UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

An Empirical Study of Programming Languages in Open-Source Software Projects based on Mining Software Repositories

by

Muna Altherwi

Thesis for the degree of Doctor of Philosophy

27th August, 2021

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
Electronics and Computer Science

Doctor of Philosophy

AN EMPIRICAL STUDY OF PROGRAMMING LANGUAGES IN OPEN-SOURCE SOFTWARE PROJECTS BASED ON MINING SOFTWARE REPOSITORIES

by Muna Altherwi

There are dozens of programming languages in use today, and new languages and language features are being introduced frequently. However, there are only a few empirical studies on the usage and practice of programming languages. In this research we explored languages from an empirical/pragmatic perspective to address their association with open-source software (OSS) projects and practices. The research was conducted in a comparative setting to investigate whether a significant association exists. That is, a comparison was made between languages both individually and in groups to understand similarities and examine differences, if any, in popularity and user adoption, feature usage, and OSS project attributes. The methodology was based on mining software repositories, and the results obtained from an analysis of possibly the largest open-source dataset (a sample of 5,350 projects from a total of 15,000 projects), where a main language was identified. The investigation revealed that a considerable association exists; however, the effect size of such association was modest. When accounting for confounding factors such as project size and type, the findings held only in a small number of the tested cases. Thus, the choice of language has a limited effect on OSS development.

Declaration of Authorship

I, Muna Altherwi , declare that the thesis entitled *An Empirical Study of Programming Languages in Open-Source Software Projects based on Mining Software Repositories* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as:
  - Muna Altherwi, Andy Gravell, "A Large-Scale Dataset of Popular Open Source Projects," Journal of Computers vol. 14, no. 4, pp. 240-246, 2019.
  - Muna Altherwi, Andy Gravell, "Assessing Programming Language Impact on Software Development Productivity Based on Mining OSS Repositories". ACM SIGSOFT Software Engineering Notes 44, 1 (March2019), 36-37.

Signed: Muna Altherwi

Date: 28-02-2021

Acknowledgements

Thanks to God for giving me the strength and blessing me to complete the PhD journey. I would like to take the opportunity to thank everyone who has helped me get through this challenging experience, starting with my dear family: Mama, Aunty Noufo, Amani, Raaed, and Abdulwahab, who has been immensely supportive and is always present at any hour to calm me down or cheer me up despite the significant time difference. A very special thank you goes to my husband, Abdullah, for his unconditional love and support all the way. Words can never express how grateful I am that he is always there for me.

Certainly, a huge thank you goes to my supervisor – Dr Andy Gravell, who has guided me through every step of this research with great patience and support and without whose expertise and knowledge, the thesis would definitely be lacking. I would like to also send my thanks to King Abdulaziz University and Jeddah University for granting me this opportunity and sponsoring my education.

Finally, I would like to thank all my friends who have stood by me and given me the strength and advice that helped me reach this point in my academic life. I thank God for bringing into my life such people, who have become a crucial part of this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming language debates are often subjective and inconclusive. Language designers and early adopters make different claims about their languages to differentiate them from others and to attract users. Unfortunately, a number of such claims is based on personal affinity and not supported by strong evidence. Claims are more overstated in modern languages than in earlier ones (Markstrum 2010).

Research on programming language links to software projects has revealed a divide on whether the choice of language has a significant effect on software development. Some studies stated that programming languages do not have a considerable effect on software development, performance, productivity, and practical programming, and that there is no hard evidence to support such claims (Boehm 1981, Wulf 1980). They expressed that the impact of languages on software is rather limited and subtle in terms of program's readability and error propensity and that choice of language is not significant in developing software projects. However, a number of empirical studies have shown that the choice of language has a considerable effect on software development. Early studies (Schneider 1978, Harrison & Adrangi 1986) in this regard revealed strong differences between low-level and high-level languages, and no comparisons were carried within the group of high-level ones due to limited sample size. Later attempts (from the 1990s onwards) on the high-level group were preliminary, their results were sometimes contradictory, and some were based on small-scale experiments (Phipps 1999, Bhattacharya & Neamtiu 2011*b*, Myrtveit & Stensrud 2008, Nanz & Furia 2015, Ray et al. 2014, Berger et al. 2019*a*). Thus, studies on programming languages from an empirical/pragmatic perspective are needed to provide supportive evidence and objective comparisons among them.

The overall goal of this research is to empirically compare a range of modern, popular, and high-level programming languages to inspect how much such languages differ from one another in open-source software development projects and practices. More importantly, this research will investigate whether an association between language and OSS

projects and practices exists in the dataset, the significance of such association, and its magnitude. This investigation is carried out on a large-scale setting based on mining software repositories.

1.1 Research Motivation

Why an empirical comparison between programming languages?
Since the appearance of modern computers, progress has been made in designing various programming languages. Nowadays, there are dozens of programming languages in use, and new languages and/or language features are being introduced continuously. The nature of such languages as *special software tools* makes it difficult to find measures to draw objective conclusions about them. Moreover, language designers, advocates, and early adopters make different claims to differentiate them from others and to attract users. Unfortunately, a number of such claims are based on personal affinity and are not supported by hard evidence. Claims are more overstated in modern languages than in earlier ones (Markstrum 2010). Thus, one approach that can be used to provide objective information about languages is *empirical comparison*. Looking into languages from an empirical perspective would provide supportive evidence and valuable conclusions about them.

Why Open-Source Software?
Open-source software (OSS) has successfully made it to the mainstream and obtained increasing popularity over time. The growth rate of the number of OSS projects has increased exponentially in the period of 1993-2007 (Deshpande & Riehle 2008). Furthermore, OSS has been embraced by the vast majority (91%) of enterprise, according to a 2016 survey (Zenoss Inc 2016) of differently sized companies worldwide. This considerable interest in OSS, along with the availability of these projects data on online code hosting platforms that allows gathering of a reasonably large sample size attracted attention to consider OSS projects for this research.

Why mining software repositories?
Mining software repositories (MSR) for uncovering patterns and discovering findings about the artifact and the delivery process have gained importance as a research area during the last decade. In 2004, a specialized conference[1]. on mining software repositories evolved from the premium International Conference On Software Engineering (ICSE) [2], in recognition of the importance and potential of this field. This active research area has utilized the availability of project data along with data mining tools and techniques to analyse and understand software projects. Moreover, it provides an opportunity to build large-scale datasets of selected, high quality, real project data for research purposes. For

[1] "Mining Software Repositories." Available: http://www.msrconf.org/
[2] "International Conference on Software Engineering (ICSE)" Available: http://www.icse-conferences.org/

instance, Github, a widely used software hosting platform, hosts some prominent open-source projects, such as Linux kernel, Ruby on Rails, and JQuery. In addition to the availability of source codes, it also offers a wealth of data related to the software artifact and the development process, making it a valuable resource for researchers.

1.2 Research aims

The overall goal of this work is to address the association between general-purpose, high-level programming languages and OSS projects and practices, if any. In particular, there is an interest in comparing languages individually and in groups based on their design using an empirical methodology that is based on data mining. The general aims of this research are summarised in the following points:

- To investigate current trends, directions, and practices in software development and programming languages.
- To investigate state-of-the-art practice of language features.
- To investigate programming language relationships with classic software development aspects.

1.3 Thesis structure

The remainder of this thesis is structured as follows:

- Chapter 2 provides the background needed to understand the work done in this thesis.
- Chapter 3 covers the research objectives, the methodology used for achieving the research objectives, the data collection process.
- Chapter 4 describes the first study in this research area current trends and directions in programming languages. In this chapter the aim is to identify popular languages in developing OSS; compare their usage; and observe trends and patterns in language use.
- Chapter 5 covers the second study; which investigates how language features are used in practice and whether there is a significant association between language design and feature usage.
- Chapter 6 covers the third study in this thesis that investigates the relation of general-purpose programming languages and open-source software development.

- Finally, in Chapter 7 the limitations of this research are provided along with the contributions and the conclusions.

Chapter 2

Background and related work

The purpose of this chapter is to give background for the presented research. Section 2.1 provides information about software development, in particular, the open-source software development covered in Section 2.1.1. Then, Section 2.2 covers the related work that investigated the relationship between software development and languages. In Section 2.3 the differences between some programming languages categories are briefly discussed, and milestones in the time-line of programming languages are presented in Section 2.4.

2.1 Software development

IT industry has made significant progress over the years, growing from non-existence early last century to a multi-billion pounds industry nowadays. In the early days of the industry (1950s-1960s), the cost of hardware exceeded the cost of software. This gradually changed during the 1970s when an increase in software costs started to dominate the scene as demonstrated in Figure 2.1. Since then the need to improve software development practices and the impact of such enhancements has been recognized (Boehm 2006). According to Boehm, a 20% improvement in software development would worth $90 billion worldwide, as of the last decade of the twentieth century (Boehm 1987).

Software development refers to tasks associated with planning, designing, building, and maintaining software. In the beginning, software projects were smaller in size and assembly was the language of writing programs. Back then, programming effort used to make about 80% of the total development cost (Jones & Bonsignour 2011) due to the difficulty of writing programs in assembly. Basic assembly is one level higher than machine code and the mapping between code statement and machine instruction is about one-to-one, making programming effort quite high. Over the time, flourishing of high-level languages along with advances in the software industry, have managed to limit the programming effort to about 20% to 30% of the total development cost (Jones & Bonsignour 2011).

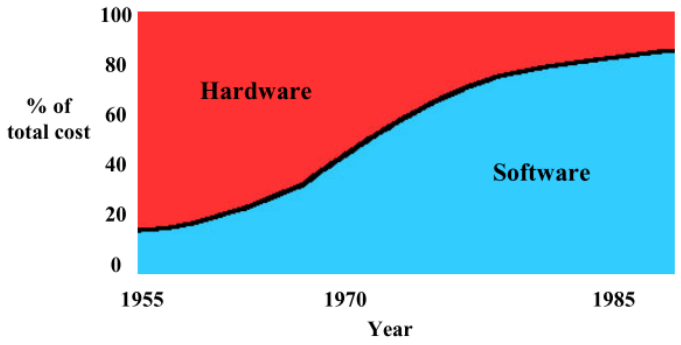

Figure 2.1: Hardware-software cost trends in large organizations (1973 prediction). Reference: Boehm (2006)

2.1.1 Open-Source Software development

Software can be closed-source, also called proprietary software, in which the use, re-distribution, and alteration of the software or its source code is protected by copyrights and controlled by the vendors. On the other hand, in the open-source software, the source code is available and the publication is licensed. That is, the use, modify, and re-distribute is permitted to anyone. Over the years, OSS has made it to the mainstream and obtained increasing popularity. An exponential growth of OSS between 1993 and 2008 has been depicted in Figure 2.2.

OSS projects has a long history and managed to gain wide popularity over the time. Prominent examples of such software include Linux operating system, Apache web server, Python programming language, and Mozilla web browser. In this open setting, a community of developers and users can collaborate to build, extend, and maintain software. This development paradigm is different from the proprietary one, where such processes are appointed to specific teams of developers (Ming-Wei Wu et al. 2001). In 1999, a milestone in the history of OSS was reached when Eric Raymond identified and differentiated the two development models (Raymond 1999). Raymond compared the OSS development model to a bazaar, where everyone can participate in order to produce a quality software, as opposed to the cathedral-builder style, where a specific group of experts is responsible for creating and maintaining the software. This work influenced Netscape to join the movement and make its Communicator suite open source.

2.2 Programming languages and software development

Empirical comparisons have been conducted with a range of programming languages to investigate whether there exist significant differences among them. More importantly, whether such differences have an effect on software development and its related aspects.

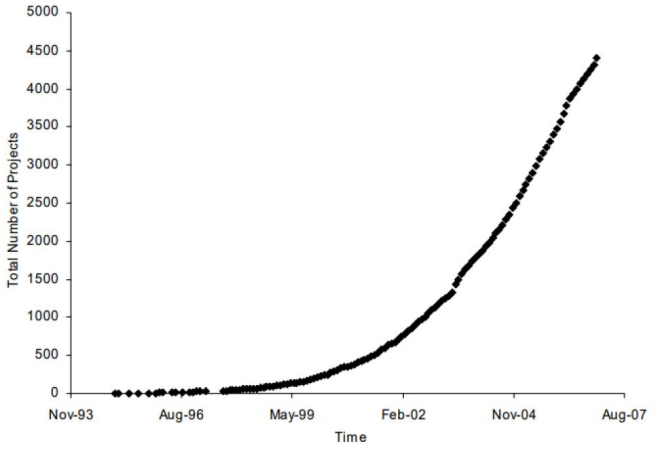

Figure 2.2: Graph showing the total number of open-source projects from November of 1993 through August of 2007. Source: Deshpande & Riehle (2008).

There is a disagreement among the studies in the body of literature as to whether programming languages are similar in their effect on software development, performance, and practical programming.

Some researchers have stated that programming languages do not have a considerable effect on software development, performance or practical programming, and that there is no hard evidence to support such claims. They claimed that the effect of languages on software development is rather limited and subtle in terms of a program's readability and error propensity (Wulf 1980). Boehm's empirical experiment in 1981 examined the influence of a programming language on the success of software projects (Boehm 1981). The study concluded that the choice of language was not significant in developing small projects. However, the experiment was focused on *programming in the small*. It was carried out with two small groups of students developing the same project in two high-level languages (Fortran and Pascal). Other studies stated that languages coupled with specific programming style, rather than other languages, could affect productivity and efficiency, (Port & McArthur 1999). Another study on a software development productivity of C and C++ concluded that there is no empirical evidence of differences between them (Myrtveit & Stensrud 2008).

Conversely, other cost models considered programming languages to be a factor affecting software development costs. Thus, the effect of languages on software development have been recognized by the following cost models: SDC (Nelson 1967), Putnam SLIM (Putnam 1978), and SOFCOST (Dircks 1981), whereas languages were excluded by others such as COCOMO (Boehm et al. 1995), Jensen (Jensen 1983), and TRW. One of the reasons for this exclusion can be the 'non-quantitative' nature of programming languages (Harrison & Adrangi 1986).

In addition, a number of studies have found some associations between languages and software development. A study by Harrison and Adrangi (1986) investigated the role

of programming languages in software development costs with an analysis of 279 Department of Defence software projects that used Assembly, COBOL, FORTRAN, and PL/1. Their findings demonstrated that any of the high-level languages will reduce the amount of effort required to develop a given size of project when compared to assembly language. However, due to the lack of sample sizes for COBOL and PL/1 projects, they could not compare the high-level languages with each other because it would have produced misleading results (Harrison & Adrangi 1986).

Such findings go along with the outcome from Schneider (1978), where the development effort of high-level and low-level languages were compared and the results indicated that high-level languages are roughly twice as efficient as low-level languages. Again, no comparison was done on the high-level languages. Another study compared defect density and programmer productivity in Java and C++ and showed that Java had two to three times fewer bugs per line of code than C++, about 15% to 50% fewer defects per line, and was about six times faster to debug (Phipps 1999). However, when defect density was measured (defects against development time) it showed no differences between the two languages. This experiment was carried out on a small scale. However, it is essential to consider that the style of programming and the programmer's proficiency can considerably affect the performance of small scale applications. Moreover, it has been found that variability of performance among programmers of the same language is greater than the variability among the different languages (Prechelt 2000).

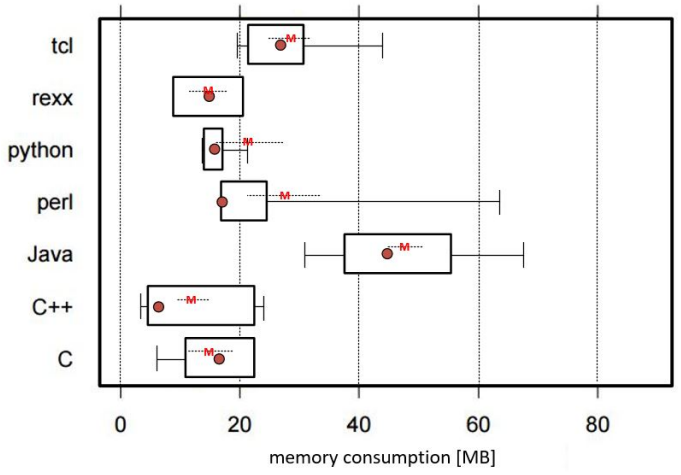

Figure 2.3: Program memory in Mbytes. Reference: Prechelt (2000)

Another study (Bhattacharya & Neamtiu 2011*a*) assessed the impact of language on software development and maintenance in an investigation of the differences between C and C++. The researchers concluded that C++ is better in software quality and effort. Their findings contradict the findings of Myrtveit & Stensrud (2008), who concluded that there was no empirical evidence of differences between C and C++ in terms of development effort and that there was no superiority of C++ over C. An experiment examining on Java and C++ showed that Java was about 30% to 200% more productive than C++

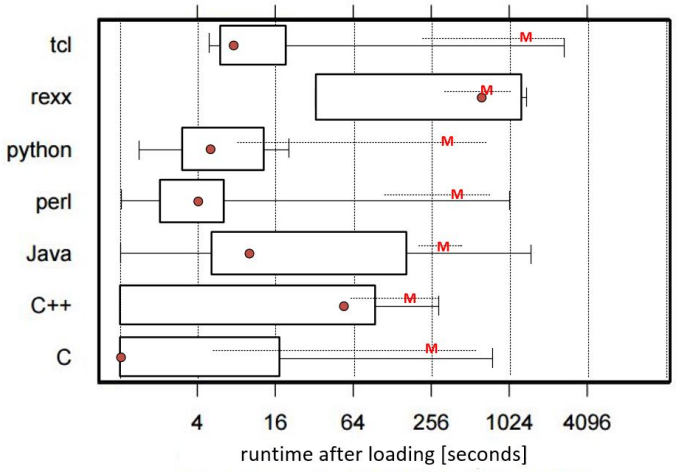

Figure 2.4: Program runtime in seconds. Reference: Prechelt (2000)

(Phipps 1999). While such attempts are valuable, the ability to draw conclusive findings are limited because these studies are based on comparisons of only a couple of languages.

A relatively large empirical comparison (Prechelt 2000) of seven programming languages that implemented the same set of requirements showed that languages have different effects on software performance in terms of memory consumption (Figure 2.3), execution speed (Figure 2.4), and on the design and writing of programs. However, no differences were found between languages in terms of program reliability. A large-scale study examining languages and software development productivity was carried out on about 500 languages and showed that different languages produced different productivity levels (Jones 1996). Nonetheless, the research stated that their findings were preliminary and noted that the error margin in the results can be quite high. Another study of nine languages aimed at determining if there is evidence of an effect on software development concluded that the choice of language was a significant factor in writing programs and that developers' productivity rates are not constant among different languages (Delorey et al. 2007).

An additional study (Lavazza et al. 2016) exploring the effect of 11 programming languages on development effort confirmed the findings of Jones (1996) and Delorey et al. (2007) that every language has its own rate. However, the study's results showed effort levels that were higher than those of Jones' (1996) and Delorey et al. (2007) for the same languages. A further work from ISBSG of comparing a database of 6,000 projects in terms of development hours per function point of software demonstrated that the choice of language had a significant effect on development schedule, as can be seen in Table 2.1 (*Software Project Benchmarking - ISBSG* 2012). A large-scale 2014 study by Ray et al. has investigated 729 projects in 17 languages for the effect of programming language on code quality found that languages can significantly effect quality, however, the effect size was modest (Ray et al. 2014). The study was replicated in 2019 by Berger et al. (2019*a*) who first reproduced the findings with the same 729 projects and following the same

methodology and analysis, resulting in a partially successful replication. Then, they re-analysed the included projects but followed different methodologies for data processing and statical analysis than the original study. As a result, a smaller number of projects were included (423 projects), and most of the claims did not hold. In addition, for the cases, where the claims did hold, the relationship between programming language and defects were found to be exceedingly small in effect size. Another large-scale study (Nanz & Furia 2015) based on mining 7,087 programs in eight languages inspected conciseness, performance, and failure proneness as language features. They found that a language's paradigms affected conciseness differently, with functional and scripting languages performing better than procedural and object-oriented languages. In performance, in terms of running time, C was the best language on large inputs, followed by Go. Procedural languages were more efficient with memory usage than languages from other paradigms.

Language	Hours per Function Point
Classic ASP	06.1
Visual Basic	08.5
Java	10.6
SQL	10.8
C++	12.4
C	13.0
C#	15.5
PL/1	14.2
COBOL	16.8
ABAP	19.9

Table 2.1: Development hours per function point of software, ranked by principal programming language (courtesy of ISBSG)

As discussed in Section 2.1, other factors affect software development and its related aspects, such as the size of the development team, their experience, and the complexity and type of application. However, our focus here is on the factor of *programming languages*.

The nature of programming languages as a special software tool makes it difficult to find objective metrics that can be used to provide meaningful comparisons among them. Programming language designers and early adopters make different claims about languages in order to differentiate them from others and to attract users. Furthermore, some developers back certain languages rather than others. Unfortunately, a number of their claims are based on personal affinity and are not supported by hard evidence. Claims are more overstated for modern languages than the earlier ones (Markstrum 2010). Thus, studies on programming languages from a pragmatic perspective are needed to provide supportive evidence and valuable comparisons among them.

2.3 Programming language categories

A programming language is a set of notations used to communicate information to a machine, in particular, a computer. They have strict syntax and defined semantics that are used to describe a program or an algorithm CITE. Programming languages are universal and can, in principle, perform any computation that the famous universal Turing machine can perform (Igarashi et al. 2014). General-purpose languages are used for a wide range of applications. They run on traditional machines and are useful for a wide variety of problems, unlike the domain-specific languages. A programming language that is machine-independent and reflects the problem it solved rather than the structure of the computer or operating system is a high-level one (Ousterhout 1998*b*).

Programming languages can be categorised according to their features, abstraction level from hardware, type system, programming paradigm, the purpose for which they are created and other factors. Based on the abstraction level, languages can have less or no abstraction from the hardware in which they called low-level languages such as Assembly and machine languages (Budd 2002). Such languages are hardware-friendly because they can be directly interpreted by them. However, they lack portability, and each machine has it own instruction set. High-level languages are designed to overcome such issues, are hardware-independent, and are user-friendly; however, a translation step into machine language is required for them to be understood by the hardware(Igarashi et al. 2014).

Another method of categorisation of high-level programming languages classifies them into two groups: *scripting languages* and *system programming languages*. This is known as *Ousterhout's dichotomy* (Ousterhout 1998*a*). According to this dichotomy, the main characteristics of scripting languages are being typeless and interpreted. Scripting languages are also considered to be a higher level of abstraction from the machine than the system programming ones. Example languages that fall under this category are Tcl, Python, Perl, Rexx, and Visual Basic. In contrast, system programming languages are strongly typed and compiled. PL/1, Pascal, C, C++, and Java are examples of such languages.

Language could also be categorised based on the programming paradigm they support, such as imperative, functional, object-oriented and more. The computation in functional languages is achieved through the evaluation of expressions (Hudak & Paul 1989). That is, functions can be passed to other functions as arguments and can be returned as a result of a function. Common features of those languages are (a) higher-order functions; (b) immutable data structure; and (c) recursion. Imperative languages have a different set of common features, which are (a) variables that are named memory storage areas, (b) assignment statements to assign values to the variables, and (c) iteration (Petricek & Skeet 2009). In such languages, programs are sets of commands that tell the computer what to do. In object-oriented languages, data are encapsulated inside objects. The

common features here include classes and objects, messages and methods, inheritance, and polymorphism. (Budd 2002)

It is important to note that categories of programming languages are not mutually exclusive. High-level languages today manage to support different features, making it difficult to draw boundaries to categorise them. For instance, a language like Java is a hybrid one (imperative and object-oriented), and it also supports anonymous functions (λ-abstractions), a functional feature that was added to Java 8 (Oracle 2016). Another example is C#, which supports imperative, declarative, and functional programming (Petricek & Skeet 2009). However, a language's syntax and services, in addition to other factors such as its libraries, coding conventions, and guidelines, may direct a programmer's choice towards a specific language.

2.4 Evolution of high-level programming languages

It is difficult to identify the first appearance of a language, in the same way as with the firsts in history. The language team may have been working for several years prior. They may have given an informal presentation. Sometimes a research paper is published in a different year of submission date. So, it is not possible or meaningful to compare the year of the first appearance. Thus, we are considering the 'conceptual' development of programming languages rather than the historical one.

One of the most significant languages appeared in the year 1957, called FORTRAN (which stands for FORmula TRANslating system), and almost all later imperative languages were influenced by it. FORTRAN is used for mathematical and scientific computing (O'Regan 2008). It included the DO loop statement, the logical IF selection statement, and input/output formatting (Rajaraman 1997). The language was developed by J. W. Backus, a Turing Award winner, and his team in IBM labs. And during the time between 1958 and 1960, Backus was also involved in the ALGOL design team.

ALGOL 60 had a strong influence on the design of other languages developed from 1960 onwards, such as Pascal, C#, C++, Java, and C#. It was designed in such a way that it is machine-independent and it is the first language to have a formally described syntax (BackusâĂŞNaur Form). In ALGOL 60 two evaluation strategies for parameters passing were supported: pass by value and pass by name (O'Regan 2008). Also, the concept of block structure (grouping declarations and statements in a local scope) was introduced, nested procedures were supported, and recursive procedures were allowed. Although recursive functions were new to the imperative languages world, LISP had already supported them back in the 1959 (W.Sebesta 2008).

LISP (LISt Processing) was developed in 1958, as a functional programming language for list processing, by John McCarthy in MIT labs. In Lisp there is no formal difference

between code and data. The language targeted Artificial Intelligence (AI) applications, and for about 25 years, LISP completely led the field of AI (W.Sebesta 2008). Along with its success in large-scale and AI applications, Lisp is also a major language in functional programming. Although now quite old, Lisp, in the forms of its big family of dialects and descendants such as Common Lisp and Scheme, still remains in use today.

Functional programming is different than the imperative paradigm, the computation in this style of programming is achieved through the evaluation of expressions (Hudak & Paul 1989). In this function-centric style, functions are considered as values, they can be passed to other functions as arguments and can be returned as a result of a function. This methodology often provides shorter programs, which are easier to read, and easier to reason about. Some common features of this paradigm are (a) higher-order functions, in which one can send function as arguments to another function, such functions, however, has limited support in some imperative languages in which we define objects to get advantage of higher-order functions; (b) immutable data structure; and (c) recursion as a core mechanisms for defining functions. Additionally, functional languages support the concept of lambda abstractions (making a term into a function of some variable). Even with these features and benefits, however, functional languages have been slower to gain mainstream acceptance than some of their advocates hoped (Petricek & Skeet 2009). Languages belong to this category are relatively small (niche languages) such as Scheme, Haskell, ML, and F#.

Returning to imperative languages, another member of the ALGOL family is Pascal, a descendant of ALGOL 60, designed in 1970 (O'Regan 2008). It adopted the concepts of structured programming defined by E. W. Dijkstra and C. A. R Hoare. Pascal helped the development of dynamic and recursive data structures (e.g. linked lists, trees, and graphs) (W.Sebesta 2008). It included some landmark features to make it possible to build these dynamic structures, such as pointers. In Pascal, pointers are powerful features, they are used to access dynamically allocated variables. However, Pascal uses manual memory management, which mean that explicit deallocation is required. This explicit deallocation process can be feasible for small programs but can be very complex and costly for large ones (W.Sebesta 2008).

Another powerful language that has its roots in Algol is C. It is a general-purpose language designed back in 1972 in Bell Laboratories. The famous operating system UNIX is written in C, which made the language traditionally used for systems programming. C is at a lower-level on the abstraction hierarchy, making it close to the machine code but still high-level language to allow program's reuse and 'portability' (O'Regan 2008). Pointers in C are used as addresses; they give us more explicit access to different memory locations since they can point at virtually any variable anywhere in memory, whereas in Pascal and Ada 83 pointers can only point into the heap (W.Sebesta 2008). Memory management in C is again manual which makes it vulnerable to bugs since allocate and

release storage need great care (Detlefs et al. 1994). C language influenced the design of both C++, Java, and others.

In the late 1960's and mid-1980's a new programming paradigm was evolving, known as Object-Oriented Programming (OOP). This paradigm gained huge popularity in the 80s and 90s. Many books and special issues of journals covered the new topic and its different techniques, while software engineers, compiler writers, and designers were bustling to move their products to the new paradigm (Budd 2002). The key characteristics of this paradigm revolve around objects, the first-class citizens here. Whereas for instance, functions are the first-class citizens in the functional programming paradigm âĂŞ we discussed above. Data in OOP is encapsulated inside objects, so, any object owns its data and the operations on them. The behavior of objects can be extended by adding more operations (methods), where at the same time, they protect their data and operations from other objects under their interface. The fundamental concepts of OOP include classes and objects, messages and methods, inheritance, and polymorphism (Budd 2002). The first major programming language that fully supported object-oriented programming concepts was Smalltalk (Smalltalk-80 specifically) back in 1980 (Igarashi et al. 2014, W.Sebesta 2008).

Smalltalk advocates two major ideas, object-oriented methodology and windowed user interfaces (known as Graphical User Interfaces today). Everything in Smalltalk is an object, from primitive types (integers, booleans, characters) to large complex systems. And classes, are likewise no exception, in the sense that a class is an instance of the metaclass of that class, and each metaclass is an instance of another parent or root class (Igarashi et al. 2014). However, this is not the case with other languages such as C++ and Java. C++ and Java are hybrid, combining imperative and object-oriented features (Wu 2009). Both of the latter languages have strongly affected the software development industry and the programming languages world over the last years. They both became and remain very popular. Similar languages also include Objective-C which played a major role in developing Apple's operating systems and applications, and had an influence on their new language Swift.

C++, which was initially called (C with Classes), was built on top of C in AT&T Bell Laboratories by B. Stroustrup in 1980 as a general-purpose language. Originally, C++ embraced traditional programming methodology besides data abstraction to improve the programmer's efficiency and the clarity of the code. Accordingly, C++ provides two constructs to exhibit data abstractions, the class, and the struct. Along with further modifications, object-oriented facilities were added to the language later in 1983 to support OOP. Eventually, C++ supported both procedural and OO programming. The language has multiple inheritance, it provides operators overloading, and has dynamic binding. C++ spread widely due to the enhancements it had over the well-known C: it is higher-level than C, it also facilitates code reuse and entity representation through

adopting OO concepts. C++ became a leading language that is suitable for large-scale commercial software projects (Stroustrup 1999, Wu 2009).

Sun Microsystems back in 1990, considered two widely used languages, C and C++ and attempted to develop a new language that supports object-oriented concepts for consumer devices and to have the power and flexibility of C++. According to this attempt, Java was intended to be a small, simple and machine independent language. Its support for the abstract datatype is similar to C++, except that in Java all user-defined data types are classes (there are no structs), and methods should always define in a class. Java is built in such a way that its program is compiled to an intermediate machine-like code for an imaginary machine, the Java Virtual Machine. Eventually, the use of this language spread rapidly in the software industry world. Java's popularity was a result of the facilities it provides for Internet application, such as portability (Wu 2009). Since Java's bytecode is machine and system independent, it allows Internet applications to run on different local machines after being downloaded. Moreover, Java supports automatic memory management instead of allocating and deallocating memory explicitly (Arnold et al. 2005). The built-in garbage collector will be responsible for reclaim memory automatically for reuse (such a language is so called garbage-collected language). Over the years many features were added to the language, its libraries became bigger and bigger, making it hard to say that it is a small or a simple language by now.

On the other hand, another category of languages that are different in style from 'system' programming languages (e.g. C++ and Java), has arisen. Such languages, referred to as *scripting languages*, are not intended for writing applications from scratch, as in system languages. They are primarily designed to extend the features of existing components, and this is where its name "glue languages or system integration languages" came from. These languages are typically interpreted from source code or bytecode, rather than compiled (Ousterhout 1998*b*). Some scripting languages are used in scripting command line applications, such as Rexx, Shell, and Perl.

JavaScript is an example of a popular scripting language. After the invention of graphical web browsers, and flourishing of the World Wide Web in the 1990s, the need for dynamic modification and creation for the static HTML documents led to the appearance of JavaScript in late 1995. JavaScript is a lightweight language, its code is embedded in HTML document and interpreted when the document displayed in a web browser. The language support object-oriented concepts, excluding inheritance, and dynamic binding of method calls to methods (Flanagan 1998).

Another example of a popular language scripting language is Python. It is a dynamically-typed language, object-oriented, and interpreted scripting language. It was created in the early 1990s (Rossum 2003). Python was typically used for relatively small programs, and because of interpretation, it is used for prototyping and rapid application development tasks rather than high-performance computing. A significant advantage of Python is its

ability to be extended. Modules with additional or new functionality and new types can be built to be used from Python. Furthermore, these extension modules can be written in any compiled language (for example C and C++).

Many of today's languages now support a mix of imperative, functional and object-oriented features. For instance, a hybrid language (imperative and OOP) like Java now support anonymous functions (λ-abstractions), a functional feature, which were added to Java 8. Another example is C#, which is a multi-paradigm language, it supports imperative, declarative and functional programming (Petricek & Skeet 2009). However, the syntax and capabilities of a language are generally biased to only one of these paradigms. A language's syntax and services, in addition to other factors such as its libraries, coding conventions and guidelines, may direct a programmer's choice toward a specific language.

Currently, there are many languages created for different purposes. Figure 2.5 shows the time-line of selected programming languages and the impact languages have on others. We can see in Figure 2.5 that some languages have a strong influence on others, for instance, ALGOL60 and C. Whereas, some could barely make an influence/existence, such as Prolog and sh. However, it is important to note that influence and popularity (usage) are different things. Even if a language did not introduce a new concept that can influence the design of others, it still can make it to the mainstream.

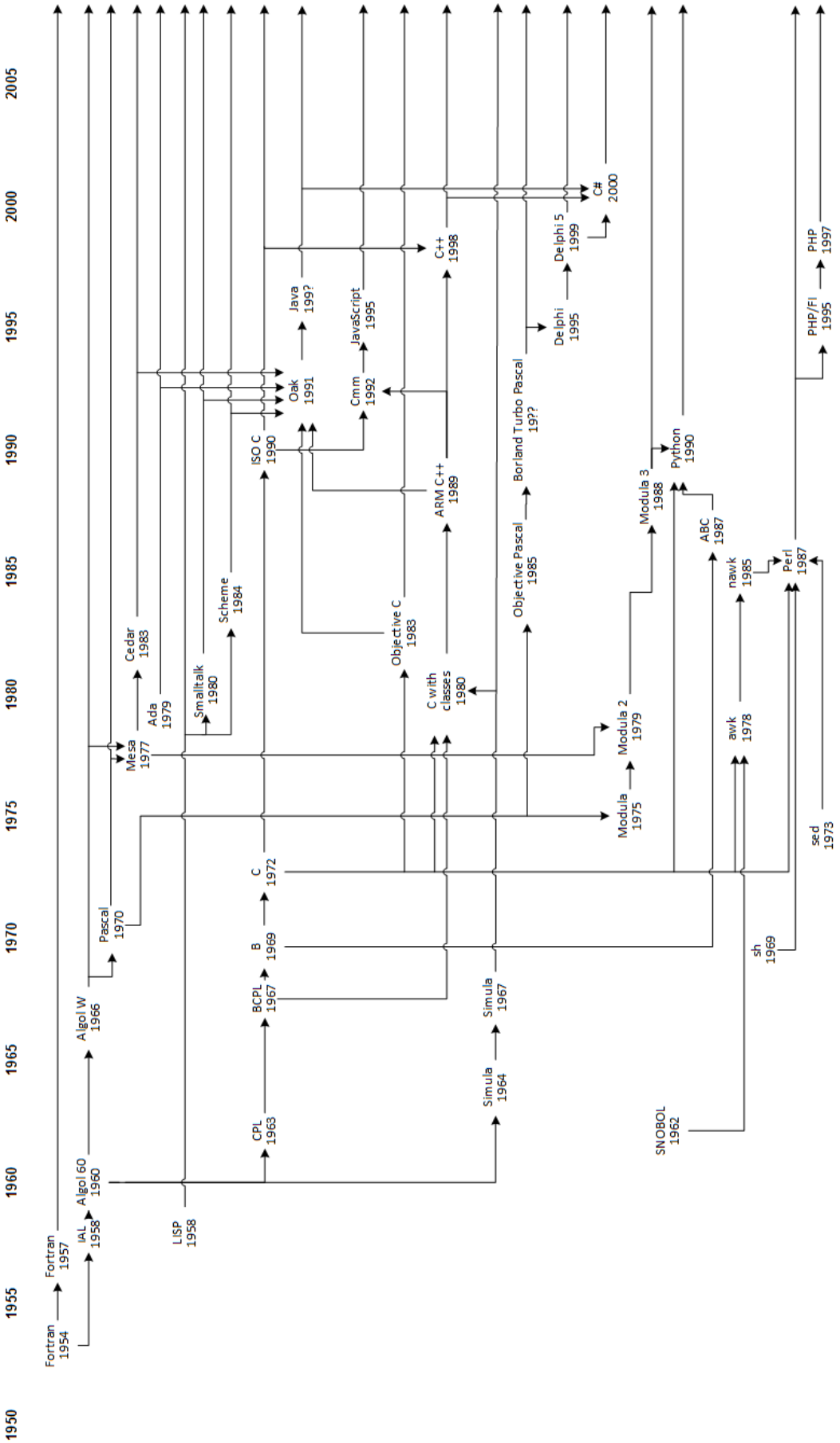

Figure 2.5: Time-line of selected languages showing their approximate date of introduction. Redrawn based on a *Computer Languages History* diagram (Lévénez 2016)

Chapter 3

Research methodology

This chapter presents research objectives and describes the methodology we followed to achieve them. The objectives are listed in Section 3.1. In Section 3.2 we present the methods used per objective, Section 3.2.1 covers the tools and describes the approach used to collect, store, and analyse the data. The dataset is presented in Section 3.3. Then in Section 3.4 we discuss the importance of the collected dataset. Finally, the conclusion is provided in Section 3.5.

3.1 Research question and objectives

There are dozens of programming languages in use today and new languages, and language features are being introduced frequently. However, there is little empirical work examining the usage and practice of programming languages. In this research we look into languages from an empirical perspective to address their relationship with software development projects and practices. This relationship is investigated in three directions: (1) the association between language/language design and user adoption and popularity, (2) language features usage, and (3) project development attributes. The research is carried out in a comparative settings to investigate the following question: *what is the impact of programming language on open-source projects and related practices?*. In other words, a comparison is made between languages both individually and in groups to understand similarities and examine differences, if any, in three aspects; popularity and user adoption, feature usage, and OSS projects attributes. This study has three primary objectives summarised next.

Research objectives:

1. Examine trends in language adoption and investigate their popularity in open-source software.

2. Examine language feature usage, and inspect statistical association between language and features usage in a large-scale setting.

3. Examine statistical association between language and the development of OSS projects in a large-scale setting.

Throughout this research we inspect the practices of developers through mining software repositories on Github. This should provide insights about the current state-of-the-art practices for both language designers and developers. It will also help language designers understand the user community, as well as how their languages are used in real projects and whether there is any difference between the usage among languages and the usage among language groups that would suggest a strong relationship between the two. Moreover, adopting data mining methods in this context encourages an evidence-based design of programming language rather than a design based on anecdote and assumed needs. This investigation was conducted in a large-scale setting using statical methods by accounting for confounding factors, such as project size and type, per language and language group. Each one of the objectives is studied in a separate chapter.

3.2 Research methods

The research methodology is primarily based on mining software repositories. Different mining techniques have been used throughout the thesis to address the research question and fulfil the related objectives. The methods we followed are summarised in Table 3.1. Nevertheless, as the research is divided into three related studies, a detailed methodology is discussed for each study.

Objective	Reasearch method
Objective #1	Hypothesis testing (Mann-Whitney U test) Statistical analysis Trend analysis Association rules and Apriori algorithm Topic modelling (Natural language processing)
Objective #2	Statistical hypothesis testing (Mann-Whitney U test) Mining source code Statistical analysis
Objective #3	Statistical hypothesis testing (Mann-Whitney U test) Mining software repositories Statistical analysis

Table 3.1: A summary of the research methodology.

We intended to use an open source dataset for this investigation. We compared a number of the mostly used online repositories for open source projects in terms of size (users and

projects), establishment date, and available features. The repositories managed to offer similar features, however, Github had the largest number of users and projects compared to other popular repositories such as SourceForge, Launchpad, and Bitbucket (Table 3.2).

Repository	Established	Users	Projects
GitHub Github (n.d.)	2007	56,000,000	100,000,000
Bitbucket (Davis, Justine 2016)	2008	6,000,000	unknown
SourceForge (Slashdot Media 2017)	1999	3,700,000	502,000
Launchpad (Canonical Ltd. 2017)	2004	3,713,633	40,457
GitLab (Babb, Luke 2016)	2011	100,000	unknown

Table 3.2: Source code repositories ranked by size (in terms of users and projects)

3.2.1 Approach and tool suite

We have checked about 15,000 repositories on GitHub, and retrieved the data of 5,350 selected projects. The methods and tools used to retrieve, store, and analyse the dataset are summarised in the following phases:

1. **Data retrieval:** Which has the following steps:

   (a) retrieval of the initial list of popular repositories from GitHub: repositories that have at least 500 stars rating,

   (b) automated identification of primary language: the language that makes up at least 95% of the project's total code, and

   (c) retrieval of the projects' data and source code from those repositories.

   The initial retrieval phase has been implemented through GitHub Archive mirroring API on Google BigQuery. After that, the resulting JSON file was parsed, and repositories were checked for their main language. This checking process was conducted using GitHub REST API (V3). If the repository had a primary language, we pulled the repository data (such as project ID, owner, commits, contributors, etc.) along with the source code files directly from Github, instead of the mirroring APIs. This was done to avoid retrieving curated data from the archiving services and ensure data freshness. It is also worth noting that Github uses a specialized tool, called *linguist*, to determine the repository's main language. This tool detects the main language by aggregating all the languages in a repository and naming the top one as the main language. Sometimes, the main language can sometimes make up just a small part of the total code. Thus, to address this issue, we decided to create our own definition of what the main language is: the main language is one that makes up at least 95% of the project's code.

2. **Data storage:** The projects' data are stored locally in JSON files as well as in a relational MySQL database. The source code files are also stored locally to be inspected for their content. After examining the top starred repositories on GitHub, we found that the projects where a main language can be identified comprised 48% of the total inspected population, whereas repositories that did not have a main language (i.e. the codebase is shared between multiple languages) constitute 44.5%, and repositories dedicated for documenting purposes only (tutorials and books) make up about 7.5% of the total population.

3. **Data analysis:** The projects' data are analysed statistically using R.

The 3-phases approach and tools used are illustrated in Figure 3.1

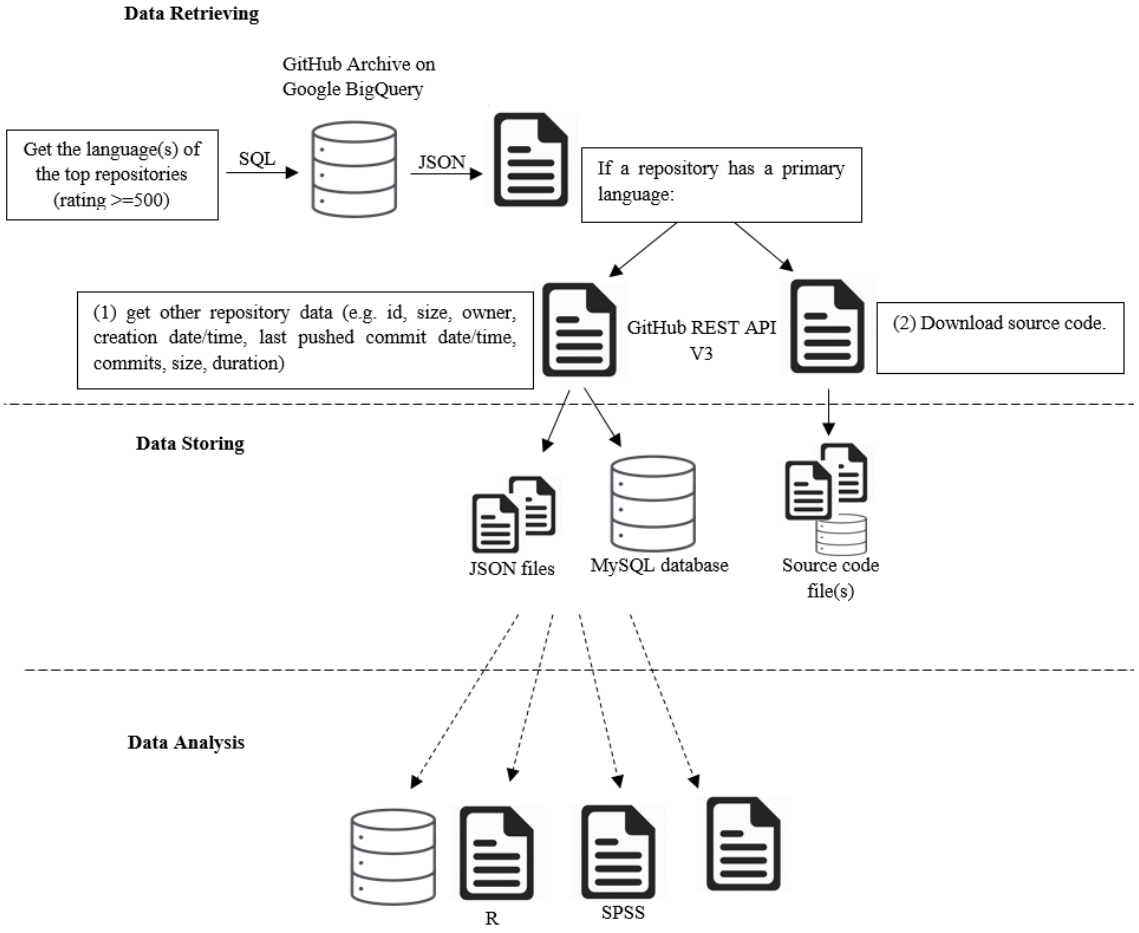

Figure 3.1: Tools used in data retrieval, storage, and analysis.

3.3 The dataset

We have examined the most popular 15,000 repositories on GitHub, based on the ratings from January 2012 to December 2019. The selected projects have the following characteristics:

1. The project should have a main language, which makes up 95% of its total code.
2. The main language should be a high-level, and general-purpose one.
3. The repository should not have a reportedly infinite number of contributors and/or commits by Github. It should not be missing key project attributes.
4. There should be at least 60 projects per language to be considered for the dataset.

The collected data per project include, but are not limited to, languages (including the main one), source code, size in bytes, creation time/data, last pushed commit time/date, contributors, and commits. Project's duration has been calculated from the creation date of the repository to the last release. However, if no releases were found in the repository, the duration is calculated from the creation date to the last pushed commit. After excluding projects with special purpose languages (such as Shell and HTML) and the ones with missing key data for the purposes of consistency, cleaning and removing any duplications, we arrived with 5,350 projects. The top 12 primary languages are: JavaScript, Java, Python, Go, Objective-C, Swift, PHP, Ruby, C#, C++, TypeScript, and C. Table 3.3 shows the programming languages along with the number of the selected projects per language in descending order. This dataset is not a random sample; it was systematically generated based on a specific criteria.

	Language	Projects
1	JavaScript	1559
2	Java	1087
3	Python	747
4	Go	455
5	Objective-C	349
6	Swift	267
7	PHP	278
8	Ruby	180
9	C#	159
10	C++	111
11	TypeScript	93
12	C	65
	TOTAL	5350

Table 3.3: Most popular languages on GitHub based on number of projects.

Per language

The dataset projects are varied in type, characteristics, and complexity. The majority of them (49%) were written in JavaScript (29%) and Java (20%), followed by Python (14%), and Go (9%). C++, TypeScript, and C languages come last with 111, 93, and 65 projects, making up 2% for each of C++ and TypeScript, and 1% for C of the dataset size. Table 3.4 shows means of selected projects data per language, whilst Table 3.5 shows the medians of selected projects data per language.

Language	Size (SLOC)	Commits	Contributors	Duration (months)
JavaScript	30663	850	57	51
Java	56510	995	32	37
Python	33673	1520	67	53
Go	119670	996	56	50
Objective-C	13244	330	25	44
Swift	13117	565	28	39
PHP	20697	1355	94	73
Ruby	27189	2692	195	93
C#	124765	2409	71	61
C++	293834	3862	76	61
TypeScript	31572	2758	293	57
C	494122	1406	549	66

Table 3.4: Means of selected projects data per language.

Language	Size (SLOC)	Commits	Contributors	Duration (months)
JavaScript	5005	316	26	47
Java	4455	131	7	30
Python	4873	359	22	52
Go	9759	338	27	52
Objective-C	2658	142	13	42
Swift	2527	231	14	41
PHP	3217	542	48.5	74
Ruby	4127	773	75.5	100
C#	49065	1300	50	64
C++	31324	767	30	62
TypeScript	7788	1354	71	57
C	11063	412	18	65

Table 3.5: Medians of selected projects data per language.

When dataset projects are classified according to size in SLOC, the majority of the included projects are small (35%) and medium (25%), as shown in Figure 3.2. Tiny projects make up 16% of the dataset, while very large projects make up 14% of it. Large size projects make up the smallest portion of the dataset (11%). The scale used here for project sizing is defined as follows:

- tiny: <1000 SLOC
- small: 1,000 - 5,000 SLOC
- medium: 5,001 - 20,000 SLOC
- large: 20,001 - 50,000 SLOC
- very large: >50,000 SLOC

As per language (individually), the majority of JavaScript (32%), Java (43%), Objective-C (46%), Swift (54%), PHP (38%), and Ruby (45%) projects are small. The small and medium projects in Python and Go form almost the same proportion at 27% and 28% respectively, that is the majority of projects in these two languages. When it comes to C# and C++, the majority of projects are of a very large size, at 52% and 44% respectively. These figures are based on the means of project sizes that are shown in Table 3.6.

On the other hand, when projects are categorised based on the number of contributors, half (51%) of the dataset projects are of a large size (see Figure 3.2). The scale used for project sizing in number of contributors is defined as follows:

- small: <5 contributors
- medium: 5 - 20 contributors
- large: >20 contributors

When languages are considered individually, the majority of JavaScript(58%), Python (54%), Go (56%), PHP (81%), Ruby(84%), C# (75%), C++ (56%), and C (47%) projects have a large number of contributors, while Objective-C (42%) and Swift (47%) projects have a medium number of contributors. Java the only language with projects that have mostly a small number of contributors (43%). These figures are shown in Figure 3.3.

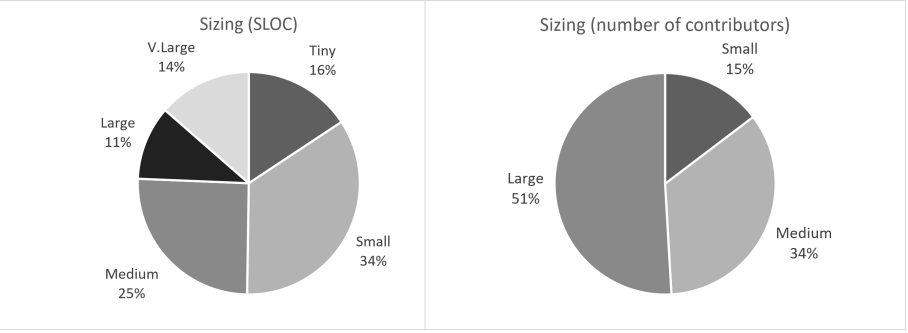

Figure 3.2: Project sizes in SLOC and in number of contributors.

%	JavaScript	Java	Python	Go	Objective-C	Swift	PHP	Ruby	C#	C++	C
Tiny	15.42%	10.22%	17.73%	7.49%	18.04%	14.16%	18.04%	8.43%	1.55%	3.19%	10.94%
Small	31.94%	43.15%	26.95%	28.24%	46.20%	53.54%	38.14%	45.18%	6.20%	17.02%	15.63%
Medium	29.58%	23.93%	27.48%	28.24%	23.73%	22.57%	25.26%	30.72%	19.38%	19.15%	32.81%
Large	12.12%	9.92%	12.59%	10.37%	7.91%	4.42%	9.28%	7.23%	20.93%	17.02%	12.50%
V.large	10.94%	12.78%	15.25%	25.65%	4.11%	5.31%	9.28%	8.43%	51.94%	43.62%	28.13%

Table 3.6: Distribution of projects' sizes (SLOC) per language.

Per language group

We categorised the included languages based on their design into three binary groups;

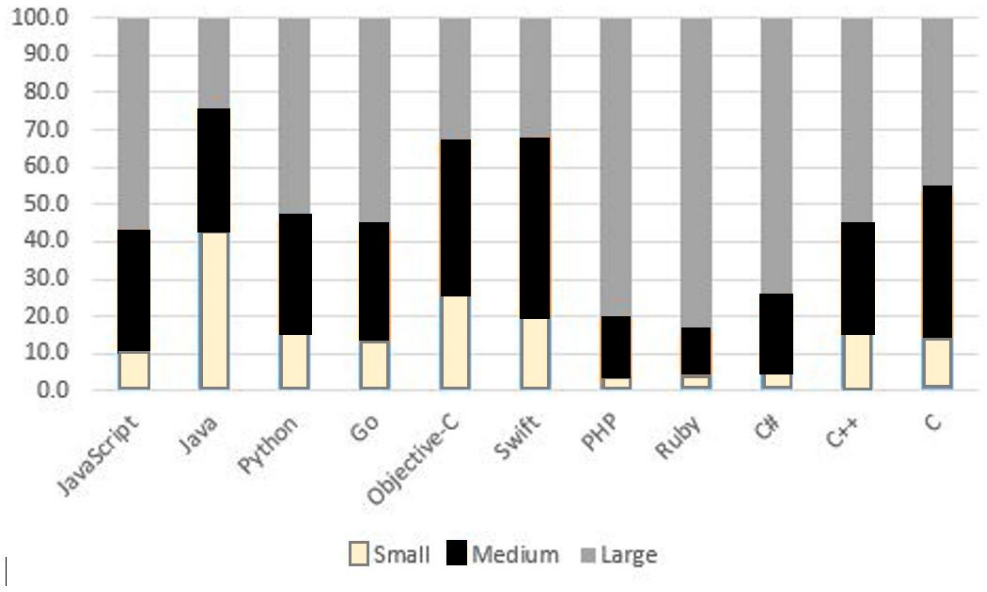

Figure 3.3: Distribution of projects' sizes (number of contributors) per language.

statically typed vs dynamically typed languages, strongly vs weakly typed ones, and managed vs unmanaged memory languages as can be seen in Table 3.7.

Language Classes		Languages
Type system	Static	C, C++, C#, Java, Go, Swift, TypeScript
	Dynamic	JavaScript, Python, Ruby, PHP
Type checking	Strong	Java, Go, Python, Ruby, Swift, Objective-C, TypeScript
	Weak	C, C++, C#, JavaScript
Memory management	Managed	Ruby, Python, PHP, JS., Java, Swift, Go, C#, TS.
	Unmanaged	C, C++, Objective-C

Table 3.7: A classification of languages based on their design.

When languages are categorised according to type system, the total number of included projects is 5001 and the sample size appears to be balanced per language group. The statically typed languages group has 2237 projects, and the dynamically typed languages group has 2764 projects. When projects are sized in source line-of-code, the majority of the two groups are small and medium sized (i.e., 59% of the statically typed group and 60% of the dynamically typed group). When they are sized according to the number of contributors, 36% of the projects in the statically typed group are of a medium size, and 42% are large. In the dynamically typed group, 31% are of medium size and 61% are large. The distribution of the included projects according to size in source line-of-code and in contributors per group are shown in Figure 3.4.

When grouped according to language type checking, 3,178 out of 5,072 projects are strongly typed while 1,894 projects are weakly typed. Again, the majority of the two groups are small- and medium-sized when projects are sized in SLOC (i.e., 63% of the strongly typed group and 56% of the weakly typed)group). When projects are sized in

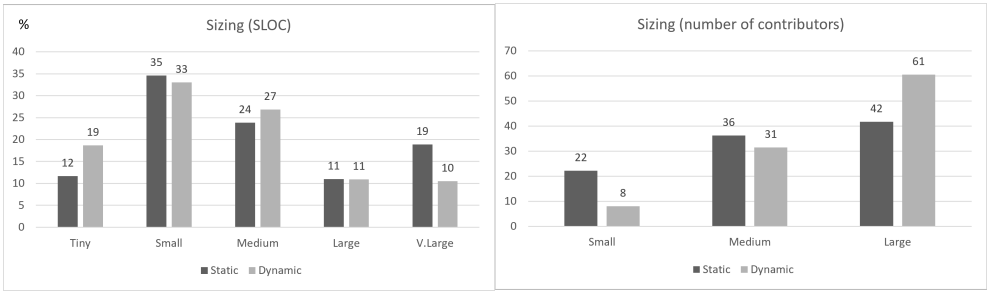

Figure 3.4: Distribution of projects' sizes per language group (static vs dynamic).

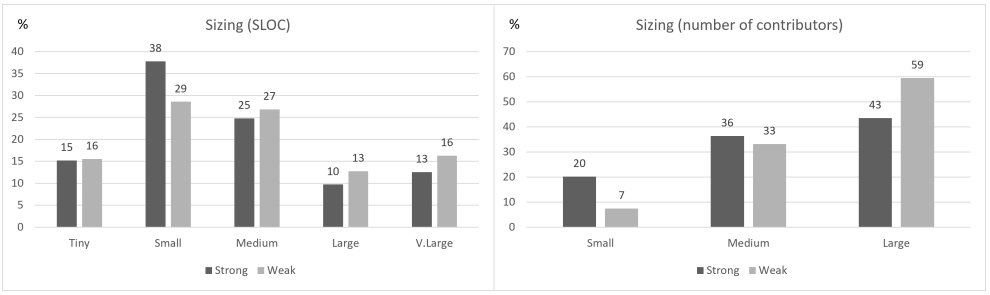

Figure 3.5: Distribution of projects' sizes per language group (strong vs weak).

number of contributors, 36% of the projects in the strongly typed group are medium-sized and 43% of them are large. The largest proportion of the projects in the weakly typed group is also large large (59%), whilst 43% are medium-sized. The distribution of the included projects according to size in source line-of-code and in contributors per group are shown in Figure 3.5.

Finally, when projects are categorised based on memory management, 3,738 out of 5,350 projects are in the managed memory group, while 1,612 in the un-managed memory languages group. The distribution of projects according to size in SLOC also shows that the majority of projects in the two groups are small- and medium-sized. This means that 32% small and 26% medium in the memory managed group make up 58% of the total projects in the group. Whereas, 40% of the projects are small in the unmanaged memory group and 24% are medium making together 64% of projects in the group. When projects are sized based on number of contributors, the majority of them are large (60%) in the managed memory group. In the unmanaged memory group, the largest proportion of projects are medium-sized (39%), and the rest of them are equally distributed between the small and large size for about 30% each. The distribution of the included projects according to size in source line-of-code and in contributors per group are shown in Figure 3.6.

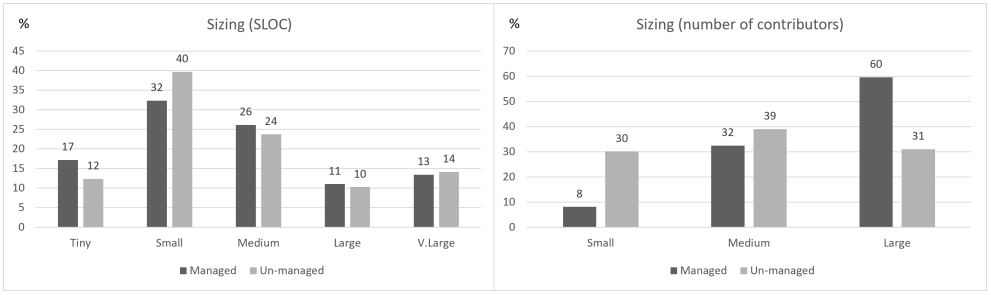

Figure 3.6: Distribution of projects' sizes per language group (managed vs unmanaged memory).

3.4 Significance and contributions

The issue and importance of the reproducibility of empirical software engineering studies have been discussed by different researchers (Leeuw et al. 2001, Shull et al. 2004, Vegas et al. 2006, Shull et al. 2008, Robles 2010). The availability of research datasets is critical in validating findings and replicating and reproducing studies which can, in turn, enhance research in the field. However, a study of 170 MSR research papers (Robles 2010) found that a very small number of the authors make their datasets publicly available (6 papers only). Another study (Sureka et al. 2015) conducted on MSR paper authors reported that only one third of the respondents stated that their datasets are publicly available. This is in line with the findings from (Cosentino et al. 2016), that more than two thirds of empirical studies on Github do not publish their dataset. Hence, despite the use of publicly available data, authors do not make their research datasets available, making it difficult to replicate the findings or even compare their outcomes.

Although Github is the most used online hosting service for open source projects in the world (Gousios et al. 2014) with more than 56 million users and more than 100 million software repositories as of December, 2020, 80% of its repositories have no stars (favouring or liking by registered users)(Sanatinia & Noubir 2016). Moreover, after excluding the non-starred repositories, 95% have 13 stars or less. The repositories in this study's dataset have at least 500 stars, making it good representative of popular open source projects that can be used to empirically validate claims about open source software (OSS) development and artifacts.

Furthermore, our dataset is relatively large and made up of real, popular projects that reflect current practices in the software industry. The data was also collected from Github directly rather than curated mirroring APIs. It is important to emphasize here that the dataset includes only projects where a main language can be identified. The main language is the programming language that comprises at least 95% of the total project code. This further restriction makes this dataset a good choice for comparative language studies.

Nevertheless, mining Github repositories and retrieving large amounts of data is a challenging task. Github allows its data to be accessed over HTTPS as JSON, but it does not provide a schema for its data. Therefore, to mine Github it is necessary to traverse back its data using REST requests and JSON responses. Moreover, Github imposes a rate limit on its API of 5000 requests per hour. Given the huge number of events generated per day with every single event leading to a series of dependent requests, pulling large amounts of data from Github would create significant delay.

In summary, this research has generated and published a dataset that meets the following objectives:

1. The volume of the dataset is relatively large, with 5,350 projects in 12 languages.
2. The projects are the most popular ones on Github, with at least 500 stars given by registered users.
3. Projects in the dataset have a primary language that makes up 95% of the source code.
4. Up-to-date project data reflects current, modern practices in the software industry.

3.4.1 Dataset availability

The dataset associated with this research is published under CC BY-NC-SA 4.0 license and is available at https://www.kaggle.com/muname/github-repos-mainlang.

3.5 Conclusion

The availability of research datasets is important in order to empirically validate claims about software development processes and the resulting artifacts. However, it has been found in the literature that more than two thirds of empirical studies on Github do not publish their dataset making it difficult to replicate findings or even compare outcomes. To address this issue, this work publishes a reasonably large, systematic, and complete for the selected criteria dataset of 5,350 projects in 12 general-purpose, programming languages for research purposes. The dataset is made up of real projects reflecting current practices in the software industry, and the data is collected from Github directly, rather than curated mirroring APIs. This dataset can be used to empirically validate claims about software development, software artifacts, pragmatic aspect of languages, and in data mining and machine learning training and test sets.

Chapter 4

Programming language popularity and trends in OSS projects

In this chapter, the first research objective is fulfilled, in which language popularity and trends are examined. In Section 4.1, the chapter objectives are listed. Next, in Section 4.2 language popularity is discussed. In Section 4.3 trends in programming languages are investigated. The most common combinations of languages in OSS projects are studied in Section 4.4 and inSection 4.5 the project types in the dataset are inspected. The chapter results are discussed in Section 4.6. Then, the related work is covered in Section 4.7 and the chapter work is concluded in Section 4.8.

4.1 Objectives

Examining the usage and practice of programming languages is important to understand their popularity and recognize trends. This provides insights for both language designers and developers to observe whether the adoption of a specific language or language feature is headed towards a particular direction. The overall aim of this chapter is to identify popular languages and compare their usage, and to observe trends and patterns of language usage in OSS projects. This is to investigate if a relation between user adopting and language design exist in the dataset. This purpose is comprised by the following objectives:

- Investigate the most popular languages in developing open source software.
- Study patterns of change in language popularity and usage over time to investigate whether language usage has increased or decreased, and the pace at which the change has occurred.
- Examine the most common combinations of languages in OSS projects.

- Explore the nature of the included projects based on their description.

Hence, we investigated the following questions:

1. What are the most popular languages in developing OSS projects? Are they the same as in other popularity indexes?
2. What are current trends in language usage?
3. Which language combinations are most common?
4. What is the nature of the projects that have been written in those popular languages?

4.2 Language popularity in OSS projects

Language popularity is defined here in three ways; (a) languages that have the largest number of projects, (b) languages with the highest number of stars (favouring by registered users) given to its projects, and (c) languages with the highest number of contributors to its projects. As explained in Chapter 3, the examined projects should have a threshold of at least 500 stars and the period of study covered projects created between 2008 and 2019.

The 15 popular languages based on the total number of projects per language are shown in Figure 4.1(a). When definitions (b) and (c) are applied, a language needs at least 10 projects to be considered for popularity lists. In Figure 4.1(b), languages are listed based on the mean number of stars given to its projects. In Figure 4.1(c), languages are listed based on the mean number of contributors to its projects.

Languages that appear in the 3 lists are C++, TypeScript, and Ruby. JavaScript, Go, PHP, Shell, CSS, Dart, MakeFile and Clojure have made appearances in 2 lists.

The languages are mainly general-purpose in the first list Figure 4.1(a), except for Shell, HTML and CSS. As can be seen, JavaScript is the most popular language with 3,000 projects, followed by Java (1,443 projects) and Python as 3rd with 1,362 projects. These 3 languages make about 50% of the total number of inspected projects (11,872).

Another story can be seen in the 2nd list Figure 4.1(b), with 9 different languages and more special-purpose languages than the first one, however, with considerably fewer projects. That is, the total number of projects in the top 3 languages in this list (Dart, Assembly, and DockerFile) is 42 compared to 796 projects in the last 3 languages on the first list (C#, Shell, and CSS). Dart is at the top here with a mean of 10,737 stars given to its projects, followed by Assembly with a mean of 7,520 stars. Dockerfile, Makefile,

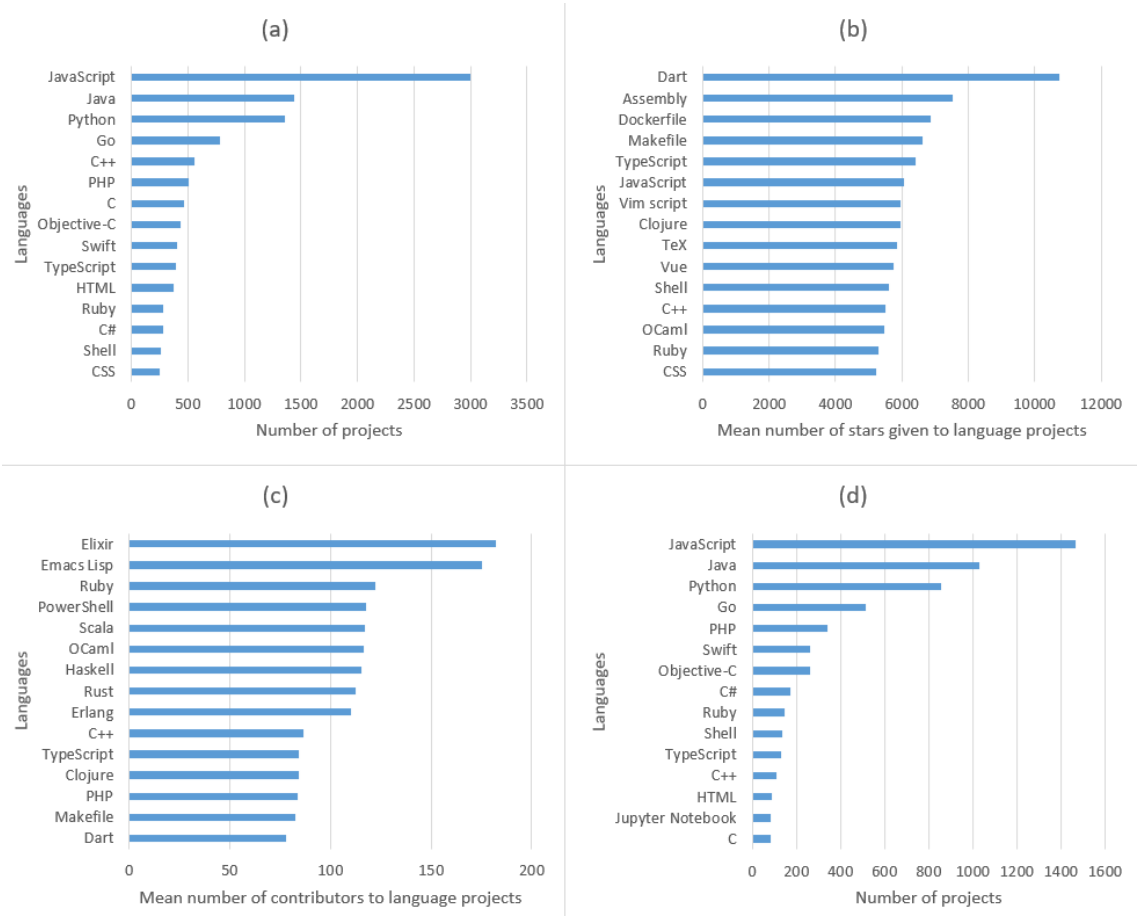

Figure 4.1: Most popular languages in GitHub based on (a) number of projects with >= 500 stars, (b) the mean number of stars given to its projects, (c) the mean number of contributors and (d) number of projects with >= 500 stars and a*main language*.

Typescript, and JavaScript come next with a mean in the range between 6,850 and 6,000 stars. The rest of the languages have a similar mean of around 5592 stars.

The 3rd list Figure 4.1(c), has a number of languages that support functional programming and did not appear in the previous 2 lists, such as Elixir, Emacs Lisp, Scala, OCaml, Haskell, Rust and Erlang. The languages with the highest mean of contributors, that are 182 and 175, are Elixir and Emacs Lisp respectively, with 14 projects each. This is followed by 7 languages of a mean around 116 contributors, and the remaining 6 languages are of a mean around 83 contributors.

The language in those lists has been reported by Github. To detect and name the project's language, Github uses a specialized tool called *linguist*[1] that aggregates all the languages in a project repository and names the mostly used one as the project's language. Accordingly, this language can make up a relatively small proportion of the total code (30% for instance); however, it occupies the largest space compared to the

[1] github.com/github/linguist

other languages in the project. Thus, we defined and applied another definition of a *main language*, as the one that makes up at least 95% of the project's code. Correspondingly, a slightly different list resulted Figure 4.1(d), that is based on the number of projects where a *main language* can be identified.

In this study, we focus on the first definition of language popularity, which is based on the number of projects. This list has almost the same languages as in the first list (4.1a), however, with a slightly different order and one language difference. The styling language CSS is replaced here with Jupyter notebook, which is a documentation language for the open-source application development environment Jupyter. The order of the other languages has been affected because the number of projects decreased when the main language definition was applied. New languages, Go and Swift, come 4th (455 projects) and 6th (267 projects), although they were introduced in 2009 and 2014, respectively. Whereas C, with the longest history (being introduced in 1972), could barely make it to the list, with 65 projects.

Additionally, for the purpose of this research, we only considered general-purpose programming languages, and excluded styling, documenting, and special purpose languages such as HTML, CSS and Jupyter notebook. Thus, the dataset of the subsequent sections is shown in Table 4.1.

	Language	Projects
1	JavaScript	1559
2	Java	1087
3	Python	747
4	Go	455
5	Objective-C	349
6	Swift	267
7	PHP	278
8	Ruby	180
9	C#	159
10	C++	111
11	TypeScript	93
12	C	65
	TOTAL	5350

Table 4.1: Most popular languages on GitHub based on number of projects which have at least 500 rating and have a primary language that made 95% of project's codebase.

4.3 Trends in language usage

Trend analysis is an approach to observe and predict statistically detectable changes. It is used to investigate a hypothesised relationship between quantitative variables statistically (Lavrakas 2008). In this section, a trend analysis based on the historical data of

the included projects is conducted to detect and describe patterns in languages' adoption over the specified time period. Trends here are analysed graphically as per language (individually), and using Sen slopes and Mann-Whitney U test to compare trends between language groups (Field et al. 2012). Mann-Whitney U test is a non-parametric, statistical hypothesis test to determines if a statistical significant difference exist between two groups. That is, the medians of the two groups are compared and if the resulted p value below the alpha value (usually 0.05), a significant difference is reported (Field et al. 2012).

The usage curve differs from one language to another as depicted in Figure 4.2. JavaScript, the language with the highest number of projects, has an arch-shape curve, starting around 2009. The curve renders a gradual increase in popularity, followed by a gradual but faster decrease after reaching the peak in 2015. The highest values occurred between start of 2014 and 3rd quarter of 2016. Then, decline in usage started to shape around the last quarter of 2016. In July 2018, a surge happened, which is rapidly followed by a drop to hit the lowest point in 2019, with no new projects introduced. Java's curve is similar in shape to JavaScript's, however, with a smaller number of projects. That is, a gradual increase, reaching the highest values between 2015 and 2016 was followed by a decline starting around the 2nd quarter of 2017. Unlike Java and JavaScript's curves with narrow peaks, Python's is more flattened, with the highest values spread over a larger period of time, September 2014 to August 2018. A drop in number of projects follows the curve reaching the lowest value in 2019, with no new projects. Go's curve is similar to Python's in shape, with a smaller number of projects and the highest values spread between the end of 2012 and the end of 2018. Again, almost no new project was introduced in 2019. PHP's curve is almost flat with values spread all over the curve and several small hills. Objective-C's curve is another flat one with most of the values centred between the end of 2012 and 3rd quarter of 2016. Swift's curve starts with a surge in June 2014, when the language was introduced. The curve then decreases gradually towards the drop of 2019. C#'s has an almost flat curve, with the highest values between 2014 and the first quarter of 2015. Another semi-flat curve is Ruby's, however, it has earlier projects than any other language, having projects introduced in 2008, whereas other languages had them later. TypeScript also has a flat curve, with the highest values between November of 2015 and November of 2017. C++ has its highest values between May 2014 and March 2015, whereas, C's curve has its highest values between mid-2015 early 2016.

After that, languages are categorised into 3 binary groups based on features they have; statically typed vs dynamically typed languages, strongly vs weakly typed ones, and managed vs unmanaged memory languages. The projects' data are plotted on a time series graph and a regression trend line representing the usage data per group as obtained as can be seen in Figure 4.3. Usage data are then compared and checked for statistical

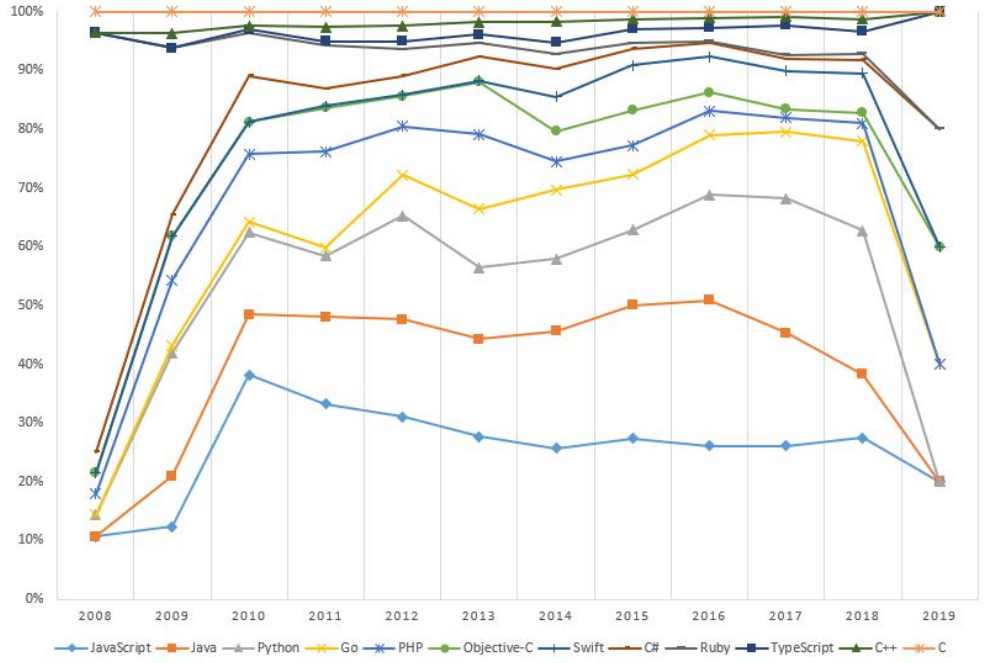

Figure 4.2: 100% Stacked Programming language popularity over time based on number of projects.

significant differences using a Mann-Whitney U test. That is, to inspect the relationship between language features and popularity over the specified time period.

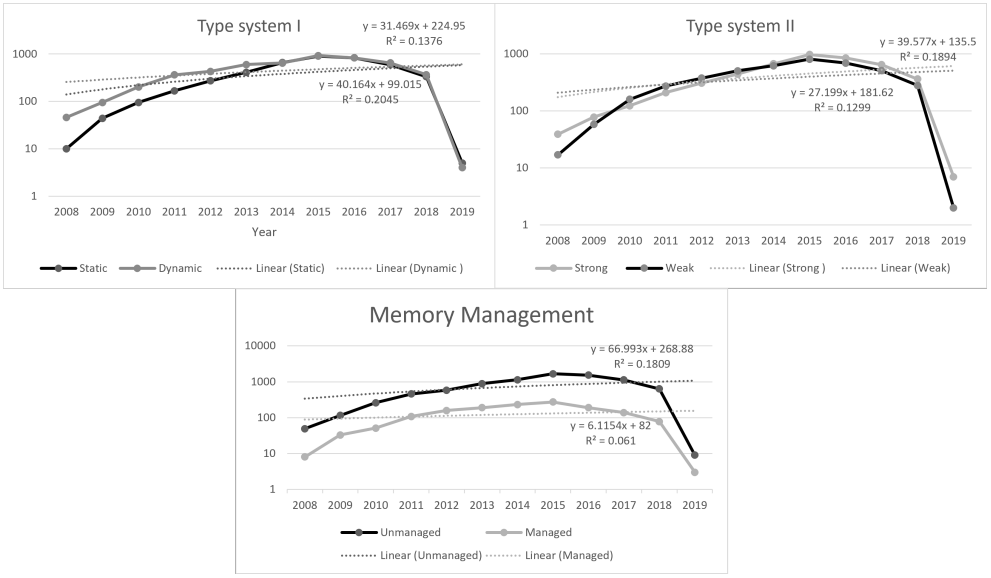

Figure 4.3: Programming language popularity over time based on number of projects per language group.

For the first language group, type system I, the statically and dynamically typed language groups share a similar usage pattern as depicted in Figure 4.3a. That is, a gradual increase since 2008, the year when GitHub was founded, until around January 2017 where a gradual, yet faster, decline starts to occur. The curve reaches its highest values

between 2015 and 2017, inclusive, for both groups. Then, usage data used to statistically test the following hypothesis:

- *Hypothesis 1.* There is no difference in usage between statically and dynamically typed language.

A two-tailed U-test at 95% confidence was used here. The results show that there is a statistically significant difference in the popularity between statically typed languages (M=89.79, SD=716.36) and the dynamically typed ones(M=356.55, SD=3827.34); at U(2721.05)=-6.45 , and p=.001. Thus, the null hypothesis is rejected. The U-test statistics are listed in Table 4.2.

In the second group, type system II, again, the two categories share a similar pattern in the usage curve as shown in Figure 4.3b. That is, a gradual increase, followed by a decline after reaching the highest values between about mid-2014 to mid-2017 for the strongly typed languages, and between start of 2015 and about end of 2016. Thus, the peak of the strongly typed group is wider with the highest values spread over a larger period of time. The usage data for the two curves are then used to test the following hypothesis:

- *Hypothesis 2.* There is no difference in usage between strongly and weakly typed language.

A two-tailed U-test was used to test the hypothesis (95% confidence). The results show a statistically significant difference in the popularity of strongly languages projects at U(4224.24)=13.34, and p =.001. Strongly typed languages are higher in the popularity as can be seen in Figure 4.3b, and the U-test statistics are listed in Table 4.2.

Likewise the usage curves of the previous language groups, the managed and unmanaged memory languages share almost the same pattern. That is, an arch-shape curve that renders a gradual incline, followed by a gradual, faster decline after reaching the peak in 2015 for the two groups. The highest values occur between start of 2015 and end of 2016, as shown in Figure 4.3c. After that, the usage data are compared and used to test the following hypothesis:

- *Hypothesis 3.* There is no difference in usage between the managed and unmanaged memory languages.

The hypothesis was tested with a two-tailed U-test at 95% confidence. The results show a statistically significant difference in the popularity of the managed languages projects and the unmanaged languages; at U(2849.37)=1.316, and p=.000. The U-test statistics are listed in Table 4.2. Managed memory languages are higher in the popularity as can be seen in Figure 4.3.c.

Language group	U	df	p
Type system I	-6.45	2721.05	.001 *
Type system II	13.341	4224.24	.001 *
Memory management	1.316	2849.37	.000 *

* 95% confidence.

Table 4.2: U-test statistics of comparing usage/popularity as per language group.

4.4 Languages combination

This section studies language usage combinations and examines which languages are frequently used together. The results are based on analysing two datasets built on number of projects, whether they have *main language* that makes up 95% of the source code or not. This is to check whether the same findings hold regardless of projects have a primary language. To study the combinations, we use the Apriori algorithm and association rules.

The Apriori algorithm (proposed in 1994) is a classical algorithm in data mining and machine learning to find patterns and regularities in data (Agrawal & Srikant 1994). It is used to mine frequent itemsets and relevant association rules based on an iterative bottom-up search. An association rule is a pattern that indicates with certain probability the co-occurrence of items in a dataset. Apriori is applied here to find the most frequent combinations of languages used per project, and the related association rules.

First, results that are based on the larger dataset, where the included projects need not to have a main language, are presented. The analyzed projects (11,918) are written in a mean of 4.15 languages, 149 languages maximum, and 1 language minimum. As can be seen in Figure 4.4, the strongest relationships (shown in bold line) based on co-occurrence frequency are between JavaScript on one side and HTML, CSS and Shell on the other. Also, between CSS and HTML and Shell on the other, and between Shell on one side and HTML and Python on the other. To further inspect relationships between language combinations, the Apriori algorithm was applied using IBM Modeler. As a result, 42 association rules were produced with a *lift* between 0.78% and 3.19%, *confidence* between 35.19% and 89.91% and *support* between 17.61% and 45.55%. The top 10 association rules ordered by lift ratio are shown in Table 4.3. Lift, confidence, and support are different criteria to measure the strength of the resulting rules. Support is the percentage of how frequently the consequent and antecedent occur together. Confidence is the probability of occurrence of consequent given the antecedent and lift is the ratio of confidence to expected confidence. A lift ratio of >1.0 indicates that the association between the antecedent and the consequent is significant and they are dependable. The larger the lift ratio, the more significant the relationship between the parts.

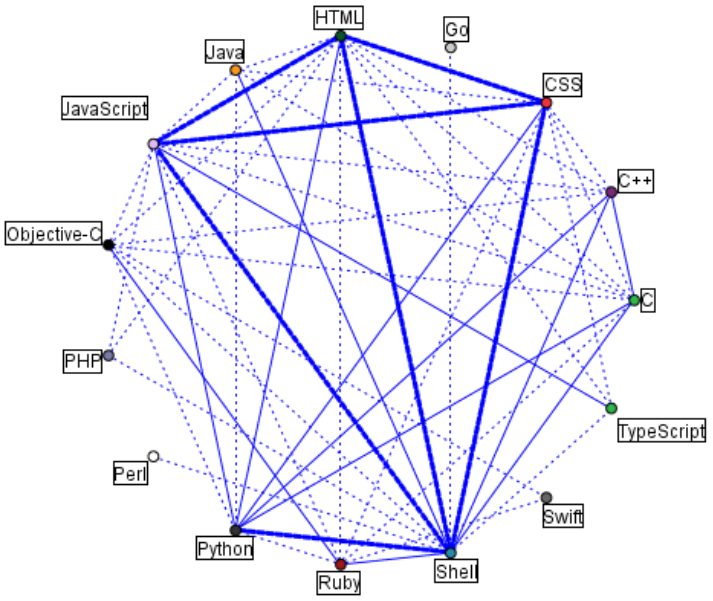

Figure 4.4: Strong/weak connections between programming languages usage in the dataset.

Consequent	Antecedent	Lift	Confidence %	Support %
C++	Python and Shell	3.19	40.45	17.61
C	Python and Shell	2.88	42.12	17.61
CSS	HTML and JavaScript	2.56	76.58	29.48
HTML	CSS and JavaScript	2.29	86.36	26.15
CSS	JavaScript and Shell	2.26	67.57	17.67
CSS	HTML and Shell	2.24	67.13	18.07
HTML	CSS	2.22	83.93	29.92
CSS	HTML	2.22	66.50	37.77
HTML	JavaScript and Shell	2.03	76.69	17.67
JavaScript	CSS and HTML	1.97	89.91	25.11

Table 4.3: The top 10 association rules based on mining all the projects in the dataset ordered by *lift*.

In the second dataset, about 5350 projects were analysed where main language could be identified. The mean number of languages used in projects is 2.38, 24 language at maximum, and 1 language at minimum. Except for projects where Java, PHP and Ruby are the main languages, the majority of projects (more than 50%) were written in more than one language. The strength of relationships between languages is shown in Figure 4.5 where the strongest relationships are shown between (1) JavaScript and each of HTML, CSS, and Shell, (2) Shell and Python, CSS and HTML, and (3) CSS and HTML. Medium strength relationships are shown between (1) Ruby and Objective-C, Swift and Shell, (2) Go and Shell, (3) JavaScript and TypeScript, (4) Shell and Java, Go and CSS. When the Apriori algorithm was applied here to identify which languages are frequently used together, nine association rules resulted, as presented in Table 4.4 with

a lift between 1.47% and 6.89%, confidence between 41.29% and 75.46%, and support between 10.6% and 19.74%.

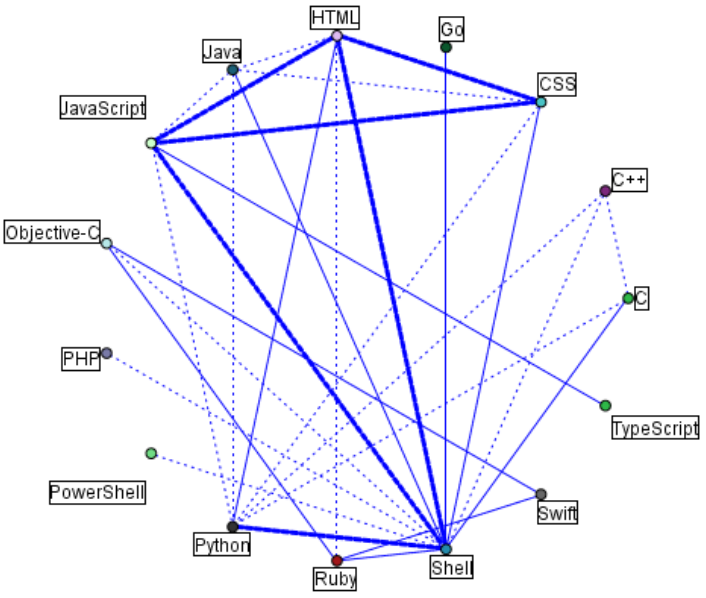

Figure 4.5: Strong/weak connections between programming languages usage.

Consequent	Antecedent	Lift	Confidence %	Support %
Objective-C	Ruby	6.86	51.04	11.06
CSS	HTML and JavaScript	5.10	54.11	11.15
HTML	CSS	3.90	68.36	10.60
CSS	HTML	3.90	41.29	17.55
JavaScript	CSS	2.47	75.46	10.60
JavaScript	HTML	2.08	63.56	17.55
Shell	Makefile	1.93	54.53	12.81
Shell	Python	1.55	43.66	19.74
Shell	CSS	1.47	41.51	10.60

Table 4.4: Association rules based on mining projects where *main language* can be identified.

It is clear that the the relationships between languages hold both datasets for the same languages as seen in in Figure 4.4 and Figure 4.5. The two graphs are almost identical except that fewer languages are included in the second case, hence, the relationships weight is affected. In addition, HTML, Shell and CSS are the languages that frequently co-occur with other languages in the investigated projects as listed in Table 4.5.

Project language	Mean#languages	Max.	Min.	Combined with languages
JavaScript	2.03	20	1	HTML, CSS, Shell
Java	2.02	18	1	Shell, HTML, CSS
Python	2.24	18	1	Shell, HTML, CSS
Go	2.91	13	1	Shell, Python, HTML
PHP	1.71	8	1	Shell, HTML, JavaScript
Objective-C	2.51	8	1	Ruby, C, Shell
Swift	2.80	10	1	Ruby, Objective-C, Shell
C#	3.97	15	1	PowerShell, Shell, HTML
Ruby	2.07	21	1	Shell, HTML, CSS
TypeScript	2.88	10	1	JavaScript, HTML, Shell
C++	5.15	21	1	C, Shell, Python
C	4.89	24	1	C++, Shell, Python

Table 4.5: Summary statistics of projects where a main language can be identified and their most frequent combinations.

4.5 Projects types

To understand the nature of the included software projects, that is, their type (web libraries, data analysis tools, mobile applications,etc.), a mining projects' description was carried out. Projects hosted on Github optionally can have short description to illustrate their purpose. The description is written usually in natural languages. Thus, to identify project types, a *topic modeling* approach was used to analyse the descriptions and detect the underlying abstracts, hence, the potential types.

In the context of *Natural Languages Processing* (NLP), topic modeling is a probabilistic text-mining method used to detect the abstracts in a set of textual documents (Hofmann 1999). Abstracts are clusters of related words that are discovered in the examined documents based on their statistics. In this work, we use *the Latent Dirichlet Allocation* (LDA) model for topic modeling, which detects the underlying topics according to word frequencies (Blei et al. 2003). The LDA model is found to be reasonably accurate in recognizing the hidden topics within text documents. We applied the model using RapidMiner version 9.6, and the approach we followed is depicted in Figure 4.6

The model was evaluated on 4 to 10 topics to find the optimal number of types those projects can be classified into. In the evaluation process, fixed alpha and beta values were used and the average coherence and perplexity scores were compared. The topics' number with the optimal average coherence values was 5. Thus, we categorised the included projects into 5 types: web application, mobile applications, machine learning and AI, web libraries and mobile services.

The largest part of JavaScript (40%), PHP (52%), TypeScript (42%) and Ruby (36%) projects are of the first type. Conversely, 47% of Java, 59% of Objective-C, and 52% of

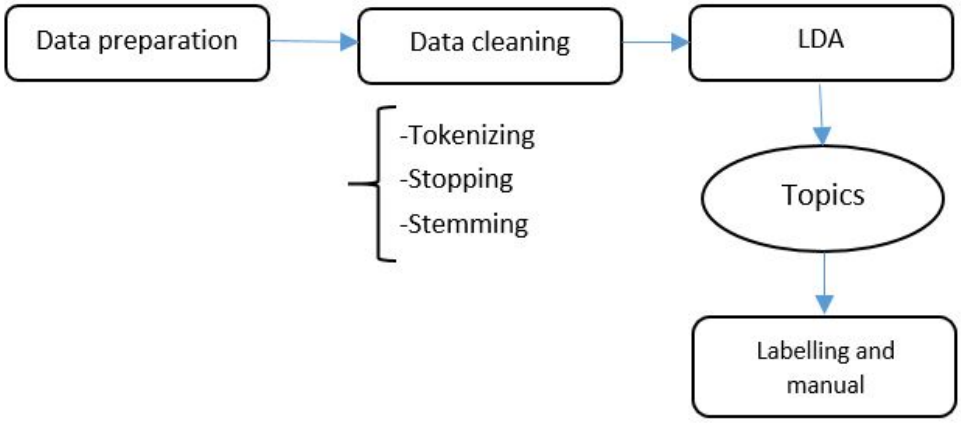

Figure 4.6: Topic modelling approach to detect projects types based on mining their description.

Swift projects are of the second type. Regarding to C# (37%), C++ (30%), Go (45%), and Ruby (35%) projects, the largest part of their projects are of the third type. Finally, 33% of C projects are of the fourth type, and 42% of Python projects are of the fifth type, as shown in Table 4.6.

	Project type ID				
Language	1	2	3	4	5
Java	11	47	20	17	6
JavaScript	40	16	19	14	12
Python	12	5	22	18	42
Objective-C	8	59	10	21	3
PHP	52	2	26	15	6
C	16	8	29	33	15
C#	25	4	37	24	10
C++	16	12	30	18	24
TypeScript	42	9	31	9	9
Go	25	3	45	14	12
Swift	12	52	17	11	8
Ruby	36	5	35	13	12

Table 4.6: Project types' distribution per language (percentage).

When projects were categorised based on language feature, the majority (56%) of statically typed language projects were of second and third type (28% each), whereas the largest portion of the dynamically typed ones were of the first type (31%). In the second group, the majority of strongly typed projects were of second (26%) and third (25%) types, however, 38% of the weakly typed ones were of the first type. In the third group of languages, 26% and 24% of the managed memory languages were of the first and third types respectively. Conversely, 37% of the unmanaged memory projects were of second type, as shown in Figure 4.7.

Category	Project Type ID				
	1	2	3	4	5
Type system I (safety)					
Static	18	28	28	16	9
Dynamic	31	14	21	16	19
Type system II					
Strong	16	26	25	16	17
Weak	38	12	22	16	11
Memory management					
Managed	26	19	24	15	15
UnManaged	11	37	18	22	11

Table 4.7: Project types' distribution per language group (percentage).

4.6 Discussion

We divide our discussion into 4 subsections, answering the four study questions:

RQ1. What are the most popular languages in developing OSS projects? Are they the same as in other indexes?

The most popular languages based on the number of projects per language where a main language can be identified are listed in Table 4.1 on page 48. As can be seen in the table, JavaScript and Java are the most popular ones, making up together about 50% of the total population, followed by Python (about 15% of the population). Go and Objective-C come next, together making up 15% of the total population.

In comparisons with other language popularity studies that have a dataset of more than 300 projects and list at least the top 10 languages (Delorey et al. 2007, Mayer & Bauer 2015, Sanatinia & Noubir 2016), we found similarities and differences in the listed languages. That is, although these studies cover different time periods and their definitions of popularity differ as well, JavaScript and Python made a strong appearance in each of them. They are the only two languages that made an appearance in each of the four studies. Java, C++, PHP and C made an appearance in three studies out of the four, and Objective-C, C# and CSS, made 2.

When compared with other language popularity indexes such as TIOBE[2] and PYPL[3] for the same period (Table 4.8), Java, JavaScript, Python, Objective-C, PHP, and C# made appearance in the three indexes. Whereas, C++ and C appeared in two lists out of the three. As in the popularity studies, those indexes have different definitions of popularity. TIOBE is based on the frequency of web searching about programming languages, whereas, PYPL uses Google Trends for developers' searches to indicate popularity of a language.

[2] tiobe.com/tiobe-index/
[3] pypl.github.io/PYPL.html

	TIOBE	PYPL	Github
1	Java	Java	JavaScript
2	C	PHP	Java
3	C#	Python	Python
4	C++	C#	Go
5	PHP	JavaScript	Objective-C
6	Python	Objective-C	Swift
7	JavaScript	Matlab	PHP
8	Visual Basic	C	Ruby
9	Perl	C++	C#
10	Objective-C	Swift	C++

Table 4.8: Most popular languages in TIOBE, PYPL, and Github.

The reasons behind the popularity of a language are varied; scalability and portability factors as in Java, the support of a large enterprise such as in Go, C#, and Swift the wide user community, libraries and tool support are some examples. However, investigating popularity-related factors are out of this research scope.

RQ2. What are current trends in language usage?
Individual languages, along with language groups, have shared a similar pattern of usage involving a gradual increase followed by faster decline, starting around 2016. This pattern of usage is affected by the popularity of Github as a hosting platform for open source projects. Although there has been a decline in the number of new projects hosted on GitHub, it is still the platform with the highest number of projects; thus, projects hosted there are good representation of current usage and practices of programming languages in OSS.

It is clear, however, that Java and JavaScript have considerably higher usage than other languages, followed by Python, as seen in Figure 4.2. The graph also shows that the introduction of some languages can affect others. This is the case of Swift, which was introduced in 2014 (*Swift.org* n.d.) and saw an increase in usage till the decline point of 2016, and Objective-C. This can be attributed to the fact that both Swift and Objective-C are used for Apple iOS applications development.

When language features were investigated, statistically significant differences were found between usage of statically and dynamically typed languages, strongly typed and weakly typed ones, and managed and unmanaged memory languages. That is, languages with managed memory had increased usage over the period included, and to a smaller degree strongly typed languages over the weakly typed ones, and the statically typed over the dynamically typed ones as illustrated in Figure 4.3. This pattern corresponds with what has been found by (Ray et al. 2014) based on mining Github repositories: strongly typed languages are better than weakly typed ones, statically typed are better than the

dynamically typed, and managed memory languages are better than the unmanaged in code quality.

RQ3. Which language combinations are most common?
Since the appearance of modern computers, progress has been made in designing various high-level programming languages. Nowadays, many languages are hybrid; they support different programming paradigms, and new features are being added to them continuously. In addition, it is also common to use multiple languages to develop a single software project. About 49% of the projects in the population of the study did not have a primary language (constituting 95% of the codebase). That is, the codebase of the projects is shared between different languages to a large extent. On average, the most popular projects are written in 4.15 languages. In the dataset where a main language can be identified, the average is 2.38 languages. Moreover, in the small dataset, although having a primary language, the majority of projects ($> 50\%$) are written in more than one language, except for projects where Java, PHP, and Ruby are the main one. HTML, Shell and CSS frequently co-occur with other languages in the investigated projects. The popular language combinations hold for both datasets for the same languages. The frequent combinations along with the projects' main language are shown in Table 4.9.

Project language	Combined with languages
JavaScript	HTML, CSS, Shell
Java	Shell, HTML, JavaScript
Python	Shell, HTML, JavaScript
Go	Shell, HTML, JavaScript
PHP	HTML, CSS, JavaScript
Objective-C	Ruby, C, Shell
Swift	Ruby, Objective-C, Shell
C#	PowerShell, Shell, HTML
Ruby	HTML, Shell, CSS
TypeScript	JavaScript, HTML, CSS
C++	C, Shell, Python
C	C++, Shell, Python

Table 4.9: The most popular languages and their most frequent combinations.

RQ4. What is the nature of the projects that have been written in these popular languages?
It is found that projects in the dataset are categorised into 5 types; web applications, mobile applications, web libraries, mobile services, and machine-learning and AI projects. This categorisation is based on mining and analysing projects' descriptions using natural language processing model, LDA, and is hence subjective. They are different than project categories in other software engineering works such as the ones by Capers Jones (2008), Sommerville (2011) and Ray et al. (2014). In the first two, the categorisations are classic and may not be applicable in the context of OSS projects. Hence, the difference can

be attributed to the open nature of the projects included in this dataset. Moreover, although the Ray et al. (2014) study is based on OSS projects gathered from Github repositories and used a similar technique for mining project descriptions, after applying the LDA algorithm, they found 30 types and reduced them to 6 after manual inspection. In this work, we found that after applying the LDA model, the optimal number of types is 5. Nevertheless, the labeling and inferring the category from keywords is subjective, and hence, different from one work to another. The types they found were: application, database, code analyzer, middleware, library, framework and other.

Nevertheless, we found that the largest portion of Java, Objective-C, Swift and C projects in our dataset were mobile-related. Conversely, the largest part of JavaScript, PHP, C#, C++, TypeScript, Ruby and Go are categorised as web-related. Finally, the largest part of Python is AI-related. When languages were classified into groups based on their features, the majority of statically and strongly typed languages projects were divided between web and mobile-related ones. While, the majority (>50%) of dynamically and weakly typed languages projects were web-related. When languages were classified based on memory management, it was found that the majority of managed memory projects were web related, whereas, the majority of the unmanaged language projects were categorised as mobile-related.

4.7 Related work

Studies of programming languages that are based on mining software repositories are varied. They cover different language-related issues and can be broadly categorised as (1) feature adoption studies (2) language usage, trend and popularity studies and (3) documents and bug reports analysis studies to derive insights about artifacts qualities that have been written in certain languages. The ones that fall under the first type include the work of Parnin et al. (2011) that mined 20 Java repositories to investigate the adoption of generics by developers and Okur & Dig (2012) which investigated the usage of another language feature, the parallel libraries based on analyzing 655 C# projects.

In the second category, on which this work relies, a study by Karus & Gall (2011) on 22 projects over 12 years to investigate the language usage evolution, with a particular interest in special-purpose languages such as XSL and XML, found that the size of the language share in the projects' codebase differs significantly. Moreover, the usage of XSL and XML has shown a significant increase. In addition, they have found some common co-changes in the dataset files such as JavaScript and XSL, and Java and XML files. Our work, however, focused more on general-purpose languages rather than the special-purpose ones. A similar work (Chen et al. 2005) empirically analyzed the evolution of 17 languages during the years 1993, 1998, and 2003. The study produced in a quantitative

model of intrinsic and extrinsic factors for the evolution of programming languages trends. However, since our study is an observational one with no control over the data we could not use this model in our analysis.

Further research (Meyerovich & Rabkin 2013) investigated language adoption empirically based on multiple surveys and including over 200,000 SourceForge projects to identify factors affecting language adoption. The study found that language adoption follows a power low, and that the most significant factor for developers in choosing the language is open source libraries, followed by code extendability and reusability, and social factors such as team experience and personal affinity. (Sanatinia & Noubir 2016)ś study also investigated programming languages' popularity and language co-concurrences in the top 1,000 repositories of Github. However, their dataset included the top languages in terms of the highest number of projects, whereas our selection criteria were based on the number of projects that received 500 stars at least. We set this threshold in an attempt to have a sample of high-quality projects that could serve as a good proxy for OSS projects, considering that 80% of Github repositories have $\leq$13 stars.

Another study that has investigated language combinations in open source software is by Mayer & Bauer (2015) mining 1,150 OSS projects and finding that the projects' codebase was shared between a mean of 5 languages. Moreover, the study showed that the number of languages is affected significantly by size, number of commits, and the main language in the project. In Delorey et al. (2007), 9,999 OSS projects from SourceForge were mined to determine the most popular languages hosted on the platform, along with examining the impact programming languages might have on code writing. In 2014, a large-scale study (Ray et al. 2014) investigating 729 projects in 17 languages based on mining Github repositories for the effect of languages on code quality found that languages have significantly affected quality; however, the effect size was modest.

4.8 Summary and conclusions

Open source software (OSS) hosting platforms, such as Github, are producing valuable and rich datasets for mining studies. They are hosting popular, high-quality software projects, providing data about the development history and its related activities including code writing, team communications, and artifact debugging. Such projects are a reasonable proxy for open source software and reflect current practices in OSS community. Thus, this study was based on mining and analyzing the most popular 15,000 repositories on Github to obtain insights about current trends and directions in programming languages that are used in developing OSS projects.

The results have shown that popular Github projects are written in 64 languages. The top 15 based on number of projects per language were mainly general-purpose, whereas the top 15 based on the number of stars (favouring by registered users) were mainly

special-purpose ones with a considerably smaller number of projects. It is also clear that languages with managed memory have increased in usage over the period included, and to a smaller degree strongly typed languages and system programming ones over the weakly typed and scripting ones respectively. Future research into language design should take note of theses trends. Further research is required to investigate and understand them in more depth. Moreover, the codebase of the projects in the dataset is shared between different languages to a considerable extent. That is, 49% of the projects in the whole dataset did not have a primary language; this comprised 95% of the codebase. Thus, the support for language interoperability should be an important goal for language and runtime tool designers as this would improve reusability and efficiency of the overall development process. Additionally, although the included data are for actual projects and would provide significant insight about the observed trends in software development, the generalization of their findings is problematic since they are based solely on open source projects from Github repositories in the reported period. The languages included have been identified as reported by Github and languages' popularity in the study is affected by Github usage.

Chapter 5

Programming language features usage

In this chapter, the second research objective is fulfilled, in which language features usage is investigated. In Section 5.1, the chapter objectives are listed. Next, in Section 5.2 the followed methodology is explained. In Section 5.3 the results are provided per mining method. Then, in Section 5.4 the study questions are answered and discussed. The related work is covered in Section 5.5 and the study work is concluded in Section 5.6.

5.1 Objectives

There is a lack of empirical work investigating how features are used by developers in different languages. Is there a tendency to use the same features regardless of the chosen language for the job? Is there any statistical link between their usage and language design?. The goal of this chapter is to quantify language features usage at a large scale. We study the practice of developers through mining source code files of OSS projects to understand the current state of the practice. This is to help language designers understand the user community, how their features are used in the popular languages in real projects, and whether there is any difference in feature usage between languages and between language groups that would suggest a link between the two. Moreover, adopting data mining methods in this context encourages evidence-based design of programming language rather than anecdotes and assumed needs.

More importantly, to understand language impact, we investigate the relationship between language design and feature usage. That is, when languages offer the same features, is there any statistical difference in their usage? The existence of such differences would suggest a substantial association between the two.

The overall aim of this chapter is to identify how language features are used in practice and whether there is a significant link between language design and feature usage. This investigation is carried out in a large scale setting using statistical methods by accounting for confounding factors such as project size and type, per language and language group. Thus, we investigate the following questions:

1. How frequently are language features used by developers in practice?
   The list of the selected features for this study is based on a prior work by (Meyerovich & Rabkin 2013). We approach this question to validate our selection empirically through investigating the practice of programming languages against the perception. The features that are included in this study are: inheritance, class interface, exceptions, threads, anonymous functions (as a proxy for higher-order functions), and generics. As we investigate the usage of language features *per se*, rather than that of the compiler, the editor, or the development tool, we included only language *intrinsic* features.

2. What are the most frequently used features per language? Is there any difference in feature usage between languages?
   As the overall aim of this research is to study programming language effect, we need to investigate whether languages are used in a similar manner. Little is known about how developers adopt features in practice. Thus, this inquiry is intended to inspect when languages provide the same features, and do developers use the same features regardless of the chosen language, or is there a tendency to use the same features at the same frequency in all of them. Such an inquiry implies an understanding of the user community and of how its members implement language features per individual languages.

3. What are the most frequently used features per language group? Is there an association between language design and feature usage?
   Rather than inspecting languages individually, for this question we group languages according to their type system and memory management technique. This allows us to inspect whether there is an association between language design and feature usage, and whether there is a tendency to use certain features more in some groups than in others. Furthermore, such an approach allows us to check if language design encourages using certain features more than others. Significant differences between the included groups would suggest a link between language design and feature usage.

Thus, we propose the following objectives:

- Analyse language intrinsic feature usage based on mining the projects' source codes.
- Examine how frequently such features are used in practice, as per language and language group, and to what extent they differ or are similar.

- Investigate statistical relationships between language design and feature usage, while accounting for confounding factors such as project size and type.

The chapter contributions are:

- The first study of such a large volume, investigating the usage of 6 features in 11 languages.
- Adopting text mining methods on source codes to find sufficient empirical evidence in the dataset.
- Accounting for confounding factors such as project size and type when investigating a language's effect on feature usage.
- Inform the state of the practice of language features usage, using rigid statistical methods to support findings.

5.2 Methodology

The methodology is based on mining source code files to inspect feature usage. We adopted what is already established in the data mining field, that is, a classic mining process goes through three fundamental steps: (a) data gathering, (b) data pre-processing and cleaning, and (c) information extraction. Here, for the first step, the source code files for the included projects were downloaded from Github in a compressed form to be searched for the features. Then, only the main language files were extracted. That is, if the project's main language is Java, which makes 95% of the total source code files, and was combined with other language and documentation files, only Java's files are extracted. For data pre-processing and cleaning, documentation annotations and the different types of comments were removed from the files. Additionally, strings, HTML, and XML tags were detected and removed to enhance the mining process through avoiding interference with language features when source codes are scanned thereby minimizing the size of the processed files and time of feature detection.

As source codes are texts with special characteristics, text mining methods were utilised to detect features and extract needed information. Thus, we use three text mining methods here:

1. Binary term occurrence: as the name indicates, this method reveals whether or not a term has occurred in the document, and therefore detects whether or not the feature has been used in the processed project.
2. Term occurrence: indicates the number of times a term occurs in the document, and therefore detects the number of feature occurrences per project.

3. Term frequency: the number of times a term occurs normalised by the occurrence of other terms in the processed document.

Features have special constructs per language, and language users can derive different forms of each within the language. Thus, we mined source code files using regular expressions to detect the different variations of features. A regular expression is a sequence of characters that follows specific rules to describe a pattern of a set of strings (Sebesta 2012). Hence, regular expressions can be used to match the occurrence of features. The overall methodology used in this chapter is summarised in Figure 5.1.

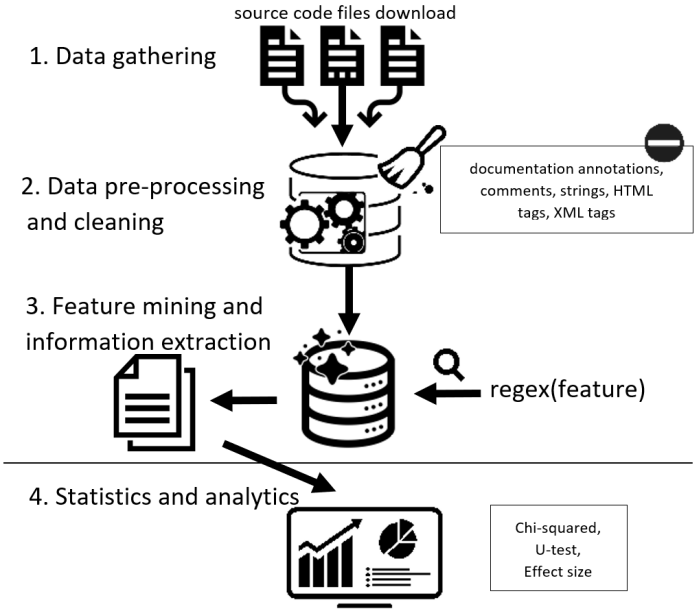

Figure 5.1: Feature mining methodology

After mining the source codes for features, we used statistical methods to analyse the outcomes. For binary term occurrence, we used Chi-squared for hypothesis testing and effect size for investigating the magnitude of an existing difference, if any, as Chi-squared is used for binary and categorical data (Field et al. 2012). For term occurrence and term frequency, we use the U-test for hypothesis testing, as the data are not normally distributed (Field et al. 2012), along with effect size and descriptive statistics. In addition, we account for different sizes and types of projects included in the dataset to observe if the outcomes hold for the different variations.

5.2.1 Feature selection

The features that are included in this study are: inheritance, class interface, exceptions, threads, anonymous functions (as a proxy for higher-order functions), and generics. The features are defined as follows:

Inheritance: this is a feature that is traditionally associated with object-oriented programming (OOP). A subtype (subclass) can inherit all the methods and variable components in order to benefit from a previously written type (or class), and can also add additional ones. Languages can have strict inheritance in which a subtype can only have one super type from which to inherit, known as single inheritance. Some languages also support multi-inheritance, in which a single subtype can inherit from many super types (O'Regan 2008).

Class interface: this is another feature that is traditionally associated with OOP. Interfaces enforce certain behaviour on a user-defined type that implements them. The implementing types are required to provide a definition for the behaviour (Petricek & Skeet 2009).

Exception: this is an abnormal event that occurs when the program's normal flow is disrupted. A program requires the following of a special mechanism to deal with this exceptional event (W.Sebesta 2008).

Threads: a thread is a lightweight flow of control inside a program that shares resources with another thread. Heavy duty programs need to split execution into a number of simultaneous threads (W.Sebesta 2008).

Anonymous functions: traditionally associated with functional programming, anonymous functions are nameless constructs that can be passed to and returned from higher-order functions (Petricek & Skeet 2009).

Generic: program units that are defined in terms of types-to-be-specified later manner. They enable multiple versions of the same construct to be instantiated on different data types to minimise code repetition. Such mechanisms sometimes called parametric polymorphism (O'Regan 2008).

The selected features are based on a study of 415 respondents who ranked the top features they value the most (Meyerovich & Rabkin 2013). As we investigate the usage of language features *per se*, rather than that of the compiler, the editor, or the development tool, we included only language *intrinsic* features. Furthermore, in an attempt to objectively compare the usage of features between languages, the included features should be supported *natively* by most of the languages in the dataset. In programming languages, features that are not natively supported can be simulated using other features. Thus, a

language has to provide a construct (or unit) that facilitates delivery of the aforementioned definitions of features, in a direct manner to be included in this study. Table 5.1 shows a summary of the included features and their associated language support.

Language	Inheritance	Exceptions	Threads	Anonymous func.	Generics	interface
Java	Y	Y	Y	Y	Y	Y
JavaScritp	N	Y	N	Y	N	N
Python	Y	Y	Y	Y	N	N
Go	N	Y	Y	Y	N	Y
Obj-C	Y	Y	Y	Y	Y	Y
PHP	Y	Y	Y	Y	N	Y
Swift	Y	Y	N	N	Y	Y
Ruby	Y	Y	Y	Y	N	N
Csharp	Y	Y	Y	Y	Y	Y
C++	Y	Y	Y	Y	Y	N
TypeScript	Y	Y	N	Y	Y	N

Table 5.1: Programming Language Support for Features

As can be seen in Table 5.1, the language support for features is arbitrary. Exceptions constitute the feature that is supported by all of the included languages (100%), followed by anonymous functions with a support of 91%. Inheritance ranks next (82%), followed by threads (73%), interfaces (64%), and finally generics at 55% support in the languages. As per language, C is the only language in the dataset that does not support any of the included features. Thus, in this chapter, we excluded C from analysis and discussion. JavaScript is the language with the least supporting features, with only two features supported, whereas, in other languages, at least 4 features are supported (36.4% of languages). Five feature are supported in 27.3% of languages, and a total support is found in another 27.3%.

To mine projects' source code for features occurrence, we tried existing text mining tools such as RapidMiner, the freely available versions. Many of these tools did not process source code files and/or they exhibited memory issues, as some projects are of a relatively large size and have multiple files. Therefore, we mined the source codes using *grep*. Grep is a line-based utility that can search and match regex pattern. In this manner, we were able to overcome issues of reading source code files and stack overflow arising from using regex engines.

5.3 Results

We present the results here as per the mining method, thus, there are three subsections. For each, we go through the results per language, followed by the results per language group, in which we also present the results of testing the chapter's main hypothesis. The hypothesis is tested in each subsection per language classification. Then, the hypothesis

is tested again after accounting for different project sizes and types. The hypothesis as follows:

Hypothesis: There is no association between feature usage and language design.

5.3.1 Binary term occurrence

When mining language features using binary term occurrence method, inheritance is found to be the most frequently used, having been used in 88% of the projects in the dataset, followed by exceptions (used in 80%), interfaces (78%), and generics (76%). Anonymous function comes after that, used in 71%, and finally threads, which has been used in 45% of the included projects. These percentages are calculated based on projects in which the language supports the feature shown in Figure 5.2.

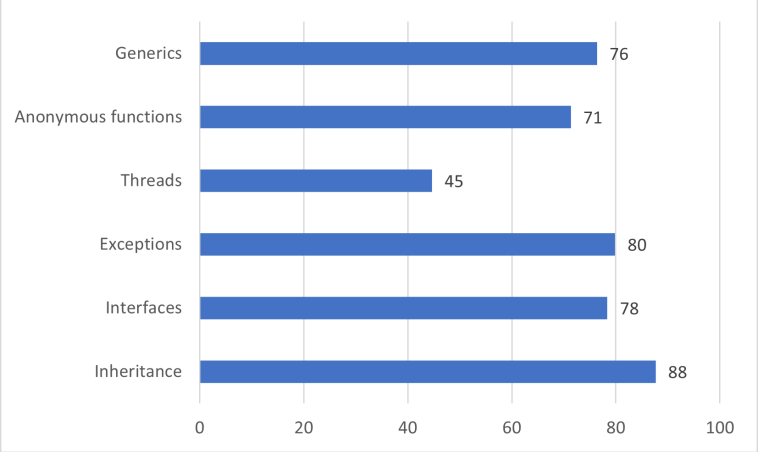

Figure 5.2: Feature Usage in Projects (%, percentage).

As listed in Table 5.2, as per language, inheritance is the most frequently used feature in Objective-C (for which it is used in 99.7% of projects), Java (99.2%), Swift (92%), PHP (88.6%), C++ (87.1%), and Ruby(80%). Anonymous functions constitute the most frequently used in Go (99.2% of projects), JavaScript (99.1%), and TypeScript in which it has been detected in all of its projects. Exceptions constitute the most frequently used feature in Python, with a usage rate of 92.5% in its projects and C# (49.7%). In addition to exceptions, C#'s most frequently used feature is shared with generics, with the same usage rate, of 49.7%. Threads and interfaces were not found to constitute the most frequently used features in any of the included languages.

When languages are categorised per group, for the first classification, type system I, inheritance is the most used feature in the statically typed group (used in 88.1% of projects). By contrast, anonymous functions constitute the least frequently used feature with a usage rate of 47.4%. In the dynamically typed group, anonymous functions constitute the most frequently used feature (86.8%) and threads constitute the least frequently used, with a 22.5% usage, as shown in Figure 5.3.

Language	Inheritance	Interfaces	Exceptions	Threads	Anonymous Fnc.	Generics
Java	99.2	87.1	90.6	58.6	22.6	96.7
JavaScript	n/a	n/a	83.9	n/a	99.1	n/a
Python	82.7	n/a	92.5	32.6	73.2	n/a
Go	n/a	84.9	95.0	83.2	99.2	n/a
Obj-C	99.7	76.6	41.9	34.1	95.8	45.8
PHP	88.6	64.1	84.5	1.0	79.0	n/a
Swift	92.0	46.2	24.1	n/a	n/a	47.6
Ruby	80.0	n/a	66.2	12.4	45.5	n/a
Csharp	49.4	45.1	49.7	28.4	42.3	49.7
Cpp	87.1	n/a	66.3	65.3	26.7	86.1
TypeScript	89.8	94.9	90.8	n/a	100.0	95.9

Table 5.2: Feature Usage per Language (%, percentage)

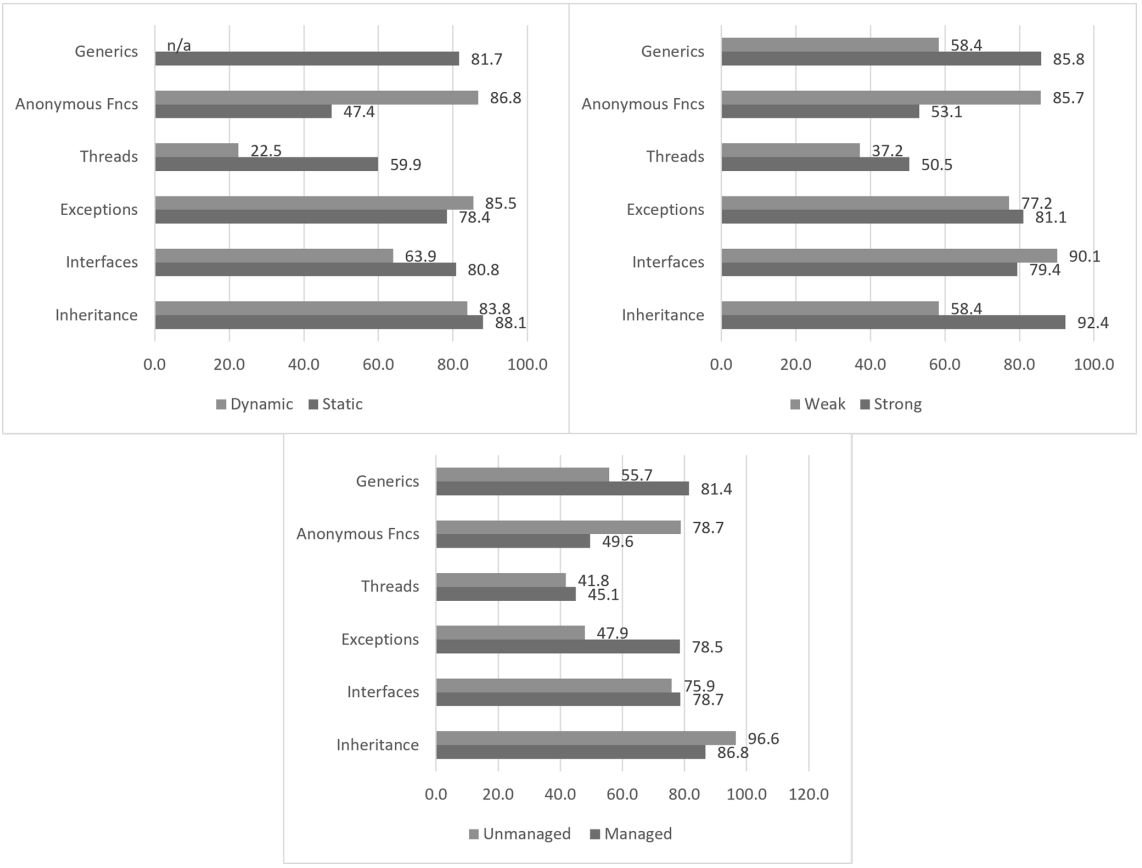

Figure 5.3: Feature Usage per Language Group (%percentage)

To further inspect the relationship between the feature usage and the language type (or group), we test the following hypothesis,

- *Hypothesis 1: There is no difference between statically and dynamically typed languages in feature usage.*

The two variables here are binary, the feature usage as a binary occurrence, and the language group (static vs dynamic types). In addition, the usage data are not normally distributed. Thus, we select the non-parametric Chi-square test of independence to evaluate the association between the two variables, that is, to determine whether the relationship is statistically significant or not (Field et al. 2012). The test was applied as per feature between the two groups, and the results are listed next in Table 5.3.

Feature	p	Chi-Square	df	Effect Size
Inheritance	0.00	202.31	2.00	0.25 M
Interfaces	0.00	3459.66	2.00	0.84 L
Exceptions	0.00	387.20	2.00	0.27 M
Threads	0.00	549.34	2.00	0.40 L
Anonymous Fncs	0.00	728.62	2.00	0.38 L
Generics	n/a	n/a	n/a	n/a n/a

Table 5.3: Hypothesis test statistics for static and dynamic languages (Binary mining).

We have excluded generics from the test, as in the dynamically typed group there is no usage for it. This is because the languages in this group do not support generics natively. For all of the other five features, the p-value is less than our chosen significance level (alpha=0.05). Hence, the null hypothesis is rejected and we can state that there is a statistically significant association between feature usage and language group ($p<0.05$) for all of the five tested features. In addition to significance, we also investigated the effect size in order to report the magnitude of the detected difference between the two groups. The effect size is reported here using *Cramer's V*. The results suggest a medium-sized effect between the two groups in inheritance, exceptions, and generics usage, whilst a large-sized effect was seen in the case of threads, anonymous functions, and interfaces usage.

The same hypothesis was tested for the different project sizes and different project types in this classification. As for statistical significance, when accounting for different projects sizes, generics was excluded from the test, as in the dynamically typed group there is no usage for it. Also, the large and very large sizes in inheritance were excluded as no projects were found in these two groups for this feature. Otherwise, the results hold in threads, anonymous functions, and interfaces for all sizes. Results for effect size for the different project sizes are rather arbitrary between one size to another, and only hold in case of interfaces. That is, the effect size is large for all of the different project sizes.

When accounting for project types, results for significance hold for all types for all features except type 5 in inheritance and anonymous functions. However, results for effect size for the different types hold in exceptions (tiny), threads (large), and interfaces (large). The full results for accounting for different project types and sizes are listed Appendix B.

In the second classification, type system II, inheritance is the most used feature in the strongly typed languages group, with a usage of 92.4% of the included projects, whilst, threads is the least frequently used with a usage of 50.5%. Interfaces is the most frequently used feature in the weakly typed language group, with a usage rate of 90.1%, whereas, similar to the strongly typed languages, threads is the least frequently used, with 37.2% usage. In order to investigate the association between the feature usage and language type, we use statistical hypothesis testing to determine whether the association is significant or not, per feature. The hypothesis we investigate is as follows:

- *Hypothesis 2: There is no difference between strongly and weakly typed languages in feature usage.*

Again, for the same previously mentioned reasons for testing Hypothesis 1,the Chi-square test of independence has been used and the testing results are listed in Table 5.4. The results show a statistically significant difference in all of the 6 included features for the our chosen alpha=0.05. Thus, the null hypothesis is rejected for all of the included features. Consequently, we can conclude there is a statistically significant association between feature usage and language type II at $p<0.05$ for the tested features. However, a trivial effect size has been found in the data for inheritance and exceptions usage, a small-sized effect in case of generics, a medium one for threads and anonymous functions, and a large effect found in interfaces usage between the two groups.

Feature	p	Chi-Square	df	Effect Size
Inheritance	0.00	11.89	2.00	0.06 <S
Interfaces	0.00	1690.05	2.00	0.59 L
Exceptions	0.00	17.18	2.00	0.06 <S
Threads	0.00	280.80	2.00	0.29 M
Anonymous Fncs	0.00	611.49	2.00	0.35 M
Generics	0.00	55.13	1.00	0.16 S

Table 5.4: Hypothesis test statistics for strong and weak languages (Binary mining).

Similarly, the hypothesis was tested for the different project sizes and types for significance and the magnitude of the difference, if any. The full results are listed Appendix B. When accounting for different project sizes, large and very large sizes for inheritance usage were excluded as well as very large sizes for generics usage, as no projects were in this category. Otherwise, statistically significant results hold for inheritance, anonymous function, and interfaces. For effect size, they hold only for interfaces (large). Effects are arbitrary otherwise for the different project sizes. As for the different project types, statistically significant results hold for inheritance, threads, anonymous for all types, and interfaces (for all except type 5) as not enough projects are found. The effect size holds for inheritance (large) and threads (large). Otherwise, the effect size is arbitrary for the different project types.

Finally, when languages are classified based on memory management, inheritance is most frequently used feature in both the managed and unmanaged language groups, with a usage of 86.8% and 96.6%, respectively. Exceptions rank second for the managed memory languages, whereas, anonymous functions constitute the second most frequently used in the unmanaged memory languages group. Threads constitute the least frequently used feature in the two groups. To further investigate the relationship between feature usage and the two groups of memory management languages, we test the next hypothesis,

- *Hypothesis 3: There is no difference between memory managed and unmanaged languages in feature usage.*

Similarly, the Chi-square test of independence is used here for the association significance, and the effect size for the results is investigated using Cramer's V value. The Chi-squared test statistics and the effect sizes are listed in Table 5.5. The results show a statistically significant difference between the two groups of memory management for all of the included features, at $p<0.05$. Thus, we further report the effect size to deduce the strength of the resulting difference. The findings suggest a trivial-sized effect for inheritance, threads, and anonymous functions usage between the two groups. However, there is a small-sized effect for exceptions and interfaces usage, and a medium-sized one for using generics between the groups.

Feature	p	Chi2	df	Effect Size	
Inheritance	0.00	11.42	1.00	0.06	<S
Interfaces	0.00	106.85	2.00	0.14	S
Exceptions	0.00	381.61	1.00	.270 S	S
Threads	0.02	5.56	1.00	0.04	<S
Anonymous Fncs	0.02	5.14	1.00	0.03	<S
Generics	0.00	300.99	1.00	.379 M	M

Table 5.5: Hypothesis test statistics for memory managed and unmanaged languages (Binary mining).

When accounting for project sizes, the results of statistical significance hold for generics, except for very large projects, as statistics were not reported because no projects were found for this size group. However, size effects are rather arbitrary and do not hold for the different sizes. When accounting for different projects types, statistically significance results hold for all features except anonymous functions and interfaces. Effect size only holds in case of generics (large), otherwise is arbitrary for the different types. The results for accounting for different project types and sizes are listed Appendix B.

5.3.2 Term occurrence

As the feature occurrence data per language are not normally distributed, the arithmetic median is chosen to describe the average. Table 5.6 shows the feature occurrence average

Language	Inheritance	Interfaces	Exceptions	Threads	Anonymous functions	Generics
Java	37	6	19	1	0	60
JavaScript	n/a	n/a	12	n/a	359	n/a
Python	18	n/a	35	0	4	n/a
Go	n/a	8	59.5	12	407	n/a
Objective-C	19	3	0	0	33.5	0
PHP	14	2	16	0	7	n/a
Swift	3	0	0	n/a	n/a	0
Ruby	6	n/a	3	0	0	n/a
C#	243	35.5	294	1	14	1422
C++	38.5	n/a	13.5	5	0	121
TypeScript	32	38	26	n/a	706.5	144.5

Table 5.6: Average Feature Occurrence per Language (Median).

per language. As listed in the table, generics is the feature with the highest average in Java (60 occurrences), next is C# (1422), and then C++ (121). Anonymous functions is the feature with the highest average in each of JavaScript, with 359 occurrences, Go with 407 occurrences, and TypeScript with an average of 707. In Python and PHP it is exceptions with averages of 35 and 16, respectively, whilst, inheritance is the feature with the highest average of occurrence in Ruby only (6 occurrence).

Next, when languages are categorised into groups based on their design, the occurrences data are compared and checked for statistically significance differences using Mann-Whitney U-test as the data is not normally distributed, that is, to inspect the relationship between language features usage and language group, followed by effect size (Field et al. 2012).

For the first language group, type system I, the statically and dynamically typed languages have different features that are most frequently used. While it is generics in the statically typed group with an average of 73 occurrences, anonymous functions has with the highest average in the dynamically typed language groups, with 55 occurrences, as displayed in Figure 5.4. The least frequently used feature in the two groups is threads, with a median of 2 and 0 in the statically and dynamically groups, respectively.

Then, the usage data used to statistically test the following hypothesis:

- *Hypothesis 1.* There is no difference between statically and dynamically typed languages in feature usage.

A non-parametric U-test at 95% confidence was used here. The results show that there is a statistically significant difference in the feature usage occurrences between the statically typed and the dynamically typed languages at $p<.05$ in five features: inheritance, exceptions, threads, anonymous functions, and interfaces. Thus, the null hypothesis is

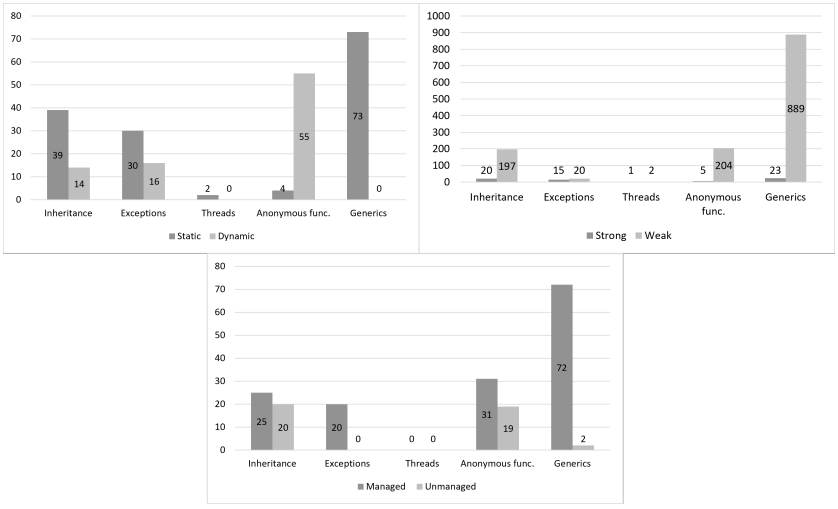

Figure 5.4: Average Feature Occurrence per Language Group (Median).

rejected for those five features. However, for generics, the Mann-Whitney test could not be performed, as the dynamically-typed languages group is empty. The U-test statistics are listed in Table 5.7.

For 5 features; inheritance, exceptions, threads, anonymous functions, and interfaces, the p-value is less than our chosen significance level alpha = 0.05. Hence, the null hypothesis is rejected and we conclude that there is a relationship between feature usage and language type for those features. Thus, there is a statistically significant association between feature usage and language type ($p<.05$) for five features out of the included six. In addition to the U-test, we also investigated the effect size to report the magnitude of the reported difference between the two groups. The results show a small-sized effect in inheritance, exceptions, anonymous functions and interfaces. Whereas, a medium-sized effect found between the two groups in threads usage.

When accounting for project size, results hold for all features in small and very large projects. They also hold for threads and anonymous functions for all sizes, and in interfaces except for tiny projects. As for effect size, it holds for threads for the different sizes (medium effect). Otherwise, it is rather arbitrary.

When we control for project types, results of statistical significance hold in inheritance and exceptions for the different types. Effect size results hold in inheritance except type1 and type3. In exceptions the effect is small per type, whereas, it is tiny when in general, i.e. when we do not control for types.

In the second categorisation, type system II, the two groups share the same feature as the most frequently used one, generics, with an average of 23 and 889 occurrences for strongly and weakly typed languages, respectively. Again, the least frequently used is threads with a median of 1 for the strongly typed and 2 for the weakly typed group, as

Features	p	Z	N	r	Effect Size
		Static vs Dynamic Languages			
Inheritance	0.000	-13.377	2975	-0.245	S
Exceptions	0.000	-8.626	4946	-0.123	S
Threads	0.000	-23.67	3142	-0.422	M
Anonymous func.	0.000	-17.724	4734	-0.258	S
Generics	n/a	n/a	1782	n/a	
Interface	0.000	-9.211	2338	-0.190	S
		Strong vs Weak Languages			
Inheritance	0.000	-16.387	2993	-0.300	S
Exceptions	0.000	-3.831	4964	-0.054	T
Threads	0.000	-3.883	3160	-0.069	T
Anonymous func.	0.000	-30.139	4752	-0.437	M
Generics	0.000	-19.237	2090	-0.421	M
Interface	0.000	-8.998	2358	-0.185	S
		Memory managed vs unmanaged Languages			
Inheritance	0.467	-0.727	3283	-0.013	T
Exceptions	0	-17.554	5254	-0.242	S
Threads	0.113	-1.586	3450	-0.027	T
Anonymous func.	0	-3.885	5042	-0.055	T
Generics	0	-15.655	2090	-0.342	M
Interface	0.000	-5.267	2649	-0.102	S

Table 5.7: U-test statistics per language group (occurrence mining).

listed in Figure 5.4. The occurrence data for the two groups are then used to test the following hypothesis.

- *Hypothesis 2.* There is no difference between strongly and weakly typed languages in feature usage.

Similarly, a U-test at 95% confidence was used to test the hypothesis. The results show that there is a statistically significant difference in the feature usage occurrences between strongly typed languages and the weakly typed ones at $p<0.05$ for all of the 6 features, as the p-value is less than the chosen significance alpha (0.05). Hence, the null hypothesis is rejected and a statistically significant association between feature usage and language group ($p < .05$) for all of the six tested features exists, as listed in Table 5.7. The results suggested a tiny-sized effect in exceptions and threads, and a small-sized effect between groups in inheritance and interfaces usage, whereas, a medium-sized effect is found in the case of using anonymous functions, and generics. When accounting for project sizes, results hold for all features in the large-sized for anonymous functions usage for the different project sizes. In inheritance usage, results hold for all sizes except the very large ones. Again, in threads they hold for all sizes except the very large projects. As for the effect size, results are rather arbitrary; mainly they are of small and medium effect for the different project sizes.

When controlled for the project types, significant results hold for anonymous functions and generics for all types. As for effect size, they hold in anonymous functions (medium) except type2 (large) and type3 (small). In generics, they hold in types (medium) except type2 and type3, in which the size is small.

As for the third categorisation, the managed and unmanaged memory languages have different most frequently used features. The highest occurrence feature is generics in the managed language group with an average of 72 occurrence, whilst it is inheritance with an average occurrence of 20 in the unmanaged group. Again, for the least used feature it is threads for the two groups along with exceptions for the unmanaged memory group, as displayed in Figure 5.4. Next, the usage data are compared and used to test the following hypothesis.

- *Hypothesis 3.* There is no difference in usage between the managed and unmanaged memory languages.

Again, the Mann-Whitney U-test is used to test the hypothesis. The results show a statistically significant difference in four features: exceptions, anonymous function, generics, and interfaces at $p<0.05$. Thus, the null hypothesis is rejected for those four, whereas, no difference was found in inheritance and threads usages between the groups for the chosen alpha, and the null hypothesis is retained for those two. Based on the results, we can state that there is a statistically significant association between feature usage occurrence and memory management in exceptions, anonymous function, generics, and interfaces. However, the effect size is found to be trivial in the data for anonymous functions usage. In the case of exceptions and interfaces, it is a small-sized effect, and medium-sized for generics. The related statistics results are listed in Table 5.7.

When accounting for projects size, results of statistical significance hold for exceptions and generics for all sizes. However, the effect size holds in tiny projects for exception usage (small), in large projects for generics (medium), and in very large projects for both exceptions and generics. Otherwise, an arbitrary effect sizes found for the different project sizes. As for project types, statistical significance holds in exceptions except type5, anonymous functions except type4, and generics except for type5. The effect size results hold in exceptions except type1 (tiny), otherwise, the results hold as small effects. For generics, the effect size only holds in type1 (small), and for anonymous functions it is rather arbitrary in the different project types.

5.3.3 Term frequency

As with feature occurrences, feature frequency usage data are not normally distributed. Thus, the median is selected to describe the average (Field et al. 2012). As shown

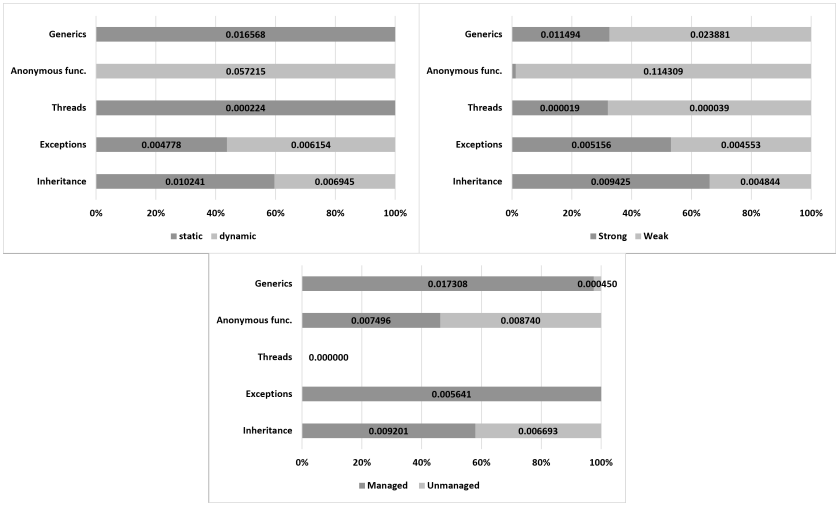

Figure 5.5: Average Feature Frequency per Language Group (Median).

in Table 5.8, anonymous functions constitute the feature with the highest average in JavaScript, Go, Objective-C, and TypeScript. Inheritance is the highest in PHP, Swift, and Ruby. In Java, C#, and C++ it is generics, whereas exceptions is the feature with the highest average of frequencies in Python. Threads has not made it to the highest average list in any of the included languages. In fact, threads is the feature with the lowest frequency average in Python, Go, Objective-c, PHP, Ruby, and C#.

Language	Inheritance	Interfaces	Exceptions	Threads	Anonymous Functions	Generics
Java	0.0110064	0.0015652	0.0056258	0.0001514	0	0.0175285
JavaScript	n/a	n/a	0.004268	n/a	0.1296646	n/a
Python	0.0064549	n/a	0.0129445	0	0.0011513	n/a
Go	n/a	0.0007519	0.0049412	0.0006809	0.0381545	n/a
Objective-C	0.0078128	0.0009712	0	0	0.0142172	0
PHP	0.0073294	0.0005916	0.0066192	0	0.0030264	n/a
Swift	0.0196025	0	0	n/a	n/a	0
Ruby	0.0099048	n/a	0.0049562	0	0	n/a
C#	0.0062048	0.0007071	0.008599	0.0000248	0.0003256	0.0336887
C++	0.0026661	n/a	0.0005219	0.0002245	0	0.0053858
TypeScript	0.0039408	0.0050547	0.0043319	n/a	0.0846009	0.0193486

Table 5.8: Average Feature Frequency per Language (Median).

Per language group, in the first categorisation, type system I, the feature with the highest average of frequency in the statically and dynamically typed languages is different, as seen in Figure 5.5. It is generics in the statically typed group, whilst, anonymous functions is the one with the highest average in the dynamically typed languages. The least frequently used feature in the statically typed group is anonymous functions, whereas it is threads in the dynamically typed group.

To further investigate the difference, the frequency usage data has been used to statistically test the following hypothesis.

- *Hypothesis 1.* There is no difference in feature usage frequencies between statically and dynamically typed languages.

Mann-Whitney U-test at 95% confidence was used to test the hypothesis (Field et al. 2012). The results show that there is a statistically significant difference in the feature usage frequencies between the statically and dynamically typed languages at $p<0.05$ in five features; inheritance, exceptions, threads, anonymous functions, and interfaces as listed in Table 5.9. Therefore, the null hypothesis is rejected for those features. Mann-Whitney test could not be performed for generics as the dynamically-typed languages group is empty. Thus, we can state that there is a statistically significant association between feature usage frequencies and languages groups of the first categorisation, type I, in ; inheritance, exceptions, threads, anonymous functions, and interfaces. For the effect size, the results suggested a tiny-sized effect in exceptions, small-sized effect between the two groups in inheritance and interfaces, a medium-sized effect in threads and anonymous functions.

When accounting for project size, significant results hold for all features in the small-sized projects, for threads and anonymous functions for all sizes. In using interfaces, they hold for all sizes except tiny projects. The effect size results hold for threads for all project sizes. In anonymous functions usage, effect size results hold (medium) for all but small projects (large).

When accounting for the different types, significant results hold for type4 for all features, and for anonymous functions for all types and threads except for type2. As for effect size, they hold for type4 for all features, otherwise it is rather arbitrary.

In the second categorisation, type system II, the two features with the highest average frequency per group are the same two in type system I, that is, generics and anonymous functions. Generics is the highest average feature in the strongly typed languages group, whilst, it is anonymous function in the weakly typed languages group, as listed in Table?. The least frequently used feature is threads in both groups. The frequency data for the two groups are then used to test the following hypothesis:

- *Hypothesis 2.* There is no difference in feature usage frequency between strongly and weakly typed languages.

Similarly, a U-test at 95% confidence was used to test the hypothesis. The results show that there is a statistically significant difference in the feature usage frequency between strongly typed languages and the weakly typed ones at $p<0.05$ in five features: inheritance, exceptions, anonymous function, generics, and interfaces, as seen in Table 5.9.

Features	p	Z	N	r	Effect Size
		Static vs Dynamic Languages			
Inheritance	0.000	-11.137	2813	-0.210	S
Exceptions	0.000	-6.502	4784	-0.094	T
Threads	0.000	-22.358	2980	-0.410	M
Anonymous func.	0.000	-30.362	4572	-0.449	M
Generics	n/a	n/a	1620	n/a	n/a
Interface	0.000	-6.721	2285	-0.141	S
		Strong vs Weak Languages			
Inheritance	0.0000	-9.75	2831	-0.183	S
Exceptions	0.004	-2.849	4802	-0.041	T
Threads	0.547	-0.602	2998	-0.011	T
Anonymous func.	0	-42.927	4590	-0.634	L
Generics	0	-7.751	1928	-0.177	S
Interface	0.000	-3.932	2303	-0.082	T
		Memory managed vs un-managed Languages			
Inheritance	0	-5.592	3121	-0.100	S
Exceptions	0	-21.8	5092	-0.306	M
Threads	0.131	-1.512	3288	-0.026	T
Anonymous func.	0	-4.649	4880	-0.067	T
Generics	0	-19.561	1928	-0.445	M
Interface	0.003	-3.008	2593	-0.059	T

Table 5.9: U-test statistics per language group (feature frequency mining).

Hence, the null hypothesis is rejected and a statistically significant association between feature usage and language group for those five features exists. However, the results show there is no difference between the two in threads usage frequency. As for the effect size, a large-sized effect is found between the two groups in anonymous functions frequency, a small-sized effect in inheritance and generics, and a tiny-sized effect between the two groups in exceptions and interfaces.

When accounting for project sizes, result of significance hold in inheritance and anonymous functions for all of the different sizes. They also hold in exceptions for all but small-sized projects, and in generics for large projects only. As for the effect size, it holds in inheritance except for very large projects, and in anonymous functions except for large and very large projects.

When accounting for project types, results hold in anonymous functions for all types, and also, in inheritance except for type5, and in exceptions except for type4. As for the effect size, results hold in inheritance. In exceptions they are small and tiny. In anonymous, they hold for all types but type3,(medium).

As in the other two categorisation, the features with the highest average are generics and anonymous functions. In the managed memory languages group it is generics, whereas, the least used is threads. In the unmanaged memory group, the most frequently used

feature is anonymous functions and the least used are exceptions and threads. After that, the usage frequency data are compared and used to test the following hypothesis.

- *Hypothesis 3.* There is no difference in usage frequency between the managed and unmanaged memory languages.

Again, the Mann-Whitney U-test is used to test the hypothesis, as the data are not normally distributed. The results show a statically significant difference in five features: inheritance, exceptions, anonymous function, generics, and interfaces, at $p<0.05$. Thus, the null hypothesis is rejected for those five. By contrast, no difference was found in threads usages between the groups for the chosen alpha, as shown in Table 5.9. Hence, the null hypothesis is retained. Based on the results, there is a statistically significant association between usage frequency and memory management in languages design in five features: inheritance, exceptions, anonymous function, generics, and interfaces. The effect size is found to be medium in the data for exceptions and generics usage. In the case of inheritance, it is a small-sized effect, and tiny-sized for anonymous functions and interfaces.

When accounting for project sizes, significant results hold for exceptions and generics for all sizes. It also holds in inheritance for all except tiny projects. The effect size in exceptions is small for some project sizes and medium for others. As for generics, the effect size is rather arbitrary for the different sizes.

When accounting for different project types, significant results hold in exceptions and generics for all types,and in inheritance except for type5, and in anonymous functions except for type4. As for the effect size results, they hold in inheritance, and in generics except for type5 (small). In anonymous function, it holds for type4, otherwise the effect is small.

5.4 Discussion

Before answering the research questions and discussing the findings, we need to point out the impact of using the three text mining methods on the findings. Depending on the mining method, results can vary. The feature binary occurrence method shows that the feature has been used in the language projects but does not show the intensity nor the frequency of using it. The feature occurrence method shows the number of occurrence(s) a feature made per project, whereas, in the feature frequency method, the number of occurrences is normalised by the size of the project in SLOC. Using such different methods is to approach the chapter questions from multiple perspectives, in addition to providing a clear and objective view of the state of the practice and its implications. The discussion is divided into 3 subsections, answering the three chapter questions, described below.

Binary Term Occurrence	Term Occurrence	Term Frequency
Inheritance (55.6%)	Generics (50%)	Generics (50%)
Anon (30%)	Anon (40%)	Anon (40%)
Exceptions (9.1%)	Inheritance (22.2%)	Inheritance (33.3%)
Generics (16.7%)	Exceptions (18.2%)	Exceptions (9.1%)
Threads 0	Threads 0	Threads 0
Interfaces 0	Interfaces 0	Interfaces 0

Table 5.10: The most frequently used feature per mining method per language.

RQ1. How frequently are such features used by developers in practice?
It is found here that the most frequently used feature applying the binary occurrence mining method is inheritance. Inheritance has been used in 88% of the projects, followed by exceptions which has been used in 80% of the projects, and then by interfaces in 78% of the projects. Generics (76%) and anonymous functions (70%) are next leaving threads to be the least frequently used feature with 45% usage in the included projects. These calculations are solely based on projects in which such features are language supported. Hence, the findings confirm that the selected features for this study, except for threads, are used in the majority of projects (>70%), hence, important to the user community and representative to get insights about the state of the practice of language features.

In addition, the feature with the highest occurrence average is generics (median 44), followed by anonymous functions (28), inheritance (25), exceptions (16), interfaces (5), and finally, threads with a median of 0. This ordering of features is similar to that depicted in the findings of feature occurrence and frequency mining methods, in which the most frequently used features per language are summarised. Table 5.10 summarises the findings per mining method. Noticeably, threads and interfaces did not make it to the most frequently used feature in any of the mining methods. Also, considerable similarities are found between term occurrence and frequency methods.

In Meyerovich & Rabkin (2013), inheritance comes first as the most important feature. This perception has also been found in our dataset as the most frequently used feature using binary mining, however, the order of the other features on the list did not hold. For instance, higher-order functions and generics came in last on their list, whereas, in ours, they both come first in usage occurrences and frequency. Additionally, higher-order functions (anonymous functions) come in second as the most used in binary term occurrence, after inheritance.

The variations in the features order can be due to the mining method, as well as to the nature of features. For instance, inheritance is typically an object-oriented feature and is usually used once, at a class-level, whereas generics is a functional feature and used inline when needed, which can occur many times in a single class. This can explain the reasons

why inheritance came first as the most frequently used feature when findings are based on binary term occurrence, whereas generics came first in occurrence and frequency.

The findings also highlight that threads usage is considerably low compared to other features. It has been used in 45% of the included projects (the lowest usage percentage) and never ranked as the most frequently used feature in any of the included languages. Interfaces also was never named the most frequently used feature in any languages although it has been used in 78% of the projects.

In summary, the studied features, except for threads, are used in the majority of projects, however, the intensity and frequency of their use are varied across projects. Thus, they are essential to have in a language and worthy of study. The feature usage findings can be affected by the mining method and the feature nature, causing variations in the most frequently used feature lists. Nevertheless, the chosen methodology helps understand user community by analysing usage and by encouraging evidence-based language design.

RQ2. What are the most frequently used features per language?
As per language, the results of the three mining methods have many similarities and some differences. In four out of the eleven languages (36.4%), there is a difference between the results of the mining methods: Java, Objective-C, PHP, and C#. Regardless of the mining method, the most used features per language are as follows:

- Java: inheritance and generics.
- JavaScript: anonymous functions.
- Python: exceptions.
- Go: anonymous functions.
- Objective-C: inheritance and anonymous functions.
- PHP: inheritance and exceptions.
- Swift: inheritance.
- Ruby: inheritance.
- C#: exceptions and generics.
- C++: generics.
- TypeScript: anonymous function.

In Java, 99.2% of projects use inheritance, however, the average occurrence is 37, and the average frequency is 0.011, which is considerably lower than generics average occurrence (60) and average frequency (0.018). This is reasonable, because inheritance is a per class

feature, whilst generics can be used many times per class. Thus, such a difference in the nature of the feature affects the findings of the three mining methods. Similarly, in Objective-C, inheritance is used in 99.7% of the projects, with an average occurrence of 19 and frequency of 0.008, whereas anonymous functions are used in 79% of projects, but with a higher average occurrence (33.5) and frequency (0.014). In PHP, inheritance is used in 88.6% of the projects, and this is the only case in which occurrence and frequency results differ. That is, although exception is of a higher occurrence average, the frequency of using inheritance is higher. In C#, both exceptions and generics are used in 49.7% of the projects, however, generics are considerably of higher occurrence and frequency of use. The numbers for this section are listed in Table 5.2, Table 5.6, and Table 5.8

In Figure 5.6, we group languages based on their similarities in regard to the most used features.

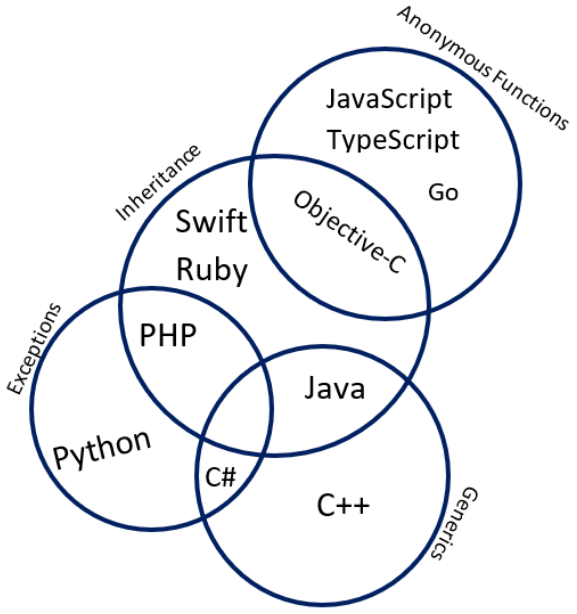

Figure 5.6: Most Used Feature per Language Group.

As per mining method, based on the binary occurrence of the features, inheritance is the most frequently used feature in Java, Objective-C, PHP, Swift, Ruby, and C++. Anonymous functions is the most used in JavaScript, Go, and TypeScript. Exceptions is most used in Python and also in C# along with generics (a tie). The results that are based on feature occurrence mining and feature frequency mining methods are almost identical, with a subtle difference. Anonymous functions is the most frequently used feature in JavaScript, Go, Objective-c, and TypeScript, using both methods. Generics is the most used in Java, C#, and C++. Inheritance is the most used in Swift and Ruby, and exceptions in Python. PHP has the only difference between the two methods. That is, the most frequently used feature using term occurrence is exceptions, whilst it

is inheritance in the case of feature frequency, with only a small margin of difference between the two. It is also noticeable that threads and interfaces were never named as the most frequently used feature in any language using any of the mining methods.

The overall aim of this research is to study programming language effect, and, as we are investigating eleven languages, it is important to investigate whether languages are used in a similar manner. This is another step toward understanding the user community, how they use features per language, and whether there is a tendency to use the same features at the same frequency regardless of the chosen language. For this purpose, we also list the least frequently used features per language, regardless of the mining method:

- Java: anonymous functions,
- JavaScript: exceptions
- Python: threads
- Go: threads, interfaces.
- Objective-C: threads, exceptions, generics
- PHP: threads
- Swift: exceptions, generics, interfaces
- Ruby: threads, anonymous
- C#: threads
- C++: anonymous
- TypeScript: inheritance, exceptions

To check whether there is a tendency to use same features at the same frequency regardless of the chosen language, the two lists of the most and least used features per language were compared. Although some similarities were found, there were also differences in features usage between languages.

RQ3. What are the most frequently used features per language group? Is there an association between language design and feature usage?

The results have shown similarities and differences in the most used and least used features between groups. The binary feature occurrence mining shows that inheritance is the most used feature in static, strong, managed and unmanaged memory language groups, whilst it is anonymous functions in dynamically typed group, and interfaces in the weakly typed language groups. Based on mining feature occurrence it is generics

in the static, strong, weak, and memory managed groups, anonymous function in the dynamically typed, and inheritance in the unmanaged memory languages. Using the third mining method, the feature frequency, the results are split equally into two groups. The first group is where generics is the most frequently used feature. This group includes statically typed, strongly typed, and managed memory languages. However, in the second group, it is anonymous functions, that is, in dynamically, weakly, and unmanaged memory languages. These findings are previously summarised in Section 5.3.

The relationship between feature usage and language design was inspected using statistical hypothesis testing and effect size for the magnitude of the difference, if any. The tested hypothesis was *there is no association between feature usage and language design.*

We investigated the hypothesis in three binary classifications of languages, using six intrinsic features, and based on the data of three feature mining methods. Then, we accounted for project size (5 sizes) and project type (5 types) to inspect whether the findings hold. The results revealed that there is a statistically significant difference between the groups of the three classifications in most of the features usage. Out of 54 tests, 47 significant differences were found; that is, all of the cases except the generics feature in type system I groups (statics vs dynamic) for all of the three mining methods, threads in memory management groups (managed vs unmanaged) in both of the feature occurrence and feature frequency mining methods, and also in threads in type system II groups (strong vs weak) in feature frequency method, and finally, inheritance in the memory management groups in the feature occurrence mining method. Those findings suggest a considerable association between language group and feature usage. However, when the effect size was inspected, more than half of the cases were of small and trivial effect. That is, although the majority of cases showed a significant difference, about 62% of them are of small and trivial effect, and 38% are of medium and large effect. The cases of the medium and large effect are arbitrary between the different groups and features. A summary of the effect sizes per group are shown in table.

Moreover, the results hold only in a few cases, when accounting for different project sizes and types. When accounting for different sizes, significant results hold only in 18 cases (out of 235 = 8% of tests) and effect size results hold only in 4 cases. When accounting for different types, significant results hold only in 16 cases, and effect results hold only in 6 cases.

In summary, although a significant association exists between language design and feature usage (87% of the tests), the effect size of such associations is trivial (<small) in 28% of the tested cases, small in 34%, medium in 28% and large in 11% cases. Additionally, when the hypothesis was tested in groups of a smaller scale, as when accounting for the different sizes and types, the results of both significant and effect size hold only in a limited number of cases.

In similar studies in which a large scale dataset was used (Ray et al. 2014, Berger et al. 2019*a*, Nanz & Furia 2015), significant statistical differences were found and associations were reported. However, such statistical difference can be due to the large size of the sample Lantz (2013). Thus, we went further in our investigation and accounted for the different sizes and types. We found that, in the smaller groups, such hypothesis of languages association do not hold in most of the tested cases.

5.5 Related work

A number of studies that examine the features of programming languages have inspected the usage of one or two features in a single or a couple of languages. The investigated features included language intrinsic and extrinsic features, such as performance, generics, quality, libraries, inheritance, and lambdas, among others. Parnin et al. (2011) have mined 20 Java repositories to investigate the adoption of generics by developers. A randomised controlled experiment on lambdas and iterators usage in C++ shed light on the difficulty of lambdas syntax in C++, compared to iterators Uesbeck et al. (2016). The adoption of exceptions by Java developers was analysed based on mining 90 projects and the factors that affect them, such as the project domain and type by Osman et al. (2017). The usage of inheritance in Python has been investigated using 51 projects in Orru et al. (2015), which is a replication of a prior study on Java's projects by Tempero et al. (2008). Okur & Dig (2012) investigated the usage of another language feature, the parallel libraries, based on analysing a dataset of 655 C# projects. Phipps (1999) compared defects density between Java and C++, the results of which showed that Java had two to three times less bugs per line of code than C++, about 15% to 50% less defects per line, and was about six times faster to debug. Another study about languages and quality by Bhattacharya & Neamtiu (2011*b*) investigated the differences between C and C++ in code quality, stating that C++ is better in software quality. Such attempts are valuable, however limited to understand the differences between languages and to investigate whether there is a link between language design and feature usage, as they are based on a couple of languages or features.

A smaller number of studies went further to inspect more languages and a variety of related features, most of which benefit from the availability of online hosting platforms, such as Github and SourceForge, to create datasets of large size that reflect current practices in programming and programming languages. Prechelt (2000)'s controlled experiment on seven languages, each one implementing the same set of requirements resulting in 80 programs, showed that performance variability exists between programmes in different languages in terms of memory consumption and execution speed. However, performance variability among programmers of the same language was found to be larger than variability among the different languages. This variability can be undermined if a larger dataset is used. Moreover, the different languages had different effects on program

conciseness and structure. Although the experiment was carried out on a small scale, such attempts are valuable indicators and point the way towards stronger evidence.

In 2014, a large-scale study (Ray et al. 2014) investigated 729 projects in 17 languages based on mining Github repositories for the association between language and code quality. The study found that a language's feature (such as the typing system) has significantly affected quality in terms of propensity for defects, however, the effect size is modest. The study was replicated in 2019 by Berger et al. (2019*a*). They first reproduced the findings based on analysing the same 729 projects following the same methodology and analysis, in which the replication process succeeded partially. Then, they re-analysed the included projects by following a different methodology for data processing and statistical analysis than did the original paper. As a result, a smaller number of projects was included (423 projects) and most of the claims did not hold. Nevertheless, for those cases were claims held, the relationship between programming languages and defects was found to be exceedingly small in effect size.

Another large-scale study (Nanz & Furia 2015) based on mining 7,087 programs in 8 languages, inspected conciseness, performance, and failure proneness as language features. The study found that the paradigms of a language affect conciseness differently, such that functional and scripting languages are better than procedural and object oriented. In performance, in terms of running time, C is the best language on large inputs, followed by Go as second. Procedural languages are more efficient in memory usage than are languages from other paradigms. Finally, strongly typed languages are less prone to defects than weakly typed ones.

5.6 Summary and conclusions

This is the first study of such a large volume that investigates the usage of 6 language-intrinsic features in eleven languages while accounting for project size and type in an attempt to investigate the link between language design and feature usage. We investigated how language features are used in practice and whether there is a significant association between language design and feature usage. The investigation was carried out in a large-scale setting using statistical methods by accounting for confounding factors such as project size and type, per language and language group. The studied features, except threads, are used in the majority of projects, however, the intensity and frequency of using them are varied across projects. Hence, they are essential to have in a language and in a representative set for investigating language features usage.

Additionally, to understand the user community and whether there is a tendency to use the same features regardless of the language, we checked the most and least frequently used features across languages. It was found that although there are some similarities, features usage is varied between languages. Finally, the association between language

design and feature usage was investigated, and although the majority of tested cases suggest a significant association, the effect sizes of 62% of them are of small and trivial, and 38% are of medium and large effect size. The cases of the medium and large effect are arbitrary between the different groups and features. Moreover, the results of significance and effect size hold only in a limited number of cases, when the hypothesis was tested on a smaller scale; when accounting for different project sizes and types. Thus, language choice does not have significant effect on feature usage. Nevertheless, the chosen methodology in this chapter helps understanding the user community by analysing their usage, and encourages evidence-based language design.

Chapter 6

Programming language and OSS development

In this chapter, the third research objective is fulfilled, in which project development attributes are investigated. In Section 6.1, the chapter objectives are listed. Next, in Section 6.2 the followed methodology is explained. In Section 6.3 the results are provided per mining method. Then, in Section 6.4 the study questions are answered and discussed. The related work is covered in Section 6.5 and the study is concluded in Section 6.6.

6.1 Objectives

This chapter investigates the impact of general-purpose programming languages on open-source software development. It examines whether there is a strong evidence that indicates a relationship between language choice and the development process of OSS projects, in a large-scale setting. A comparison is made between projects written in the included languages based on mining project's repository data. Comparisons were made per language, (individually) and per language group to examine differences, if any. Project attributes such as size, duration, and contribution rate were used for this comparison to inspect and understand the relationship between language design and OSS development. The chapter objectives are summarised as follows:

1. **Examine to which extent projects in the different languages differ or similarise.** This study compares projects written in the different languages to examine whether significant differences exist between them in developing open-source software. Project attributes are examined, per language, to determine whether they are the same in their attributes regardless on the chosen language.

2. **Investigate the relationship between language design and OSS project attributes statistically while accounting for the different sizes and types**

as confounding factors. This study aims to investigate whether a significant statistical relationship exists between language design and project attributes. Such an association may suggest a strong effect of languages on software projects. Moreover, this study also to investigate the statistical relationships between language design and development attributes while accounting for project size and type as confounding factors.

3. **Examine the findings of language association to software development in a large-scale setting.** Early studies on languages and software development have revealed significant differences between low-level and high-level languages, while no comparisons were carried within the group of high-level languages due to limited sample size. Research from the 1990s onwards on the high-level group are preliminary, their results are sometime contradictory, and some were based on small-scale experiments. Thus, this study is a step towards understanding the relationship between language choice and it's possible association to the development process.

To address these objectives, we identified the following research questions:

1. How do projects written in the different languages vary and similarise in their development attributes?
2. Is there any association between language design and development attributes? After accounting for different sizes and types as confounding factors will findings hold?
3. What is the effect of programming languages on software development?

Lastly, This chapter offers the following contributions:

- It investigates a large volume of data to find sufficient empirical evidence of language association with software development.
- It accounts for confounding factors such as project size and type when inspecting language's effect.
- It informs state-of-the-practice of OSS project development attributes using rigid statistical methods to support findings.

6.2 Methodology

Here we utilise the availability of open-source data to understand the development process of OSS and inspect the impact of the language factor on it. The methodology is primarily

based on the mining software repositories (MSR) of Github statistical methods were used to inspect the outcomes. As with classic data mining process, we go through the following steps: (a) data gathering; (b) data cleaning and preparing; and (c) information extraction (Liao et al. 2012). The data gathering has been discussed in chapter 3. Here we list the steps briefly:

- Get the list of the popular repositories on Github.
- Identify the project primary language.
- Pull the repository data.
- Save the data as JSON files.

In the second step; the selected data for this chapter are: artifact size in KB, artifact source code files, repository creation date/time, last pushed commit date/time, last release date/time, repository contributions, and contributors data. Those data were utilised to get the following development process attributes: project size in SLOC, project duration calculated in two ways; from the creation date of the repository to the last release and from the creation date to the last pushed commit if no releases were found, total number of contributions to the repository, total number of contributors, and rate of contribution. Those attributes are used to investigate the effect of the chosen languages on the development process and extract information on the relationship strength and magnitude, if any.

The motivation behind choosing this methodology is summarised in the following pints:

- The intended dataset is open source due to its availability. Thus, we compared a number of the most used online repositories for open source projects in terms of size (users and projects), establishment date, and available features. The repositories managed to offer similar features. However, Github was the one with the largest number of users and projects compared to other popular repositories such as SourceForge and Launchpad.
- Github hosts some prominent open source projects, such as Linux kernel, Ruby on rails, and JQuery. In addition to the availability of source codes, it also offers a wealth of data related to the software artifact and the development process, making it a valuable source for researchers. Thus, it provides an opportunity to build large-scale datasets of selected, high quality, real project data for research purposes.
- Mining software repositories (MSR) to uncover patterns and discover findings about the artifact and the delivery process has renown as a research area over the last two

decades. In 2004, a specialised conference[1] on mining software repositories evolved from the premium international conference on software engineering (ICSE)[2], in recognition of the importance and potential of this field.

Nevertheless, mining Github repositories and retrieving large amounts of data from Github is a challenging task. Github allows their data to be accessed over HTTPS as JSON. However, it does not provide a schema for its data. Thus, for it to be examined it is necessary to traverse back its data using REST requests and JSON responses. Moreover, Git hub imposes a rate limit on its API: 5000 requests per hour. Given the huge number of events generated per day, and that every single event would lead to a series of dependent requests, pulling large amounts of data from Github would result to significant delay.

6.2.1 The dataset

The total number of projects in the dataset is 5350. The mean size of projects is 55161 SLOC (very large), whereas, the median is 4949 SLOC (small). Commits mean is 1174, with a mean of 67 contributors and 26 commits per person. The durations of projects vary between a minimum of less than a month up to 154 months maximum with a mean of 51.Table 6.1 shows the descriptive statistics of the dataset used for this chapter.

Attribute	N	Mean	Std. Deviation	Min.	Max.	Median
Project size (SLOC)	5350	55161	387408	8	11885756	4949
Commits	5350	1174	4484	2	204379	296
Contributors	5350	67	435	1	21111	21
Commits per contributor	5350	26	51	0	1214	14
Duration (Months)	5350	51	34	0	154	47
Project Size (KB)	5350	23823	104384	7	2778030	2808

Table 6.1: Descriptive statistics of dataset projects attributes.

6.2.2 The statical methods

As the included attributes for the study are quantitative data, they have been inspected for normality to decide which statical method to use for the analysis phase. Thus, we used hypothesis testing and graphical representation such as the Q-Q normal probability plot and histogram (Field et al. 2012). We first proposed to investigate the following normality hypothesis for the included five attributes using Shapiro–Wilk normality test:

- *Hypothesis 1.* The sample data of the statically and dynamically typed language are not significantly different from a normal population.

[1] "Mining Software Repositories." http://msrconf.org/
[2] "International Conference on Software Engineering (ICSE)" http://icse-conferences.org/

For the five attributes; project size, commits count, contributors count, contribution rate, and duration, the p-value is less than our chosen significance level alpha (0.05). Thus, the null hypothesis is rejected and we state that data do not follow a normal distribution. Test statistics are listed inTable 6.2. In addition to the Shapiro–Wilk test, we also went through the Q-Q plot and histogram to inspect data normality, and same results hold.

Project attribute	Lang. Group	Statistic	df	Sig.
SLOC	S	0.155	1482	0.00
	D	0.134	1958	0.00
Commits	S	0.284	1482	0.00
	D	0.152	1958	0.00
Contributors	S	0.052	1482	0.00
	D	0.301	1958	0.00
Commits per contributor	S	0.446	1482	0.00
	D	0.408	1958	0.00
Duration (Months)	S	0.964	1482	0.00
	D	0.973	1958	0.00

Table 6.2: Normality test statistics of development attributes per language group (static vs dynamic).

Similarly, the data in the second language classification has been tested for normality using same methods: Shapiro–Wilk test and graphical representation. The tested hypothesis is:

- *Hypothesis 2.* The sample data of the strongly and weakly typed language are not significantly different from a normal population.

The results showed that the p-value is less than the chosen significance level alpha, which is 0.05. The null hypothesis is rejected accordingly, and it is revealed that the data do not follow a normal distribution for all of the included five attributes. Test statistics are listed inTable 6.3.

Finally, the normality was inspected for third language classification for projects attributes using Shapiro–Wilk test along with the Q-Q plot and histogram. The tested hypothesis is:

- *Hypothesis 3.* The sample data of the memory managed and unmanaged languages are not significantly different from a normal population.

For the five included attributes, the results shows that the p-value is less than our chosen significance level alpha (0.05) as listed in Table 6.4. Thus, the null hypothesis is rejected and we state that the data do not follow a normal distribution. In addition to the

Project attribute	Lang. Group	Statistic	df	Sig.
SLOC	S	0.153	1822	0.00
	W	0.1	1372	0.00
Commits	S	0.177	1822	0.00
	W	0.258	1372	0.00
Contributors	S	0.096	1822	0.00
	W	0.051	1372	0.00
Commits per contributor	S	0.393	1822	0.00
	W	0.531	1372	0.00
Duration (Months)	S	0.955	1822	0.00
	W	0.973	1372	0.00

Table 6.3: Normality test statistics of development attributes per language group (strong vs weak).

Shapiro–Wilk test, the Q-Q plot and histogram were used to discern data normality, and the same results hold.

Project attribute	Lang. Group	Statistic	df	Sig.
Commits	Mgd.	0.172	2710	0.00
	Unmgd.	0.295	730	0.00
Contributors	Mgd.	0.113	2710	0.00
	Unmgd.	0.045	730	0.00
SLOC	Mgd.	0.183	2710	0.00
	Unmgd.	0.116	730	0.00
Duration (Months)	Mgd.	0.975	2710	0.00
	Unmgd.	0.925	730	0.00
Commits per contributor	Mgd.	0.418	2710	0.00
	Unmgd.	0.467	730	0.00

Table 6.4: Normality test statistics of development attributes per language group (managed vs unmanaged memory).

As the normality tests showed that the data of the included project attributes is not normally distributed for the three language classifications, the arithmetic median were chosen to describe the average, and the Mann-Whitney U-test was used for hypothesis testing (Field et al. 2012) along with effect size. In addition, we accounted for different sizes and types of projects included in the dataset to observe if outcomes hold for the different variations. Thus, for this chapter we propose to investigate the following main hypothesis:

- *Hypothesis 1.* There is no difference between languages in software development attributes.

6.3 Results

Per language As the project attributes data is not normally distributed, the median has been selected to describe the average. The average size of projects in the dataset is 4949 source-line-of-code (SLOC). The project sizes vary between a lowest average of 2527 SLOC for Swift and the highest average of 49065 SLOC for C#. The three languages with the highest SLOC average are C#, C++, and C respectively. For the contributions count, the dataset has an average of 296 commits/contributions. The language with the lowest average is Java with 131 commits, and the language with the highest average is Typescript followed by C#, and Ruby, with a median of 1354, 1300, and 773 commits, respectively. For the third attribute, the number of contributors, the average is 21 person per project for the whole dataset. Similar to commits, Java is the language with the lowest average contributors,(7 contributor per project).Meanwhile, Ruby is the language with the highest average of contributors to its projects (76 contributors), followed by TypeScript, and C#, with a median of 71 and 50 contributors. When it comes to the contribution rate, C++, C#, and Java have the highest average rates with 28, 25, an 18 contributions per person, respectively. Whilst, Ruby's average is the lowest with 10 contributions per person. The average contribution rate is 13.53 commits/contributions per contributor for the whole dataset. Finally, the language with the lowest average project duration is Java with 30 months, whereas Ruby's projects have the highest average duration with 100 months, followed by PHP, and C with medians of 74 and 65 months, respectively. The average duration in the dataset is 47 months. The averages are shown in Figure 6.1.

Per language group When languages are categorised into groups based on their design, the attributes data are compared and checked for statistically significant differences using the Mann-Whitney U-test.

For the first language classification, type system I, the sample size of the projects in the statically typed language group is 2237 projects. The project size average is 5872 SLOC with an average of about 15 contributors and 252 commits. Accordingly, the contribution rate is about 17 contributions per person. The duration of projects in this group has an average of 45 months with a maximum of 146 months. The average statistics of statically typed languages projects are shown in Figure 6.2.

In the dynamically typed languages group, the project size average is 4653 SLOC, with an average of 29 contributors and 364 commits per project. The contribution rate is about 12 contributions per person. Projects duration in this group has an average of 55 months and up to 154 months maximum. Those numbers are based on a sample size of 2764 projects. Figure 6.2 shows the average statistics of the related variables in this language group projects.

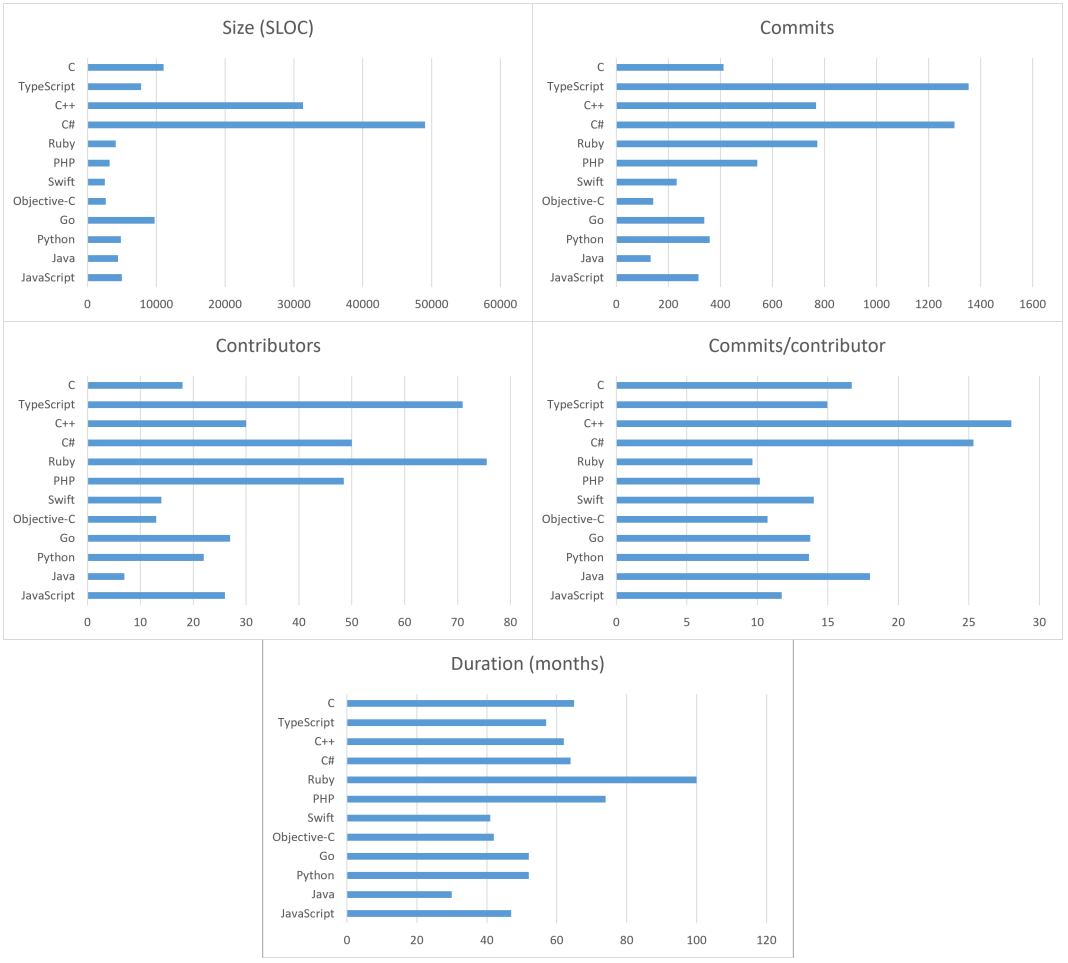

Figure 6.1: Project attributes per language (median).

To further investigate the differences, the attributes data has been used to statistically test the following hypothesis:

- *Hypothesis 1.* There is no difference between statically and dynamically typed language in software development attributes.

A non-parametric U-test at 95% confidence was used. The results show that there is a statistically significant difference in the projects development attributes between the statically typed and the dynamically typed languages at $p<.05$ in all of attributes; project size (SLOC), commits, contributors, contribution rate, and project duration. Thus, the null hypothesis is rejected for these attributes. The U-test statistics are listed in Table 6.5. As listed, the p-value is less than our chosen significance level (alpha = 0.05). Hence, there is a relationship between project's development and language design: a statistically significant association between OSS development and language design for the included attributes. The dynamically typed languages group is have larger commits, contributors, and project duration. On the other hand, the statistically typed projects have a higher project size and contribution rate. In addition to the U-test, we

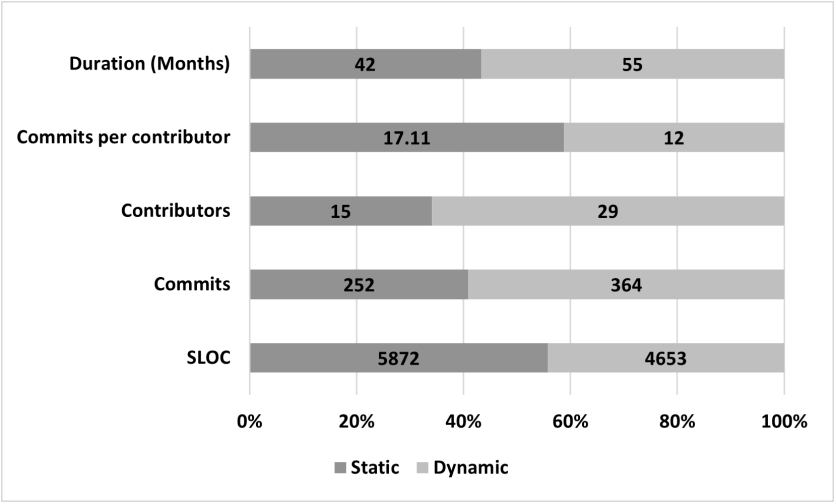

Figure 6.2: Project attributes per language group (static vs dynamic).

also investigated the effect size to demonstrate the magnitude of the reported difference between the two groups. The results show a small effect size in project size, number of contributors, contribution rate, and projects duration. Meanwhile, the effect size was found to be less than small (trivial) for the commits between the two groups.

	p	z	n	r	Effect size
SLOC	0.00	-7.908	5001	-0.1118248	S
Commits	0.00	-7.062	5001	-0.0998618	T
Contributors	0.00	-15.095	5001	-0.2134542	S
Commits per contributor	0.00	-13.192	5001	-0.1865444	S
Duration	0.00	-13.061	5001	-0.184692	S

Table 6.5: Hypothesis test and effect size statistics per language group (static vs dynamic).

When project size is accounted for, results of statistical significance hold for all the attributes in the tiny and large projects. As per attribute, results hold for contributors and commits rate for all sizes. For effect size results, they hold only for commits for the different project sizes (i.e, they have trivial). Otherwise, the effect size is rather arbitrary between the different groups for the different attributes. However, no large effect has been shown for any attribute in any of the project sizes.

When the different project types are accounted for, results of statistical significance hold only in comments rate for the different project types. Effect size results are arbitrary between the different types for the different attributes. Nevertheless, as with when project size is accounted for, no large effect size has been found in any of the included attributes for any project type. Results per project type and size are summarised in the Appendix C.

In the second categorisation, type system II, the sample size of the projects in the strongly typed language group is 3178 projects. The project size average is 4431 SLOC with an average of about 16 contributors and 242 commits. Accordingly, the contribution rate is about 14 contributions per person. The projects duration here has an average of 43, and 154 months maximum. The average of statically typed languages projects attributes are listed in Figure 6.3.

On the other hand, for the weakly typed languages group, the sample size of projects in the dataset group is 1894 projects. The average size is 6743 SLOC per project, with an average of 27 contributors and 361 commits per project. The contribution rate here is 13 commits per person. The duration of projects has an average of 50 months, and up to 146 months maximum. Figure 6.3 shows averages of the development attributes in this group projects.

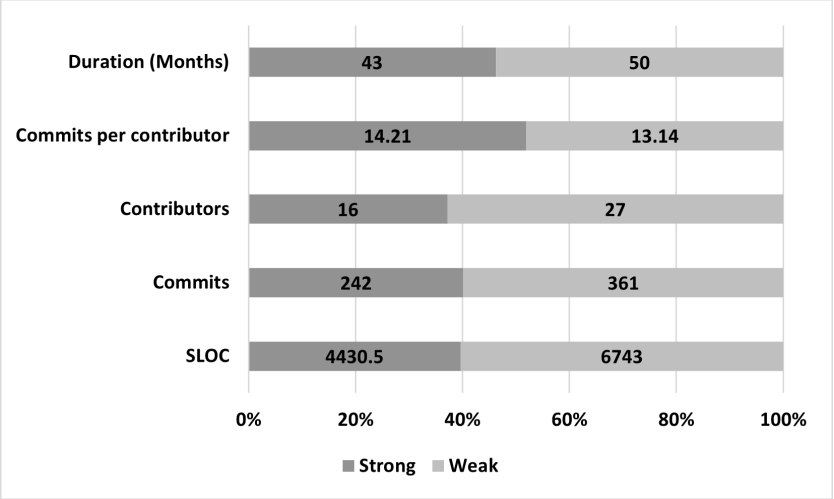

Figure 6.3: Project attributes per language group (strong vs weak).

The attributes data for the two groups are then used to test the following hypothesis:

- *Hypothesis 2.* There is no difference between strongly and weakly typed language in software development attributes.

Similarly, a U-test at 95% confidence was used to test the hypothesis. The results show that there is a statistical significant difference in the development attributes between strongly typed languages and the weakly typed languages at $p<0.05$ for all of the five attributes project size in SLOC, commits count, contributors, contribution rate, and project duration as seen in Table 6.6. Hence, the null hypothesis is rejected and a statistically significant association between project development and language design for those five attributes exists.

We also investigated the effect size to get insights into the strength of the reported association between the development attributes and language group/design. The effect

size was found to be small and trivial between the two groups for the included attributes. It is tiny in projects size, contribution rate, and project duration and small in commits and contributors between the two groups.

	p	z	n	r	Effect size
SLOC	0	-5.234	5072	-0.0734927	T
Commits	0	-9.068	5072	-0.1273274	S
Contributors	0	-12.099	5072	-0.1698869	S
Commits per contributor	0	-3.81	5072	-0.0534977	T
Duration	0	-7.525	5072	-0.1056615	T

Table 6.6: Hypothesis test and effect size statistics per language group (strong vs weak).

When project sizes are accounted for, result of significance hold only in the tiny size projects for all of the included attributes. As per attribute, no significance result hold per attribute for all of the project sizes. Effect size results did not hold between the different groups. However, they are either trivial or small. No medium or large effect size was found for any attribute in any of the project sizes.

When project types are accounted for, result of significance did not hold for project size in any of the attributes. As for the effect size, results did not hold for the different types, while there are either trivial or small and no medium or large effect for any attribute in any of the project types. Results per project type and size are summarised in the Appendix C.

In the third categorisation (memory-managed) the sample size of the projects in the memory managed language group is 3738 projects. The project size average is 5170 SLOC with an average of about 28 contributors and 374 commits. Accordingly, the contribution rate is about 13 contributions per person. The projects duration average in this group is 53 months, and the maximum duration for projects in this group is 154 months. The statistics of memory managed languages projects are listed in Figure 6.4.

Meanwhile, in the unmanaged memory languages group, the sample size of projects in the dataset group is 1612 projects. The average size is 4627 SLOC per project, with an average of 10 contributors and 151 commits per project. The contribution rate here is about 16 contributions per person. The duration of projects varied between a minimum of 1 month up to 146 months. Figure 6.4 shows the averages of the related attributes in these group projects.

After this, the development attributes data are compared and used to test the following hypothesis:

- *Hypothesis 3.* There is no difference in software development between the managed and unmanaged memory languages.

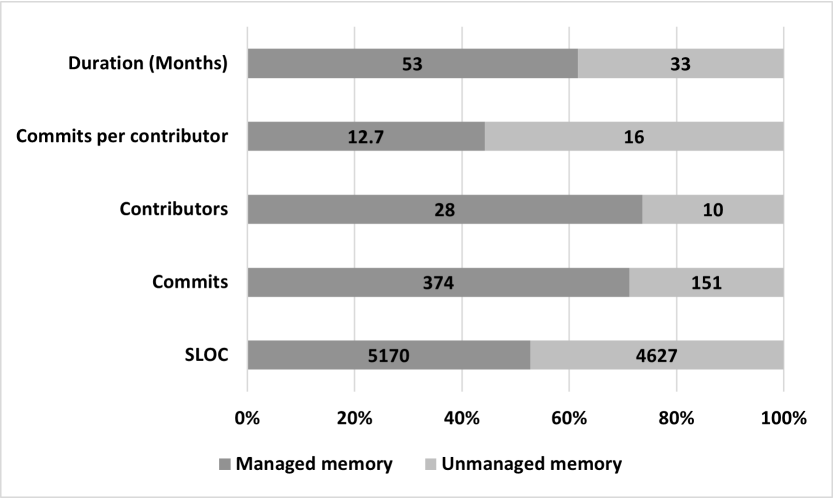

Figure 6.4: Hypothesis test and effect size statistics per language group (managed vs unmanaged memory).

Again, the Mann-Whitney U-test was used to test the hypothesis as the data is not normally distributed. The results show a statically significant at pt<0.05 in four of the five attributes: out of the included five: number of commits, contributions, contribution rate, and project duration at p<0.05. Thus, the null hypothesis is rejected for those four. On the other hand, no difference was found in project size between the two groups for the chosen alpha, as shown in Table 6.7. Hence, the null hypothesis is retained for this attribute. Based on the results, there is a statistically significant association between software development and memory management in languages' design in four attributes: commits count, contributions, contribution rate, and project duration. The effect size is found to be medium in the data for contributors only. In the case of commits, commits rate, and project duration, the effect size was small.

	p	z	n	r	Effect size
SLOC	0.681	-0.412	5350	-0.0056327	none
Commits	0.00	-16.649	5350	-0.2276204	S
Contributors	0.00	-22.567	5350	-0.3085297	M
Commits per contributor	0.00	-8.556	5350	-0.1169752	S
Duration	0.00	-11.154	5350	-0.1524943	S

Table 6.7: Hypothesis test and effect size statistics per language group (managed vs unmanaged memory).

When project size is accounted for, significant results hold for commits count and duration for all sizes. It also holds in small size projects for all of the included attributes. However, the effect size results are rather arbitrary for the different sizes between trivial, small and medium, and it is never found to be large for any of the attributes in any project size.

When the different project types are accounted for, results of significance did not hold for any of the included attributes. Results per project type and size are summarised in the Appendix C.

6.4 Discussion

RQ1. How do projects written in the different languages vary and similarise in their development attributes?

When projects in different languages are compared based on their attributes, we get the following results per attribute:

- **Project size (SLOC)** Swift, Objective-C, and PHP projects have the smallest average sizes. They are followed by Ruby, Java, and Pythons projects, which also have an average of small size (between 1000 and 5000 source line-of-code). The average size for JavaScript, TypeScript, Go, and C projects is medium, which means their projects have between 5000 and 20000 SLOC. The projects of C++ and C# are the ones with the largest codebases.
- **Commits count** Java, Objective-C, and Swift have fewer commits compared to other languages. On the other hand, C# and TypeScript have the highest number of commits with an average of 1300, and 1354 commits respectively.
- **Number of contributors** The average project size based on number of contributors among the included languages is either medium or large. (i.e, no language has an average number of contributors is less than 5) Java comes first with the smallest number of contributors, followed by Objective-C, Swift, and C. Each of those languages is of a medium sized team (between 5 and 20). C#, TypeScript, and Ruby are the languages with the largest number of contributors, averaging 50, 71, and 76 contributors respectively.
- **Contribution rate** The contribution rates of Ruby and PHP projects are similar. These are followed by Objective-C and JavaScript, which have the lowest contribution rate (commits per contributor) among the languages. On the other hand, Java, C# and C++ have the highest rates.
- **Duration** Java projects have the shortest time period, with an average of 30 months, followed by Swift (41 months), and Objective-C (42). The languages with the longest project duration are PHP with an average of 74 months and Ruby with an average 100 months.

Based on the similarities and differences in the aforementioned attributes, languages are categorised into five groups as shown in Figure 6.5. The first group (A) has Objective-C, Python, Java, and Swift. Projects written in these languages are associated with small attributes: size, commits, contributors, contribution rate, and short project duration. Group B projects are associated with medium-sized attributes and are written in TypeScript, C, and C++. The third group of projects (C) posses large-sized attributes and are written in C#. Group D projects are written in PHP and Ruby. Although they are small in size (SLOC), they have medium to large attributes, except for contribution rate. Finally, Go and JavaScript projects make up the fifth group (E). Projects in this group are of medium size (SLOC), and have relatively small attributes.

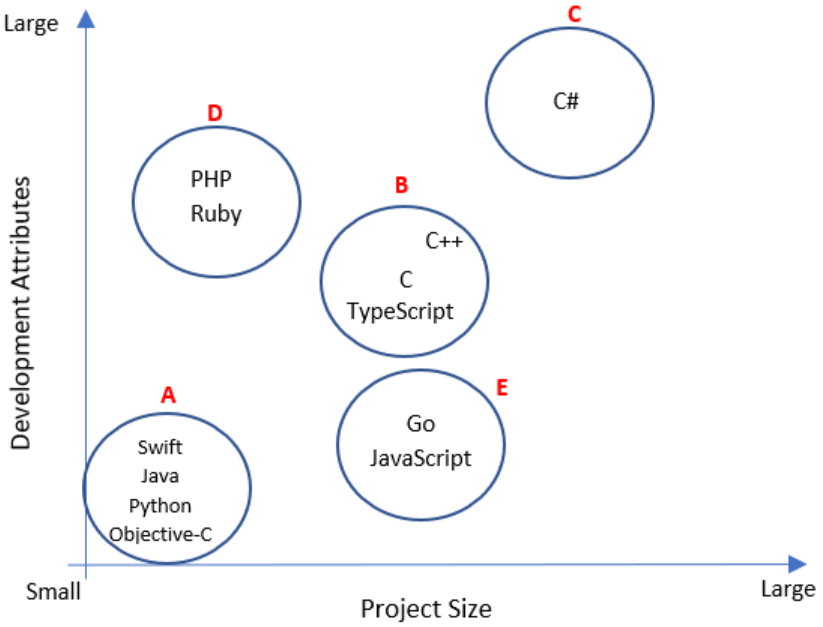

Figure 6.5: A classification of languages based on their similarities and differences in project attributes.

Here we looked into languages to address to which extent they are used in a similar/different way. The aforementioned classification categorised them based on the their usage in open-source projects. The set of the included languages are of a general-purpose, and results revealed similarities and differences between them in development attributes. However, those can be related to language characteristics, project characteristics, or personnel experience.

RQ2. Is there any association between language design and development attributes? After accounting for different sizes and types as confounding factors will findings hold?

The relationship between project attributes and language group was inspected using statistical hypothesis testing. The hypothesis was *there is no association between open-source projects attributes and language design.* It was investigated in three binary classifications

of languages using five attributes based, on the data of mining software repositories. Then, we controlled for project size (5 sizes) and project type (5 types) to see whether findings hold. The results revealed that there is a statistically significant difference between the groups in almost all of the included projects attributes. Out of 15 tests, 14 significant differences were found (i.e, all of the cases except project size in the memory management classification).

Projects that have been written in statically typed languages are of a significantly larger project size and higher contributions rate than those written in the dynamically typed ones. On the other hand, the dynamically typed projects have higher numbers of commits, have more contributors to their projects, and are of a longer duration.

In the second classification, projects written in strongly typed languages have a significantly higher contribution rate compared to the weakly typed ones. Meanwhile, the projects in the weakly typed group have larger sizes, have more commits, more contributors, and longer duration.

In the third classification, projects from the memory managed languages group have higher number of commits, contributors, and longer duration than the unmanaged memory group. On the other hand, the unmanaged languages have a higher contribution rate. No significant difference was found between the two groups in project size. Such findings suggest a strong association between language design and OSS development attributes. Nevertheless, when the effect size was inspected, the vast majority of cases were small and had trivial effect. In other words, although the majority of cases showed significant difference, the magnitude of this difference is small and tiny in about 93% of tested cases.

Moreover, the statistical significance results hold in fewer cases when different project size and types are accounted for. When accounted for project sizes, significance results hold in 8.6% of tests and effect size results remain trivial and small in the majority of them. When different types are accounted for, significance and effect size results did not hold.

In summary, although a significant statistical association exists between language design and projects attributes (93% of the tests), the size of such associations is trivial (<small) in 28.6% of the tested cases, small in 64.3%, and medium only in 7.14% cases. Meanwhile, no large size effect was reported in tested cases. Additionally, when the hypothesis was tested in small scale groups, as when accounted for the differences in project size and types, results for both significance and effect size hold only in a limited number of cases.

RQ3. What is the effect of programming languages on software development?

The role of programming languages in software development has been recognised by a number of cost models, such as SDC, Putnam SLIM, and SOFCOST. Meanwhile, they were excluded from TRW, COCOMO, and Jensen. One of the reasons for this exclusion can be the ‘non-quantitative" nature of programming languages. Research

that investigated languages effect, approached it using comparative settings in corpus studies and controlled experiments. Their outcomes revealed a divide on whether there is an effect and whether the magnitude of such one is critical.

Studies that inspected and compared languages individually and reported differences between them in aspects such as software quality, productivity, defects and memory consumption, did not inspect the strength and the magnitude of such difference.

Benefiting from the availability of open-source software projects data along with exercising data mining methods, we approached this potential effect, also in a comparative setting, in an attempt to expose more about it. This investigation looked for significant associations in the dataset between language design and open-source software development. The development was identified in terms of project's attributes such as projects size(SLOC), commits, contributors, contribution rate, and project duration. The comparison of individual languages have shown similarities and differences in included attributes. However, it is difficult to link them to language alone.

Thus, this study went further to investigate the association in groups rather than individually. The groups were based on the languages underlying design. The findings showed that 4 out of 5 attributes have shown significant statical association between language design and the investigated projects attributes. However, the magnitude of the reported association is rather modest in the vast majority of the tested cases.

In similar studies in which a large scale dataset was used (Ray et al. 2014, Berger et al. 2019*a*, Nanz & Furia 2015), significant statistical differences were found and associations were reported. However, such statistical difference can be due to the large size of the tested sample Lantz (2013). Thus, we went further in our study and accounted for the variability in sizes and types. We found that, in the smaller groups, such hypothesis of languages association do not hold in most of the tested cases.

6.5 Related work

Empirical comparisons have been conducted with a range of programming languages to investigate whether there exist significant differences among them. More importantly, whether such differences have an effect on software development and its related aspects. There is a disagreement among the studies in the body of literature as to whether programming languages are similar in their effect on software development, performance, and practical programming.

Some researchers have stated that programming languages do not have a considerable effect on software development, performance or practical programming, and that there is no hard evidence to support such claims. They claimed that the effect of languages on software development is rather limited and subtle in terms of a program's readability and

error propensity (Wulf 1980, Boehm 1981, Port & McArthur 1999, Myrtveit & Stensrud 2008).

Conversely, a number of studies have found some associations between languages and software development. A study compared defect density and programmer productivity in Java and C++ and showed that Java had two to three times fewer bugs per line of code than C++, about 15% to 50% fewer defects per line, and was about six times faster to debug. However, when defect density was measured (defects against development time) it showed no differences between the two languages (Phipps 1999). Another study (Bhattacharya & Neamtiu 2011*a*) assessed the impact of language on software development and maintenance in an investigation of the differences between C and C++. The researchers concluded that C++ is better software quality and effort. Their findings contradict the findings of (Myrtveit & Stensrud 2008), who concluded that there was no empirical evidence of differences between C and C++ in terms of development effort and that there is no superiority of C++ over C. An experiment examining on Java and C++ (Phipps 1999) showed that Java was about 30% to 200% more productive than C++. While such attempts are valuable, the ability to draw conclusive findings are limited because these studies are based on comparisons of only a couple of languages.

A relatively large empirical comparison (Prechelt 2000) of seven programming languages implemented the same set of requirements showed that languages have different effect on software performance in terms of memory consumption, execution speed, and on the design and writing of programs. However, no differences were found between languages in terms of program reliability. Another study of nine languages aimed at determining if there is evidence of an effect on software development concluded that the choice of language was a significant factor in writing programs and that developers' productivity rates are not constant among different languages (Delorey et al. 2007).

An additional study exploring the effect of 11 programming languages on development effort confirmed the findings of Jones 1996 and Delorey et al. 2007 that every language has its own rate. A large-scale2014 study by Ray et al has investigated 729 projects in 17 languages for the effect of programming languages on code quality found that languages can significantly effect quality; however, the effect size was modest (Ray et al. 2014). The study was replicated in 2019 by (Orru et al. 2015) who first reproduced the findings with the same 729 projects and following the same methodology and analysis, resulting in a partially successful replication. Then, they re-analysed the included projects but followed different methodologies for data processing and statical analysis than the original study. As a result, a smaller number of projects were included (423 projects) and most of the claims did not hold. In addition, for the cases, where the claims did hold the relationship between programming language and defects were found to be exceedingly small in effect size. Another large-scale study (Nanz & Furia 2015) based on mining 7,087 programs in eight languages inspected conciseness, performance, and failure proneness as language features. They found that a language's paradigms affected conciseness differently, with

functional and scripting languages performing better than procedural and object-oriented languages. In performance, in terms of running time, C was the best language for large inputs, followed by Go. Procedural languages were more efficient with memory usage than languages from other paradigms.

Besides languages, there are other factors affecting development productivity, such as the size of the developing team, their experience, and type of the application. However, our focus here is the factor of programming languages.

6.6 Summary and conclusions

In this study we investigated whether this is a strong statistical association between language design and open-source projects. The investigation was carried out in a large-scale setting using statistical methods and by accounting for confounding factors such as project size and type. The study was carried out on 12 languages and 5 project attributes: project size in source line-of-code, commits count, contributors count, contribution rate, and project duration. It was found that individual languages show similarities and differences in those attributes. However, although differences exist between projects written in the included languages, it is difficult to link them to language alone. We investigated the extent to which they differ and similarise and a classification of five groups were suggested. After this, the study investigated the possible association of at language groups level instead of individual ones. Accordingly, it was found that a significant statistical association exists between language design and the investigated project attributes. However, when the effect size was inspected, the vast majority of the cases (93%) were of small and trivial, 7% of the cases were of medium effect size, and no large size effect was reported in tested cases. Additionally, when the association was tested in small scale groups, as when accounted for the differences in project size and types, results of both significance and effect size hold only in a limited number of cases. Thus, although strong association is found in the dataset between high-level, general-purpose language/language design and OSS project attributes, the magnitude of such association is limited. In other words, the choice of language has a very limited (small) effect on open-source software development.

Chapter 7

Conclusions

7.1 Summary and conclusions

Early studies on language's programming effect on software development, performance, and practical programming, managed to point out significant distinctions between low-level and the higher-level ones (Schneider 1978, Harrison & Adrangi 1986). Nonetheless, such attempts could not provide similar findings within the group of high-level languages due to the limited sample size of software projects in such languages at that time. From the 1990's onwards, research has been continued on high-level languages and their association with software development and practices. Some have based their findings on a couple languages (Phipps 1999, Myrtveit & Stensrud 2008, Bhattacharya & Neamtiu 2011*b*), some went to a larger size (from five to eleven languages) (Port & McArthur 1999, Delorey et al. 2007, Lavazza et al. 2016, Nanz & Furia 2015, Ray et al. 2014, Berger et al. 2019*b*), and another part went further to cover about 500 languages (Jones 1996). Nevertheless, such attempts have shown preliminary, sometimes contradictory, results.

We have examined the association between languages and software projects and practices in three studies: the first study discussed in Chapter 4 investigated the trends, directions and the popularity of languages. More importantly, it inspected the relation between user adoption and language design. It has revealed that statistical association exists.

The second study, Chapter 5, investigated how language features are used in practice and whether there is a significant association between language design and feature usage through mining source code files of OSS projects. The findings also shows significant association between language design and feature usage. However, the effect sizes of 62% of them are of small and trivial, and 38% are of medium and large effect size. Additionally, the results of significance and effect size hold only in a limited number of cases, when accounting for different project sizes and types. That is, when the hypothesis is tested in a small scale.

Finally, the third study on language association to OSS project attributes showed similar results (Chapter 6). That is, when the hypothesis of association was tested in a large scale, significant association is reported. However, the effect size in the vast majority of cases (93%) was modest. Moreover, when accounting for variability in project sizes and types, findings hold in exceedingly small cases.

The large size of the dataset can be attributed to the statistical significance, and this can explain the variations in the results when investigating the association in the small. Nevertheless, the effect size of the association is found to be modest in the majority of the tested cases. Thus, the choice of language has only a limited effect on OSS projects and practices.

The results provided here are based on the analysis of possibly the largest open source dataset through examining a population of 15,000 and by including a sample of about 5,350 projects that reflects current practices in programming and development practices. The study utilised a rigorous mining and statistical approach to investigate associations and inform state-of-the-practice of OSS projects and practices. The methods we have used to contribute to knowledge are:

- Investigates a large volume of data to find sufficient empirical evidence of language association with software development projects and practices.
- Investigates a dataset that consists of highly rated projects, where a main language that makes up at least 95% of the source code can be identified.
- Accounts for confounding factors such as project size and type when inspecting the effect of language.

7.2 Future work and limitations

Looking into programming languages from a pragmatic, empirical perspective helps understand current directions in programming and programming languages, and inform the design of new languages and language features. Future work should include an investigation into the language's extrinsic features, projects and teams nature and complexity, and examine the relationship between these aspects, and language choice. However, this research is based on data from OSS projects which may differ from closed source projects. In addition, the utilised methods can provide subjective outcomes, such as mining projects description using NLP algorithms and mining features using regular expressions. Moreover, besides language, software project and practices can be affected by other factors, such as programmer skills, project complexity, the development environment, etc. and assessing the effect of an individual factor can be quite challenging.

Appendix A

Programming languages ranked by popularity

Top 10 popular languages according to TIOBE, PYPL, RedMonk, and TrendySkills indexes during 2012-2019:

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	PHP	Java	JavaScript
3	C#	C	PHP	XML
4	C++	C++	Python	HTML
5	Objective-C	Python	Ruby	C#
6	PHP	JavaScript	C#	C++
7	Visual Basic	C#	C++	PHP
8	Python	Objective-C	C	C
9	Perl	Visual Basic	Objective-C	Perl
10	JavaScript/Ruby	Matlab	Shell	Python

Table A.1: Programming languages ranked by popularity in 2012.

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	PHP	Java	C#
3	C#	Python	PHP	JavaScript
4	C++	C#	Python	HTML
5	Objective-C	C++	C#	XML
6	PHP	C	C++	C++
7	Visual Basic	JavaScript	Ruby	PHP
8	Python	Objective-C	CSS	Python
9	Perl	Matlab	C	C
10	JavaScript/Ruby	Visual Basic	Objective-C/Perl/Shell	Perl

Table A.2: Programming languages ranked by popularity in 2013.

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	PHP	Java	JavaScript
3	C#	Python	PHP	C#
4	C++	C#	Python	HTML
5	Objective-C	C++	C#	PHP
6	PHP	C	C++	C++
7	Visual Basic	JavaScript	Ruby	XML
8	Python	Objective-C	CSS	Python
9	JavaScript	Matlab	C	C
10	Transact-SQL	Visual Basic	Objective-C	HTML5

Table A.3: Programming languages ranked by popularity in 2014.

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	PHP	Java	JavaScript
3	C#	Python	PHP	C#
4	C++	C#	Python	HTML
5	Objective-C	C++	C#	PHP
6	PHP	C	C++	C
7	Visual Basic	JavaScript	Ruby	C++
8	Python	Objective-C	CSS	HTML5
9	Perl	Matlab	C	Python
10	JavaScript	R/Swift	Objective-C	XML
	PL/SQL			
	Delphi/Object Pascal			

Table A.4: Programming languages ranked by popularity in 2015.

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	Python	Java	JavaScript
3	C#	PHP	PHP	C#
4	C++	C#	Python	HTML
5	PHP	C++	C#	PHP
6	Visual Basic	C	C++	Python
7	Python	JavaScript	Ruby	C++
8	Perl	Objective-C	CSS	HTML5
9	JavaScript	R	C	C
10	Ruby/Assembly	Swift/Matlab	Objective-C	XML

Table A.5: Programming languages ranked by popularity in 2016.

Table A.7 summarizes the total number of languages occurrences during 2012-2017:

	TIOBE	PYPL	RedMonk	TrendySkills
1	Java	Java	JavaScript	Java
2	C	Python	Java	JavaScript
3	C#	PHP	PHP	C#
4	C++	C#	Python	PHP
5	PHP	JavaScript	C#	HTML
6	Visual Basic	C++	C++	C++
7	Python	C	Ruby	C
8	Perl	Objective-C	CSS	HTML5
9	JavaScript	R	C	Python
10	Assembly	Swift	Objective-C	XML

Table A.6: Programming languages ranked by popularity in 2017.

Language	Total occurrences
C	24
C++	24
C#	24
Java	24
JavaScript	24
PHP	24
Python	24
Objective-C	16
Visual Basic	9
Ruby	9
Perl	8
HTML	6
XML	6
CSS	5
Matlab	5
HTML5	4
R	3
Swift	3
Assembly	2
Shell	2
Delphi	1
Object Pascal	1
PL	1
SQL	1
Transact-SQL	1

Table A.7: Programming languages ordered by total number of occurrences in TIOBE; PYPL; RedMonk; and TrendySkills popularity indexes during 2012-2017.

Appendix B

Extended statistics for the 2d study

Term occurrence	p	Z	N	e
S/D types				
Inheritance	0.000	-13.248	3283	0
Exceptions	0.000	-12.192	5254	0
Threads	0.000	-24.147	3450	0
Anonymous func.	0.000	-17.812	5042	0
Generics	0.000	-18.788	2090	0
S/D type no Obj-C				
Inheritance	0.000	-13.377	2975	0
Exceptions	0.000	-8.626	4946	0
Threads	0.000	-23.67	3142	0
Anonymous func.	0.000	-17.724	4734	0
Generics		Mann-Whitney Test cannot be performed on empty groups	1782	
S/W type				
Inheritance	0.000	-10.234	3283	0
Exceptions	0.000	-3.291	5254	0
Threads	0.000	-6.003	3450	0
Anonymous func.	0.000	-27.352	5042	0
Generics	0.000	-19.237	2090	0
S/W type_NoPHP				
Inheritance	0.000	-16.387	2993	0
Exceptions	0.000	-3.831	4964	0
Threads	0.000	-3.883	3160	0
Anonymous func.	0.000	-30.139	4752	0
Generics	0.000	-19.237	2090	0
S/W type_NoTS				
Inheritance	0.000	-10.291	3186	0
Exceptions	0.000	-3.449	5157	0
Threads	0.000	-6.003	3450	0
Anonymous func.	0.000	-28.912	4945	0
Generics	0.000	-19.447	1993	0
S/W type_NoPHPNoTS				
Inheritance	0.000	-16.424	2896	0
Exceptions	0.000	-3.975	4867	0
Threads	0.000	-3.883	3160	0
Anonymous func.	0.000	-31.6	4655	0
Generics	0.000	-19.447	1993	0
mgd/unmgd memory				
Inheritance	0.467	-0.727	3283	0
Exceptions	0	-17.554	5254	0
Threads	0.113	-1.586	3450	0
Anonymous func.	0	-3.885	5042	0
Generics	0	-15.655	2090	0

		Inheritance	Exceptions	Threads	Anonymous func.	Generics
Static	Mean	336	504	20	693	1558
	STD	1031	3126	68	3032	6355
	Min	0	0	0	0	0
	Max	12693	70626	1188	50022	133766
	Median	39	30	2	4	73
	Mode	2	0	0	0	0
Dynamic II	Mean	98	113	1	1169	0
	STD	335	511	6	6002	#DIV/0!
	Min	0	0	0	0	0
	Max	4551	20520	124	164607	0
	Median	14	16	0	55	#NUM!
	Mode	0	0	0	0	#N/A
Strong I	Mean	169	207	12	512	615
	STD	632	797	50	2594	2582
	Min	0	0	0	0	0
	Max	10733	18879	953	50022	33763
	Median	20	15	1	5	23
	Mode	0	0	0	0	0
Weak II	Mean	656	412	17	1634	4131
	STD	1497	3305	76	7049	11609
	Min	0	0	0	0	0
	Max	12693	70626	1188	164607	133766
	Median	197	20	2	204	889
	Mode	0	0	0	0	0
Managed	Mean	237	294	12	984	1570
	STD	826	2176	49	5000	6451
	Min	0	0	0	0	0
	Max	12693	70626	953	164607	133766
	Median	25	20	0	31	72
	Mode	0	0	0	0	0
Unmanaged	Mean	126	47	14	106	349
	STD	548	223	73	276	2294
	Min	0	0	0	0	0
	Max	9361	2163	1188	3173	40948
	Median	20	0	0	19	2
	Mode	6	0	0	0	0

Size	p	Z	N	eta squared n^2=z^2/(n-1)	r r = Z/âĹŽN	size n	size r
S/D type no Obj-C							
commits	0.000	-7.062	5001		-0.100		T
All_contributors	0.000	-15.095	5001		-0.213		S
Total/Code	0.000	-7.908	5001		-0.112		S
commits per contributor	0.000	-13.192	5001		-0.187		S
MONTHS	0.000	-13.061	5001		-0.185		S
Months(release)	0.000	-9.896	5001		-0.140		S
Tiny							
commits	0.000	-4.985	778		-0.179		S
All_contributors	0.000	-6.333	778		-0.227		S
Total/Code	0.000	-4.216	778		-0.151		S
commits per contributor	0.000	-3.956	778		-0.142		S
MONTHS	0.000	-4.108	778		-0.147		S
Months(release)	0.000	-4.744	778		-0.170		S
Small							
commits	0.000	-15.188	1687		-0.370		M
All_contributors	0.000	-16.942	1687		-0.412		M
Total/Code	0.700	-0.386	1687		-0.009		T
commits per contributor	0.000	-6.243	1687		-0.152		S
MONTHS	0.000	-12.393	1687		-0.302		M
Months(release)	0.000	-9.647	1687		-0.235		S
Medium							
commits	0	-7.861	1276		-0.220		S
All_contributors	0	-11.41	1276		-0.319		M
Total/Code	0.879	-0.152	1276		-0.004		T
commits per contributor	0	-6.882	1276		-0.193		S
MONTHS	0	-7.787	1276		-0.218		S
Months(release)	0	-5.494	1276		-0.154		S
Large							
commits	0.024	-2.262	548		-0.097		T
All_contributors	0	-4.758	548		-0.203		S
Total/Code	0.022	-2.296	548		-0.098		T
commits per contributor	0	-3.692	548		-0.158		S
MONTHS	0.006	-2.774	548		-0.118		S
Months(release)	0	-3.575	548		-0.153		S
V.Large							
commits	0.747	-0.323	712		-0.012		T
All_contributors	0.047	-1.986	712		-0.074		T
Total/Code	0	-5.22	712		-0.196		S
commits per contributor	0.001	-3.468	712		-0.130		S
MONTHS	0.01	-2.583	712		-0.097		T
Months(release)	0.647	-0.458	712		-0.017		T

S/W type_NoPHP					
commits	0.000	-9.068	5072	-0.127	S
All_contributors	0.000	-12.099	5072	-0.170	S
Total/Code	0.000	-5.234	5072	-0.073	T
commits per contributor	0.000	-3.81	5072	-0.053	T
MONTHS	0.000	-7.525	5072	-0.106	S
Months(release)	0.000	-5.521	5072	-0.078	T
Tiny					
commits	0	-4.378	777	-0.157	S
All_contributors	0	-5.261	777	-0.189	S
Total/Code	0	-4.021	777	-0.144	S
commits per contributor	0.008	-2.643	777	-0.095	T
MONTHS	0.019	-2.344	777	-0.084	T
Months(release)	0.014	-2.455	777	-0.088	T
Small					
commits	0	-9.463	1741	-0.227	S
All_contributors	0	-10.608	1741	-0.254	S
Total/Code	0.284	-1.072	1741	-0.026	T
commits per contributor	0.001	-3.22	1741	-0.077	T
MONTHS	0	-5.508	1741	-0.132	S
Months(release)	0	-4.503	1741	-0.108	S
Medium					
commits	0.592	-0.536	1295	-0.015	T
All_contributors	0	-4.476	1295	-0.124	S
Total/Code	0.107	-1.614	1295	-0.045	T
commits per contributor	0	-5.796	1295	-0.161	S
MONTHS	0	-3.542	1295	-0.098	T
Months(release)	0.014	-2.461	1295	-0.068	T
Large					
commits	0.161	-1.402	551	-0.060	T
All_contributors	0.303	-1.031	551	-0.044	T
Total/Code	0.339	-0.957	551	-0.041	T
commits per contributor	0.008	-2.658	551	-0.113	S
MONTHS	0.926	-0.092	551	-0.004	T
Months(release)	0.194	-1.298	551	-0.055	T
V.Large					
commits	0.838	-0.205	708	-0.008	T
All_contributors	0.909	-0.115	708	-0.004	T
Total/Code	0.118	-1.563	708	-0.059	T
commits per contributor	0.328	-0.977	708	-0.037	T
MONTHS	0.145	-1.456	708	-0.055	T
Months(release)	0.827	-0.219	708	-0.008	T

mgd/unmgd memory					
commits	0.000	-16.649	5350	-0.228	S
All_contributors	0.000	-22.567	5350	-0.309	M
Total/Code	0.681	-0.412	5350	-0.006	T
commits per contributor	0.000	-8.556	5350	-0.117	S
MONTHS	0.000	-11.154	5350	-0.152	S
Months(release)	0.000	-12.63	5350	-0.173	S
Tiny					
commits	0.000	-10.627	841	-0.366	M
All_contributors	0.000	-10.764	841	-0.371	M
Total/Code	0.000	-7.938	841	-0.274	S
commits per contributor	0.366	-0.905	841	-0.031	T
MONTHS	0.000	-5.058	841	-0.174	S
Months(release)	0.000	-6.649	841	-0.229	S
Small					
commits	0.000	-18.908	1847	-0.440	M
All_contributors	0.000	-19.093	1847	-0.444	M
Total/Code	0.036	-2.101	1847	-0.049	T
commits per contributor	0.000	-4.277	1847	-0.100	T
MONTHS	0.000	-10.531	1847	-0.245	S
Months(release)	0.000	-11.914	1847	-0.277	S
Medium					
commits	0.000	-10.445	1359	-0.283	S
All_contributors	0.000	-14.052	1359	-0.381	M
Total/Code	0.071	-1.806	1359	-0.049	T
commits per contributor	0.000	-7.032	1359	-0.191	S
MONTHS	0.000	-6.654	1359	-0.180	S
Months(release)	0.000	-7.306	1359	-0.198	S
Large					
commits	0.000	-4.349	576	-0.181	S
All_contributors	0.000	-6.491	576	-0.270	S
Total/Code	0.910	-0.113	576	-0.005	T
commits per contributor	0.001	-3.407	576	-0.142	S
MONTHS	0.171	-1.369	576	-0.057	T
Months(release)	0.001	-3.285	576	-0.137	S
V.Large					
commits	0.006	-2.74	727	-0.102	S
All_contributors	0.885	-0.145	727	-0.005	T
Total/Code	0.796	-0.258	727	-0.010	T
commits per contributor	0.000	-4.825	727	-0.179	S
MONTHS	0.015	-2.444	727	-0.091	T
Months(release)	0.016	-2.411	727	-0.089	T

Type	p	Z	N	eta squared	r	size r
Type_1						
commits	0.414	-0.817	898		-0.027	T
All_contributors	0.095	-1.672	898		-0.056	T
Total/Code	0.000	-5.399	898		-0.180	S
commits per contributor	0.000	-4.432	898		-0.148	S
MONTHS	0.003	-3.01357537	898		-0.101	S
Months(release)	0.002	-3.043	898		-0.102	S
Type_2						
commits	0.000	-6.693	627		-0.267	S
All_contributors	0.000	-8.652	627		-0.346	M
Total/Code	0.195	-1.295	627		-0.052	T
commits per contributor	0.016	-2.402	627		-0.096	T
MONTHS	0.000	-6.65500777	627		-0.266	S
Months(release)	0.000	-5.269	627		-0.210	S
Type_3						
commits	0.853	-0.185	888		-0.006	T
All_contributors	0.001	-3.247	888		-0.109	S
Total/Code	0.000	-7.470	888		-0.251	S
commits per contributor	0.000	-5.521	888		-0.185	S
MONTHS	0.000	-5.52235578	888		-0.185	S
Months(release)	0.000	-4.342	888		-0.146	S
Type_4						
commits	0.568	-0.571	544		-0.024	T
All_contributors	0.001	-3.419	544		-0.147	S
Total/Code	0.000	-4.711	544		-0.202	S
commits per contributor	0.000	-7.201	544		-0.309	M
MONTHS	0.002	-3.07300835	544		-0.132	S
Months(release)	0.012	-2.517	544		-0.108	S
Type_5						
commits	0.001	-3.420	495		-0.154	S
All_contributors	0.030	-2.166	495		-0.097	T
Total/Code	0.000	-4.991	495		-0.224	S
commits per contributor	0.001	-3.381	495		-0.152	S
MONTHS	0.055	-1.9174002	495		-0.086	T
Months(release)	0.976	-0.030	495		-0.001	T
S/W type_NoPHP						
commits	0.196	-1.292	804		-0.046	T
All_contributors	0.167	-1.381	804		-0.049	T
Total/Code	0.992	-0.010	804		0.000	T
commits per contributor	0.758	-0.309	804		-0.011	T
MONTHS	0.261	-1.125	804		-0.040	T
Months(release)	0.185	-1.325	804		-0.047	T
Type_2						
commits	0.000	-6.329	734		-0.234	S
All_contributors	0.000	-7.852	734		-0.290	S
Total/Code	0.001	-3.218	734		-0.119	S
commits per contributor	0.174	-1.361	734		-0.050	T
MONTHS	0.000	-5.028	734		-0.186	S
Months(release)	0.000	-5.027	734		-0.186	S
Type_3						
commits	0.225	-1.213	843		-0.042	T
All_contributors	0.511	-0.657	843		-0.023	T
Total/Code	0.256	-1.137	843		-0.039	T
commits per contributor	0.262	-1.122	843		-0.039	T

Ranks

	S_0/D_1	N	Mean Rank	Sum of Ranks
commits	0	2237	2340.73	5236210
	1	2764	2630.71	7271292
	Total	5001		
All_contributors	0	2237	2158.5	4828561
	1	2764	2778.2	7678941
	Total	5001		
Total/Code	0	2237	2680.47	5996220
	1	2764	2355.75	6511282
	Total	5001		
size	0	2237	2829.76	6330175
	1	2764	2234.92	6177326
	Total	5001		
commits per contributor	0	2237	2800.37	6264436
	1	2764	2258.71	6243066
	Total	5001		
Months(release)	0	1482	1527.9	2264345
	1	1958	1866.28	3654175
	Total	3440		

SW Test Statisticsa

	commits	All_contributors	Total/Code	commits per contributor	Month
Mann-Whitney U	2552117	2399369	2745518	2817384	125931
Wilcoxon W	7603548	7450800	7796949	4611949	339246
Z	-9.068	-12.099	-5.234	-3.81	-5.521
Asymp. Sig. (2-tailed)	0	0	0	0	0

a Grouping Variable: S_0/W_1

Ranks

	SW	N	Mean Rank	Sum of Ranks
commits	0	3178	2392.56	7603548
	1	1894	2778.03	5261581
	Total	5072		
All_contributors	0	3178	2344.49	7450800
	1	1894	2858.67	5414328
	Total	5072		
Total/Code	0	3178	2453.41	7796949
	1	1894	2675.91	5068179
	Total	5072		
size	0	3178	2625.39	8343494
	1	1894	2387.35	4521634
	Total	5072		
commits per contributor	0	3178	2596.97	8253180
	1	1894	2435.03	4611949
	Total	5072		
Months(release)	0	2065	1642.84	3392461
	1	1372	1833.63	2515742
	Total	3437		

Feature	p	Chi-Square	df	Effect Size	
Inheritance	0.00	202.31	2.00	0.25	M
Interfaces	0.00	3459.66	2.00	0.84	L
Exceptions	0.00	387.20	2.00	0.27	M
Threads	0.00	549.34	2.00	0.40	L
Anonymous Fncs	0.00	728.62	2.00	0.38	L
Generics	0.00	391.48	1.00	0.43	M

Table B.1: Hypothesis test statistics per language group (Binary mining) for static and dynamic languages.

Feature	p	Chi-Square	df	Effect Size	
Inheritance	0.00	11.89	2.00	0.06	<S
Interfaces	0.00	1690.05	2.00	0.59	L
Exceptions	0.00	17.18	2.00	0.06	<S
Threads	0.00	280.80	2.00	0.29	M
Anonymous Fncs	0.00	611.49	2.00	0.35	M
Generics	0.00	55.13	1.00	0.16	S

Table B.2: Hypothesis test statistics per language group (Binary mining) for strong and weak languages.

Feature	p	Chi2	df	Effect Size	
Inheritance	0.00	11.42	1.00	0.06	<S
Interfaces	0.00	106.85	2.00	0.14	S
Exceptions	0.00	381.61	1.00	.270 S	S
Threads	0.02	5.56	1.00	0.04	<S
Anonymous Fncs	0.02	5.14	1.00	0.03	<S
Generics	0.00	300.99	1.00	.379 M	M

Table B.3: Hypothesis test statistics per language group (Binary mining) for managed and unmanaged memory languages.

Appendix C

Extended statistics for the 3rd study

Mann-Whitney U-test per group:

S/D	Test Statisticsa				
		commits	All_contributors	Total/Code	commits per contributor
	Mann-Whitney U	2733007	2325358	2690052	2421836
	Wilcoxon W	5236210	4828561	6511282	6243066
	Z	-7.062	-15.095	-7.908	-13.192
	Asymp. Sig. (2-tailed)	0.000	0.000	0.000	0.000

Per project size:

Ranks				
	S_0/D_1	N	Mean Rank	Sum of Ranks
commits	0	2237	2340.73	5236210
	1	2764	2630.71	7271292
	Total	5001		
All_contributors	0	2237	2158.5	4828561
	1	2764	2778.2	7678941
	Total	5001		
Total/Code	0	2237	2680.47	5996220
	1	2764	2355.75	6511282
	Total	5001		
size	0	2237	2829.76	6330175
	1	2764	2234.92	6177326
	Total	5001		
commits per contributor	0	2237	2800.37	6264436
	1	2764	2258.71	6243066
	Total	5001		
Months(release)	0	1482	1527.9	2264345
	1	1958	1866.28	3654175
	Total	3440		

SW Test Statisticsa

	commits	All_contributors	Total/Code	commits per contributor	Month
Mann-Whitney U	2552117	2399369	2745518	2817384	125931
Wilcoxon W	7603548	7450800	7796949	4611949	339246
Z	-9.068	-12.099	-5.234	-3.81	-5.521
Asymp. Sig. (2-tailed)	0	0	0	0	0

a Grouping Variable: S_0/W_1

Ranks

	SW	N	Mean Rank	Sum of Ranks
commits	0	3178	2392.56	7603548
	1	1894	2778.03	5261581
	Total	5072		
All_contributors	0	3178	2344.49	7450800
	1	1894	2858.67	5414328
	Total	5072		
Total/Code	0	3178	2453.41	7796949
	1	1894	2675.91	5068179
	Total	5072		
size	0	3178	2625.39	8343494
	1	1894	2387.35	4521634
	Total	5072		
commits per contributor	0	3178	2596.97	8253180
	1	1894	2435.03	4611949
	Total	5072		
Months(release)	0	2065	1642.84	3392461
	1	1372	1833.63	2515742
	Total	3437		

Memory Test Statisticsa

	commits	All_contributors	Total/Code	commits per contributor	M
Mann-Whitney U	2149792	1843288	2991497	2569346	95
Wilcoxon W	3449870	3143366	9979688	9557537	14
Z	-16.649	-22.567	-0.412	-8.556	-1
Asymp. Sig. (2-tailed)	0	0	0.681	0	0

a Grouping Variable: Mgd_0/Un_1

Ranks

	Memory	N	Mean Rank	Sum of Ranks
commits	0	3738	2906.38	
	1	1612	2140.12	3449870
	Total	5350		
All_contributors	0	3738	2988.38	11170559
	1	1612	1949.98	3143366
	Total	5350		
Total/Code	0	3738	2669.79	9979687.5
	1	1612	2688.73	4334237.5
	Total	5350		
size	0	3738	2531.44	9462537
	1	1612	3009.55	4851388
	Total	5350		
commits per contributor	0	3738	2556.86	9557537
	1	1612	2950.61	4756388
	Total	5350		
Months(release)	0	2710	1974.59	5351149.5
	1	973	1472.7	1432936.5
	Total	3683		

Size	p	Z	N	eta squared n^2=z^2/(n-1)	r r = Z/âĹŽN	size n	size r
S/D type no Obj-C							
commits	0.000	-7.062	5001		-0.100		T
All_contributors	0.000	-15.095	5001		-0.213		S
Total/Code	0.000	-7.908	5001		-0.112		S
commits per contributor	0.000	-13.192	5001		-0.187		S
MONTHS	0.000	-13.061	5001		-0.185		S
Months(release)	0.000	-9.896	5001		-0.140		S
Tiny							
commits	0.000	-4.985	778		-0.179		S
All_contributors	0.000	-6.333	778		-0.227		S
Total/Code	0.000	-4.216	778		-0.151		S
commits per contributor	0.000	-3.956	778		-0.142		S
MONTHS	0.000	-4.108	778		-0.147		S
Months(release)	0.000	-4.744	778		-0.170		S
Small							
commits	0.000	-15.188	1687		-0.370		M
All_contributors	0.000	-16.942	1687		-0.412		M
Total/Code	0.700	-0.386	1687		-0.009		T
commits per contributor	0.000	-6.243	1687		-0.152		S
MONTHS	0.000	-12.393	1687		-0.302		M
Months(release)	0.000	-9.647	1687		-0.235		S
Medium							
commits	0	-7.861	1276		-0.220		S
All_contributors	0	-11.41	1276		-0.319		M
Total/Code	0.879	-0.152	1276		-0.004		T
commits per contributor	0	-6.882	1276		-0.193		S
MONTHS	0	-7.787	1276		-0.218		S
Months(release)	0	-5.494	1276		-0.154		S
Large							
commits	0.024	-2.262	548		-0.097		T
All_contributors	0	-4.758	548		-0.203		S
Total/Code	0.022	-2.296	548		-0.098		T
commits per contributor	0	-3.692	548		-0.158		S
MONTHS	0.006	-2.774	548		-0.118		S
Months(release)	0	-3.575	548		-0.153		S
V.Large							
commits	0.747	-0.323	712		-0.012		T
All_contributors	0.047	-1.986	712		-0.074		T
Total/Code	0	-5.22	712		-0.196		S
commits per contributor	0.001	-3.468	712		-0.130		S
MONTHS	0.01	-2.583	712		-0.097		T
Months(release)	0.647	-0.458	712		-0.017		T

S/W type_NoPHP					
commits	0.000	-9.068	5072	-0.127	S
All_contributors	0.000	-12.099	5072	-0.170	S
Total/Code	0.000	-5.234	5072	-0.073	T
commits per contributor	0.000	-3.81	5072	-0.053	T
MONTHS	0.000	-7.525	5072	-0.106	S
Months(release)	0.000	-5.521	5072	-0.078	T
Tiny					
commits	0	-4.378	777	-0.157	S
All_contributors	0	-5.261	777	-0.189	S
Total/Code	0	-4.021	777	-0.144	S
commits per contributor	0.008	-2.643	777	-0.095	T
MONTHS	0.019	-2.344	777	-0.084	T
Months(release)	0.014	-2.455	777	-0.088	T
Small					
commits	0	-9.463	1741	-0.227	S
All_contributors	0	-10.608	1741	-0.254	S
Total/Code	0.284	-1.072	1741	-0.026	T
commits per contributor	0.001	-3.22	1741	-0.077	T
MONTHS	0	-5.508	1741	-0.132	S
Months(release)	0	-4.503	1741	-0.108	S
Medium					
commits	0.592	-0.536	1295	-0.015	T
All_contributors	0	-4.476	1295	-0.124	S
Total/Code	0.107	-1.614	1295	-0.045	T
commits per contributor	0	-5.796	1295	-0.161	S
MONTHS	0	-3.542	1295	-0.098	T
Months(release)	0.014	-2.461	1295	-0.068	T
Large					
commits	0.161	-1.402	551	-0.060	T
All_contributors	0.303	-1.031	551	-0.044	T
Total/Code	0.339	-0.957	551	-0.041	T
commits per contributor	0.008	-2.658	551	-0.113	S
MONTHS	0.926	-0.092	551	-0.004	T
Months(release)	0.194	-1.298	551	-0.055	T
V.Large					
commits	0.838	-0.205	708	-0.008	T
All_contributors	0.909	-0.115	708	-0.004	T
Total/Code	0.118	-1.563	708	-0.059	T
commits per contributor	0.328	-0.977	708	-0.037	T
MONTHS	0.145	-1.456	708	-0.055	T
Months(release)	0.827	-0.219	708	-0.008	T

mgd/unmgd memory					
commits	0.000	-16.649	5350	-0.228	S
All_contributors	0.000	-22.567	5350	-0.309	M
Total/Code	0.681	-0.412	5350	-0.006	T
commits per contributor	0.000	-8.556	5350	-0.117	S
MONTHS	0.000	-11.154	5350	-0.152	S
Months(release)	0.000	-12.63	5350	-0.173	S
Tiny					
commits	0.000	-10.627	841	-0.366	M
All_contributors	0.000	-10.764	841	-0.371	M
Total/Code	0.000	-7.938	841	-0.274	S
commits per contributor	0.366	-0.905	841	-0.031	T
MONTHS	0.000	-5.058	841	-0.174	S
Months(release)	0.000	-6.649	841	-0.229	S
Small					
commits	0.000	-18.908	1847	-0.440	M
All_contributors	0.000	-19.093	1847	-0.444	M
Total/Code	0.036	-2.101	1847	-0.049	T
commits per contributor	0.000	-4.277	1847	-0.100	T
MONTHS	0.000	-10.531	1847	-0.245	S
Months(release)	0.000	-11.914	1847	-0.277	S
Medium					
commits	0.000	-10.445	1359	-0.283	S
All_contributors	0.000	-14.052	1359	-0.381	M
Total/Code	0.071	-1.806	1359	-0.049	T
commits per contributor	0.000	-7.032	1359	-0.191	S
MONTHS	0.000	-6.654	1359	-0.180	S
Months(release)	0.000	-7.306	1359	-0.198	S
Large					
commits	0.000	-4.349	576	-0.181	S
All_contributors	0.000	-6.491	576	-0.270	S
Total/Code	0.910	-0.113	576	-0.005	T
commits per contributor	0.001	-3.407	576	-0.142	S
MONTHS	0.171	-1.369	576	-0.057	T
Months(release)	0.001	-3.285	576	-0.137	S
V.Large					
commits	0.006	-2.74	727	-0.102	S
All_contributors	0.885	-0.145	727	-0.005	T
Total/Code	0.796	-0.258	727	-0.010	T
commits per contributor	0.000	-4.825	727	-0.179	S
MONTHS	0.015	-2.444	727	-0.091	T
Months(release)	0.016	-2.411	727	-0.089	T

Type	p	Z	N	eta squared	r	size r
Type_1						
commits	0.414	-0.817	898		-0.027	T
All_contributors	0.095	-1.672	898		-0.056	T
Total/Code	0.000	-5.399	898		-0.180	S
commits per contributor	0.000	-4.432	898		-0.148	S
MONTHS	0.003	-3.01357537	898		-0.101	S
Months(release)	0.002	-3.043	898		-0.102	S
Type_2						
commits	0.000	-6.693	627		-0.267	S
All_contributors	0.000	-8.652	627		-0.346	M
Total/Code	0.195	-1.295	627		-0.052	T
commits per contributor	0.016	-2.402	627		-0.096	T
MONTHS	0.000	-6.65500777	627		-0.266	S
Months(release)	0.000	-5.269	627		-0.210	S
Type_3						
commits	0.853	-0.185	888		-0.006	T
All_contributors	0.001	-3.247	888		-0.109	S
Total/Code	0.000	-7.470	888		-0.251	S
commits per contributor	0.000	-5.521	888		-0.185	S
MONTHS	0.000	-5.52235578	888		-0.185	S
Months(release)	0.000	-4.342	888		-0.146	S
Type_4						
commits	0.568	-0.571	544		-0.024	T
All_contributors	0.001	-3.419	544		-0.147	S
Total/Code	0.000	-4.711	544		-0.202	S
commits per contributor	0.000	-7.201	544		-0.309	M
MONTHS	0.002	-3.07300835	544		-0.132	S
Months(release)	0.012	-2.517	544		-0.108	S
Type_5						
commits	0.001	-3.420	495		-0.154	S
All_contributors	0.030	-2.166	495		-0.097	T
Total/Code	0.000	-4.991	495		-0.224	S
commits per contributor	0.001	-3.381	495		-0.152	S
MONTHS	0.055	-1.9174002	495		-0.086	T
Months(release)	0.976	-0.030	495		-0.001	T
S/W type_NoPHP						
commits	0.196	-1.292	804		-0.046	T
All_contributors	0.167	-1.381	804		-0.049	T
Total/Code	0.992	-0.010	804		0.000	T
commits per contributor	0.758	-0.309	804		-0.011	T
MONTHS	0.261	-1.125	804		-0.040	T
Months(release)	0.185	-1.325	804		-0.047	T
Type_2						
commits	0.000	-6.329	734		-0.234	S
All_contributors	0.000	-7.852	734		-0.290	S
Total/Code	0.001	-3.218	734		-0.119	S
commits per contributor	0.174	-1.361	734		-0.050	T
MONTHS	0.000	-5.028	734		-0.186	S
Months(release)	0.000	-5.027	734		-0.186	S
Type_3						
commits	0.225	-1.213	843		-0.042	T
All_contributors	0.511	-0.657	843		-0.023	T
Total/Code	0.256	-1.137	843		-0.039	T
commits per contributor	0.262	-1.122	843		-0.039	T

Bibliography

Agrawal, R. & Srikant, R. (1994), Fast Algorithms for Mining Association Rules, *in* 'Proc. 20th int. conf. very large data bases, VLDB', pp. 487—-499.
URL: *http://www.vldb.org/conf/1994/P487.PDF*

Arnold, K., Gosling, J. & Holmes, D. (2005), *The Java programming language*, Addison Wesley Professional.

Babb, Luke (2016), 'Gitlab'.
URL: *https://about.gitlab.com/*

Berger, E. D., Hollenbeck, C., Maj, P., Vitek, O. & Vitek, J. (2019*a*), 'On the impact of programming languages on code quality: A reproduction study', *ACM Transactions on Programming Languages and Systems* **41**(4).

Berger, E. D., Hollenbeck, C., Maj, P., Vitek, O. & Vitek, J. (2019*b*), On the impact of programming languages on code quality: A reproduction study, *in* 'ACM Transactions on Programming Languages and Systems', Vol. 41.
URL: *http://eds.b.ebscohost.com/eds/pdfviewer/pdfviewer?vid=3&sid=15c2405b-a22e-4195-bdf7-62ea306b1a87%40sessionmgr101*

Bhattacharya, P. & Neamtiu, I. (2011*a*), 'Assessing Programming Language Impact on Development and Maintenance : A Study on C and C ++', *Software Engineering (ICSE), 2011 33rd International Conference on* pp. 171–180.

Bhattacharya, P. & Neamtiu, I. (2011*b*), 'Assessing programming language impact on development and maintenance: a study on c and c++', *Software Engineering (ICSE), 2011 33rd International Conference on* (3), 171–180.

Blei, D. M., Ng, A. Y. & Jordan, M. I. (2003), 'Latent dirichlet allocation', *Journal of machine Learning research* **3**(Jan), 993–1022.

Boehm, B. (2006), A view of 20th and 21st century software engineering, *in* 'Proceedings of the 28th international conference on Software engineering', ACM, pp. 12–29.
URL: *http://portal.acm.org/citation.cfm?doid=1134285.1134288*

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. & Selby, R. (1995), 'Cost models for future software life cycle processes: COCOMO 2.0', *Annals of Software*

Engineering **1**(1), 57–94.
URL: *http://link.springer.com/10.1007/BF02249046*

Boehm, B. W. (1981), 'An Experiment in Small-Scale Application Software Engineering', *IEEE Transactions on software engineering* (5), 482–493.

Boehm, B. W. (1987), 'Improving Software Productivity', *Computer* pp. 43—-47.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.365.9038*

Budd, T. (2002), *An Introduction to Object-oriented Programming*, Addison-Wesley.

Canonical Ltd. (2017), 'Launchpad'.
URL: *https://launchpad.net/people*

Capers Jones (2008), *Applied Software Measurement: Global Analysis of Productivity and Quality*, McGraw-Hill Education Group.
URL: *https://dl.acm.org/citation.cfm?id=2823929*

Chen, Y., Mili, A., Wu, L., Dios, R. & Wang, K. (2005), 'Programming Language Trends: An Empirical Study', *IEEE software* **22**(3), 72–79.

Cosentino, V., Luis, J. & Cabot, J. (2016), Findings from GitHub: Methods, Datasets and Limitations, *in* 'Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16', ACM Press, New York, New York, USA, pp. 137–141.
URL: *http://dl.acm.org/citation.cfm?doid=2901739.2901776*

Davis, Justine (2016), 'Bitbucket'.
URL: *https://blog.bitbucket.org/2016/09/07/bitbucket-cloud-5-million-developers-900000-teams/*

Delorey, D. P., Knutson, C. D. & Chun, S. (2007), Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects, *in* 'Emerging Trends in FLOSS Research and Development, 2007. FLOSS'07. First International Workshop on', IEEE.

Deshpande, A. & Riehle, D. (2008), The total growth of open source, *in* 'IFIP International Federation for Information Processing', Vol. 275, pp. 197–209.

Detlefs, D., Dosser, A. & Zorn, B. (1994), 'Memory Allocation Costs in Large C and C++ Programs', *Software: Practice and Experience* **24**(6), 527–542.

Dircks, H. (1981), 'Sofcost'- grumman's software cost estimating model', *NAECON 1981* pp. 674–683.

Field, A., Miles, J. & Field, Z. (2012), *Discovering statistics using R*, Sage publications.

Flanagan, D. (1998), *JavaScript: The Definitive Guide*, O'Reilly Media.

Github (n.d.), 'GitHub'.
URL: *https://github.com/about*

Gousios, G., Vasilescu, B., Serebrenik, A. & Zaidman, A. (2014), Lean GHTorrent: GitHub Data on Demand, *in* 'Proceedings of the 11th working conference on mining software repositories', pp. 384–387.
URL: *https://bvasiles.github.io/papers/lean-ghtorrent.pdf*

Harrison, W. & Adrangi, B. (1986), 'The role of programming language in estimating software development costs', *Journal of Management Information Systems* **3**(3), 101–110.

Hofmann, T. (1999), Probabilistic latent semantic indexing, *in* 'Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval', pp. 50–57.

Hudak, P. & Paul (1989), 'Conception, evolution, and application of functional programming languages', *ACM Computing Surveys* **21**(3), 359–411.

Igarashi, Y., Altman, T., Funada, M. & Kamiyama, B. (2014), *Computing: A Historical and Technical Perspective*, CRC Press.

Jensen, R. (1983), An improved macrolevel software development resource estimation model, *in* '5th ISPA Conference', pp. 88–92.

Jones, C. (1996), 'Programming Languages Table, Release 8.2', *Software Productivity Research, Burlington, MA* .

Jones, C. & Bonsignour, O. (2011), *The Economics of Software Quality*, 1st edn, Addison-Wesley Professional.
URL: *http://dl.acm.org/citation.cfm?id=2025240*

Karus, S. & Gall, H. (2011), A study of language usage evolution in open source software, *in* 'Proceeding of the 8th working conference on Mining software repositories - MSR '11', ACM Press, New York, New York, USA, p. 13.
URL: *http://portal.acm.org/citation.cfm?doid=1985441.1985447*

Lantz, B. (2013), 'The large sample size fallacy', *Scandinavian journal of caring sciences* **27**(2), 487–492.

Lavazza, L., Morasca, S. & Tosi, D. (2016), An Empirical Study on the Effect of Programming Languages on Productivity, *in* 'Proceedings of the 31st Annual ACM Symposium on Applied Computing', ACM, pp. 1434—-1439.

Lavrakas, P. J. (2008), Trend Analysis, *in* 'Encyclopedia of Survey Research Methods', Sage Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States of America.

URL: *http://methods.sagepub.com/reference/encyclopedia-of-survey-research-methods/n589.xml*

Leeuw, J. D., De Leeuw, J., Udina, F. & Greenacre, M. (2001), 'Reproducible Research: the Bottom Line'.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.3507*

Lévénez, É. (2016), 'Computer Languages History'.
URL: *https://www.levenez.com/lang/*

Liao, S.-H., Chu, P.-H. & Hsiao, P.-Y. (2012), 'Data mining techniques and applications–a decade review from 2000 to 2011', *Expert systems with applications* **39**(12), 11303–11311.

Markstrum, S. (2010), Staking claims: a history of programming language design claims and evidence, *in* 'Evaluation and Usability of Programming Languages and Tools', ACM, p. 7.

Mayer, P. & Bauer, A. (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.
URL: *http://dx.doi.org/10.1145/2745802.2745805*

Meyerovich, L. A. & Rabkin, A. S. (2013), Empirical analysis of programming language adoption, *in* 'Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications - OOPSLA '13', Vol. 48, ACM Press, New York, New York, USA, pp. 1–18.
URL: *http://dl.acm.org/citation.cfm?doid=2509136.2509515*

Ming-Wei Wu, Ying-Dar Lin, Wu, M.-w. & Lin, Y.-d. (2001), 'Open source software development: an overview', *Computer* **34**(6), 33–38.
URL: *http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=928619*

Myrtveit, I. & Stensrud, E. (2008), 'An empirical study of software development productivity in C and C++', *Proc. Norsk Informatikkonferanse* p. 131.

Nanz, S. & Furia, C. A. (2015), A comparative study of programming languages in rosetta code, *in* 'Proceedings - International Conference on Software Engineering', Vol. 1, IEEE, pp. 778–788.
URL: *http://ieeexplore.ieee.org/document/7194625/*

Nelson, E. A. (1967), Management handbook for the estimation of computer programming costs, Technical report, SYSTEM DEVELOPMENT CORP SANTA MONICA CA.

Okur, S. & Dig, D. (2012), How do developers use parallel libraries?, *in* 'Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software

Engineering - FSE '12', ACM Press, New York, New York, USA, p. 1.
URL: *http://dl.acm.org/citation.cfm?doid=2393596.2393660*

Oracle (2016), 'Java Programming Language Enhancements'.
URL: *http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html*

O'Regan, G. (2008), *A brief history of computing*, Springer Science & Business Media.

Orru, M., Tempero, E., Marchesi, M. & Tonelli, R. (2015), How do python programs use inheritance? a replication study, *in* '2015 Asia-Pacific Software Engineering Conference (APSEC)', IEEE, pp. 309–315.

Osman, H., Chis, A., Corrodi, C., Ghafari, M. & Nierstrasz, O. (2017), Exception Evolution in Long-Lived Java Systems, *in* 'IEEE International Working Conference on Mining Software Repositories', pp. 302–311.
URL: *http://scg.unibe.ch/archive/papers/Osma17d-exception-evolution.pdf*

Ousterhout, J. (1998*a*), 'Scripting: higher level programming for the 21st Century', *Computer* **31**(3), 23–30.
URL: *http://ieeexplore.ieee.org/document/660187/*

Ousterhout, J. K. (1998*b*), 'Scripting: Higher level programming for the 21st century', *Computer* **31**(3), 23–30.

Parnin, C., Bird, C. & Murphy-Hill, E. (2011), Java generics adoption: How New Features are Introduced, Championed, or Ignored, *in* 'Proceeding of the 8th working conference on Mining software repositories - MSR '11', ACM Press, New York, New York, USA, p. 3.
URL: *http://portal.acm.org/citation.cfm?doid=1985441.1985446*

Petricek, T. & Skeet, J. (2009), *Real World Functional Programming: With Examples in F# and C*, Manning Publications Co.

Phipps, G. (1999), 'Comparing observed bug and productivity rates for Java and C++', *Software, Practice and Experience* **29**(4), 345–358.

Port, D. & McArthur, M. (1999), A study of productivity and efficiency for object-oriented methods and languages, *in* 'Software Engineering Conference, 1999.(APSEC'99) Proceedings. Sixth Asia Pacific', IEEE, pp. 128–135.

Prechelt, L. (2000), 'An Empirical Comparison of Seven Programming Languages', *Computer* **33**(10), 23–29.

Putnam, L. H. (1978), 'A general empirical solution to the macro software sizing and estimating problem', *IEEE transactions on Software Engineering* (4), 345–361.

Rajaraman, V. (1997), *Computer Programming in Fortran 90 and 95*, PHI Learning Pvt. Ltd.

Ray, B., Posnett, D., Filkov, V. & Devanbu, P. (2014), A large scale study of programming languages and code quality in github, *in* 'Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014', ACM Press, New York, New York, USA, pp. 155–165.
URL: *http://dl.acm.org/citation.cfm?doid=2635868.2635922*

Raymond, E. (1999), 'The cathedral and the bazaar', *Knowledge, Technology & Policy* **12**(3), 23–49.

Robles, G. (2010), Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings, *in* '2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)', IEEE, pp. 171–180.
URL: *http://ieeexplore.ieee.org/document/5463348/*

Rossum, G. v. (2003), *An Introduction to Python*, Network Theory Ltd.

Sanatinia, A. & Noubir, G. (2016), 'On GitHub's Programming Languages', *arXiv preprint arXiv:1603.00431* .
URL: *https://api.github.com/repositories.*

Schneider, V. (1978), 'Prediction of software effort and project duration- four new formulas', *ACM SIGPLAN Notices* **13441**(1), 49–59.

Sebesta, R. W. (2012), *Concepts of Programming Languages (10th edition)*.

Shull, F. J., Carver, J. C., Vegas, S. & Juristo, N. (2008), 'The role of replications in Empirical Software Engineering', *Empirical Software Engineering* **13**(2), 211–218.
URL: *http://link.springer.com/10.1007/s10664-008-9060-1*

Shull, F., Mendonccça, M. G., Basili, V., Carver, J., Maldonado, J. C., Fabbri, S., Travassos, G. H. & Ferreira, M. C. (2004), 'Knowledge-Sharing Issues in Experimental Software Engineering', *Empirical Software Engineering* **9**(1/2), 111–137.
URL: *http://link.springer.com/10.1023/B:EMSE.0000013516.80487.33*

Slashdot Media (2017), 'Sourceforge'.
URL: *https://sourceforge.net/about*

Software Project Benchmarking - ISBSG (2012).
URL: *http://isbsg.org/*

Sommerville, I. (2011), *Software Engineering 9 (International Edition)*, 9th edn, Pearson, Boston, USA.

Stroustrup, B. (1999), 'An Overview of the C++ Programming Language', *Handbook of object technology* .
URL: *http://www.stroustrup.com/crc.pdf*

Sureka, A., Tripathi, A. & Dabral, S. (2015), 'Survey Results on Threats To External Validity, Generalizability Concerns, Data Sharing and University-Industry Collaboration in Mining Software Repository (MSR) Research', *arXiv preprint arXiv:1506.01499* .

URL: *http://bit.ly/1CXOV3r*

Swift.org (n.d.).

URL: *https://swift.org/*

Tempero, E., Noble, J. & Melton, H. (2008), How do java programs use inheritance? an empirical study of inheritance in java software, *in* 'European Conference on Object-Oriented Programming', Springer, pp. 667–691.

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J. & Daleiden, P. (2016), An empirical study on the impact of C++ lambdas and programmer experience, *in* 'Proceedings of the 38th International Conference on Software Engineering - ICSE '16', ACM Press, New York, New York, USA, pp. 760–771.

URL: *http://dl.acm.org/citation.cfm?doid=2884781.2884849*

Vegas, S., Juristo, N., Moreno, A., Solari, M. & Letelier, P. (2006), Analysis of the influence of communication between researchers on experiment replication, *in* 'Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06', ACM Press, New York, New York, USA, p. 28.

URL: *http://portal.acm.org/citation.cfm?doid=1159733.1159741*

W.Sebesta, R. (2008), *Concepts of Programming Languages*, Pearson Education.

Wu, T. (2009), *An Introduction to Object-Oriented Programming with Java*, McGraw-Hill.

Wulf, W. A. (1980), 'Trends in the Design and Implementation of Programming Languages', *Computer* **13**(1), 14–25.

Zenoss Inc (2016), 2016 State of Open Source Software, Technical report.

URL: *https://ownit.zenoss.com/assets/state-of-oss.pdf*