Thesis no:
URI: urn:nbn:se:bth-16837

Programming Languages

Improvements, popularity, and the need of the future

Philip Norlin

Valentin Wannesian

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the bachelor degree in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

CONTACT INFORMATION:

Authors:
Philip Norlin
phille.norlin@gmail.com

Valentin Wannesian
valentin.wanne@gmail.com

University advisor:
Nina Dzamashvili Fogelström
Institution of Software Engineering

ABSTRACT

Keywords: programming, languages, characteristics, improvements.

Programming languages have come a long way over the past decades and a lot of options are available. To find out where developers want languages to go in the future, and how developers determine what language is suited for their projects.

The methods of research were performed using a compiled survey and a interview made for this research as well as external ones complemented with literatures used to crosscheck to determine accountability.

Results show that modern languages have improved compared to their predecessors in the terms of syntax, ease of use, and automation. Through large communities, languages have gained popularity from the vast resources available to learn from. Developers have shown their opinions on the future of languages and what they want to see from them.
What languages a programmer decides for a project weighs heavily between what they are accustomed to, or what the project requirements limits them too. It was shown that if the programmer would be able to pick a language, it would be based on what they already know. Sticking to what the project's requirements was the most picked option in the survey conducted for the result. Developers want more simplicity, functionality, safety, and compatibility in the future. Through more compatibility with cross platforms, nicer looking code structure with easier to write syntaxes, more complex functionality, and better safety against bad code.

Table of content

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Internet: www.bth.se
Phone: +46 455 38 50
Fax: +46 455 38 50 57

1. Introduction

In this chapter we shortly explain what a programming language is and what kinds there are as well as what differs them in the major sense. We will present some background to give an understanding of how the language situation has evolved and and what we question with the situation. We also present the purpose of this study and its value as well as the scope we intend to cover in an attempt to limit our research to only the that of interest.

1.1 Background

A programming language is a collection of instructions used to create various kinds of outputs from a computer. They are often divided into two major categories, low level and high level. A low level language is machine readable form such as *machine code* which contains *binary code*. High level language is human readable form such as *Java*, and *C*. Low level language are platform dependent meaning they can run on the same hardware and configuration while high level are platform independent as they can run on different hardware and configuration [12]. From the first high level programming languages of the early 1960s developed by *Konrad Zuse* [13] to the hundred of choices in 2018, there have been many languages with diverse functionality and usage [14]. New languages are created to increase functionality, to better performance, or solve current problems.

How do these languages compare to the older ones? What have they improved? What is it that the developers, that will utilize these languages, really want in the future?
As a vast amount of languages have appeared, researching to see why some reach popularity and why some fall in the cracks provide a good historical understanding of the state they are in.

1.2 Definitions

1.2.1 Developer

An individual that creates software and applications.

1.2.2 Business

In the case of this paper, a business will be referring to an organization that works within the IT field either by developing software or managing it.

1.2.3 Programming language

Also referred to as 'language', is a collection of instructions used to create various kinds of outputs from a computer.

1.2.4 Memory leak

A resource which continuously allocates memory, in which the end result could be a program crash due to all memory being in use.

1.2.5 Undefined behavior

When executed code violates the language specification and its behaviour cannot be determined.

1.3 Purpose

The purpose is to provide a basic overview for businesses and developers to research on and get a general understanding of what factors in a language, and opinions that the users have on languages and their usage. As a invaluable source, it allows businesses to better choose a language to support their developers and software, and developers to better identify the pros of a language.

The goal for this document is to bring an understanding of what causes a language to become a success, what can developers do to improve upon their languages, and what can be done to make a language attractive to developers. Written in a way that business owners and technologically interested people can understand without much in-depth knowledge on the subject, this paper aims to bring answers to a series of questions regarding comparisons, popularity factors, wants and needs of the language users.

1.4 Scope

This paper is about researching the different factors that affect a programming languages popularity, how the languages have improved compared to their older counterparts, what developers think and want from the future of languages, and how they decide what to use for their projects.

1.4.1 Limitations

The paper will be limited to include programming languages, developers, their opinions, and comparisons between languages, and will focus solely on the languages in their most basic form, not covering external tools, business models, or creators. The comparisons of languages and their improvements compared to their predecessors were limited to C-like languages, mainly because of its legacy and because active languages used today have based their syntax and design on C while also advertising them as being a part of the C-family of languages.

1.4.2 Motivation

The reason for our limitations is because we're covering a large subject and do not have the time or manpower to go through every available language and a very large data set.
What we are doing is to skim the top of the programming languages and find out certain details regarding their popularity and the improvements they bring. We also skim the top of the developers opinion on the future of programming languages and what they actually want themselves to see in the future.

This is an important topic because it shows what is necessary for a successful language, it gives developers and idea of what is needed to evolve in the future of languages, it allows companies to understand the process of which developers decide the foundation to build software.

2. Research Questions

RQ1. How do newer languages compare to their predecessors in regards to what they do better?

Comparisons between the new languages and their predecessor are important to determine if a language is superior or has improved. This could be in relation to *speed*, *maintainability*, *performance*, or *functionality* available. It will assist in being able to tell when a language is better as well as valuable information if previously existing issues have been resolved which can in turn determine if it is a worthwhile investment to change language.

The goal of the research here is to find why some are considered better, and what gives them this point. The importance of this is to see if languages have improved over the decades and if they have solved the problems older languages are facing.

We expect to find factors between languages that are considered better than their predecessors and counterparts, such as improved *performance*, *nicer syntax*, *easier maintainability*. We hope that we uncover these factors to be able to show that languages are improving and bringing new technology to the table.

RQ2. What are the deciding factors for when choosing a programming language for a project?

Deciding what programming language to use in a project could have a significant impact on the product. Whether projects want to be easily maintained, implemented fast, or perform at its best relies heavily on what the language is capable of, and the associated binaries that can translate the language to produce functioning and a well performed application.

The goal of the research here hopes to answer what the standard thought process are and what results are to be expected once a decision has been made, and what challenges could occur. The importance of this lies in the ability to quickly be able to determine what limits a project when it comes to choice of language and how developers think when deciding the path to take.

We expect to find a thought process involving certain factors such as performance, stability, or scalability, when it comes to deciding the language for a project. Due to the different limitations of languages and how requirements for projects differ and don't always match with the developers ability, we want to know how a developer thinks when they decide on a language.

RQ3. Why are some languages more popular than others?
A popular language is usually defined by certain factors, a few examples are *syntax*, *accessibility*, *ease of use*, *community* and *security*. Finding out why a programming language fades in popularity could help pave the way for making future languages better. How to avoid the same mistakes or to bring better design and market choices to the table, this is highly important to not repeat old mistakes and eventually help filter out 'bad' languages by not giving them the opportunity to grow due to them not having the 'popularity factors'.
Our goal is to find out what exactly causes a programming language to gain popularity, whether it is a newly release language, or something that has existed for a while.
We expect to find some sort of pattern of popularity and be able to compare it to the less popular ones following the pattern to find if this is true, a pattern could be anything from a language being popular due to its easy syntax and large community, or high complexity and functionality. We believe this to be the case because most popular languages tend to be similar in some ways and being able to pinpoint that detail should be of great benefit to the future languages.

RQ4. What is it developers expect and want from newer languages?
Developers in all the IT fields are the target audience for these languages so their opinion should be of the highest interest to the language developers, knowing what their user base want will help in bringing quality, features, functionality, and fixing older problems [15]. This is important to allow the evolution of programming languages to head the direction we need it to go.
The goal is to provide a small base of information where developers have stated what they expect and want from a language and how they work with them.
This is highly important for someone creating a language due to the direct opinion they can focus their work on making the improvements that are necessary to please their user base.
We expect to have a collection of information from our survey data and be able to find a common consensus as to what factors, needs, or wants that developers wish to see in the future. Due to us collecting and comparing survey data to other sources we hope that it will help us answer this question as knowing the what the most sought out 'want' with future languages is will be valuable for language developers to bring this to their languages both to increase popularity and open up new avenues for the developers to work with..

3. Literature Review

3.1.1 Reviews

Language & Documentations [20, 21, 22, 23]

Language developers provides documentations and specifications about languages, and hosts them freely on their dedicated developer pages. *Microsoft*, *Oracle*, *Apple* and *Google* have all provided and update their documentations, even allowing community contributions to create examples which display the language and newer features for every new release they put out.

Because the sites hold documentations provided by the developers, and considering that it is being actively maintained by the developers themselves, the information provided is considered reliable and therefore invaluable for our research.

TIOBE Index 2018 [7]

"TIOBE indexes and calculates a languages popularity based on skilled engineers world-wide, courses taught, third party vendors", as well as through popular search engines and the amount of hits each language gets. It provides an interesting overview of how widely and frequently a language is used, in this case that translates to popularity.

TIOBE were the first to start indexing programming languages back in 2003 and have continued to measure them monthly since then. It shows a good and easy to understand overview of the past decade and how languages have gone up and down in popularity. In the recent years others have begun to do the same thing but *TIOBE* remains most trusted one as its measurements have been done for over a decade and is deemed to be reliable due to its reputation and longevity.

A invaluable resource for our research due to the need to find and compare how languages have gained/lost popularity over the years, it gives you a easy to access database of just the necessary information to determine language popularity. However it is missing any sort of reasoning *why* a language is as popular as it is.

Most Popular and Influential Programming Languages of 2018 [8]

Ben Putano has made comparisons supported by the three major sources for language popularity, those being the above mentioned *TIOBE's* index, *GitHub's* 'Most Pull Requests', and *Indeed.com*'s job opening. Using these resources *Putano* has made a list of the most popular languages and attempted to argue for *why* they are so popular. A great source for our research as *Putano* lists different factors to why a language gains popularity, similar to the research we are to do on languages however *Putano* sticks to those that are at the top and does not compare those lower, to determine if these factors truly are the cause of its popularity.

Which programming languages are most popular (and what does that even mean)? [9]
David Gewirtz has made his own comparison based on several sources, amongst others, supported by *TIOBE*, *Gewirtz* compiles a cross-reference of popular languages and compares the results for accountability. *Gewirtz* also makes a definition of what popularity is and what it tells us, he comes to the conclusion that the definition of popularity is based on the publicly accessible data based on the language such as lines of code, tutorials available, job openings requiring it, forums posts mentioning it. *Gewirtz* definition of popularity matches well with both *Ben Putano's* and *TIOBE's* definition of popularity.

However *Gewirtz* cross-references shows us that *TIOBE's Index* doesn't fully agree with the other popularity sources, a bit of research shows that *TIOBE* uses a wider range of sources for its indexing than the others do.
Gewirtz analysis provides us with valuable cross-referencing that strengthens the case of *TIOBE's Index*, in addition it gives us overall idea of what can be considered popular and where the popularity comes from. *Gewirtz* does not go in-depth as to what a language provides that could be the reason for its popularity.

What coding skills do devs want to develop [10]
Hannah Yan Han does a dissection and representation of *Stack Overflows Developers' Survey*, showing a visual representation of which languages developers are using and which they want to learn in the future. *Han's* representation makes it clear and easy to tell that many of the developers wish to learn much of the non-popular languages, by looking into the nichés presented by *TechBeacon*, we can see a trend rising in niché languages. Further mentioned in the interview conducted by *Jason Pontin* we notice that the niché languages tend to cover and fix problems.
An invaluable representation of what languages developers want to learn combined with our research gives us a good understanding of why developers want these skills.

13 programming languages defining the future of coding [11]
TechBeacon makes a summary of the nichés/positive things that new languages have brought forth as well as some negative parts, it covers 13 languages that are on the rise. *TechBeacon* gives us a short description based on the language to motivate their rise and reason for appearing on the list. The list also gets supported by *TIOBE's Index* as the languages chosen show up on the *Index* and are on the rise.
The article provides an invaluable resource for our research as it gives us the exact 'main' focus of the language and why it is there.

The Problem with Programming [15]
Jason Pontin interviews *Bjarne Stroustrup*, the creator of C++, asking questions about the design of C++ amongst other, but our primary focus is on the question regarding what the problems with programming languages are as well as software. How the design choices of C++ has affected the language in both a positive and negative way. The interview provides a great support for *TechBeacon's* list as it shows that we are working towards fixing those problems.
This interview gives us a great resource as a direct opinion from a developer of a highly frequently and widely used language that has paved the way for many languages today.

3.1.2 Our Research

Our research intends to fill a gap in this topic, mainly *why* a language becomes popular, *what* exactly developers feel they are lacking, instead of grasping at the languages that exist, developers might want something that no one has thought of yet.
The research we do aims to give this information in a clearly displayed way.

3.1.3 Conclusion

The amount of available are great but there is a gap missing that our research intends to cover. Available resources cover popularity of a majority of programming languages, comparisons on popularities, nichés they provide.
Information about the problems we used to face in programming and how they are still relevant today.

Our research want to fill the gaps that are *why* languages gain this popularity, *what* developers want from the future, *how* they decide what languages to use, and *if* languages have improved overtime.

4. Research Method

RQ1. How do newer languages compare to their predecessors in regards to what they do better?
RQ2. What are the deciding factors for when choosing a programming language for a project?
RQ3. Why are some languages more popular than others?
RQ4. What is it developers expect and want from newer languages?

4.1 External data

Theoretical research was done by reviewing, and reading through literature that touches on subjects of pros of programming languages, comparisons of languages and general information about capability of a language and its limits. By focusing on research made by those in the relevant field and scientifically published papers, articles, or other online source on programming languages, we have found references ranging from empirical studies, shallow studies on languages and more in-depth ones *(RQ3)*. These are used to draw conclusions, and make predictions based on trends.

Empirical research that has been conducted is a combination of a survey sent to students of IT courses within Blekinge Tekniska Högskola as well as a variety of technical experts and programming enthusiasts in the area local to the university.

4.2 Survey data

The survey attempts to pull in general data and more detailed data based on what the developers answer to the diverse questions. Questions on the survey range from experience in the field, what type of developer, which languages are frequently used, what factors are in play when choosing a language to work with *(RQ2)*, and what is wanted from future languages *(RQ4)*.

4.3 Interview

Due to the difficulty in finding willing people within the relevant field only one interview was conducted. Using a series of questions based on our survey questions, but altered to promote a more in-depth answer, the interview was done one-on-one, in person, and recorded for later analysis. The interviewee is a professor holding a PhD in computer science with his current field of work being in compilers and language design. The interview touched mostly the same type of questions as the survey but is more aimed at the current state of languages, what they do well/bad, how/why some languages are considered the replacements for others *(RQ1)*, language design such as syntaxes and which features are considered good design in new and old languages *(RQ1)*, what direction we're going with languages, and if we have reached a point where we are releasing way too many for our own good.

4.4 Research methods

The research method we use is a mix of quantitative and qualitative methods, this is because the research questions we use required more than statistical data from surveys and other data sources to answer each of the research questions.

We utilize existing surveys which has a broader and diverse audience, as well as combining them with our own data that we have gathered through the use of surveys and interviews. We have found literatures that are used for areas where information requires more depth, and where statistical data is insufficient to conclude an answer. Most documentation about language design and characteristics were referenced from the languages' own documentation site, and some of them have allowed community contributions to further add content to any documentation to the site. Certain sites, such as *cppreference*, are community driven sites in which user contributions to the documentation are allowed by anyone.

This may seem unreliable, but with the option to review the history of each documentation, one can determine what information have been altered and what was the original. There have been use of articles in which they provide their own set of data to conclude with. Some of the data they use may not represent the data that was compiled and used for this research document, and one would remain neutral with the findings.

4.4.1 Portals

The portals used most frequently are as follows.

- http://bth.diva-portal.org/smash/search.jsf?dswid=2696
- https://books.google.es
- https://dl.acm.org
- https://link.springer.com/
- https://ieeexplore.ieee.org
- https://scholar.google.com

4.4.2 Keywords

The keywords used have been largely a combination of the following.

- programming language
- programming
- language
- comparison
- improvement
- history
- problems
- popularity
- developer
- syntax

When searching for material for specific languages, the language name has been added to the keywords.

5. Analysis and Results

In the following chapter we will introduce our analysis of each research question, what the results are and how they were acquired as well as a short conclusions that answers the question itself.

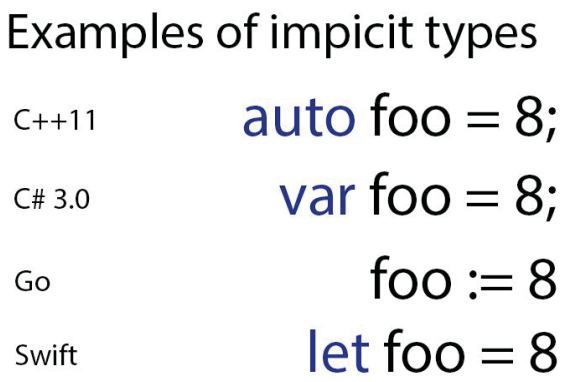
5.1 RQ1

5.1.1 The languages

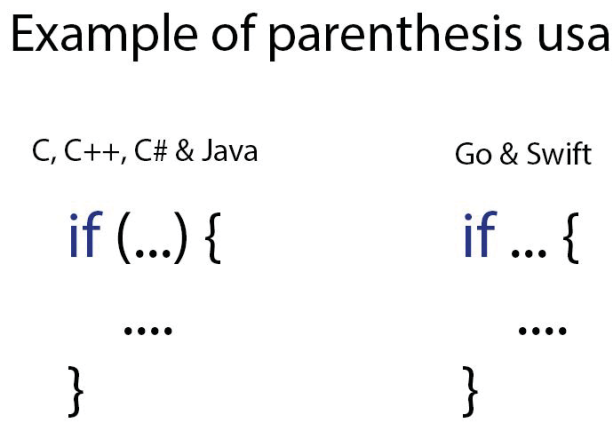
C++, Java, and C# are languages which are older than 10 years and are still widely used in software development. They were picked for this question because of their popularity and how similar they are to C. Modern languages that we picked to compare to the older languages are: Go and Swift. They were picked because they remain similar and influenced by C. The comparisons and improvements were looked based on the language itself. How it is typed, and what has been done to make the programmers work easier.

5.1.2 Small and noticeable changes

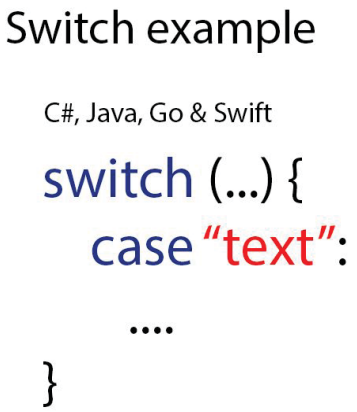
Modern languages have minimized the effort for the developer to explicitly state what type a variable should associate with. C++11, C#, and C# 3.0 already had this implicit type feature implemented using the following. C++11 uses the keyword *auto,* and C# uses *var* for implicit variable typing [1][2]. Both Go and Swift came with this feature implemented since day one.



Go and Swift has minimized the use of tokens and symbols to their syntax for it to make sense to the compiler. Such changes are the optional use of the semicolon, and statements which does not require the use of parentheses for expressions.



Although the semicolon remains optional, and perhaps useful if the intent is to combine several statements in one line, according to Go's documentation, the lexer performs automatic insertions so the grammar can determine when the next statement is [3].



Since C and C++, the *switch* statements limitation was that it could only evaluate results based on numbers only. However, with later versions of Java [4] and C# [16], the switch statement can match results on strings, making it a better approach compared to a series of else-then-if to create logic for every match. Go and Swift had their implementation of the *switch* statement support string based matches since release.

5.1.3 Memory management

Newer languages follows the approach to automate memory management for the developers, avoiding the need to explicitly state when memory should be released. This was already evident in older languages (and their runtime) such as Java [17] and C# [18]. Go and Swift have implemented their ways of memory management differently.

Swift uses Automatic Reference Counting (ARC) to decide when allocated memory gets released back to the operating system [5]. Every object has a reference counter in which it decreases or increases depending on how many variables uses that resource. Once the reference count reaches zero, the garbage collector will deallocate the object and return the memory to the operating system.

Go uses a garbage collector for memory management [6]. It comes packaged natively as a part of Go's runtime and runs concurrently alongside with it [19]. This approach is very similar to that of Java and C#, only that Go is natively compiled while Java and C# runs as virtual machines. Go emphasises heavily the performance of their garbage collector and aims to improve it through the use of algorithms to properly determine what the best course of action is to take during resource deallocation.

5.5.4 Conclusion

Modern languages have improved in the sense that they do not force the programmer to be very specific with what they want to achieve, and leaves most of the hard work for the compilers to deal with. Modern, as well as some old languages continue to make memory management automatic for the developer. This allows for the runtime to better determine when resources should be freed without causing memory leaks or undefined behaviour should resources accidently be dereferenced while it has been freed already.

The newer C-family of languages continues to make improvements while keeping the syntax of C, closely followed by their predecessors continuous upgrades to improve themselves. The changes has stripped away certain tokens to allow for less typing and making the source code less cluttered with optional semicolons and parentheses, as well as implicit types on declared variables has eased the way code is written by leaving much of the work for the compiler to figure out. The limitations C had has now been improved with that of the *switch* statement, allowing for matches on string types thus improved the way strings are handled, making the *else-then-if* method obsolete, and at the same time allows for less clutter which makes the source code more readable.

5.2 RQ2

5.2.1 Survey results

Based on the results from the survey that was conducted and sent out through Blekinge Tekniska Högkolans public mailing system, analysing the deciding factors of choosing a language for a project, lead to the answers being divided up into four categories:

- Experience (11): a language that a developer is already accustomed to and picked it for that very reason.
- Requirements (13): language picked to fit the specified requirements.
- Limitations (2): when certain projects only work in certain environments, a limited set of tools are available.
- Performance (6): language was picked mainly for performance reasons.

Total answers received from the survey was 37 out of 41 participants. 5 answers were ambiguous and could not be properly determined, which we could only resort to the remaining 32 answers.The results were judged by the nature of the answer if they could be determined. Four categories were chosen for the focus on the answers because they were the most frequent ones and once counted we could determine what the factors were.

Out of all the valid responses received on that question (see annex 9.1.3), the biggest deciding factor was based on the project requirements.
Decisions based on the project's limitation was the least deciding factor.
Choosing a language that the developers were already accustomed to were the second most picked answer. If the project was not based on any requirements that they felt were to be enforced, then the language they prefered would be more suited for their project, because of the previous knowledge would provide with productivity.
Performance and limitations were the least picked reason that language the project would go for closely followed by simplicity.
Performance was the biggest reason out of the two, due to it being the most needed attribute for applications which requires heavy tasks to be processed.
Limitations, in the sense of available tools for the project, were determined to be the second least picked out of all the categories. With simplicity being the least picked reason and as such omitted from the results as it was determined too insignificant compared to the other choices.

5.2.2 The interview

In the interview, when asked about what the deciding factors are for the interviewee when choosing a programming language for a project, they explained that it is mostly based on what they are already familiar with, and the productivity bonus that comes with that decision. There was also a second reason, that if there was a language they were not experienced with, but interested in, it would be used instead, for the purpose of learning a new programming language.

"Personally, for my projects, it is something I know really well. Quite simply because it is a productivity bonus. Things I've been working in for 5 to 10 years, I will write code a lot faster in them than something that I pick up. Or, is it a language that I don't know, but which looks interesting." (see annex 9.2.1)

They also mentioned the pitfalls of just going for a language that someone is already accustomed to that may not be the best fit for a specific project, and what led to their decision to rewrite their website's back-end. They took this opportunity to learn a new language as they went on with the rewrite.

"So, previously, that was written in Python, and that's an example of a former case: it was written in Python not because it's good for running [a web-backend], but just because it's something I knew really well, so it was very productive. The new version has been written in Go, which is a language I did not know at all, and I wanted an excuse to specifically learn it..." (see annex 9.2.1)

5.2.3 Conclusion

A conclusion can be drawn from the results that it mostly is from personal preference and what they think would be the most appropriate for their project, but also what the project requires. It weighs close to both ways: either based on the project's requirements, or personal preference where the developer feels most comfortable and experience in.
In order to boost the productivity for the sake of quick prototyping and less problems when dealing with semantic errors, the likely choice would be to pick a language in which the developer is already familiar with.

Projects with requirements in which the language has already been decided upon may be really there to provide a more suitable approach towards a specific problem, to remain stable and efficient in the long run. Another reason going with requirements could also be projects which are already in development and where community contributions are available, means that the language was already been decided prior to the communities' involvement in the project.

Limitations on choice of languages became the least picked, our own assumption is that it is possibly because certain fields involving legacy projects, or poorly supported tools are fading away or being replaced by newer software and better tools.

5.3 RQ3

5.3.1 Analysis

Based on the empirical studies *(*see annex 9.1.3 & *Hannah Yan Han, 2017)*, we have made an analysis and conclusions on language popularity drawn upon what languages the responder 'likes' and 'what characteristics draws them to a language'. These are compared using a source that is based on language popularity which calculates how popular a language is through the use of a reliable rating system made by TIOBE [7].

"*The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings.*"
(Ben Putano, 2018)

Using the popular search engines, TIOBE[7], calculates most used languages based on projects on Github, SourceForge, and freecode; books sold that the teach them, job advertisements that mention them, Google Trends, and based on number of times it is mentioned in web searches. This allows them to determine the popularity of a language and allows us to determine what makes them popular by looking into what the languages can provide for a developer and how high on the chart they are[8][10].

5.3.2 Result

Our empirical study data (see annex 9.1.3) points towards a few obvious points:

- Syntax
- Ease of use
- Functionality
- Community
- Freedom

These results compared to the top ranking languages [7], show that a language that fit at least 3 of these points end up high on the popularity rating. The more the language caters to the developers wants the more popular it becomes, while those that fall behind often lack these points. By cross checking with the popularity ratings, reviewing each language on the top 20 to these 5 points, and comparing it to our survey data where responders have said which language they like, the conclusion is that most languages that are within the top 20 and fit at least 3 of these popularity points (with mostly aimed at community) are the ones that reach high popularity.

5.3.3 Conclusion

Either it has a aesthetically beautiful syntax, is easy to use, has a large/available community [10], gives the developers freedom to mess up and handle otherwise 'unsafe' stuff, or provides functionality that is isn't widely available.
The most important point being community which includes tutorials, documentation, and general availability. A language that does not have this point falls far down on popularity and often gets overlooked.

5.4 RQ4

5.4.1 Analysis

By cross checking the nichés [11] of popular languages [7][8], and our empirical study (see annex 9.1.3) on the wants of developers, we have found a pattern with languages and certain factors that make them attractive. We have found that consistently the languages that developers want to learn in the future often have a particular niché that they want to use [11]. By looking at the popularity of languages [10] over the last 10 years [7], we notice that the niché languages have slowly been making their ways higher on the rankings. We assume this is due to the niché they provide cause a large amount of developers to use the language in hopes of it being the 'next best thing'.

Those are boiled down to:

- Simplicity
- Safety
- Compatibility
- Functionality

5.4.2 Result

The empirical study conducted by us shows that developers want more simplicity, compatibility, more functionality, and safety.

Simplicity has been stated to be in the sense of better syntaxes and quicker ways to write code, as code gets more complex and intricate by the day, it also gets more and more difficult to write code that isn't a mess in both looks and function. Developers have been asking for cleaner, better syntaxes to make code look less messy and quicker ways of writing more complex code as a result of simpler syntax.

Compatibility with cross platform support and backwards compatibility. Languages that can create code that works on all platforms are highly sought after and often end up high on the rankings of popularity.

Functionality is a major want with languages as not all of them share the same functionality, some do other things that aren't as common but frequently used and implement them into their base functionality, developers often ask for these things to become a standard.

Safety has been stated as both a major want and a major dislike, developers want safety in the sense that their code can not be broken into, but they dislike the safety done to keep them from making errors. As such this factor can be split into two internal ones. Security and freedom, as it better describes what the developers want.

5.4.3 Conclusion

Conclusion leads us to believe that a majority of developers agree to what they wish to see in the future of programming languages, our analysis gives us a base idea of what the current programming languages are lacking and what we should do to in the future to ensure that the needs of developers are taken care of.

The future needs of developers are to get better languages which provide improved syntaxes, better compatibility, more freedom for the developer but more safety of the code, and a larger variety of functionality.

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Internet: www.bth.se
Phone: +46 455 38 50
Fax: +46 455 38 50 57

6. Conclusion

Automatic memory management allows the programmer to allocate necessary data without needing to explicitly state when the resources should be returned back to the operating system, decreasing the chances memory leaks will occur in the software. The memory manager are implemented differently depending on what the language was designed for. Languages such as Go and Swift uses different implementations of garbage collection to handle the memory.

Languages have become less pedantic and leaves many tokens to remain optional. In Go and Swift, semicolons, which have remained as a standard way to terminate a statements, now rely on line endings to determine when a statement would end. Other design efforts in that area are seen with how expressions are determined in Go and Swift; statements does not require parenthesis to determine where an expression is in the language. Both of these languages came with support for string handling with the *switch* statement, making it a more attractive choice over the *else-if-then* method. However, even older languages, like Java and C# have this feature supported, although in later version.

Type systems have been improved to allow for implicit typing, leaving the compiler to decide what type a variable should associate with based on the value it receives. Languages which still are being updated have seen the same improvement, and not just modern languages.

How languages were decided for projects are heavily determined either by the projects requirements, or what the programmer is already familiar with. Determining the language based on what is best for the project to match the requirements but if the requirements are not that specific, the language often ends up being chosen based on the developers past experience.

How does a language become popular then? Survey data points clearly towards a few points.

- Syntax that is aesthetically beautiful.
- Easily used where you can quickly get into the language and start developing.
- A large community with good documentation, and functionality.
- Complex functionality made possible.
- Freedom to go past the common safety nets of high level languages.

Languages that provide most of these points are often seen high on the ratings for popularity and survive the longest, while those that don't have focus on the few points mentioned often get forgotten or 'left behind'.

Studies shows that developers want more safety, simplicity, functionality, and compatibility in the future for languages. Developers wanting more compatibility with cross platforms, simplicity with easier to write syntaxes, nicer looking code structure, more complex functionality built in at the core, and safety against bad code.

7. Future Work

To create a follow up and a more throughout continuation of the study conducted in this thesis, several more areas, as well as covering the existing areas in more depth which would greatly improve the research more.

Existing survey results exists in several community hubs which has a more wider audience and more data to use. This could be more effective to use and strengthen the conclusions. If a survey would be used to gather more data, consider targeting companies or representatives to also include how companies think about the subjects.

8. References

1. cppreference.com community, *Auto* (2017). Fetched 18th may, 2018 from http://en.cppreference.com/w/cpp/language/auto
2. Microsoft, *Overview of C# 3.0* (2007). Fetched 18th may, 2018 from https://msdn.microsoft.com/en-us/library/bb308966.aspx#csharp3.0overview_topic2
3. Go team, *Effective Go* (2018). Fetched 7th may, 2018 from https://golang.org/doc/effective_go.html#semicolons
4. Oracle, *Strings in switch Statements*. Fetched 22th of August from https://docs.oracle.com/javase/8/docs/technotes/guides/language/strings-switch.html
5. Apple, *Automatic Reference Counting* (2018). Fetched 16th may, 2018 from https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html
6. Go team, *Frequently Asked Questions* (2012). Fetched 9th may, 2018 from https://golang.org/doc/faq#garbage_collection
7. TIOBE, *TIOBE Index*. Fetched 8th may, 2018 from https://www.tiobe.com/tiobe-index/
8. Ben Putano, *Most Popular and Influential Programming Languages of 2018*, December 18, 2017. Fetched 8th may, 2018 from https://stackify.com/popular-programming-languages-2018/
9. David Gewirtz, *Which programming languages are most popular (and what does that even mean)?*, October 4, 2017. Fetched 8th may, 2018 from https://www.zdnet.com/article/which-programming-languages-are-most-popular-and-what-does-that-even-mean/
10. Hannah Yan Han, *What coding skills do devs want to develop*, July 8, 2017. Fetched 8th may, 2018 from https://towardsdatascience.com/what-coding-skills-do-devs-want-to-develop-a952ee620312
11. Peter Wayner, *13 programming languages defining the future of coding*, November 10, 2015. August 21, 2018. Fetched 21th of August from https://techbeacon.com/13-programming-languages-defining-future-coding
12. TechTerms, *Programming Language*, September 23, 2011. Fetched 21th of August from https://techterms.com/definition/programming_language
13. Wolfgang K. Giloi, *Konrad Zuse's Plankalkül: The First High-Level, "non von Neumann" Programming Language*, 1997. Fetched 21th of August from http://thecorememory.com/Zuse_Plan_Kalkul.pdf
14. Richard L. Wexelblat, *History of programming languages*, May 27, 2014. Fetched 21th of August from https://books.google.es/books?id=Hy-jBQAAQBAJ&lpg=PP1&ots=jl-J6nuYU2
15. Jason Pontin, *The problem with programming*, November 28, 2006. Fetch 21th of August from https://www.technologyreview.com/s/406923/the-problem-with-programming/
16. Genevieve Warren, *switch (C# reference)*, August 14, 2018. Fetched 22th of August from https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/switch#the-match-expression
17. Michael J Williams, *Java Garbage Collection Basics*. Fetched 22th of August from https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

18. Ron Petrusha, *Garbage Collection*, March 30, 2017. Fetched 22th of August from https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/
19. Richard L. Hudson, *Getting to Go: The Journey of Go's Garbage Collector*, July 12, 2018. Fetched 22th of August from https://blog.golang.org/ismmkeynote
20. Go team, *Documentation - The Go Programming Language*. Fetched 2th September, 2018 from https://golang.org/doc/
21. Apple, *Swift - Resources - Apple Developer*. Fetched 2th of September, 2018 from https://developer.apple.com/swift/resources/
22. Microsoft, *.NET Documentation | Microsoft Docs*. Fetched 2th of September, 2018 from https://docs.microsoft.com/en-us/dotnet/index
23. Oracle, *Java Documentation - Get Started.* Fetched 2th of September, 2018 from https://docs.oracle.com/en/java/

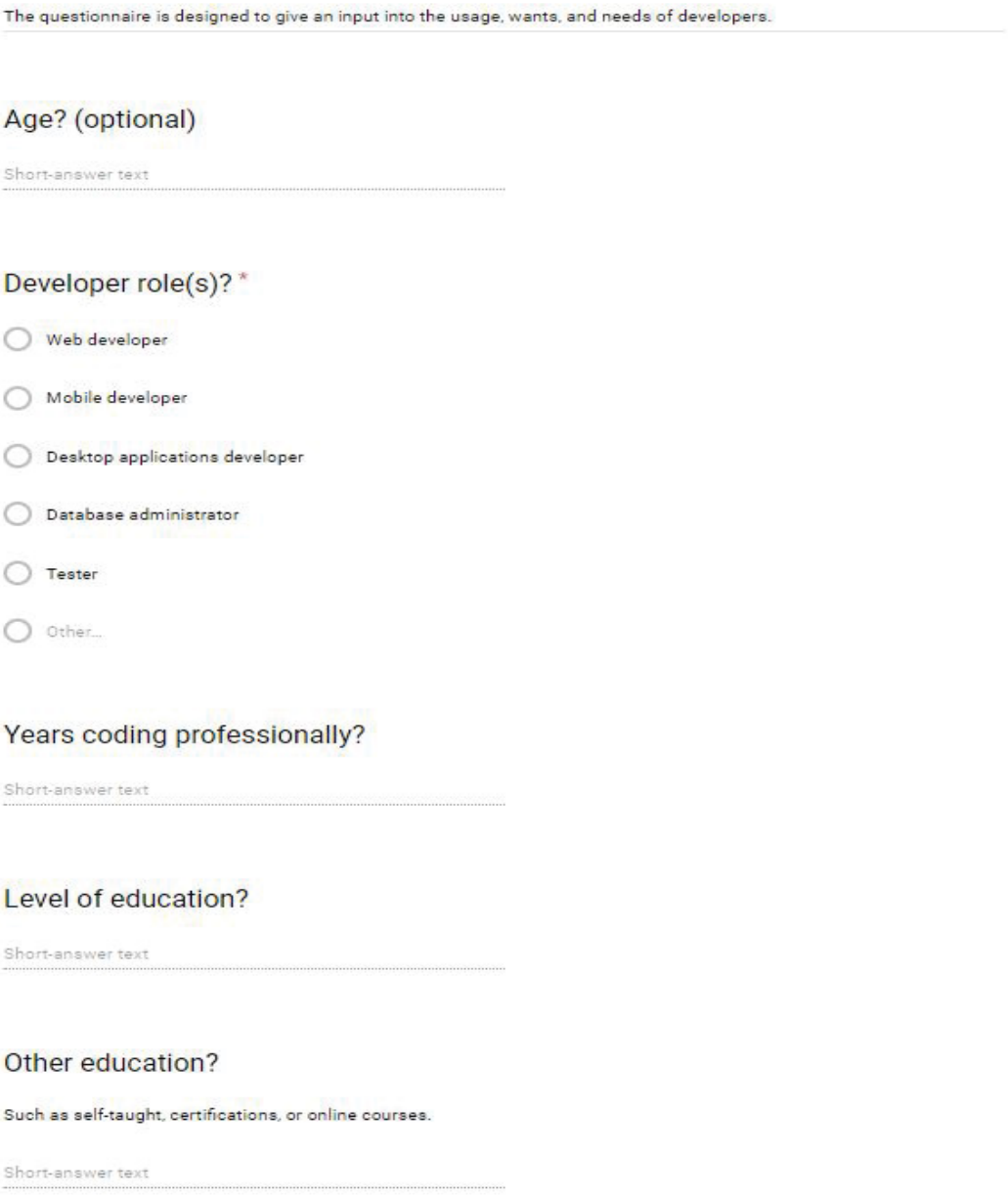
9. Annexes

9.1 Survey

9.1.1 Author notes

Few questions were omitted from the survey when collecting data as a discussion between the authors revealed that the questions were either poorly formulated, being irrelevant, or provided no value to the research.

9.1.2 Survey design


The questionnaire is designed to give an input into the usage, wants, and needs of developers.

Age? (optional)

Short-answer text

Developer role(s)? *

- Web developer
- Mobile developer
- Desktop applications developer
- Database administrator
- Tester
- Other...

Years coding professionally?

Short-answer text

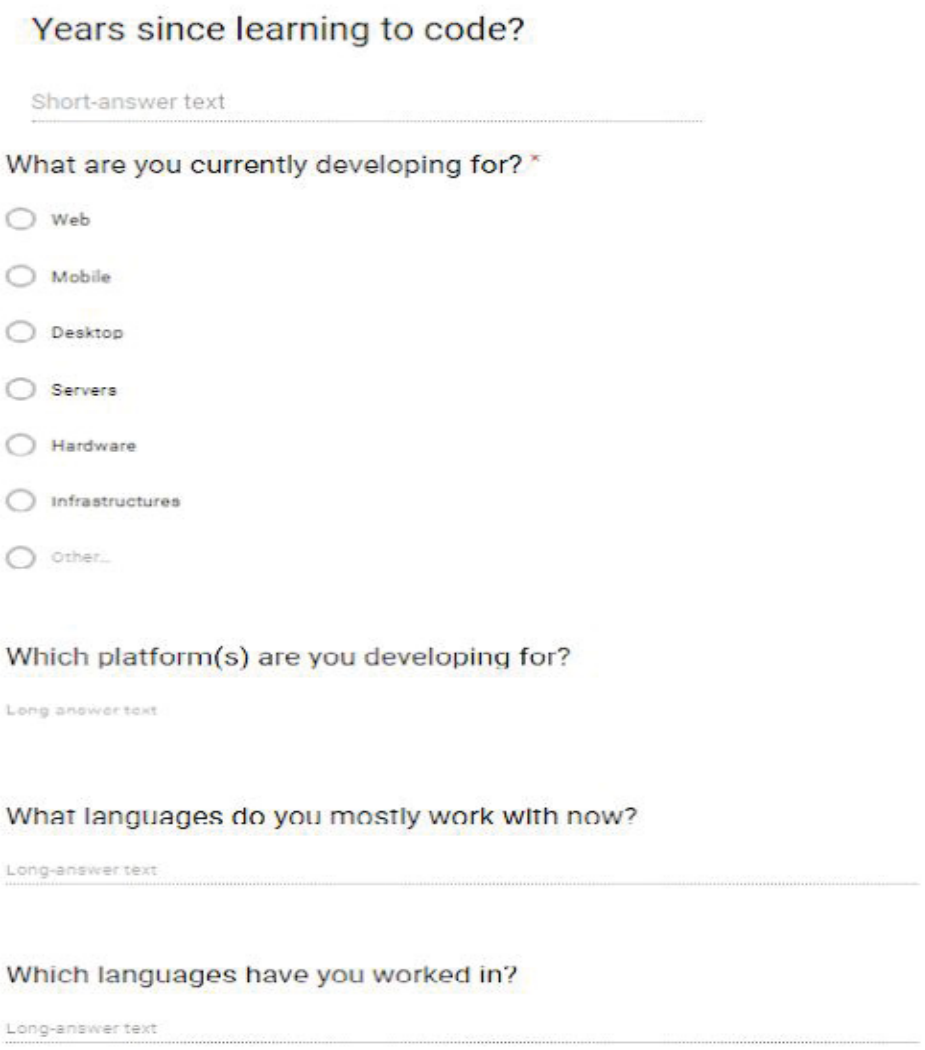
Level of education?

Short-answer text

Other education?

Such as self-taught, certifications, or online courses.

Short-answer text


Years since learning to code?
Short-answer text
What are you currently developing for? *
Web
Mobile
Desktop
Servers
Hardware
Infrastructures
Other...
Which platform(s) are you developing for?
Long answer text
What languages do you mostly work with now?
Long-answer text
Which languages have you worked in?
Long-answer text

What would you like to develop for?

- Web
- Mobile
- Desktop
- Servers
- Hardware
- Infrastructure
- Other...

What language would you like to learn/develop in?

Long-answer text

What is your favourite language?

Long-answer text

What feature, or change would you like to see in future languages?

Long-answer text

What would you change, remove, or add to a language of your choice? (Specify which)

Long-answer text

What characteristic draws you to a language?

Long-answer text

9.1.3 Survey data

What languages do you mostly work with now?	Which languages have you worked in?
C and Matlab	C and matlab
Linux	C,Python,Perl,Bash
html,css,php	c, c++, python, node.js
Python, C and Java	Swift
PHP, JavaScript, Java	PHP, JavaScript, Java, C#
COBOL	EasyTrieve, Java, C#/.NET, Python, Assembler, C/C++ and a whole lot others.
C#	C#
Javascript, CSS, HTML5	Most of web languages
Javascript	Javascript php c c++
Java	Java, C#, C++, C, Python, Javascript, Assembler
Java, Javascript	Java, Javascript, C#, Python, Bash (not exactly WORK, but progams and projects)
Objective-C, Swift, C# and Java	Python, JavaScript, Java, C#, Objective-C and Swift
Java/C#	Java, Python, PHP, C#, Javascript, TypeScript,
C++	Java,JavaScript,php , python, c++
Python, C#, C, Haskell, Assembler (ARM & AVR), Javascript, Lisp	C, C++, Fortran, Erlang, Python, Lisp, Scheme, Tcl, Awk, Haskell, Coq, Agda, C#, Java, Scala, Assembler (Z80,M68k,x86,ARM,AVR), Javascript, Rust
JavaScript, PHP	Java, JavaScript, PHP, MySQL
c++	c++, sql, javascript, html, css
Bash/Python	Bash/Python/ruby
C, C++, C#	C#
Swift	C++, python, java, C
HTML, CSS, JavaScript	HTML, CSS, JavaScript, PHP, VBScript, C#.NET, VB.NET, Python, Java, Pascal, QuickBASIC, Assembler (x86)
C, C++ and Python	C, C++, Python, x86 assembly, MIPS assembly, PHP, JavaScript, Lua
Python, C++, C#, C	HTML, ASP.net, vb.net, Basic, Forth, Assembly, Java, Lua, Bash
C++	C/C++, js
JavaScript	JavaScript and Java
JavaScript	Scala, JS, C++, Java, TypeScript
C++	C, C++, Python, PHP, Lua
python c/c++	java, c/c++, shell scripting, python
JavaScript	Java
C, java, Python	
HTML, PHP	C
Php, C#	Php, C#, Android
Java,R,SQL, AndroidSDK,VuforiaSDK	Java,R,SQL, AndroidSDK,VuforiaSDK
Java and Python	C, C++, assembly, Java, C#, PHP, Python, Javascript
C#	C# and C++

html and PHP	HTML, CSS, SQL, PHP, C#, C++, C, JavaScript
Angular js and Java	Php java
Php python JavaScript bash	Som ovan
C, C++, Lua, PHP, MySQL	C, C++, Java, Lua, Python, MySQL, PHP, HTML, CSS, C#
C++, assembler, R, C	Java, HTML, C#, Shell
Java and C++	Java, C++, C#, TCL, Shell Scripting

What language would you like to learn/develop in?	**What is your favourite language?**
C++	C
	C
perl	html
Python	Python
Java	Java
C	Haskell
C++	
Javascript, Java	Javascript
Ocaml, reasonml, haskell, elm	Elm, ocaml
Java	Java
1. Javascript (React, etc), 2. C#, 3. Java, 4. Python	Javascript
Java	JavaScript and Objective-C
Java/C#	C#
All of them	C++
Idris	Haskell
I would like to learn frameworks like ReactJS, AngularJS and VueJS	JavaScript
swift	c++
C/C++/Haskell	Ruby
Vhdl, pascal	Vhdl
Javascript	C++
JavaScript, C#	JavaScript
C++	C++
C++, C#, C	C++
C++	C++
C#	JavaScript
JavaScript, GO, Scala	Js
C++, Python	C++
javascript, react js...	python currently
Java, C++, JavaScript	C++
PHP, JS	C
Python	Php
java	Jav

C++, Java, Python	Depends on which type of project I am working on. Anyway, Java for most of the cases.
C#, C++ and HTML	Mostly worked in C# so I guess that one
C++	C++ or HTML
Web	Java
C#	Javascript
I would like to further hone my skills in C++, PHP and MySQL	C++
R, Python	C++
R and Python	Java

How do you decide what language to use for a project?
Legacy [4]
use case and simplicity in an application *
based on requirements [3]
Easiest *
Knowledge [1]
There is not a lot of choice on the IBM mainframe. So usually if I want others to be able to read my code it's in COBOL, if I want something easy to write it's EasyTrieve and if I want to optimize performance it's Assembler. I don't have many more languages available to me and I don't develop on unix/Microsoft systems currently. [4] [2]
2
Secure, Efficient, Easy to develop with [1]
I don't decide *
The most common one used for the subject [3]
I depends on what I am going to develop (purpuse of application, platform, target user group, etc). When developing for e.g. the web I would choose ReactJS on frontend and C# on backend. [3]
Languages used in similar projects in the past, advice from co-workers, Google [1]
Depends on platform I am developing for and depends what I am developing. If it is a Windows server then I will use C#. If it is Linux, then I can go with Java or PHP. [3]
Whatever I know will work for the given project depending on current infrastructure and what is used in the industry [3]
Project requirements, runtime dependencies, language expressivity. [3]
Based on requurements of the project like validation, login and signup [3]
Vad jag känner mig mest bekväm med [1]
Weather or not its natively supported on the platform *
Whats compatible with our robot [3]
Platform availability, technical needs and limitations [3]
Performance and prior experience [2]
When performance is of no importance, or for quick prototyping I use Python or C# on Windows. Otherwise I mostly use C++. [1] [2]
Performance is critical for game development [2]
I choose the one I will work on mainly and that I know the most. Which is JavaScript [1]
Knowledge and time [1]

For short projects, Python. For longer, more complicated projects, C++. *
check usability and performance [2]
Application specific [3]
Based on experience and professiony of the team members [1]
by analyzing how application work and keep it efficient *
C++ for games and high performance, Java as an all-rounder, Python for networking. [2] [1]
Based on what preferences, experiences and wants of the group I'm working with [1]
Depends? if i do websites i basicly uses what i feel is necessary *
By necessity [3]
Php mot Web python mot styrning *
If it's for a PC application I always go with C++, if it's for scripting within a game it's Lua. [1]
Context and availability of library [3]
Its based on problem statement and while choosing which platform to choose to solve the problem [3]

Note: The additional numbering or asterix is added by the authors when checking answers for RQ2.

Summary of what we determined: 37 answers, 5 ambiguous answers, 32 valid answers
Marked with * means ambiguous, [x] means what category the answer is in
1 - [accustomed/experience]: 11
2 - [performance reasons]: 6
3 - [Requirement]: 13
4 - [Limitations]: 2

What feature, or change would you like to see in future languages?	What would you change, remove, or add to a language of your choice? (Specify which)
Support for safety, such as RUST	Tool support
easy to code	more efficient
Easier to send routes	C string handeling
Delegates, Properties	Remove primitives from Java
More abstract code and easier-to-write code. The inclusion of lambda statements and streams for Java 8 was great but C# already had LINQ for years - I am curious to see what is the next step.	I would like easier string handling in COBOL please.
Make them even simplier and more secure	Make it even simplier
ML-like syntax, functional first	Remove parenthesis and other visual noise
Automatic multicore capabilities	Fix nullpointerexceptions
More stable libraries for e.g. network communication (UDP, TCP) and easier-to-implement access management (login features). More structured and open APIs among existing services.	
Don’t know	Don’t know
More type safety	Type safety
More expressive type systems without necessarily taking the step into dependent types.	I'd remove/phase out partial functions from Haskell. Deprecate them in the standard library and make the compiler reject them by default.
I am not that experienced programmer to answer this question	Depends, someone like to use JQuery and some others like to use DOM. I am trying to use vanilla JS
vet ej	Inget speciellt
Accesability and development speed	native support on more systems
-	An itelligent Compiler that actually is capable of giving solution possibilities, that worked for other people, kinda like a neutonal net
	Bättre pekarhantering i swift
Composability by design	Remove fake classes from JavaScript!
Rule matching	Add rule matching to C++, add proper multithreading to Python (no global lock)
Better control over memory (No garbage collection)	Rule Matching, More metaprogramming and better support for compile-time functionallity
Don't know	Don't know
More functional programming	Merge TypeScript and JavaScript :)
more flexible and compatible	not sure...not fully exploit them yet
Don't know	Don't know

Supporting for multiple platforms	Nothing
keeping them more object oriented	
Different types of "linked list" need to be more easy to manipulate.	I do not want to change anything in a language but I want to add libraries for manipulating linked list.
Easier to understand and get ahold of at the start	Nothing, really
Web languages	
I don't really see the languages as the problem, rather the programmers. It's us who need to learn how to write faster and more beautiful/readable code.	I'd like class handling in Lua, better than the "class" handling that is now.
nothing	nothing

What characteristic draws you to a language?
Syntax
easy to learn
Easy
Type-safety, OOP
Easy writability
Same as in How I decide language for project
Estetics
Usability and readability
Good and simple documentation, having a community and forum for questions, easy-to-set-up development environment.
Their beauty
Community, Docs, syntax and ability to get a picture in the head on „What could be made in such language”
Control
Conceptual consistency. A language needs to have a few central ideas that it makes no or few compromises on.
Mainly methods and complexity
Komplexiteten då det är en utmaning
Development speed and ease of use in the syntax
Pekare
Freedom, lack of verbosity
Performance, ability to work close to the hardware. No "safety" bullshit restrictions where the language treats you like an idiot and thinks pointers are dangerous.
Full control of the memory
Hard typed variables, unmanaged memory
Simple dynamic typed that is flexible
Easy learning
A good structure and a logical naming system
assignment requirement
Strong typing

Ease of use
Good structure, meaningful libraries names and the market demand
Syntax and Compiling abilities
performance, can develop programs with less lines of code (for example: not assembly), flexibility (runs in many platforms and machines) and rich libraries.
What engine uses it at the moment
what possibillities the lanuguage has to offer
Testing skills
Enkelhet punktnotation konsekvent
The ability to write fast, lean and beautiful object oriented code.
usefulness

9.2 Interview questions and responses

Place: Blekinge Institute of Technology (BTH)
Date: May 3rd, 2018 (10:30-11:00)
Recorded for future usage with permission of the interviewee.

9.2.1 What are deciding factors when you choose a language to work with?

Personally, for my projects, it is something I know really well. Quite simply because it is a productivity bonus; things I've been working in for 5 to 10 years, I will write code a lot faster in them than something that I pick up. Or, is it a language that I don't know, but which looks interesting. So for example: in the past year, I've been rewriting the web back-end that drives the mechani site (a teaching tool made by the interviewee). So previously that was written in Python, and that's an example of a former case: it was written in Python not because it is a good for running, but just because it's something I knew really well, so it was very productive. The new version has been written in Go, which is a language I did not know at all, and I wanted an excuse to specifically learn it, and I always found that having something specific to implement works as a driving motivation for learning a new language because it gives you specific things to go off and look at 'how do I implement this particular kind of thing', or, 'what's the right idiom' to make this fit in a language. So as an answer to your question it's either a language which I know really, really well, or one that I don't know at all.

9.2.2 What is lacking in the languages that we work with today?

Absolutely everything. I would say that we still… we're using things that are quite primitive. The biggest impact that I see day-to-day is the lack of native support of packaging and tooling. So, what is a good example?... Well if I pick on Javascript, because it is a good example of a language to pick on for many reasons. The Javascript eco-system is an absolute mess; there are a variety of package managers for Javascript, all of which that have their own package repos, their own hierarchies, their own tooling for accessing them and none if it is exposed in the language. So, when you write Javascript code, you assume you have a bunch of base source files as your library. There's nothing in the language that actually integrates the package managers to check they're up-to-date; that they've been audited; that they're actually the version the author wrote; there's no checksumming; there are no signatures. There was a massive catastrophe, I think it was last year, where a very popular set of packages on npm (Node Package Manager) were pulled because the author decided he wasn't going to do it anymore, but those bits of Javascript integrated into millions of projects and somebody name-squatted the author as soon as he closed his account, there were no signatures; there was no cryptography to make sure that the code that was pushed out from the repos (repository) were actually designed by him [the author], and so they injected fake bunch of libraries into hundreds and thousands of web servers and compromised a whole bunch of stuff. It was a huge fiasco. You could see the same issues in NodeJS and anything else that is one these dynamically binding libraries that has no integrated package managers in the language itself, but relies of some kind of external tool which just happens to put the right files in the right place in the file system so that the runtime works. That's probably the biggest issue that we have in software development today - tracking changes and where they have come from, tracking dependencies, and managing the risks in developing software.

9.2.3 Is there a language that does things right that others do not, or should be able to?

Yes, I think most languages do something right. In fact, I would think I would be hard-pushed to think of a language that does everything wrong. Every language has some strength, typically the research that lead to that language was investigated whether or not that particular aspect of it would be useful. I mean, even if I pick on Java, because I'm not a big fan of Java, the original experiment in Java was

'Can we write a language which works reasonably well in a platform independent way'. It was largely successful in that sense, it fails in a whole bunch of other senses, but if you write Java code it does mostly work on any platform you would put it on in the same way. The only exception is there is to do with embedded software, games, and Android. And I think John Carmack has written a lot of stuff on exactly where platform portability breaks in Java, but it tends to hit that one goal really well in most of its deployed environments.

If you pick another language, say, Python, the original goal Guido (Guido Van Rossum, creator of Python) was working on was 'how do you write a language which is easy to learn because it has one well documented idiom to solve a particular problem', and again, that is something that Python has hit really strongly. You can see in the example of Python that it's not just something that's encoded in a structure of a language, it's also something about the community around that language, and the ecosystem around the language actually provide. There's a lot of social process that works around Python development: they've got their own RSC process for trying to work out how features will interact, or what's the best way to put things in, [and] how a program will use them.

If you look at C, then it works particularly well of having a very cut down, simplistic, low-level (machine?) that the programmer can understand. Fails in a lot of ways that have to do with robustness of the code, platform independence, and memory management and many of the other headaches that we've been chasing for decades. I can't think of a good reason to use C++ though, so that could be the one example of a language that does not seem to have a defining feature which works. I think the original rationale was 'let's make a little bit more complex than C. So most of mainstream languages that are in use tend to solve at least one problem very well, and then have different levels of success on anything else.

9.2.4 Is there things you would like to see in future languages (changes, additions, etc.)?

I think what I'd like to see in future languages is an integrated view of version control. I have to say up-front: I don't have an idéa for how to make this work, it's just something that I would like to see. I'd like to see a language that has first-class support for the history of the code that's running, and also first-class native support for changes in the code, and that should also pull in things like packaging and dependencies, where the different libraries are coming from, what versions those libraries are at. All of the stuff that we currently have to do, like outside the compiler, in the ecosystem around the code, I would like to see languages that actually develop some way of reasoning about that so that a programmer can manipulate it directly. I don't know if that'll happen in the near future, don't really know if someone is working on that particular angle. It would be cool, though.

9.2.5 In what direction are we heading when it comes to language design?

That's a very tough question to answer. So (Popul?) is the premier conference in academia for investigating high-level principles of languages and analysing how languages are designed, and they kind of poll what is the state of the field on a regular basis every year or two. They have a panel [where] they draw in good answers from a lot of very good academics, and I wouldn't hope to condense down, like even the last time they did that was the state of the art of in to anything like a cohesive answer. I think the biggest change that I've seen in the past five or ten years about the direction the language design is heading is that it tends to be more interactive than it used to be. So there was once the case that somebody would spend a lot of time, kind of like, working on version one of a language, they would have a really good idéa of what that language was for, what kind of things they wanted to do try out and which kinds of idioms, which problems it would work well on. They would package that all up, and then they would release it in some way and say to the world 'Ok, come and play with my new language and show me what it can do', and then you would have some kind of interactive process that would go on after that first release. I think what I've seen over the past 5 or

10 years that it has kind of shifted so that more of that design work, about what is the language for, what the set of problems that the language will make easy to solve is being done collectively, there's more community building that happens before the language kind of reaches that milestone of its big release. I'd say that's one of the biggest trends that's changing, and it kind of make sense as the industry, that depends on IT and depends on computer science matures, we are tending to address things that we could do ten or twenty years ago, and as the scow gets bigger, you need more people, and if you need more people, there has been kind of a collective agreement that's good to get them into the process nice and early. I'd actually point to Go (Golang) as a really good example of that, so the original sketch of the language that Rob Pike made was heavily revised based on feedback and kind of exploded into a community very early in the design process. There are also really good example to point to for how you build an inclusive community around language design, and pull in a lot of different views in order to shape it. Too early to say that's worked or not; GO is quite a young language, but hopefully five or ten years from now on we would be able to work out what the impact of those decisions that were made early in the design process.

9.2.6 are newer languages becoming superfluous?

I wouldn't say we've reached that point yet; the languages that are on the horizon at the moment, things like Rust, looks quite interesting. Ownership of memory has been a big issue in programming for a long time, it'll take a while to see how much of that they've hit. Go is becoming increasingly popular, there's some very neat idéas in Go about data types and how you represent things. Some very strange idéas about how you structure the code which go in a very different direction to other mainstream languages at the moment. There are two separate issues that are hit by two of the languages that are becoming popular at the moment, I would say that's far from complete; (and?) they haven't covered all of the issues in language design. I think we'll know when we've reached the point when future languages are superfluous because stack overflow (Website) would disappear, so as long as we still got stack overflow, it probably indicates that we've not designed a language that works well enough that we can get by without any secondary social mechanisms to help people learn it.

9.2.7 Is there an area where limitations force us to design a new language for it?

There are some areas that spring to mind, there's some initial research work in languages for quantum computation. One of the issues is that we don't really understand what we can do with it yet. So there's and area where it's too early to design a language that captures the common problems and the specific styles or idioms that we would use to solve all of those problems. I'd point to some really old stuff: I'd say the world still has a/some problem with low-level code, in particular, there's a security issue which doesn't seem to go away. Seems to get worse by the year to year. We have one language which (has?) occupied that niche for 30 or 40 years now, which is the C programming language. It doesn't do a good enough job at solving the problems in that niche in a way that scale up to the kind of solution the industry wants, but it's successful enough that it has stopped any language from occupying that niche successfully. As a result of which, I would say: Yeah, we still don't know how to write low-level code in a way that's robust, in a way that's maintainable, in a way we can analyse cleanly. So, we're still looking for that language which is the successor of C. There has been a lot of attempts do design a successful language for C. That's what C++ started out as originally and ended up carving up its own niche, as well as the D language, quite literary from the name; it was supposed to be the successor of C that it never really achieved. Any mainstream adopters… So yeah I would say we've got some problems that are so new but we don't know how to address them yet, and we've got some problems that are old, but which we've never come up with good solutions for.

9.2.8 Is there a language you think will replace another?

I think what we're watching happening at the moment, is that Python won't form a complete replacement for C, but it will redefine the niche that we use C for. So many, many problems that we would've used C or C++ for 10 years ago are now being handed over to languages like Python. Partly that is just because they're not that performance intensive and running on modern hardware; Python is fast enough to solve many problems, and it's party because everything Python isn't fast enough for tends to get re-coded as a low-level C library and then plugged into Python. I think we will see a dwindling of the application areas that we will need low-level C code in, I think we will also see a dwindling of the areas where we need C++ for. And I think Python, and the other dynamic languages, like in that area, will lagerly take over those niches. That's not to say they will completely be replace that language because there will still be a few, like, sticky problems where we do turn to something so, for example: writing an operating system kernel, is one where you still need C. I don't think we'll see languages kill off older languages, I think what we'll see is they'll largely replace maybe in 9 out of ten application areas so that languages we want to mainstream become very specialist and very niche. Perhaps a slow death.

9.2.9 What do you like specifically in a language (feature, syntax, etc.)?

I'm very fond of Haskell, which is a functional programming language. Syntax wise and semantics wise, it's one of the crispest, kind of most minimal ways to prototype code, inspect things out and see what they do. I'm also a fan of Python, but for completely opposite reasons; Python is nowhere near as clean, but it's very flexible and it lets you do lots of things that are very bad, so, if I'm sitting trying to bang code out when I'm in a state where I don't know what it should do yet, and I want to explore, then hopefully converge on it doing something, then I would probably reach for Python because its design is clean enough and it's malleable enough that it's easy to work with. And then if I have something I want to lock down and understand exactly what it should do, and I want to try to get to the point where it's exactly that thing, I would probably reach for Haskell. Clean syntax, in both cases but meaning in a different way.