A Controlled Experiment on the Effect of Ownership Rules and Mutability on Localizing Errors in Rust in Comparison to Java

Lukas Poos[1], Stefan Hanenberg[2][a], Stefan Gries[3] and Volker Gruhn[2][b]
[1]*Independent Researcher, Germany*
[2]*University of Duisburg–Essen, Essen, Germany*
[3]*codecentric AG, 42697 Solingen, Germany*

Keywords: Programming Languages, Empirical Study, User Study.

Abstract: The programming language Rust introduces language constructs such as ownership rules and mutability whose effect is, that undesired side-effects can be detected by the compiler. However, it is relatively unknown what the effect of such constructs on developers is. The present work introduces an experiment, where Rust and Java code was given to ten participants. The code, that consisted of ten function calls, contained one function that performed an undesired side-effect which led to an error in the main function. The participants' task was to identify the function that caused this effect. The experiment varied (in the Rust code) the number of calls where a parameter was passed as mutable (which is inherently the case in languages such as Java). Such variation had a strong ($p < .001$) and large ($\eta_p^2 = .459$) effect on participants. On average, it took the participants 29% more time to identify the function in Java. However, this number varied between -4.3% and 117%, depending on how many parameters where passed as mutable. Altogether, the experiment gives evidence that an explicit passing of variables as mutable has a positive effect on developers under the experimental conditions.

1 INTRODUCTION

The programming language Rust[1] is a relatively new programming language whose development started before 2010 and whose version 1.0 appeared in 2015. The language has its focus on performance, type and memory safety. While its syntax shares some similarities with other languages, it contains a number of innovative constructs that differ widely from main stream languages such as Java, C++, etc. One of these constructs are the ownership rules, where variables can be passed to other functions, but where the target function becomes owner of the passed value, and the client method loses its ownership. Another construct is the definition of mutability, where developers explicitly have to declare that a variable is permitted to change while the program is running.

While the theory behind ownership rules and mutability is quite well-understood and well-described (see, for example, (Clarke et al., 2013) among others), it is not clear what the effect of such constructs on developers is: whether developers are able to cope with such language constructs remains unclear. I.e., while there is no doubt about the theoretical implications of ownership rules and mutability, there is not much empirical evidence about the effect of ownership rules on developers: possibly developers have troubles to understand the effect of ownership rules. Having said this, a survey from 2024 by the Mozilla Rust team reports that a larger number of developers consider Rust to be hard to learn.[2] A survey from the same group reported in 2020 that ownership rules are considered as the most difficult part of the language.[3]

It is in general a well-documented phenomenon that not much empirical evidence exists for programming languages constructs in general. For example, Kaijanaho documented that the number of human-centered studies using randomized controlled trials (RCTs) in the field of programming language design up to 2012 on specific languages

[a] https://orcid.org/0000-0001-5936-2143
[b] https://orcid.org/0000-0003-3841-2548
[1] https://www.rust-lang.org/

[2] https://blog.rust-lang.org/2025/02/13/2024-State-Of-Rust-Survey-results.html
[3] https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html

Poos, L., Hanenberg, S., Gries, S. and Gruhn, V.
A Controlled Experiment on the Effect of Ownership Rules and Mutability on Localizing Errors in Rust in Comparison to Java.
DOI: 10.5220/0013647500003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 410-421
ISBN: 978-989-758-757-3; ISSN: 2184-2833

features was just 22–such low number of studies was confirmed by other authors as well (see, for example, (Buse et al., 2011; Ko et al., 2015) among others).

The present paper introduces a controlled experiment, where participants were required to identify a function that accidentally causes an undesired side-effect. More precisely, participants were given a main function where a data structure was defined and passed to other functions. It turned out that the explicit declarations of mutability had a measurable, positive effect on participants: on average, participants required 29% more time when the programming language Java was used.

2 OWNERSHIP RULES AND MUTABILITY IN RUST

We first introduce ownership rules in Rust, followed by an introduction of mutability. Then, we discuss the possible effects of both concepts on developers.

2.1 Ownership Rules

The principle of ownership rules in Rust is relatively simple: a function that defines a variable owns such variable. But when such variable is passed to another function, the passing function looses the ownership of the passed variable. As a consequence, the passing function can no longer access such variable.

Figure 1 illustrates the effect of ownership rules in Rust. There, the variable `text` is passed from the client function to the target function `pass_and_print`. Since the client function no longer owns `text`, it is not possible to access it a second time–the Rust code cannot compile.

Ownership in Rust is strongly connected to references. For example, it is possible to rewrite the code from Figure 1 in a way that it can be compiled (and executed). Instead of passing the ownership of `text` to another function, one can pass a reference to that variable. This requires to declare in the client method that a reference is being passed (and it requires the target method to declare that it expects a reference).

Figure 2 illustrates the previous code in a compilable way. The variable is passed as a reference (`&text`) which implies that only the ownership of such reference is passed to the function `pass_and_print_referenced_string`. As a consequence, the main method can still access `text` and print it out (although text–as a reference–was previously passed).

```
1  fn main() {
2    let text:String = String::from("Hello, world!");
3
4    // text is passed to function pass_and_print,
5    // i.e., ownership
6    pass_and_print(text);
7
8    // ERROR: tries to access variable text
9    // that is no longer owned by the present function
10   println!("{}", text);
11 }
12
13 fn pass_and_print(text: &String) {
14   println!("{}", text);
15 }
```

Figure 1: Faulty code in main function.

```
1  fn main() {
2    let text:String = String::from("Hello, world!");
3
4    // reference to text is passed to function
5    // pass_and_print_referenced_string
6    pass_and_print_referenced_string(&text);
7
8    // access variable text
9    // that is still owned by the present function
10   print!("{}", text);
11 }
12
13 fn pass_and_print_referenced_string(text: &String) {
14   print!("{}", text);
15 }
```

Figure 2: Valid code that accesses a variable a second time.

Ownership rules also imply that the ownership of returned values is passed from the returning function to the invoking function. As a consequence, it is not unusual to use multiple variables in a client method (where some variables contain values that are passed to functions and some variables contain values returned from functions).

From first glance, ownership rules do not look as essential as they are–it looks like one must just take care how variables are passed. Practically, the effect for developers is larger than one would expect, especially for developers that are used to side-effects. For example, it is usual in object-oriented programming to keep an object's state and pass this state to other objects, while the passed state is still available in the original object. For example, in Java it is usual to share parts of an object's state via so-called `getter` methods. I.e., while Rust has ownership rules with a special focus on the potential problems caused by side-effects, there are programming languages where side-effects are not considered as a notable problem.

2.2 Mutability

Although the connection to ownership rules is not directly obvious, the concept of mutability is another language feature in Rust that has an effect on how variables can be passed and used: only variables that

```
fn main() {
  let vector = vec![1, 2, 3, 4, 5, 6, 7] ;
  work_on_vector(vector);
}

fn work_on_vector(mut v: Vec<i32>) {
  v[2] = 42;
  println!("{}", v[3]);
}
```

Figure 3: Example code where a vector is passed and the target method changes the passed variable.

are declared as `mut` are mutable. Mutability plays (again) a larger role, when variables are passed to other functions. Such invoked functions cannot just decide to change a given variable. Instead, they must declare whether they require to mutate a variable.

Figure 3 illustrates and example, where a function `work_on_vector` is invoked with a vector as a parameter and where the target method changes the vector at index two (and prints out the vector at index three). Because of the change of the vector, the parameter `v` needs to be declared as mutable.

2.3 Combined Occurance of Mutability and Ownership

Although ownership rules and mutability do not seem to be closely related, there are frequent situations where both constructs need to be applied in combination. Such a situation is given when a client function owns a variable that needs to be changed by multiple functions. Actually, in such situation developers have the choice whether a passed variable should be passed as reference, or whether the invoked functions returns the changed variable. Figure 4 illustrates for an even more simplified example two alternatives how the implementation could look like.

- In the first alternative, the invoking function passes the ownership of the list to the function `remove_endings` that returns the ownership back. Therefore, it is necessary to declare the variable `vector` as mutable and to add an assignment operator to the client function in addition to a return statement in the target method. Additionally, it is necessary to declare the target method's return type `Vec<i32>` in its header. In the example, it is necessary to declare the list as mutable, because the returned value is passed to that variable. The alternative would have been to introduce a new variable.
- In the second alternative, the list is passed as a reference. Since the target method changes that reference, it is necessary to pass the list as mutable reference (parameter `&mut list`).

```
// Alternative 1:
//  passed ownership, returned vector
fn main() {
  let mut list = vec![1, 2, 3] ;
  list = remove_endings(list);
}

fn remove_endings(mut l: Vec<i32>)->Vec<i32> {
  l.remove(0); l.pop();
  l
}

// Alternative 2:
//  passed reference
fn main() {
  let mut list = vec![1, 2, 3] ;
  remove_endings(&mut list);
}

fn remove_endings(l: &mut Vec<i32>) {
  l.remove(0); l.pop();
}
```

Figure 4: Two alternative implementations of a client function that invokes a function that changes the passed list.

There might be reasons for or against one alternative, but from our perspective it is plausible that the second alternative should be preferred over the first one, because the header of `remove_endings` is probably easier to understand and it is neither necessary to assign the result of `remove_endings` to the variable `list` nor is it necessary to return the incoming parameter in `remove_endings`. However, we are aware that in the second alternative the invocation of the function is more difficult than in the first alternative.

2.4 Possible Effects on Developers

It is plausible that developers who use ownership rules and mutability for the first time (and who are used to side-effects), have problem with these concepts. However, this is from our perspective not problematic, because we think it is plausible that new technology requires some training.

However, we think that it is worth thinking about the potential problem that the design of programs is (probably) different than the design of programs in traditional programming languages with side-effects. For example, Figure 5 illustrates a simple example where a string is passed to a function and where the string (in addition to its length) is returned. The target function returns a tuple consisting of the string and its length, and the invoking method declares a new variable with corresponding names. Such a

```
1  fn main() {
2      let s1 = String::from("hello");
3      let (s2, len) = calculate_length(s1);
4      println!("The␣length␣of␣'{s2}'␣is␣{len}.");
5  }
6
7  fn calculate_length(s: String) -> (String, usize) {
8      let length = s.len();
9      (s, length) // returning tuple
10 }
```

Figure 5: Example code of a trivial function where a returned value is used in a method (taken with minor changes from https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html).

program looks noteworthy different than in traditional languages with side-effects where no second variable `s2` is required in the client method and where a target method is not required to return the incoming parameter in addition to result of the method's computation (the variable `length`). Considering that an additional identifier appears in the program (the identifier `s2`), it is plausible to assume that this decreases the readability of a program.

Additionally, we already discussed that mutability and ownership rules might lead to different program designs (see Figure 4) that probably differ in the readability of the code. As discussed, the design where the number of identifiers increases potentially decreases the readability of the code.

While we agree that Rust might imply a number of challenges for developers, there could be still a positive effect on code readability. We believe that the explicit passing of an object as mutable helps developers to identify more quickly where an undesired side-effect potentially occurs in a program. I.e., it is plausible to us that the additional annotation `&mut`–once it is in a valid piece of code–permits developers to identify more quickly potential sources of errors.

Hence, the present work's goal is to test, whether the previously said holds in a controlled setting.

3 RELATED WORK

In the following, we describe works that are related to the present one. We consider works as related, where empirical data was collected on the use or the usability either of the programming language Rust, or on the language construct mutability.

3.1 Studies on Rust

Astrauskas et al. studied developer code declared as unsafe in the programming language Rust (see (Astrauskas et al., 2020)) – a language feature that permits to write Rust code that does not rely on the typical static guarantees Rust give. The study is based on the collection of existing source code snippets. It turned out that – while the majority of code does not use the unsafe feature–that most often occurences of unsafe is in blocks where the motivation for such blocks "*result from need to provide interoperability with custom hardware and native code written in C*" (see (Astrauskas et al., 2020, p. 136:25)). A comparable study by Evans et al. (on the same code repository) came to comparable conclusions (see (Evans et al., 2020)).

A study by Zhu et al. concentrated on the learnability of Rust (see (Zhu et al., 2022)). In a first step, the authors inspected questions on Stack Overflow. Based on that, an survey with more than 100 Rust programmers was executed where four Rust programs where shown to each participant, who were asked questions about these programs (from identifying a program's error root cause up to subjective rating of the program's difficulty). One of the results of the study was that developers are at least sometimes confused about Rust's lifetime (or ownership) rules.

Zhang et al. studied 790 bug fixes of Rust programs (Zhang et al., 2024) in order to characterize common bugs and their fixes beyond memory safety concerns. Most interesting, the authors identified several index out-of-bounds exceptions, i.e., the error that is in the focus of the present work. However, the authors identified that only a small number of bugs were caused by violations of ownership rules.

A study by Coblenz et al. (Coblenz et al., 2023) used observations of students doing Rust tasks to classify Rust questions from Stack Overflow. It turned out that most questions are related to data structures, followed by libraries. A similar study by Chakraborty et al. (Chakraborty et al., 2021) (based on the analysis of question and answers of QA-sites) came to a comparable conclusion with respect to Rust.[4]

Scott et al. (Scott et al., 2024) studied on a single undergraduate student the possible effects of the programming language Rust for a given project (web server program). The study concluded that "memory-related features take a high learning curve" (Scott et al., 2024, p. 120).

Based on 178 interviews, the study by Fulton et al. (Fulton et al., 2021) tried to identify challenges developers face when adopting Rust. It turned out that while developers are more convinced to write bug-free programs, developers did not agree that Rust

[4]The mentioned study was not only restricted to Rust, but considered other languages as well.

requires less time to produce prototypes.

The study by Crichton and Krishnamurthi (Crichton and Krishnamurthi, 2024) focused on the more general process of learning a programming language. There, quizzes were added to the main textbook for Rust (Klabnik and Nichols, 2023). While several results of the study are related to how learners handle the content of the book, an interesting insight is that most readers only read the first chapters of a teaching book and then give up. It is noteworthy that the first chapters of the text book introduce ownership rules and references. Under the assumption that one of the reasons why learners give up early is, that the concepts are hard to understand, this study's results is in line with previous studies stating that developers consider ownership rules as hard to understand.

3.2 Studies on Mutability

While the previous studies were related to the Rust programming language, there are studies that explicitly focus on immutability / mutability.

Coblenz et al. (Coblenz et al., 2016) interviewed eight software engineers. It turned out that "*important requirements, such as expressing immutability constraints, were not reflected in features available in the languages participants used*."(Coblenz et al., 2016, p. 736), respectively, that "*the participants who worked on software with significant amounts of state said that incorrect state change was a major source of bugs*"(Coblenz et al., 2016, p. 742).

While a different study by Coblenz et al. (Coblenz et al., 2017) focused on a new language feature that permits the declaration of immutable state, it turned out that a number of developers using the programming language Java used the keyword final in a rather problematic way – which seems to suggest that the differences between final variables and immutable state is not as good understood by developers as one would expect.

4 EXPERIMENT

4.1 Initial Considerations

A number of initial considerations were required in order to design the experiment.

Task: The experiment's goal is to give developers a piece of code where an accidental side-effect leads to an error. As discussed in Section 2.4, the underlying idea is give participants code where functions operate on a previously defined data structure–and it is up to the participant to detect the function that accidentally changes the data structure. We decided to show one main method and eight additional functions to developers, where the name of the functions is simply `function_1` to `function_8` (in the following, we call the called functions API functions). That way, the developer can respond to the given task by pressing the keys 1–8.

Code Difficulty (Main Function): In order to determine where in a given code base a data structure is accidentally changed, it is necessary to read the code where such data structure was originally defined. Additionally, one must read the code where the data structure is potentially changed. The problem is that differences in difficulty in the code have an influence on how quickly developers are able to find the code that causes the error. Consequences, the goal is to design the experiment in a way, that as few as possible differences in the difficulty in the code exist–and that the code itself is relatively simple. We decided to design the code in the following way: there is the main function that defines the original data structure (a vector) that is changed by invoked functions. Such main method consists of 10 lines of code.

Code Difficulty (API Functions): Participants of the experiment need to read those functions that potentially change the incoming vector. Due to this, we see the need to design the code of such functions as homogeneous as possible to avoid confounding factors caused by such functions. The invoked functions only consist of declarations of new variables (that are initialized by a previous variable, the passed parameter, a clone of a previous variable or the passed parameter, or a new vector). The last line of the invoked functions is just the call `clear()` to one of the local variables. Each of the invoked methods has exactly 12 lines of code.

Distribution of Candidates and Error Function: Functions that potentially change the state of the vector (we call such functions candidate functions in the following) differ to the other functions that they require a mutable reference as incoming parameter. It is plausible that the more candidate functions are available, the more time the participant probably requires to identify the function that causes the error (in the following, we call this candidate function the error function). Furthermore, it is plausible that the position of the error function plays a role: if, for example, the error function is the first function to be read, there is no need to read any other function. Because of this, it is not only necessary to control the number of candidate functions: it is also necessary to control the position of the error function to some extent.

Position of Error Function: While it sounds

plausible that the position of the error function should randomly vary, we think that it is plausible that this variation should be controlled to a certain extent. We decided that the position of the error function among the candidate functions should be controlled by a relative position among the candidate functions. Thereto, we only distinguish whether the error function appears in the first or the second half of the candidate functions.

Language: Since the goal of the experiment is to test the effect of certain language features in Rust, there is a need to have another language to compare with. We decided to use Java as such a language. An implication of using a language that does not provide ownership rules and mutability as language constructs is that each called function is a candidate function. I.e., a main method in Java that matches the previously described form corresponds to a main method in Rust, where each method call passes its parameters as mutable references.

Finally, the experiment followed the general idea of N-of-1 studies (see (Hanenberg and Mehlhorn, 2021; Hanenberg et al., 2024)), where all treatment combinations are assigned to each single participant.

4.2 Experiment Layout

The experiment layout consisted of the following variables:

- **Dependent Variable:**
  - Time until an answer was given (response time).
- **Independent Variables:**
  - **Group:** Rust 2 (Rust with 2 candidate function), Rust 5, Rust 8, Java 8.
  - **Area:** 1st = the error function occurs in the first half of the candidate functions, 2nd = the error function occurs in the second half of the candidate functions.
- **Fixed Factors:**
  - **Main Function:** the main function consists of one line of code where a vector is defined, eight function calls that only pass this vector, and one line where a non-existing element was accessed. Figure 6 illustrates an example for Rust with two candidate functions, Figure 7 illustrates the translation to the treatment Java 8.
  - **API Functions:** Each API function receives a vector. In case, the API function is a candidate function, the parameter is declared as mutable. The API function's body consists of 10 lines where new local variables are introduced that are either initialized with a previous variable (including the incoming parameter), clone, or a new vector. The variable names were randomly chosen from a dictionary. The last two lines of each method are a new line and a line, where the vector of one variable is cleared. Figure 8 illustrates an API function that is a candidate function (incoming parameter declared as mutable) that is responsible for the error in the application (the last line cleares the incoming vector).
  - **Number of Repetitions:** Each treatment combination was repeated four times (32 tasks in total).
- **Random Factors:**
  - **Candidate Position in Function Call:** For the Rust code, the number of candidates were randomly set among the eight function calls. For Rust 8 and Java 8, no such random position was necessary (because each called function is

```
1  fn main ( ) {
2    let mut vector: Vec<i32 > =
3      vec! [32, 53, 24, 18, 56, 33, 58] ;
4    function_1(&mut vector);
5    function_2(&vector);
6    function_3(&vector);
7    function_4(&mut vector);
8    function_5(&vector);
9    function_6(&vector);
10   function_7(&vector);
11   function_8(&vector);
12   // ERROR: OUT OF BOUNDS
13   let vec_value_7 = vector[6];
14 }
```

Figure 6: Example code for treatment Rust 2. The line break in line 2 was only added for illustration purposes and was not contained in the original experiment.

```
1  public static void main(String[] args) {
2    var vector =
3      Vector.new(32 , 53, 24, 18, 56, 33, 58);
4    function_1(vector);
5    function_2(vector);
6    function_3(vector);
7    function_4(vector);
8    function_5(vector);
9    function_6(vector);
10   function_7(vector);
11   function_8(vector);
12   // ERROR: OUT OF BOUNDS
13   var vec_value_7 = vector.get(6);
14 }
```

Figure 7: Translation of Figure 6 to the treatment Java 8. The line break in line 2 was only added for illustration purposes and was not contained in the original experiment.

a candidate function).

– **Error Position in Error Function:** The line that is responsible in the error function–the assignment of the incoming parameter to a variable that is (directly or indirectly) cleared–was randomly chosen.

```
1  fn function_1 (v_start: &mut Vec<i32>) {
2    let &mut v_album = v_start;
3    let mut v_berry = v_album.clone() ;
4    let &mut v_crown = v_album;
5    let mut v_dance = vec![60, 84, 28, 37, 64, 63, 61] ;
6    let mut v_elbow = v_crown.clone( ) ;
7    let mut v_fruit = vec![75, 21, 2, 67, 3, 86, 91] ;
8    let &mut v_ghost = v_berry ;
9    let mut v_hear t = v_dance.clone () ;
10   let mut v_image = v_elbow.clone () ;
11   let mut v_judge = v_dance.clone () ;
12
13   v_crown.clear();
14 }
```

Figure 8: Example Rust candidate function that changes the state of the incoming parameter (the variable v_crown is defined as the incoming parameter in line 4).

4.3 Execution

The experiment was executed on ten volunteers based on purposive sampling. Each volunteer was a computer science student in the sixth semester or higher (age 22-26). The volunteers received a link to the online application that collected the data. The online application contained a short training phase including a description of the language constructs ownership rules, references, and mutability.

4.4 Results

An ANOVA was performed on the resulting data using the software Jamovi (version 2.3). The variable participant was additionally used in order to detect the possible influence of each individual on the results. Table 1 illustrates the results of the analysis.

It turns out that all independent variables were significant (with p <.001) and the effect size of all variables was large ($\eta_p^2 \geq .345$) where the variable participant had the lowest effect size ($\eta_p^2 = .345$). Additionally, there was a strong interaction effect between the variables group and area ($p < .001$; $\eta_p^2 = .123$).

Figure 9 illustrates this interaction effect. With an increasing number of candidate functions the differences between the first and the second area get larger. For eight candidate functions, there seems to be hardly a difference between Java and Rust. To test this, we ran a Tukey post-hoc test and found neither a difference between the first area for Rust 8 and Java 8 (p > .999), as well a no difference for the second area

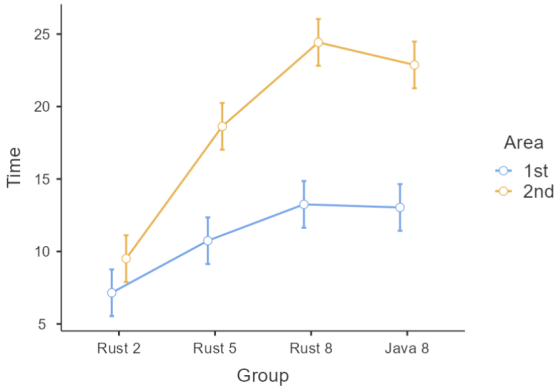

Figure 9: Estimated means for group * area.

(p=.881).[5]

Due to the strong effect of the variable participant (and its interaction with the variable area), it is plausible to take a closer look to the different participants.

Table 2 illustrates the results for each individual. It turns out that only three participants revealed the same effect as the results for all participants (variable group and area significant, significant interaction between both variables). However, taking a close look at the mean variables for the treatments R2, R5, R8, and J8 reveals that the same tendencies can be seen for all participants: the smaller the number of candidate functions, the faster is the response. Another notable result is that only three participant had a larger mean for Java 8 than for Rust 8 – there is some small tendency that the Java code was slightly easier to read than the Rust code with a comparable number of candidate functions.

To understand the main result of the study, Table 3 summarizes the ratios of Java response times and Rust response times. What can be seen is that the less candidate functions exist, the larger the ratio – and in case the number of candidate functions are comparable, there is even a (small) negative effect of Rust.

However, when considering Table 3 we need to keep in mind that the ratios also operate on averages of the variable area (from which we know that it has a strong and large effect on the result).

5 THREATS TO VALIDITY

Code Difficulty: The code in the experiment was designed in a way that it has (from our perspective)

[5]Actually, due to the strong overlapping confidence intervals for Rust 8 and Java 8, it is obvious that no difference could be found.

Table 1: Experiment results. Confidence intervals (CI) and means (M) are given in seconds; Treatments (TRT), respectively treatment combinations, are abbreviated to ease the readability of the table (R2 = Rust 2, etc.)

ANOVA on Response Times

Variable	df	F	p	η_p^2	TRT	$CI_{95\%}$	M
group (g)	3	67.84	<.001	.459	R2	7.68; 8.98	8.3
					R5	12.9; 16.5	14.7
					R8	16.9; 20.8	18.8
					J8	15.9; 20.0	18.0
area(a)	1	182.53	<.001	.432	1st	9.91; 12.2	11.0
					2nd	17.6; 20.1	18.9
participant(p)	9	14.07	<.001	.345	(see Table 2)		
g * a	3	11.26	<.001	.123	R2/1st	6.5; 7.8	7.15
					R2/2nd	8.5; 10.5	9.51
					R5/1st	8.1; 13.4	10.7
					R5/2nd	16.8; 20.4	18.6
					R8/1st	11.2; 15.4	13.3
					R8/2nd	22.2; 26.6	24.4
					J8/1st	10.3; 15.8	13.0
					J8/2nd	20.7; 25.1	22.9
g * p	27	1.10	.338	.110	skipped, because not significant		
a * p	9	4.82	<.001	.153	(see Table 2)		
g*a*p	27	1.45	.076	.140	skipped, because not significant		

Table 2: Results per participant. The columns g, a, and g * a show the p-value for each individual as well as the means for the treatments R2, R5, R8, and J8. The highlighted participants show significant results for all independent variables.

ANOVA on each participant

p	g	a	g * a	R2	R5	R8	J8
p1	.003	.613	.648	9.98	18.4	27.0	25.4
p2	.001	<.001	.061	7.62	12.7	19.7	18.8
p3	.004	<.001	.037	8.13	14.0	15.3	14.7
p4	.004	.005	.051	11.0	24.7	25.9	22.0
p5	<.001	<.001	.003	8.26	14.2	19.4	16.7
p6	<.001	<.001	.113	7.25	12.2	16.0	16.7
p7	.008	<.001	.081	10.1	14.9	20.5	19.9
p8	<.001	<.001	.145	7.82	13.3	14.3	14.6
p9	<.001	<.001	.165	5.15	8.61	12.7	12.3
p10	<.001	<.001	<.001	7.48	14.0	17.5	18.4

Table 3: Ratios of response times of Java (with eight candidates) and Rust. The last column describes the ratio of Java response times in comparison to average response times in Rust.

$\frac{M_{Java8}}{M_{Rust2}}$	$\frac{M_{Java8}}{M_{Rust5}}$	$\frac{M_{Java8}}{M_{Rust8}}$	$\frac{M_{Java8}}{\emptyset M_{Rust}}$
2.17	1.22	.957	1.291

a low difficulty. The main function only consisted of a vector definition and function calls, while the called fuctions just consisted of the definition of local variables, that were either initialized with other local variables, or which defined a new vector. We believe that the results of the experiment would be quite different if more difficult code would be used, because more difficult code (probably) introduces a larger deviation in response times. It is also possible that very difficult code–such as algorithmic code–would hide the here measured positive effect of ownership rules and mutability. However, this would be from our perspective not mean that the effect is not there. Instead, the effect would just be hidden.

Difficulty of Candidate Functions: We are aware that the measured differences depend to some extent on the difficulty: it is plausible that the measured differences occur not only because of the mutable declaration of a passed parameter, but also because of the time spent in each candidate function. Hence, increasing the time spent on each candidate function would probably increase the effect measured in the present experiment. However, such a change in the experiment would also require that the difficulty of each candidate function is comparable.

Flat Code Structure: The present experiment relies on code with a flat structure: a main function invokes some functions that work on the passed parameter–but those ones do not pass the parameter to other functions. We are aware that a change in the structure of the code (which probably goes in line with an increase in the code's difficulty) will probably change the experiment results.

Variable Names and Types: The code in the experiment used random words from a dictionary for parameter names and we are aware that the identifier style has an effect on the readability of the code (see, for example (Binkley et al., 2013; Lawrie et al., 2006) among others). Furthermore, the code uses no type declarations for the local variables (we use the keyword `var` in the Java code, in Rust, we do

not explicitly define the type of the variables in each candidate function), and we are aware that type information influence the readability of the code (see, for example, (Gannon, 1977; Prechelt and Tichy, 1998; Endrikat et al., 2014; Fischer and Hanenberg, 2015; Ray et al., 2017) among others).

No Unnecessary Mutable Declarations: The experiment relied on the assumption that developers use mutable declarations only in situations, where a function potentially changes a variable's state. In case, one would always pass parameters as mutable (with references), one would (probably) not measure a difference between Java and Rust. The experiment already gave evidence for that, because the experiment did not reveal a difference between the treatments Rust and Java with eight candidate functions.

Unsafe Rust: The code in the experiment assumes that Rust is used in a safe manner. We are aware that it is common to use Rust by using code blocks declared as `unsafe`. Actually, we are not aware how unsafe code would change the results of the experiment.

Error Identification Versus Error Fixing: In the present experiment, participants were only asked to identify the error function–but it was not up to them to fix the error. Actually, fixing an error in a completely randomized experiment is (probably) quite difficult: for the present experiment, a strategy could be to add a method call `clone()` to all variables, another strategy could be to simply delete all function calls in the main function–both strategies are obviously not the intention of such an experiment. We think in order to design a randomized experiment on error fixing, one has to articulate clearly what kinds of changes are permitted (and which ones are not). Hence, we think that it is possible that results for error fixing experiments can differ from the here presented results–and this might also be partly because of required differences in the experiment design.

Reading Direction: The experiment introduced the variable area, which indicated that it leads to smaller response times in case the error function appears in the first half of the functions. However, this probably depends on the reading direction of the code: probably, participants read the code from top to bottom–which inherently means (under the assumption that they spent a comparable amount of time in each candidate function) that the sooner the error function appears, the quicker is a participant able to answer. However, this probably changes as soon as participants change their reading direction. I.e., we think that the variable area mainly indicates the reading direction and in case most participants read the code from bottom to top, the effect of that variable would be the opposite.

Measurement Technique: The proposed experiment uses the response times of participants. We are aware that this measurement has a potential problem: participants could just guess what the function is, or could just type in all possible answers (until the correct one is found). So far, we do not have a solution to this potential problem. Actually, we think that this is a common problem today's controlled experiments potentially suffer from.

Influence of Used Programming Languages: The present study relies on the languages Rust and Java, where the latter one was considered as a language without the constructs to be studied. We cannot judge to what extent the used languages influence the result of the study. For example, it is plausible to us that the syntax of Rust was (under the experimental conditions) relatively easy to understand, because in principle one just needs to distinguish parameters that were passed as `&mut`. However, whether the syntax of this construct has other effects cannot be judged from the experiment.

Generalizability: It is unclear to what extent the results are generalizable. The most general problem is, to what extent the participants can be considered as being representative. We cannot answer this question, but we assume that participants who are more familiar with Rust probably require less time than participants who are not familiar with reading code in general. In our case, the participants were students who are familiar with reading code. Another general threat comes from the relatively small sample size: the experiment was just executed on 10 participants. However, we should emphasize that the main effect (significance of variable group) was shown for each single participant. I.e., the experiment does not give much evidence to doubt that under the experimental conditions no positive effect is measured. Furthermore, one should not forget, that the design of N-of-1 studies explicitly focuses on a small number of participants–while the number of data points is (still) quite high (see, for example, (Hanenberg and Mehlhorn, 2021; Hanenberg et al., 2023; Klanten et al., 2024)).

6 SUMMARY AND DISCUSSION

The present work introduces an experiment with the focus on the language features ownership and mutability that are provided by the programming language Rust: features that are commonly considered as one of the most innovative elements of Rust. Both features in combination are used in

situations where side-effects occur in a program–and both features in combination guarantee that no undesired side-effects occur.

While the theoretical aspects of both features are unquestioned, the present experiment has the focus on the readability of the resulting code. Thereto, an experiment was designed where a given data structure was passed to eight functions, where one of the functions performs an undesired side-effect. The result of this undesired side-effect is that one line in the invoking method is invalid.

We summarize the present work in different steps: first, we summarize and discuss the experiment design, then we summarize and discuss the results, followed by the interpretation. Finally, we discuss the possible implications for future work.

6.1 Experiment Design

To compare the effect of language features with a language without such features, the programming language Java was used for the comparison. However, based on some initial considerations, such an experiment should also consider how many candidate functions possibly cause an undesired side-effect. Therefore, the experiment introduced an independent variable group with different numbers of candidate functions: two, five, and eight candidate functions for Rust, and eight candidate functions for Java. In principle, it would have been desirable to use the number of candidate function as independent variable as well as the programming language as an independent variable. However, this is not possible under the assumption that the rest of the code is comparable, because Java does not permit to distinguish between mutable and non-mutable parameters.

Additionally, the experiment introduced a variable area which defined in what half of the candidate functions the error function occurred. Again, it can be argued that it would have been desirable to introduce a variable that states, what invoked function leads to the error (with the treatments 1–8). However, combining such a variable with the variable group would lead to a large increase of treatment combinations and the insights would be (from our perspective) rather low. Hence, the variable area should only be considered as an indicator that not only the number of candidate functions matter, but also, where the call of the error function occurs.

6.2 Results

The result of the experiment is quite clear: the lower the number of candidate functions, the larger is the difference in the response times in the experiment: the variable group influences the response times (with $p < .001$; η_p^2=.459), and so does the interaction group * area ($p < .001$, $\eta_p^2 = .123$). This implies for the comparison to Java, that there is a large difference between Java and Rust that has only 2 candidate functions ($\frac{M_{Java8}}{M_{Rust2}} = 2.17$), while this ratio decreases, if the number of candidate functions increases (up to $\frac{M_{Java8}}{M_{Rust8}} = .957$)[6]. On average, participants required 29% more time in Java in comparison to Rust ($\frac{M_{Java8}}{\emptyset M_{Rust}}$=1.291).

It is important to emphasize not only the effect of the variable group, but also the effect of the variable area. I.e., when speaking about the readability effect one must take into account where the error function is (among the list of candidate functions).

However, the effects were only observable on all participants in combination. On an individual basis, only three out of ten participants revealed a significant effect of group, area, and group * area. However, for each individual participant a positive effect of Rust was measured, although the differences between the participants were remarkable: while, for example, for participant p3 the ratio $\frac{M_{Java8}}{M_{Rust2}}$ was just 1.8, participant p10 revealed a ratio of 2.46.

Although not all effects could be observed on each participant, we think that this might be due to the relatively low number of repetitions: the experiment just used four repetitions for each treatment combination.

6.3 Interpretation

The present experiment should and must not be understood as a general proof that Rust increases the readability of code (in comparison to Java). Instead, the experiment only focuses on a situation where somewhere in a program some undesired side-effect occurs that leads to an error. We tested (in a controlled setting) whether the passing of a variable as a mutable reference has a positive effect on the identification of a function that changes the passed parameter. In such situation, these additional annotations turned out to be beneficial for participants–and the effect (with $\frac{M_{Java8}}{\emptyset M_{Rust}}$=1.291) is large enough that it can be hardly argued that this effect is not relevant.

The interesting part of the work is, that

[6]This difference in response times was not significant.

the language constructs ownership and mutability–although especially the first one is considered as hard to learn–can help developers to identify problematic situations in the code. I.e., a technical construct that was mainly designed to increase memory safety can even have a positive effect on developers. However, we should keep in mind that the code for the experiment was already given. I.e., it is not possible to infer from the experiment, what the effect of such language constructs on the construction of new code is–and it should remind us, that studies indicate that ownership rules are probably hard to learn (see Section 3).

6.4 Implications for Future Work

The general setup of the experiment–the random generation of code snippets–shows, that it is possible to study the readability of constructs in a controlled setting. Such approach could be used to study more constructs–and this approach could be used to study language constructs upfront, i.e., before such constructs are added to a language. This permits to check upfront, whether a language construct implies problems for developers. I.e., we think that the approach of generating random code and give it participants to read to identify certain phenomena could be a general approach for language design. However, we should make explicit that we do not think the readability of ownership and mutability is solved by the present work–we only consider this as a starting point. Further experiments could, for example, check, to what extend mixtures of passed ownerships, references and mutability potentially confuse developers. Additionally, we think is it reasonable to use the same approach to identify errors in existing code (such as a wrong use of ownership rules).

In the same way, it seems plausible that syntax could be studied in more detail. For the given scenario, we think it is plausible to test whether a syntax such as the phrase `&mut` is a good design choice. This could help language developers in the future to test even syntax constructs upfront.

Having said this, we think that future work should also emphasize the construction of code. I.e., instead of giving developers a piece of code to read (to identify some phenomenon), one should give participants concrete code snippets to write. However, we are aware that such an experiment will probably largely differ from the present one.

7 CONCLUSION

The present work studied the effect of ownership rules and mutability in Rust. In a controlled setting, participants were asked to identify a function that changes a program's state in an undesired way. The result was that ownership rules and mutability helped participants to identify a problematic function faster in comparison to the situation, where it cannot be judged from a passed parameter, whether it potentially changes the program state.

However, we should emphasize that the experiment just tests the readability of Rust code in a concrete setting with pre-defined code. I.e., we cannot judge from the experiment what the effect of such constructs on the design of new code is.

REFERENCES

Astrauskas, V., Matheja, C., Poli, F., Müller, P., and Summers, A. J. (2020). How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA).

Binkley, D. W., Davis, M., Lawrie, D. J., Maletic, J. I., Morrell, C., and Sharif, B. (2013). The impact of identifier style on effort and comprehension. *Empir. Softw. Eng.*, 18(2):219–276.

Buse, R. P., Sadowski, C., and Weimer, W. (2011). Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 643–656, New York, NY, USA. Association for Computing Machinery.

Chakraborty, P., Shahriyar, R., Iqbal, A., and Uddin, G. (2021). How do developers discuss and support new programming languages in technical q&a site? an empirical study of go, swift, and rust in stack overflow. *Inf. Softw. Technol.*, 137:106603.

Clarke, D., Östlund, J., Sergey, I., and Wrigstad, T. (2013). *Ownership Types: A Survey*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg.

Coblenz, M., Nelson, W., Aldrich, J., Myers, B., and Sunshine, J. (2017). Glacier: Transitive class immutability for java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 496–506.

Coblenz, M., Porter, A., Das, V., Nallagorla, T., and Hicks, M. (2023). A Multimodal Study of Challenges Using Rust.

Coblenz, M., Sunshine, J., Aldrich, J., Myers, B., Weber, S., and Shull, F. (2016). Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 736–747, New York, NY, USA. Association for Computing Machinery.

Crichton, W. and Krishnamurthi, S. (2024). Profiling programming language learning. *Proc. ACM Program. Lang.*, 8(OOPSLA1).

Endrikat, S., Hanenberg, S., Robbes, R., and Stefik, A. (2014). 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. ACM.

Evans, A. N., Campbell, B., and Soffa, M. L. (2020). Is rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 246–257, New York, NY, USA. Association for Computing Machinery.

Fischer, L. and Hanenberg, S. (2015). An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. *SIGPLAN Not.*, 51(2):154–167.

Fulton, K. R., Chan, A., Votipka, D., Hicks, M., and Mazurek, M. L. (2021). Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Proceedings of the Seventeenth USENIX Conference on Usable Privacy and Security*, SOUPS'21, USA. USENIX Association.

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

Hanenberg, S. and Mehlhorn, N. (2021). Two N-of-1 self-trials on readability differences between anonymous inner classes (AICs) and lambda expressions (LEs) on Java code snippets. *Empirical Software Engineering*, 27(2):33.

Hanenberg, S., Morzeck, J., and Gruhn, V. (2024). Indentation and reading time: a randomized control trial on the differences between generated indented and non-indented if-statements. *Empir. Softw. Eng.*, 29(5):134.

Hanenberg, S., Morzeck, J., Werger, O., Gries, S., and Gruhn, V. (2023). Indentation and reading time: A controlled experiment on the differences between generated indented and non-indented JSON objects. In Fill, H., Mayo, F. J. D., van Sinderen, M., and Maciaszek, L. A., editors, *Software Technologies - 18th International Conference, ICSOFT 2023, Rome, Italy, July 10-12, 2023, Revised Selected Papers*, volume 2104 of *Communications in*, pages 50–75. Springer.

Klabnik, S. and Nichols, C. (2023). *The Rust Programming Language, 2nd Edition*. No Starch Press.

Klanten, K., Hanenberg, S., Gries, S., and Gruhn, V. (2024). Readability of domain-specific languages: A controlled experiment comparing (declarative) inference rules with (imperative) java source code in programming language design. In Fill, H., Mayo, F. J. D., van Sinderen, M., and Maciaszek, L. A., editors, *Proceedings of the 19th International Conference on Software Technologies, ICSOFT 2024, Dijon, France, July 8-10, 2024*, pages 492–503. SCITEPRESS.

Ko, A. J., Latoza, T. D., and Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Softw. Engg.*, 20(1):110–141.

Lawrie, D. J., Morrell, C., Feild, H., and Binkley, D. W. (2006). What's in a name? A study of identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 3–12. IEEE Computer Society.

Prechelt, L. and Tichy, W. F. (1998). A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312.

Ray, B., Posnett, D., Devanbu, P., and Filkov, V. (2017). A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100.

Scott, J., Zuo, F., and Rhee, J. (2024). Student-perspective observations from the comparison of rust and c++ languages. *J. Comput. Sci. Coll.*, 40(1):112–121.

Zhang, C., Feng, Y., Zhang, Y., Dai, Y., and Xu, B. (2024). Beyond memory safety: an empirical study on bugs and fixes of rust programs. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, pages 272–283.

Zhu, S., Zhang, Z., Qin, B., Xiong, A., and Song, L. (2022). Learning and programming challenges of rust: a mixed-methods study. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1269–1281, New York, NY, USA. Association for Computing Machinery.