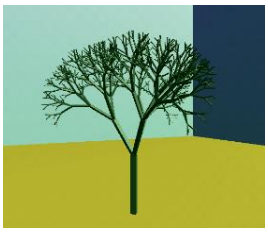

GATS Quick Guide
Programming Paradigms

Author: Garth Santor
Editors: Trinh Hān
Version/Copyright: 0.2.0 (2023-12-29)

Definition

A ***programming paradigm*** is a framework or model of a programming language that dictates how you think about programming and how you express a solution in that language.

Paradigm: a framework containing the basic assumptions, ways of thinking, and methodology that are commonly accepted by members of a scientific community (from Greek *parádeigma* "pattern, model, precedent, example").[1]

Programming: the act or process of planning or writing a program.[2]

Multi-paradigm

A *multi-paradigm* language supports more than one *programming paradigm.*

For example, the C++ programming language supports the paradigms:

- Imperative
- Procedural
- Object oriented
- Functional
- Generic
- Modular

Paradigms

Imperative

The *imperative programming paradigm* changes a program's state by issuing commands to a computer's processing unit in a step-by-step manner.

There are three significant imperative paradigms: von Neumann, procedural, and object oriented.

Example

An imperative program to compute the area of a circle:

1. Get a value representing a radius from the user
2. Square that value
3. Multiply that value by π
4. Report the result to the user

Analogy

The *plot device* of the *treasure map* common to pirate and adventure stories is an example of the imperative paradigm as they instruct the characters in the story how to move step-by-step to their goal (see Indiana Jones films).

[1] https://www.dictionary.com/browse/paradigm
[2] https://www.dictionary.com/browse/programming

What is the opposite of the *imperative* paradigm?

The opposite of the imperative paradigm is the declarative paradigm. In a declarative language, the code expresses the logic behind a computation, not the steps taken to produce the computation. Think of a declarative language as describing *what* the program must do, not *how* the program will do it.

von Neumann

The *von Neumann* paradigm derives its name from the von Neumann (computer) architecture that almost all modern computers follow.

The von Neumann architecture implements a state machine with the following hardware components:

- A unit to process arithmetic and logic that contains registers to hold the value being worked upon (the current state).
- A control unit that holds the current instruction and a pointer or index to the current or next instruction to be executed by the processor.
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms

The von Neumann paradigm is considered an isomorphism between a programming language and a von Neumann architecture computer.

The essential features of a von Neumann paradigm language are:

- An I/O mechanism, such as C's `printf()/scanf()` functions, or C++'s `cout/cin` objects.
- Program variables, such as `int, float, bool`.
- Expressions, such as `x = 2.0 * radius * radius * std::numbers::pi;`
- Control statements, such as if-then-else, for and while loops.
- Mass storage facilities, such as C's `fopen()/fclose()` functions, and C++'s `fstreams`.

Procedural

The procedural paradigm expands the von Neumann paradigm with the addition of *procedure calls*.

The generic term for *procedure call* is *callable unit*. Other names describing the *procedure call* concept are *function*, *routine*, *subroutine*, *subprogram*, and within object-oriented programming – the *method*. In C++, operator overloading is also a form of *procedure call*.

Purpose

The primary purpose of the procedural paradigm is to split up code into cohesive blocks and make that code accessible by a meaningful name.

Syntax

Syntax varies from language to language, but procedures have two fundamental elements: the interface that includes the name and parameters (and optionally a return type), and the implementation.

Example

The following is a C function that computes the area of a triangle using Heron's formula:

```
#include <math.h>

double area_from_sides(double a, double b, double c)       ← interface
{                                                          |
    double s = (a + b + c)/2;                              |
    return sqrt(s*(s - a)*(s - b)*(s - c));                | ← implementation
}                                                          |
```

Note that this function calls a function from a C math library – `sqrt()`.

The same function in Python:

```
import math

def areaFromSides(a, b, c):                          ← interface
    s = (a + b + c)/2                                |
    return math.sqrt(s*(s - a)*(s - b)*(s - c))      |← implementation
```

Benefits

As programs grew in the size of the code, it became increasingly more difficult for programmers to understand and manage the code – the proverbial "can't see the forest for the trees problem."

Procedural programming benefits the programmer by:

- **Reducing program size**. Repeated use of the same code doesn't require writing the code each time its effect is needed. A *call* to the procedure executes the associated code, then returns to the location from which the procedure was called.
- **Hiding complexity**. The details of a computation need not be visible at point of the call allowing the programmer to focus on the use of the computation, and not the implementation of the computation.
- **Higher-order thinking**. By giving a name to the computation a programmer can think more in terms of outcomes and less in terms of steps.
- **Improving performance**. By reducing the overall use of memory, more of a program's code will fit into the CPU cache thereby reducing the frequency of delays caused by reloading the CPU cache. A smaller overall memory footprint also provides performance gains as the operating system has less need to manage the virtual memory system.
- **Faster program development**. Typically, a programmer ends up thinking about functions in the same way that they think about fundamental commands – as the building blocks of their program. Reuse avoids having to rewrite the same code. Procedures can be stored in libraries, precompiled, and shared.
- **Shared wisdom**. Programmers need only understand the complexity of the code that they wrote. Programmers can become experts in different areas and share their expertise through their libraries.
- **Easier testing**. Programmers will not have test the entire program all at once. Unit test can be employed to test the parts of a program separately from the whole.
- **Easier maintenance**. Bug fixes within procedures are immediately applied to any program calling that procedure.

Terminology

I find the terminology not as precise as I would like it to be. Here is a list that I hope clarifies some of ambiguity.

Function	A callable unit that returns a value through the name as functions in mathematics do. Functions can be called within an expression. Example: `a = b * sqrt(c);` Functions are typically named with a noun by the value they return.
Procedure, routine, subroutine	Callable units that represent actions. Example: `printf("Display this on a console\n");` Procedures are typically named with a verb indicating what the procedure will do.
Method	A callable unit linked to a class or object.

Object-oriented

WIP

References

[Programming Paradigms: Wikipedia]

Document History

Version	Date	Activity
0.0.0	2023-09-15	Document created.
0.1.0	2023-09-16	von Neumann paradigm added.
0.2.0	2023-12-29	Procedural paradigm added.