programming languages during the development process, is common practice in modern software development. This study investigates this practice through a randomized controlled trial conducted under the context of database programming. Participants in our study were given coding tasks written in Java and one of three SQL-like language embedded languages. One was plain SQL in strings, one was in Java only, and the third was a hybrid embedded language that was closer to the host language. We recorded 109 valid data sets and the results showed significant differences in how developers of different experience levels code using polyglot techniques. Notably, less experienced programmers wrote correct programs faster in the hybrid condition (frequent, but less severe, switches), while more experienced developers that already knew both languages performed better in traditional SQL (less frequent but more complete switches). Our results indicate that the productivity impact of polyglot programming is complex and experience level dependent.

CCS CONCEPTS

• **Human-centered computing** → *Empirical studies in HCI*; • **Software and its engineering** → *Domain specific languages.*

KEYWORDS

polyglot programming, programming languages, computer language switching, productivity, database programming, experience, randomized controlled trial

ACM Reference Format:

???. 2020. A Randomized Controlled Trial on the Effects of Embedded Computer Language Switching. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Polyglot programming is common in software development industry. Notably, Tomassetti and Torchiano found that ninety-seven percent of open source projects used two or more computer languages [25]. Further, the average number of computer languages in open source projects is about five [13, 25] and developers claim in surveys that they "know" ten different computer languages [14]. Shrestha et al. observed that even experienced professional developers have significant challenges in learning new languages [15].

With programming in different languages comes the need to switch between them, which we observe happens at three levels: the 1) project level, 2) file level, and 3) embedded level. The project and file level switches are implied by their name, while embedded in this context means one language is embedded in another which can take many different forms.

In the linguistics literature and in reference to natural language, these ideas have similar, but not exact, parallels. For example, embedded switching has parallels to *intra*-sentential code switching [10, 12, 29], a rapid switch between natural languages in conversation. File level switching might be considered a parallel to *inter*-sentential code switching [10, 12, 29], which is the process of switching languages between utterances. Project level switches are a possible parallel to natural language switching [3]. The analogies are not exact, but it is important to recognize that the linguistics literature has shown evidence that there is a cost to switching between natural languages [16, 17], which might provide evidence for the underlying causes observed by Shrestha et al. [15]. Given that program comprehension and natural language comprehension may use the same areas of the brain [18, 20], that the cost of hiring software developers is $103,560 per year [5], and given polyglot programming is so commonplace, it is important to better understand this practice.

In this paper, we present a double-blind randomized controlled trial focusing on one aspect of polyglot programming, i.e., embedded switching in a database context. This is a large-scale replication of a study presented in <citation removed for anonymity>. We chose this particular approach as a first step, in part, because the practice is commonplace. We illustrate a common example using SQL to query databases from within general purpose programming languages, as shown in the following SQL query for querying in Java using the JDBC (Java Database Connectivity, algorithm 1) API[1].

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

Algorithm 1: Example of a simple JDBC query

The goal was to determine whether polyglot programming would impact developers at different experience levels in different ways, hypothesizing that younger developers with less experience might have more difficulty with the practice of polyglot programming. To model different ways in which language switching in an embedded context could occur, three different versions of a querying APIs were created and these were evaluated with freshman, sophomores,

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

juniors, and seniors at <University Removed for Anonymity>, in addition to professional programmers. The results showed that the style of polyglot significantly impacts people of different experiences in different ways and that the style of polyglot is also impacted by experience. Specifically, those that already knew both SQL and Java were able to use a polyglot approach effectively, but that this was clearly a skill learned through experience. Interestingly, we also found that by altering embedded language to be closer to the host, our hybrid condition, even less experienced students as inexperienced as freshmen in college, were much more likely to be able to complete the tasks. This implies both that polyglot programming could use more study under new conditions to fully understand its implications and that, as a community, we may be able to make such programming easier by carefully considering the semantic mismatch between host and embedded languages.

Finally, our research team is interested in the concept of evidence standards, as used in other fields. We report this paper under an adapted CONSORT format [9], which defines the structure of this paper and was designed to increase reporting standards in studies. As such, our section headers match this externally vetted approach to reporting on our methodology.

2 RELATED WORK

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"* [2], a view that seems to drive the field of domain specific language research which proposes using specialized languages for better productivity and maintenance [27]. Claims have been made 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 [8], but also that the need to learn more languages creates a strain on the developers and that introduction of more languages to a project can reduce the pool of developers able to maintain the project [8].

This latter view, that additional languages causes strain, was recently explored by Shrestha et al. as part of a study examining professional developers and their questions on stack overflow, in addition to interviews [15]. Findings from that study showed that the mismatch between programming languages, which may not be a 1 to 1 mapping, is reported as being a barrier by even professionals and that this difference varies across language pairs (e.g., Java to Kotlin vs. other pairs). We use a very different methodology here, a randomized controlled trial on a larger variety of experience levels as opposed to a mixed-method study on only professionals, but our results are complementary. Notably, our study here contributes how these challenges are impacted by experience level and, as opposed to looking at developer questions, we more directly measure how effectively developers can program using polyglot approaches.

Programmer productivity is studied in a variety of aspects of programming. Studies range from programming language features such as syntax [21] and type systems [11], over API design [22] and the effect of errors [4] to studies trying to investigate the cognitive processes involved [6, 20]. These studies on APIs, syntax, or others provided guidance to this work, and while some studies have focused on the effect of polyglot programming on code quality [25, 28], there is little in the literature that contains measurements of polyglot programming itself.

2.1 Objective

The experiment described in this chapter aims to compare if differing amounts of computer language switching impact 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 is 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 exclusively uses Java (monoglot), while the string-based API requires language switches from Java to SQL and back when a query is being written (polyglot). The hybrid approach it still polyglot, but we carefully controlled the word choices and syntax to be closer to the host language of Java—in effect, we were careful about the mismatch between languages. The different API designs will be discussed in more detail in section 3.4. To keep the tasks simple for participants, this study focuses on the implementation of simple queries.

The null-hypotheses are as follows:

$H_0 1$: *There is no relationship between the 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 differences between groups.

The second hypothesis was 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. For example, while previous work looked at professionals, for the younger end of our experience spectrum, we wanted to know whether students even realized they were doing polyglot programming, and if so, whether their perceptions of their performance matched with our observations.

The aim of hypothesis three was 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.

2.2 Design Progress

We pilot tested our experiment in several ways. First, 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. Notably, in five preliminary pilot studies, the experiment was tested with different numbers of participants. 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 pilots 4 and 5. This extensive piloting process, with iteration at each step, provided us an evidence-based way for us to critique our experiment over time, get feedback from our research team and other stakeholders, and improve our reporting before we submit a final paper for publication.

2.3 Structure

The layout of this write-up was inspired by the CONSORT (Consolidated Standards of Reporting Trials) standard as it is used in the medical sciences [1], 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 structured as follows: First, we will go over the design of the experiment in section 3. That section will include information about the design of the trial in subsection 3.1, the participants in subsection 3.2, the study setting in subsection 3.3, the intervention in subsection 3.4, the outcome variables in subsection 3.5, the sample size in subsection 3.6, the randomization in 3.7, and the blinding in subsection 3.8. Results will be shown and discussed in subsection 4, followed by qualitative results in section 5 and a discussion in section 6, finally the chapter will end in a conclusion in section 7. We point out that CONSORT reports studies somewhat differently than is sometimes observed in our field and we cannot match it exactly (e.g., computer science venues rarely allow for trial registration under CONSORT rules), but this was a simple way to ensure our reporting was as rigorous and transparent as it could be.

3 METHODS

3.1 Trial Design

Our experiment was 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.

3.2 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 <University Removed for Review>, 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 [19] and significant differences between levels have been found in respect to time to solution [26]. 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, as is required under ethics guidelines at our university.

3.3 Study Setting

The study was conducted using the <removed for anonymity> 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 number of test-cases. The test-cases 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 were asked to give some feedback on the experiment and then they 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 groups 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.

3.4 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.

```
public Table query(Table table) throws Exception {
    Query query = new Query();
    q.Where("value1").LessThan(234).And("value2").
        ↪ GreaterThan(42)
    query.Prepare("SELECT Field1, Field2 "
        +"FROM table WHERE Field1 < 234 AND Field2 > 42 "
        +"ORDER BY Field3 DESC");
    Table result = table.Search(query);
    return result;
}
```

Algorithm 2: Example of the String-based Design

The first design (see example in algorithm 2) 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 [24]). 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.

```
public Table query(Table table) throws Exception {
  Query query = new Query();
  query.AddField("Field1")
    .AddField("Field2");
  query.Filter(q.Where("Field1").LessThan(234).And("
      ↪ Field2").GreaterThan(42));
  query.SortHighToLow("Field3");
  Table result = table.Search(query);
}
```

Algorithm 3: Example of the Object-Oriented Design

The second design (see example in listing 3) requires a programmer to use a number of method calls to produce a query. This approach loses the flexibility that comes with using 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.

```
public Table query(Table table) throws Exception {
    Query query = new Query();
    query.AddFields("Field1, Field2");
    query.Filter("Field1 < 234 AND Field2 > 42");
    query.SortHighToLow("Field3");
    Table result = charts.Search(query);
    return result;
}
```

Algorithm 4: 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 in string form. Notably, this approach avoids both the use of objects to build a filter statement and SQL syntax. This custom syntax is thus somewhere in between a switch from SQL to Java because the syntax is closer to the host language. We can imagine many different kinds of hybrid conditions being interesting to study empirically, but derived ours by looking at pilot data and trying to keep the language as close to the host as we could, while also removing the object mapping required in the monoglot condition.

Put another way, we see polyglot as a spectrum of language design decisions. On one side, if SQL is embedded raw into Java, there is no direct connection between the languages—they were designed separately and are embedded. On the other side of the spectrum is a monoglot approach, where users create objects inside of Java and manipulate them to do SQL-like operations. Finally, the hybrid condition was designed to be embedded and is cognizant of its application. Thus, languages may not be 1 to 1 mappings between each other, for good reason, but adjustments to the syntax and semantics still might impact developer productivity in interesting and important ways and by evaluating this spectrum in our study we can test whether this matters, how much, and under what experience levels.

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.

```
package library;
import library.*;
public class Task1 {
    /** Please write this method to return a Table object
         ↪  containing all columns for all entries with
     *  an id smaller than 32 and sorted from high salary
          ↪  to low salary
     * Table information:
     * - prof -
     * id (int) | firstname (String) | lastname (String)
          ↪ | salary (int)
     * Use the technique shown to you in the samples
          ↪ given
     */
    public Table query(Table prof) throws Exception {
        // Your Code here
        return null;
    }
}
```

Algorithm 5: Task 1 scaffolding.

```
    public Table query(Table prof) throws Exception {
        Query q = new Query();
        q.Prepare("SELECT * FROM professors" +
        " WHERE id < 32 ORDER BY salary DESC");
        Table r = prof.Search(q);
        return r;
    }
```

Algorithm 6: Task 1 Solution for group SQL.

```
    public Table query(Table prof) throws Exception {
        Query q = new Query();
        q.SortHighToLow("salary");
        q.Filter(q.Where("id").LessThan(32));
        Table r = prof.Search(q);
        return r;
    }
```

Algorithm 7: Task 1 Solution for the object-oriented group.

```
    public Table query(Table prof) throws Exception {
        Query q = new Query();
        q.SortHighToLow("salary");
        q.Filter("id < 32");
        Table r = prof.Search(q);
        return r;
    }
```

Algorithm 8: Task 1 Solution for the hybrid group.

An example of what the tasks looked like to the participants can be seen in algorithm 5, 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 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 5 can be seen in algorithm 6, 7, and 8. Additional code has to be omitted for space reasons and will be included in the final version of the paper as the repository is not anonymous.

3.5 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.

Finally, a question about switching computer languages was added to the end of the experiment 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?"

3.6 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 cannot be compared to the current version as changes affecting the results have been made. There are 5 levels of education to target for this experiment: sophomores, juniors, seniors, graduate students, and professionals.

3.7 Randomization

The process of assigning participants to the three groups happened on the online platform and followed the covariate adaptive randomization approach [23]. 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.

3.8 Blinding

The experiment was double blind, as the assignment of participants to their group was automatic. The researchers could not determine who was assigned to which group, nor did participants know what group they were assigned to or even that there were groups. 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 QUANTITATIVE RESULTS

This section will present the results of the experiment. For this experiment 149 participants were recruited within computer science classes at <University removed for anonymity> as well as professional software developers. The script used to analyze the data will be included in the final version of the paper as the current repository is not anonymous.

4.1 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.

4.2 Baseline Data

An overview of the participants' average time per group and per task can be found in table 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 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 1 shows the average task times between the three groups. Figure 2 shows the difference in task times broken down by level of education.

As the occurrence of participants not being able to finish all tasks 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% 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 3.

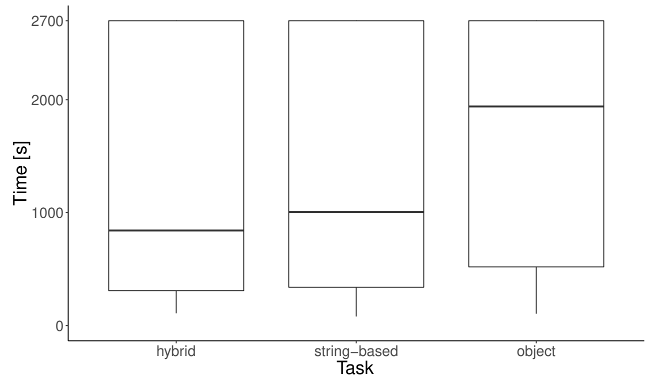

Figure 1: Boxplot of results between the groups

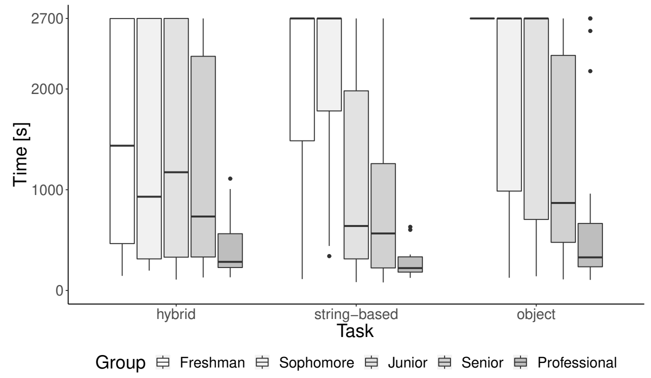

Figure 2: Boxplot of results between the groups based on their level of education

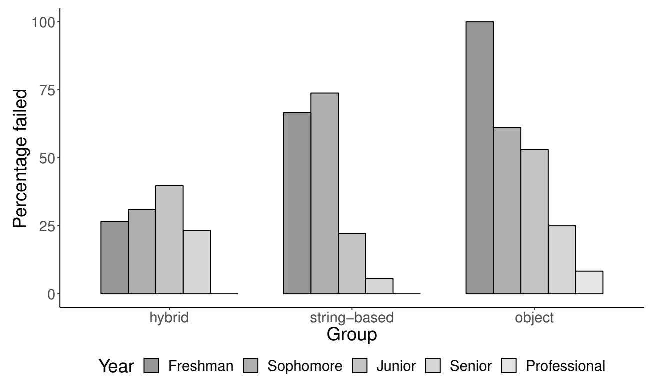

Figure 3: Barchart of results between the groups showing the percentage of failed tasks by level of education

Table 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

4.3 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. Thus, we used the standard Greenhouse-Geisser corrections when appropriate.

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 visual representation of the differences can be seen in figure 4. Testing the natural language differences for only professional programmers 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). The difference between the hybrid and string-based groups was not significant (p = 1.0000).

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. 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

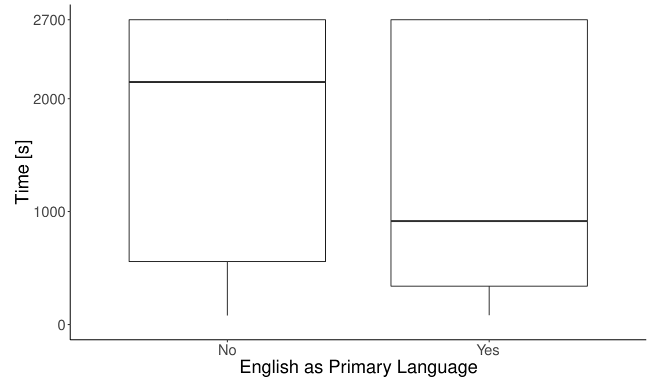

Figure 4: Boxplot of results between primary English speakers and non-primary English speakers

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 [7]. 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 and p = 1 might look wrong, it is in the nature of a model fitting the data "perfectly" as the term is defined in the cited literature.

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 odds 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," not literally but as defined by the numbers above, 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).

5 QUALITATIVE RESULTS

At the end 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. We explored answers to this question thematically as a way to check our assumptions about our experiment, but we did not formally code these answers. As such, while we think these responses are interesting, they should be interpreted as being also preliminary.

5.1 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 from more experienced participants. This is generally the picture one gets from the responses. Additionally, some of the participants report on the experience of having had to switch between languages to work on the experiment, as they are mostly familiar with another language, in those responses their main computer language (CL1) tends to be C++. Understandably, some participants were confused by the question as there was no switching in this group.

5.2 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 as said by a professional: "yes, java to SQL and back. Minimal." The other two participants that noticed the switch were another professional and a senior. Only one of them remarked that there might have been an impact on their productivity regarding the equality operator in both languages being different. All three were familiar with SQL and Java.

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 (typically C++) and Java, instead of the switch within the tasks, similar to the findings in the object-oriented group. One junior answered the question: "No, I did not feel like I needed to switch languages during the experiment. I think this increased my velocity while solving the tasks." Participants not noticing the differences between the languages is remarkable, however, since the syntax and semantics of SQL are very different from Java and C++. It might suggest that participants take the languages they are not familiar with as one single, combined language.

5.3 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. Once again, the majority of participants did not notice the switch. However, a hand full of participants did notice such as this statement by a junior: "I felt like it had a bit of SQL and Java." Once again, participants were more likely to discuss the switch from their CL1 to Java and most did not think there was a performance impact, regardless of the fact that the loglinear analysis shows us that these developer perceptions do not match actual performance.

6 DISCUSSION

6.1 Limitations

While much care was taken in creating the study in a way to adhere to high standards of reporting [1], this experiment has limitations that might make the results less correct or generalizable.

The design of generalizable tasks is always a significant challenge in any experiment. The typical trade-off is that easy to complete and short tasks are tractable in a study, whereas real-world, for some definition of the term, tasks "might" be longer, harder, or have other properties. In our case, we threaded the needle by first choosing tasks that were clearly relevant in the real world (e.g., inserting, deleting rows, joins), while pilot testing to ensure participants could actually complete them in a reasonable time frame. This natural limitation is one reason why it is important to cross-tab across experimental methodologies in multiple studies. For example, in reviewing the work of Shrestha et al. [15], our observations largely match despite very different methodologies. Despite this, no matter what tasks we ultimately choose, variations in the design of the tasks can change the results and, as such, it is important that our broader community replicate such findings in new and unique ways to know when they apply and when they do not. Finally, our tasks are small and while participants complete typical operations (e.g., joins, insertions), less artificial tasks should be compared using a similar methodology. For example, in our tasks, we pass code to the API via strings, but the impact of alternative strategies should be evaluated and compared.

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. Participants' environment is therefore only controlled by the participants themselves, making distractions and cheating possible. Mitigation 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 had a relatively small sample at each experience level. Despite clear statistical findings, we want to point out that while the results show clear trends that seem worth further investigation, more work is needed to know when and where these findings generalize. Further, neither students nor professionals are homogeneous groups. All people have different backgrounds and experiences. To really understand the impact of polyglot, significantly more tests, under different conditions, with different kinds of people, will be necessary to understand the big picture. For example, very young students are now learning computer science in K-12 education and polyglot may impact them or their teachers. Further, professionals at different points in their careers, or at different companies, might have unique experiences that impact our results under various conditions.

6.2 Interpretation

6.2.1 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 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 [2, 8]. 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 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 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 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 and chi-squared tests of task failures, which shows that there is an interaction between level of education, group, and the occurrence of maximum time task events. There are clear significant effects for all college levels between the string-based and the hybrid group with odds ratios indicating that freshmen and sophomores were 5 and 6 times more likely to fail a task in the string-based group, respectively. Juniors and seniors, on the other hand, were less likely to fail in the string-based than in the hybrid group, which makes sense that these students were likely exposed to both languages at this point in time. This is interesting because keep in mind that those in our hybrid group could not have taken a course that included the programming language—it was one we generated only for the experiment. As such, that the impact on younger students was so positive, while being comparable to those with experience, implies that the amount of mismatch in polyglot could be important for language designers.

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.

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 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 experienced developers.

Overall, the results regarding the differences between the groups seem clear: The monoglot object-oriented group was the slowest 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 that already had training in this approach coming in to the experiment. 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.

6.2.2 *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 66.66% denied that any switching was taking place. 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, while only occasional statements hinted at syntax issues when switching.

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. 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, it becomes reasonable to assume that the effect of computer language switching might be limited on the embedded language switching level.

6.2.3 *Productivity Difference based on Native Language.*

The results of the study show that, overall, participants that stated their primary language is different from English solved tasks significantly slower than participants that stated that their primary language is English. However, testing only experienced programmers, there was no significant difference. The survey did not track language history in sufficient detail. 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). 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 [2].

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.

7 CONCLUSION

This paper 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 software development. Three groups were developed, a monoglot group, a polyglot group with SQL and Java, and a hybrid group. Results showed that the impact of polyglot programming is not simple. We observed significant impacts based on experience level, the designs of the languages, and that the mismatch between languages, as observed in our hybrid group, can impact people in different ways. We believe future studies should investigate different kinds of polyglot approaches, in addition to perhaps formalizing how much mismatch there is between polyglot approaches. Further, language designers themselves should consider that language embedding can have very large impacts on different kinds of users, especially younger developers, and that reducing this mismatch may have positive impacts.

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

REFERENCES

[1] [n. d.]. Consort - Welcome to the CONSORT Website. http://www.consort-statement.org/

[2] [n. d.]. Polyglot Programming. http://nealford.com/memeagora/2006/12/05/Polyglot_Programming.html. Accessed: 2018-04-25.

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

[4] Brett A Becker. 2016. 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*. ACM, 296–301.

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

[6] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*. IEEE, 255–265.

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

[8] Hans-Christian Fjeldberg. 2008. *Polyglot programming. a business perspective*. Ph.D. Dissertation. Master thesis, Norwegian University of Science and Technology.

[9] The CONSORT Group. [n. d.]. CONSORT: Transparent Reporting of Trials. http://www.consort-statement.org/. Accessed: 2017-10-13.

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

[11] Michael Hoppe and Stefan Hanenberg. 2013. 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*. ACM, 457–474.

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

[13] Philip Mayer and Alexander Bauer. 2015. 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*. ACM, 4.

[14] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption. *SIGPLAN Not.* 48, 10 (Oct. 2013), 1–18. https://doi.org/10.1145/2544173.2509515

[15] Titus Barik Nischal Shrestha, Colton Botta and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020)*. ACM, New York, NY, USA. https://doi.org/3377811.3380352

[16] Daniel J Olson. 2016. The gradient effect of context on language switching and lexical access in bilingual production. *Applied Psycholinguistics* 37, 3 (2016), 725–756.

[17] Daniel J Olson. 2017. Bilingual language switching costs in auditory comprehension. *Language, Cognition and Neuroscience* 32, 4 (2017), 494–513.

[18] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 378–389.

[19] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334.

[20] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 140–150.

[21] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages.

[22] Jeffrey Stylos and Brad A Myers. 2008. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 105–112.

[23] KP Suresh. 2011. An overview of randomization techniques: an unbiased assessment of outcome in clinical research. *Journal of human reproductive sciences* 4, 1 (2011), 8.

[24] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. 2018. Errors and complications in SQL query formulation. *ACM Transactions on Computing Education (TOCE)* 18, 3 (2018), 15.

[25] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 17.

[26] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. 2016. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 760–771.

[27] Arie Van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36.

[28] Antonio Vetro, Federico Tomassetti, Marco Torchiano, and Maurizio Morisio. 2012. Language interaction and quality issues: an exploratory study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 319–322.

[29] W QUIN YOW, JESSICA SH TAN, and SUZANNE FLYNN. 2017. Code-switching as a marker of linguistic competence in bilingual children. *Bilingualism: Language and Cognition* (2017), 1–16.