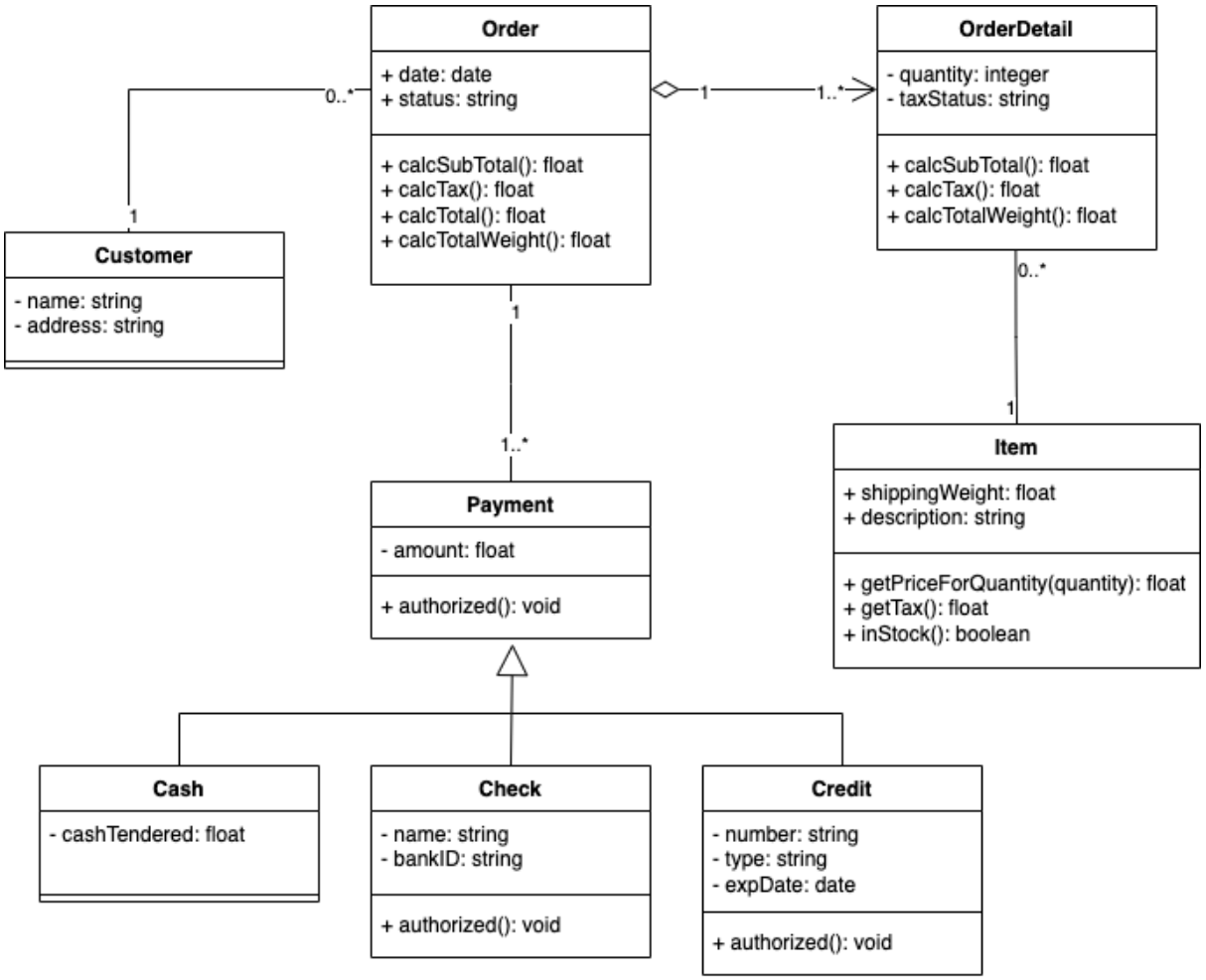
Recuperatorio I2: Ingeniería de Software
Semestre 1- 2023

Nombre Completo: _____

Sección (profesor): _____

P1 (4 pts). Considere el siguiente diagrama de clases y suponga que se realizo la implementación correcta en base al diagrama de clases. Teniendo en cuenta lo anterior, indique que afirmaciones son verdaderas.



I. Existen 8 clases, donde todos los atributos son privados y todos los metodos son publicos.

II. La ejecución de my_item.inStock() retorna un booleano, considerando que a my_item se le asigna un objeto Item.

III. Una instancia de Credit tiene un total de 4 atributos.

IV. La relación que existe entre Order y OrderDetail es una agregación y se entiende como que un objeto Order está compuesto de uno o más objetos OrderDetail.

V. La relación que existe entre Order y OrderDetail es una composición y se entiende como que un objeto Order está compuesto de uno o más objetos OrderDetail.

Alternativas:

A. Solo I, II, III y V son verdaderas.
B. Solo II, III y IV son verdaderas.
C. Solo I, II, III y IV son verdaderas.
D. Solo II, III y V son verdaderas.

P2 (4 pts). Indique cuales de las siguientes afirmaciones son verdaderas acerca de la arquitectura de microservicios:

I. Permite una escalabilidad mas sencilla
II. El versionamiento es mas sencillo
III. Requiere un esfuerzo de desarrollo menor a la arquitectura cliente servidor.
IV. Permite una flexibilidad tecnologica.

Alternativas:

A. Solo I y IV son verdad.
B. Todas son verdad
C. Solo IV y III son verdad
D. Solo I y II son verdad
E. Todas son falsas.

P3 (8 pts). Escribe los nombres de los patrones correspondientes a cada una de las siguientes definiciones **El puntaje es directamente proporcional a las respuestas correctas que tuvieron, es decir, 1,3333 pts c/u**

Nombre	Descripción
Proxy	proporciona un sustituto o representante de otro objeto para controlar el acceso a este.
Decorator	permite agregar funcionalidad adicional a un objeto dinámicamente. Permite extender las capacidades de un objeto sin modificar su estructura básica.
Adapter	permite que dos interfaces incompatibles trabajen juntas. Actúa como un puente entre dos clases diferentes, convirtiendo la interfaz de una clase en otra interfaz que el cliente espera
Composite	permite tratar un grupo de objetos de manera similar a un objeto individual.
Template	define el esqueleto de un algoritmo en una clase base, delegando la implementación de ciertos pasos a las clases hijas.
Strategy	permite definir una familia de algoritmos, encapsulándolos y haciéndolos intercambiables.

P4 (4 pts). Bruno menciona que implementó la clase Profiler usando el **patrón Singleton**. Considerando el siguiente código, responda las siguientes preguntas:

1) **(2 pts)** ¿Qué imprime al ejecutarse el código de Bruno?
2) **(2 pts)** ¿Bruno implementó bien la clase Profiler según el patrón Singleton? Si la respuesta es negativa, modifique el método self.instance con la implementación correcta.

```
class Profiler
 private_class_method :new
 def self.instance
  @instance = new
  return @instance
 end
end


ob1 = Profiler.instance
ob2 = Profiler.instance


if ob1.equal?(ob2)
 puts 'El código funciona, ob1 y ob2 son la misma instancia.'
else
 puts 'El código falla, ob1 y ob2 son distintas instancias.'

end
```

R1. Imprime *"El código falla, ob1 y ob2 son distintas instancias".*

R2.

```
class Profiler
 private_class_method :new
 def self.instance
   if @instance == nil
      @instance = new
   end
  return @instance
 end
end
```

P5 (20 pts). Considere el código de este reproductor de música:

```
class Track
    attr_reader :title
    attr_reader :duration
    def initialize(title,duration)
        @title = title
        @duration = duration
        @playing = false
    end
    def playing?
        return @playing
    end
    def play()
        @playing = true
    end
    def stop()
        @playing = false
    end
end


class MusicPlayer
   def initialize
       @tracks =[]
       @current_index = 0
       @strategy = "sequence"
   end
   def add(track)
       @tracks.push(track)
   end
   def strategy=(name)
       @strategy = name
   end
   def playFirst()
       if @tracks.length > 0
           @tracks[0].play()
       end
   end
   def playNext()
       if @tracks.length > 0
           @tracks[@current_index].stop()
           if @strategy == "sequence"
               @current_index = (@current_index + 1) % @tracks.length
           elsif @strategy == "random"
               @current_index = rand(@tracks.length)
           end
           @tracks[@current_index].play()
       end
   end
   def print()
       @tracks.each do |track|
           if track.playing?
               puts (track.title + ":" + track.duration.to_s).green
           else
               puts (track.title + ":" + track.duration.to_s).blue
           end
       end
   end
end
```

El mismo modela un reproductor de música que permite diversas estrategias para cambiar de canción. Sin embargo, no es extensible para poder agregar nuevas estrategias de reproducción.

- **(2 pts)** Explique porque el código anterior no es extensible para poder agregar nuevas estrategias de reproducción.
- **(2 pts)** Explique porque el código anterior rompe el encapsulamiento e incrementa el acoplamiento entre clases.
- **(10 pts)** Modifique el código anterior aplicando el **patrón strategy**.
- **(3 pts)** Modifique el código anterior para reducir el acoplamiento entre clases y mejorar la cohesión de la clase Track.
- **(3 pts)** Escriba un código de ejemplo, donde usted cree un dos music players uno con estrategia de reproducción aleatoria y otro con estrategia de reproducción secuencial.

A. En el código anterior para agregar una nueva estrategia es necesario modificar el método **playNext(),** por lo que no cumple la definición de extensibilidad.
    a. cualquier respuesta que tenga relación con lo anterior puede ser considerada valida.
B. El método **print()** accede de forma directamente a los atributos de la clase Track, esto rompe el encapsulamiento. El atributo "title" debería ser privado (encapsulado), ya que al ser publico permite agrega dependencias innecesarias como en el metodo **print().**
    a. cualquier respuesta que tenga relación con lo anterior puede ser considerada valida.
C. ...

```
class PlayStrategy

    def nextIndex(current_index,len)
        raise NotImplementedError
    end
end
class RandomStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return rand(len)
    end
end
class SequenceStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return (current_index + 1) % len
    end
end
class RepeatStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return current_index
    end
end
```

```
class MusicPlayer
  ...
  def playNext()
    if @tracks.length > 0
      @tracks[@current_index].stop()
      @current_index = @strategy.nextIndex(@current_index,@tracks.length)
      @tracks[@current_index].play()
    end
  end
end
```

La idea este tener una jerarquia de clases y encapsular una operación (estrategia o algoritmo) en un método que es sobre-escrito por las clases hijas (como se ve en amarillo). Este método debe ser llamado en la clase ***MusicPlayer*** (como el código en naranja).

El el método en la jerarquía de estrategias puede variar (diferente nombre o argumento) ya que cada persona puede tener perspectiva del problema. Lo que no de faltar es que cada sub-clase redefina ese método, además, el método debe tener cierta lógica.

D. ...

```
class MusicPlayer
  def print()
    @tracks.each do |track|
      track.print()
    end
  end
end

class Track
  ...
  def print()
    if playing?
      puts (@title + ":" + @duration.to_s).green
    else
      puts (@title + ":" + @duration.to_s).blue
    end
  end
end
```

La solución es utilizar delegación, es decir, llamar a un método de la clase Track que haga la tarea sin romper el encapsulamiento. Es la solución anterior se creo un metodo print y se llamo al mismo desde MusicPlayer, el nombre del metodo puede variar, la restriccion es que la clase MusicPlayer no llame a ningun atributo de la clase **Track**, de esta forma se reduce el acoplamiento.

E. ...

```
class MusicPlayer
  ...
  def strategy=(strategy)
    @strategy = strategy
  end

end
```

```
a= MusicPlayer.new
a.strategy = RandomStrategy.new
b= MusicPlayer.new
b.strategy = SequenceStrategy.new
```

El objeto **MusicPLayer** debe tener una forma de poder cambiar la estrategia, normalmente esto se hace a través de un método que permite settear la estrategia, pero podria existir varias opciones. La idea principal que uno pueda cambiar de estrategia al reproductor.

P6 (20 pts). Ivan analiza una parte de un sistema de banco encargada de notificar al dueño de la cuenta si existe algun cambio en su balance. Nota que esa parte del sistema está ilustrada en el código siguiente, y que por el momento solo envia correos al dueño de la cuenta cuando corresponde utilizando la siguiente instruccion. Sin embargo, Ivan te comenta que muchos clientes del banco desean también recibir notificaciones a sus celulares mediante SMS. Para realizar este cambio, es necesario modificar el código actual (agregar una linea en 2 metodos de la clase) e Ivan no quiere hacer esto. Ayuda a Ivan a refactorizar está parte del programa aplicando el **patron observer**, tal que pueda agregar las notificaciones según sea necesario:

- Notificaciones por email: Cada que el balance cambia, se debe imprimir en consola *"Send email to owner: the actual balance is ….".*
- Notificaciones por SMS: Cada que el balance cambia, se debe imprimir en consola *"Send SMS to owner: the actual balance is ….".*

```
class Account
      def initialize()
            @balance = 0
      end
      def deposit(amount)
            @balance += amount
            puts "Send email to owner: the actual balance is #{@balance}"
      end
      def withdraw(amount)
            if @balance - amount >= 0
                  @balance -= amount
                  puts "Send email to owner: the actual balance is #{@balance}"
            end
      end
end
```

Nota. Si aplica correctamente el patron observer, sera posible agregar mas tipos de notificaciones de forma extensible (sin modificar el código actual) y flexible. Es decir, se podra quitar o agregar las notificaciones de forma dinamica (mientras el programa se sigue ejecutando).

```
class Notification
  def update(balance)
    raise NotImplementedError
  end
end

class SMSNotification
  def update(balance)
    puts "Send SMS to owner: the actual balance is #{balance}"
  end
end

class EmailNotification
  def update(balance)
    puts "Send email to owner: the actual balance is #{balance}"
  end
end
class Account
        def initialize()
                @balance = 0
                @observers = []
        end
        def addObserver(obs)
          @observers.push(obs)
        end
        def notifyAll()
          for obs in @observers
            obs.update(@balance)
          end
        end
        def deposit(amount)
                @balance += amount
                notifyAll()
        end
        def withdraw(amount)
                if @balance - amount >= 0
                        @balance -= amount
                        notifyAll()
                end
        end
end
```

Debe existir una jerarquia de clases que tenga el método update (amarillo), la clase cuenta tiene que tener una lista de observers, una forma de agregar un observador y una forma de notificarlos (naranja claro). La clase account debe notificar a los observadores cada vez que cambia de estado (naranja fuerte).

Es posible asignar puntajes parciales si se olvidaron algun detalle de la implementación del decorator, por ejemplo si olvidaron el addObserver (-3 puntos), si olvidaron el notifyAll (-10) por que es el mas importante.

Si no tienen la jerarquia con el update, la solución no es valida porque significaria que conocimiento del patron observador es nulo.

P7 (20 pts). Lucia está trabajando en un juego de estrategia online que tiene estrategias por defecto. Según el estudiante todas las razas del juego cuando les toca su turno realizan casi los mismos pasos, ya que pueden construir similares estructuras y atacan considerando ciertos aspectos. Considere que en la primera versión del juego existen dos tipos de raza: humanos y orcos. Dado el siguiente codigo en Ruby:

- Indique que problemas de diseño tiene la implementación.
- Utilize el **patron Template method** para realizar una refactorización del código ilustrado.

```ruby
class HumansAI
  def turn
    collectResources()
    buildStructures()
    smartMove()
  end
  def collectResources()
    puts "Collecting resources"
  end
  def buildStructures()
    puts "Building farms"
    puts "Building barracks"
    puts "Building strongholds"
  end
  def smartMove()
    enemy = closestEnemy()
    if enemy == nil
      puts "Sending scouts to center"
    else
      puts "Sending warriors to enemy"
    end
  end
end
```

```ruby
class OrcsAI
  def turn
    collectResources()
    buildStructures()
    attack()
  end
  def collectResources()
    puts "Collecting resources"
  end
  def buildStructures()
    puts "Building barracks"
    puts "Building strongholds"
  end
  def attack()
    enemy = closestEnemy()
    if enemy != nil
      puts "Sending warriors to enemy"
    end
  end
end
```

Nota. Si su solucion es correcta se eliminara casi todo el codigo duplicado entre ambas clases.

```ruby
class ObjectAI
  def turn
    collectResources()
    buildStructures()
    finalAction()
  end
  def collectResources()
    puts "Collecting resources"
  end
  def buildStructures()
    puts "Building barracks"
    puts "Building strongholds"
  end
  def sendWarriors()
    puts "Sending warriors to enemy"
```

```
  end
end
class HumansAI < ObjectAI
  def finalAction()
    smartMove()
  end

  def buildStructures()
    puts "Building farms"
    super
  end
  def smartMove()
    enemy = closestEnemy()
    if enemy == nil
      puts "Sending scouts to center"
    else
      sendWarriors()
    end
  end
end
class OrcsAI < ObjectAI
  def finalAction()
    attack()
  end

  def attack()
    enemy = closestEnemy()
    if enemy != nil
       sendWarriors()
    end
  end
end
```

A) (5 pts) Indica que hay codigo duplicado.

B) (15 pts) Debe existir una clase padre que tenga un método plantilla (como el de naranja), el mismo debe llamar a uno o mas métodos que esten sobre-escritos en la clase hija.

Ademas la idea era eliminar código duplicado como dice el enunciado, habian varias formas de eliminar el código duplicado, en la solucion anterior se mostraron algunas. Se debe aplicar un descuento (a criterio) a pedazos de código duplicado que era sencillo de eliminar y no se hizo. Por ejemplo,

- subir a la clase padre el método collectResults, si no se hizo -3 puntos.
- llamar al método `buildStructures` de la clase padre desde la clase HumansAI, -3 puntos.
- abstraer el ultimo paso del método **turn,** en la solucion anterior se llamo al ultimo paso **finalAction** que varia dependiendo a la subclase. -3 puntos.

P8 (20 pts). Laura es nueva en el equipo de desarrollo y observa un programa para pedidos de cafe. Por el momento, Laura nota que este programa está en su primera versión y es posible vender expressos con algunos agregados (p.ej., leche batida, mocha). Teniendo en cuenta el código de la primera versión, Laura debe modificar el programa para : (i) agregar más variedad de bebidas y agregados y (i) calcular los precios de la manera más simple y más manejable si se realizan cambios. Ayuda a Laura a modificar el código utilizando el **patron Decorator** para facilitar la extensibilidad.

```
class Beverage
      def description
            puts "Sample beverage"
      end
      def cost
            raise NotImplementedError
      end
end
```

```
class Expresso < Beverage
      def description
            puts "Expresso"
      end
      def cost
            return 5.5
      end
end
```

```
class ExpressoMocha < Beverage
      def description
            puts "Expresso"
            puts "Mocha"
      end
      def cost
            return 5.5 + 1.5
      end
end
```

```
class ExpressoMochaWhip < Beverage
      def description
            puts "Expresso"
            puts "Mocha"
            puts "Whip"
      end
      def cost
            return 5.5 + 1.5 + 0.5
      end
end
```

```
class Beverage
      def description
            puts "Sample beverage"
      end
      def cost
            raise NotImplementedError
      end
end

class Expresso < Beverage
      def description
            puts "Expresso"
      end
      def cost
            return 5.5
      end
end

class DecoratedExpresso < Beverage
    def initialize(decoratedExpresso)
        @decoratedExpresso = decoratedExpresso
    end
```

```
end



class ExpressoMocha < DecoratedExpresso
      def description
            @decoratedExpreso.description()
            puts "Mocha"
      end
      def cost
            return @decoratedExpreso.cost + 1.5
      end
end

class ExpressoMochaWhip < DecoratedExpresso
      def description
            @decoratedExpreso.description()
            puts "Whip"
      end
      def cost
            return @decoratedExpreso.cost + 0.5
      end
end
```

Debe existir una nueva clase que sea igual a la **DecoratedExpresso,** esta tener dentro un objeto expreso (el decorado). Las clases hijas deben heredar de esta clase (código naranja) y tener una llama recursiva al objeto decorado (por ejemplo @decoratedExpresso.description()) para poder reutilizar el código.

Se puede descontar puntos si no se olvideron algun detalle, pero en general no existen muchas otras soluciones, todas las soluciones deberian tender a la solución de arriba