IIC 2143 Ingeniería de Software
Interrogación 3 - Semestre 1 /2019
Secciones 01 y 02

Responda cada pregunta en hoja separada
Entregue una hoja con su nombre para cada pregunta aunque sea en blanco
Tiempo: 2:15
Recuerden que están bajo el código de honor

Pregunta 1:

Uno de los patrones de diseño vistos en clase es el patrón *Observer*, el cual le permite a una clase *Subject* notificar cambios a una clase *Observer* sin necesidad de aumentar el nivel de acoplamiento entre ambas. Sin embargo, una debilidad de la implementación vista en clases de este patrón es que, para realizar notificaciones, es necesario agregar a la clase *Subject* un atributo adicional a su definición que contenga todos los *Observers* que tendría que notificar. Se podría argumentar que lo anterior no debiese ser responsabilidad de la clase *Subject*, por lo que dicha implementación disminuiría levemente su cohesión.

Una implementación alternativa del patrón *Observer* provista por algunas plataformas es la posibilidad de canalizar todas las observaciones y notificaciones a través de una clase única que llamaremos *NotificationCenter*. Esta clase define los siguientes tres métodos:

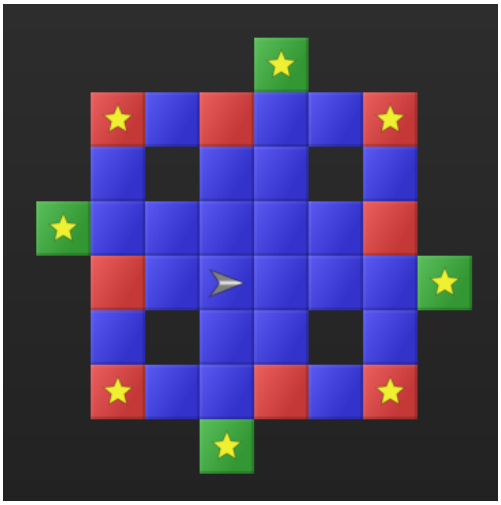
- `register_listener(event_id, listener)`
- `delete_listener(event_id, listener)`
- `notify(event_id, *arguments)`

El parámetro `event_id` es un identificador (que en Ruby puede representarse por un símbolo) que permite identificar un evento específico. Al invocar el método `notify` del *NotificationCenter*, se invoca subsecuentemente el método `update(…)` de todos los *Observers* que se hallan registrado con el respectivo `event_id`. Para que esto funcione, dichos *Observers* deben efectivamente implementar ese método. Con lo anterior en mente, el flujo para observar eventos y notificar cambios es el siguiente:

- Observer implementa método update
- Observer consigue una referencia al NotificationCenter
- Observer se registra como listener para un evento X
- Subject realiza una operación de interés
- Subject consigue una referencia al NotificationCenter
- Subject invoca el método notify para el evento X
- NotificationCenter llama al método update de los Observers registrados con el evento X

Escriba una implementación para la clase NotificationCenter aquí descrita en Ruby e ilustre su funcionamiento escribiendo un breve script de ejemplo que reproduzca el flujo anteriormente descrito. (6 puntos)

Pregunta 2:



El juego gratuito en línea Robozzle™ (ver imagen adjunta) es una aplicación que permite a los usuarios programar un sencillo robot que luego se desplace por un tablero en formato grilla predefinido. El juego define solamente tres comandos:

- Forward: mueve el robot una casilla hacia adelante en la dirección hacia la que esté apuntando.
- Turn Left: rota la orientación del robot 90 grados hacia la izquierda sin cambiar de posición.
- Turn Right: rota la orientación del robot 90 grados hacia la derecha sin cambiar de posición.

La idea del juego es, dada una posición inicial del robot, definir el listado de comandos que el robot deberá ejecutar a priori y luego ejecutarlos uno por uno para calcular la posición y orientación final del robot.

Usted deberá escribir una aplicación en consola que lea strings ingresados por el usuario. En primera instancia, la aplicación debe permitir definir los comandos que el robot deberá ejecutar y guardarlos en una lista. Cuando el usuario ingresa el string “finish”, se pasa a la siguiente etapa. Luego, el usuario debe ser capaz de ejecutar las siguientes instrucciones:

- Step: sea X un contador con valor inicial 0, ejecuta el comando en la posición X del listado de comandos, aumenta el contador en 1 e imprime la posición y orientación del robot. Si no hay más comandos por ejecutar, esta instrucción no hace nada.
- Undo: deshace el último comando *Step* ejecutado e imprime la posición y orientación del robot.
- Execute: ejecuta todo el listado de comandos desde el último *Step* en orden e imprime la posición y orientación final del robot. El programa entonces termina.

En su implementación, es imperativo el uso del patrón Command visto en clases. Puede utilizar coordenadas (x,y) para representar la posición del robot. Asuma que el robot parte en la posición inicial (0,0) y mirando hacia arriba. (6 puntos)

Pregunta 3:

Se quiere construir una calculadora que sume y multiplique números complejos. Un número complejo puede darse en forma cartesiana (x, y) o en forma polar (r, θ). En ambos casos se usará un arreglo con dos elementos para representarlos.

(1) El resultado c_3 de las operaciones de suma y producto para dos números complejos c_1 y c_2 expresados en forma cartesiana son:

$Dados\ c_1 = [x_1, y_1],\ c_2 = [x_2, y_2],\ c_3 = [x_3, y_3]$

$Si\ c_3 = c_1 + c_2\ entonces$:

$$x_3 = x_1 + x_2$$

$$y_3 = y_1 + y_2$$

$Si\ c_3 = c_1 * c_2\ entonces$:

$$x_3 = x_1 * x_2 - y_1 * y_2$$

$$y_3 = x_1 * y_2 + x_2 * y_1$$

(2) El resultado c_3 de las operaciones de suma y producto para dos números complejos c_1 y c_2 expresados en forma polar son:

$Dados\ c_1 = [r_1, \theta_1],\ c_2 = [r_2, \theta_2],\ c_{3_{cart}} = [x_3, y_3],\ c_{3_{polar}} = [r_3, \theta_3]$

$Si\ c_{3_{cart}} = c_1 + c_2\ entonces$:

$$x_3 = r_1 \cos\theta_1 + r_2 \cos\theta_2$$

$$y_3 = r_1 \sin\theta_1 + r_2 \sin\theta_2$$

$Si\ c_{3_{polar}} = c_1 * c_2\ entonces$:

$$r_3 = r_1 * r_2$$

$$\theta_3 = \theta_1 + \theta_2$$

(3) Recuerde que dado un número [x,y] expresado en forma cartesiana, se puede obtener su forma polar [r, θ] como: $r = sqrt(x^2 + y^2)$ y $\theta = tg^{-1}\left(\frac{y}{x}\right)\ para\ x > 0$. Y también que a la inversa, desde un número [r, θ] expresado en forma polar se puede obtener su forma cartesiana [x,y] como: $x = r * \cos(\theta), y = r * \sin(\theta)$.

a) Escriba la clase ComplexArith que lleva a cabo las operaciones aritméticas con ayuda de un objeto estrategia. (1.5 puntos)

b) Escriba las clase abstracta Strategy y las dos clases concretas Strategy1 y Strategy2. Las estrategias concretas funcionan de la siguiente forma:
   - Strategy1 hace todos los cálculos en coordenadas cartesianas (1).
   - Strategy2 convierte los operandos a coordenadas polares (3) y luego usa el cálculo en polares (2) para finalmente convertir el resultado nuevamente en cartesianas (3). Puede asumir que Strategy2 solo opera con números complejos con parte real positiva.

   (3 puntos)

c) Escriba un pequeño programa de prueba que efectúe la operación (c1 * c2) + c3 en que todos los números c1 = [4, 7], c2 = [12, 2], c3 = [1, 4] están dados en forma cartesiana primero usando Strategy1 y luego usando Strategy2. (1.5 puntos)

Nota: En Ruby el módulo Math provee de todas las funciones necesarias para los cálculos (Math.sqrt, Math.sin, Math.cos, Math.atan).

Pregunta 4:

Digamos que se ha diseñado una pequeña aplicación web orientado a las Pymes que comprende 4 grandes subsistemas:

- Compras a Proveedores (seguimiento de ordenes de compra, pagos, cuentas por pagar)
- Ventas a Clientes (seguimiento de ordenes de venta, facturación, cuentas por cobrar)
- Contabilidad (ingreso de movimientos, emisión de libros diarios, balances)
- Control de Existencias (insumos y productos terminados)

Se discuten 3 posibles arquitecturas:

I. Cada componente se encapsula en un módulo que contiene las clases que implementan el componente.
II. Usar una arquitectura orientada a servicios (SOA).
III. Usar una arquitectura de microservicios.

a) Explique las principales diferencias entre esos 3 enfoques. (1.5 puntos)
b) De tres ejemplos de los servicios que podrían ser implementados en la opción II. (1.5 puntos)
c) De tres ejemplos de los servicios que podrían ser implementados en la opción III. (1.5 puntos)
d) Compare las opciones II y III en términos de ventajas y desventajas. (1.5 puntos)

Pauta Pregunta 1:

En la siguiente solución propuesta, se utiliza como ejemplo una clase Employee que sigue el rol de Subject, y una clase Payroll que sigue el rol de Observer. Como parte del enunciado se dice que el Subject debe hacer una operación de interés. En este ejemplo en particular, esta operación de interés es la actualización de valores de sueldo e impuestos, pero es válido poner cualquier cosa.

Más allá de la implementación de NotificationCenter, lo más importante es notar que para obtener referencias a NotificationCenter es fundamental implementar el patrón Singleton, dado que no tiene sentido la existencia de múltiples instancias de esta clase y en el enunciado se expresa que esta clase es única. Tampoco se puede pasar el NotificationCenter como parámetro a Subject y Observer dado que eso implicaría guardarlo en un atributo y ese es justamente el detalle que queremos resolver.

```ruby
module Exercise1

  class NotificationCenter
    @@instance = nil

    def self.instance
      @@instance ||= new
    end

    def initialize
      @listeners = {}
    end

    def register_listener(event_id, listener)
      if @listeners.key? event_id
        @listeners[event_id] << listener
      else
        @listeners[event_id] = [listener]
      end
    end

    def delete_listener(event_id, listener)
      @listeners[event_id].delete(listener) if @listeners.key?(event_id)
    end

    def notify(event_id, *arguments)
      @listeners[event_id].each do |listener|
        listener.update(event_id, *arguments)
      end
    end

    private_class_method :new
  end
```

```ruby
  class Employee
    attr_accessor :salary, :taxes

    def initialize(salary, taxes)
      @salary = salary
      @taxes = taxes
    end

    def update_salary_and_taxes(salary, taxes)
      @salary = salary
      @taxes = taxes

      notification_center = NotificationCenter.instance
      notification_center.notify(:my_event, salary, taxes)
    end
  end

  class Payroll

    def initialize
      notification_center = NotificationCenter.instance
      notification_center.register_listener(:my_event, self)
    end

    def update(event_id, salary, taxes)
      puts "new salary is #{salary}"
      puts "new taxes are #{taxes}"
    end
  end


  subject = Employee.new(1_000_000, 100_000)
  observer = Payroll.new
  subject.update_salary_and_taxes(1_100_000, 120_000)

end
```

Nota de Pauta: Los ejemplos de Subject y Observer pueden ser definidos por el alumno, no tiene por qué ser un Employee y Payroll.

Cada uno de los siguientes ítems tiene 1 punto:

- Método “register_listener” de NotificationCenter
- Método “delete_listener” de NotificationCenter
- Método “notify” de NotificationCenter
- Método “update” de Observer
- Script de ejemplo que ilustra el flujo
- Implementación del patrón Singleton en NotificationCenter

En estricto rigor, register_listener debiese chequear que el listener que se registre sea distinto de nil y no se encuentre previamente registrado, pero no hace falta que el alumno haya considerado lo anterior.

Si el alumno no logra identificar la necesidad de implementar el patrón Singleton en NotificationCenter y decide guardar una instancia de dicha clase como atributo del Subject y Observer, asignar 0 puntos para ese ítem. Si por otro lado, el alumno crea NotificationCenter distintos para Subject y Observer, aparte de asignar 0 puntos, descontar 1 punto sobre el total de esta pregunta, dado que la implementación anterior no funcionaría.

Ser criterioso con errores de sintaxis. En general, uno o dos errores menores (como escribir mal el nombre de una función de ruby (ej: key en vez de key?), u omitir un tag end) pueden omitirse sin necesidad de aplicar descuento. Solo si son muy numerosos o se trata de errores flagrantes descontar puntaje.

Pauta Pregunta 2:

En la siguiente solución propuesta, se propone modelar como una clase el Robot, y como clases distintas cada uno de los 3 comandos que uno puede ejecutar. Esto último no es estrictamente necesario, se puede usar una única clase Comando donde un atributo distinga el tipo de acción a realizar. Independiente de lo anterior, es necesario implementar el método *execute* y *undo* en los comandos. Ambos pueden tener la lógica a ejecutar en su misma definición, o bien, ejecutar un Block, Proc o Lambda que se guarde como atributo del Comando.

Para la parte del script, se considera que el string "forward" corresponde al comando Forward, el string "left" corresponde al comando "Turn Left" y el string "right" corresponde al comando "Turn Right". Exactamente qué string utilizar para cada comando depende del alumno (es perfectamente válido usar números por ejemplo, aunque no sea muy intuitivo).

```ruby
module Exercise2
  class Robot
    # Van a haber 4 orientaciones posibles que representaremos con
símbolos: :up, :right, :down, :left
    attr_reader :position_x, :position_y, :orientation

    def initialize(position_x, position_y, orientation)
      @position_x = position_x
      @position_y = position_y
      @orientation = orientation
    end

    # Se asume que la coordenada (0,0) corresponde a la esquina inferior izquierda
    def move_forward
      case @orientation
      when :up
        @position_y += 1
      when :right
        @position_x += 1
      when :down
        @position_y -= 1
      when :left
        @position_x -= 1
      end
    end

    def turn_right
      case @orientation
      when :up
        @orientation = :right
      when :right
        @orientation = :down
      when :down
        @orientation = :left
      when :left
        @orientation = :up
      end
    end
```

```ruby
    def turn_left
      case @orientation
      when :up
        @orientation = :left
      when :right
        @orientation = :up
      when :down
        @orientation = :right
      when :left
        @orientation = :down
      end
    end

    def move_backwards
      case @orientation
      when :up
        @position_y -= 1
      when :right
        @position_x -= 1
      when :down
        @position_y += 1
      when :left
        @position_x += 1
      end
    end

    def print_position
      puts "Current position is (#{@position_x},#{@position_y}) and orientation is
#{@orientation}."
    end
  end

  class CommandForward
    def initialize(target)
      @target = target
    end

    def execute
      @target.move_forward
    end

    def undo
      @target.move_backwards
    end
  end

  class CommandTurnRight
    def initialize(target)
      @target = target
    end

    def execute
      @target.turn_right
    end

    def undo
      @target.turn_left
    end
  end
```

```ruby
  class CommandTurnLeft
    def initialize(target)
      @target = target
    end

    def execute
      @target.turn_left
    end

    def undo
      @target.turn_right
    end
  end


  commands = []
  current_step = 0
  robot = Robot.new(0, 0, :up)
  loop do
    command = gets.chomp
    break if command == "finish"

    commands << CommandForward.new(robot) if command == "forward"
    commands << CommandTurnLeft.new(robot) if command == "left"
    commands << CommandTurnRight.new(robot) if command == "right"
  end

  loop do
    command = gets.chomp

    if command == "execute"
      (current_step...commands.length).each do |step|
        commands[step].execute
      end
      robot.print_position
      break
    elsif command == "step"
      if current_step < commands.length
        commands[current_step].execute
        current_step += 1
        robot.print_position
      end
    elsif command == "undo"
      if current_step > 0
        current_step -= 1
        commands[current_step].undo
        robot.print_position
      end
    end
  end
end
```

Nota de Pauta: Se debe evaluar aparte la construcción de la o las clases Command y el script. Dado que se exige el uso del patrón Command para la realización de este ejercicio, si el alumno no modela los comandos como objetos, evaluar toda la pregunta con 0 puntos. Modelar el Robot como una clase aparte es muy aconsejable, pero al no estar explícito en el enunciado, es opcional.

El desglose de puntaje en la evaluación es el siguiente:

- Correctamente implementada lógica de los 3 comandos Forward, Move Left y Move Right: 0.5 puntos cada uno
- Clase o clases Command bien definidas: 1.5 puntos
- Script permite recopilar comandos hasta encontrar el string “finish”: 1 punto
- Comando Execute: 0.5 puntos
- Comandos Step y Undo: 0.75 puntos cada uno
    - Verificar condiciones de borde para ambos. Para ambos, descontar 0.3 puntos en caso de no manejo o mal manejo de condiciones de borde.

Ser criterioso con errores de sintaxis. En general, uno o dos errores menores (como escribir mal el nombre de una función de ruby (ej: get en vez de gets, olvidar el chomp), u omitir un tag end) pueden omitirse sin necesidad de aplicar descuento. Solo si son muy numerosos o se trata de errores flagrantes descontar puntaje.

Pauta Pregunta 3

a)

```ruby
class ComplexArith
    attr_accessor :strategy

    def initialize(strategy)
      @strategy = strategy
    end

    def strategy=(strategy)
      @strategy = strategy
    end

    def add(c1, c2)
      c3 = @strategy.add(c1, c2)
    end

    def prod(c1, c2)
      c3 = @strategy.prod(c1, c2)
    end

  end
```

b)

```ruby
 class Strategy
    def add(c1, c2)
      raise NotImplementedError, "abstract class"
    end

    def prod(c1, c2)
      raise NotImplementedError, "abstract class"
    end
  end


  class Strategy1 < Strategy
    def add(c1, c2)
      c3 = []
      c3[0], c3[1] = c1[0] * c2[0], c1[1] + c2[1]
    end

    def prod(c1, c2)
      c3 = []
      c3[0], c3[1] = c1[0] * c2[0] - c1[1] * c2[1], c1[0] * c2[1] + c2[0] * c1[1]
    end
  end
```

```ruby
class Strategy2 < Strategy
  def polar(c)
    cp = []
    cp[0], cp[1] = Math.sqrt(c[0]* c[0]+ c[1]* c[1]), Math.atan(c[1]/c[0])
  end

  def cartesiana(cp)
    c = []
    c[0], c[1] = cp[0] * Math.cos(cp[1]), cp[0] * Math.sin(cp[1])
  end

  def add(c1, c2)
    c3 = []
    c1p = polar(c1)
    c2p = polar(c2)
    c3[0], c3[1] = c1[0] * Math.cos(c1[1]) + c2[0] * Math.cos(c1[2]),
                   c1[0] * Math.sin(c1[1]) + c2[0] * Math.sin(c1[2])
  end

  def prod(c1, c2)
    c3p = []
    c1p = polar(c1)
    c2p = polar(c2)
    c3p[0], c3p[1] = c1p[0] * c2p[0], c1p[1] + c2p[1]
    cartesiana(c3)
  end
end
```

Nota de Pauta: Para la transformación de coordenadas cartesianas a polares, es aceptable que el alumno asuma como input solo valores complejos con parte real positiva distinta de 0 (para la cual la implementación que se ofrece en esta pauta es suficiente, el caso general es más complejo e implica considerar varios casos borde).

c)

```ruby
c1 = [4.0, 7.0]
c2 = [12.0, 2.0]
c3 = [1.0, 4.0]

arithmetic = ComplexArith.new(Strategy1.new)
result1 = arithmetic.add(arithmetic.prod(c1, c2), c3)

arithmetic.strategy = Strategy2.new
result2 = arithmetic.add(arithmetic.prod(c1, c2), c3)
```

Nota de Pauta: Ser criterioso con errores de sintaxis. En general, uno o dos errores menores (como escribir mal el nombre de una función de ruby u omitir un tag end) pueden omitirse sin necesidad de aplicar descuento. Solo si son muy numerosos o se trata de errores flagrantes descontar puntaje.

Pauta Pregunta 4

a) El enfoque I produce una sola unidad ejecutable que contiene todas las componentes. Si se quiere cambiar alguna componente habría que reintegrar y compilar todos.
En los enfoques II y III las componentes al ser servicios están débilmente acopladas. Esto significa que pueden reemplazarse incluso sin tener que detener la ejecución de programa. Asimismo, estos servicios quedan disponibles para ser utilizados por otros programas.

b) En la opción II los servicios son relativamente grandes y corresponden con los procesos de negocio. Ejemplos son:

- proceso de ingreso de ordenes de compra
- proceso de facturación
- proceso de ingreso de una venta

c) En la opción III los servicios son mas pequeños y probablemente se requerirán varios de ellos para poder implementar un proceso de negocio. Ejemplos son:

- servicio de autentificación
- consulta de existencia de un producto
- consulta de status de un cliente

d) La opción II es la mas antigua y mas clásica. La ventaja es que hay un mapeo casi directo de servicios con procesos de negocio. La desventaja principal es que requieren de middleware de orquestación y los servicios, al ser mas específicos son mas difíciles de reutilizar. Otra desventaja es que generalmente comparten bases de datos y se hace más difícil el desarrollo y despliegue independiente. Como los servicios son relativamente complejos el equipo de desarrollo no puede ser muy pequeño.

Desventajas de III respecto a II son por ejemplo el tener que mantener un numero mucho mas elevado de servicios lo que puede requerir un registro. A veces es necesario construir una fachada (API gateway) para presentar a los programas clientes una interfaz mas sencilla.