Interrogación 2: Ingeniería de Software
Semestre 1- 2023
Pauta

P1 (3 pts). Indique qué afirmaciones se consideraron como buenas prácticas de diseño en clases.

I. Alto acoplamiento y baja cohesión.
II. Evitar código duplicado.
III. Alta cohesión y bajo acoplamiento.
IV. Maximizar el código duplicado.
V. Facilitar el mantenimiento y la extensibilidad.

Alternativas:

A. Solo I, II y V son consideradas como buenas prácticas.
B. Solo IV y III son consideradas como buenas prácticas.
C. Solo II, III y V son consideradas como buenas prácticas.
D. Solo III y V son consideradas como buenas prácticas.

P2 (3 pts). Conecte las siguientes descripciones con las arquitecturas comunes correspondientes.

I. No es necesario una capa intermedia middleware (API Gateway) entre los consumidores de los servicios y los servicios mismos.
II. Organiza el sistema en capas, tal que una capa de servicios a la capa de encima.
III. Es necesario un bus de servicios (enterprise service bus) que permite conectar o comunicar los servicios.
IV. Arquitectura popular en los 80's y 90's, donde la responsabilidad se divide entre el cliente y el servidor o en más de dos.

Alternativas:

A. Arquitectura de microservicios = I, Arquitectura clásica de servicios = III, Arquitectura Cliente-Servidor o Multi-Tier = IV, Arquitectura por capas = II.
B. Arquitectura de microservicios = I, Arquitectura clásica de servicios = IV, Arquitectura Cliente-Servidor o Multi-Tier = III, Arquitectura por capas = II.
C. Arquitectura de microservicios = III, Arquitectura clasica de servicios = I, Arquitectura Cliente-Servidor o Multi-Tier = IV, Arquitectura por capas = II.
D. Arquitectura de microservicios = I, Arquitectura clásica de servicios = IV, Arquitectura Cliente-Servidor o Multi-Tier = III, Arquitectura por capas = II.

P3 (3 pts). Indique qué afirmaciones representan ventajas de las arquitecturas de microservicios.

I. Para escalar es necesario hacer copias de la aplicación completa.
II. Permite desarrollar y realizar deploy en forma independiente cada servicio.
III. El versionamiento es menos complejo y son menos complejos de mantener.
IV. Es posible que cada componente tenga su propia tecnología y su propia base de datos.

Alternativas:

A. Solo I, II y IV representan ventajas.
B. Solo II representa ventajas.
C. Solo II y IV representan ventajas.
D. Solo II, III y IV representan ventajas.

P4 (3 pts). Indique qué afirmaciones son verdaderas sobre los aspectos a considerar para un buen diseño.

I. Una buena práctica es evitar código duplicado, ya que es necesario modificar un clon, entonces es necesario realizar la misma modificación en los demás clones.
II. Los elementos dentro de cada módulo deben estar lo menos relacionados posibles.
III. Un programa facilita la extensibilidad si permite extender (agregar) nuevas funcionalidades sin tener que modificar el código actual.
IV. Los módulos deben estar altamente relacionados entre sí.

Alternativas:

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

P5 (3 pts). Indique cual (es) de las siguientes afirmaciones sobre diagramas de secuencia son verdaderas:

I. Los objetos se representan con líneas horizontales llamadas líneas de vida.
II. Los mensajes de retorno se muestran con una flecha discontinua (punteada).
III. Los mensajes se colocan en el orden cronológico en el que se envían.
IV. Los mensajes de llamada a métodos se representan con una flecha que apunta hacia la línea de vida del objeto envía el mensaje.

Alternativas:

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

P6 (5 pts). Considere el siguiente código en Ruby:

```ruby
class A
   def bar
     self.foo + 7
   end
   def foo
     2
   end
end
```

```ruby
class B < A
   def bar
    super + 5
  end
end
```

```ruby
class C < B
   def foo
    super + 10
  end
end
```

Dado el código anterior, indique el output correspondiente que aparece en consola al ejecutar el siguiente código en Ruby:

```ruby
puts A.new.bar
puts B.new.bar
puts C.new.bar
```

Respuesta. 9 (1 punto)
14 (1.5 puntos)
24 (2.5 puntos)

P7 (5 pts). Considere el siguiente código en Ruby:

```ruby
class A
   def bar
     80
   end
   def foo
     10
   end
end
```

```ruby
class B < A
   def foo
    super + bar - 5
  end
end
```

Dado el código anterior, indique el output correspondiente que aparece en consola al ejecutar el siguiente código en Ruby:

```ruby
b = B.new
puts b.foo
```

Respuesta. 85

P8 (5 pts). Un estudiante crea un clase Account y quiere restringir a que las personas solo puedan crear dos objetos de esta clase. Basado en el **patrón singleton** implementó el siguiente código. El estudiante piensa que su solución es novedosa e incluso piensa publicar un artículo científico al respecto llamando a su código el patrón *Binbleton*.

```
class BDAccount
   def initialize(user,pass)
       @username = user
       @password = pass
   end
   def username
       return @username
   end
   def password
       return @password
   end
   def self.dbaccount1()
       if @@account1 == nil then
           @@account1 = DBAccount.new('admin','admin')
       end
       return @@account1
   end
   def self.dbaccount2()
       if @@account2 == nil then
           @@account2 = DBAccount.new('admin2','admin2')
       end
       return @@account2
   end
end
```

¿Usted cree que el código anterior restringe sólo la creación de dos objetos como máximo? Explique su respuesta.

Respuesta. En el ejemplo anterior faltó cambiar a privado el método de clase ***:new***.

private_class_method :new

Si el **new** no es privado entonces es posible crear muchos objetos de la clase a través de él.

Las respuestas que dé a entender que es posible crear muchos objetos porque falto poner el new en privado son válidas. Cualquier otra respuesta es incorrecta.

P9 (20 pts). Considere el siguiente segmento de código que fue extraído de la aplicación WeChat (la versión china de whatsapp).

```
class Contact
   …
   def receiveMessage(msg)
       puts "you receive this message #{msg}"
       …
   end
   …
end
class MessageSender
   …
   def initialize(cnt)
       @contact = cnt
       …
```

```
    end
    def sendMessage(msg)
        @contact.receiveMessage(msg)
        …
    end
    …
end
secretContact = Contact.new
sender = MessageSender.new(secretContact)
sender.sendMessage('saludos desde marte')
```

- **(10 pts)** Implementar el **patrón proxy** de forma tal que el objeto proxy imprima en consola "Potencial Capitalista" cuando un mensaje enviado tenga la palabra *Trump*.
- **(10 pts)** Implementar el **patrón adapter** para que cuando alguien envíe la palabra "F", adapte el mensaje enviando el texto "@#@$" en su lugar.

Note que para implementar ambos patrones, no debe modificar ni la clase mail MailSender, ni la clase Contact.

Respuesta. Para cada inciso escribir la clase Proxy y Adapter era suficiente.

Proxy.

```
class ProxyContact
    def initialize(originalContact)
       @originalContact = originalContact
    end

    def receiveMessage(msg)
       if msg.include? "Trump"
           puts "Potencial Capitalista"
       end
       @originalContact.receiveMessage(msg)
    end
end
```

La parte naranja puede variar, la idea es que el estudiante escriba un código que dé a entender que busca la palabra Trump en el mensaje. Es posible que haya usado otro método o función distinta. No importa si el estudiante no recordó el nombre exacto de la función.

Adapter.

```
class AdapterContact
    def initialize(originalContact)
       @originalContact = originalContact
    end

    def receiveMessage(msg)
       newMsg = msg.replace("F","@#@$")
       @originalContact.receiveMessage(newMsg)
    end
end
```

La parte naranja puede variar, la idea es que el estudiante escriba un código que dé a entender que quiere reemplazar la palabra F por "@#@$". Es posible que haya usado otro método o función distinta. No importa si el estudiante no recordó el nombre exacto de la función. También es válido si el estudiante busca mensajes iguales a "F" (msg=="F").

P10 (18 pts). Considere el siguiente código de un estudiante de Programación Avanzada (kokito) que está iniciando en programación orientada a objetos. El quiere simular dos tipos de cobro: cobro en efectivo y otro cobro con tarjeta. Kokito no sabe mucho de diseño orientado a objetos, por lo que su código actual tiene varias falencias, por ejemplo, tiene mucho código duplicado. Su tarea es mejorar el código de kokito utilizando el **patrón template method**. Debe escribir una versión mejorada del código de kokito de forma tal que no exista código duplicado. *Escriba la solución en una hoja separada indicando su número de alumno y el número de pregunta.*

Código actual de Kokito

```
class CobroEfectivo
   def cobrar(monto)
       puts "Su cuenta es de #{monto.to_s}"
       puts "Mas 10 Bs de comision"
       puts "En total debe #{(monto + 10).to_s}"
       puts "Ingrese el monto a depositar:"
       pagoCliente = gets.chomp.to_i
       if pagoCliente < monto
           puts "Monto insuficiente"
       else
           puts "Su cambio es: #{(pagoCliente - (monto + 10)).to_s}"
           puts "Transacción exitosa"
       end
       puts "Gracias"
   end
end
class CobroTarjeta
   def cobrar(monto)
       puts "Su cuenta es de #{monto.to_s}"
       puts "Mas 1 Bs de comision"
       puts "En total debe #{(monto + 1).to_s}"
       puts "Ingrese el número de tarjeta:"
       numeroTarjeta = gets.chomp.to_i
       if numeroTarjeta > 1000
           puts "Se ha descontado un total de #{(monto + 1).to_s}"
           puts "De su cuenta ***#{(numeroTarjeta % 1000).to_s}"
           puts "Transacción exitosa"
       else
           puts "Tarjeta no válida, consulte a su banco"
       end
       puts "Gracias"
   end
end
cajero1 = CobroEfectivo.new
cajero2 = CobroTarjeta.new
cajero1.cobrar(100)
cajero2.cobrar(50)
```

Posible Respuesta.

```
class Cobro
  def initialize(comision)
    @comision = comision
  end

  def cobrar(monto)
    puts "Su cuenta es de #{monto}"
    puts "Mas #{@comision} Bs de comision"
    puts "En total debe #{monto + @comision}"
    realizarTransaccion(monto)
    gracias
  end

  def realizarTransaccion(monto)
    raise NotImplementedError
  end

  def gracias
    puts 'Gracias'
  end

  def trasaccionExitosa
    puts 'Transacción exitosa'
  end
end
```

```
class CobroEfectivo < Cobro
  def initialize
    super(10)
  end
  def realizarTransaccion(monto)
    puts 'Ingrese el monto a depositar:'
    pagoCliente = gets.chomp.to_i
    if pagoCliente < monto
      puts 'Monto insuficiente'
    else
      puts "Su cambio es: #{pagoCliente - (monto + 10)}"
      trasaccionExitosa
    end
  end
end
```

```
class CobroTarjeta < Cobro
  def initialize
    super(1)
  end
  def realizarTransaccion(monto)
    puts 'Ingrese el número de tarjeta:'
    numeroTarjeta = gets.chomp.to_i
    if numeroTarjeta > 1000
      puts "Se ha descontado un total de #{monto + 1}"
      puts "De su cuenta ***#{numeroTarjeta % 1000}"
      trasaccionExitosa
    else
```

```
      puts 'Tarjeta no válida, consulte a su banco'
    end
  end
end
cajero1 = CobroEfectivo.new
cajero2 = CobroTarjeta.new
cajero1.cobrar(100)
cajero2.cobrar(50)
```

Pueden haber muchas soluciones pero una solución válida debe tener lo siguiente:

- (3 puntos) Una clase padre de la cual heredan las dos clases.
- (12 puntos) Debe existir un método en la clase padre (como el marcado con naranja) que llame a un método (como el amarillo) que es sobre-escrito por las clases hijas. Esto es indispensable ya que este método por definición sería un método plantilla.
- (3 puntos) Finalmente debe eliminar un poco de código duplicado, porque eso pide el enunciado. Este puntaje solo se agrega, si es que se cumple los dos puntos anteriores. No es necesario que elimine el 100% del código duplicado aunque es deseable.

P11 (17 pts). Te contrataron para refactorizar el siguiente código en Ruby que permite que un cliente pueda agregar items a su carrito de compras y pagar los *ítems* seleccionados por tarjeta de crédito, débito y Paypal. Note que si se desea agregar una nueva forma de pago se debe modificar el método *payBy(modeOfPay)*, por lo cual el programa actual no facilita la extensibilidad. Tu misión es refactorizar el código aplicando el **patrón Strategy** y facilitando la extensibilidad. *Escriba la solución en una hoja separada indicando su número de alumno y el número de pregunta.*

```
class Item
   attr_reader :name, :price
   def initialize(name, price)
     @name = name
     @price = price
   end
   def print
     puts "#@name costs
#@price"
   end
   def cost
     @price
   end
end
```

```
class ShoppingCart
   def initialize
     @items = []
   end
   def add(item)
     @items.push(item)
   end
   def payBy(modeOfPay)
     amount = 0
     puts "Items:"
    @items.each do |item|
      amount += item.cost
      item.print
    end

    if modeOfPay == "Credit"
       puts "Paying #{amount} with credit card"
    elsif modeOfPay == "Paypal"
      puts "Paying #{amount} with Paypal"
    elsif modeOfPay == "Debit"
      puts "Paying #{amount} with debit card"
    end
   end
 end
```

Posible Respuesta.

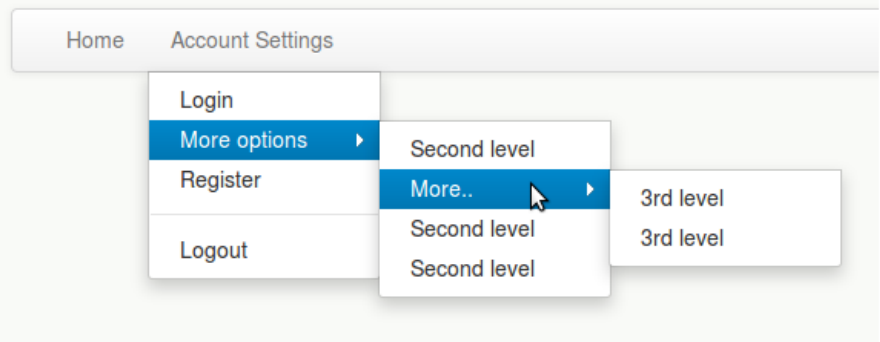
```
class Payment
    def charge(amount)
      raise NotImplementedError
    end
end
class Credit < Payment
  def charge(amount)
      puts "Paying #{amount} with credit card"
  end
end
class Paypal < Payment
  def charge(amount)
      puts "Paying #{amount} with Paypal"
  end
end
class Debit < Payment
  def charge(amount)
      puts "Paying #{amount} with debit card"
  end
end

class ShoppingCart
   def initialize
     @items = []
   end
   def add(item)
     @items.push(item)
   end
   def payBy(modeOfPay)
    modeOfPay.charge(totalCost)
   end
  def printItems
    puts "Items:"
    @items.each do |item|
      item.print
    end
  end
  def totalCost
    amount = 0
    @items.each do |item|
      amount += item.cost
    end
    return amount
  end
 end
```

Pueden haber muchas soluciones pero una solución válida debe tener lo siguiente:

- (12 puntos) Una jerarquía de clases (código en amarillo), la clase padre debe tener al menos un método que es sobreescrito por las clases hijas. El nombre del método, y argumentos puede variar dependiendo de la abstracción del estudiante.
- (5 puntos) La clase ShoppingCart debe llamar al método del punto anterior, como por ejemplo, el código en naranja.

P12 (15 pts). Una empresa está trabajando en crear una herramienta para crear, editar, y mostrar slides. Así, es

muy parecida a power point. Como usted recién fue contratado, el project manager le asignó la tarea de crear un modelo que permita agregar menús y submenús a la barra de opciones de la herramienta.

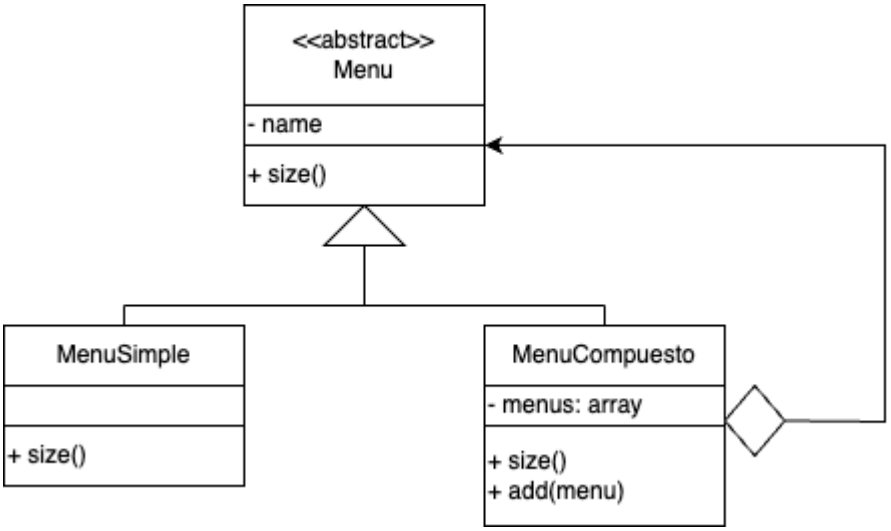


Existen dos tipos de componentes: menu-compuesto y menu-simple. Los menús compuestos pueden contener sub-menus que a su vez pueden ser compuestos o simples, como se muestra en la figura. En esta pregunta usted debe dibujar un diagrama de clases, lo mas detallado posible, para la ilustrar la solución propuesta utilizando el **patrón composite**.

Dibuje su diagrama de clases en una hoja separada, indicando su número de alumno y el número de pregunta.

Cómo es su primera tarea, un desarrollador señor (Peter Parker), le dio un caso de prueba para ayudarlo en su diagrama:

```
menu = Menu.new("file")
sub_menu1 = Menu.new("new...")
sub_menu11 = Menu.new("ruby ...")
sub_menu11.add(MenuItem.new("rb file ..."))
sub_menu11.add(MenuItem.new("rake file ..."))
sub_menu1.add(sub_menu11)
menu.add(sub_menu1)
sub_menu2 = Menu.new("open recent ...")
sub_menu2.add(MenuItem.new("interrogacion1 file"))
sub_menu2.add(MenuItem.new("interrogacion2 file"))
sub_menu2.add(MenuItem.new("interrogacion2 recuperativa"))
menu.add(sub_menu2)

puts "#{menu.size}"
puts "#{menu1.size}"
puts "#{menu2.size}"
```



Si bien pueden existir más o menos detalles lo que no debiera faltar es:

- (5 puntos) La jerarquía de clases, clase padre, dos subclases con sus nombres correspondientes y el triángulo en el lugar adecuado.
- (5 puntos) La linea que conecta de MenuCompuesto a la clase padre, con un rombo en el lugar correcto.
- (5 puntos). El método size en las 3 clases, el método add en la clase MenuCompuesto (el nombre puede variar) y el atributo name en la clase padre. En realidad el nombre del atributo puede variar, por ejemplo, nombre, name o cualquier otra variable que almacene el string que se manda como argumento en el constructor.