

Amrita Vishwa Vidyapeetham
TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

Sudhir R.T

21st January, 2024

Paradigm 1: Imperative Programming

A paradigm is also a way of approaching a problem or carrying out a task. A programming paradigm is a way of approaching problems with programming languages, or alternatively, it is a way of approaching problems with the tools and techniques at our disposal while adhering to a certain approach. There are many well-known programming languages, but when they are used, they all need to adhere to a technique or strategy, and this approach is known as paradigms. In addition to several programming languages, there are numerous paradigms to meet every need. See Figure 1.

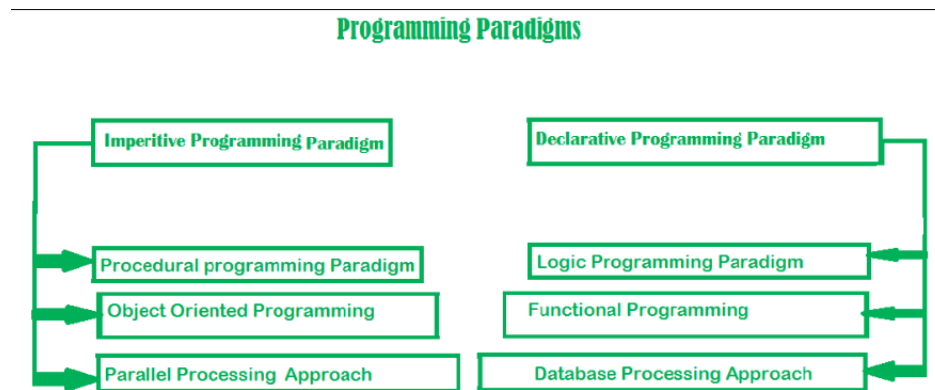


Figure 1: Programming Paradigms

In computer science, imperative programming is a software development methodology that uses state-ments to change a program's state. In imperative programming, functions are implicitly coded at every stage required to address a problem; therefore, pre-coded models are not utilized. Imperative programming instructs the computer "how" to perform a task, as opposed to declarative programming, which tells the computer "what" the program should do.

An imperative program includes commands for the computer to execute.

Advantages:

- Very simple to implement
- It contains loops, variables etc.

Disadvantages:

- Complex problem cannot be solved
- Less efficient and less productive
- Parallel programming is not possible

The imperative programming paradigm is based on the idea that a program's state can be changed and manipulated by a sequence of statements. The concept of a program state, which denotes the values kept in variables at a specific moment throughout the program's execution, is at the heart of this paradigm. Data is stored and manipulated via variables, and maintaining the program's state depends significantly on the assignments made to them.

The idea of assignment is one of the cornerstones of imperative programming. When variables in imperative languages are assigned values, the state of the program is altered.

Another integral part is the notion of control flow. Programmers can control how the execution proceeds depending on certain conditions by using constructs like conditionals (if-else) and loops (for, while). Since the sequence of statements dictates the overall flow of the program, sequential execution is emphasized.

Imperative programs frequently break down into procedures or functions, which contain sequences of steps to complete certain tasks. Procedural abstraction allows programmers to modularize code, breaking down complicated issues into more manageable components.

Mutable data structures are widely used here, allowing the content of data structures to be changed after they are created. This mutability allows for efficient state manipulation and is frequently connected with side effects, in which the execution of a statement might result in observable consequences outside of the immediate scope.

In imperative languages, programmers may explicitly control the execution flow and state changes. The step-by-step instructions for a certain task are explicitly stated in the code, resulting in a straightforward approach for problem resolution.

Full control is very important to developers in low-level programming.

Imperative programming languages often have input and output methods, which allow for communication with the external environment. Statements for input and output make it easier to communicate with people or other systems, making imperative languages ideal for an extensive range of applications.

Furthermore, parallel processing approach, object-oriented programming, and procedural programming are the types of imperative programming.

Language for Paradigm 1: C++

The predominant paradigm in C++ programming is imperative programming, and the language offers an abundance of features for expressing imperative constructs. In the context of C++, imperative programming has the following features:

- Variables and State:

C++ allows the declaration of variables to store data, and these variables can be modified throughout the program, altering the program's state.

```
int main() {  
    // Variable declaration and state modification  
    int x = 5;  
    x = x + 3; // Modifying the state  
    return 0;  
}
```

Figure 2: Example of declaring a variable and modifying its value.

- Assignment:

Assignment is a fundamental concept in C++. It is used to assign values to variables, thereby updating the program's state.

```
int main() {  
    // Assignment updating program state  
    int y = 10;  
    y = y * 2; // Update using assignment  
    return 0;  
}
```

Figure 3: Assigning new value to a variable.

- Control Flow:

C++ provides constructs for controlling the flow of execution, including if, else, for, while, and do-while statements.

If and else are conditionals.

For, While and Do while are loops.

```
int main() {
    // Control flow in C++
    int a = 8;
    if (a > 5) {
        // Code executed if condition is true
        a = a - 5;
    } else {
        // Code executed if condition is false
        a = a + 5;
    }

    for (int i = 0; i < 2; i++)
    {
        a++; //code executed for 2 times here(i=0 and i=1)
    }

    while (a!= 0)
    {
        a--; //code executed until condition is false.
    }

    return 0;
}
```

Figure 4: Conditions and Loops

- Procedural Abstraction:

C++ supports procedural abstraction through functions. Functions encapsulate a sequence of statements and can be called with different arguments.

```
// Function declaration
int add(int x, int y) {
    return x + y;
}

int main() {
    // Procedural abstraction using functions
    int result = add(3, 4);
    return 0;
}
```

Figure 5: Example function

- Mutable Data Structures:

C++ allows the creation and manipulation of mutable data structures, such as arrays, vectors, and user-defined classes.

```
#include <vector>

int main() {
    // Mutable data structures in C++
    std::vector<int> numbers = {1, 2, 3, 4};
    numbers.push_back(5); // Modify the vector
    return 0;
}
```

- Side Effects:

C++ allows side effects, where functions or operations can have effects beyond their return values, such as modifying global variables.

```
// Side effect example
int globalVar = 10;

void modifyGlobal() {
    globalVar = globalVar * 2;
}

int main() {
    modifyGlobal(); // Modifies the global variable
    return 0;
}
```

- Arrays and Pointers:

C++ supports arrays and pointers, enabling the manipulation of memory directly. This allows for efficient data structures and algorithms.

```
int main() {
    // Arrays and Pointers in C++
    int arr[3] = {1, 2, 3};
    int* ptr = arr; // Pointer to the first element of the array

    // Accessing array elements using pointers
    int firstElement = *ptr;
    ++ptr; // Move to the next element

    return 0;
}
```

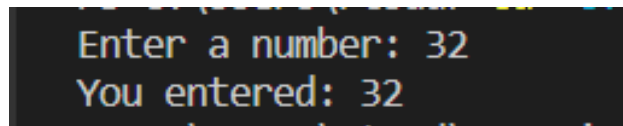
- Input and Output Operations:

C++ provides I/O operations to interact with the user and external devices. The iostream library allows reading from and writing to the console.

```
#include <iostream>

int main() {
    // Input and Output in C++
    int userInput;
    std::cout << "Enter a number: ";
    std::cin >> userInput;
    std::cout << "You entered: " << userInput << std::endl;

    return 0;
}
```



```
Enter a number: 32
You entered: 32
```

Paradigm 2: Aspect-Oriented Programming

Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be characterized as the division of code into distinct modules, commonly referred to as modularization, in which the aspect serves as the primary unit of modularity . Features make it possible to implement overarching issues like transaction and logging that are not essential to business logic without complicating the code that is essential to its operation. It accomplishes this by advising the current code with new behavior. Security, for instance, is a cross-cutting concern since security rules can be applied to multiple methods inside an application. This means that code must be repeated at each method, functionality must be defined in a common class, and controls must be placed to apply that functionality throughout the entire program.

Aspect-oriented programming is a technique for building common, reusable routines that can be applied application wide. During development this facilitates separation of core application logic and common, repeatable tasks (input validation, logging, error handling, etc.).

It works by adding behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with 'set'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code core to the functionality.

Understanding the problem:

An aspect is usually scattered or tangled and twisted as code, which makes it more difficult to comprehend and update. It is dispersed because a function (like logging) may be used by several unrelated functions, some of which may be in completely unrelated systems or have distinct source languages. As a result, altering the logging may need altering every module that is impacted. Aspects get intertwined with each other as well as with the primary function of the systems in which they are expressed. Hence, in order to address one issue, it is necessary to comprehend all of the interconnected issues or have a method for determining how

changes may affect other issues.

For example, consider a banking application with a conceptually very simple method for transferring an amount from one account to another:

```
void transfer(Account fromAcc, Account toAcc, int amount) throws Exception {
    if (fromAcc.getBalance() < amount)
        throw new InsufficientFundsException();

    fromAcc.withdraw(amount);
    toAcc.deposit(amount);
}
```

However, this transfer method overlooks certain considerations that a deployed application would require, such as verifying that the current user is authorized to perform this operation, encapsulating database transactions to prevent accidental data loss, and logging the operation for diagnostic purposes.

A version with all those new concerns could look somewhat like this:

```
void transfer(Account fromAcc, Account toAcc, int amount, User user,
    Logger logger, Database database) throws Exception {
    logger.info("Transferring money...");

    if (!isUserAuthorised(user, fromAcc)) {
        logger.info("User has no permission.");
        throw new UnauthorisedUserException();
    }

    if (fromAcc.getBalance() < amount) {
        logger.info("Insufficient funds.");
        throw new InsufficientFundsException();
    }

    fromAcc.withdraw(amount);
    toAcc.deposit(amount);

    database.commitChanges(); // Atomic operation.

    logger.info("Transaction successful.");
}
```

In this example, other interests have become tangled with the basic functionality (sometimes called the business logic concern). Transactions, security, and logging all exemplify cross-cutting concerns.

Now consider what would happen if we suddenly need to change the security considerations for the application. In the program's current version, security-related operations appear scattered across numerous methods, and such a change would require a major effort.

AOP attempts to solve this problem by allowing the programmer to express cross-cutting concerns in stand-alone modules called aspects.

So for the example above implementing logging in an aspect:

```
aspect Logger {
    void Bank.transfer(Account fromAcc, Account toAcc, int amount, User user, Logger logger) {
```

```

    logger.info("Transferring money...");
}

void Bank.getMoneyBack(User user, int transactionId, Logger logger) {
    logger.info("User requested money back.");
}

// Other crosscutting code.
}

```

Here are some key principles and concepts of Aspect-Oriented Programming:

Aspect: A module that encompasses a cross-cutting issue is called an aspect. It specifies a set of join points, which are specific points in a program's execution (like method calls or object instantiations), as well as the advice which is code that needs to run at each join point. Concerns that could otherwise be dispersed throughout the codebase are isolated and made modular with the aid of aspects.

Join Point: A join point is a specific point in the execution of a program, such as the execution of a method or the handling of an exception. Join points are defined in the context of an aspect and serve as the points where the advice associated with that aspect can be applied.

Advice: Advice is the code that runs at a specified join point. There are different types of advice, including:

- Before advice: Executed before the join point.
- After returning advice: Executed after the join point completes normally.
- After throwing advice: Executed if the join point throws an exception.
- After advice: Executed after the join point, regardless of its outcome.
- Around advice: Most powerful type, allowing the aspect to completely control the execution of the join point.

Pointcut: A pointcut is a set of join points. It defines a set of criteria that match specific join points in the program's execution. Pointcuts allow aspects to specify where their advice should be applied. They help in selecting the appropriate places in the code to weave the cross-cutting concerns.

Weaving: Weaving is the process of integrating aspects into the main business logic of the program. There are two main types of weaving:

- Compile-time weaving: Aspects are woven into the code at compile time.
- Runtime weaving: Aspects are woven into the code at runtime.

Cross-cutting Concerns: These are concerns that affect multiple modules or components in a program, such as logging, security, and error handling. AOP aims to address the modularization and encapsulation of these cross-cutting concerns.

Language for Paradigm 2: AspectJ

AspectJ is an aspect-oriented programming (AOP) extension created at PARC for the Java programming language.

AspectJ has become a widely used de facto standard for AOP by emphasizing simplicity and usability for end users. It uses Java-like syntax, and included IDE integrations for displaying crosscutting structure.

Aspect Declaration:

AspectJ allows you to declare aspects using the aspect keyword.

```
public aspect LoggingAspect {
```

```
    // Aspect-specific code goes here
}
```

Pointcut Declarations:

AspectJ enables the definition of pointcuts, which specify where the advice should be applied.

```
pointcut myPointcut(): execution(* com.example.MyClass.myMethod());
```

Advice Definitions:

AspectJ provides various types of advice to execute code at different points in the program's execution.

```
after() returning: myPointcut() {
    System.out.println("After method execution");
}
```

Join Points:

AspectJ supports various join points like method executions, object instantiations, field access, etc.

```
pointcut methodExecution(): execution(* com.example.MyClass.*());
```

```
before(): methodExecution() {
    System.out.println("Before method execution");
}
```

Introduction:

AspectJ allows you to introduce new fields and methods to existing classes.

```
public aspect MyIntroduction {
    private String MyAspect.myField = "Aspect-introduced field";

    public String MyAspect.getMyField() {
        return myField;
    }
}
```

Weaving:

AspectJ supports both compile-time and runtime weaving.

Compile-time weaving: This is typically done during the build process using the AspectJ compiler (ajc).

```
ajc -cp aspectjrt.jar MyClass.java LoggingAspect.java
```

Runtime weaving: You can use tools like load-time weaving (LTW) to weave aspects at runtime.

AspectJ Annotations:

AspectJ supports annotations for defining aspects and pointcuts, providing an alternative to the traditional syntax.

@Aspect

```
public class LoggingAspect {
    @Pointcut("execution(* com.example.MyClass.myMethod())")
    public void myPointcut() {}

    @AfterReturning("myPointcut()")
    public void afterMyMethodExecution() {
        System.out.println("After myMethod execution");
    }
}
```

Aspect Libraries:

AspectJ provides a rich set of libraries and tools for AOP, making it easier to handle common cross-cutting concerns.

```
public aspect ExceptionHandlingAspect {
    // Handle exceptions globally
    after() throwing(Exception e): handler(Exception) {
        System.out.println("Exception caught: " + e.getMessage());
    }
}
```

Aspect Inheritance:

Aspects in AspectJ can extend other aspects, enabling the reuse of aspect code.

```
public aspect CommonLoggingAspect {
    before(): execution(* com.example.*.*()) {
        System.out.println("Logging common to all methods");
    }
}

public aspect SpecificLoggingAspect extends CommonLoggingAspect {
    // Additional aspect-specific code goes here
}

.
```

Analysis

Imperative Programming (C++):

Strengths:

- **Performance:** Imperative languages like C++ allow fine-grained control over memory and system resources, resulting in high-performance applications.
- **Low-Level Access:** C++ provides low-level features like pointers and manual memory management, giving developers precise control over system resources.
- **Procedural Paradigm:** Well-suited for procedural programming, making it suitable for tasks where step-by-step execution is natural.

Weaknesses:

- **Complexity:** The manual memory management and low-level features can lead to complex and error-prone code.
- **Concurrency Issues:** Concurrency and parallelism require careful consideration and are prone to issues like race conditions and deadlocks.

-
- **Verbosity:** Code can be verbose, requiring more lines of code to achieve certain tasks compared to higher-level languages.

Notable Features:

- **Object-Oriented Programming (OOP):** C++ supports OOP principles, allowing encapsulation, inheritance, and polymorphism.
- **Template Metaprogramming:** C++ templates enable compile-time metaprogramming, providing powerful abstractions and generic programming capabilities.
- **Performance-Critical Applications:** C++ is often chosen for system-level programming, game development, and other performance-critical applications.

Aspect-Oriented Programming (AspectJ):

Strengths:

- **Modularity:** AOP promotes better modularity by separating cross-cutting concerns from the main business logic, making code more maintainable and understandable.
- **Cross-Cutting Concerns:** Provides a clean and centralized way to address cross-cutting concerns, such as logging, security, and error handling.
- **Reusability:** Aspects can be reused across multiple parts of the codebase, reducing duplication of concern-specific code.

Weaknesses:

- **Learning Curve:** AOP concepts and tools may have a steeper learning curve for developers unfamiliar with the paradigm.
- **Tool Support:** While there are tools like AspectJ for Java, not all programming languages have mature and widely adopted AOP implementations.
- **Overuse of Aspects:** Overusing aspects can lead to a codebase that is difficult to understand and maintain.

Notable Features:

- **AspectJ Language Extensions:** AspectJ provides language extensions for defining aspects, pointcuts, and advice, making it a comprehensive AOP solution for Java.
- **Dynamic Weaving:** AspectJ supports runtime weaving, allowing aspects to be applied dynamically, which can be useful for certain scenarios.

-
- Aspect Libraries: AspectJ comes with a rich set of libraries and tools, making it easier to address common cross-cutting concerns.

Comparison

Imperative Programming vs. AOP:

Common Strengths:

- Both paradigms aim to improve code organization and maintainability.
- Both can be used for developing performance-critical applications.

Differences:

- Abstraction Level: Imperative programming focuses on step-by-step instructions, while AOP introduces higher-level abstractions for cross-cutting concerns.
- Code Organization: AOP excels in organizing code related to cross-cutting concerns, while imperative programming relies on procedural and object-oriented structures.
- Flexibility: Imperative programming provides low-level control, while AOP offers flexibility in managing cross-cutting concerns without scattering code throughout the application.

Challenges Faced

For 1st paradigm trying to bring down the content to its crux points was tough. And for the 2nd not enough reliable resources were available.

Conclusion

- Choosing between imperative programming and AOP often depends on the nature of the application and the development team's familiarity with the paradigms.
- In scenarios where low-level control and performance are crucial, imperative languages like C++ are appropriate.
- For applications with complex cross-cutting concerns and a need for modularity, AOP can be beneficial, especially when using languages like Java with mature AOP implementations like AspectJ.
- Imperative programming in C++ is characterized by its high performance, enabled by fine-grained control over memory and resources. Offering low-level features like pointers and manual memory management, it is well-suited for procedural programming tasks. However, this approach can lead to complexity and potential errors, especially with challenges related to concurrency. Despite its advantages, C++ code tends to be more verbose compared to higher-level languages.

-
- Aspect-Oriented Programming (AOP) with AspectJ enhances modularity by separating cross-cutting concerns from the core business logic. This paradigm provides a centralized approach to handling common concerns like logging and security, promoting reusability through aspects across different code sections. While AOP introduces a learning curve for developers unfamiliar with its concepts, AspectJ, with its language extensions, dynamic weaving, and rich set of libraries, is a powerful tool for addressing and managing cross-cutting concerns in Java applications.

References

Respective Geeksforgeeks, wikipedia, javatpoint, science direct sites for each topic. Chatgpt for example code generation.