CAB402 Assignment 2

Programming Language Design Principles

By John Santias n9983244

2nd June 2019

Executive Summary

Design plays a major role in programming languages. The aim of a creator is to make programming simple and efficient for programmers, easing the use of different features. In this report, we discuss the design principles of programming languages, along with programming paradigms, and design elements. In addition, we discuss the creation of a programming language. Examples are shown throughout this report to support each argument.

Introduction

The first high-level programming languages were designed in the 1950s. Programming languages have come a long way since then, becoming a fascinating and productive area to study, and creating a world of programmers to have endless debates on the distinctions of their favourite programming language. Sometimes with almost religious zeal.

The term ***programming linguistics*** [1] is sometimes used to mean the study of programming languages. Using an analogy on the study of natural languages (natural linguistics). Both natural and programming languages have syntax (form) and semantics (meaning). English is an example of a natural language that most humans can write and speak. Other natural languages can include, Portuguese, Spanish, Chinese and so forth. They are more expressive and subtler than programming languages. However, natural linguistics is only restricted to analysing human natural languages. Programming linguists, on the other hand, design and specify programming languages, and implement them on computers.

In today's modern world, technology continues to accelerate and transform how we interact, live and work. Almost everything in technology is run by a computer program formed by a programming language. For example, robots are run by a program and listen to requests. When a user gives it a command, it will try to match the request to a method in its program and execute it. Regardless of how smart or dumb this robot is, these type of events run similarly in other technologies like applications, TVs, smart devices upon user interaction.

There may be a whole range of programming languages to choose from but may not always be the best solution to a problem. Creating a new language is a fun project to take on for a computer scientist, but when doing so, one must need to take the design principles in to account.

Design is important in a programming language aimed to make programming simple and efficient for programmers. In this report, we will explore further into the linguistics of programming through structures and discussing paradigms, design elements, and design

principles. By exploring these principles, we could create another language for the programming world to debate on.

Programming Languages

Like many existing human languages, computers have a range of programming languages that programmers can use to communicate with it. These are used to develop software, scripts, or other sets of instructions, then compiled for a computer to understand and execute.

Human-computer communicators have a range of programming languages to choose from. Some common ones you may hear often are Haskell, Python, C, and Java. New programmers will always start somewhere and continue to build skills through projects and exploring other languages. As you move from language to language, you may notice some similarities in features and concepts.

```
(C# - Object-Oriented Program)
using System; //Standard library

namespace HelloWorld
{
      class Program
      {
           static void Main(string[] args)
           { Console.WriteLine("Hello World!"); } //print line
      }
}
```

```
(F# - Functional program)
open System //Standard library

[<EntryPoint>]
let main argv =
      printfn "Hello World from F#!" //print string
      0 // return an integer exit code
```

(See Appendix A for more *'Hello world'* examples)

Each language will structure code differently because each have a different approach to solving problems [2]. We can classify these programming languages based on their features. This is called ***programming paradigms***. Some common paradigms [3] are:

- **Imperative**: Instruct the program on how to change states through assignment.
    - **Procedural**: Ability to reuse code. (C, Basic, Fortran, Pascal)
    - **Object-Oriented**: A collection of classes and objects. (e.g. Python, C#, Java, C++)
- **Declarative**: Declare properties for a desired result but not how to compute it.
    - **Functional**: The desired result declared as the value of a series of function applications. (e.g. JavaScript, Haskell, Scala, F#)
    - **Logic**: The desired result is the answer to a question about facts and rules.
    - Database processing (e.g. Prolog)

(See Appendix B for paradigm examples)

In designing a new language, one should consider a paradigm that is suitable for their end-goal. The next section will explore the elements of language designs.

Design Elements

The design elements of a programming language have a large impact on how much a human can understand and remember [4]. Computer scientists have develop these languages designed to help complex programs fit into tiny human brains. These design elements include:

- **Syntax**: The glyphs for expressing concepts. They include symbols and alphabet keywords (`%&#@!` and `if`, `continue`, `break`, etc). Some languages tend to exaggerate symbols whilst others eliminate them entirely. Either way, it's all syntax.
- **Vocabulary**: Naming common functions, methods, classes and so forth. Including conventional naming schemes and libraries/packages that are a crucial part of the vocabulary of a language.

- **Conventions**: This may not technically be part of a language but it evolves a set of strong principles ("`==` shouldn't be used to compare strings!") to help programmers work around booby traps.

Let's have a look at C# naming conventions as an example, variables are created by camelCase and classes are created in the form of TitleCase, by convention. This helps identification and allows IDEs to colourise their labels.

```
float rectangleArea = 12.f * 15.f;
private int[] numPlayers;
private int numRounds, currentPlayer, currentRound, opponent, Wind;
class Program { ... }
public int PlayerCount() { ... }
```

In helping with recognition ("What is that?"), these standards help programmers with naming convention ("What should I name this?") [5]. They help you think about patterns underlying what you're making, and potentially helping with further implementation.

There are more conventions across most modern languages, e.g.:

- Variables are changed by assignment when they are on the left side of the equals operation.
- Lines are executed top to bottom. In order or in an understandable order [6].

...and so forth.

Design Principles

Paradigms have different approaches to solving problems and don't necessarily follow all the design principles. Bruce J. Maclennan [7] defines sixteen language design principles whilst others such as Alex Feinman [8] and Mira Balaban [9] defines eight principles. To boil down this list, here are five principles I strongly believe is important in designing a language:

1. **Abstraction:** "*Avoid requiring something to be stated more than once; factor out the recurring pattern.*"

Developers should be able to define a program as a set of relatively small subprograms. By doing so, the design process difficulty will be decreased since each subprogram can be written, debugged and read as an independent unit.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

The minimax example shown above is an example of a subprogram. This is executed when it is called with values passed to its parameters *'node', 'depth', 'maximizingPlayer'*. This idea is called modularising because the program is divided up into multiple manageable modules. Abstraction is a common method of modularisation where the programs abstract out common parts of a system.

2. **Structure:** "*Should be as simple as possible and short. The structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.*"

   It should be possible to visualise the program's behaviour from its written form. For example, programmers should easily identify the statements that are executed by a loop.

```
x = []
for i = 0 to 10:
    x.append(i)
```

Here a new empty array is initialised and values from one to nine are appended. The final array is `[1, 2, 3, 4, 5, 6, 7, 8, 9]`. When one segment of code is

executed, the statements of the first segment should precede those of the second in the program.

3. **Information hiding:** "*Users can access all the modules' information so they can implement them correctly.*"

   Construct programming languages that support this principle and controls access to declarations is the package. This information inside must be known to the user, and the information about the way it will be used must be known to the implementor. A package can be in this form:

```
package TYPE is
   ...definitions...
   ...variables...
   ...functions...
end TYPE;
```

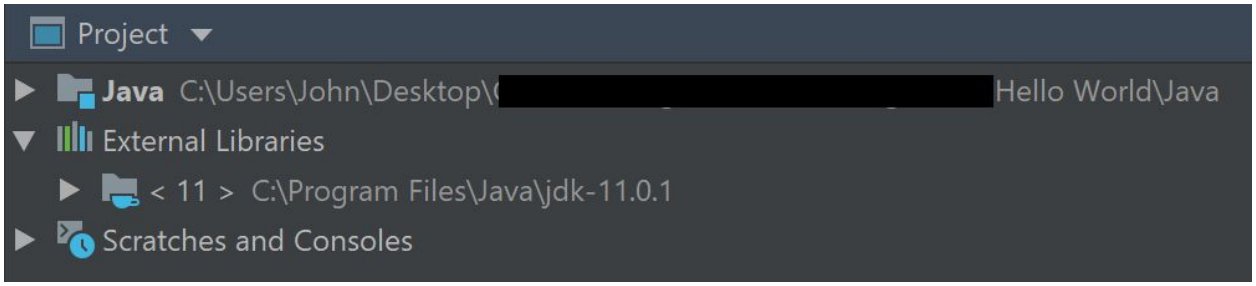
   Packages can be stored inside a directory where users wouldn't dare to edit or touch the files. For example, Java will install its libraries in the 'Program Files' directory in Windows. This will happen in a similar fashion on other operating systems. This contains all the files needed to compile and run programs as such:

```
System.out.println("Hello World")

//or

for (int i = 0; i < 10; i++){ ... }
```

   Packages can be seen in IDEs to show users how the modules are used.



Languages also add libraries by appending them to the top of the program. Users also have the ability to add their own libraries.

C# .NET	Python

`#include <stdlib.h> //standard utility functions`
`#include <stdint.h> //standard library`
`#include <stdio.h> //standard I/O functions`
`#include "game.h" //user created file`
	`import sys`
	`import itertools`
	`import math #add module`
	`import search #all in user created file`
	`from sokoban import find_2D_iterator #user defined function`

Adding libraries to the programming language (or own program) can help users see what modules are being used and how they function. It also allows users to move their methods into separate files, preventing confusion in calling methods, variables, etc.

4. **Labeling:** *"Program does not require the user to know the absolute position of an item in a list. Labels with any position must be referenced elsewhere. "*

   Suppose we are implementing a package to draw graphs:

```
(Ada)
procedure DRAW_GRAPH(X_AXIS, Y_AXIS: COORD; X_SCALE, Y_SCALE: FLOAT;
X_SPACING, Y SPACING: NATURAL; X_LOG, Y_LOG: BOOLEAN; X_LABEL, Y_LABEL:
BOOLEAN; FULL_GRID: BOOLEAN);
```

   Functions (or procedures) that have a large number of parameters, like the example above, aren't uncommon in subprogram libraries that try to achieve flexibility and generality. This can be called in the form like this:

```
DRAW_GRAPH(400, 400, 2.0, 0.5, 15, 15, FALSE, TRUE, TRUE, TRUE, FALSE);
```

   By calling this method, it can be difficult to remember the meanings of the parameters so programmers will have to find the function to find the meaning of the parameters.

5. **Regularity:** Regular rules, without exceptions, makes it easier for users to learn, use, describe and implement.

   Parts of the language are regular and roughly recur in the same way. This is a benefit because it allows the mind to separate what's important from the frame around it.

   There are two important parts to achieving this: *Things that look similar should be similar,* and inverse*, Things that look different should be different.* [7]

   Javascript violates the important maxim in user experience because of the equality operators:

```
0 == ""      (True, Empty string is zero and false)
"0" == 0     (True, zero is upconverted to a string)
"" != "0"    (True, somehow equal!?)
```

   `==` will look like a regular operand that programmers have experienced in other programming languages. However, the outcome is different. Designers of this language have implemented an alternative that uses `===` to work the way people expect equality to work.

   Strongly typed languages such as C# or Python would be a counter-example for this issue:

```
(Python)
if (x = 100){
   print("Comparison worked!")
}
```

   The if statement expects a boolean value and the comparison shown `x = 1` is an assignment. This will get flagged immediately when you try to compile it. Operators in these languages give programmers confidence on the outcome.

Other principles can be added to the list [8] [7] such as:

- **Error Defence:** A series of safety nets are in place to catch the error. If one error isn't caught, it might get caught by another.

- **Portability:** The language should be able to run on any machine. Prevent a language's features and facilities to be dependant on a particular machine.
- **Expressibility**: Can easily figure out how to write what they have in their mind.
- **Predictability**: From looking at examples, a programmer can use it to generalise new code.

Creating a programming language

In forming a new language, Alex Feinman [4] states "*The art of the designer is in balancing these principles and coming up with something that forms a cohesive whole.*" The Python programming language is an example that follows all of the principles stated in the previous section.

```
import math #module to access math functions

#Function to calculate the volume of the cylinder
def volume_cylinder(r, h):
   volume = math.pi * (r ** 2) * h #Uses volume formula
   return volume

#Listens for user input
radius = input("What is the radius of the cylinder?")
height = input("What is the height of the cylinder?")

#Calls the function to calculate volume
volume = volume_cylinder(float(radius), float(height))

#prints result
print("The volume is {0}".format(volume))
```

Python is a widely used language in the modern world [10]. It's considered to be a universal language that meets various development needs. This language is used in multiple areas such as web development, machine learning, artificial intelligence (overtaking R) and much more. Its a language that is easy to learn and best for beginners due to its simplicity and efficiency. It uses fewer lines of code to solve a problem, simple syntax, easy to read and English-like commands [11].

By creating a language with similar structure and following the design principles, you may come up with the next best solution to various problems. However, in doing so you must also consider the purpose of your language as to what it's aimed to solve and which paradigms you are targeting.

Conclusion

In the modern world, almost everything technology uses a programming language to function. Technology continues to accelerate how we interact, live and work. The programming languages available may not be always suitable for such problems but there is room for improving them or creating a much powerful tool. Whether it be building on top of a language, combining or creating another paradigm (if you're that crazy).

As we have seen some problems in programming, design plays a major role in the development of programming languages. The aim is to make programming simple and efficient for programmers, easing the use of operators, repeated programs, etc. By following the principles in this report, and learning from the designs of past & present programming languages, we can hope to have another useful tool for the programming world to debate on.

Reflection

I realised that the design principles of programming languages is a broad topic. There is so much more I could discuss in terms of low vs high-level languages, advanced concepts, and even compilers that is a big part of computing languages. These subtopics could have been a great addition to this report that could further validate my arguments.

After conducting research on this topic, it has made me appreciate programming language more because of the different features, simplicity, and efficiency. I can see how each language is structured with a few similarities that allow me to think and solve in a similar fashion.

Designing and creating my own programming language is definitely a project that I'd like to work on. It may be a big learning curve to overcome but it's a great opportunity to discover

new things. This could help me further understand how programs are designed, compiled, the creation of libraries, and much more.

References

---

[1] A., D., 2004. Programming Language Design Concepts. Wiley.

[2] Paradigms. 2019. paradigms. [ONLINE] Available at: https://cs.lmu.edu/~ray/notes/paradigms/.

[3] GeeksforGeeks. 2019. Introduction of Programming Paradigms - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/introduction-of-programming-paradigms/.

[4] Alex Feinman. 2019. Design principles for programming languages – Alex Feinman – Medium. [ONLINE] Available at: https://medium.com/@afeinman/design-principles-for-programming-languages-bb152538a85c.

[5] martinfowler.com. 2019. TwoHardThings. [ONLINE] Available at: https://martinfowler.com/bliki/TwoHardThings.html.

[6] Wikipedia. 2019. Befunge - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Befunge.

[7] Maclennan, B., 1983. Principles of Programming Languages: Design, Evaluation, and Implementation. 2nd ed. Oxford University Press: Oxford University.

[8] Alex Feinman. 2019. Design principles for programming languages, Part 2a: Readability, Expressability, Concision, and…. [ONLINE] Available at: https://medium.com/@afeinman/design-principles-2a-1874c14975ab.

[9] Balaban, M., 2017. Principles of Programming Languages. *Principles of Programming Languages*, 1, 423.

[10] Bobby. 2019. Why Python is Popular Despite Being (Super) Slow – Bobby – Medium. [ONLINE] Available at: https://medium.com/@trungluongquang/why-python-is-popular-despite-being-super-slow-83a8320412a9.

[11] 6 Reasons Why Python Is Suddenly Super Popular. 2019. 6 Reasons Why Python Is Suddenly Super Popular. [ONLINE] Available at: https://www.kdnuggets.com/2017/07/6-reasons-python-suddenly-super-popular.html.

Appendix A Hello World in Different languages

Appendix A-1: C#

```
(C# - Object-Oriented)
using System; //Standard library

namespace HelloWorld
{
      class Program
      {
            static void Main(string[] args)
            { Console.WriteLine("Hello World!"); } //print line
      }
}
```

Appendix A-2: F#

```
(F# - Functional)
open System //Standard library

[<EntryPoint>]
let main argv =
      printfn "Hello World from F#!" //print string
      0 // return an integer exit code
```

Appendix A-3: Bash

```
(BASH - Procedural)
#!/bin/sh //defines is a bash script
echo "Hello, World!" //print string
```

Appendix A-4: C

```
(C - Imperative, Procedural)
#include <stdio.h>
int main()
{
  // printf() displays the string inside quotation
  printf("Hello, World!");
  return 0;
}
```

Appendix A-5: C++

```
(C++ - Object-Oriented Program)
#include <iostream>
int main()
{
   std::cout << "Hello, world!\n";
}
```

Appendix A-6: Golang

```
(GOLANG - Multiparadigm, functional/object-oriented/imperative)
package main
import fmt "fmt"
func main()
{
   fmt.Printf("Hello, World!\n");
}
```

Appendix A-7: Pascal

```
(Pascal - Procedural)
program hello;
begin
writeln('Hello, World!');
end.
```

Appendix A-8: PHP

```
(PHP - Functional/object-oriented/procedural)
<?php
   echo 'Hello, World!';
?>
```

Appendix B Different Paradigm examples

Appendix B-1: Procedural Programming

```
(Fortran)
program name
!This program writes my name five times
integer::i
do i=1,5
print*,'Hello world, my name is John!'
end do
end program namee
```

Output:

```
$gfortran -std=f95 *.f95 -o main
$main
Hello World! My name is John!
 Hello World! My name is John!
 Hello World! My name is John!
 Hello World! My name is John!
 Hello World! My name is John!
```

Appendix B-2: Object-Oriented Programming

```
public class Car {

	private int doors;
	private int wheels;
	private String model;
	private String engine;
	private String colour;

	public void setModel(String model){
	   String validModel = model.toLowerCase();
	   if (validModel.equals("roadster") || validModel.equals("model
s")){
	      this.model = model;
	   } else {
	      this.model = "Unknown";
	   }
	}

	public String getModel(){
	   return this.model;
	}
}

public class Main {

	public static void main(String[] args) {
	   Car tesla = new Car(); //creates a new class object
	   System.out.println("Model is " + tesla.getModel());
	   tesla.setModel("Roadster");
	   System.out.println("Model is " + tesla.getModel());
	   tesla.setModel("Model S");
	   System.out.println("Model is " + tesla.getModel());
	}
}
```

Appendix B-3: Functional Programming

Appendix B-3.1: Impure

```
var tax = 0;
var rate = 0.19;

function calculateTax(income){
   tax = rate * income; //mutates data outside
}

calculateTax(1200)
console.log(tax) //228
```

Appendix B-3.2: Pure

```
function calculatePure(x, y){
   return (x * x) + y;
}

console.log(calculatePure(10, 20)) //120
//Pure returns the exact result everytime
```

Appendix B-4: Logic Programming

```
(Prolog file - sunny.pl)
john_is_outside.              /* John is cold */
sunny.                        /* it is raining */
john_forgot_his_hat.          /* john forgot his hat */
john_is_wearing_sunnies.      /* john is wearing his sunnies */
john_is_walking.              /* john is walking */
```

```
(Prolog Interpreter)
?- john_forgot_his_hat
|     .
true.

?- sunny.
true.

?- john_is_outside.
true.

?- john_is_inside.
ERROR: Undefined procedure: john_is_inside/0 (DWIM could not correct goal)
?- raining.
ERROR: Undefined procedure: raining/0 (DWIM could not correct goal)
?- john_forgot_his_hat.
true.
```