AMRITA VISHWA VIDYAPEETHAM

Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

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

Nitin G R

21st January, 2024

Paradigm 1: Object Oriented Programming

A Brief History

The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.

- The first object–oriented language was Simula (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
- In 1970, Alan Kay and his research group at Xerox PARK created a personal computer named Dynabook and the first pure object-oriented programming language (OOPL) - Smalltalk, for programming the Dynabook.
- In the 1980s, Grady Booch published a paper titled Object-Oriented Design that mainly presented a design for the programming language, Ada. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
- In the 1990s, Coad incorporated behavioral ideas into object-oriented methods.

What is object-oriented programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software.

The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability, and efficiency.

The first step in OOP is to collect all of the objects a programmer wants to manipulate and identify how they relate to each other – an exercise known as data modeling.Examples of an object can range from physical entities, such as a human being who is described by properties like name and address, to small computer programs, such as widgets.

Once an object is known, it is labeled with a class of objects that defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. Objects can communicate with well-defined interfaces called messages.

What is the building blocks of object-oriented programming?

The structure, or building blocks, of object-oriented programming include the following:

1. Classes

Classes offer templates to better characterize objects. In effect, classes serve as blueprints for generating objects. Within a class, programmers must define the variables and methods that its corresponding objects can reference. Given the car example, a class would denote the properties of the car object, encase the car's functionality, as well as declare the car as a class in the first place.

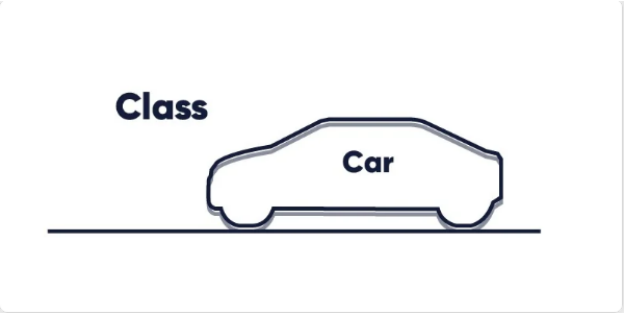

Figure 1: class

2. Attributes

Attributes (or variables) refer to the characteristics of the object. Appearance, state, and other qualitative traits are common object attributes. Class attributes in combination with object instances differentiate objects from one another. The following program demonstrates a class declaration in Python:

```
class Car:
    def __init__(self, color, type):
        self.color = color
        self.type = type
```

Here, 'self' represents an instance of the class for future referencing of the object's attributes and methods. And 'color' and 'type' represent attributes of the class.

3. Methods

Programmers also must define methods alongside attributes. Methods encapsulate functions that handle the data and behavior of an object instance. In a car, a drive method might be appropriate. You can define such a method right below the car's attribute definitions. Though it is possible via code to render an actual car and simulate a driving application, programming this method is a bit more complex than the lines of code below.

```
def drive(self):
    print('I'm driving a ' + self.color + ' ' + self.type)
```

4. Objects

Objects exist alongside classes. They are essentially data fields with distinct structures that the programmer can determine. Once you call an object, the program creates an instance. An instance is a specific object

generated from a class. To call an object, you will need to provide information relevant to the class, such as the particular color and type of the car.

```
automobile = Car('red', 'Sedan')
```

The code above will formally establish a concrete instance of the unique automobile object. Then, you can glimpse how the drive method works in action.

```
automobile.drive()
```

Altogether, by implementing all the concepts you just learned, you will have the following Python program.

```
class Car:
    def __init__(self, color, type):
        self.color = color
        self.type = type

    def drive(self):
        print('I'm driving a ' + self.color + ' ' + self.type)

automobile = Car('red', 'Sedan')
automobile.drive()
```

As a result, the output on your computer screen will read, "I'm driving a red Sedan."

Figure 2: object

Principles of OOP

There are four main principles of OOP:

- Encapsulation: A desired outcome of organizing code in classes to keep things from being mixed with other unrelated bits of code. Encapsulation makes it easier to reason about code because of the modularity of code written in object-oriented styled classes.

  In the following Python example, a Robot class is created with a version property, initialized as a number. Getter and setter methods are made to allow any instance of Robot to be set with a version and have it changed later:

  ```
  class Robot(object):
   def __init__(self):
     self.__version = 22

   def getVersion(self):
     print(self.__version)
  ```

```
 def setVersion(self, version):
   self.__version = version

obj = Robot()
obj.getVersion()
obj.setVersion(23)
obj.getVersion()

print(obj.__version)
```

- Inheritance: A principle which allows an instance of an object to borrow attributes and methods from its parent class.

```
// Parent Class
class Machine {
  // Machine class member
}

// Child Class
class Car extends Machine {
  // Car inherits traits from Machine

  // Additional Car class members, functions, etc.
}
```

- Polymorphism: Polymorphism means designing objects to share behaviors. Using inheritance, objects can override shared parent behaviors with specific child behaviors. Polymorphism allows the same method to execute different behaviors in two ways: method overriding and method overloading.

  Runtime polymorphism uses method overriding. In method overriding, a child class can implement differently than its parent class. In our dog example, we may want to give TrackingDog a specific type of bark different than the generic dog class.

```
//Parent class Dog
class Dog{
    //Declare protected (private) fields
    _attendance = 0;

    constructor(namee, birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    getAge() {
        //Getter
        return this.calcAge();
    }

    calcAge() {
        //calculate age using today's date and birthday
        return this.calcAge();
    }

    bark() {
        return console.log("Woof!");
    }
```

```
    updateAttendance() {
        //add a day to the dog's attendance days at the petsitters
        this._attendance++;
    }
}

//Child class TrackingDog, inherits from parent
class TrackingDog extends Dog {
    constructor(name, birthday)
        super(name);
        super(birthday);
    }

    track() {
        //additional method for TrackingDog child class
        return console.log("Searching...")
    }

    bark() {
        return console.log("Found it!");
    }


//instantiate a new TrackingDog object
const duke = new TrackingDog("Duke", "1/12/2019");
duke.bark(); //returns "Found it!"
```

Compile Time polymorphism uses method overloading. Methods or functions may have the same name but a different number of parameters passed into the method call. Different results may occur depending on the number of parameters passed in.

```
//Parent class Dog
class Dog{
    //Declare protected (private) fields
    _attendance = 0;

    constructor(namee, birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    getAge() {
        //Getter
        return this.calcAge();
    }

    calcAge() {
        //calculate age using today's date and birthday
        return this.calcAge();
    }

    bark() {
        return console.log("Woof!");
    }
```

```
    updateAttendance() {
        //add a day to the dog's attendance days at the petsitters
        this._attendance++;
    }

    updateAttendance(x) {
        //adds multiple to the dog's attendance days at the petsitters
        this._attendance = this._attendance + x;
    }
}

//instantiate a new instance of Dog class, an individual dog named Rufus
const rufus = new Dog("Rufus", "2/1/2017");
rufus.updateAttendance(); //attendance = 1
rufus.updateAttendance(4); // attendance = 5
```

- Abstraction: A principle that highlights the benefit of hiding complex parts of code from other parts to make it easier to reason and make decisions about the code.

Features of Object-Oriented Programming

You can gain a better understanding of object-oriented programming by understanding its four fundamental features.

- Easily Upgradable: Programmers can use object-oriented programming software independently and can also easily upgrade OOP packages in the existing software. So, OOP can make software development more modular, reusable, and maintainable, which can make it easier to upgrade and update the system.
- Flexible: OOP software is flexible to use for programmers. hence, they can pass different objects through the same interface using polymorphism, which implies the ability of a single function or method to operate on multiple data types.
- Smooth Interface: The user interface of OOP software is smooth and easy to handle. It can provide a consistent interface for users by encapsulating the implementation details and exposing an efficient set of methods and properties.
- Modularity: The encapsulation method in OOP software helps objects to be self-contained. Also, it facilitates troubleshooting while developing a program. Moreover, the software enables modularity by allowing developers to break down a complex problem into self-contained units to make specific designs.

Advantages of Object-Oriented Programming

Developers utilize object-oriented programming to build software based on data. Here is a breakdown of its benefits.

1. Enables Code Reusability It is no longer necessary for programmers to manually develop the same code multiple times because they can reuse code through OOP inheritance.

2. Increases Productivity in Software Development The OOP framework allows programmers to construct programs taking help from existing packages, such as Python, which can save time and boost productivity

3. Reinforces Security Programmers filter out limited data using OOP mechanisms, such as data hiding and abstraction, to keep the application secure. This ensures that only important data can be seen by the users.

4. Simplifies Code Maintenance The code in object-oriented software is easy to use and maintain. The software simplifies the code maintenance process by promoting modular and reusable designs.

Language for Paradigm 1: Ruby

What is Ruby?

Ruby is an open-source object-oriented scripting language invented in the mid-90s by Yukihiro Matsumoto.

Unlike languages such as C and C++, a scripting language doesn't talk directly to hardware. It's written to a text file and then parsed by an interpreter and turned into code. These programs are generally procedural in nature, meaning they are read from top to bottom.

Object-oriented languages, on the other hand, break out pieces of code into objects that can be created and used as needed. You can reuse these objects in other parts of the program, or even other applications.

Yukihiro wanted to create a scripting language that leveraged object-oriented programming and increase code reuse to help speed up development. And so the Ruby programming language was born, using simple language and syntax to handle data and logic to solve problems.

Why should I learn Ruby?

The Ruby programming language is designed for programmer productivity and fun. Developers like using Ruby because it's high level and has a simple syntax. You have less code to write and can focus on finding a solution to your problem.

Because of the high level and abstracted nature of Ruby, this adds up to a language that is easy to learn and put into practice. While many low-level languages require lines and lines of code for the smallest thing, with Ruby, you can write your first cloud application in just a few hours.

What can Ruby be used for?

The Ruby programming language is a highly portable general-purpose language that serves many purposes. Ruby is great for building desktop applications, static websites, data processing services, and even automation tools. It's used for web servers, DevOps, and web scraping and crawling. And when you add in the functionality of the Rails application framework, you can do even more, especially database-driven web applications.

Why Use Object-Oriented Programming in Ruby?

Object-oriented programming (OOP) is advantageous in Ruby because it offers a systematic method for creating software. It enables programmers to arrange their code in a logical and modular fashion, simplifying understanding, maintainability, and reusability.

Take a banking application as an example to demonstrate this. We can design a class named `BankAccount` that acts as a blueprint for displaying a customer's bank account. The `account_number`, `balance`, and `customer_name` are a few possible properties for this class. Additionally, we may define class-level methods like `deposit`, `withdraw`, and `check_balance`.

For each of the clients, we can build a `BankAccount` class object. Each of the accounts' properties will be kept in these objects. Using the class's methods, we can additionally carry out actions like withdrawals, deposits, and balance checks.

Classes and Objects in Ruby

In Ruby, classes provide a reusable framework using which objects can be instantiated, which are also of the same type. It is a user-defined block containing the features of the object in consideration and related functions for performing related operations. Let's take a deeper look into objects and classes, which form the basic backbone of OOP in Ruby.

What are Objects?

Objects are instances of a class in object-oriented programming. They communicate with one another through method calls and contain both the data and the behavior specified by the class. Each object has a distinct state (data), as well as behavior (methods).

What is a Class in Ruby?

Ruby uses classes much like a blueprint for creating objects of the same type. The characteristics and abilities that objects in that class will possess are specified. A class is created and defined in Ruby using the class keyword, followed by the body of the class which contains the data members and member functions of the class.

Creating an Object from a Class

The Ruby new keyword is used to create an object belonging to a particular class. By doing so, memory is allotted for the object and a fresh instance of the class is initialized. For example, if we have a class called Shape, we can make an object of that class by using Shape.new. This creates a new object based on the class definition that is prepared to be used in our program.

```
class Shape
end

shape = Shape.new
```

Defining Classes in Ruby

How to Define a Class in Ruby?

In Ruby, the class keyword must be included before the class name to declare a class. A class definition allows for the specification of characteristics and methods of the class's objects. Methods define an object's behavior, whereas attributes show an object's current state. Consider the following example:

```
class Person
    def initialize(name)
        @name = name
    end

    def greet
        puts "Hello {@name}"
    end
end
```

Creating an Instance of a Class

We create an instance of a class using the new method. The new method calls the class's constructor, which is equivalent to Ruby's initialize method. The constructor is required to initialize the object's state.

```
person = Person.new("Rohit")
person.greet # Output: Hello Rohit
```

In the example above, the person's name is passed as a parameter to the initialize method, which then assigns it to the instance variable @name. The name is then printed by the greet method.

Instance Variables and Methods

Object instance variables are denoted in Ruby by the @ symbol. These variables can be accessed and modified by class-specific methods, and they serve as a representation of the object's current state. In the

above illustration, greet is a method of the class Person, and @name is an instance variable.

Constructor in Ruby

Calling the constructor creates the object's initial state in Ruby and is used to create object instances. This constructor's name in Ruby is initialize. The @name variable of the class object is set in the above example's initialize method.

Attribute Accessors

The Ruby programming language provides an advantageous feature known as attribute accessors for interacting with instance variables. Attribute accessors can be used to read and modify the values of the instance variables of a class. Attribute accessors come in three basic categories, each of which fulfills a particular need.

- `attr_reader`:

  First, there is `attr_reader`, which is used in the development of read-only accessors. While we cannot directly alter an instance variable's value using `attr_reader`, we can obtain it. When we wish to expose a variable's value to other areas of our program without permitting external modifications, this kind of accessor comes in handy.

- `attr_writer`:

  Access to instance variables using this accessor is restricted to writing. With `attr_writer`, an instance variable's value can be modified, but it cannot be directly retrieved. This form of accessor is helpful when we want to restrict direct access to a variable's value while still allowing external updates.

- `attr_accessor`:

  The `attr_reader` and `attr_writer` functionalities are combined to jointly form the `attr_accessor`. Our access to an instance variable using this accessor is for reading as well as writing so that we can retrieve its value and make the appropriate adjustments. The `attr_accessor` makes it simple to get and modify an instance variable's value, making it particularly useful when we need total control over the variable.

Let's see an example:

```
class Person

  attr_reader :name
  attr_writer :age
  attr_accessor :email

  def initialize(name, age, email)
    @name = name
    @age = age
    @email = email
  end

end
```

In this example, we define a `Person` class with three attribute accessors: `attr_reader :name`, `attr_writer :age`, and `attr_accessor :email`. The `attr_reader` creates a read-only accessor for the `name` attribute, allowing us to retrieve the person's name but not modify it externally. The `attr_writer` creates a write-only accessor for the `age` attribute, enabling external updates to the person's age but preventing direct retrieval. Lastly, the `attr_accessor` creates both read and write accessors for the `email` attribute, allowing us to both

retrieve and modify the person's email address as needed. The `initialize` method sets the initial values for the attributes when a new `Person` object is created.

OOP's concept in Ruby

Abstraction, polymorphism, inheritance and encapsulation form four of the main pillars of OOP. The acronym "A PIE" is a helpful mnemonic to remember them. Let's explore how they apply to Ruby:

Abstraction

Ruby exhibits Abstraction by allowing us to form mental models of problems using familiar 'real-world' concepts, which allows us to abstract the problem to a more familiar domain. It puts the emphasis of the programming language on human needs over machine needs. We concern ourselves with a higher-level sense of the problem without worrying about implementation details. This allows us to restrict our focus to objects with properties (states) and behaviours (methods).

Encapsulation

Speaking about restricting our focus, Ruby does something similar in the form of encapsulation. Just like how we don't need to worry about the meaning of life while we're engaged in our work as programmers, Ruby restricts its focus to only what is relevant to the task at hand. It's perfectly valid to think about the meaning of life, but it's probably not relevant most of the time that you're programming. Encapsulation is the deliberate erection of boundaries in code that prevents erroneous accessing and modifying of states and behaviours that don't make sense for what our intention is. If for example, we create a CityPark class to form a blueprint of properties and behaviours that we expect out of a CityPark object, we expect that there may be a relation in terms of behaviours with a Forest class, but we don't want their particular attributes to overlap.

We would expect both a Park object and a Forest object to contain tree attributes, but the specifics should be unique to the particular object. In other words, that information should be encapsulated within that specific instance of the class. If we try to retrieve the information regarding the number of trees in a specific park, we don't want to return the value of the number of trees in some forest. The state of each object is said to be unique, because it's bound to a specific object. We can therefore create multiple CityPark objects and expect to have the ability to retrieve information about each specific park individually.

```
#city_park.rb

class CityPark
  attr_reader :name, :num_trees

  def initialize(name, num_trees)
    @name = name
    @num_trees = num_trees
  end
end

class Forest
  attr_reader :name, :num_trees

  def initialize(name, num_trees)
    @name = name
    @num_trees = num_trees
  end
end
```

```
high_park = CityPark.new("High Park", 5000)
durham_forest = Forest.new("Durham Forest", 125000)
dufferin_park = CityPark.new("Dufferin Park", 2000)

high_park.name # => "High Park"
high_park.num_trees # => 5000
durham_forest.name # => "Durham Forest"
durham_forest.num_trees # => 125000
dufferin_park.num_trees # => 2000
```

Notice how on line 25 we chain the `name` method to the `high_park` local variable, which is where the specific instance of the `CityPark` class is stored. This works because the `name` method is actually an instance method of the `CityPark` class (`CityPark#name`). The `name` method is a 'getter' method which retrieves the data stored in an instance variable. Instance variables are where we store the specific unique attributes (or states) for instances of the class.

Polymorphism

Also notice how we used what looks like the same `name` method on objects of two different classes — and Ruby didn't get confused!? This is a demonstration of polymorphism, which is being able to have a single interface perform different functionality depending on the context in which it's invoked. Because of Ruby's method lookup path, (and encapsulation) the Ruby interpreter first looks for the method that's being invoked in the class of the calling object (here on line 25 it would be the `CityPark` class). Because a `name` method is found within the `CityPark` class, it stops looking and invokes this method. Note that in our example, the two `name` methods are fully independent: `CityPark#name` and `Forest#name`. If we wished, we could modify one of those methods for different functionality without affecting the other.

This flexibility allows us to re-write methods of the same name within custom classes to make their implementation details specific to the needs we want, while exposing a familiar public interface with which we interact with.

But hold up. So far in our example, both the `CityPark` and `Forest` class share exactly the same behaviors and attributes (objects of each class have `name` and `num_trees` attributes, and an instance method to retrieve that information). Doesn't it seem inefficient to have written repetitive code? Anyone say DRY? As in: "Don't Repeat Yourself", not: "that insult was harsh and accurate".

Inheritance

Remember that other pillar of OOP? Inheritance is the ability of related classes to share behaviours through a hierarchical structure of single inheritance. Subclasses inherit the methods from their parent classes (which includes the methods that it inherits through its parent class and so on so forth up the chain). It's called single inheritance because a given class can only ever directly subclass from one parent class. In our example, we can re-imagine our class relationships by creating a superclass that we'll call GreenSpace. By subclassing the CityPark and Forest classes from this shared ancestor GreenSpace, both of those subclasses will inherit the behaviours and attribute domains of the GreenSpace class (the $<$ symbol on lines 12 and 14 denote a subclass relationship). So many characters saved, what relief! Seriously though, efficiency is wonderful.

```
# greenspace.rb

class GreenSpace
  attr_reader :name, :num_trees

  def initialize(name, num_trees)
    @name = name
    @num_trees = num_trees
  end
```

```
end

class CityPark < GreenSpace; end

class Forest < GreenSpace; end

high_park = CityPark.new("High Park", 5000)
durham_forest = Forest.new("Durham Forest", 125000)
dufferin_park = CityPark.new("Dufferin Park", 2000)

high_park.name # => "High Park"
high_park.num_trees # => 5000
durham_forest.name # => "Durham Forest"
durham_forest.num_trees # => 125000
dufferin_park.num_trees # => 2000
```

Now, when we create a new CityPark object on line 16, there is no initialize constructor method within the CityPark class, so the Ruby interpreter will go through the method lookup path in search of an initialize method (as a constructor method, the initialize method is called automatically upon instantiation of a new object of the class). The exact method lookup path for a particular calling object can be found by invoking the ancestors class method on the class of the calling object (it will return an array, which is the names of the classes and modules that will be searched (in order) for the method being invoked). Ruby will stop looking and invoke the first method that it finds by the name provided.

Class methods are methods that are invoked on class itself, rather than on an object of the class. They do not have scope to the individual attributes of objects of the class, they're focused on functionality that is more general to the class (such as finding out the method lookup path for any object within the class, which doesn't concern itself with any specific instance).

Modules

Modules are Ruby's solution to multiple inheritance. Modules can be containers of methods that are out of place elsewhere in your code (applicable in some places but not others), or it can contain multiple whole classes to group classes that are similar but not hierarchically related. Any number of modules can be mixed into a class (remember that a class can only directly inherit from one other class) by using the include reserved word followed by the module name. This is called a mixin. Note that objects cannot be instantiated from modules; that is restricted to classes.

To demonstrate when to use a module vs. inheritance it can be useful to think about the relationship. If it's an "is-a" relationship such as "A City Park is a Green Space", then it's likely more sensible to inherit behaviours through hierarchy from a GreenSpace parent class. If it's more of a "can-do" relationship such as "You can go swimming in a CityPark", then that behaviour might better suit being mixed in through a module*. The convention with module naming is to append the "-able" suffix to the module name, such as with Swimmable.

If we add some more classes to our example, a RecreationCentre class that subclasses from CityPark, and a Lake class that subclasses from Forest, we see how modules are useful. Despite all of the classes sharing the methods from the GreenSpace class higher up in the hierarchy, we don't want to put the swim method there because that would include the behaviour in places it doesn't belong such as our CityPark class. Instead of creating a mess by including the swim behaviour in a shared parent class (or being gross and repeating ourselves), we include the Swimmable module to add the swim method functionality only where it's needed: within the Lake and RecreationCentre classes.

```
# recreation_centre.rb

module Swimmable
  def swim
```

```
    "Can swim here!"
  end
end

class GreenSpace
  attr_reader :name, :num_trees

  def initialize(name, num_trees)
    @name = name
    @num_trees = num_trees
  end
end

class CityPark < GreenSpace; end

class RecreationCentre < CityPark
  attr_reader :philanthropist
  include Swimmable

  def initialize(name, num_trees, philanthropist)
    super(name, num_trees)
    @philanthropist = philanthropist
  end
end

class Forest < GreenSpace; end

class Lake < Forest
  include Swimmable
end

dufferin_park = CityPark.new("Dufferin Park", 2000)
wallace_emerson = RecreationCentre.new("Wallace Emerson", 2, "Joe Beef")

dufferin_park.num_trees # => 2000
wallace_emerson.num_trees # => 2
wallace_emerson.swim # => "Can swim here!""
dufferin_park.swim # NoMethodError => undefined method 'swim' for CityPark...
wallace_emerson.philanthropist # => "Joe Beef"
```

Setter and Getter Methods

Let's investigate our new code further. Notice the `attr_reader` on line 10. This is the line of code that creates that getter method that we mentioned earlier. It's part of the `attr` family, which creates attribute setter and getter methods for the arguments passed in as symbols (here `:name`, and `:num_trees`). `attr_reader` is the short-form way to create a reader method, `attr_writer` will create a writer method, and `attr_accessor` will create both. It's convention to pass the instance variable as a symbol rather than to use some other more descriptive name like `get_name` or `show_num_trees`. This makes the invocation of the method clean and straightforward. It's also possible to define setter and getter methods using the `def` reserved word like with traditional instance methods if you want to add functionality (like formatting a name in a setter method).

Super

Did you notice the word `super` on line 25? The reserved word `super` passes any arguments supplied up the method lookup path to the first method of the same name that Ruby finds, which it then invokes. In our example, `super` on line 25 passes the `name` and `num_trees` arguments up to the `initialize` method in the `GreenSpace` class. Ruby then continues to execute the originally invoked `initialize` method on line 24 and assigns the `philanthropist` argument to the `@philanthropist` instance variable, where it is stored as a unique state of the instance of the `RecreationCentre` class (see the return value of line 43). `Super` can also be invoked without arguments to pass all supplied arguments up the lookup path.

Self

Why don't we talk about some more Ruby OOP basics, such as the reserved keyword self. Self is interesting, because its meaning is dependant on context. Self refers to the calling object, which for us can mean a specific instance of the class, or the class itself, depending on where it's used.

```
# recreation_centre_expanded.rb

module Swimmable
  def swim
    "Can swim here!"
  end
end

class GreenSpace
  attr_accessor :name, :num_trees

  def initialize(name, num_trees)
    @name = name
    @num_trees = num_trees
  end
end

class CityPark < GreenSpace; end

class RecreationCentre < CityPark
  attr_reader :philanthropist

  include Swimmable

  @@num_rec_centres = 0

  def initialize(name, num_trees, philanthropist)
    super(name, num_trees)
    @philanthropist = philanthropist
    @@num_rec_centres += 1
  end

  def whats_this
    self
  end

  def self.num_rec_centres
    @@num_rec_centres
  end
```

```
end

class Forest < GreenSpace; end

class Lake < Forest
  include Swimmable
end

dufferin_park = CityPark.new("Dufferin Park", 2000)
wallace_emerson = RecreationCentre.new("Wallace Emerson", 2, "Joe Beef")
scadding_court = RecreationCentre.new("DunBat", 25, "Jim Balsillie")


RecreationCentre.num_rec_centres # => 2
scadding_court.name # => "DunBat"
scadding_court.name = "Scadding Court"
scadding_court.name # => "Scadding Court"
scadding_court.whats_this # => #<RecreationCentre:0x00007fc73b9a0060 @name="Scadding Court", @num_trees=
```

In our latest example, on line 37 inside the `RecreationCentre` class we define a class method. Remember those? We know we're defining a class method because the context is inside a class, but outside of an instance method definition. The `self` here refers to the class itself, which makes it clear why we use it to define a class method.

On lines 33–35 we defined a new instance method called `whats_this` which only invokes `self`. Because the context here is within an instance method, `self` now refers to the specific instance of the class that the method is invoked on, which we do on line 57. Does any of that return value look familiar to you?

Inside some encoding is `RecreationCentre` which a class we've defined, then an octal number followed by `@name=''Scadding Court''`, which is an attribute we've assigned, followed by `@num_trees=25`, and then `@philanthropist=''Jim Balsillie''`. Interesting! This is all the information about the object we invoked the `CityPark#whats_this` instance method on. This is our calling object.

Did you notice another change in the code on line 10? The `attr_reader` has been changed to an `attr_accessor`, which now creates not just the getter, but both the setter and getter methods. We demonstrate this change in functionality on lines 54–56. Because these setter and getter methods are public (by default), we can invoke them outside of the class definition.

Features of the programming language Ruby

- Interpreted: That is, Ruby interpreter needs to parse the code and translate it into machine language understandable by a computer, but there is no previous compilation process like in C or Java.
- Dynamic and flexible: It can be used to alter code at runtime. This allows for greater flexibility in code writing and the creation of more expressive programs.
- Readable and expressive: Ruby takes pride in having a clean and readable syntax that resembles natural language. This facilitates code understanding and maintenance, fostering rapid development.
- Open source and cross-platform: It can be freely downloaded from the official website and run on different operating systems.
- Object-oriented: Everything in Ruby is an object, even basic data types like strings, numbers, or even boolean values. This allows for the creation of classes, inheritance, polymorphism, and encapsulation, making it easier to build modular and reusable programs.
- Backed by a large community: Ruby has an active community of developers and a wide range of libraries and frameworks available. For instance, Ruby on Rails is a highly popular web framework built on Ruby, used for developing robust and scalable web applications.

Paradigm 2: Aspect-Oriented

What is Aspect-Oriented Programming?

Aspect-oriented programming (AOP) is a programming paradigm that aims to improve the modularity of software by allowing the separation of cross-cutting concerns. In other words, AOP helps to decompose complex problems into smaller, more manageable pieces by identifying and addressing common concerns that cut across the entire application.

Aspect Oriented Programming AOP is based on the concept of "aspects," which are modular units that represent a particular concern or feature of the application. These aspects can be woven into the application code at specific points, known as "join points," to add new behavior or modify existing behavior.

One of the main benefits of AOP is that it allows developers to write code that is more modular and easier to maintain. Because concerns are separated and modularized, developers can focus on specific concerns without being distracted by unrelated code. This can make it easier to understand and modify the application.

AOP also enables developers to add new features or behavior to an application without having to modify the existing code. This can be particularly useful when adding features that are not directly related to the main functionality of the application.

To implement AOP, developers typically use an AOP framework, which provides the necessary tools and infrastructure to define and apply aspects to an application. Some popular AOP frameworks include AspectJ, Spring AOP, and AspectWerkz.

There are several common types of concerns that are often addressed using AOP, including logging, security, and performance. For example, a logging aspect might be used to add logging statements to an application to track the flow of execution or to record errors. A security aspect might be used to add authentication and authorization checks to an application. And a performance aspect might be used to optimize the performance of an application by identifying and addressing bottlenecks or other inefficiencies.

How AOP can help you?

Let's imagine you want to log a message inside methods of your domain model:

Example: Logging without AOP:

```
namespace Examples\Forum\Domain\Model;
use Examples\Forum\Logger\ApplicationLoggerInterface;

class Forum {

        /**
         * @Flow\Inject
         * @var ApplicationLoggerInterface
         */
        protected $applicationLogger;

        /**
         * Delete a forum post and log operation
         */
        public function deletePost(Post $post): void
        {
                $this->applicationLogger->log('Removing post ' . $post->getTitle(), LOG_INFO);
                $this->posts->remove($post);
        }

}
```

If you have to do this in a lot of places, the logging would become a part of you domain model logic. You would have to inject all the logging dependencies in your models. Since logging is nothing that a domain model should care about, this is an example of a non-functional requirement and a so-called cross-cutting concern.

With AOP, the code inside your model would know nothing about logging. It will just concentrate on the business logic.

Example: Logging with AOP (your class):

```
namespace Examples\Forum\Domain\Model;

class Forum {

        /**
         * Delete a forum post
         */
        public function deletePost(Post $post): void
        {
                $this->posts->remove($post);
        }

}
```

The logging is now done from an AOP aspect. It's just a class tagged with @aspect and a method that implements the specific action, an before advice. The expression after the @before tag tells the AOP framework to which method calls this action should be applied. It's called pointcut expression and has many possibilities, even for complex scenarios.

Example: Logging with AOP (aspect):

```
namespace Examples\Forum\Logging;
use Examples\Forum\Logger\ApplicationLoggerInterface; use Neos\Flow\AOP\JoinPointInterface;

/**
 * @Flow\Aspect
 */
class LoggingAspect {

        /**
         * @Flow\Inject
         * @var ApplicationLoggerInterface
         */
        protected $applicationLogger;

        /**
         * Log a message if a post is deleted
         *
         * @Flow\Before("method(Examples\Forum\Domain\Model\Forum->deletePost())")
         */
        public function logDeletePost(JoinPointInterface $joinPoint): void
        {
                $post = $joinPoint->getMethodArgument('post');
                $this->applicationLogger->log('Removing post ' . $post->getTitle(), LOG_INFO);
        }

}
```

As you can see the advice has full access to the actual method call, the join point, with information about the class, the method and method arguments.

AOP concepts and terminology

At the first (and the second, third, ...) glance, the terms used in the AOP context are not really intuitive. But, similar to most of the other AOP frameworks, we better stick to them, to keep a common language between developers. Here they are:

Aspect

An aspect is the part of the application which cross-cuts the core concerns of multiple objects. In Flow, aspects are implemented as regular classes which are tagged by the @aspect annotation. The methods of an aspect class represent advices, the properties may be used for introductions.

Join point

A join point is a point in the flow of a program. Examples are the execution of a method or the throw of an exception. In Flow, join points are represented by the `Neos,Flow,AOP,JoinPoint` object which contains more information about the circumstances like the name of the called method, the passed arguments, or the type of the exception thrown. A join point is an event that occurs during the program flow, not a definition that defines that point.

Advice

An advice is the action taken by an aspect at a particular join point. Advices are implemented as methods of the aspect class. These methods are executed before and / or after the join point is reached.

Pointcut

The pointcut defines a set of join points which need to be matched before running an advice. The pointcut is configured by a pointcut expression which defines when and where an advice should be executed. Flow uses methods in an aspect class as anchors for pointcut declarations.

Pointcut expression

A pointcut expression is the condition under which a join point should match. It may, for example, define that join points only match on the execution of a (target-) method with a certain name. Pointcut expressions are used in pointcut- and advice declarations.

Target

A class or method being adviced by one or more aspects is referred to as a target class /-method.

Introduction

An introduction redeclares the target class to implement an additional interface. By declaring an introduction it is possible to introduce new interfaces and an implementation of the required methods without touching the code of the original class. Additionally introductions can be used to add new properties to a target class.

In AOP, aspects, advice, pointcuts, and join points work together to allow developers to modularize cross-cutting concerns and separate them from the main functionality of the application. By defining aspects and pointcuts, developers can apply advice at specific join points in the application to add new behavior or modify existing behavior without modifying the existing code.

Language for Paradigm 2: JBoss AOP

What's JBoss AOP?

JBoss AOP is a 100% Pure Java aspected oriented framework usable in any programming environment or tightly integrated with our application server. Aspects allow you to more easily modularize your code base when regular object oriented programming just doesn't fit the bill. It can provide a cleaner separation from application logic and system code. It provides a great way to expose integration points into your software.

JBoss AOP is not only a framework, but also a prepackaged set of aspects that are applied via annotations, pointcut expressions, or dynamically at runtime. Some of these include caching, asynchronous communication, transactions, security, remoting, and many more.

An aspect is a common feature that is typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can not find a way to express this structure in code with traditional object-oriented techniques.

For example, metrics is one common aspect. To generate useful logs from your application, you have to (often liberally) sprinkle informative messages throughout your code. However, metrics is something that your class or object model really should not be concerned about. After all, metrics is irrelevant to your actual application: it does not represent a customer or an account, and it does not realize a business rule. It's simply orthogonal.

Creating Aspects in JBoss AOP

In short, all AOP frameworks define two things: a way to implement crosscutting concerns, and a programmatic construct — a programming language or a set of tags to specify how you want to apply those snippets of code. Let us take a look at how JBoss AOP, its cross-cutting concerns, and how you can implement a metrics aspect in JBoss Enterprise Application Platform. The first step in creating a metrics aspect in JBoss AOP is to encapsulate the metrics feature in its own Java class. The following code extracts the try/finally block in our first code example's BankAccountDAO.withdraw() method into Metrics, an implementation of a JBoss AOP Interceptor class. The following example code demonstrates implementing metrics in a JBoss AOP Interceptor

```
public class Metrics implements org.jboss.aop.advice.Interceptor
{
   public Object invoke(Invocation invocation) throws Throwable
   {
     long startTime = System.currentTimeMillis();
     try
     {
       return invocation.invokeNext();
     }
     finally
     {
       long endTime = System.currentTimeMillis() - startTime;
       java.lang.reflect.Method m = ((MethodInvocation)invocation).method;
       System.out.println("method " + m.toString() + " time: " + endTime + "ms");
     }
   }
}
```

Under JBoss AOP, the Metrics class wraps withdraw(): when calling code invokes withdraw(), the AOP framework breaks the method call into its parts and encapsulates those parts into an Invocation object. The framework then calls any aspects that sit between the calling code and the actual method body.

When the AOP framework is done dissecting the method call, it calls Metrics's invoke method at line 3. Line 8 wraps and delegates to the actual method and uses an enclosing try/finally block to perform the

timings. Line 13 obtains contextual information about the method call from the Invocation object, while line 14 displays the method name and the calculated metrics.

Applying Aspects in JBoss AOP

To apply an aspect, you define when to execute the aspect code. Those points in execution are called pointcuts. An analogy to a pointcut is a regular expression. Where a regular expression matches strings, a pointcut expression matches events or points within your application. For example, a valid pointcut definition would be, "for all calls to the JDBC method executeQuery(), call the aspect that verifies SQL syntax." An entry point could be a field access, or a method or constructor call. An event could be an exception being thrown. Some AOP implementations use languages akin to queries to specify pointcuts. Others use tags. JBoss AOP uses both. The following listing demonstrates defining a pointcut for the Metrics:

```
<interceptor name="SimpleInterceptor" class="com.mc.Metrics"/>
<bind pointcut="execution (public void com.mc.BankAccountDAO->withdraw(double amount))" >
    <interceptor-ref name="SimpleInterceptor" />
</bind>
<bind pointcut="execution (* com.mc.billing.->(..))">
    <interceptor-ref name="com.mc.Metrics" />
</bind>
```

- Defines the mapping of the interceptor name to the interceptor class.
- Lines 2-4 define a pointcut that applies the metrics aspect to the specific method BankAccount-DAO.withdraw().
- Lines 5-7 define a general pointcut that applies the metrics aspect to all methods in all classes in the com.mc.billing package. There is also an optional annotation mapping if you prefer to avoid XML.

Packaging AOP Applications

To deploy an AOP application in JBoss Enterprise Application Platform you need to package it. AOP is packaged similarly to SARs (MBeans). You can either deploy an XML file directly in the deploy/ directory with the signature *-aop.xml along with your package (this is how the base-aop.xml, included in the jboss-aop.deployer file works) or you can include it in the JAR file containing your classes. If you include your XML file in your JAR, it must have the file extension .aop and a jboss-aop.xml file must be contained in a META-INF directory, for instance: META-INF/jboss-aop.xml. In the JBoss Enterprise Application Platform 5, you must specify the schema used, otherwise your information will not be parsed correctly. You do this by adding the xmlns="urn:jboss:aop-beans:1:0 attribute to the root aop element, as shown here:

```
<aop xmlns="urn:jboss:aop-beans:1.0">
</aop>
```

Analysis

Strengths Of OOP

- Software maintenance improved: For the reasons referenced above, object-oriented programming is likewise simpler to keep up with. Since the plan is secluded, a piece of the framework can be refreshed if there should arise an occurrence of issues without a need to roll out huge scope improvements.
- Quicker improvement: Reuse empowers quicker advancement. Object-situated programming dialects accompany rich libraries of articles, and code created during projects is additionally reusable in later ventures.

- Cost of development lowered: The reuse of programming likewise brings down the expense of advancement. Normally, more exertion is placed into the article situated examination and plan, which brings down the general expense of improvement.
- Good quality software: Faster improvement of programming and lower cost of advancement permits additional time and assets to be utilized in the confirmation of the product. Albeit quality is reliant upon the experience of the groups, object-situated programming will in general bring about greater programming.

Weakness of OOP

- Steep learning curve: The thought process involved in object-oriented programming may not be natural for some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be challenging to comprehend initially.
- Overhead: OOP code can be more verbose than code written in other paradigms, which can result in slower performance. Additionally, OOP often requires more memory and processing power than other paradigms.
- Slower programs: Object-oriented programs are typically slower than procedure-based programs, as they typically require more instructions to be executed.

Features of OOP in Ruby

- Mixins: Ruby has a feature of single inheritance only. Ruby has classes as well as modules. A module has methods but no instances. Instead, a module can be mixed into a class, which adds the method of that module to the class. It is similar to inheritance but much more flexible.
- Dynamic typing and Duck typing: Ruby is a dynamic programming language. Ruby programs are not compiled. All class, module and method definition are built by the code when it run.

  Ruby variables are loosely typed language, which means any variable can hold any type of object. When a method is called on an object, Ruby only looks up at the name irrespective of the type of object. This is duck typing. It allows you to make classes that pretend to be other classes.
- Singleton methods: Ruby singleton methods are per-object methods. They are only available on the object you defined it on.
- Missing method: If a method is lost, Ruby calls the `method_missing` method with the name of the lost method.
- Case Sensitive: Ruby is a case-sensitive language. Lowercase letters and uppercase letters are different.

Strengths of Aspect-oriented programming

- Modularity: AOP helps to improve the modularity of software by allowing developers to separate cross-cutting concerns into modular units known as aspects. This can make it easier to understand, modify, and reuse code.
- Maintainability: By separating concerns, AOP can make it easier to maintain and modify code over time. This can be particularly useful when adding new features or behavior to an application without modifying the existing code.
- Reusability: Aspects can be reused across different projects, which can help to improve the efficiency and productivity of developers.
- Performance: AOP can be used to optimize the performance of an application by identifying and addressing bottlenecks or other inefficiencies.

Weakness of Aspect-oriented programming

- Complexity: AOP can add complexity to an application, as it requires developers to learn and use an AOP framework and to understand the concepts of aspects, advice, pointcuts, and join points.
- Overhead: AOP can introduce additional overhead, as it requires the use of an AOP framework and the execution of additional code at runtime.
- Compatibility: AOP may not be compatible with all programming languages and environments, which can limit its use in some situations.
- Debugging: Debugging an application that uses AOP can be more challenging, as it may be difficult to understand the execution flow and the interactions between different aspects.

Features of Aspect-Oriented - JBoss AOP

- Cross-cutting concerns: AOP helps in addressing cross-cutting concerns, which are aspects of a program that affect multiple modules or components. Examples include logging, security, and transaction management. AOP allows you to modularize and encapsulate these concerns separately from the main business logic.
- Aspect: In AOP, an aspect is a module that encapsulates cross-cutting concerns. Aspects are defined separately from the main business logic and can be applied selectively to specific parts of the codebase.

Comparison

Paradigm 1: Object-Oriented Programming (OOP) in Ruby

Similarities with Aspect-Oriented Programming (AOP):

- Modularity: Both paradigms aim to improve code modularity. In OOP, modularity is achieved through classes and objects, while in AOP, it's achieved by separating cross-cutting concerns into aspects.
- Encapsulation: OOP promotes encapsulation by bundling data and methods that operate on the data within a class. Similarly, AOP encapsulates cross-cutting concerns in aspects.

Differences with Aspect-Oriented Programming (AOP):

- Inheritance vs. Cross-Cutting Concerns: OOP relies heavily on inheritance for code reuse and structure. In contrast, AOP focuses on addressing cross-cutting concerns like logging, security, and performance that may not fit neatly into a class hierarchy.
- Aspect vs. Class: In OOP, you create classes to represent objects and their behavior. In AOP, you create aspects to encapsulate cross-cutting concerns, which are then applied to different parts of the code.

Paradigm 2: Aspect-Oriented Programming (AOP) with JBoss AOP

Similarities with Object-Oriented Programming (OOP):

- Modularity: Both paradigms aim for modularity, but AOP achieves it by separating cross-cutting concerns, while OOP achieves it through class-based modularity.
- Encapsulation: AOP encapsulates cross-cutting concerns in aspects, similar to how OOP encapsulates data and behavior within classes.

Differences with Object-Oriented Programming (OOP):

- Cross-Cutting Concerns vs. Class Hierarchy: AOP focuses on cross-cutting concerns that affect multiple classes, whereas OOP often involves creating class hierarchies to model relationships and behaviors.

- Pointcuts and Join Points: AOP introduces concepts like pointcuts and join points to specify when and where aspects should be applied, which are not present in traditional OOP.

Few more comparisons:

- Modularity: Both paradigms address modularity concerns, but they approach it differently. OOP achieves modularity through classes and inheritance, while AOP achieves it through aspects and cross-cutting concerns.
- Encapsulation: Both paradigms promote encapsulation, keeping related functionality together. In OOP, encapsulation is achieved within classes, while in AOP, it's achieved within aspects.
- Code Reusability: OOP promotes code reuse through class inheritance and polymorphism. AOP promotes reuse by separating cross-cutting concerns into aspects that can be applied to multiple classes.

Challenges Faced

I faced many challenges early in the project, understanding a new programming language such as JBoss was a significant obstacle. It took some time to analyze its logical structure and specific vocabulary.

The compilation of the project report also presented its own challenges. Striking the right balance between technical detail and clarity for different readers required careful consideration. The report writing program includes collaborative learning and seeking help with specific language difficulties.

Despite these challenges, I pulled through, implementing collaborative learning and asking for help with specific language difficulties. The report was thoroughly reviewed and revised, resulting in a concise but comprehensive document.

Conclusion

In conclusion, the text provides an overview of OOP concepts in Ruby, including classes, inheritance, modules, and the use of self. It then introduces AOP, highlighting its benefits, key terms, and the implementation of aspects using JBoss AOP. The two paradigms offer different approaches to structuring and organizing code, addressing modularity and maintainability concerns in distinct ways.

References

Include any references or sources you consulted for your assignment.

- www.tutorialspoint.com/object_oriented_analysis_design/index.htm
- www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP
- www.trio.dev/blog/object-oriented-programming
- www.codecademy.com/resources/docs/general/programming-paradigms/object-oriented-programming
- emeritus.org/blog/coding-what-is-object-oriented-programming/
- www.pluralsight.com/resources/blog/cloud/what-is-the-ruby-programming-language
- www.scaler.com/topics/ruby/oops-in-ruby/
- medium.com/launch-school/the-basics-of-oop-ruby-26eaa97d2e98
- spiceworks.com/tech/devops/articles/what-is-aop/
- medium.com/hprog99/aspect-oriented-programming-b9a06ca256db
- https://access.redhat.com/documentation/en-us/jboss_enterprise_application_platform/5/html/administration_and_configuration_guide/ch12s02