Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages

Assignment-01: Exploring Programming Paradigms

Arjun C Santhosh

21st January, 2024

Paradigm 1: Object-Oriented

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can encapsulate data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions). The primary goals of Object-Oriented Programming are to organize and structure code in a way that promotes modularity, reusability, and extensibility
- few lanagues that support it are C++ , Java , C# , python etc
- Objects
  - Objects are instances of classes and represent real-world entities. They encapsulate data (attributes or properties) and behavior (methods or functions.
- Classes
  - Classes are blueprints or templates for creating objects. They define the structure and behavior that objects instantiated from them will have.
- Encapsulation
  - A desired outcome of organizing code in classes in order to keep things from being mixed with other unrelated bits of code. Encapsulation make it easier to reason about code because of the modularity of code written in object oriented styled classes
- Inheritance
  - A principle which allows an instance of an object to borrow attributes and methods from its parent class
- polymorphism describes the concept that you can access objects of different types through the same interface. Each type can provide its own independent implementation of this interface.

  - The ability of the class to be dynamic in its use of class methods so that objects with the same parent class can make use of these parent class methods

- Abstraction
  - simplifying complex systems by modeling classes based on essential features, hiding unnecessary details, and providing a high-level, generalized interface for interacting with objects

Language for Paradigm 1: C#

- Discuss the characteristics and features of the language associated with Paradigm 1.
- C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in .NET
- It is language created by Microsoft that runs on the .NET Framework.
- Similar to languages like C++ and Java
- Populary used in Desktop applications , Web services ,Games development , Database applications etc

- **Encapsulation** Hiding the internal state and functionality of an object and only allowing access through a public set of functions this achieved in C# via through classes, access specifiers, and properties. - Properties provide a controlled way to access and modify private data member such as
- **Inheritance** Ability to create new abstractions based on existing abstractions but C# only supports single inheritances this means that a class can only inherit from one base class this done such that common diamond inheritance problem The diamond problem arises when a class inherits from two classes that have a common ancestor, potentially causing ambiguity in method or attribute resolution
- **Polymorphism** is Ability to implement inherited properties or methods in different ways across multiple abstractions. there are two types of Polymorphism *Static / Compile Time Polymorphism* and *Dynamic / Runtime Polymorphism* as seen in Figure-1
  - Static polymorphism
    * Method overloading is when in a class there are multiple function with same name but different arguments so according to number parameter function definition may vary
    * Operator overloading allows you to redefine existing operators such as +, -, *, etc., to perform custom operations or manipulations on instances of user-defined classes . we use *operator* keyword followed by the operator to overload when defining the class to perform operator overloading
  - Dynamic/runtime polymorphism

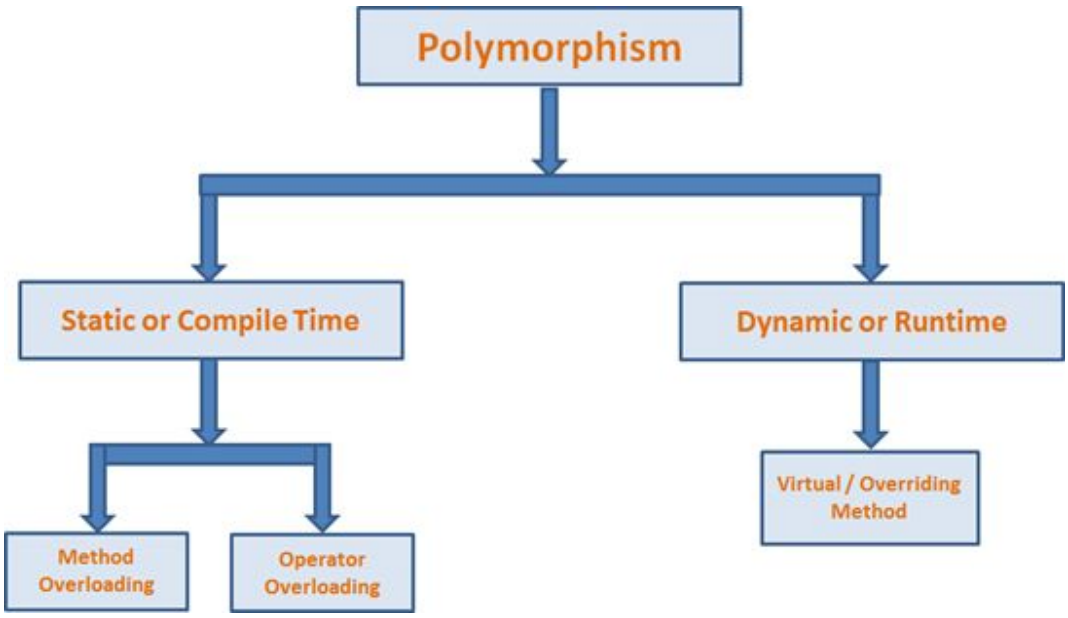

Figure 1: types of polymorphism

  * Virtual/Method Overriding can be done using inheritance. With method overriding, it is possible for the base class and derived class to have the same method name and the same something. The compiler would not be aware of the method available for overriding the functionality, so the compiler does not throw an error at compile time. The compiler will decide which way to call at runtime, and if no method is found, it throws an error.
  * we define a base class with a virtual function that is later overridden we use *virtual* keyword for this and On the child class redefine the function with *overwrite* keyword
    base keyword can be used access the definition of the base class function . *sealed* keyword can be used to block any further Inheritance if needed

- **Abstraction** we implement this by by making classes not associated with any specific instance. Abstraction is needed when we only inherit from a certain class but do not need to instantiate objects of that class. In such a case, the base class can be regarded as "Incomplete". Such classes are known as an "Abstract Base Class". it is achieved by using *abstract* while defining a class and methods in it and abstract class is latter is inherited and we can use *override* to redefine the methods if needed be

Now That we discoursed the Concepts let up look at some examples

- Code in Figure-2 demonstrates the concept of encapsulation in C# In the *Student* class, *name* and *age* are private fields. This means they can't be accessed or modified directly from outside the class.
- The *Student* class provides public methods to get and set the values of these fields.
- The *GetName()* method is a getter for the *name* field. It returns the value of *name* when called.

- The *SetAge()* method is a setter for the age field. It takes an integer *newAge* as a parameter. If *newAge* is greater than 0, it sets the age field to *newAge*. If *newAge* is not greater than 0, it prints an error message to the console. This method ensures that age can't be set to an invalid value.
- This way, the *Student* class has full control over its fields. It can validate new values, reject invalid ones, or even perform some action when a field is changed. This is the essence of encapsulation
- Figure-3 shows how different access modes and level of access to data

main.cs

```
using System;
class Student
{
    private string name;
    private int age;

    public Student(string studentName, int studentAge)
    {
        name = studentName;
        age = studentAge;
    }
    public string GetName()
    {
        return name;
    }
    public void SetAge(int newAge)
    {
        if (newAge > 0)
        {
            age = newAge;
        }
        else
        {
            Console.WriteLine("Invalid age. Age must be a positive number.");
        }
    }
    public void DisplayInfo()
    {
        Console.WriteLine($"Student: {name}, Age: {age}");
    }
}
class Program
{
    static void Main()
    {
        Student student1 = new Student("Alice", 20);
        Console.WriteLine($"Original Age: {student1.GetName()} is {student1.GetName()}");
        student1.SetAge(21);
        Console.WriteLine($"Updated Age: {student1.GetName()} is {student1.GetName()}");
        student1.DisplayInfo();
        Console.ReadLine();
    }
}
```

Figure 2: Encapsulation in C#

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Figure 3: Access modifiers

- Now we will look at a example for Abstraction , as seen in Figure-4
- In this code, *Shape* is an abstract class, which is a class that cannot be instantiated and is meant to be subclassed by other classes. It serves as a blueprint for its subclasses.
- The *Shape* class has an abstract method *Draw()*. Abstract methods are methods declared in an abstract class without any implementation. They must be overridden in any non-abstract class that directly inherits from the abstract class
- The *Circle* and *Rectangle* classes are concrete classes that inherit from *Shape* and provide their own implementations of the *Draw()* method. this is done using the override keyword.

main.cs

```cs
using System;
abstract class Shape
{
    public abstract void Draw();
}
class Circle : Shape
{
    private double radius;
    public Circle(double r)
    {
        radius = r;
    }
    public override void Draw()
    {
        Console.WriteLine($"Drawing a circle with radius {radius}");
    }
}
class Rectangle : Shape
{
    private double length;
    private double width;

    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }
    public override void Draw()
    {
        Console.WriteLine($"Drawing a rectangle with length {length} and width {width}");
    }
}

class Program
{
    static void Main()
    {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);
        circle.Draw();
        rectangle.Draw();
        Console.ReadLine();
    }
}

```

Figure 4: abstraction

- Figure-4 can be used as an example for Inheritance as Shape class is inherited by both Rectangle and Circle
- Figure-5 demonstrates the concept of method overloading, which is a feature in object-oriented programming that allows a class to have multiple methods with the same name but different parameters
- In the *Calculator* class, there are two methods named *Add*. The first *Add* method takes two integer parameters , the second *Add* method takes three integer parameters
- These methods are considered different and the correct one to call is determined by the number of arguments passed in this is known as method overloading, and it increases the readability and re-usability of the code.

main.cs

```
1   using System;
2   class Calculator
3   {
4       public int Add(int a, int b)
5       {
6           return a + b;
7       }
8
9
10      public int Add(int a, int b, int c)
11      {
12          return a + b + c;
13      }
14  }
15
16  class Program
17  {
18      static void Main()
19      {
20          Calculator calculator = new Calculator();
21
22          int sumIntTwo = calculator.Add(5, 10);
23          int sumIntThree = calculator.Add(5, 10, 15);
24
25          Console.WriteLine($"Sum of two integers: {sumIntTwo}");
26          Console.WriteLine($"Sum of three integers: {sumIntThree}");
27      }
28  }
29
```

Figure 5: Method Overloading

- Operator Overloading Example from the Figure-6 we can see that ther exist a ComplexNumber class and in line 14 and line 19 is where + - operators are getting overloaded
- we can see that we specify the operator along the function definition and we give custom behavior to the operator via code in the method and we return the Object of the class

main.cs

```
using System;

class ComplexNumber
{
    public double Real { get; set; }
    public double Imaginary { get; set; }

    public ComplexNumber(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public static ComplexNumber operator +(ComplexNumber num1, ComplexNumber num2)
    {
        return new ComplexNumber(num1.Real + num2.Real, num1.Imaginary + num2.Imaginary);
    }

    public static ComplexNumber operator -(ComplexNumber num1, ComplexNumber num2)
    {
        return new ComplexNumber(num1.Real - num2.Real, num1.Imaginary - num2.Imaginary);
    }

    public override string ToString()
    {
        return $"{Real} + {Imaginary}i";
    }
}

class Program
{
    static void Main()
    {
        ComplexNumber num1 = new ComplexNumber(3, 4);
        ComplexNumber num2 = new ComplexNumber(1, 2);

        ComplexNumber sum = num1 + num2;
        ComplexNumber difference = num1 - num2;

        Console.WriteLine($"Complex Number 1: {num1}");
        Console.WriteLine($"Complex Number 2: {num2}");
        Console.WriteLine($"Sum: {sum}");
        Console.WriteLine($"Difference: {difference}");

        Console.ReadLine();
    }
}
```

Figure 6: Operator overloading

- Figure-7 shows Virtual/Method Overriding The code defines a base class Shape and two derived classes Circle and Square. The Shape class includes a virtual method Draw(), which is designed to be overridden by any class that inherits from Shape. The Draw() method in the Shape class simply writes the string "Drawing a shape" to the console
- The Circle and Square classes both inherit from Shape and override the Draw() method to provide their own specific implementations. The override keyword is used to indicate that these methods are intended to replace the Draw() method in the Shape class. In the Circle class, the Draw() method writes "Drawing a circle" to the console, and in the Square class, it writes "Drawing a square"
- In the Main() method of the Program class, instances of Shape, Circle, and Square are created. The Draw() method is then called on each of these instances. Due to the power of polymorphism, the specific version of the Draw() method that gets executed depends on the actual type of the object. So, for the Shape instance, the Draw() method of the

Shape class is called. For the Circle and Square instances, the Draw() methods of the Circle and Square classes are called, respectively. This results in the console output "Drawing a shape", "Drawing a circle", and "Drawing a square"

main.cs

```
1   using System;
2   class Shape
3   {
4       public virtual void Draw()
5       {
6           Console.WriteLine("Drawing a shape");
7       }
8   }
9
10  class Circle : Shape
11  {
12      public override void Draw()
13
14      {
15          Console.WriteLine("Drawing a circle");
16      }
17  }
18
19  class Square : Shape
20  {
21      public override void Draw()
22      {
23          Console.WriteLine("Drawing a square");
24      }
25  }
26
27  class Program
28  {
29      static void Main()
30      {
31          Shape shape = new Shape();
32          Circle circle = new Circle();
33          Square square = new Square();
34
35          shape.Draw();
36          circle.Draw();
37          square.Draw();
38
39          Console.ReadLine();
40      }
41  }
42
```

Figure 7: Virtual/Method Overriding

Case Study and Analysis

- C# is Language used for creation build many appication but most popular use of C# is its Use in Game Engine called Unity

- Unity is a versatile cross-platform game development engine that facilitates the creation of interactive 2D and 3D experiences. It provides a robust scripting API using C# for game logic.
- Unity's visual editor streamlines the design process, offering tools for scene creation, asset management, and debugging.
- With support for various platforms, Unity enables developers to deploy games to PC, Mac, Linux, mobile devices, and consoles. Its asset store and a large community contribute to a rich ecosystem of pre-built assets and resources for developers.

C# is extensively used in Unity for scripting and programming game logic.
Here are key uses cases of C# in Unity

1. Game Logic Scripting
    - Developers use C# to define and manipulate the behavior or game objects characters, and overall game functionality.
2. Component-Based Architecture
    - Unity's component-based architecture allows developers to create custom components using C#. These scripts are attached to game objects, providing them with specific functionalities.
3. Event Handling and Callbacks
    - C# is used to handle events and callbacks in Unity. For example, functions like *Start()* and *Update()* are automatically called at specific points, allowing developers to respond to events during the game's lifecycle
4. User Interface (UI) Development
    - C# also used for creating and managing user interfaces in Unity. Developers can use C# scripts to control UI elements, handle user input, and create menus
5. Physics and Animation
    - C# is used to control physics and animations within Unity. Developers use scripts to define how objects interact with each other through Unity's physics engine like collection , particle effects , force etc and how animations are triggered and controlled.
6. Co routines for Asynchronous Tasks:
    - C# coroutines are used in unity for asynchronous programming. This allows developers to perform actions over time without blocking the main thread, facilitating tasks such as animations, delays, and timed events.

Implantation of Object-Oriented Programming in Unity via C#
Object-oriented programming (OOP) is a fundamental paradigm in Unity development using C#. Here are key ways Object-Oriented Programming is utilized in Unity

- Game Objects
  Unity's entities called as game objects, and Object-Oriented Programming concepts like classes and inheritance are used to create these objects. Each game object can have components, which are often implemented using Object-Oriented Programming principles.

- Inheritance for Specialization
  Inheritance is employed to create specialized game objects with shared characteristics For example you can create a base class for a type of enemies and it can be inherited to create multiple Unique enemies which can have some common attack pattern

- Polymorphism for Flexibility
  Polymorphism allows the same method name to be used in different classes, providing flexibility.In Unity, this is often seen in methods like custom event handlers, where different behaviors can be implemented in various classes. for example their may be a Weapon class and it may have method damage which define who output damage is calculated but different types of weapon have different damage so we override the method according to the weapon specification

- Abstraction for Complexity Management
  Abstraction is employed to manage complexity by hiding unnecessary details and exposing only what is essential. Unity scripts often abstract away low-level implementation details, allowing developers to focus on high-level game logic.For example Game Ui such as Health Bar ,Stamina Bar which are common in all Levels so we can abstract or template it and reuse it each Level so the developer does not need to re implement the logic every time

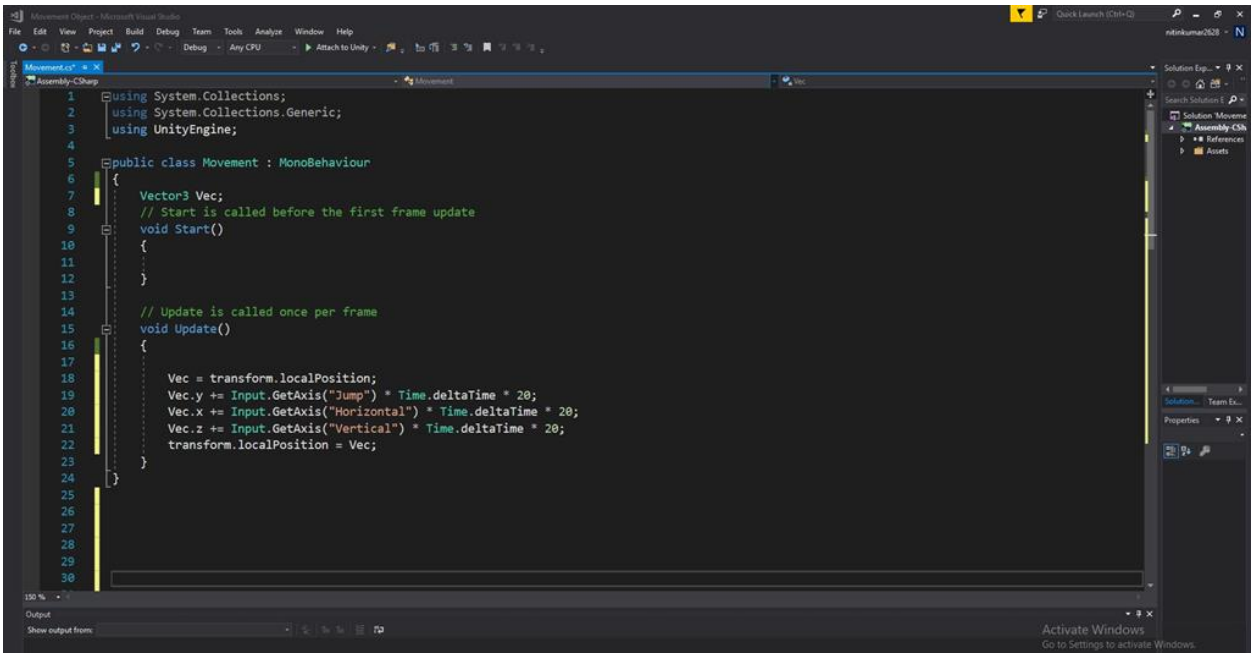

Figure 8: example Unity Script

Paradigm 2: Aspect-Oriented

Discuss the principles and concepts of Paradigm 2.

- Aspect oriented programming(AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules, also known as modularisation, where the aspect is the key unit of modularity
- In traditional programming, concerns such as logging, security, error handling, and other aspects are often intertwined with the core business logic of a program. As a result, modifying or extending one of these concerns can lead to changes in multiple places throughout the codebase.
- Aspect-oriented programming addresses this issue by providing a way to modularize cross-cutting concerns and encapsulate them in separate modules called aspects
- An aspect is a module that encapsulates a concern and defines how it should be woven into the main application.
- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore insted of repeating the code at every method define the functionality in a common class and control were to apply that functionality in the whole application. thats the basic idea of Aspect Oriented Programming
- Aspect-oriented programming introduces a set of concepts to achieve this
    1. **Join point** A point during the execution of a program, such as the execution of a method or the handling of an exception
    2. **Advice** Action taken by an aspect at a particular join point. Different types of advice include "around", "before", and "after" advice.
        - **Before advice**: Advice that runs before a join point but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
        - **After returning advice**: Advice to be run after a join point completes normally (for example, if a method returns without throwing an exception).
        - **After throwing advice**: Advice to be run if a method exits by throwing an exception.
        - **After (finally) advice**: Advice to be run regardless of the means by which a join point exits (normal or exceptional return).
        - **Around advice**: Advice that surrounds a join point such as a method invocation
    3. **Pointcut** A set of join points. Pointcuts define a criterion for selecting join points where advice should be applied.
    4. **Aspect** A module containing advice and pointcut definitions. Aspects encapsulate cross-cutting concerns.
    5. **Weaving** linking aspects with other application types or objects to create an advised object.

Language for Paradigm 2: Spring

Discuss the characteristics and features of the language associated with Paradigm 2.

- Spring is a comprehensive Java-based framework for building enterprise applications. It facilitates Inversion of Control (IoC) through dependency injection, promoting loose coupling.
- To create a Aspect in Spring we need to create a service which is cross cutting concern like logging and we need to create a *applicationContext.xml* as seen in Figure-10 file this file will have all your components defined in it
- to define a aspect we create a class with methods that are needed we use syntax such as *@Aspect* to define a class to aspect we use *@Before* to specify that bellow method should run before the target method and like wise *@After* to specify that bellow method should run after the target method
- In the example give in the Figure 9 we can see Implementation Aspect-Oriented Programming , code defines an interface MyService with a single method doSomething(). This method is implemented in the MyServiceImpl class
- The LoggingAspect class is an aspect that defines two advices, logBeforeMethod() and logAfterMethod(). These advices are methods that are executed before and after the doSomething() method of any class, respectively. This is specified by the @Before and @After annotations and the pointcut expressions "execution(public void doSomething())".
- In the main method of the Main class, a ClassPathXmlApplicationContext is created with the configuration file applicationContext.xml , it defines the beans for MyService and LoggingAspect and enable Spring Aspect-Oriented Programming auto-proxying.
- The MyService bean is then retrieved from the application context and its doSomething() method is called
- Because of the LoggingAspect, this will result in the logBeforeMethod() advice being executed before the doSomething() method, and the logAfterMethod() advice being executed after it.
- so what if one aspect has to applied on multiple methods in such case we use Pointcuts to generalize and specify for expample if all the Class names ends with Service and method name ends with Loger we modify aspect to be like
  *@Before("execution(void com.example.*Service.*Loger())")* this will match to all service which has class name ends with Service and method name ends with Loger

main.java

```java
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
interface MyService {
    void doSomething();
}

class MyServiceImpl implements MyService {
    @Override
    public void doSomething() {
        System.out.println("Doing something...");
    }
}
@Aspect
class LoggingAspect {

    @Before("execution(public void doSomething())")
    public void logBeforeMethod() {
        System.out.println("Logging before method execution...");
    }

    @After("execution(public void doSomething())")
    public void logAfterMethod() {
        System.out.println("Logging after method execution...");
    }
}
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        MyService myService = context.getBean(MyService.class);
        myService.doSomething();
    }
}

```

Figure 9: Simple Aspect Code in Spring

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- Enable Spring AOP -->
    <aop:aspectj-autoproxy/>

    <!-- Define beans (services and aspects) -->
    <bean id="myService" class="com.example.MyServiceImpl"/>
    <bean id="securityService" class="com.example.SecurityServiceImpl"/>
    <bean id="loggingAspect" class="com.example.LoggingAspect"/>

</beans>
```

Figure 10: applicationContext.xml

Now that we have explained the code we will now see how the concepts of Aspect-oriented programming is seen in Figure-9

- Join Point: A join point is a point in the execution of the program, such as the execution

of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution. In the provided code, the doSomething() method in the MyService interface is a join point.

- Advice: Advice is the action taken by an aspect at a particular join point. Different types of advice include "around," "before," and "after" advice. In the provided code, logBeforeMethod() and logAfterMethod() in the LoggingAspect class are advices. They are executed before and after the doSomething() method, respectively.
- Pointcut: A pointcut is a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. In the provided code, the pointcut is defined as execution(public void doSomething()), which matches the execution of the doSomething() method in any class.
- Aspect: An aspect is a modularization of a concern that cuts across multiple classes. It encapsulates behaviors that affect multiple classes into reusable modules. In the provided code, LoggingAspect is an aspect that encapsulates the cross-cutting concern of logging.
- Weaving: Weaving is the process of applying aspects to target objects to create new proxy objects. The weaving can be done at compile time (requires a special compiler), at load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime. In the provided code, the weaving is done when the MyService bean is retrieved from the application context and its doSomething() method is called.

Analysis

Provide an analysis of the strengths, weaknesses, and notable features of both paradigms and their associated languages.
strength and weakens of Object-Oriented Programming (OOP) with C#

- Object-Oriented Programming promotes modality through classes and objects, enabling code reuse through concepts like inheritance and polymorphism.which is very efficient and elegant over functional programming
- **Encapsulation** in Object-Oriented Programming allows us control over what is visible to public which is very usefull to have when you have to hide sensitive data but have to use it which out reveling it
- **Abstraction** can be used in C# to create template classs by using abstract classes and interfaces which can we implemented down the line
- C# is a **Strongly typed** language their for we catch and fix type related errors while compelling providing better code safety
- C# has its own disadvantages such as
- **Steep learning curve** C# can be a challenging language to learn, particularly for developers who are new to programming. With its advanced features and syntax, it can take time to become proficient in C#

- **Proprietary technology** C# and the .NET framework are owned and maintained by Microsoft, which means that developers may be limited in their ability to customize or modify the technology to suit their specific needs
- **Resource-intensive** C# applications can be resource-intensive, which means that they may require more memory and processing power than other applications. This can make C# less suitable for building applications that need to run on older or less powerful devices

now we will take a look at strength and weakens of Aspect-Oriented Programming (AOP) with Spring Framework

- **Cross-Cutting Concerns** Aspect-oriented programming excels in handling cross-cutting concerns like logging, security, and transaction management, allowing for a cleaner separation of concerns
- **Modularity** Aspect-oriented programming promotes modularity by allowing developers to encapsulate aspects (cross-cutting concerns) separately from the core business logic.
- **Improved Code Maintainability** Aspect-oriented programming can lead to more maintainable code by reducing code duplication and promoting a more modular architecture
- **Dynamic Aspect Weaving** Spring Aspect-oriented programming allows dynamic weaving of aspects at runtime, offering flexibility in applying aspects to code.
- now we will look at weakness of Aspect-oriented programming with Spring Framework
- **Seating up** Spring as their a lot of prep work to do such as writing xml downloading the spring jar and dependency library before you can even code the core logic
- **Learning Curve** Understanding and implementing Aspect-oriented programming concepts might be challenging for developers unfamiliar with the paradigm
- **Overuse of Aspect-oriented programming** Inappropriate use of Aspect-oriented programming for every aspect of a program can lead to code that is harder to understand and maintain.
- **Debugging Challenges** Debugging can be more challenging when aspects are applied, as the flow of control may be influenced by aspects at runtime.

Comparison

Compare and contrast the two paradigms and languages, highlighting similarities and differences.
Now we will Compare and contrast the two paradigms and languages and usecases
Similarities between Object Oriented and Aspect-Oriented Programming (AOP)

- Both Object-Oriented Programming and Aspect-oriented programming aim to enhance modularity in software design. Object-Oriented Programming achieves modularity through the organization of code into classes and objects, while Aspect-oriented programming separates cross-cutting concerns into aspects

- Both paradigms support the concept of encapsulation, where data and methods that operate on the data are bundled together. In Object-Oriented Programming, this is achieved through classes and objects.

Diffrense between Object Oriented and Aspect-Oriented Programming

- While everything in OOPs is classes and object even the cross cutting concern while in Aspect-oriented programming these cross cutting concerns are convertd to aspects which are called uppon in main class when needed

Why and When Do we need them ? Aspect-oriented programming or Object-Oriented Programming

- OOP is beneficial when you want to organize your code into classes and objects, promoting encapsulation, modularity, and code reuse
- but issues arrives when suppose ther is a method which is required in every class so ideally we can put this method in independent class and inherit for every class that use this method so finally if we draw a dependency map it would show that this class would be used the most which is not good , In such case we use Aspect-oriented programming
- In Aspect-oriented programming such dependency are defined as Aspect and only when program requires it
  ie during the weaving process
- Aspect-oriented programming is only used in cases where cross cutting concerns arises
- Issues that can form while using it are Overuse of Aspect-oriented programming can cause difficulty when trying to Debug as Aspect-oriented programming is Dependency injection when their is error or un expected output tracing the program flow to find the error will be challenging
- now lets comeback to the question what should we use between Aspect-oriented programming or Object-Oriented Programming to create most ideal flexibly program it is advised to use a blend of both Object-oriented programming for most of things and Aspect-oriented programming for crosscutting concerns

Why and When Do we need them ? Spring or C#

Spring

- Srping is Java Framwork as such if your project is heavily tided to Java ecosystem, Spring can be a natural choice. It's widely used in Java enterprise applications.
- Spring provides a robust and flexible dependency injection container. dependency injection simplifies the management of components and promotes loose coupling, making your code more modular and testable.
- Spring has excellent support for Aspect-oriented programming, allowing you to separate cross-cutting concerns
- Spring integrates well with other Java technologies and frameworks . It supports integration with databases, messaging systems, and other enterprise technologies.

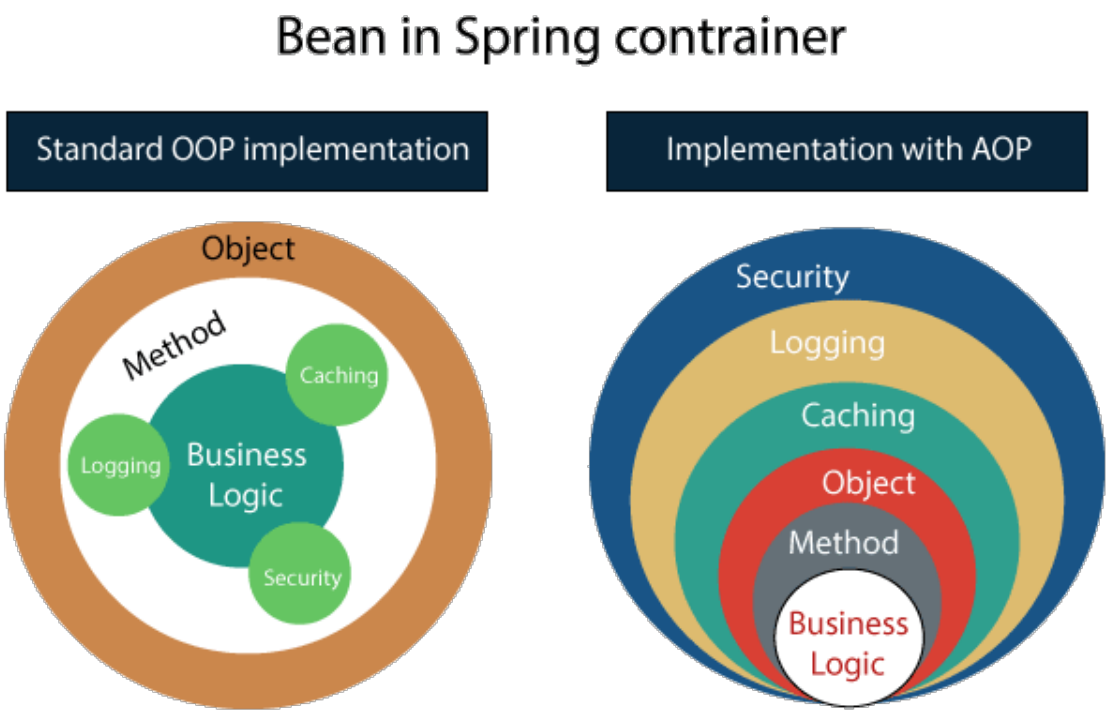

Figure 11: OOP vs AOP

- Spring is widely used in building enterprise-level applications, especially those requiring salable and maintainable architecture.

C#

- If your organization is focused on the Microsoft ecosystem or you are developing applications for Windows, C# with the .NET framework or .NET Core is a natural choice
- C# is a modern and feature-rich language with support for object-oriented, functional, and asynchronous programming paradigms
- With .NET Core, C# supports cross-platform development, allowing you to build applications that can run on Windows, Linux, and macOS
- As C# is owned by Microsoft you would need to wait for any updates from Microsoft for any core security patches

Challenges Faced

Discuss any challenges you encountered during the exploration of programming paradigms and how you addressed them.

- Aspect-oriented programming had a sharp learning curve to understand the core mechanics . so did C# but C# Object-oriented programming was much similar to C++
- Setting up Spring and C# was difficult

- How i over came these challenges where reading documenting searching for Answers to quarries on StackOverFlow and YouTube

Conclusion

- Object-oriented programming is a programming paradigm that structures code around objects, encapsulating data and behavior.
- It promotes reusability, modularity, and abstraction, enhancing code organization and maintenance.
- Core concepts of Object-oriented programming are :-
  - Encapsulation
  - Inheritance
  - Abstraction
  - Polymorphism
- Aspect-oriented programming (AOP) is a programming paradigm that allows modularizing cross-cutting concerns, such as logging or security, separately from the main application logic
- It enhances code separation, maintainability, and promotes a cleaner design by isolating aspects
- Core concepts of Aspect-oriented programming are :-
  - Join point
  - Advice
  - Pointcut
  - Aspect
  - Weaving
- C# is a powerful, object-oriented programming language designed by Microsoft, emphasizing strong support for OOP principles such as encapsulation, inheritance, and polymorphism.
- Spring is a comprehensive Java-based framework that promotes modularity, scalability, and simplifies enterprise application development
- With built-in support for Aspect-Oriented Programming (AOP), it facilitates the separation of cross-cutting concerns, enhancing code organization and maintainability by using Aspects

References

www.codecademy.com/resources/docs/general/programming-paradigms/object-oriented-programm
https://learn.microsoft.com/en-us/dotnet/csharp/
https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/
https://www.c-sharpcorner.com/UploadFile/4624e9/abstraction-in-C-Sharp/
https://unity.com/how-to#c-programming-unity
https://www.linkedin.com/pulse/exploring-pros-cons-c-comprehensive-guide-baraa-mayasa/
https://docs.spring.io/spring-framework/reference/core/aop.html
https://chat.openai.com