0.1 Paradigms of programming language

In computer science, four main paradigms of programming languages exists [?]. In this section these paradigms will be shortly described followed by a subsection, explaining the choice of programming paradigm of the language in this project.

0.1.1 Imperative Programming

Imperative programming is a very sequential or procedural way to program, in the sense that this step is performed, then that step and so on. These steps are controlled by control-structures for example the if-statement. An example of a imperative programming language is C. Imperative programming language describes programs in terms of statements which alter the program state. This makes imperative languages very simple, and are also a good starting point for new programmers.

0.1.2 Functional Programming

Functional programming originates from the theory of functions in mathematics. In functional programming all computations are done by calling functions. In functional programming languages calls to a function will always yield the same result, if the function are called with the same parameters as input. This is in contrast to imperative programming where function calls can result in different values depending on the state of the program at that given time. Some examples of functional programming languages are Haskell and OCaml.

0.1.3 Logic Programming

Logic programming is fundamentally different from the imperative-, functional-, and object-oriented programming languages. In logic programming, one cannot state how a result should be computed, but rather the form and characteristics of the result. An example of a logic programming language is Prolog.

0.1.4 Object-Oriented Programming

Object-Oriented programming is based on the idea of data encapsulation, and grouping of logical program aspects. The concept of parsing messages between objects are also a very desirable feature when programs become of certain size. In object-oriented programming, each class of object can be given methods, which is a kind of functions which can be called on that object. For example the expression "foo.Equals(bar)", would call the Equals-method in the class of 'foo', and evaluate if 'bar' equals 'foo'. It is also easy in object-oriented languages to specify access-levels of classes, and thereby protect certain classes from external exposure. Classes can inherit form other classes. For example one could have a 'Car'-class, which inherits all properties and methods of a 'Vehicle'-class. This allows for a high degree of code-reuse.

0.1.5 Choice of Paradigm in This Project

For this project, an imperative approach has been chosen. The reason for this is that the programming language of this project should be very easy to understand for newcomers to programming. Also the programs in this programming language will likely remain of a relatively small length, which does not make object-orienting desired.

Readability	How easy it is to understand and comprehend a computation
Write-ability	How easy it is for the programmer to write a computation clearly, correctly, concisely and quickly
Reliability	Assures a program behaves the way it is suppose to
Orthogonality	A relatively small set of primitive constructs can be combined legally in a relatively small number of ways
Uniformity	If some features are similar they should look and behave similar
Maintainability	Errors can be found and corrected and new features can be added easily
Generality	Avoid special cases in the availability or use of constructs and by combining closely related constructs into a single more general one
Extensibility	Provide some general mechanism for the programmer to add new constructs to a language
Standardability	Allow programs to be transported from one computer to another without significant change in language structure
Implementability	Ensure a translator or interpreter can be written

Table 0.1: Brief explanation of language characteristics [?]

Characteristic	Readability	Writability	Reliability
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Table 0.2: Overview of trade-offs [?]

0.1.6 Design Criteria in this Project

To determine how a programming language should be syntactically described, one must have a look at the trade-offs of designing a programming language. The different characteristics of a programming language, which will be used to evaluate trade-offs can be seen on table **??**.

These characteristics are used to evaluate the the trade-offs of programming language. An overview of these can be seen on table **??**.

Based on these trade-offs, it is clear that having a simple programming language affects both readability, writability and reliability. This is because having a very simple-to-understand language, might not make it very writable. On the other hand, having a simple-to-write programming language, might not make it very readable. An example of this is the if-statement in C, which can be written both with the 'if'-keyword, or more compact. This can be seen by comparing listing **??** with listing **??**, which both yield the same result. It is then clear, that the compact if-statement might be faster to write, but slower to read and understand, and opposite with the if-statement witht he 'if'-keyword.

minipage‼

Listing 2: [