Programming Paradigms and Object-Oriented Concepts

Sujith Samuel Mathew, College of Interdisciplinary Studies (CIS), Zayed University, Abu Dhabi, UAE.

1. What is a programming paradigm?

A computer program is a set of instructions for a computer to execute a task, written in a programming language like Python or Java. A programming paradigm categorizes a programming language on its features and structure. Here we discuss three programming paradigms, unstructured programming, structured programming, and object-oriented programming.

2. Unstructured Programming

It is the earliest programming paradigm. The programs are written as a single continuous block, as shown in Figure 1. The data used in the program would be *Global Data*.

- **Global Data:** These are variables that store data and are not within any function. Instead these variables are accessible for the entire life-cycle of the program and can be accessed from anywhere within the program.

Some of the early programming languages supported unstructured programming, like Assembly Language, JOSS, FOCAL, TELCOMP, MS-DOS batch files, and early versions of BASIC, Fortran, COBOL, and MUMPS.

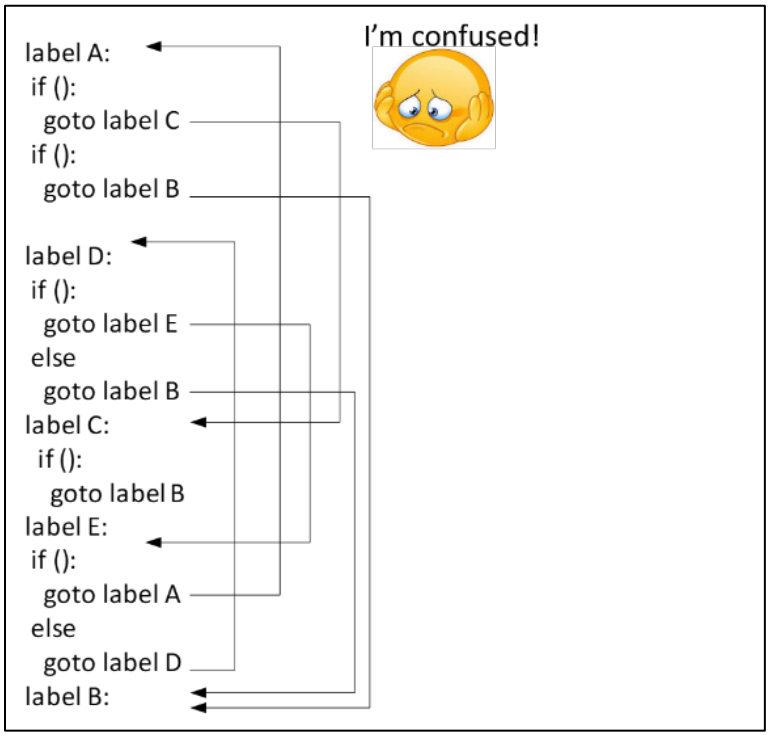

Figure 1: Unstructured program

- Advantages of Unstructured programs:
    - Freedom to express and test logical sequence of execution of small programs.
    - Typically used as a scratchpad to check functionality.
- Disadvantages of Unstructured programs:
    - The data is global, and the code operates on it. This creates the possibility of unintentional changes to the state of the code.
    - Reusability of code is not supported.
    - Very difficult to manage, modify, scale, and debug unstructured programs
    - Large programs tend to turn into *Spaghetti Code*.

- **What is spaghetti code?**
  Spaghetti code is the general term used for a computer program that's hard to understand. While the program may work, it is difficult to follow or read as the code is too convoluted—like a bowl of tangled spaghetti. When the size of an unstructured program is small it would be easy to read and understand, but as the size of the program increases, it becomes spaghetti code, and it becomes very difficult to read or manage.

3. Structured Programming

Structured Programming introduced the concept of Modularity.

- **Modularity**: A modular program is decomposed into a set of cohesive but loosely coupled modules or functions or procedures

The data used in a structured program can be Local Data or Global Data.

- **Local Data**: The variables that are declared within a function body. Their scope is limited to within the function. Local variables cannot be accessed outside the function.
- **Global Data**: These are variables that are declared outside all functions. Their scope is within the whole program and is also accessible within the functions

Structured Programming is a very powerful concept and is supported by all modern programming languages. Sometimes structured programming is also referred to as *procedural programming*. In structured programming, common functionalities are grouped into separate, smaller modules or functions. The code within a function is written and maintained independently of other functions.

- Advantages of Structured Programming
    - Easy to manage, modify and debug.
    - Data can be either global or local.
    - Reusability of code is supported.

- Disadvantages of Structured Programming
    - As the code size grows large, maintaining the code becomes difficult.
    - Any major changes required to the functionality may result in a lot of changes in the code.
    - Very large programs may still tend to turn into Spaghetti Code, if coding standards are not followed and the code is not well documented.
    - Difficult to represent real-life entities and relationships.

4. Software Complexity

Software is a collection of interoperable computer **programs**. A computer program is a logical sequence of **instructions** to achieve a certain requirement. An instruction (command) is a combination of **bits** (0s and 1s) that is understood by a device (computer).

Real-World systems are becoming more and more reliant on software. For example, systems used in Banking, Transportation, Navigation, Health Care, Education, and Security are all heavily reliant on software. For all real-world systems, the software has become very complex and the relative size has increased enormously. The need to manage software has become very critical in all areas of life. Therefore, the impact of reducing software complexity is important in terms of reducing cost, reducing the effort to build and maintain software, and many times, life-saving, because software control real-world systems.

One major reason for software complexity is the difficulty in understanding the software model which represents a software system[1]. Having a good software model (like the UML[2] diagrams) helps to translate ideas (requirements) into working software and it also helps to easily change/update software based on new requirements. Another reason for software complexity is the inability of programming languages to easily map real-world entities into programs, and also the inability to manage the growing size and complexity of writing programs. Fortunately, we have the Object-Oriented programming paradigm which helps to overcome these issues.

5. Object-Oriented Programming (OOP)

OOP is a programming paradigm that relies on the concept of an *Object*. Objects are small independent units of code. These objects define the attributes of a real-world entity and emulate its behaviors. The concept closely relates to the real world and therefore this also helps to define and understand relationships between objects. OOP uses the Bottom-Up approach to programming, which pieces together smaller units of code to give rise to the larger systems. In a bottom-up approach the smaller parts of the system are first specified in detail and then these are integrated. A program or software is created as a collection of interacting objects. OOP enables data hiding features and therefore it is more data security. These programs are easy to manage, modify, scale and debug.

6. Comparing Structured and OO Programming Paradigms

Both structured programming and object-oriented programming are widely used to create software. A comparison of these paradigms is presented.

Structured Programming	**Object-Oriented Programming**
In structured programming, the program is divided into small parts called *functions*.	Object-oriented programming divides the program into small independent parts called *objects*.

[1] *Evaluating and Mitigating the Impact of Complexity in Software Models*, TECHNICAL REPORT, CMU/SEI-2015-TR-013, Carnegie Mellon University https://resources.sei.cmu.edu/asset_files/TechnicalReport/2015_005_001_448093.pdf

[2] Unified Modeling Language (UML) is a standard set of notations to gather, understand, and design the requirements and functionalities of a software.

Structured programming follows a *top-down approach*.	Object-oriented programming follows a *bottom-up approach*. Here, the objects are first created and then assembled together.
There is no access specifier in structured programming. Structured programming does not have any proper way of hiding data, so it is *less secure*.	Object-oriented programming has access specifiers like private, public, and protected to provide data security. Object-oriented programming provides data hiding so it is *more secure*.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Code reusability is comparatively limited in structured programming.	Code reusability is a major benefit of object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	Examples: C++, Java, Python, C#, etc.

7. The Class

An object in any software context is a uniquely identifiable representation of a real-world entity. In other words, any real-world entity that has to be represented in a program is considered to be an object. For example, John, who is an employee in an organization, Samir, who is a student at the university, or my red colored Toyota Camry.

A class is the template of an object. It is used to create or model an object. Therefore,

- A class **Car** can be used to create any instance of a car, like a red-colored Toyota Camry or a black-colored Nissan Patrol.
- A class **Student** can be used to create any instance of a student, like Samir or Mariam who are students in a university.
- A class **Employee** can be used to create any instance of an employee, like John or Carson, who works in an organization.

8. Creating an object

- How do we represent a real-world entity like a car in an object-oriented program?
    - First identify the object.
        - For example, a 2017 model red colored Toyota Camry, let's call it *myCar*
        - So, ***myCar*** is an object that is to be represented in a program.
    - Next we need a class (template) for the object.
        - Let's call the class – **Car**
        - So, both these statements are true.
            - **Car** is the class that will help model or create the object **myCar**
            - **myCar** is an object which is an instance of the class **Car**
    - Python syntax to create classes and objects.

```
class Car:
    """Class to represent a car"""
    pass

# Create an object of class Car
myCar = Car()
```

- In the code above:
  - In the first line, the keyword **`class`** defines the class that would model or define an actual car.
  - The second line in the code
    - It is indented to be included in the class
    - The line is NOT a comment. It is called the **doc_string** and is used to provide documentation for a class.
  - The third line **`pass`** is a placeholder for code that we will write in the future. Nothing happens when this line is executed.
  - The fourth line is a comment
  - The fifth line is where the object is created as an instance of a particular class.
    - *myCar* is created as an object of the class *Car*

9. Identifying attributes of an object

- Typically, attributes are nouns that define the object's characteristics.
- When attributes are selected, it is important to also identify the data type of each attribute.
- While there can be many attributes that define a car, let's identify seven attributes of *myCar*:
  - `manufacturer :String`
  - `model :String`
  - `color :ENUM`
  - `numDoors :Integer`
  - `fuelCapacity :Float`
  - `engineState :Boolean`
  - `currentSpeed :Float`

**ENUM stands for enumerated value, which is a list of possible values for the attribute. For example, color would be one of the colors from a list like,* ['Blue Green', 'Silver White', 'Rock Grey'].

10. State of an object

The state of an object is defined by **the value** of its attributes. For example, the state of myCar changes depending on the values of the attributes *engineState* and *currentSpeed*.

  - engineState :Boolean
  - currentSpeed :Float
- myCar could be in a state that the engine is On and speed is 0
- myCar could be in a state that the engine is On and speed is 40
- myCar could be in a state that the engine is Off, and speed is 0
- myCar **CANNOT** be in a state that the engine is Off, and speed is 40

Therefore, the state myCar is determined by the changes to the attributes, and therefore it is **important to protect the state of an object, by protecting the values assigned to the attributes.** The values of the attributes are accessed, assigned and modified using the behaviors of the object.

11. Identifying the behaviors of an object

A behavior is an action that retrieves or changes the state of an object. It provides the means for interaction with the object and is also called a function or method. They are verbs that define the object's action or function. For example, myCar would have functions like:

- `turnLeft(angle: Float) :Boolean`
    - The car turns left at a certain angle, and the function returns True or False
- `turnRight(angle: Float) :Boolean`
    - The car turns right at a certain angle, and the function returns True or False
- `stopCar() :Boolean`
    - The car stops, and the function returns True or False
- `accelerate(speed: Float) : Boolean`
    - The car increases speed at a certain value, and returns True or False
- `checkRemainDistance(remainFuel:Float, currentSpeed:Float): remainingDistance:Float`
    - Calculate remaining distance based on remaining fuel, and current speed. The function returns the distance which is a float value

Also, for all the attributes identified there are the associated setter (mutator) functions and getter (accessor) functions also known as the setter/getter functions.

- Getter functions like:
    - `getManufacturer(): String`
    - `getNumberOfDoors() :Integer`
    - `getEngineState() :Boolean`
    - `getCurrentSpeed() :Float`
- Setter functions like:
    - `setEngineState(Boolean):Boolean`
    - `setCurrentSpeed(float) :Boolean`
    - `setColor(ENUM) :Boolean`

12. Encapsulation

Unlike earlier programming paradigms, in OO programming an object encapsulates the attributes, behaviors, and states of a real-world entity in a single unit. The state of an object represented by the value of the attributes, is hidden inside the object and is accessible via the behaviors or functions (see Figure 3).

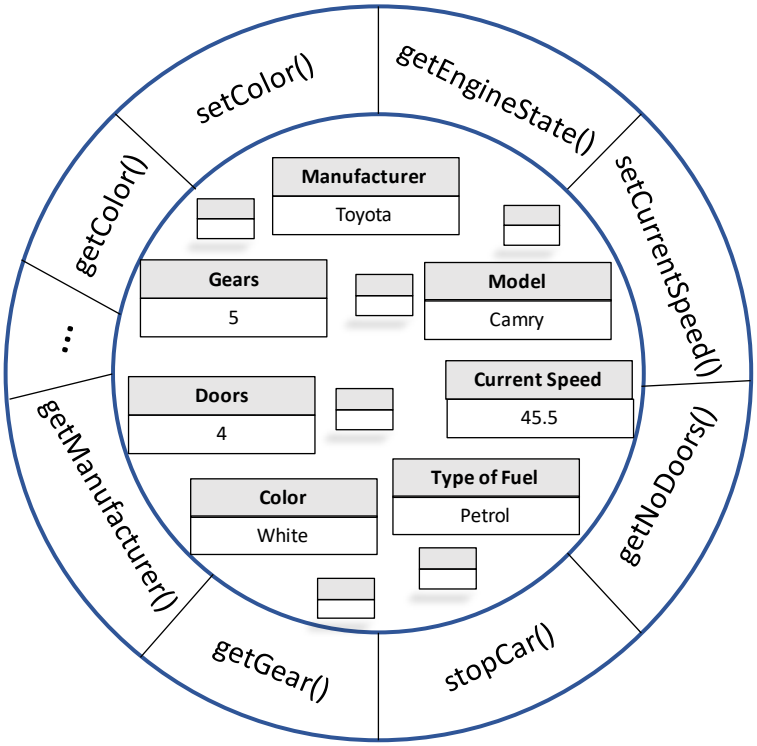

Figure 2: Encapsulation of attributes and behaviors

For instance, in Figure 2, the *currentSpeed* of the car should not be accessed by another entity without proper access rights and therefore the access to *currentSpeet* is only through the functions *setCurrentSpeer()* and *getCurrentSpeed()*. This concept of Encapsulation is also understood as Information Hiding and this provides the required protection or security for the attributes to ensure that the object does not get into an error state.

13. Abstraction

Abstraction is the concept of keeping or removing certain attributes from an object in order to present it only with essential attributes. Abstraction simplifies the understanding and use of complex systems by not including data that is not relevant to the context of the software. It reduces complexity and increases efficiency.

For example, a Car class created for a children's game would have fewer attributes than a Car class created for the automotive industry. The *myCar* object that was created here does not include all the attributes of a real-world car. Also, functions for the engine's working or the shifting of gears were not included. The *myCar* object we created, is the abstract form of the real-world car. Abstraction enables an object to have only those functions and attributes that are required for the context of the software being built.

14. Sample Python class

An example of a Python class is provided in Figure 3, to help understand the syntax and usage of a class to create objects.

- All lines within a class are indented to be inside the class.
- The __init__() function, called the **constructor** of a class, declares and defines all the attributes of the class. The constructor is called when an object is first created, which thereby initializes the attributes of the object.
- The functions provide access to the attributes and also help to define the behaviors of the object

```
class Car:
    """Class to define a car"""

    # Constructor defines attributes
    def __init__(self):
        self.manufacturer = ""  # String
        self.numDoors = 0  # integer
        self.remainingFuel = 0.0  # Float
        self.engineState = False  # Boolean
        self.currentSpeed = 0.0  # Float

    # Function to define behaviors
    def checkRemainDistance(self, remainingFuel=0.0, currentSpeed=0.0):
        pass

    # Setter/Getter Functions
    def setmanufacturer(self, manufacturer=""):
        self.manufacturer = manufacturer
    def getmanufacturer(self):
        return self.manufacturer

    def setenginestate(self, state = False):
        self.engineState = state
    def getenginestate(self):
        return self.engineState

# Creating objects of class Car
myCar = Car()
thatCar = Car()
# Set and get values of objects
myCar.setmanufacturer("Toyota")
thatCar.setmanufacturer("Nissan")
print("My car is a ", myCar.getmanufacturer())
```

Figure 3: Example of a class, written in Python with some attributes and some behaviors

Once the class is created, then the objects are created. For instance, the code above includes two objects of the class Car i.e. *myCar* and *thatCar*. Each object will have a copy of the attributes and behaviors defined in the class, for itself.

15. Creating a software object

Let's consider another example to see how a real-world entity is represented as a software object. Say, we had to represent a university student as a software object in a program.

1. Let's call the object *zuStudent1* and the class to create the object *Student*.
2. We create the skeleton of the class with the docstring.

```
class Student:
    """Class to represent a ZU student"""
    pass
```

3. Next we identify attributes of the class. For this example, let's limit the number of attributes to four.
    a. For representing the attributes, we need to include the class constructor
    b. In Python, the constructor is the built-in function **__init__()**.
    c. The **pass** is removed and replaced with the constructor

```
class Student:
    """Class to represent a ZU student"""

    # Constructor
    def __init__(self):
        self.studentid = ""
        self.studentname = ""
        self.studentgpa = 0.0
        self.studentage = 0
```

4. Next we write the setter and getter functions for the attributes.

```
class Student:
    """Class to represent a ZU student"""

    # Constructor
    def __init__(self):
        self.studentid = ""
        self.studentname = ""
        self.studentgpa = 0.0
        self.studentage = 0

    # Setter/Getter functions
    def setstudentid(self, id = ""):
        self.studentid = id
    def getstudentid(self):
        return self.studentid

    def setstudentname(self, name = ""):
        self.studentname = name
    def getstudentname(self):
        return self.studentname
```

5. Finally, the object of the class is created, and the values of the attributes are set.

```
# Create object
zuStudent1 = Student()
zuStudent1.setstudentid("201938387")
zuStudent1.setstudentname("Jaffer Ali")
print("Name: ", zuStudent1.getstudentname())
print("ID: ", zuStudent1.getstudentid())
```

6. **Note**: The example above is only partially implemented. The reader is encouraged to add a few more attributes and complete the respective setter/getter functions as shown above and run the program to see the output of the program.

16. Conclusion

The Object-Oriented Programming (OOP) paradigm has changed the landscape of computer programming to create a better way to write, debug, manage, and scale software. OOP allows programmers to map real-world entities as software objects, and also represent the relationships between objects. With capabilities of encapsulation and abstraction, OOP provides flexibility to create robust software applications.