Unit 3

Reflection

This unit focused on the understanding the evolution of programming languages from their humble beginnings to their latest incarnations. The idea of a programming paradigm was one I previously encountered, but did not give much consideration to it. However, researching this topic in this unit was fascinating because it highlighted that programming languages are not merely rules for syntax but have deeper meanings, namely:

- Support programming paradigms such as *recursion, imperative (or procedural), object-oriented, declarative, logic, functional.*
- Enable programming concepts such as *abstraction*, *polymorphism, data type conversions, data encapsulation, memory management, type hierarchy, pointer references,*
- Provide a means of data abstraction.

Paradigms hold true irrespective of programming language and, as Van Roy (2009) mentions, “there are many fewer programming paradigms than programming languages.”. Such a statement highlights to me why multitude of programming languages exist: creativity. The creative manner in which paradigms and concepts are realised to compute the same outcome.

Reading through the module’s content, I am fascinated by the observation of *technological evolution* as it pertains to programming languages: today’s languages are the result of what came before. As Wegner (1976) notes—the following is summarised form his work—, the history of programming languages can be described as follows:

1. The EDVAC report of 1944. The introduction of **Von Neumann machines** proposed by Von Neumann and describes a one-address machine language.
2. Introduction of **subroutines** and “boot loaders” and conversion of decimal to binary conversions for operation codes and addreses.
3. Introduction of **assemblers** (Wilkes et al., 1951) that translate from symbolic machine language into an internal machine representation and assembles multiple subroutines into a single executable.
4. **Fortran** proved to be an important practical milestone in demonstrating the feasibility and viability of higher level languages. It introduced concepts such as *variables, expressions, statements, array* and *iterative and conditional branching.*

5. Algol was an important conceptual milestone and introduced **Backus-Naur Form** (BNF) to define language syntaxes. It introduced block structure, explicity type declarations, scopre rule for variables, nested if-then-else blocks, call by-value, recursive subroutines, arrays with dynamic bounds
6. COBOL introduced the notion of records, file manipulation and a natural style of programming. It is best seen as contributing toward the idea of **data descriptions**.
7. Simula 67 (Dahl & Hoare, 1972) generalised the idea of a **class** that consists of procedures and data declarations.
8. Lisp was developed to tackle problems in artificial intelligence and is a higher level programming language.
9. Development of **grammars** by Chomsky and Miller (1968) in the study of language theory.
10. Development of **compiler theory**.
11. Pascal (developed by Wirth, 1971), provided richer data structures and was concerned about simplicity at the conceptual, user and implementation levels.
12. The rise of **structured programming** which influences if-then-else and while-do constructs.

At their core, high level programming languages translate human-readable instructions in operation codes understandable by machines. The human-readable instructions follow a fixed structure (or “syntax”). Any errors in the syntax will be shown to the program developer that they can correct all errors before the instructions are converted to machine code. Programming languages allow developers to segregate their code into functional blocks which often follow the object-oriented programming paradigm, where a collection of closely-related blocks are contained by a single “class”.

Programming languages define the “rules” developers can use to build software—loops, conditional checks, variable declarations—, however software patterns are considered tried and tested, reusable solutions to standard software design problems generally not confined specific domains; they are generally considered best practice. Software patterns help deliver better quality code, quicker when used correctly for an existing system design problem; they provide a common vocabulary to express insights and experience about such problems and solutions; patterns help to codify what good software architecture looks like and they aid in fomenting sound reason about them.

Some of the key programming challenges are seen as:

- *clarity* and *readability* of code. Using software metrics such as Source Lines of Code (SLOC) or Cyclomatic complexity can help others assess the code quality. This is a key challenge since lesser experienced developers will not write code at the same level of more experienced developers, due to their lack of understanding of the programming language concepts, patterns of paradigms available to them.
- Inherent introduction of bugs into software development. For this, a best practice is to utilise test procedures and attempt to reproduce the buggy data inputs.
- Plethora of tools and frameworks. Best practice can be for an organisation to standardise on acceptable tools and frameworks, those that have fewer security vulnerabilities and get the job done.
- Continually changing requirements. Best practice is to utilise a software development lifecycle that allows changes to freeze for the duration of a development cycle, for example, the agile process.
- Complexity. Developers tend to over complicate systems development. My personal recommendation is to follow the SOLID principles focusing particularly on the Single responsibility aspect.

References

Chomsky, N. & Miller, G.A. (1968). Introduction to the formal analysis of natural languages. *Journal of Symbolic Logic*, *33*(2).

Dahl, O.J. & Hoare, C.A.R. (1972) Chapter III: Hierarchical program structures. In *Structured programming:* 175-220).

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, *104*: 616-621.

Wegner, P. (1976) Programming Languages? The First 25 Years. *IEEE Transactions on computers*, *25*(12): 1207-1225.

Wilkes, M.V., Wheeler, D.J. & Gill, S., 1951. *The Preparation of Programs for an Electronic Digital Computer: With special reference to the EDSAC and the Use of a Library of Subroutines*. Addison-Wesley Press.

Wirth, N. (1971) The programming language Pascal. *Acta informatica*, *1*(1):35-63.