FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Comparison and Evaluation of JavaScript Preprocessing Languages

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Mertl, BSc

Matrikelnummer 0828834

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Franz Puntigam

Wien, 13.03.2016

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Comparison and Evaluation of JavaScript Preprocessing Languages

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Manuel Mertl, BSc
Registration Number 0828834

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dr. Franz Puntigam

Vienna, 13.03.2016

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Manuel Mertl, BSc
Engerthstraße 137/7/6, A-1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgments

Writing this thesis represents the end of my university time. Finishing the informatics master study, definitely is the biggest achievement at this point of my life. I want to use the next couple of lines to thank my best friends and everyone else who was somehow involved in my live, getting me to this point where I am today.

First of all I want to thank all of my friends, that have been there for me, at anytime. Many thanks to all of you, who always had time to grab some beer now and then in the evening, during my rare free time, and spend some fun hours.

I also want to thank my mother Margit, for not taking me serious and never being satisfied with nearly anything i accomplished. This encouraged me my whole live to go beyond my limits and I hope will keep this attitude till the end of my live, to get as successful as any how possible.

Special thanks to a very important friend of mine, Clemens Degendorfer. Without him I may have not ever started working with JavaScript. Working with him has teached me a lot of lessons that most people will never be able learn in their whole live.

Next, I want to thank Jürgen Cito, a good friend and student fellow. I really did not know many of my study colleagues, therefore buddies like Jürgen played an even bigger role during my time at the university. Also I want to thank Janos Tapolczai for helping me with theoretical informatics problems and related topics during the last couple of years. Big thanks to Mario Uher for giving me web development advices all the time.

Last but not least, I would also like to express my sincere gratitude to my advisor Franz Puntigam, for giving me the chance to write about this topic and for his support at any time.

Abstract

Over the past few years, the Internet has seen dramatic changes. In the beginning it was just a simple document browsing environment and now it has evolved to a rich software platform. A big reason for this development was the desire of people to access information anywhere at any time as comfortable as possible. This need has been incredibly boosted since smartphones and tablet computers entered the market. Around this point of time the use of Ajax technology has increased enormously as well. This and the fact that some of the upcoming mobile devices did not support Flash technology, which was used a lot until that time, leveled of the way for JavaScript. Since then the number of JavaScript-based websites and web applications has been growing steadily.
JavaScript has been developed in a very short period of time, without proper design considerations. Therefore, it has some major drawbacks like the missing of a module system, weak-typing, late binding, to name a few. To solve these disadvantages JavaScript preprocessors have been developed. These preprocessing languages are not only encapsulating some of the problems mentioned above, but also extending the language with useful features. A lot of JavaScript preprocessors exist these days, each having its own strengths and weaknesses and adding specific functionality to JavaScript. As the use of JavaScript in the web is increasing also the number of JavaScript preprocessing languages is rising. For a developer it is not easy to keep track which JavaScript preprocessors are available and decide which one to use for which kind of software project. Therefore choosing the appropriate language for a certain problem can be a very difficult decision. Finding the most fitting solution for an application, can be very hard to achieve.

In this thesis we will have a detailed look at web application projects, and determine important web project properties for a good project outcome. Therefore we will have a look at related work that points out which factors are affecting such a software development project outcome. Then we will further investigate the given web project properties and see which pre-processing language features are related to those properties and how they influence them. Given that list of pre-processing language features, we will then describe and measure them in detail. We will again have a look at related work, to see how language features can be properly measured and compared across languages. Given that comparison results we will be able to decide which of the four inspected JavaScript preprocessors (CoffeeScript, TypeScript, LiveScript, Dart) fits which kind of projects best, using a cost-benefit analysis.
Our comparison will show that in most of the cases Dart and TypeScript will be the best fitting pre-processing languages, followed by CoffeeScript. LiveScript is only a good choice for a very limited set of web projects.

Kurzfassung

In den vergangenen Jahren hat das Internet große Veränderungen erlebt. Anfangs war es nur eine Umgebung um einfache Dokumente darzustellen. Heutzutage bietet das Netz eine Plattform für eine Vielzahl von modernen Web-Anwendungen. Ein entscheidender Grund für diese Entwicklung war unter anderem das Bedürfnis der Menschen, Informationen zu jeder Zeit, an jedem beliebigen Ort, so komfortabel wie möglich zu konsumieren. Dieses Bedürfnis ist mit dem Aufkommen von Smartphones und Tablet Computern weiter gewachsen. Etwa zu dieser Zeit stieg auch die Verwendung von Ajax Technologie. Dies war, neben der nicht vorhandenen Unterstützung von Flash Technologie auf einigen mobilen Endgeräten, ein Grund dafür, dass JavaScript diesen rasanten Aufstieg erleben durfte. Seit dieser Zeit erhöhte sich die Anzahl an JavaScript basierten Webseiten und wächst noch immer stetig.

JavaScript wurde in sehr kurzer Zeit entwickelt, dabei wurden wichtige Design-Entscheidungen vernachlässigt. Diese wirken sich unter anderem in einem fehlenden Modul-System, schwacher Typisierung und später Variablen-Bindung aus, um nur einige Schwächen zu nennen. Um diese Nachteile zu lösen wurden JavaScript-Präprozessoren entwickelt. Diese Präprozessoren kapseln nicht nur einige dieser oben genannten JavaScript-Schwächen, sondern fügen der Sprache auch noch sinnvolle Erweiterungen hinzu. Heutzutage existiert eine große Anzahl an JavaScript-Präprozessoren, jeder mit eigenen Stärken und Schwächen. Da die Verwendung von JavaScript im Web wächst, steigt auch die Anzahl an JavaScript-Präprozessoren. Für einen Entwickler ist es daher sehr aufwendig auf dem Laufenden zu bleiben, welche JavaScript-Präprozessoren existieren und welcher Präprozessor für welche Art von Software-Projekt besonders geeignet ist. Den richtigen Präprozessor zu finden kann sich als äußert mühsam herausstellen.

In dieser Arbeit werden wir uns die Eigenschaften von Web-Projekten genauer ansehen, welche für einen guten Projektausgang erforderlich sind, davon abhängige Präprozessoren-Eigenschaften suchen, diese beschreiben und in den jeweiligen Sprachen miteinander vergleichen. Mit Hilfe dieser Vergleichswerte werden wir, unter Anwendung einer Kosten-Nutzen Analyse, die Entscheidung treffen, welche der vier untersuchten JavaScript Präprozessoren (CoffeeScript, TypeScript, LiveScript, Dart) für ein bestimmtes Web-Projekte am besten zu wählen wäre.
Wir werden sehen, dass Dart und Typescript in den meisten Fällen die beste Wahl ist, gefolgt von CoffeeScript. LiveScript ist bei sehr bestimmten Projekten überlegen ist.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1
Introduction

"Either write something worth reading or do something worth writing."

— Benjamin Franklin,
Founding Father of the United States

What started as a small net for researches, became the world's biggest network known as the Internet. In the last couple of years the Internet has seen some huge changes. From a simple network to view documents, it has evolved to an environment for rich web applications, alias Web 2.0. This progress would not have been possible without JavaScript. Using this language, it is possible to give web applications the look and feel of desktop software, which is a very important factor for working with web applications nowadays, to get accepted by users. Another relevant point is the possibility to do Asynchronous JavaScript and XML (Ajax) requests with JavaScript, instead of doing normal Hypertext Transfer Protocol (HTTP) requests, which forces the browser to load the given Uniform Resource Locator (URL), then reconstruct the view of the browser and therefore destroy the native application look and feel, that is known to be preferred in modern web applications [85, pp. 80-81] [3, pp. 800-801].

However, there are some major drawbacks in JavaScript, like missing of a module system and weak-typing [1]. The reason for these and many other weaknesses is, that JavaScript was developed in only a couple of days. Therefore, some design considerations were simply not thought through enough [74]. JavaScript preprocessors address the weaknesses in JavaScript and also extend the language with useful features. Today the number of JavaScript libraries as well as the number of JavaScript powered websites is increasing steadily. As side effect also the number of JavaScript preprocessors is increasing. This results in the problem, that developers have to decide which JavaScript preprocessing language is best suited for their needs, depending on certain software project criteria.

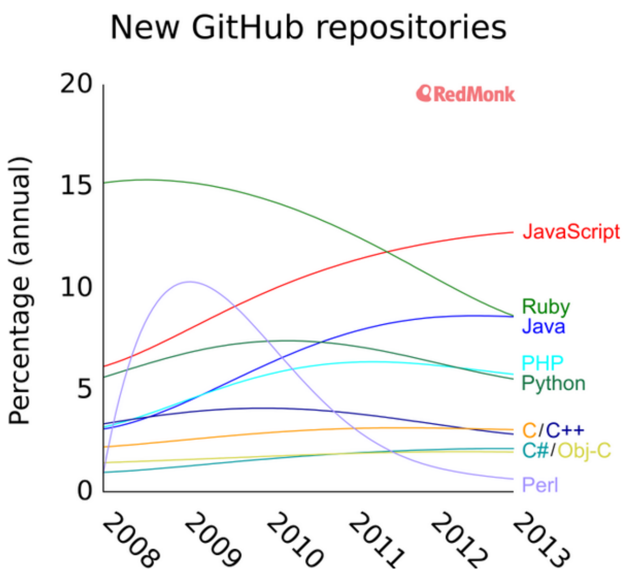


Figure 1.1: New GitHub repositories by language from 2008 - 2013 [5]

1.1 Problem Statement

A lot of different JavaScript preprocessing languages can be found on the Web nowadays. Therefore, choosing the appropriate language for a certain problem, can be a very difficult decision. JavaScript preprocessing languages, popular on GitHub are CoffeeScript [36], TypeScript [18], LiveScript [94] and Dart [41]. Each of these languages has its own strengths and weaknesses. Finding the most fitting solution for an application, can be very hard to achieve, especially if the developer has no experience with any of them at all. Often software engineers simply choose an arbitrary JavaScript preprocessing language without even evaluating if this choice will be best suited for their web project. They just pick a random preprocessor, to bypass some of the JavaScript weaknesses, but do not consider the extra features that each JavaScript preprocessor brings along. If developers would spend more time evaluating what their most important project criteria are and which JavaScript preprocessor fits this situations best, then the chance of a better outcome would increase.

1.2 Expected Results

Each of the JavaScript preprocessing languages we are having a look at has its own benefits. Therefore, different languages suit different kinds of projects, depending on certain criteria. We will see, that one of the most famous ones, CoffeeScript, is not the best language to use in every case, and there is not even a general best solution, because it depends very much on the environment of the project. Furthermore, this evaluation will give us an overview of the features of different JavaScript preprocessing languages used nowadays and show us the big as well as the smaller differences between them.

1.3 Contribution

This work is mainly interesting for software development companies, who are curious which of the below mentioned languages should be used in which kind of web project. By definition a project is a unique set of actions and properties. Some of these properties are important for choosing the best fitting solution to reach the goal of the project. In this work we are concentrating on software projects, more precisely web applications with special focus on JavaScript. By 'best fitting solution' we are referring to the optimal JavaScript preprocessing language, mentioned in Section 1.1, used under certain circumstances, influenced by the project itself as well as the projects environment. More specifically, we want to address the following research question:

Which specific JavaScript preprocessing language (CoffeeScript, TypeScript, LiveScript, Dart) is best suited for a new web application project, depending on some of the projects properties and its environment?

1.4 Methodology

In this thesis we will try to give the answer to the above mentioned research question, by following the below shown steps in the given order:

- Determine important web project properties for a good project outcome. Therefore we will have a look at related work that points out which factors are affecting such a software development project outcome
- Then we will further investigate the given web project properties and see which preprocessing language features are related to those properties and how they influence them
- Given that list of pre-processing language features, we will then describe and measure them in detail. We will again have a look at related work, to see how languages features can be properly measured and compared across languages

1.5 Organization

The topics of the thesis will be organized as follows:

- Chapter 2 provides an overview of the background, covering scripting languages in general and JavaScript in detail. Further, it will give an introduction to the four inspected JavaScript preprocessing languages.
- Related research work will be presented in Chapter 3 to look over the rim of the mug and see the bigger picture.
- In Chapter 4, we will apply the steps given in Section 1.4 to gather the data, that will later be used as foundation for a cost-benefit-analysis.
- Concluding remarks and an outlook to possible future work can be found in Chapter 5.

CHAPTER 2

Background

"Education is not preparation for life; education is life itself."

— John Dewey,
philosopher and psychologist

To give some background information, and help to understand the topic of this thesis, this chapter will provide the necessary information about scripting languages, JavaScript, preprocessors in general as well as detailed information about our four inspected JavaScript preprocessors (CoffeeScript, TypeScript, LiveScript, Dart).

2.1 Script Languages

In this section we will describe what script languages are, where they are being used and name some representatives. In addition we will also explain what system programming languages are, where they are getting applied and name some prominent examples as well.

2.1.1 What are Script Languages?

Script languages also known as scripting languages, or very high-level programming languages (VHLL) are programming languages that are usually interpreted during runtime[1], so there is no need for an ahead-of-time-compiler[2]. A VHLL is on a very high level of abstraction, and can be seen as a goal-oriented programming language. The primary focus of scripting languages is not to build applications from scratch, but rather to glue components together and therefore

[1]The time during which a program is executed

[2]Transforms source code into an other computer language, before the execution of the program in contrast to just-in-time-compiler, where the code gets compiled during runtime

make rapid development possible [73, p. 17]. The components are usually written in a system programming language [68] (see Section 2.1.2). Famous scripting languages are JavaScript [89], Perl [69], Python [29] and Tcl [10] to name only a few. All of these languages can be seen as gluing languages and are mainly used if components already exist and are supposed to be working together, hence shall be integrated. Therefore this languages are also called system integration languages [68, pp. 24–25]. The popularity of scripting languages is increasing, not only because the development process is faster by just gluing components together, but also because computer hardware speed is increasing and therefore offering a better user experience while working with script languages in general [68, p. 28]. System integration languages are used in all kind of different areas. One of this areas would be bioinformatics, where these languages are used to acquire, store, organize, archive, analyze and visualize the corresponding data. For these tasks, scripting languages play a vital role, because people were using Lisp and related languages in that area for a long time [67, p. 1174].

2.1.2 What are System Programming Languages?

System programming languages have evolved as an alternative to assembly languages. In assembler each statement represents a single machine instruction, so developers have to deal with complex low level problems (e.g. register allocation). That changed as higher level languages appeared (e.g. Fortran). Here a compiler translates each statement into a sequence of binary operations. [68, pp. 23–24]

2.1.3 Differences to System Programming Languages?

To recap, Figure 2.1 gives an overview about the types of languages, we have been talking so far.

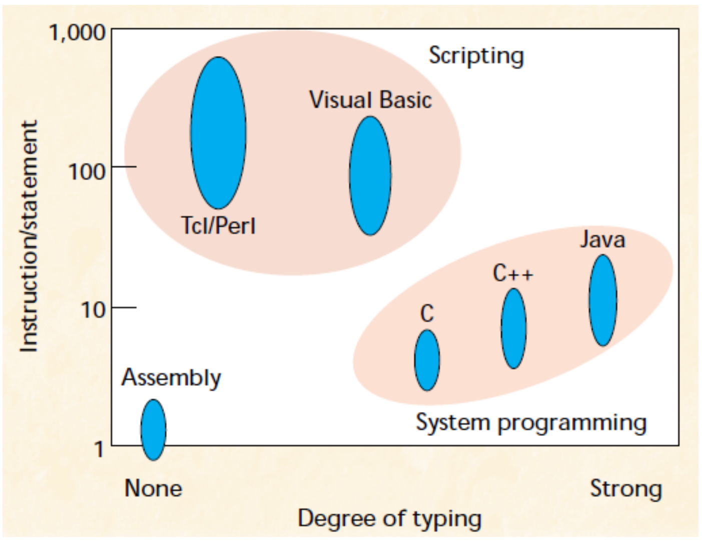


Figure 2.1: Comparison of various programming languages [68, p. 25]

System programming languages are normally strongly typed to manage the complexity of big applications, and have to be compiled in order to be executed. They are used to build components which may later be reused in system integration languages, while scripting languages are generally not statically typed, and do not have to be compiled in order to be executed, they just get interpreted. VHLL can be seen as languages which glue together components instead of defining them new.

2.2 JavaScript

The following subsections will describe JavaScript. Additionally we will hear about its history, from the beginning up to now, its application areas and some future prospects. Furthermore, in the last subsections we will discuss the major drawbacks of this language and a new standard for JavaScript.

2.2.1 Introduction

First of all we need to know that JavaScript is not some long version for Java. Java has nothing to do with JavaScript, it was just an unfortunate pick of a name[3]. As shown in Figure 2.1, Java is a system programming language in contrast to JavaScript which is a scripting language, as its name states. That concludes that Java is not anyhow related to JavaScript. JavaScript currently implements the ECMA5 standard, described in Section 2.2.1.1 below. In fact, JavaScript is even more linked to functional languages like Lisp than to C or Java [21]. E.g. JavaScript uses arrays and objects instead of lists and property lists. Ahn et al. writes [2, p. 497]:

> "JavaScript is a dynamically-typed language, where variable types are not declared and everything, including functions and primitives, may be treated as an object. An object is a collection of properties, where a property has a name and a value. The value of a property can be a function, which is then referred to as a method of the object."

2.2.1.1 ECMA5 Standard

European Computer Manufacturers Association (ECMA) is an international non-profit organization that was founded in 1961 to standardize computer systems. One of their standards is the ECMAScript also known as ECMA-262, which is a standard for scripting languages (e.g. JavaScript). Until now, five versions of ECMAScript have been published, starting in 1997. In Section 2.2.4 we will have a look at the newest standard, and describe some of the new features of ECMA6 more detailed.

2.2.2 History

In the next paragraphs we will hear about how it all started, the Ajax support and the reasons for the boost of JavaScript. Furthermore we will discuss JavaScript at the server-side and finally give some information about the future of this language.

2.2.2.1 The Beginning: Mocha

JavaScript was developed in April 1995. During this time it was known under the codename Mocha, and later released under the name LiveScript [4]. In only 10 days, Brendan Eich had to create a programming language that would run in Netscape's browser Netscape 2.0 Beta

[3]In the beginning it was not even called JavaScript, see the history section for more details in Section 2.2.2

[4]This name has nothing to do with the LiveScript from Section 2.3.7

which was released later that year. For most developers designing and building a programming language in only this short period of time, with some given constraints seems impossible, but Brendan Eich already had experience building programming languages from his time as student at the University of Illinois. Brendan Eich made an interesting statement about JavaScript and why he had not built in classes, years after JavaScript was released [74, p. 7]:

> “If I had done classes in JavaScript back in May 1995, I would have been told that it was too much like Java or that JavaScript was competing with Java (...) I was under marketing orders to make it look like Java but not make it too big for its britches (...) [it] needed to be a silly little brother language.”

2.2.2.2 The Change: Ajax

About one decade after JavaScript was released, a revolution started, Ajax came up. Using this technology it allowed JavaScript to retrieve data from servers and update the HTML document without doing a full-page-request-response cycle, which would include the reload of the page and therefore give the user a bad working experience. With Ajax it was finally possible to build incredible high interactive graphical user interfaces and give the user the feeling of a desktop-like application.

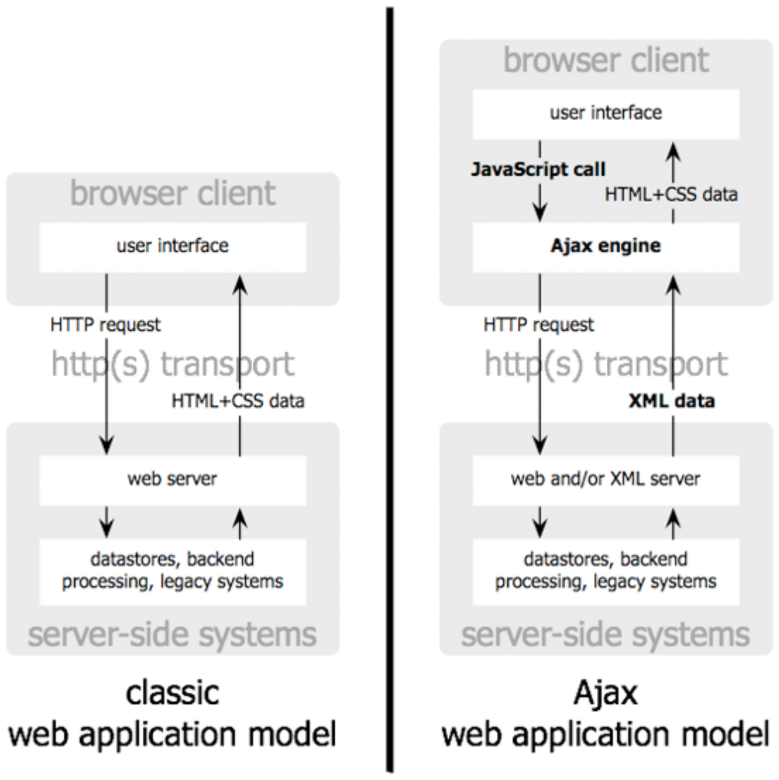


Figure 2.2: Traditional model (left) compared to the Ajax model (right). [31]

To give a better understanding and overview about this technology, Figure 2.2 shows the traditional model for web applications on the left side and the Ajax version on the right sight.

Ajax has basically laid the foundation for the Web, as we know it today. Since nearly all browsers support JavaScript and therefore also Ajax requests are possible, the use of this technology has increased enormously. Today there is a shift to the Web. A large number of applications, that would have been developed for the desktop, some years ago, are nowadays written for the Web [3, p. 800]. To give a good user experience developers are nearly forced to use Ajax. It is hard to imagine how the web would look like today, if there were no JavaScript as well as the Ajax technology.

2.2.2.3 The Boost: Mobile

JavaScript experienced an incredible push as mobile devices like smartphones and tablets were brought to the market. Around this point of time, people more than ever, wanted to access information anywhere at anytime. This forced companies to develop proprietary applications for the mobile operation systems (e.g. iOs [38], Android [47]) and/or develop websites, which support the mobile view in a usable way, to satisfy the users needs. Also some JavaScript frameworks were developed specially for building mobile web applications with JavaScript like jQuery Mobile [53], Sencha Touch [48] and Ionic [9] to name only a few. An additional aspect would be that Apple's mobile devices like iPads and iPhones, are selling extremely well these days, but Apple does not support Flash on their devices, so most websites will use JavaScript powered HTML5 [83] to be accessible on these gadgets [37], if they do not want to develop proprietary applications for different mobile operating systems. These facts boost the number of JavaScript powered websites as well as the number of JavaScript frameworks that were brought to the web, and therefore the JavaScript community is getting bigger and bigger.

2.2.2.4 The Server-Side: Node.js

JavaScript is not only a client-side programming language anymore. With Node.js JavaScript made its way to the server-side.

> "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices." [28]

Since the number of JavaScript developers is increasing it was only a matter of time until this language would also find its way into the server environment. In 2009 Node.js was initially launched, and quickly got international attention. From there on the popularity of this new server-side language increased crucially. As a result of this, today JavaScript is more popular than ever, even more popular than Java, PHP or C as shown in Figure 2.3, related to job trends. As a consequence also the number of jobs for JavaScript developers are rising as indicated in Figure 2.4.

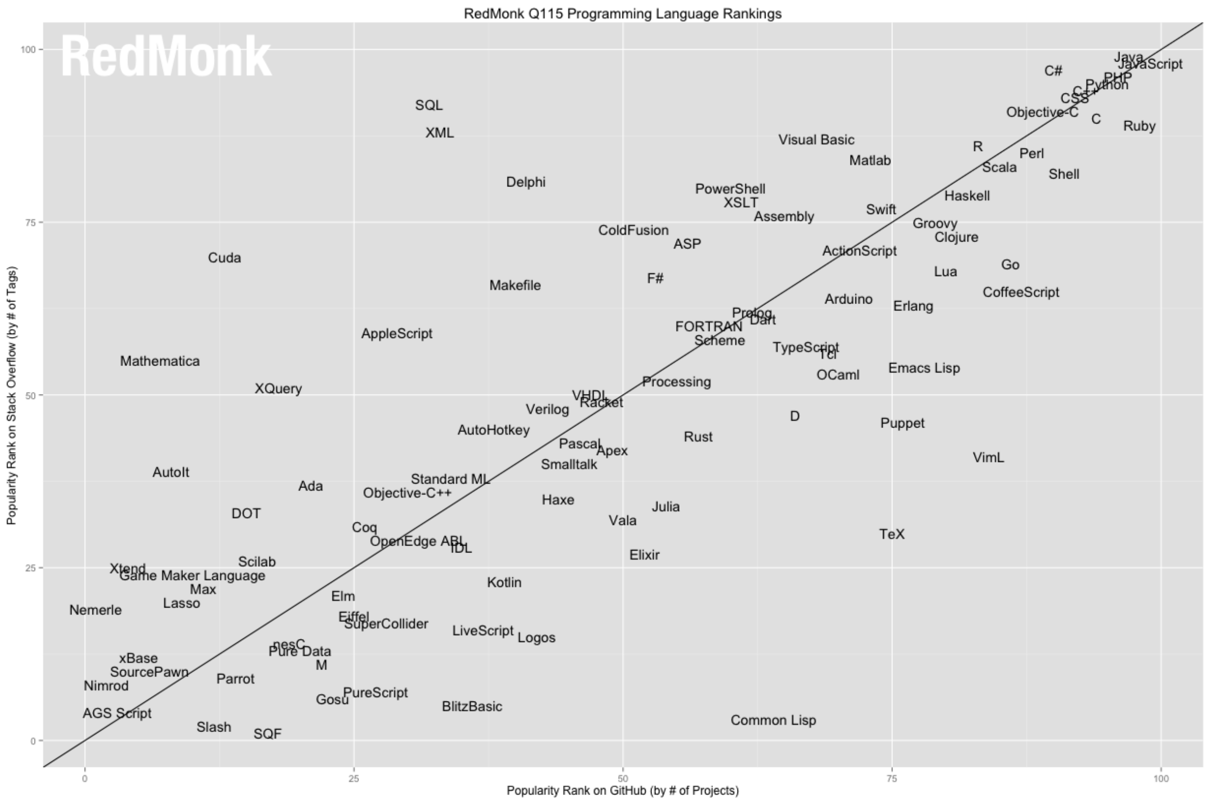


Figure 2.3: Popularity Rank on GitHub (by Number of Projects) [84]

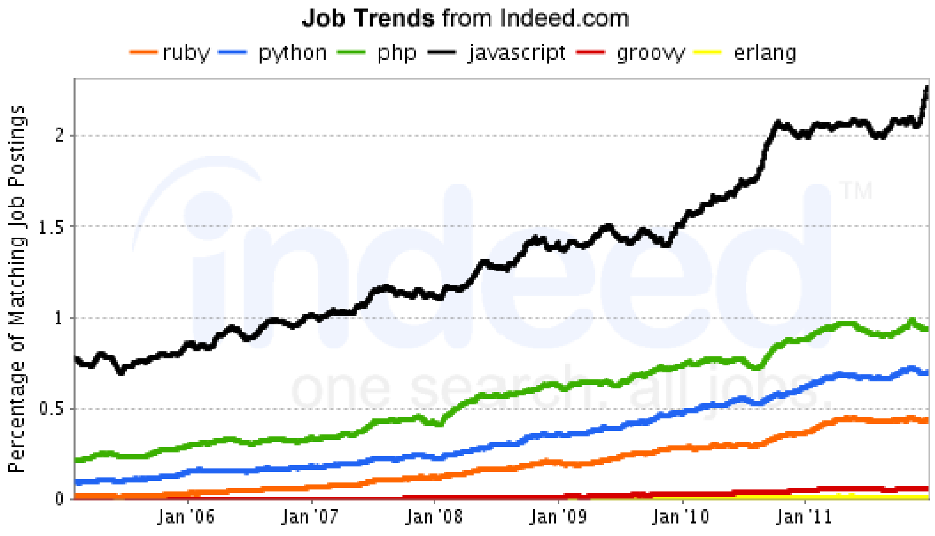


Figure 2.4: Job Trend Analysis [24]

2.2.3 Drawbacks

First of all we have to say that JavaScript is not as bad as its reputation is. Although the early versions of JavaScript were pretty weak, lacked of exception handling, inheritance and were quite buggy at all, compounding that, the web browsers environments were horribly buggy as well during these days. The present form of JavaScript can be seen as a matured language, but still many opinions are based on its immature form of the language and there are also still a couple of major drawbacks (e.g. missing of a module system) [1, pp. 13–16]. The reason for these and many other weaknesses is, that JavaScript was developed in only a couple of days. Therefore, some design considerations were simply not thought through. In the next paragraphs we will have a detailed look at some of these drawbacks. Some of them will be tackled by the new ECMA6 Standard, which we will describe in Section2.2.4. Also preprocessors can be used to solve many of the following mentioned disadvantages. In Section2.3 we will therefore discuss preprocessors more detailed.

2.2.3.1 Scoping

A scope can be seen as an enclosing context where variables and expressions are associated. Through scopes it is possible to keep the visibility of those items apart.

No Access Modifier

Access modifiers define the way an object can be accessed. Therefore the risk of accidental side-effects will be reduced. Additionally it will help building modular programs. In Listing 2.1 we will show three kinds of different modifiers in Java.

- private: only accessible within the declared class itself
- public: accessible from any other class
- protected: accessible by subclasses in other packages or any class within the package of the protected item

```
public class Example {
   private String id; public String firstname; protected String city;
}
```

Listing 2.1: Java Access Modifier

In JavaScript there are simply no modifiers as in Java. But there is a known workaround, which offers an access modifier-like behaviour. By using the keyword *this* the item can be seen as a public declaration, and by using the keyword *var* the item behaves as a private item. Listing 2.2 will show the Java example from above ported to JavaScript without the city property, since there is no way to build a *protected*-like behaviour in JavaScript.

```
function Example(){
   var id; this.firstname;
}
```

Listing 2.2: JavaScript Access Modifier

No Block Scoping

Under *Block Scoping* we understand to make an item only accessible in a certain area of the code. This drawback and its solution in ECMA6 will be discussed in Section 2.2.4 - Block Scoping.

Global Variables

If not specified in an other way, each JavaScript program runs in a global scope. The same is true for variables. Each declared variable will be created in the global scope if not specified in an other way. This is a problem because, if different libraries use the same variable name and define it globally, the libraries will use each others variable. Therefore sooner or later, a conflict will occur. To define a variable local the keyword *var* has to be used, for global definition no keyword is necessary. Since most of the time developers will need local variables, this is kind of a weird behaviour.

2.2.3.2 Object-Orientation

Though some might say that JavaScript is not an object-oriented language, it is possible to code in an object-oriented way. The reason for this is that the language supports inheritance through prototyping as well as properties and methods. Using JavaScript, one can inherit implementation via prototyping but this goes to the expense of encapsulation. We will give a short example where prototyping breaks the encapsulation.

```
function Car() {
  var speed = 0;
  this.getSpeed = function() { return speed; }
}
```

Listing 2.3: Basic JavaScript

Shown in Listing 2.3 the variable *speed* can not be modified directly by external code. In the next example we will use the prototype approach to add the same method.

```
function Car() {
  var speed = 0;
}
Car.prototype.getSpeed = function() { return speed; }
```

Listing 2.4: Basic JavaScript extended via Prototyping

This new added function will return *undefined* because, the *getSpeed* method is not in the same scope as the variable *speed* and therefore is not able to access its value. To be able to access the value of *speed* one has to change the visibility of this variable, by using the *this* declaration instead of *var*. But by writing *this.speed = 0* the variable would be accessible from outside of the *Car* function, therefore the encapsulation would be broken.

2.2.3.3 Modularization

By modularization, we refer to the method to split a program into independent parts of code. But JavaScript does not support the *import* statement as well as namespaces in ECMA5. Without namespaces it is very hard to be sure that every code fragment stays in its scope. So using different libraries, it would be possible that some of them may use global variables with the same variable name and therefore, they would assign each others values. We can see that this is a very major problem in this environment.

2.2.3.4 Type System

JavaScript has a dynamic type system. Therefore it can store an arbitrary type of value. It does not matter with which value the variable has been initialized. The type of value can be changed afterwards. Listing 2.5 will show a valid example of JavaScript assignments. From this it follows, that it is very hard to create an interface, since the type of given data is not checked during compile time, but rather during runtime, which may result in an error.

```
var text = 'JavaScript';
console.log(text); //'JavaScript'
text = 1288;
console.log(text); //1288
text = new Array('JavaScript', 'Preprocessing');
console.log(text); //['JavaScript', 'Preprocessing']
console.log(text.length); //2
```

Listing 2.5: JavaScript Type System

2.2.3.5 Interpretation

JavaScript does not get ahead-of-time compiled, therefore it is very hard to find errors, since finding an error implies that the corrupt part of the code is executed. If the developer does not write any unit tests or other kind of system or integration tests, the error will occur during runtime. Listing 2.6 will show a valid JavaScript example, but the *if* branch will never be entered due the typo mistake, instead the *else* branch will be entered every single time the code is executed. This would not be possible in an ahead-of-time compiled languages, which would detect, that the variable *entre* does not exist. Of course this is not just a JavaScript problem, but a problem for any language that does not get ahead-of-time compiled.

```
var enter = true;
if(entre){ //typing mistake
  console.log('never reached due entre variable is undefined');
}else{
  console.log('allways will prompt this text');
}
```

Listing 2.6: JavaScript Interpretation

2.2.3.6 Rendering

One major problem is that different browsers may render JavaScript differently. The reason for this is, that different browsers use different JavaScript engines. Though a universal standard is getting closer, there are still some differences that may result in rendering problems. Therefore the developer is nearly forced to test the code in all major browsers, to be sure that the code will be displayed correctly in the main environments.

2.2.3.7 Development

When developing an application with JavaScript, there are some points that have to be considered. In the next paragraphs we will discuss the development of large JavaScript applications, the maintainability of them as well as the compatibility problems that might go along.

Large Application

All of the before mentioned drawbacks, like the missing of a module system or some scoping disadvantages, make it difficult to develop a large application with JavaScript. Nevertheless, creating a big system using this language is possible. One should use many third party libraries, that are well tested, and consider using a JavaScript preprocessor as well. This might reduce the number of possible problems that would occur without a JavaScript preprocessor. Additionally the web offers many good template structures for large JavaScript applications, which one should have a look at, before starting a new project. In most cases this templates are really good and help to stick to certain criteria, that are important for bigger applications.

Maintainability

To ensure a high level of maintainability it is somehow necessary to break the complex application down in smaller more maintainable parts. Therefore the use of a good file structure is as important as well defined coding guidelines and writing comments for later traceability. Following these principles it will help to increase the maintainability of a complex JavaScript project.

Compatibility

To be sure that a JavaScript program will also run with a future version of JavaScript, some rules should be observed. An example would be the use of certain keywords like *enum* and *class*. This words have no special meaning in ECMA5 and can be used without a problem, but in ECMA6 this words will be keywords and therefore could destroy the application. Whether errors will occur depends on the implementation of the browsers JavaScript engine.

2.2.4 The Future: ECMA6

The current version in progress is ECMA6 [5]. This sixth version adds many features that will make building complex applications easier (e.g. ECMA6 will include classes as well as modularization). In the next paragraphs we will introduce some of the features that will be brought up by ECMA6. Most of the following features are already used by the preprocessors, we will have a look at later on. Using these preprocessors gives you the benefits of some future ECMA6 standard features, that we will explain in detail in the following paragraphs:

2.2.4.1 Classes

ECMA5 has no class support, but there exist some workarounds. One way to implement a class-like behaviour is shown in Listing 2.7.

```
function Car(type) {
  this.type = type;
  this.speed = 0;

  this.accelerate = function(){
    this.speed += 1;
    console.log('current speed: ' + this.speed);
  }
}
//Instantiate a new car
var car = new Car('Lamborghini');
car.accelerate(); //'current speed: 1'
```

Listing 2.7: ECMA5 Car Class

[5] Short for ECMAScript version 6 also known as ES6 Harmony

As we can see, there is a way to use functions as some kind of classes, but there still is no constructor. ECMA6 solves this problem by providing a keyword *class* and *constructor*, as well as making inheritance possible by using the keyword *extends*. This features will be shown in Listing 2.8 below.

```
class Car {
  constructor(type) {
    this.type = type;
    this.speed = 0;
  },
  accelerate(){
    this.speed += 1;
    console.log('current speed: ' + this.speed);
  }
}

class RaceCar extends Car {
  constructor(type, nr) {
    super(type);
    this.nr = nr;
  },
  doubleSpeed(){
    this.speed *= 2;
    console.log('current speed: ' + this.speed); //output: 'current speed: 2'
  }
}
//Instantiate a new car
let rcar = new RaceCar('Lamborghini',1);
rcar.accelerate(); //'current speed: 1'
rcar.doubleSpeed(); //'current speed: 2'
rcar.doubleSpeed(); //'current speed: 4'
```

Listing 2.8: ECMA6 Car Class

2.2.4.2 Modules

ECMA5 does not support modularization in its core. There are some third party libraries like CommonJS [23] or node modules, which will add this feature to ECMA5. In ECMA6, modularization will be built in the core of the language, therefore it will use the keywords *import* and *export*. Listing 2.9 will show a sample module using an import statement.

```
<module>
  import $ from 'lib/jquery';
  var nr = 123;

  //scope is not global, therefore $ and nr will not be found in the window
  console.log('$' in window); // false
  console.log('nr' in window); // false
  //(`this` still refers to the global object)
  console.log(this === window); // true
</module>
```

Listing 2.9: ECMA6 Modul

2.2.4.3 Block Scoping

JavaScript may be a little confusing for Java or C developers because variables in ECMA5 are either globally or locally defined. ECMA5 simply does not support block scoping. In the example, shown in Listing 2.10 the variable *x* can be seen as such a block scope variable.

```
for (let x = 0; x < 3; x++) {
  console.log('value of x in block: ' + x);
}
console.log((typeof x === 'undefined')); //true
```

Listing 2.10: ECMA6 Block Scoping

2.2.4.4 Promises

Promises handle the results of asynchronous operations. They can be seen as a kind of callbacks with improved readability. That is because promises can use chaining of methods as we can see in the following example shown in Listing 2.11, where a JSON request, to a specific URL, is made, returning an object called *post*. *Then* an other request is made to an other specific URL, which *then* returns an other object, here called *comments*.

```
jsonReq("/customer/51288").then(function(post) {
  return jsonReq(post.commentURL);
}).then(function(comments) { //multiple then statements (chaining) possible
  // proceed with access to customer
}).catch(function(error) {
  // handle errors in either of the two requests
});
```

Listing 2.11: ECMA6 Promises

2.2.4.5 Arrow Functions

This feature in ECMA6 is more or less just syntactic sugar. By using arrow functions the use of the keyword *this* gets reduced. Listing 2.12 will show a counter snippet using ECMA5 while Listing 2.13 will show the same functionality using ECMA6. The code shown here, implements a simple timer, that outputs, every 1000 milliseconds, the current number starting with 1 and also increasing this number every 1000 milliseconds by 1.

```
function Timer(){
  var self = this;
  self.nr = 0;

  setInterval(function start(){
    self.nr += 1;
    console.log('current number: ' + self.nr);
  },1000);
}
```

Listing 2.12: ECMA5 Timer

```
class Timer {
  constructor(){
    this.nr = 0;
  },
  setInterval(() => {
    this.nr += 1;
    console.log('current number: ' + this.nr);
  }, 1000);
}
```

Listing 2.13: ECMA6 Timer

Another example of arrow function usage can be seen in Listing 2.14 below.

```
var sum = function(num1, num2) {
  return num1 + num2;
};
// equivalent to:
var sum = (num1, num2) => num1 + num2;
```

Listing 2.14: ECMA5 vs ECMA6 Arrow Function

2.3 Preprocessors

In the following sections we will discuss what preprocessors are, what they do and how they work. Additionally we will talk about why JavaScript preprocessors should be used, and give some example preprocessing languages. Furthermore we will introduce four major JavaScript preprocessing languages and take a closer look at them.

2.3.1 What is a Preprocessor?

A preprocessor can be seen as a program which takes data as input and uses this data to generate output (preprocessed data), which can then be used as input for some other program.

Figure 2.5 illustrates a sample flowchart of a C++ application. Since JavaScript is interpreted and not compiled, this flow is only valid up to the *Preprocessed Source Code* node, for JavaScript.

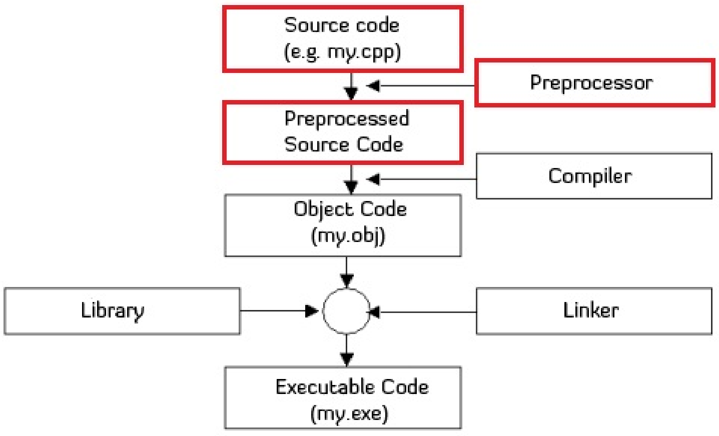


Figure 2.5: Sample C++ Flowchart [79]

We can distinguish two different major types of preprocessors:

- **Lexical preprocessors** can be seen as low-level preprocessors, they simply perform a substitution of tokenized characters for other tokenized characters.
- **Syntactic preprocessors** are more complex and can be used to customize the syntax of a language or embed domain-specific programming language.

All our later investigated JavaScript preprocessing languages are therefore syntactic and not lexical preprocessors.

2.3.2 What does a JavaScript Preprocessor do?

JavaScript preprocessing languages take a given code (e.g. CoffeeScript), written in their specific syntax and compile this code into JavaScript. The resulting code gets interpreted by the browser since it is just normal JavaScript, without any preprocessing specific language syntax.

2.3.3 Why should we use JavaScript preprocessors?

JavaScript was developed in only a couple of days, therefore it lacks of features that nowadays are appreciated by many developers (e.g. modularization) and is also missing important language specific functions (e.g. type systems). Preprocessors are adding different features and trying to wrap the weakness of JavaScript. Since using a JavaScript preprocessor involves compilation, we have an additional step, which we would not have, writing normal JavaScript applications. Through this step we are able to throw errors before they would happen during runtime as using JavaScript without a preprocessor. Therefore the use of a preprocessing languages makes it easier to find errors and prevent them. Each preprocessing language has its own strengths and weaknesses. Therefore the choice, which preprocessing language to use, should not be made in hurry due to the fact, that a large number of JavaScript preporcessors can be found these days.

2.3.4 Which JavaScript Preprocessors do exist?

Since various developers address different problems, a big number of JavaScript preprocessors, treating different issues, coexist nowadays. The list below shows only some of them, to give an idea about how much development goes on in this area [25].

CoffeeScript – Unfancy JavaScript

TypeScript - Typed superset of JavaScript

LiveScript – Compatible with CoffeeScript, more functional, and with more features

Dart - C/Java-like syntax with optional typing

GWT - Google Web Toolkit, compiles Java to JavaScript

Pyjamas - Python to JavaScript

Opal - Ruby to Javascript compiler

Scala.js - A Scala to JavaScript compiler

Kaffeiine - Enhanced Syntax for JavaScript

Move - A simpler and cleaner programming language

RedScript - Ruby flavored JavaScript

Roy - JavaScript semantics with some features common in static functional languages

In the next subsections we will discuss four JavaScript preprocessing languages:

- **CoffeeScript** (Section 2.3.5)
- **TypeScript** (Section 2.3.6)
- **LiveScript** (Section 2.3.7)
- **Dart** (Section 2.3.8)

We chose the four above mentioned languages, out of the list [25], for various reasons. CoffeeScript has the biggest community at this time, in the area of JavaScript preprocessing languages measured by the number of tags on Stackoverflow as well as the number of repositories on Github. The focus of LiveScript lies on adding features in the area of functional style programming as well as imperative programming. TypeScript and Dart are backed by big companies like Microsoft and Google which lifts them off the other two.

We will always start with a short quote from the respective website followed by some general information of this preprocessing language and finally, and most important, the features it adds to JavaScript. Figure 2.6 shows the distribution of tags on Stackoverflow and the number of GitHub repositories by preprocessing language [07.07.2014 - 21:00].

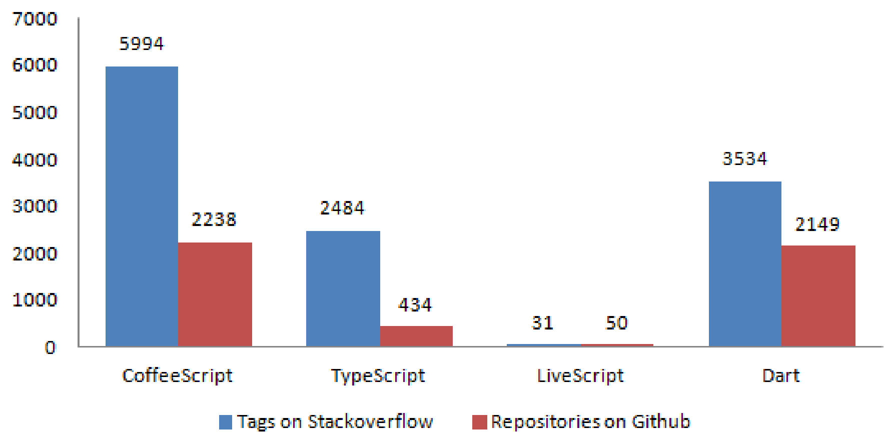


Figure 2.6: Preprocessing Overview by Tags and Repositories

2.3.5 CoffeeScript

"CoffeeScript is a little language that compiles into JavaScript. Underneath that awkward Java-esque patina, JavaScript has always had a gorgeous heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way." [36]

Developed by	*Jeremy Ashkenas* [90]
First Appeared in	*2009*
Latest Stable Release	*1.7.1 / January 29, 2014*
Influenced by	*Haskell, JavaScript, Perl, Python, Ruby*
License	*MIT License* [63]
Filename extension	*.coffee*
Tags on Stackoverflow	*5994*
Repositories on GitHub	*2238*

Table 2.1: CoffeeScript facts [87]

This section describes the features of CoffeeScript, starting with some general information about the language and then going into more detail where the general information is presumed. For a full list of all the features CoffeeScript adds to JavaScript see coffeescript.org [36].

General Information

Using CoffeeScript, there is no need to declare variables. They get automatically declared, without the need of the *var*-keyword. Also the keyword *function* will be replaced by an arrow symbol and the number of brackets gets reduced. Furthermore CoffeeScript uses spaces and tabs to do the code nesting. Semicolons are optional, which means they can be used, to improve readability sometimes, but do not have to be used. A function automatically returns the result of the last statement in its block.

Functions

Functions are written using an arrow symbol as shown in the following examples. If no parameters are passed, there is no need for brackets. Additionally the use of default parameter values is possible, if an argument is missing [6]. In this example no *return* statement has to be used in the CoffeeScript code, since CoffeeScript simply returns the last statement in a function.

[6]Missing in this context means the argument is *null* or *undefined*

```
var hello_world = function(){
  return 'Hello World!';
};

var sum = function(number1,
    number2){
  if(number2 == null){
    number2 = 0;
  }
  return number1 + number2;
};
```

Listing 2.15: JS Function

```
hello_world = -> 'Hello World'


sum = (number1, number2 = 0) ->
    number1 + number2
```

Listing 2.16: CS Function

Objects

As told in the general information Section 2.3.5, brackets are often optional and not necessary as the following examples will show. The nesting is simply done by considering the tabulators or spaces.

```
var forms = {
  circle: {
    color: 'blue'
  },
  recktangle: {
    color: 'green'
  }
};
```

Listing 2.17: JS Objects

```
forms =
  circle:
    color: blue
  recktangle:
    color: green
```

Listing 2.18: CS Objects

Scope

The CoffeeScript compiler automatically takes care of the scoping. There is never the need of using the *var*-keyword. CoffeeScript detects in which scope which variables are declared and can be used.

```
var outer = 1988;

var addTwo = function() {
  var inner = 2;
  return outer + inner;
};

var inner = addTwo();
```

Listing 2.19: JS Scope

```
outer = 1988
addTwo = ->
  inner = 2
  outer + inner
inner = addTwo()
```

Listing 2.20: CS Scope

If Statements

There are different ways to use an *if* statement in CoffeeScript, they can be used to make the code more structured and easier to read.

```
var color, lunch;

if (isColorAvailable()) {
  color = getColor();
}

if (green && red) {
  greenRed();
} else {
  blue();
}

lunch = monday ? pizza : burger;
```

Listing 2.21: JS If statement

```
color= getColor() if
    isColorAvailable()

if green and red
  greenRed()
else
  blue()

lunch = if monday then pizza else
    burger
```

Listing 2.22: CS If Statement

Splats

Splats are indicated by three dots and can be used to define a variable number of arguments. In the following example we will show that a function with only three parameters is able to except even more than these three. This feature can be very useful if the number of arguments has a minimum limit but can extend to a higher amount of parameters.

```
numbers = ['one','two','three','four']

printNumbers = (one, two, others...) ->
  alert 'One ' + one //'One one'
  alert 'Two ' + two //'Two two'
  alert 'Others ' + others //'Others three,four'

printNumbers numbers...
```

Listing 2.23: CS Splats

Loops

There are several ways to use loops in CoffeeScript. Different loops can be used to increase the readability of the code. We will give a few examples below.

```
//simple iteration, calling method play with argument game
play game for game in ['football', 'soccer', 'baseball']

//calling method watch with argument i+1 as index and game
games = ['football', 'soccer', 'baseball']
watch i + 1, game for game, i in games

//calling method play with argument game if game is not 'baseball'
play game for game in games when game isnt 'baseball'

//creating an array with numbers 10 to 1
countdown = (num for num in [10..1])
```

Listing 2.24: CS Loops

Arrays

Arrays can be sliced and spliced in a simple way. It is even allowed to use ranges to get the specific elements. Additionally negative indices allow us to start at the end of an array. The listing below will give a few examples.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
start = numbers[0..2] //[1,2,3]
middle = numbers[3...-2] //[4,5,6,7]
end = numbers[-2..] //[8,9]
copy = numbers[..] //[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listing 2.25: CS Arrays

Expressions

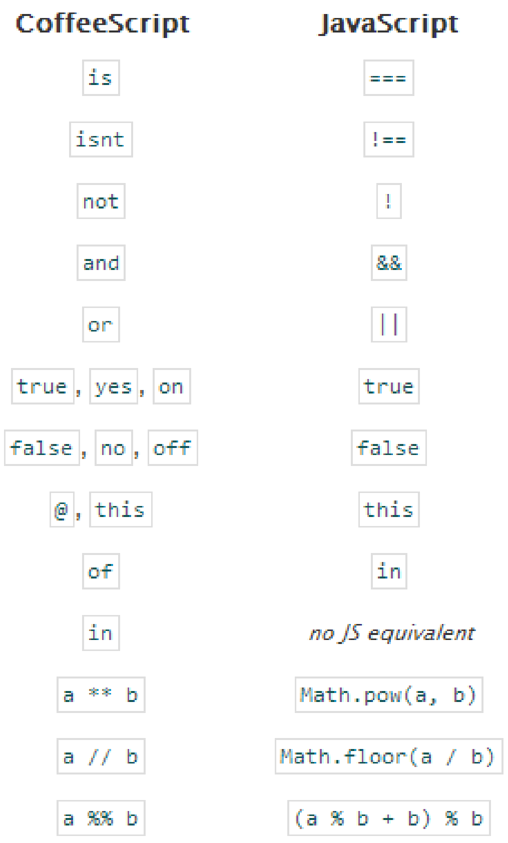
In CoffeeScript there is nearly no need to write a *return* statement, as the following examples will show. The result of the last statement executed in a function will be automatically used as return value.

```
number = (nr) ->
  if isOne(nr)
    1
  else if isTwo(nr)
    2
  else
    "3 or bigger"

x = if 10 > 1 then "Nr1" else "Nr2" // equivalent to x = "Nr1"

five = (two = 2) + (three = 3) //five == 5
```

Listing 2.26: CS Expressions

Operators and Aliases

To make the code more readable and easier to understand, CoffeeScript introduces some aliases for operators. The figure below shows the equivalent operations from CoffeeScript and JavaScript.



CoffeeScript	JavaScript
`is`	`===`
`isnt`	`!==`
`not`	`!`
`and`	`&&`
`or`	`||`
`true`, `yes`, `on`	`true`
`false`, `no`, `off`	`false`
`@`, `this`	`this`
`of`	`in`
`in`	*no JS equivalent*
`a ** b`	`Math.pow(a, b)`
`a // b`	`Math.floor(a / b)`
`a %% b`	`(a % b + b) % b`

Figure 2.7: CoffeeScript - JavaScript Aliases and Operators [36]

Existential Operator

This operator can be used instead of checking if a value is zero, the empty string or false. It always returns *true* unless its value is *null* or *undefined*.

```
var speed;

speed = 0;

if (speed == null) {
  speed = 15;
}
```

Listing 2.27: JS Existential Operator

```
speed = 0
speed ?= 15
```

Listing 2.28: CS Existential Operator

Classes, Inheritance and Super

CoffeeScript supports classes and inheritance, by introducing the keywords *class* and *extend*.

```
class Car
  constructor: (type) ->
      this.type = type;

  race: (speed) ->
    alert (this.type + "set speed to " + speed);

class RaceCar extends Car
  race: ->
    super 100

racecar = new RaceCar "Mercedes"
racecar.race() //Mercedes set speed to 100
```

Listing 2.29: CS Classes and Inheritance

Destructuring Assignment

With the destructing assignment it is possible to switch values or assign multiple values at once.

```
one = 1
two = 2

//switches the values of one and two -> one will be 2 and two will be 1
[one, two] = [two, one]

getNumbers = (number) ->
  return [number-1, number, number+1]

//will assign the following values: nr1=7, nr2=8, nr3=9
[nr1,nr2,nr3] = getNumbers 8
```

Listing 2.30: CS Destructuring Assignment

2.3.6 TypeScript

“TypeScript lets you write JavaScript the way you really want to. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open Source.” [18]

Developed by	*Microsoft*
First Appeared in	*2012*
Latest Stable Release	*1.0.1 / May 13, 2014*
Influenced by	*JavaScript, Java, C#*
License	*Apache License 2.0* [27]
Filename extension	*.ts*
Tags on Stackoverflow	*2484*
Repositories on GitHub	*434*

Table 2.2: TypeScript facts [92]

In the following paragraphs we will have a look at the features provided by TypeScript, but before, we will start with some general information which is necessary to understand the examples given in this section. For a full list of all the features TypeScript offers see typscript-lang.org [18].

General Information

As the name already states, TypeScript enhances JavaScript with types. Additionally it supports classes, interfaces, mixins and other useful features. But not everyone is satisfied with the features Microsoft is offering with TypeScript. The famous author of *JavaScript: The Good Parts* thinks that TypeScript is fixing a wrong problem.

“I think that JavaScript’s loose typing is one of its best features and that type checking is way overrated. TypeScript adds sweetness, but at a price. It is not a price I am willing to pay.” - Douglas Crockford [22]

Basic Types

TypeScript is adding the following types: *Boolean*, *Number*, *String*, *Array*, *Enum*, *Any* and *Void*. The first four types in this list we already know from JavaScript. *Enum* simply defines a list of possible alternative values. *Any* is used if the type of a variable can change. *Void* states that there is no type at all. Below we will give some short examples of these basic types written in TypeScript.

```
//Boolean
var isTrue: boolean = true;
//Number
var nr: number = 1205;
//String
```

```
var language: string = 'TypeScript';
//Array
var numbers: number[] = [5,12,88];
//Enum
enum Animal {Cat, Dog};
var animal: Animal = Animal.Dog
//Any
var x: any = 4; x = 'now string'; x = false;
//Void
function msg(): void {alert('no return');}
```

Listing 2.31: TS Types

Interfaces

Interfaces help to define, how a function should be called to work correctly. There are several ways how interfaces can be used with TypeScript. We will only show very basic examples below, to give an idea about the usage. Additional optional properties, extending interfaces and so on are possible as well.

```
function showMessage(obj:{message: string}){
  alert(obj.message);
}
showMessage({id: 12, message: 'Hello World!'});

//alernative interface defintion
interface ObjInt {
  message: string
}
function showMessage(obj: ObjInt){...
```

Listing 2.32: TS Interfaces

Generics

Generics are components that can work with variable types. The use of generics allows us to develop more reusable components. TypeScript adds this feature to JavaScript to prepare the language for large software projects, where reusable components play a huge role. We will give a short example of how to use generics in TypeScript below.

```
function identity<T>(arg: T): T {
   return arg;
}
var ret = identity<string>("TypeScript"); //ret will be of type string

//An other example for arrays
function identity<T>(arg: Array<T>): Array<T> {
   return arg;
}
```

Listing 2.33: TS Generics

Classes and Inheritance

Since most modern day developers feel more comfortable with object-oriented programming instead of functional and prototype-based approaches, TypeScript offers a built-in class system. Below is a short example of a basic class with inheritance as well as private modifiers using TypeScript.

```
class Car {
  private type: String;
  private speed: number;
  constructor(type: string){
    this.type = type;
  }
  accelerate(){
    this.speed += 1;
  }
  setSpeed(speed: number){
    this.speed = speed;
  }
}

class RaceCar extends Car {
  private tspeed = 100;
  constructor(type: string){
    super(type);
  }
  setTopSpeed(){
    super.setSpeed(this.tspeed);
  }
}
```

Listing 2.34: TS Classes and Inheritance

Modules

Building a large application, modularization is a very important topic. Therefore Microsoft has build a sophisticated system to achieve a flexible development. For a detailed overview about the various ways how modules can be created and organized please have a look at the TypeScript Handbook section Modules [13].

2.3.7 LiveScript

"LiveScript is a language which compiles to JavaScript. It has a straightforward mapping to JavaScript and allows you to write expressive code devoid of repetitive boilerplate. While LiveScript adds many features to assist in functional style programming, it also has many improvements for object oriented and imperative programming.way." [94]

Developed by	*Jeremy Ashkenas, Satoshi Murakami, George Zahariev*
First Appeared in	*2011*
Latest Stable Release	*1.2.0 / August, 2013*
Influenced by	*Haskell, JavaScript, F#, CoffeeScript*
License	*MIT License* [63]
Filename extension	*.ls*
Tags on Stackoverflow	*31*
Repositories on GitHub	*50*

Table 2.3: LiveScript facts [91]

This section will deal with the features of LiveScript. We will start with some general information about the preprocessing language followed by a list of features LiveScripts adds to JavaScript. For a full list of all the features LiveScript enhances JavaScript, visit livescript.net [94].

General Information

LiveScript is the JavaScript preprocessing language to use, if the need of functional programming is given, but it also offers improvements in the area of object-oriented programming as well as imperative programming. LiveScript, like CofffeeScript, tries to eliminate as much brackets as possible and makes use of whitespaces and tabulators instead. Not only the brackets are handled as in CoffeeScript, also the *function*-keyword gets replaced by the arrow symbol. Since LiveScript is an indirect descendant of CoffeeScript, they have a lot in common, but there are still some differences. For a full list of LiveScript features and the differences to CoffeeScript visit livescript.net [94]. In the next few paragraphs we will only have a look at features that we have not already described in the CoffeeScript 2.3.5 section.

Lists

Lists play a huge role in functional programming, therefore LiveScript supports different variations of defining lists. For example, commas are not necessary if the items are not callable. See the following examples for more details.

```
[5, 12, object.value, 'LiveScript'] //[5,12, object.value, 'LiveScript'];
[5 12 object.value 'LiveScript']    //[5,12, object.value, 'LiveScript'];

obj-list =
   * name: 'LiveScript'
     version: '1.2.0'
   * name: 'CoffeeScript'
     version: '1.7.1'

//this would result in the following JavaScript object

objList = [
   {name: 'LiveScript', version: '1.2.0'},
   {name: 'CoffeeScript', version:'1.7.1'}
];

//Some other list examples
<[ one two three ]> ++ [\four] //['one','two','three','four']
[\ha] * 3                      //['ha','ha','ha']
<[ one two three ]> * \|       //'one|two|three'
```

Listing 2.35: LS Lists

Ranges

Ranges are a candy way to iterate over some list. LiveScript supports different keywords to define the exact range. The keyword *to* includes the number while *til* means until a certain number but not including this one. To define a step it is possible to use the keyword *by*. Below some range examples are given.

```
[1 to 3]       //[1, 2, 3]
[1 til 3]      //[1, 2]
[1 to 6 by 2]  //[1, 3, 5]
[3 to 1]       //[3, 2, 1]
[to 3]         //[0, 1, 2, 3]

//Also iteration over chars is possible
[\A to \C]     //['A', 'B', 'C']

//Also the use of a variable is possible
x = 2
[1 to x]       //[1, 2]
[x to 0 by -1] //[2, 1, 0]
```

Listing 2.36: LS Ranges

Piping

Piping is used instead of nested calling, which helps to make the code more readable and better structured if used correctly. Below we will give some examples how piping is working in LiveScript.

```
[1 2 3] |> reverse |> head //3
reverse <| [1 2 3]         //[3,2,1]
```

Listing 2.37: LS Piping

Composing

Under composing we understand the possibility to create a function build by the composition of other functions. For this LiveScript offers two different operators:

- » Forward operator
- « Backward operator

To give a theoretical example:

- *(f « g) x* can be seen as *f(g(x))*
- while *(f » g) x* is equivalent to *g(f(x))*.

Below we will show a quick example of both operators in LiveScript.

```
add-two-times-two = (+ 2) >> (* 2)
times-two-add-two = (+ 2) << (* 2)

add-two-times-two 3 //(3+2)*2 => 10
times-two-add-two 3 //(3*2)+2 => 8
```

Listing 2.38: LS Composing

Property Access

This is a very useful feature, to access your properties. There are several ways to access the value of a property. Below we will give some examples of how to access a property.

```
[1 2 3 4][2]      //3
{a: 10, b: 20}.b //20

x = "LiveScript": [3 [5 test: 6]] //{"LiveScript":[3,[5,{"test":6}]]}
x.'LiveScript'.1.[0]              //5
```

Listing 2.39: LS Composing

Object Orientation

Not only functional programming is possible, also object orientation features are offered by LiveScript. Below is a simple example showing two classes with inheritance.

```
class A
  ->
    this.x = 1

  method: (num) ->
    this.x + num

class B extends A
  ->
    this.y = 2
    super!

  method: (num) ->
    this.y + super ...

b = new B
b.y         //2
b.method 10 //13
```

Listing 2.40: LS Object Orientation

2.3.8 Dart

“Dart is a new platform for scalable web app engineering. programming.way.” [41]

Developed by	*Google* [90]
First Appeared in	*2013*
Latest Stable Release	*1.5.3 / July 4, 2014*
Influenced by	*JavaScript, Smalltalk, Erlang, C#*
License	*BSD License* [66]
Filename extension	*.dart*
Tags on Stackoverflow	*3534*
Repositories on GitHub	*2149*

Table 2.4: Dart facts [88]

In the next few paragraphs we will discuss how Dart enhances JavaScript, starting with some general information about the language, followed by some of its features. A list of all features offered by Dart can be found on dartlang.org [41].

General Information

Dart has been engineered by Google to support large scale web application development. Though Dart can be compiled to JavaScript its actual goal is to replace it someday. The chances that this will really happen are looking pretty good, since the Dart community is growing bigger and bigger these days. Dart offers a large number of benefits like automatic optimizations of code, it also has it’s own Document Object Model (DOM) to fix JavaScript problems as well as problems with the browsers application programming interface (API). At the moment the only browser, that is able to run Dart virutal machine (VM) is Dartium. It is is a special version of the Chrome browser (both are developed by Google). To run Dart native in other browsers, the browser developers would have to implement the Dart VM into their browser versions.
Google Web Toolkit (GWT) is also a project of Google, developed back in 2006. With GWT it is possible to write Java code which gets cross-compiled to JavaScript. The difference between Dart and GWT is that GWT is based on an existing language, while Dart can be seen as a new language developed to suppress JavaScript and go for the leading position of web develop languages.

Entry Point

In Comparison with JavaScript or any other before mentioned preprocessing language, Dart offers an entry point where your application gets started. The *main* method optionally takes a list of arguments. This way it is possible to pass parameters to the program.

```
main() {}

//with arguments
main(List<String> args) {}
```

Listing 2.41: Dart Entry Point

Modularization

Dart offers the possibility to divide your applications into several libraries, which can later be included by using the keyword *import*. Below we will give a simple example of defining and using a library.

```
library animals;

class Dog {
  noise() => 'BARK!';
}
```

Listing 2.42: Dart define a library

```
import 'animals.dart';
var fido = new Dog();

import 'animals.dart' as pets;
var fido = new pets.Dog();
```

Listing 2.43: Dart use a library

Variables

Since Dart is dynamically typed as well as statically typed, variables can have a type, but it is not necessary. If a variable is not initialized with a certain value, it will hold the *null* value. Additionally variables can be declared final. Below we will give some examples showing these different kinds of variables.

```
String name = 'Alice'; var name = 'Alice';
var lastname; //null
int x; //null
final firstname = 'Bob';

// you can combine types and final
final String firstname = 'Bob';

// Trying to reassign a final variable raises an error
firstname = 'Alice';
```

Listing 2.44: Dart Variables

Collections

Dart offers not only lists, but also queues. The queues are treated after the first in first out (FIFO) principle. Additionally it is possible to use generics in Dart and combine them with these collections. The example below will give a quick overview.

```
var list = new List<String>();

var queue = new Queue();
queue.add('Dart');
queue.add('Lang');

print(queue.length);  // 2
queue.removeFirst();  // returns 'Dart'
print(queue.length);  // 1
```

Listing 2.45: Dart Collections

Strings

A very nice feature of Dart is the interpolation of Strings. It is possible to use variables directly in a string or even do calculations inside of a string assignment.

```
var name = 'Alice';
var greeting = 'My name is $name.';
greeting += 'I am ${12 + 5} years old.';
```

Listing 2.46: Dart Strings

Functions

It is not necessary to define a return type of a function. Additionally functions can be assigned to variables. Furthermore optional parameters as well as default values and names for parameters are possible.

```
helloWorld() {return "Hello World";}

//can be written as
String helloWorld() {return "Hello World";}

//a function without a return value will return null
helloWorld() {} //returns null

var upper = (text) => text.toUpperCase();
upper("hello world"); //"HELLO WORLD"

show(text, [text2]) => text2;
show("hello World") // will return null

String display(text, [text2 = "system"]) {
  return "${text} from ${text2}";
}
```

```
display("hello");           // hello from system
display("hello", "Alice"); // hello from Alice

String display(text, {sender: "system"}){
  return "${text} from ${sender}";
}

display("hello", sender: "Bob"); // hello from Bob
```

Listing 2.47: Dart Functions

Object Orientation

Dart also supports object oriented programming. Below we will give an example, showing how classes, subclasses, inheritance and constructors can be used.

```
class Person {
  String name;

  Person(String name) {
    this.name = name;
  }

  String greet() => 'Hello, $name';
}

class Employee extends Person {
  num salary;

  Employee(String name, num salary) {
    this.salary = salary
    super(name);
  }

  void grantRaise(num percent) {
    percent /= 100;
    salary += (salary * percent).toInt();
  }
}
```

Listing 2.48: Dart Object Orientation

Operator Overloading

A very useful feature of Dart is operator overloading. It is used to assign specific functions to operators depending on its operands. A short example how to use the operator overloading is given below.

```
class Money {
  num amount;

  Money(num amount){
    this.amount = amount;
  }

  Money operator +(Money other) => new Money(amonunt + other.amount);
}

main() {
  var money = new Money(5);
  var money = new Money(10);
  var more = money + money; // new Money instance with amount 15
}
```

Listing 2.49: Dart Operator Overloading

Document Object Model

> "... is a set of platform- and language-neutral application programming interfaces (APIs) that describe how to access and manipulate information stored in structured XML and HTML documents." [58]

As stated in the general information paragraph, Dart uses its own DOM, and therefore also its own DOM functions to find and create elements.

```
querySelector('#main'); //Find one element by id
querySelector('.visible'); //Find one element by class
querySelectorAll('.visible'); //Find many elements by class
querySelector('div'); //Find one element by tag
querySelectorAll('div'); //Find many elements by tag
querySelector('[name="form"]'); //Find one element by name
querySelectorAll('[name="form"]'); //Find many elements by name

//Find elements by combination
querySelector('#main').querySelector('div').querySelectorAll('.visible');
//Find elements by combination
querySelectorAll('#main div:first-of-type .visible');

var element = new Element.tag('div'); //Create an element
var element = new DivElement(); //Create an element

//Create an element and set its contents
var element = new Element.html('<p>A quick brown <em>fox</em>.</p>');
```

Listing 2.50: Dart DOM

CHAPTER 3

Related Work

"Research is what I'm doing when I don't know what I'm doing."

— Wernher von Braun,
"Father of Rocket Science"

This chapter gives an overview over existing research work in the area of comparing coding languages in general as well as the comparison of JavaScript preprocessors, and finally also in the scope of risk factors in the web project development.

3.1 Types and programming languages

Types and Programming Languages by Benjamin C. Pierce [70] is a well-known book on type systems. It isn't directly related to our languages but it describes a large number of language features (static/dynamic typing, inheritance, polymorphism, classes, etc.) that we used as criteria for our evaluation. Though it's a very general book, we used it a lot to specifically define what each language should provide.

3.2 Comparing Coding Languages

Comparing coding languages is a very difficult task, since there are many factors, some subjective, that we have to consider in such a comparison. There are many paradigms and many language features that influence each other in complex ways. This makes a task like this very complex. Our four languages are very similar, which makes our work a bit easier, but our evaluation will still contain some opinion. In this section we want to take a look at how others have compared languages before.

3.2.1 An Empirical Comparison of Seven Programming Languages

In *An Empirical Comparison of Seven Programming Languages* [71], C, C++, Java, Perl, Python, Rexx and Tcl were compared. A program called phone-code [71, p. 25]) was used. The requirements were the same for each of the listed languages. Additionally multiple implementations from different developers were analyzed. Some of the investigated aspects were runtime, programming effort, program length, memory consumption and others. Furthermore the languages were divided into scripting languages (Perl, Python, Tcl, Rexx and Tcl) and the conventional programming languages C, C++ and Java. Some of the results, after 80 implementations of the phone-code problem were analyzed in the above mentioned seven languages, were [1] [71, p. 29]):

- Writing the program in a scripting languages takes only half as much time as writing it with the more conventional ones, but the scripting languages consumed twice as much memory.
- Java even consumed three up to four times as much memory as the C languages.
- The initialization phase of the phone-code problem consists of reading a file and creating entries from this data. Here C and C++ were the fastest languages, Java needed about four times longer and the script languages were the slowest and needed about ten times longer than C.
- The searching phase however brought some other results. Here C and C++ run only about twice as fast as Java, and the scripting languages were faster than Java but slower than the C languages.

To summarize the obtained results, Lutz Prechelt states:

> "(...) Java's memory overhead is still huge compared to C or C++, but its runtime efficiency has become quite acceptable. The scripting languages, however, offer reasonable alternatives to C and C++, even for tasks that must handle fair amounts of computation and data. Their relative runtime and memory consumption overhead will often be acceptable, and they may offer significant advantages with respect to programmer productivity, at least for small programs like the phone-code problem."
> [71, p. 29]:

3.2.2 A Comparison of Object-oriented Programming in Four Modern Languages

Robert Henderson and Benjamin Zorn compared the languages Oberon, Modula-3, Sather and Self in 1994 [34]. The languages aren't widely used and the paper is a bit old, but the criteria they used are still valuable. They implemented a simple application for handling university personnel files. The database consisted of students and teachers that were both subclasses of personnel.

[1] The observations are only valid for the phone-code problem and are not reliably adaptable arbitrary other problems.

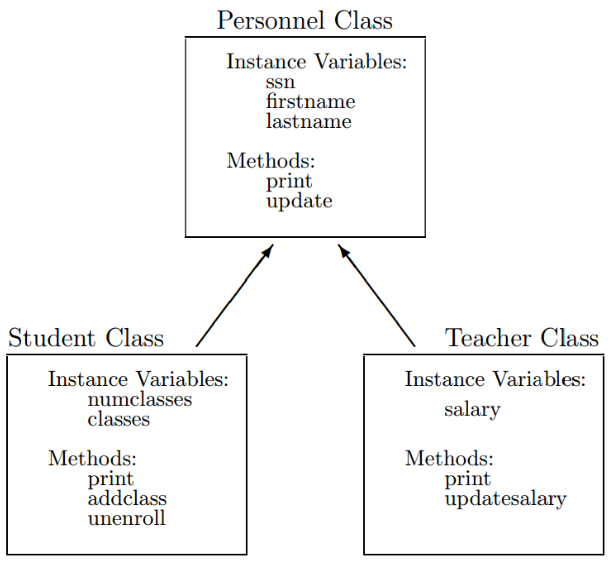


Figure 3.1: Henderson's and Zorn's example applications [34].

They compared the languages in a qualitative and in a quantitative way. First they looked at how each language with the criteria of inheritance, dynamic dispatch, code reuse and information hiding. After that they did a quantitative analysis by comparing execution time, code size and compile time for each implementation. They found that Oberon was a very small language, while Modula-3 was large and complex with many features, while both only really implemented object-orientation halfway. Sather and Self were fully object-oriented and simple, with Self using prototyping and cloning. The languages aren't really relevant today anymore, but language complexity and the degree of object orientation are still issues in our examination of our preprocessors. Compile time and executable size have also become less important over time, but the kinds of qualitative features that Henderson and Zorn listed are still valuable and we will use similar ones.

3.2.3 A Comparative Study of Programming Languages in Rosetta Code

Rosetta Code is a repository of solutions for common programming tasks written in many different languages. In this study [65] 7078 solution programs were used to solve 745 tasks using 8 different programming languages (C, C#, Go, Java, F#, Haskell, Python and Ruby). The study tried to compare various features like the runtime of programs, the memory usage, failure proneness and size of the executables. The following results were concluded from *A Comparative Study of Programming Languages in Rosetta Code* [65] :

- Procedural and object-oriented languages are not as concise as functional and scripting languages are.

- C was the fastest language in this comparison and very good at handling large inputs, but using small inputs also interpreted languages were competitive.
- Interpreted or weakly-typed languages are more prone to runtime failures compared to compiled and strongly-typed languages since failures can be found earlier in compile time.

3.3 JavaScript Preprocessors

In *Evaluation of Alternate Programming Languages to JavaScript* [1], Aansa Ali has compared CoffeeScript, TypeScript and Dart, by developing an example web application in all of the three languages, among other things, as his master of sciences thesis at the Tampere University of Technology.

He started by defining scripting languages in general and described JavaScript and its paradigms. Furthermore he also showed some of the major drawbacks of JavaScript. Then he listed a few preprocessors that solve some of the given drawbacks. In the end he described CoffeeScript, Dart and TypeScript im more detail.

Regarding the comparison, there were only made two very basic ones.

- *Performance comparison*
  Here a comparison between the Dart2js and Dart VM is given. As benchmarks, *Deltablue* and *Richards* were used. Both ran a suite of tests and trying to measure the performance of the code in large web applications. The chart shows that Dart VM is outstripping Dart2Js, especially in the *Richards* benchmark.

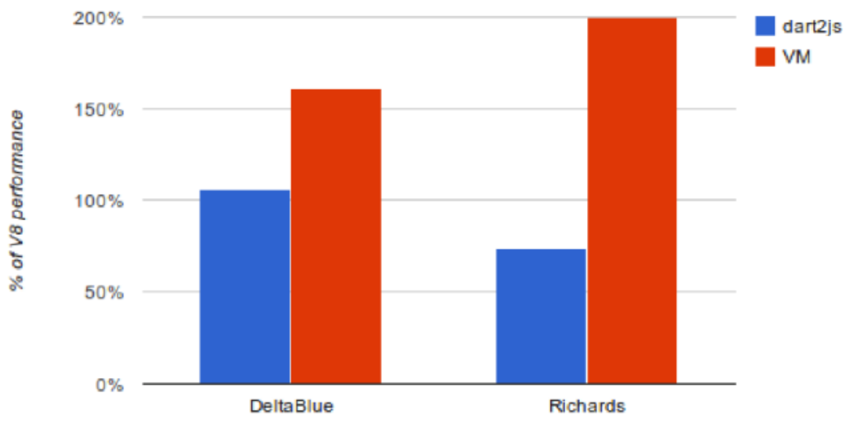


Figure 3.2: Benchmark Dart2js versus Dart VM [1]

- *Robust comparison*
  A very simple textual comparison has been made, where the strict mode [2] of ECMAScript 5 is explained. Concluding that CoffeeScript does the checks itself at compile-time, while Dart has its own *isolate*-libraries to achieve more security.

A basic TODO web application was developed four times, each time with the same functionality: once in JavaScript and then once in CoffeeScript, TypeScript and Dart. After writing the application in these languages, Ali came to the following conclusions [1, p. 56]:

- **JavaScript** has some major drawbacks which are the reason the preprocessors came into existence.
- **CoffeeScript** has a bad IDE support, but provides syntactic constructs to reduce unnecessary code and should be used to develop medium to large scale applications.
- **Typescript** has a much better IDE support thanks to Visual Studio and can be used to develop enterprise applications. Debugging is also made very easy using Visual Studio's built-in functionality.
- **Dart** also has an excellent IDE support, since it has its own editor and browser to run applications. Because Dart is a clean and modern language it should definitely be considered when thinking about developing a large scale application.

An other very interesting question is asked in the end [1, p. 56]:

> "Which scripting language is best and should be used to develop the new web application?"

Which is followed by the statements below:

> "The answer to this question varies. For the enterprise applications, website development and if you are using the Microsoft tools for development, TypeScript would be a good choice, it uses the JQuery and Ajax for dynamic updation. A major part of benefit of TypeScript comes from the tooling within Visual Studio."

This is a very simple answer and in contrast, we will try to investigate more deeply, taking a more detailed look at the language features and web application projects. Since the answer to our questions depend on many parameters, we will try to identify and list the ones we think are more important and evaluate them systematically.

[2]This is a mode where more conditions lead to errors and warnings than in regular mode

3.4 Web Project Development

In software projects there exist many factors which are relevant for a good outcome. The results of such IT projects also depend on many so-called risk factors. Risk factors are able to influence the outcome of a project in a negative way, under certain circumstances, so it is very important that they are being treated accordingly. Since web projects are a specific type of software projects, specific appraisal factors as well as specifc risk factors are also relevant.

In *Factors that Affect Software System Development Project Outcomes: A Survey of Research* [60], Laurie Mcleod and Stephen G. Macdonell addressed factors that influence the outcomes of software project development processes using a classificatory framework. This framework separated four major areas, where each area was again split up into several subareas:

- **Project Content** is about the project itself and includes factors like project characteristics, project scope, project goals, as well as the used technology and resources that are needed.

- **Development Processes** includes the requirements determination, project management, user training and management of change.

- **Institutional Context** describes the factors related to the development organization itself and its environment.

- **People and Action** includes people who are involved in the project like developers, users, management and many other project related persons and groups as well as their actions and relation between them.

Figure3.3 illustrates the above mentioned factors. Additionally some examples in the respective dimension are given. All of the items shown there play a role in the outcome of a project. The subitems can be further dividied into more detailed properties, as shown in Figure3.4.

We will use only some of the properties/attributes below in the next chapter, since not all of them are related to the language one chooses. We will also group together some of the items that are dependent on each other for simplicity and a better understanding.

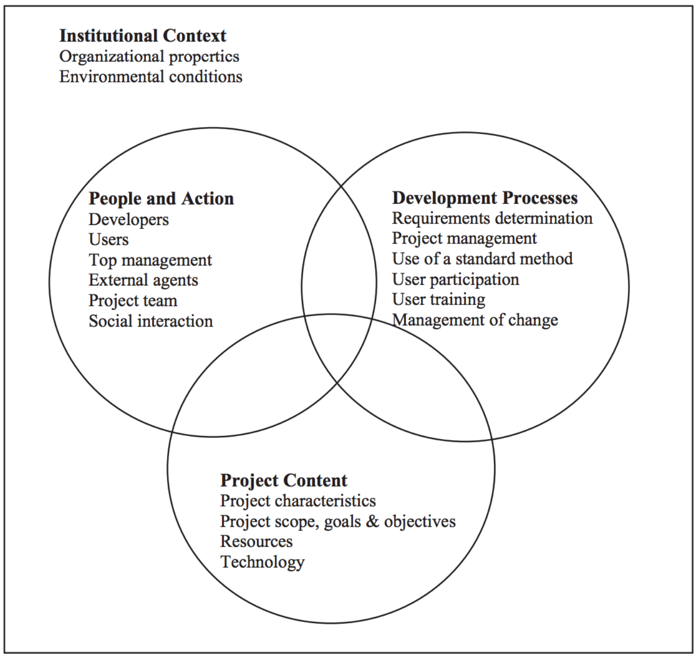
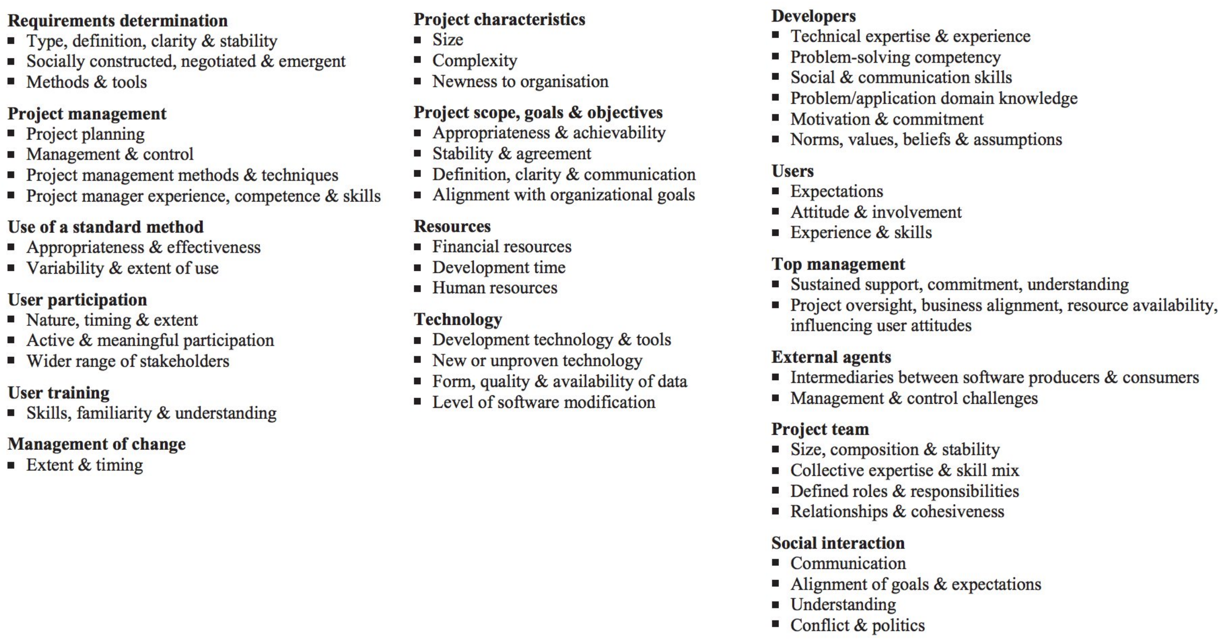


Figure 3.3: Factors that affect software project outcomes [60, p. 5].



Requirements determination
- Type, definition, clarity & stability
- Socially constructed, negotiated & emergent
- Methods & tools

Project management
- Project planning
- Management & control
- Project management methods & techniques
- Project manager experience, competence & skills

Use of a standard method
- Appropriateness & effectiveness
- Variability & extent of use

User participation
- Nature, timing & extent
- Active & meaningful participation
- Wider range of stakeholders

User training
- Skills, familiarity & understanding

Management of change
- Extent & timing

Project characteristics
- Size
- Complexity
- Newness to organisation

Project scope, goals & objectives
- Appropriateness & achievability
- Stability & agreement
- Definition, clarity & communication
- Alignment with organizational goals

Resources
- Financial resources
- Development time
- Human resources

Technology
- Development technology & tools
- New or unproven technology
- Form, quality & availability of data
- Level of software modification

Developers
- Technical expertise & experience
- Problem-solving competency
- Social & communication skills
- Problem/application domain knowledge
- Motivation & commitment
- Norms, values, beliefs & assumptions

Users
- Expectations
- Attitude & involvement
- Experience & skills

Top management
- Sustained support, commitment, understanding
- Project oversight, business alignment, resource availability, influencing user attitudes

External agents
- Intermediaries between software producers & consumers
- Management & control challenges

Project team
- Size, composition & stability
- Collective expertise & skill mix
- Defined roles & responsibilities
- Relationships & cohesiveness

Social interaction
- Communication
- Alignment of goals & expectations
- Understanding
- Conflict & politics

Figure 3.4: Properties/attributes influencing the corresponding area [60] (Columns from left to right: development processes, project content, people and action)

CHAPTER 4

Selecting JavaScript Pre-Processing Language

"Get your facts first, then you can disort them as you please."

— Mark Twain,
American author and humorist

In this chapter we will define the criteria and collect the data that are needed to compare CoffeeScript, TypeScript, LiveScript and Dart rigorously instead of following popular opinion. To do so, we will go through the steps listed below. Additionally we will apply a cost-benefit analysis in Section 4.4 on the gathered data, since we need an analysis form which supports a number of different attributes as well as individual factors. The cost-benefit analysis will give us a table where one axis contains our four preprocessors and the other axis contains a list of language features clustered into related groups. The corresponding cells will represent the evaluated value. The data given by this analysis will be used to find the best-fitting preprocessor for a given web project.

- Determine important web project properties for a good project outcome. Therefore we will have a look at related work that points out which factors are affecting such a software development project outcome (Section 4.1).
- Then we will further investigate the given web project properties and see which pre-processing language features are related to those properties and how they influence them (Section 4.2).
- Given that list of pre-processing language features, we will then describe and measure them in detail. We will again have a look at related work, to see how features can be properly measured and compared across languages (Section 4.3).

Note that, in this thesis, we focus on **web projects**, so the properties may differ from other types of software projects. They are not applicable to all types of projects in general.

4.1 Determining Important Project Properties

In this section we will try to find the most important web project properties for a good project outcome in regard of the used preprocessing language. That means we will try to find out, which properties, of such a type of project, have a big influence on the result. We will take the properties from *Factors that Affect Software System Development Project Outcomes: A Survey of Research* [60], as foundation for our examination.

In the above mentioned survey, a framework had been created by Laurie Mcleod and Stephen G. Macdonell, which separates the following four areas:

- **Project Content** (Section 4.1.1)
- **People & Action** (Section 4.1.2)
- Development Process
- Institutional Context

We will only concentrate on the *Project Content* and *People & Action* area, because we do assume that the domains of *Development Process* and *Institutional Context* have no direct influence on the choice of the preprocessor. They would be the same no matter which language we choose.
In the next subsection we will describe some of the factors from the two chosen areas, and why they can be rated as important.

4.1.1 Project Content

In this section we will describe factors in the area of project content which are relevant for a satisfying project outcome and why they are that important to make a successful project possible.

> "(...) includes factors that are typically considered as properties of the software systems project itself, including its dimensions, scope and goals, the resources it attracts, and the hardware and software used in development." [60, p. 4]

4.1.1.1 Project Characteristics

The outcome of a project is dependent on some different project characteristics. In the following paragraphs we will have a detailed look on how the size of a project, the complexity as well as the newness of a project to the organization and how they influence the outcome.

Size

One of the most important project characteristics is the size of a project, since it influences many other areas. Depending on the size of a project, the development time and the planning time have to be adapted, as well as the financial and human resources calculated. All of these factors will be discussed in the next subsection Resources 4.1.1.2.
The size of a project is one of the key points influencing the project outcome. If the size is underestimated the project will most likely fail or at least leave the customer dissatisfied with the result, which might be seen as an unsuccessful outcome as well, leading to the project being stopped during the development phase. As a consequence this customer may not ever give a project to that company again, or even worse, that customer may talk about the bad experience with others and damage the reputation of the project team or the whole company.
In the long term, getting negative project outcomes or not delivering anything at all will destroy the company. Besides the costs of the project, many companies depend on their reputation and should be interested in long-term relationships with their customers. Examples are consulting and well-paid maintenance contracts which many companies live off of.

Complexity

The complexity of a project consists of many different factors, starting from the various number of technologies that have to be used, to the number of different external systems the software has to connect with in order to work correctly. The type of software also plays a huge role, for instance, software for sending simple chat messages will be not as complex as a software that is responsible for controlling some military defense systems.
Another important fact, concerning the complexity of a project, is the planning time. If a project has a short planning phase or no planning phase at all and/or an unexperienced project manager, the complexity of a project can be underestimated very easily. This will result in a miscalculation of the needed resources to gain a good project outcome. Since not every project is completely elaborated in the beginning, before the development starts, complexity problems can also occur during the implementation process. It is very important to handle this emerging complexity problems as early as possible to avoid getting into a more and more technical debt, which will get more and more difficult to handle.

Newness to Organization

The newness of a project is very tightly coupled to the complexity of a project, since in a new type of project, the complexity often gets underestimated. If the organization did similar projects in the past, it will calculate the resources it needs in a similar way for the current project. In new types of projects this is not that easy to do so, since the rules from other past projects might not apply. A new calculation system has to be applied and even adapted several times due to many occurring changes or dead ends. This process can be very frustrating, but should be considered extremely carefully, since if an error is made at the very early stage of a new project might already mean the project's end before it has even really started.

4.1.1.2 Resources

The resources needed for a project are essential, and should be calculated as exactly as anyhow possible. This is mainly the task of the management. The three most important resources in each project are the financial resources, the development time and last but not least the human resources which are usually the most expensive resources in the software development process. In the following paragraphs we will take a closer look at these resources.

Financial Resources

This type of resources might be the most important one, since a wrong calculation at this type of resource might cost the most money and all other resources can be compensated with this factor. For example, if the complexity of a project has been underestimated and therefore the project seems to go beyond the deadline, it would be easy to hire more developers that can focus on a certain problem, to stay in the deadline if enough money is available and placed back for a situation like this.
Financial resources are also very important in the area of hardware and software acquisition. Hardware for a large number of developers and the needed servers as well as the whole IT infrastructure including network systems can be very expensive and often have to be pre-financed by the software companies. Also the costs for software licenses needed for development can be extremely costly and have to be considered.

Development Time

Development time is a very crucial factor in each software project, normally the management tries to minimize the time as much as possible while the project team, the developers and other executive persons, try to maximize the given time. So it is really important to find an agreement,which both parties are okay with. Development time can be decreased by hiring more personal which concludes more financial resources and/or more qualified personal which again increases the need for financial resources. If the development time is calculated too low, the personal might deliver bad work and therefore the customer might not be happy. So the development time should not be decided only by the management but rather in consense with the development team to get the most probably time that is really needed to finish the project with a good outcome, to make the customer happy. Once the customer is happy the management then might be happy as well.

Human Resources

This resource is the most expensive factor in general. Comparing the hardware and software expenses with the financial resources needed for the personal, the expenses for hardware as well as software might even seem negligibly. Though human resources are very expensive, saving at this factor might cost the company way more money than it would, paying a little more for a more qualified person. If a developer achieves twice as much as an other developer but costs only one and half time more, the company should definitely think about hiring the more expensive developer, because the company will benefit from this decision all along the way though the monthly expenses are higher, the generated work will be worth way more. Also higher paid developers may be much happier with their job, which results in a bigger motivation which again may end up in also working for company projects in their freetime.

4.1.1.3 Technology

The technologies used in web projects play a very huge role, since using the wrong technology for a project may end up in a disaster, due to the big number of various technologies existing these days, this happens more often. To prevent this from happen, to use the wrong technology, an evaluation phase should be done before the start of the development process, as part of the planning phase. In the next few paragraphs we will discuss why it is important to use the right development technology as well as development tool, how the form, quality and availability of data influence the development and why new and unproven technologies might not be the wisest choice.

Development Technology & Tools

It is crucial to choose the right development technology for a project. By choosing the wrong technology it is possible that problems might occur that would not have been there if the suited technology had been used. Not only the technology itself plays a role, but also the developers using that technology. Developers for a certain technology might be found more easily than for some other technology. This should also be considered, since a personal loss can happen at any time, which might influence the development progress in a negative way. It will also be in the interest of the customer, that the used technology is not that outdated but to have a positive future outlook using it.
Development tools are also very important, since they can increase productivity immensely but also reduce it to a minimum, if wrong tools are used or configured badly. So investing some time at the beginning of a project to evaluate the perfectly fitting tools, supporting the development process, and configuring them will definitely pay off. Though the license costs of professional development tools should not be underestimated, but in the web development area there are many very good tools to download for free.

Form, Quality & Availability of Data

In bigger projects it is often necessary to work with external data from other companies, or other products and integrate that data into the new system. To work correctly it is important that the form of the data is represented in a correct form.
Not only the format but also the quality of the data is relevant, since data delivered in the fitting form but not underlying a certain quality might produce a lot of work that might not have been considered during the planing phase.
The availability of data plays a huge role as well. If the data is not available at a certain moment, where it is needed by the development team, might represent a huge problem and produce much overhead.

New or Unproven Technology

It could be very risky to use new or unproven technologies in a new project. If the technology does not work as expected, the project might be blown off before it even started due to an impasse. An other problem with new technologies is, that there might be not enough documentation to find and not many people working with it yet, so there could be only few experience reports available. Comparing a new or unproven technology with a field-tested technology might be a little hard, due to the fact that there may only be a few feedbacks found from other developers.

4.1.2 People & Action

In this section we will describe factors in the area of people & action which are relevant for a satisfying project outcome and why they are that important to make a successful project possible.

> "Finally, as can be drawn from these content, context, and process descriptions, consideration needs to be given to the various people, both individuals and groups, who are involved or interested in the software systems development project. Their characteristics, actions, interactions and relationships shape the development trajectory and project outcomes in multiple ways, so an understanding of their roles and actions during system development is also necessary." [60, p. 4]:

4.1.2.1 Developers

Developers are the most important factor in each software project. Without good developers the project might not come to a successful end. Therefore the choice of a developer should not be made quickly but be well thought off. In the following paragraphs we will take a look at the technical expertise and experience of developers, the problem-solving competency and last but not least the motivation and commitment of a developer.

Technical Expertise & Experience

Developers should have much experience with the technology they are working with. This will make them more productive. Experienced developers will not get into some troubles that inexperienced developers might have to go through, so this will save much time. Also skilled developers might handle problems in a different more reliable way. Problems that might reoccur are treated way better by developers who already have experience in that area. Therefore experienced developers should also be preferred even if they want a higher salary, but paying this upcharge will definitely pay off.

Problem-solving Competency

Good developers are able to solve problems in a general way, which means their solutions are easily adoptable to problems that are relative to the given problem instead of figuring out a solution that might just be working for a specific situation. Writing code in such an adoptable way will save much time during development.

Motivation & Commitment

Motivation and commitment should not be underestimated. Motivation plays a huge role in a developers life. It can boost the performance enormously. Developers might even work on the project in their freetime, if they are really committed to what they are doing, because it does not feel like work, but instead it might even be a lot of fun for them. Motivated and committed people will also stay longer at work if something has to be done at the same day and they will also have a lower sickness absence rate due to the fact that they like what they are working on. Another point is that these developers might also look for different and new technologies in their freetime and bring the so evaluated results back into work, which is a very important fact for the companies to stay up to date and be prepared for the future.

4.1.2.2 Project Team

The project team consists of a number of people, where each person has a specific role and should give their best to fulfill the role requirements in order to achieve the best outcome for the team. In the next paragraphs we will describe how size and composition influences a project teams outcome, why skill mix matters, and why roles as well as responsibilities should be well defined.

Size, Composition & Stability

The size of a project team should be chosen with care, since a project team, which consists of too many people will complicate the communication sooner or later, but if a project team might consist of only very few people a loss of a group member might have catastrophic consequences under certain circumstances. Not only the size but also the composition of a team is important for a good outcome. The team members should be sympathetically so they like working together to achieve the best performance possible. If the members will like each other, this might boost the progress of a project enormously, because even in very small projects the necessary communication should not be underestimated.

Collective Expertise & Skill Mix

It is a wise decision to mix people with different skills and different approaches to get things done. This way it is most likely to find a better solution, then just depending on one opinion. A problem only will occur if people are not accepting other opinions, and think their point of view is always the proper way to do it. That might end up into a conflict, which could have a bad influence on the team as well as on the project's progress, but in general it is suggested to assemble a team with people that have different characteristics and skills.

Defined Roles & Responsibilities

It is very important to define roles as well as responsibilities and assign them to the proper persons. If roles are not well defined and the corresponding responsibilities are not clear. Sooner or later it will come to conflicts between the team members, because they might blame others if something went wrong, even if it is not that other person's fault at all.

4.2 Finding the Related Pre-Processing Language Features

In this section we will start by defining what pre-processing language features are. After that we will try to find the pre-processing language features that are related to the important project properties given in the section *Determining Important Project Properties* 4.1.

4.2.1 Pre-Processing Language Features

By pre-processing language features we refer to a set of certain characteristics which are directly related to the given pre-proecessing language. Some of this features are related to the syntax and semantic of the code, some others are related to the environment of the inspected language. For example, the feature *modularity* would be directly related to the language while the feature *development tool support* is a feature in the environmental area of the language.

4.2.2 Project Properties and their Related Language Features

Here we will investigate the pre-processing language features related to the given project properties, in the area of project content as well as in the area of people & action, given from the section before. Since we are comparing very similar languages we have to find an abstraction layer to map the above given project property groups that are relevant for this comparison. Due to the fact that the output of all four investigated languages is functionally the same, we have to do the comparison on a higher level. We create the following abstraction groups that are important for a good outcome based on the project groups above: *Financial Resources*, *Code Quality* and *Development Time*

Financial resources as well as development time can be directly mapped to the the project property group resources in the area of project content, while code quality consists of multiple factors, like the skill and expertise of a developer which can be mapped to the developers project property group in the area of people & action as well, but also the used development tools can influence this factors. The size and complexity of a project are also playing a role concerning the code quality among some others.

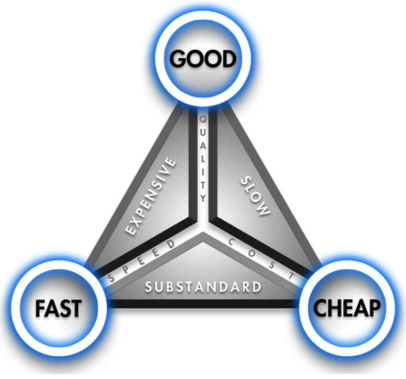


Figure 4.1: Project Triangle [55]

Therefore these three factors are influencing each other and are crucial in software projects. The triangle above visualizes the dependencies between those three, where the area inside the triangle stands for a good project outcome and therefore the size of the area must not be changed, but stay the same. For example if the goal is to achieve a good code quality with a little amount of money the development time will increase. An other example would be to finish the project in a short period of development time with a low budget, then the code quality will suffer. Last but not least, if the code quality should be as good as possible while the development time is very small the expenses for this project will increase.

As can be concluded from this triangle we have two different groups of interests. On the one hand there is the management which is responsible for the financial resources and interested in the depending development time. The goal of the management is to use as little financial resources as anyhow possible but at the same time decrease the development time. On the other hand we have the development team. Their goal should be to produce the best code quality possible and therefore try to increase the given development time.

As can be concluded from this, we have two groups, where the management has a conflict in its own goals, and one shared conflict with the development team, the development time.

- goals of the management

  minimize the financial resources

  decrease the *development time*

- goals of the development team

  increase the code quality

  increase the given *development time*

- shared conflict

  development time

This means that our cost-benefit analysis template given in the next section should be used by both groups and not only be applied by one of them. For one reason, because the management will not be able to rate the code quality features properly, while the development team might not be able to rate the financial resources properly. The other reason would be that they will have to find accordances concerning the rating of the development time.

In the next paragraphs we will have a detailed look at each of the before mentioned abstract project properties groups and their related language features. Of course the explanations in the following paragraphs only reflect my personal experience in software projects as well as the environment and could be seen different by anyone else.

Financial Resources

Financial resources might be the most important factor to the management. Most of the decisions made by the managment are based on that point. Since we have similar languages to compare, there are not many points that differ in terms of finance. For example, the hardware that is needed to run the systems is likely to be the same so there is no big difference in that area. This also holds for development tools and related software that is used. Therefore we will only take a look at two groups of items that might influence the managements decision.

On the one hand we have the costs for the developers. Here we can distinguish between *employed developers* and *freelancers*. We also should take into reference the *availability* of developers which are working with these languages. The cost of freelancers and the availability of these developers are important factors, if a developer leaves the team and replacement is needed shortly. It also should be considered that if the availability of developers, working with a specific language is low, that this might be a problem for the replacement process. Also if developers for a certain language are rare, their honorar might be higher than those of developers that are working with a widespread language and therefore are not that rare.

On the other hand we have kind of a risk factor. We divide this factor into two sub topics, the *popularity* and the *trends* factor. If a language is more popular than an other, there might be more information and examples to be found online, and the community will be bigger, which is always an important point during development. Not only the popularity of language is important, which shows the state of the language at the moment, but also the trend of such a language is crucial for the management, which will give a look into the future of the language. No management wants to invest time into a project written with a language which might be outdated in the near future.

All decision concerning financial resources should be made by the management. The development team should not be involved in any of this decisions, since normally they do not have enough knowledge in this area.

Code Quality

The code quality is related to many different factors. *Static typing* for instance helps to get cleaner code, due to the fact, that variables are having the same type all the time and not changing it, during runtime. Also the use of *interfaces* increases the code quality because they separate specification from implementation. This brings us to *object orientation*, which plays a huge role in modern day development and makes code more structured. The possibility to use *mixins* as well as *generics* also can improve the code quality because of the reuse of already written code. Some languages also offer *operator overloading* to increase the quality. Last but not least *modularization* is a crucial feature to increase the code quality and helps getting better structured code. All decisions about the code quality should only be made by the development team and not by the management, in contrast to the financial resources described above. Since the code quality is a very complex topic and the management generally has no skills concerning this topic.

Development Time

The development time is a shared conflict between the management and the development team. The development team tries to increase the given time to produce the best code quality possible, while the management tries to decrease the development time as much as possible to save development costs. Therefore the decision made concerning the development time should not be made by the management alone, and also not by the development team alone, but should be made together. If one group will make the decision in this area on their own, most likely there will be conflicts occurring during the development process between the management and the development team. For example, if the management goes solo on this topic and sets too harsh deadlines, the development will not be able to stay in the given boundaries or the set code quality might not be fully fulfilled. We will divide the development time into three related sub topics, and describe them below.

An important factor concerning the development time is the quality of *online documentation* as well as the quality and number of online examples. Having a good documentation, that includes the API as well as getting started tutorials helps a lot during the learning phase and the development process.

Also the used *development tools* can make a big difference concerning the development time. An Integrated Development Environment (IDE) might support a specific language better than some other. Therefore, having a good development environment, fitting to the used language is a crucial factor. Features of such tools like syntax highlighting, auto-completion and integrated help can speed up the development process.

Some language-features also have effects on the development time. Dynamic typing for instance makes rapid development possible, because the developer does not have to care about any type conversions or type definitions. An other important point that is improving the coding speed is the *reuse of code*. By reuse of code we refer to modularization, classes, inheritance, generics etc. These reuse features play a huge role in modern day development, and also help to make the code more structured if used correctly.

4.2.3 List of Features

To summarize the features given in the prior descriptions, we will use the following criteria, grouped by the three abstract layers we introduced in Section 4.2.2.

- Financial resources (management) (Section 4.3.1)
  - Developers
    - Costs of employed developers (Section 4.3.1.1)
    - Costs of freelance developers (Section 4.3.1.2)
    - Availability of developers (Section 4.3.1.3)
  - Risks
    - Popularity (Section 4.3.1.4)
    - Trends (Section 4.3.1.5)
- Code quality (development team) (Section 4.3.2)
  - Language features
    - Suppport for static typing (Section 4.3.2.1)
    - Object support (Section 4.3.2.2)
    - Classes (Section 4.3.2.3)
    - Interfaces (Section 4.3.2.4)
    - Inheritance and mixins (Section 4.3.2.5)
    - Generics (Section 4.3.2.6)
    - Modularization (Section 4.3.2.7)
    - Operator overloading (Section 4.3.2.8)
- Development time (management & development team) (Section 4.3.3)
  - Language features
    - Code reuse (Section 4.3.3.1)
  - Tool support
    - IDE support (Section 4.3.3.2)

This list of features will be the foundation for our evaluation. In the next subsection, each feature will be described and the way of measuring it will be explained. After that we will compare each feature across our four investigated preprocessing languages.

4.3 Describing and Measuring the Features

In this section we will go over all features listed in the previous subsection. For each feature we will give a short description, followed by an explanation on how we will measure it. After that we will compare this feature across our four preprocessing languages. The outcome of this subsection is a ranking of the languages for each investigated feature. The ranking will go from 0 to possibly 4, related to our four investigated languages, where 4 indicates the best rank and 0 indicates that the language doesn't support the feature at all. If two languages support a features equally well, they will share the same rank.

4.3.1 Financial resources (management)

We start by evaluating the costs and the availability of developers. Language features do not matter much to a company if it cannot find people who can program in it at a reasonable price. In addition, we should be reasonably sure that the language is future-proof and that we will be able to find developers for it some years from now as well.

4.3.1.1 Costs of Employed Developers

The costs of employed developers are a crucial factor in the area of financial resources and should be calculated as exactly as anyhow possible by the management.

Measuring Method

For our comparison we will investigate the price of developers, by having a look at the employment-related search engine indeed.com. As reference location we will use San Francisco, CA. Lower salaries are obviously ranked better.

Comparison

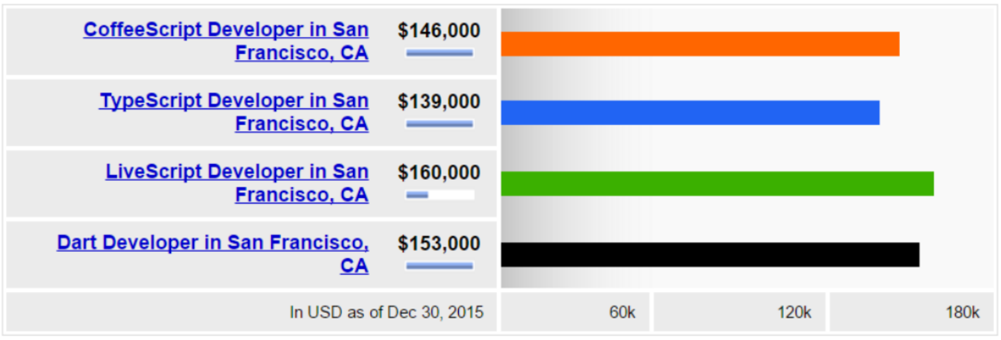


Figure 4.2: Average salaries in San Francisco, CA [49].

In that list, we can see that LiveScript developers earn the highest salaries, followed by Dart developers. Close behind are CoffeeScript and TypeScript developers.

Language	**Average salary in SF (CA)**
LiveScript	*$160,000*
Dart	*$153,000*
CoffeeScript	*$146,000*
TypeScript	*$139,000*

Table 4.1: Average salary in San Francisco (CA) ordered by rank.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	3	4	1	2

Table 4.2: Ranking for the cost of employed developers.

4.3.1.2 Costs of Freelance Developers

Some developers do not like to be employed as regular employees but rather prefer freelance work. This is why we also have a separate look at this type of developer and how much they cost the company.

Measuring Method

For our comparison we will investigate the price of freelance developers, by having a look at the search engine upwork.com, looking at the salaries of developers for our four investigated languages. Low salaries are ranked better.

Comparison

The following table shows how many people offer their work in the respective language in a certain price range and we will calculate the average wage from that. Lower average wages are ranked higher.

Language	**$10/hr or below**	**$10/hr - $30/hr**	**$30/hr -$60/hr**	**$60/hr and above**
CoffeeScript	*231*	*1,115*	*580*	*164*
TypeScript	*31*	*150*	*60*	*14*
LiveScript	*0*	*4*	*6*	*0*
Dart	*70*	*135*	*250*	*18*

Table 4.3: Freelance salaries [81].

Language	**AVG. $/hour**
CoffeeScript	*$28.97*
TypeScript	*$26.86*
LiveScript	*$27.80*
Dart	*$33.05*

Table 4.4: Average freelance salaries in San Francisco (CA).

Our comparison tells us that the best earning freelancers are Dart developers followed by CoffeeScript freelancers. LiveScript and TypeScript freelancers are not paid as well as the other two.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	2	4	3	1

Table 4.5: Ranking for the cost of freelance developers.

4.3.1.3 Availability of Developers

For a company it is not only important that the developers are in a certain price range. The availability of developers is a crucial factor. If a developer quits the job and the company has to look for a replacement, they will want to quickly hire a replacement and not search for months.

Measuring Method

Since upwork.com already had numbers of available developers, we will take Table 4.3 as our foundation for this comparison. We will assume that languages with more developers will be the easiest to find a replacement developer for and rank those higher.

Comparison

Language	**Number of freelancers**
CoffeeScript	*2,090*
TypeScript	*255*
LiveScript	*10*
Dart	*473*

Table 4.6: Availability of freelancers.

The highest number of freelance developers are definitely found in the area of CoffeeScript, followed by Dart then TypeScript and only a few developers are offering LiveScript skills.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	4	2	1	3

Table 4.7: Ranking for the availability of freelancers.

4.3.1.4 Popularity

Popular languages have higher chances to get updated more often, and it is way easier to find developers for popular languages than for less known ones.

Measuring Method

We will have a look at github.com and see how many repositories are tagged with each of our four investigated languages. We assume that, the more repositories we will find for a certain language, the more popular it is. More popular languages are ranked higher.

Comparison

We already did the comparison in Chapter 2, Figure 2.6. Table 4.8 shows the results.

Language	**Number of Github tags**
CoffeeScript	*2,238*
TypeScript	*434*
LiveScript	*50*
Dart	*2,149*

Table 4.8: Number of repositories for each language on github.com.

The highest number of repositories are found in the area of CoffeeScript, closely followed by Dart repositories. Not as many TypeScript repositories can be found on github.com and the least number of repositories are clearly in the area of LiveScript.

This result matches the findings in the Section Availability of Developers 4.3.1.3. Here we can see that the popularity of a language goes hand in hand with the availability of the developers for it.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	4	2	1	3

Table 4.9: Ranking for the number of GitHub tags.

4.3.1.5 Trends

The trend factor will tell us if a language will be outdated in the near future, or if a company is on the safe side if they are developing a software with it. Languages that are becoming popular will be said to be safe, and languages becoming less popular will be said to be unsafe. Since no company wants to use a language for a product which might be not supported or not further developed and improved in the near future, safer languages rank better.

Measuring Method

It's difficult to forecast the future, but to get a result in this area, we will use the results of Google-Trends for this comparison and say that if the trend is growing more for a specific language than for another, then the chances that this specific language will last longer are higher.

Comparison

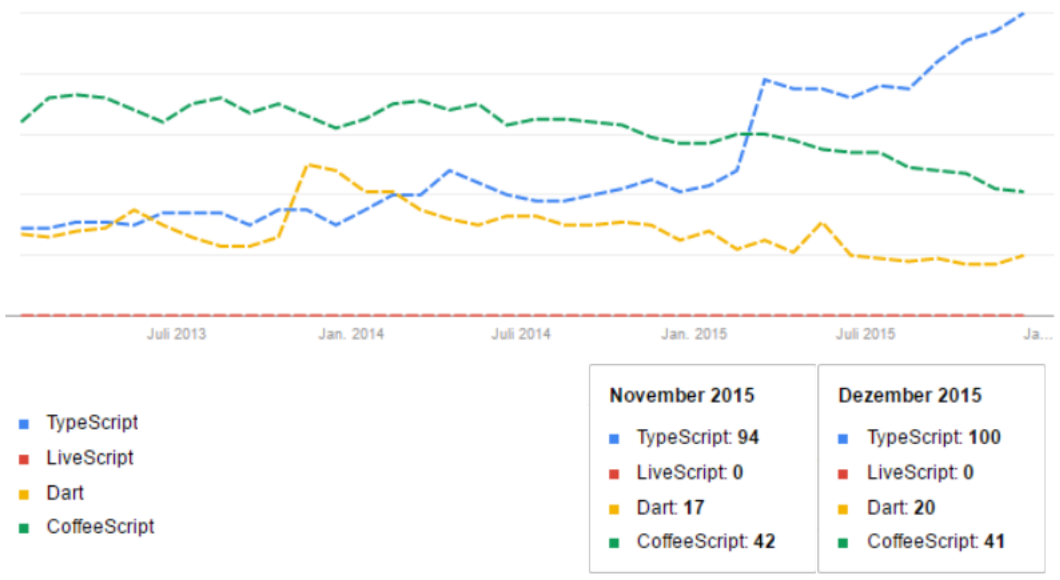


Figure 4.3: Google Trends for 01/2013 to 01/2016 [44].

Our comparison shows us that the number of TypeScript related queries is increasing the most, followed by the number of Dart related requests, while the searches for CoffeeScript are decreasing. The number of LiveScript queries is very small in comparison with the three other ones. In fact, Google Trends often shows it to be zero and we consider it worse than if the number of a language is just decreasing like it is the case with CoffeeScript. Therefore we end up with the following result.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	2	4	1	3

Table 4.10: Ranking for trends in languages.

4.3.2 Code quality

Next we examine the technical aspects and the language features of the four preprocessors. Since they were based on JavaScript, they have a lot in common, but there are also important differences that influence developer productivity.

4.3.2.1 Support for Static Typing

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." [70]

This means that every value has a type and every function only accepts values from a certain type. In statically type-checked languages, the types of expressions are known at compile time and in dynamically typed languages, they are only known at runtime. In addition, languages can be strongly or weakly typed. In a strongly typed language, the compiler reports an error in case of any type inconsistencies. In weakly typed languages, programs with certain inconsistencies may still be successfully compiled. JavaScript itself is a dynamically and weakly typed language where the compiler doesn't report any type errors [51], but some preprocessors add optional static type-checking to improve safety and reduce bugs.

Measuring Method

We will say that a language has support for static typing if any syntactically valid code can throw type errors at compile time. Otherwise we consider it dynamically typed. Static typing vs. dynamic typing is a controversial issue [61], but since static typing is an only optional feature in some of our languages and can give additional security to the programmer, we will rank the languages with support for it higher. If a language reports type errors but still compiles the code we rank it higher than a language without static typing support but lower than a language that doesn't compile code with type errors. We order the values like this:

$$\text{No support} < \text{Warnings only} < \text{Errors}$$

Comparison

CoffeeScript is completely dynamically typed. There is no static type checking [36, Language]. TypeScript is a superset of JavaScript that adds optional static typing and limited type inference [1, Section 7.2]. The basic types are numbers, booleans, strings, arrays, any, void, functions and classes [13, Basic Types]. These are not required and the programmer can write code without type annotations. LiveScript is dynamically typed by default, but the *–const* flag turns all variables into constants. This means that we can no longer cast variables because variables can't be changed at all [94, Assignment]. This is unusual for a scripting language but it does mean that LiveScript has support for static typing according to our definition.

Dart is unique because it has type checking but the type checker only produces warnings, not errors [50, Section 19]. No matter how the types are, the Dart code will still run:

> "Adding types will not prevent your program from compiling and running—even if your annotations are incomplete or plain wrong." [45]

Language	**Static typing support**
CoffeeScript	*No support*
TypeScript	*Errors*
LiveScript	*Errors*
Dart	*Warnings only*

Table 4.11: Static typing support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	0	2	2	1

Table 4.12: Ranking for static typing support in languages.

4.3.2.2 Object Support

There is no agreed-on definition on what exactly object oriented programming is. Different people include different criteria such as encapsulation, multiple implementations for the same interface, open recursion via a late-bound *this*-pointer, and more [70, Section 18.1]. Pierce gives the following list:

1. **Multiple representations.** Perhaps the most basic characteristic style is that, when an operation is invoked on an object, the object itself determined what code gets executed. Two objects responding to the same set of operations (i.e., with the same *interface*) may use entirely different operations, as long as each carries with it an implementation of the operations that works with its particular representation (...)
2. **Encapsulation.** The internal representation of an object is generally hidden from view outside of the object's definition: only the object's own methods can directly inspect or manipulate its fields (...)
3. **Subtyping.** The type of an object— its *interface* — is just the set of names and types of its operations (...) Object interfaces fit naturally into the subtype relation. If an object satisfies an interface I, then it clearly also satisfies any interface J that lists fewer operations than I (...)
4. **Inheritance.** Objects that share parts of their interface will also often share some behaviors, and we would like to implement these common behaviors just once (...)

5. **Open recursion.** Another handy feature offered by most languages with objects and classes is the ability for one method body to invoke another method of the same object via a special variable called `self` or, in some languages, `this`. The special behavior or `self` is *late-bound*, allowing a method defined in one class to invoke another method that is defined later, in some subclass of the first.

We will look at some of these features later, but we start with looking at whether a language supports basic objects, i.e. data structures that can have both fields and methods:

> "In its simplest form, an object is just a data structure encapsulating some internal state and offering access to this state to clients via a collection of methods. The internal state is typically organized as a number of mutable instance variables (or fields) that are shared among the methods and inaccessible to the rest of the program." [70, Section 18.2]

Measuring Method

We'll only rate our languages with *Yes/No*, depending on whether they support objects according to the definition above.

Comparison

Just like JavaScript, all four preprocessors support basic objects [13, 36, 41, 94].

Language	**Object support**
CoffeeScript	*Yes*
TypeScript	*Yes*
LiveScript	*Yes*
Dart	*Yes*

Table 4.13: Object support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	1	1	1	1

Table 4.14: Ranking for object support in languages.

4.3.2.3 Classes

If we look again at point 4 of the definition above, Pierce defines a class as a template from which we can instantiate new objects:

> Objects that share parts of their interfaces will also often share some behaviors, and we would like to implement these common behaviors just once. Most object-oriented languages achieve this reuse of behaviors via structures called classes —

templates from which objects can be instantiated — and a mechanism of subclassing that allows new classes to be derived from old ones (...)" [70, Section 18.1]

Classes are connected to subtyping. Subtyping occurs when we can say that a class is subtype of another class and when we can use instances of the subclass wherever we could use an instance of a parent class:

"The goal of subtyping is to refine the typing rules so that they can accept terms like the one above. We accomplish this by formalizing the intuition that some types are more informative than others: we say that S is a subtype of T, written S <: T, to mean that any term of type S can safely be used in a context where a term of type T is expected. This view of subtyping is often called the principle of safe substitution." [70, p. 182]

Note that subtyping isn't the same as inheritance: inheritance (see below) is the sharing of code. Inheritance means that a class uses the code of another class, but subclassing only means that we can use a subclass wherever a parent class is expected.

Measuring Method

If we can declare classes in a language and instantiate objects from them, we'll say that the language has classes. If the compiler also can statically enforce that certain functions only take members of a certain class, we say that the language has statically enforced classes. If we have subtyping on top of that, i.e. if we can say at compile time that some class X is a subclass of a class Y so that we can use an instance of X wherever we can use an instance of Y, then we say that a language also supports statically enforced subclassing. We order the values like this:

$$\text{No classes} < \text{Classes} < \text{Statically enforced classes} < \text{Statically enforced Subclassing}$$

The compiler doesn't have to throw an error if we use the wrong class somewhere, but it should at least throw a warning.

Comparison

CoffeeScript is dynamically typed, but we can still create classes with the *class*-keyword and instantiate objects from them with *new* [36, Classes]. We can also inherit code from other classes with *extends* but that's inheritance, not statically enforced subclassing according to our definition, since we want the compiler to make that check at compile time.

TypeScript supports classes with the *class*-keyword and subclassing with the *extends*-keyword. Subclasses inherit the members of their parent class [13, Inheritance].

LiveScript is similar to CoffeeScript as it supports classes and uses the keywords *class*, *new* and *extends* for creating classes, creating objects and inheriting from classes. It doesn't statically enforce classes, however [94, OOP].

Dart supports both classes with the *class*-keyword and subclassing via *extends* [46, Classes]. As we discussed above, the compiler only throws warnings if we use an object of the wrong class as an argument of a function.

Language	**Class support**
CoffeeScript	*Classes*
TypeScript	*Subclassing*
LiveScript	*Classes*
Dart	*Subclassing*

Table 4.15: Class support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	1	2	1	2

Table 4.16: Ranking for class support in languages.

4.3.2.4 Interfaces

Interfaces describe what an object is supposed to be able to do. An interface is a list of methods and perhaps fields. If an object implements that interface, it should provide those methods and fields. They allow developers to work together more easily, since they don't have to be interested in the implementation details of an object, just in the methods they can work with.

> "Two objects responding to the same set of operations (i.e., with the same interface) may use entirely different representations, as long as each carries with it an implementation of the operations that works with its particular representation." [70, p. 226]

Measuring Method

Languages can support interfaces in different ways. In some, the programmer has to explicitly tell the compiler that some class implements an interface and if we call a function that expects a parameter that expects some interface, it will only accept classes that implement it. Other languages use "duck typing", so any objects that has the required methods will be accepted, even if it doesn't explicitly implement the interface. It was coined by the Python engineer Alex Martelli:

> "(...) don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need (...)" [59]

We don't want to say which way is better so we will just say that any language where a function can specify that it expects some interface has interface support. Also if the programmer tries to pass an object that doesn't implement the right interface to a function, there should be an error/warning at compile-time.

Comparison

TypeScript supports interfaces via duck typing. There's no way to say that an object explicitly implements an interface [13, Interfaces]. The type checkers throws an error at compile time if an argument for a function does not have the required fields.

Dart has abstract classes (see 4.3.2.3) that function as interfaces. A class can inherit methods from an abstract class with the *implements*-keyword [50, Section 10.10]. Unlike interfaces, abstract classes can contain implemented methods. The type checker gives us a warning if we try to instantiate an abstract class or if we forget to implement an abstract method:

> "An abstract class is a class that is explicitly declared with the abstract modifier (...) We want different behavior for concrete classes and abstract classes. If A is intended to be abstract, we want the static checker to warn about any attempt to instantiate A, and we do not want the checker to complain about unimplemented methods in A. In contrast, if A is intended to be concrete, the checker should warn about all unimplemented methods, but allow clients to instantiate it freely." [50, p. 24]

Because CoffeeScript has no static type support, it also doesn't have interface support by the way we defined it above. There's no way to say that a function expects an object that implement some interface [36, Classes]. LiveScript doesn't have any support for interfaces either [94, OOP].

Language	**Interface support**
CoffeeScript	*No*
TypeScript	*Yes*
LiveScript	*No*
Dart	*Yes*

Table 4.17: Interface support across languages.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	0	1	0	1

Table 4.18: Ranking for interface support in each language.

4.3.2.5 Inheritance and Mixins

Mixins are used to give a class a combination of methods from other classes. If a class gets all methods from the other class using as a mixin, it can be seen as multiple inheritance. Some languages support both mixins and single inheritance, but since mixins can do everything that single inheritance can do, we will only concentrate on mixins.

Measuring Method

In languages like Java and C#, a subclass inherits its methods from its parent class and it is in a formal is-a relationship [12, 19].

> "Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. A derived class can have only one direct base class." [12]

So when class A inherits from class B in this way, A inherits all the methoods of B and A also becomes a sublass of B (is-a relationship). With mixins we can just inherit the methods without subclassing:

> "A mixin specifies a set of modifications (overrides and/or extensions) to be applied to a superclass parameter. A mixin differs from an ordinary subclass definition in that it abstracts over the identity of its superclass.
>
> (...)
>
> Mixins do not affect the semantics of subclass construction (...)" [4, p. 2-3]

Some languages have built-in support for mixins, while in others we can implement it ourselves. We therefore order the values like this:

$$\text{No} < \text{Can be implemented} < \text{Native}$$

Comparison

TypeScript supports mixins [13, Mixins]. We can declare a class and let another class inherit its methods with the *implements*-keyword. We also have to call a special method *applyMixins* for each class. Dart supports mixins via the keyword *with* [11]. We can mixin multiple classes, as shown in Listing 4.1.

```
class Maestro extends Person with Musical, Aggressive, Demented {
   Maestro(name):super(name);
}
```

Listing 4.1: Mixin example for Dart.

Here *Maestro* is a subclass of *Person*, but with the features of three other classes mixed in. CoffeeScript doesn't support mixins natively, but we can implement the feature ourselves [7], as shown in Listing 4.2.

```
extend = (obj, mixin) ->
obj[name] = method for name, method of mixin
obj
```

```
include = (klass, mixin) ->
extend klass.prototype, mixin

# Usage
include Guitar,
strings: true
guitar = new Guitar()
guitar.strings
```

Listing 4.2: Mixin implementation for CoffeeScript.

LiveScript is very similar to TypeScript as it supports mixins with the *implements*-keyword. One can mixin both classes and objects [94, OOP].

Language	**Mixin support**
CoffeeScript	*Can be implemented*
TypeScript	*Native*
LiveScript	*Native*
Dart	*Native*

Table 4.19: Mixin support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	1	2	2	2

Table 4.20: Ranking for mixin support in each language.

4.3.2.6 Generics

Generics are also called parametric polymorphism. They allow methods or classes to work over a range of types in a uniform way:

> “Parametric polymorphism, the topic of this chapter, allows a single piece of code to be typed "generically," using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same.” [70, p. 340]

Using generics, a class or a method can safely deal with many different types in a safe way. The same algorithm can thus be used for multiple data types, eliminating the need for type-specific implementations if we don’t care about the argument’s type.

Measuring Method

If a language supports type parameters and can statically enforce parametric polymorphism, we’ll say that it supports generics and otherwise, we’ll say that it does not.

Comparison

TypeScript, and Dart support generics, but CoffeeScript and LiveScript do not, since they're dynamically typed. While it's true that we can give any type of argument to any function in CoffeeScript and LiveScript, we can't say something like "the return type of a function has to be the same as the argument's type", which would be possible with generics (see Listing 4.3). TypeScript supports generics through type parameters that look similar to Java [14, Generics]. Dart supports generics with almost the same syntax [46, Generics].

```
function identity<T>(arg: T): T { return arg; }
```

Listing 4.3: Generics in TypeScript.

Language	**Generics support**
CoffeeScript	*No*
TypeScript	*Yes*
LiveScript	*No*
Dart	*Yes*

Table 4.21: Support for generics in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	0	1	0	1

Table 4.22: Ranking for generics support in each language.

4.3.2.7 Modularization

Modularization is more than just the ability to write code in different files. It includes selectively hiding some of code and only exposing some parts to the userss, which makes code reuse and cooperation among developers easier. It's also necessary to be able to refer to the contents of a module by their qualified names when two modules export the same name.

Measuring Method

We will split up the evaluation into four different subcriteria and count how many of them each language fulfills. They are the ability to:

1. selective exporting,
2. qualified names,
3. aliases,
4. selective importing.

Even in JavaScript we can create simple modules by putting code into different files, but our languages should support more than just that. Only exporting some parts of a module means that we can hide certain fields or functions so that they are not accessible by whoever uses the module. Referring to imported objects via their qualified names means that if we have two modules N and M that both export an object X, we can refer to X via $M.X$ or $N.X$ to resolve the ambiguity. Lastly, we should be able to import modules via a shorter alias so that, if we import a module $MasA$, we should be able to refer to its object X with $A.X$ instead of $M.X$, which is especially useful in the case of long module names. Selective importing is the ability to only import some of the names that a module exports. In many cases we only want to import one or two functions from a very large module and in these cases selective import keeps our namespace clean of clutter.

Comparison

CoffeeScript imports modules with the *require*-keyword. By default nothing is exported and we have to explicitly add members to the module's *export*-field. We can alias imported modules by assigning the result of *require* to a variable with the desired name. Selective importing is not possible [57, Chapter 6]. Listing 4.4 shows an example for module usage.

```
# module.js
module.exports.myProperty = ->
   ...

# user.js
m = require("random_module")
m.myProperty
```

Listing 4.4: Module usage in CoffeeScript.

TypeScript supports modules with selective exporting: all exported objects/functions/values have to be marked with the *export*-keyword. One can also refer to imported objects with the fully qualified named and import modules under an alias [13, Modules]. As with CoffeeScript, there is no support for selective exporting. Listing 4.5 from the TypeScript Handbook shows us an example.

```
module Shapes {
   export module Polygons {
      export class Triangle { }
      export class Square { }
   }
}

import polygons = Shapes.Polygons;
var sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Listing 4.5: Module usage in TypeScript.

We can export things with the *export*-keyword and import them with *require*. Aliasing and selective importing are also possible. Listing 4.6 shows an example from the LiveScript manual [94, Operators].

```
# module
export func = ->
export func2 = ->
hiddenFunc = -> # not exported

# user
require! module : {func} : m # only import one function, alias as m
```
Listing 4.6: Module usage in LiveScript.

Dart has a very advanced module system:

> "The import, part, and library directives can help you create a modular and shareable code base. Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (_) are visible only inside the library. Every Dart app is a library, even if it doesn't use a library directive.
>
> Libraries can be distributed using packages." [46, Libraries and visibility]

We can selectively export, import with a qualified name, import with an alias and only import parts of a library if we want. Dart also has more advanced features like lazily loading libraries, re-exporting them and splitting libraries into more than one file [46, Libraries and visibility]. We can also import all of a library and just hide some names we don't want as shown in Listing 4.7.

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2; //aliasing
import 'package:lib3/lib3.dart' show foo; //import only foo
import 'package:lib4/lib4.dart' hide foo; //import everything except foo

var element1 = new Element(); // Uses Element from lib1.
var element2 =
new lib2.Element(); // Uses Element from lib2.
```
Listing 4.7: Module usage in Dart.

Language	**Selective exporting**	**Qualified names**	**Aliases**	**Selective importing**	**Sum**
CoffeeScript	*Yes*	*Yes*	*Yes*	*No*	3
TypeScript	*Yes*	*Yes*	*Yes*	*No*	3
LiveScript	*Yes*	*Yes*	*Yes*	*Yes*	4
Dart	*Yes*	*Yes*	*Yes*	*Yes*	4

Table 4.23: Modularization support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	1	1	2	2

Table 4.24: Ranking for modularization in each language.

4.3.2.8 Operator Overloading

When the same function can have different implementations depending on the types of the arguments, we say that a function is overloaded. Operator overloading is a type of *ad-hoc polymorphism* which Pierce defined like this:

> "Ad-hoc polymorphism (...) allows a polymorphic value to exhibit different behaviors when "viewed" at different types. The most common example of ad-hoc polymorphism is overloading, which associates a single function symbol with many implementations; the compiler (or the runtime system, depending on whether overloading resolution is static or dynamic) chooses an appropriate implementation for each application of the function, based on the types of the arguments." [70, p. 340]

All of our languages support the open recursion mentioned in 4.3.2.2. We can just define a method with the name X in different classes and when we call X on some object at runtime, the actual implementation will depend on the type of the object. However, not all of our languages support the overriding of built-in operators like +, *, or &&.

Measuring Method

If we can redefine at least some built-in operators in an object or a class, we'll say that the language supports operator overloading, otherwise it doesn't.

Comparison

CoffeeScript and TypeScript have no syntax for overloading operators and don't support the feature at all [15, 36]. LiveScript does have some basic polymorphism built in [93], but we can't overload operators for our own types since there's no syntax for it [94].

In Dart we can overload operators with the *operator*-keyword [46, Overridable Operators], but each class can only override an operator once. Listing 4.8 shows an example class.

```
class MyClass {
   int content = 0;

   MyClass (content) { this.content = content; }

   //Overloading +
   operator +(MyClass other) {
      content == other.content;
   }
}

//Usage
MyClass c = new MyClass(1) + new MyClass(2);
```

Listing 4.8: Overloading in Dart.

Language	**Operator overloading**
CoffeeScript	*No*
TypeScript	*No*
LiveScript	*No*
Dart	*Yes*

Table 4.25: Overloading support across languages.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	0	0	0	1

Table 4.26: Ranking for overloading support in each language.

4.3.3 Development time

Lastly we look at the things that aren't language features, but still influence developer productivity: code reuse and tool support.

4.3.3.1 Code Reuse

When we can use the same code for multiple projects or for multiple problems in the same project, we speak of code reuse. The ability to reuse code in a language has a huge influence on developer productivity. Splitting code into reusable modules, writing generic implementations, providing interfaces which different classes can implement all enable a developer to implement projects in less time and also save time from on project to the next.

Code reuse is a very complicated field with a great deal of literate that deals with detecting and measuring code reuse across software projects [32,76]. We have to keep in mind that that our languages are relatively new and not too widespread. Because good data isn't available yet, we won't do a quantitative comparison. Instead, we will compare the four languages qualitatively, looking at features they support which enable developers to write reusable code. For other qualitative analyses about code reuse, see Cary [6], Vachharajani [82] and Harrison [33].

Measuring Method

Sadly, empirical measurements about code verbosity are hard to find for our preprocessors because all of them are in pretty small niches. For example, at the time of this writing there isn't a single problem on the algorithm comparison website Rosetta Code [64] that is solved in CoffeeScript, TypeScript, LiveScript and Dart. Instead we will take those language features from Section 4.2.3 that we can reasonably say influence the verbosity and the reuse of the code. This analysis isn't perfect, but it gives us a rough estimate of how concisely we can program in a language. The foundation for our comparison will be the following list of language features related to code quality:

- Interfaces (Section 4.3.2.4)
- Inheritance and mixins (Section 4.3.2.5)
- Generics (Section 4.3.2.6)
- Modularization (Section 4.3.2.7)
- Operator Overloading (Section 4.3.2.8)

The items of this list influence code reuse directly: interfaces let us provide multiple implementations that provide the same functionality, inheritance and mixins allow us to share code across classes, generics allow us to write implementations that work for many types, modularization allows us to split our code and reuse parts of it and operator overloading allows us to use built-in operators for our custom types.

We will take the rank for each language and each category and sum them. The higher the rank of a given language in some category, the better.

Comparison

The table below summarizes the ranking from the previous subsections.

Language	**Interfaces**	**Inheritance and mixins**	**Generics**	**Modularization**	**Operator overloading**	**Sum**
CoffeeScript	0	1	0	1	0	**2**
TypeScript	1	2	1	1	0	**5**
LiveScript	0	2	0	2	0	**4**
Dart	1	2	1	2	1	**7**

Table 4.27: Code reuse of each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	1	3	2	4

Table 4.28: Ranking for the code reuse of each language language.

4.3.3.2 IDE support

IDE (short for integrated development environment) is an application that supports the developer writing code. An IDE offers a source code editor and often tools like compilers, debuggers, type checkers, profilers and code completion that all help in the process of developing software. IDEs often support many different languages, either natively or with plugins.

Measuring Method

For this comparison we will look at feature-rich IDEs (Visual Studio [17], Eclipse [30], NetBeans [20]) as well as more streamlined popular editors (Sublime Text [75], Atom [39]). We will also note whether the language has its own native editor. In the end we will sum up how many editors support the language (a native editor will count as 1).

Various plugins offer different levels of support. For the feature-rich IDEs, we wanted at least syntax highlighting, error checking and debugging support. For the editors, syntax highlighting was enough. Each program should provide this support for a language natively or there should be some plugin that does it. We explicitly didn't count plugins for IDEs that only provided basic syntax highlighting.

Comparison

CoffeeScript is supported through plugins by Visual Studio [26], NetBeans [77], Sublime Text [35] and Atom [40]. TypeScript is supported natively by Visual Studio [18] and through plugins by Eclipse [78], Sublime Text [16] and Atom [80]. LiveScript is not supported to the degree described above by any IDE. It is supported through plugins by Sublime Text [52] and Atom e [8]. Dart is supported through plugins by Eclipse [43] and NetBeans [86], Sublime Text [56] and Atom [42]. It also has its own native IDE called Dartium [46, Tools].

Language	**Visual Studio**	**Eclipse**	**NetBeans**	**Sublime Text**	**Atom**	**Native editor**	**Sum**
CoffeeScript	*Yes*	*No*	*Yes*	*Yes*	*Yes*	*No*	4
TypeScript	*Yes*	*Yes*	*No*	*Yes*	*Yes*	*Yes*	5
LiveScript	*No*	*No*	*No*	*Yes*	*Yes*	*No*	2
Dart	*No*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	5

Table 4.29: IDE support in each language.

	CoffeeScript	TypeScript	LiveScript	Dart
Rank	2	3	1	3

Table 4.30: Ranking for tool support in each language.

4.4 Applying a Cost-Benefit Analysis

In this section we define what a cost-benefit analysis is as well as where and how to use it. Cost-benefit analysis is an established method for comparing the values of alternatives according to sets of criteria [62,72]. Layard gives this basic definition [72][p.1]

> "The basic notion is very simple. If we have to decide whether to do A or not, the rule is: Do A if the benefits exceed those of the next best alternative course of action, and not otherwise. If we apply this rule to all possible choices, we shall generate the largest possible benefits, given the constraints with which we live."

In a cost-benefit analysis, we create a list of criteria and give each of them a weighting, where higher weights mean more important criteria. The value of an alternative for a criterion multiplied by the criterion's weight is called the weighed value. Then we sum up all weighed values of an alternative to get its total benefit.
Our four preprocessors will be our four alternatives and the criteria will be the ones from Section 4.3. However, since we want to evaluate our languages from the point of view of different people, we will use different weights depending on the group of interest.
Cost-benefit analysis is sometimes done in computer science [54], but only rarely for comparing programming languages. Most papers that compare languages only list their features and give a plain English summary [71], but we wanted to compare our languages in a more systematic way.

4.4.1 Method

We apply the cost-benefit to our case like this: we create a table where the four languages are in the columns and the criteria are in the rows. We put the ranking of the given language in as the value for each criterion but we leave the weighting blank. This table will be our template for the later analyses where we will do the cost-benefit analysis from the point of view of different groups. If all languages have the same ranking for some criterion, we will just delete it from the template since it can't change the comparison.
For the actual analysis, we will take each group separately. We will only extract those rows from the template which are actually relevant to that group and we will give those criteria weights. Based on that table we made from the template, we will do the cost-benefit analysis to see how beneficial each language would be from the point of developers, management or both.
Since different organizations might value some attributes in a language more than others, we won't just take one set of weights and then say that one language is better than another. Instead, we will give two use case scenarios with different weights for the criteria and show how some real organizations might evaluate languages differently, but still systematically, based on their attributes.

4.4.2 Template

The template below was created from the data gathered in the previous section and will be used as the basis our cost-benefit analysis in the next subsection, where we will apply it on a sample use case scenario.

	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Financial Resources					
Developers					
Costs of employed developers	3	4	1	2	
Costs of freelance developers	2	4	3	1	
Availability of developers	4	2	1	3	
Risks					
Popularity	4	2	1	3	
Trends	2	4	1	3	
Code Quality					
Language Features					
Support for static typing	0	2	2	1	
~~Object support~~	~~1~~	~~1~~	~~1~~	~~1~~	
Classes	1	2	1	2	
Interfaces	0	1	0	1	
Inheritance and mixins	1	2	2	2	
Generics	0	1	0	1	
Modularization	1	1	2	2	
Operator overloading	0	0	0	1	
Language Features					
Code Reuse	1	3	2	4	
Tool Support					
IDE Support	2	3	1	3	

Final Templates

Now we will select the criteria that are relevant for developers, management, or both. We will therefore divide our template from before into three parts.

Below are given our three templates corresponding to the three groups of interest.

- Group of interest: **Management**

	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Financial Resources					
Developers					
Costs of employed developers	3	4	1	2	
Costs of freelance developers	2	4	3	1	
Availability of developers	4	2	1	3	
Risks					
Popularity	4	2	1	3	
Trends	2	4	1	3	

- Group of interest: **Development Team**

	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Code Quality					
Language Features					
Support for static typing	0	2	2	1	
Classes	1	2	1	2	
Interfaces	0	1	0	1	
Inheritance and mixins	1	2	2	2	
Generics	0	1	0	1	
Modularization	1	1	2	2	
Operator overloading	0	0	0	1	

- Group of interest: **Management and Development Team**

	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Development Time					
Language Features					
Code Reuse	1	3	2	4	
Tool Support					
IDE Support	2	3	1	3	

4.4.3 Use Case Scenario

In this subsection we will introduce a sample use case scenario where a cost-benefit analysis, using our three templates above, will be done to come to a decision. The end of this cost-benefit analysis will show which language should be used under the given circumstances to get an optimal outcome.

Initial Situation

A software development company, with a focus on web projects has existed for 25 years and currently has 11 web developers employed. Until now all of these developers were using plain old JavaScript for their projects, but because of the benefits of JavaScript preprocessing languages, the company wants to switch to either CoffeScript, TypeScript, LiveScript or Dart for their next application.

The new application, the company was contracted for, should be ready to run in four months. The company will use three developers on this project. The application is planned to be running for at least the next 15 years. The client also mentioned, that there could be some changes of the application in about 5 years due to an adaption in the clients company.

Question

The software development company now wants to know which JavaScript preprocessing language they should be using to develop the clients software in that case.

Setting the Weighting Factor

First of all the software development company has to define the weighting factor range, and the weighting factor values for each attribute, for each of the three tables.

Financial Resources

The company defines a range from 1 to 3 for the weighting factors in the area of financial resources. The company has enough developers for this project and does not have to hire new employees or even look for freelancers, so neither the availability nor the costs of the developers are playing a huge role in this case. The client wants to run the software for about 15 years, so the trend of this language plays a huge role as well as the popularity. Therefore the management comes to the following weighting factors:

Financial Resources	Weighting
Developers	
Costs of employed developers	1
Costs of freelance developers	1
Availability of developers	1
Risks	
Popularity	3
Trends	3

Code Quality

The company defines a range from 1 to 3 for the weighting factors in the area of code quality.

In this case the development team consists of three persons. All three developers prefer static typed languages and languages that offer a good modularization. Two developers like to have interface support. Everything else does not really matter to them in this case. They treat the other features as nice to have.

Therefore the development team comes to the following weighting factors:

Code Quality	Weighting
Language Features	
Support for static typing	3
Classes	1
Interfaces	2
Inheritance and mixins	1
Generics	1
Modularization	3
Operator Overloading	1

Development Time

For the last table the **management and the development team** define a range from 1 to 4 in the area of development time. They agree on the following weighting factors, because the development team wants to have a good language support by the IDE since it is their first project using it, and they do not care about code reuse in their first project that much. Therefore the management and the development team comes to the following weighting factors:

Development Time	Weighting
Language Features	
Code Reuse	1
Tool Support	
IDE Support	4

Calculating the new Numbers

In the next few paragraphs, we will first show our table with the evaluated values and the weighting factor for all three areas. (The tables will represent the evaluated data from Section 4.3 using our weighting factors from Section 4.4.3.) After that table, we will calculated the new numbers using the weighting factor and sum up the values for each language.

Financial Resources

Financial Resources	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Developers					
Costs of employed developers	3	4	1	2	1
Costs of freelance developers	2	4	3	1	1
Availability of developers	4	2	1	3	1
Risks					
Popularity	4	2	1	3	3
Trends	2	4	1	3	3

Financial Resources	CoffeeScript	TypeScript	LiveScript	Dart
Developers				
Costs of employed developers	3	4	1	2
Costs of freelance developers	2	4	3	1
Availability of developers	4	2	1	3
Risks				
Popularity	12	6	3	9
Trends	6	12	3	9
Points	**27**	**28**	**12**	**24**

Code Quality

Code Quality	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Language Features					
Support for static typing	0	2	2	1	3
Classes	1	2	1	2	1
Interfaces	0	1	0	1	2
Interfaces and mixins	1	2	2	2	1
Generics	0	1	0	1	1
Modularization	1	1	2	2	3
Operator overloading	0	0	0	1	1

Code Quality	CoffeeScript	TypeScript	LiveScript	Dart
Language Features				
Support for static typing	0	6	6	3
Classes	1	2	1	2
Interfaces	0	2	0	2
Inheritance and mixins	1	2	2	2
Generics	0	1	0	1
Modularization	3	3	6	6
Operator Overloading	0	0	0	1
Points	**5**	**16**	**15**	**17**

Development Time

Development Time	CoffeeScript	TypeScript	LiveScript	Dart	Weighting
Language Features					
Code Reuse	1	3	2	4	1
Tool Support					
IDE Support	2	3	1	4	3

Development Time	CoffeeScript	TypeScript	LiveScript	Dart
Language Features				
Code Reuse	1	3	2	4
Tool Support				
IDE Support	6	9	3	12
Points	**7**	**12**	**5**	**16**

Come to a Decision

The following table will sum up all the points per investigated language in all of the three areas, and will therefore give us the end result.

Areas	CoffeeScript	TypeScript	LiveScript	Dart
Financial Resources	27	28	12	24
Code Quality	5	16	15	17
Development Time	7	12	5	16
Points	**39**	**56**	**32**	**57**

Our analysis tells us, that the company should use **Dart** for the given application, corresponding to the application requirements and the company environment. (Very closely followed by TypeScript.)

4.4.4 Alternative Use Case Scenario

Here we will introduce an alternative use case scenario. Since we were already describing the detailed steps in the section above, here we will simply provide the summary tables for our three areas - financial resources, code quality and development time.

Initial Situation

A software company with 150 developers has to develop a new web platform for an airline to handle the internal communication. The company decided, that they do not want to use plain JavaScript, but to use CoffeeScript, TypeScript, LiveScript or Dart for this application. The timeframe for this project is very tight, therefore the management decides to dedicate at least 100 developers to this project.

Financial Resources

The company has enough developers hired and available to complete this project, without external resources. Since the company has to pay the developers anyway, the management does not care about the developers cost for this project and also does not care about the popularity or trend of the used language and set the weightning factor for each item to 1. Therefore our first result table looks like shown below.

Financial Resources	CoffeeScript	TypeScript	LiveScript	Dart
Developers				
Costs of employed developers	3	4	1	2
Costs of freelance developers	2	4	3	1
Availability of developers	4	2	1	3
Risks				
Popularity	4	2	1	3
Trends	2	4	1	3
Points	**15**	**16**	**7**	**12**

Code Quality

Since many people are involved in the development of this web platform, the developers decide that the most important features are support for static typing, classes, interfaces, inheritance and mixins and modularization. The team rates these features with 2 all others with 1. Therefore the second result table looks like shown below.

Code Quality	CoffeeScript	TypeScript	LiveScript	Dart
Language Features				
Support for static typing	0	4	4	2
Classes	2	4	2	4
Interfaces	0	2	0	2
Inheritance and mixins	2	4	4	4
Generics	0	1	0	1
Modularization	2	2	4	4
Operator Overloading	0	0	0	1
Points	**6**	**17**	**14**	**18**

Development Time

Here the **management and the development team** define a range from 1 to 3 in the area of development time. They decide that they want to have a good IDE support but the code reuse plays an more important role in that project. So they rate IDE support with 2 and code reuse with 3. Therefore the last result table looks like shown below.

Development Time	CoffeeScript	TypeScript	LiveScript	Dart
Language Features				
Code Reuse	3	9	6	12
Tool Support				
IDE Support	4	6	2	6
Points	**7**	**15**	**8**	**18**

Come to a Decision

The following table will sum up all the points per investigated language in all of the three areas, and will therefore give us the end result.

Areas	CoffeeScript	TypeScript	LiveScript	Dart
Financial Resources	15	16	7	12
Code Quality	6	17	14	18
Development Time	7	15	8	18
Points	**28**	**48**	**29**	**48**

In this case our result tells us, that the company should use **Dart** or **TypeScript** for the given application, corresponding to the application requirements and the company environment.

CHAPTER 5

Conclusion

This chapter will give a short summary of the methods that have been used in this thesis and the results that have been found. After that, we will revisit our research question, asked at the beginning of this thesis, followed by some examples for possible future work in the area of this thesis.

5.1 Summary

We started off by introducing the web in the early days and how it has changed over time to become the web as we know it today. JavaScript played an important role in that process, so we introduced Script Languages in general, described JavaScript in a more detailed way and had a brief look at the future of this language. Then we discussed the problems that go along with JavaScript, and how some of the drawbacks can be solved using preprocessors. After that we listed a couple of JavaScript pre-processing languages that exist these days, picked four of them (CoffeeScript, TypeScript, LiveScript, Dart), and described them in detail with the help of some code examples to show a few of their differences.

Then we had a look at some related work, where different kind of programming languages had been compared

Followed by this background information we were able to gather our data used for the cost-benefit analysis by finding language features related to a good project outcome. We described each feature and the way we were measuring it. Afterwards we compared the feature in all four investigated languages.

Using our cost-benefit analyis template, created from the gathered data, we were able to use it on an use case scenario. This helped us, deciding which language of the four investigated, to use on a certain sotware project.

5.2 Research Question Revisited

The research question, asked in the beginning of this thesis was:

Which specific JavaScript preprocessing language (CoffeeScript, TypeScript, LiveScript, Dart) is best suited for a new web application project, depending on some of the projects properties and its environment?

To answer this question we first had to figure out which project properties are essential for a good project outcome, because not all project properties are influenced by the used language. Since all of our four investigated languages were very similiar we had to find some features where they were differing, to get a significant result.

To give a proper answer to the research question for an arbitrary web application project, we introduced a cost-benefit analysis, which should be applied by the management as well as the development team to gain a good result. The cost-benefit analysis fits this problem statement, since the factor column of this analysis can be adapted to the project needs and therefore will give a flexible way of finding the best suited pre-processing languages.

Though the answer to our research question depends very much on the used weightning factors, the result of our two example analyses show that Dart and TypeScript might be the best option to choose, followed by CoffeeScript and LiveScript.

5.3 Future Work

Since the cost-benefit analysis template can be seen as the main contribution of this thesis, it would be kind of interesting to expand this template in some future work in several ways. Expand the number of features by finding some other important language features, that are not given yet and/or dividing them into sub features to give a more detailed way to rate them. Another possibility would be to expand the number of languages by adding new languages to the given four to have a bigger range. A further improvement would be to create standardized weighting factors for different kind of web projects to make the application easier. All given suggestions would enhance the cost-benefit analysis template and help providing better results.

APPENDIX A

Acronyms

Ajax Asynchronous JavaScript and XML

IDE Integrated Development Environment

HTTP Hypertext Transfer Protocol

URL Uniform Resource Locator

VHLL very high-level programming languages

ECMA European Computer Manufacturers Association

DOM Document Object Model

API application programming interface

VM virutal machine

GWT Google Web Toolkit

FIFO first in first out

SPEC Standard Performance Evaluation Corporation

SUT System Under Test

TCP Transfer Control Protocol

TPC Transaction Processing Performance Council

Bibliography

[1] *Evaluation and Comparison of Alternate Programming Languages to JavaScript*, volume 1. EDIS - Publishing Institution of the University of Zilina, 2013.

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 496–507, New York, NY, USA, 2014. ACM.

[3] Matti Anttonen, Arto Salminen, Tommi Mikkonen, and Antero Taivalsaari. Transforming the web into a real application platform: New technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 800–807, New York, NY, USA, 2011. ACM.

[4] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in strongtalk. In *From the ECOOP 2002 Inheritance Workshop*, 2002.

[5] Donnie Berkholz. Github language trends and the fragmenting landscape – donnie berkholz's story of data. `http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape/`. Accessed: 2014-06-10.

[6] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V. W. Reynders, and Paul J. Hinker. Comparison of c++ and fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 11 1996.

[7] Alex Chaplinsky. Mixins and extending classes. `http://hardrockcoffeescript.org/classes/extending_classes.html`. Accessed: 2016-03-04.

[8] Yuanhsiang Cheng. Livescript language support in atom. `https://github.com/yhsiang/language-livescript`. Accessed: 2016-03-10.

[9] Drifty Co. Ionic: Advanced html5 hybrid mobile app framework. `http://ionicframework.com/`. Accessed: 2014-07-01.

[10] Tcl Community. Tcl developer xchange. `http://www.tcl.tk/`. Accessed: 2014-06-23.

[11] Google Corporation. Mixins in dart. https://www.dartlang.org/articles/mixins/. Accessed: 2016-03-04.

[12] Microsoft Corporation. Inheritance (c# programming guide). https://msdn.microsoft.com/en-us/library/ms173149.aspx. Accessed: 2016-03-04.

[13] Microsoft Corporation. Typescript handbook. http://www.typescriptlang.org/Handbook. Accessed: 2014-07-15.

[14] Microsoft Corporation. Typescript handbook. http://www.typescriptlang.org/Handbook. Accessed: 2016-03-03.

[15] Microsoft Corporation. Typescript language spec. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md. Accessed: 2016-03-06.

[16] Microsoft Corporation. Typescript plugin for sublime text. https://github.com/Microsoft/TypeScript-Sublime-Plugin. Accessed: 2016-03-10.

[17] Microsoft Corporation. Visual studio. https://www.visualstudio.com/. Accessed: 2016-03-10.

[18] Microsoft Corporation. Welcome to typescript. http://www.typescriptlang.org/. Accessed: 2014-04-10.

[19] Oracle Corporation. The java tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html. Accessed: 2016-03-04.

[20] Oracle Corporation. Netbeans ide. https://netbeans.org/. Accessed: 2016-03-10.

[21] Douglas Crockford. Javascript: The world's most misunderstood programming language. http://javascript.crockford.com/javascript.html. Accessed: 2014-06-30.

[22] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[23] Kevin Dangoor and contributors. Commonjs: Javascript standard library. http://www.commonjs.org/. Accessed: 2014-07-01.

[24] Robert Diana. Web and scripting programming language job trends – february 2012. http://regulargeek.com/2012/02/17/web-and-scripting-programming-language-job-trends-february-2012/. Accessed: 2014-07-01.

[25] Dan el Khen. List of languages that compile to js. https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS. Accessed: 2014-07-07.

[26] Web Essentials. Coffeescript. http://vswebessentials.com/features/coffeescript. Accessed: 2016-03-10.

[27] Apache Software Foundation. Apache license. http://www.apache.org/licenses/LICENSE-2.0. Accessed: 2014-07-08.

[28] Node.js Foundation. Node.js. http://nodejs.org/. Accessed: 2014-07-01.

[29] Python Software Foundation. Welcome to python.org. https://www.python.org/. Accessed: 2014-06-23.

[30] The Eclipse Foundation. Eclipse. https://eclipse.org/. Accessed: 2016-03-10.

[31] Jesse James Garrett. Ajax: A new approach to web applications. https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf. Accessed: 2014-06-29.

[32] Stefan Haefinger, Georg von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Managemet Science*, 54:180–193, 11 2007.

[33] R. Harrison, L. G. Samaraweera, M. R. Dobie, and P. H. Lewis. Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal*, 11:247–254, 7 1996.

[34] Robert Henderson and B. Zorn. A comparison of object-oriented programming in four modern languages, 1994.

[35] Logan Howlett. Better coffeescript. https://github.com/aponxi/sublime-better-coffeescript. Accessed: 2016-03-10.

[36] http://coffeescript.org/. Coffeescript online. http://coffeescript.org/. Accessed: 2014-04-10.

[37] Apple Inc. Thoughts on flash. https://www.apple.com/hotnews/thoughts-on-flash/. Accessed: 2014-07-01.

[38] Apple Inc. What is ios? https://www.apple.com/ios/what-is/. Accessed: 2014-07-01.

[39] GitHub Inc. Atom. https://atom.io/. Accessed: 2016-03-10.

[40] GitHub Inc. Coffeescript language support in atom. https://github.com/atom/language-coffee-script. Accessed: 2016-03-10.

[41] Google Inc. Dart. https://www.dartlang.org/. Accessed: 2014-04-10.

[42] Google Inc. Dart plugin for atom. https://github.com/dart-atom/dartlang. Accessed: 2016-03-10.

[43] Google Inc. Dart plugin for eclipse. https://marketplace.eclipse.org/content/dart-plugin-eclipse. Accessed: 2016-03-10.

[44] Google Inc. Google trends. http://google.com/trends. Accessed: 2016-01-01.

[45] Google Inc. Optional types in dart. https://www.dartlang.org/articles/optional-types/. Accessed: 2016-03-03.

[46] Google Inc. A tour of the dart language. https://www.dartlang.org/docs/dart-up-and-running/ch02.html. Accessed: 2016-03-03.

[47] Google Inc. What is android? http://www.android.com/. Accessed: 2014-07-01.

[48] Sencha Inc. Sencha touch. http://www.sencha.com/products/touch/. Accessed: 2014-07-01.

[49] indeed.com. Salary search. http://www.indeed.com/salary/. Accessed: 2015-12-30.

[50] Ecma International. *Dart Programming Language Specification*. Ecma International, 4th edition, 2015.

[51] Ecma International. *ECMAScript 2015*. Ecma International, 6th edition, 2015.

[52] Hardy Jones. Livescript sublime text 2 package. https://github.com/joneshf/sublime-livescript. Accessed: 2016-03-10.

[53] The jQuery Foundation. jquery mobile. http://jquerymobile.com/. Accessed: 2014-07-01.

[54] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Capello, and Davig P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *Parallel & Distributed Processing*, pages 1–12, 5 2009.

[55] Mitchell Lazear. Project triangle. http://mitchelllazear.deviantart.com/art/Project-Triangle-134005334. Accessed: 2014-11-04.

[56] Guillermo López-Anglada. Dart plugin for sublime text 3. https://github.com/guillermooo/dart-sublime-bundle. Accessed: 2016-03-10.

[57] Alex MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, Inc., 2012.

[58] Joe Marini. *Document Object Model*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2002.

[59] Alex Martelli. Re: Type checking in python? https://groups.google.com/forum/?hl=en#!msg/comp.lang.python/CCs2oJdyuzc/NYjla5HKMOIJ. Accessed: 2016-03-03.

[60] Laurie McLeod and Stephen G. MacDonell. Factors that affect software systems development project outcomes: A survey of research. *ACM Comput. Surv.*, 43(4):24:1–24:56, October 2011.

[61] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[62] Edward J. Mishan and Euston Quah. *Cost-benefit analysis*. Routledge, 2007.

[63] MIT. The mit license. `TheMITLicense(MIT)`. Accessed: 2014-07-08.

[64] Mike Mol. Rosetta code. `http://rosettacode.org`. Accessed: 2016-03-08.

[65] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. *CoRR*, abs/1409.0252, 2014.

[66] Regents of the University of California. The bsd 3-clause license. `https://opensource.org/licenses/BSD-3-Clause`. Accessed: 2014-07-08.

[67] O. Oluwagbemi, A. Adewumi, S. Misra, F. Majekodunmi, and L. Fernandez. An analysis of scripting languages for research in applied computing. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 1174–1180, Dec 2013.

[68] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, Mar 1998.

[69] Perl.org. The perl programming language. `http://www.perl.org/`. Accessed: 2014-06-23.

[70] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[71] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct 2000.

[72] Layard Richard and Stephen Glaister. *Cost-benefit analysis*. Cambridge University Press, 1994.

[73] Chanchal K. Roy and James R. Cordy. Are scripting languages really different? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 17–24, New York, NY, USA, 2010. ACM.

[74] C. Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, Feb 2012.

[75] Jon Skinner. Sublime text. `https://www.sublimetext.com/`. Accessed: 2016-03-10.

[76] Manuel Sojer and Joachim Henkel. Code reusein open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, 11:868–901, 3 201.

[77] Denis Stepanov. Coffeescript support for netbeans ide. `https://github.com/dstepanov/coffeescript-netbeans`. Accessed: 2016-03-10.

[78] Palantir Technologies. Eclipse typescript plug-in. `https://github.com/palantir/eclipse-typescript`. Accessed: 2016-03-10.

[79] CodingUnit Programming Tutorials. C++ preprocessor directives. `http://www.codingunit.com/cplusplus-tutorial-preprocessor-directives`. Accessed: 2014-07-07.

[80] TypeStrong. Atom typescript. `https://github.com/TypeStrong/atom-typescript`. Accessed: 2016-03-10.

[81] upwork.com. Salary search. `https://www.upwork.com/o/profiles/browse/`. Accessed: 2015-12-30.

[82] Manish Vachharajani, Neil Vachharajani, and David I. August. A comparison of reuse in object-oriented programming and structural modeling systems. Technical report, Liberty Research Group, 10 2003.

[83] W3Schools. Html5 introduction. `http://www.w3schools.com/html/html5_intro.asp`. Accessed: 2014-07-01.

[84] Zach Walton. Javascript leads the pack as most popular programming language. `http://www.webpronews.com/javascript-leads-the-pack-as-most-popular-programming-language-2012-09`
Accessed: 2014-07-01.

[85] Ye Diana Wang and Nima Zahadat. Teaching web development in the web 2.0 era. In *Proceedings of the 10th ACM Conference on SIG-information Technology Education*, SIGITE '09, pages 80–86, New York, NY, USA, 2009. ACM.

[86] Geertjan Wielenga. Dart and netbeans ide 7.4. `https://blogs.oracle.com/geertjan/entry/dart_and_netbeans_ide_7`. Accessed: 2016-03-10.

[87] Inc Wikimedia Foundation. Coffeescript. `http://en.wikipedia.org/wiki/CoffeeScript`. Accessed: 2014-07-08.

[88] Inc Wikimedia Foundation. Dart (programming language). `http://en.wikipedia.org/wiki/Dart_(programming_language)`. Accessed: 2014-07-08.

[89] Inc Wikimedia Foundation. Javascript. `http://en.wikipedia.org/wiki/JavaScript`. Accessed: 2014-06-23.

[90] Inc Wikimedia Foundation. Jeremy ashkenas. `http://en.wikipedia.org/wiki/Jeremy_Ashkenas`. Accessed: 2014-07-08.

[91] Inc. Wikimedia Foundation. Livescript. `http://en.wikipedia.org/wiki/LiveScript`. Accessed: 2014-07-08.

[92] Inc Wikimedia Foundation. Typescript. `http://en.wikipedia.org/wiki/TypeScript`. Accessed: 2014-07-08.

[93] George Zahariev. 10 things you didn't know livescript can do. `http://livescript.net/blog/10-things-you-didnt-know-livescript-can-do.html`. Accessed: 2016-03-06.

[94] George Zahariev. Livescript - a language which compiles to javascript. `http://livescript.net/`. Accessed: 2014-04-10.