IIC 2143 Ingeniería de Software
Semestre 2021-2
Profesores M. Peralta y J. Navón

Solución Problema 1

Cada respuesta correcta tanto en si es verdadera o falsa y también la justificación vale 2 puntos. Si la justificación está totalmente errada la respuesta tiene 0 puntos. Si no es precisa pero se acercó la respuesta tiene 1 punto.

a) En el método third de la clase B se puede acceder al atributo c
`V - un método siempre tiene acceso a los atributos de su propia clase`

b) En el método fifth de la clase D se puede acceder al método second
`V - second es protegido por lo que puede accederse desde una subclase`

c) En el método fifth de la clase D se puede acceder al método third
`F - el método third es privado`

d) En el método fourth de la clase C se puede acceder al método third
`F - el método third es privado`

e) En el método fourth de la clase C se puede acceder al método first
`V - first tiene visibilidad de paquete y fourth pertenece al paquete Alfa`

f) En el método third de la clase B se puede acceder al método sixth
`V - sixth tiene visibilidad pública`

g) En el método fourth de la clase C se puede acceder al atributo d
`V - d ha sido declarado público (poco usual para atributos)`

h) En el método sixth de la clase E se puede acceder al método fifth
`V - fifth tiene visibilidad de paquete y sixth pertenece al paquete Beta`

i) En el método eight de la clase F se puede acceder al método seventh
`V - seventh es protegido y eight pertenece a una subclase de C`

j) En el método eight de la clase F se puede acceder al método fourth
`F - fourth es privado`

k) En el método sixth de la clase E se puede acceder al atributo d
`V - d es un atributo público`

l) En el método eight de la clase F se puede acceder al atributo g
`V - un método siempre puede acceder a los atributos de la clase`

Solución Problema 2

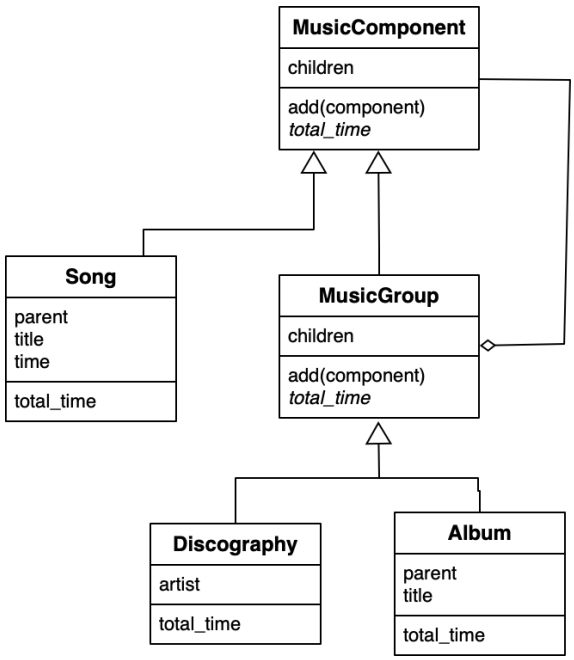
a) 10 puntos si el patrón Composite está bien implementado

Aplicación directa del patrón "composite" (ver ejemplo de clases). Lo que es novedoso aquí es que en lugar de tener unsolo objeto compuesto, tenemos de dos tipos (discografía y album). Desde el punto de vista del patrón los tratamos de igual forma.

La clase MusicComponent representa ya sea una canción (hoja en el coomposite) o un subárbol (MusicGroup) que contiene una referencia a la lista de sus hijos que son del tipo MusicComponent. Finalmente el MusicGroup lo especializamos en Discografía y Album porque son inicializados en forma distinta (artista o título respectivamente)

Así por ejemplo una discografía con dos álbums y 10 canciones en cada uno de ellos tendrá un objeto MusicGroup (discografía) que contiene dos objetos MusicGroup (Album), cada uno de los cuales contiene 10 objetos MusicComponent (Song

En la implementación el método tiempo_total de un MusicGroup suma el resultado de invocar tiempo_total sobre sus componentes o el valor de duración en el caso que sea solo una canción



b) Código Ruby

```
class MusicComponent
   def total_time
    raise('abstract method')
  end
end
```

```
class MusicGroup < MusicComponent
  def initialize
    @children = Array.new
  end
  def add_child(component)
    @children << component
   end
  def total_time
    total_time = 0
    @children.each {|item| total_time += item.total_time}
    total_time
  end
end

class Discography < MusicGroup
  def initialize(artist)
    super()
    @artist = artist
  end
end

class Album < MusicGroup
  attr_accessor :parent, :title
  def initialize(title)
    super()
    @title = title
  end
end

class Song < MusicGroup
  attr_accessor :parent, :title
  def initialize(title, time)
    @title = title
    @time = time
  end
  def total_time
    @time
  end
end
```

Código de prueba

```
beatles_discography = Discography.new('The Beatles')
rubber_soul = Album.new('Rubber Soul')
revolver = Album.new('Revolver')
rubber_soul.add_child Song.new('Norwegian Wood', 1.5)
rubber_soul.add_child Song.new('Nowhere Man', 2.0)
revolver.add_child Song.new('Eleanor Rigby', 1.5)
revolver.add_child Song.new('Tomorrow Never Knows', 2.5)
beatles_discography.add_child rubber_soul
beatles_discography.add_child revolver
puts rubber_soul.total_time
puts revolver.total_time
puts beatles_discography.total_time
```

Resultado

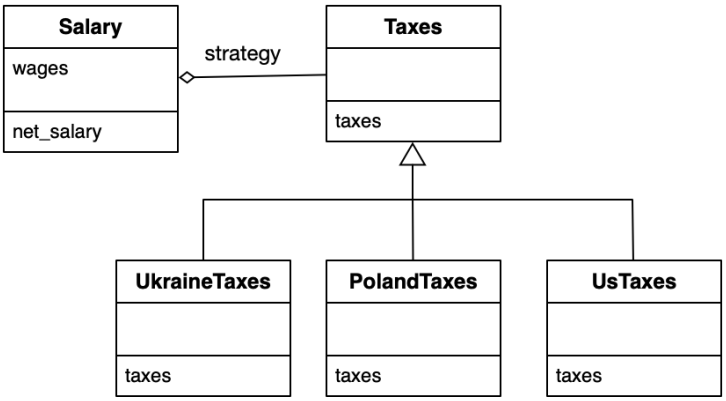
```
jnavon@Tatooine-2 rubyex % ruby P2.rb
3.5
4.0
7.5
```

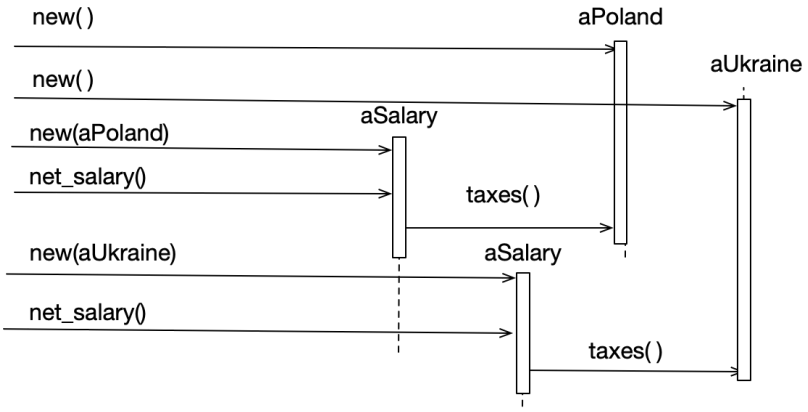
Solución Problema 3

a) Correcta implementación del patrón Estrategia y clases correctas 12 puntos
Código Ruby que ilustra el funcionamiento 4 puntos

```
class Salary
    def initialize (wages,country)
        @strategy = country
        @wages = wages
    end

    def net_salary
        @wages - @strategy.taxes(@wages)
    end
end

class Taxes
    def taxes(amount)
        raise 'abstract method'
    end
end

class UkraineTaxes < Taxes
    def taxes(amount)
        (amount * 0.05) + 313
    end
end

class PolandTaxes < Taxes
    def taxes(amount)
       amount * 0.3
     end
end

class UsTaxes < Taxes
    def taxes(amount)
        (amount * 0.2) + 100
    end
end

aPoland = PolandTaxes.new
aUkraine = UkraineTaxes.new

salPoland = Salary.new(1000, aPoland)
puts salPoland.net_salary

salUkraine = Salary.new(1000, aUkraine)
puts salUkraine.net_salary
```

b)





Notas de Corrección

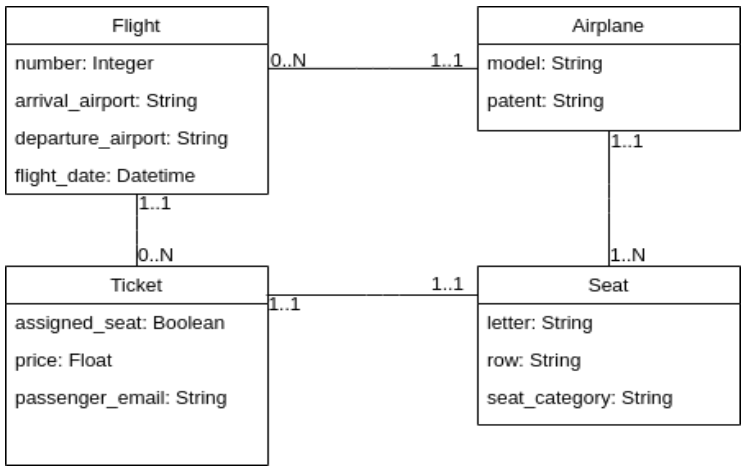
Diagrama de clases 4 puntos, diagrama de secuencia 6 puntos.

En el diagrama de clases podría aparecer la estrategia como atributo de Salary y no en forma explícita como una clase. Dado que en el enunciado aparecen 3 países deben aparecer las 3 clases una para cada país.

En el diagrama de secuencia deben participar los objetos Tax y las estrategias del códgo ejemplo. Deben aparecer los dos mensajes net_salary sobre los objetos Tax y los mensajes taxes sobre las dos estrategias.
Si los objetos todos "nacen" al mismo tiempo (comienzan arriba) es un error menor.

Solución Problema 4

a) 10 ptos
   i) (4 ptos): 1 pto por la existencia de cada clase correcta: Airplane, Flight, Ticket y Seat
   ii) (4 ptos): 1 pto por relación correcta entre clases:
      - entre Flight y Airplane es 1-1
      - entre Airplane y Seat es 1-N
      - entre Flight y Ticket es 1-N
      - entre Ticket y Seat es 1-1
   iii) (2 ptos): Por colocar todos los atributos mencionados.



b) 10 ptos:
   i) (0.5 pto): Existe la migración para crear Airplane.
   ii) (0.5 pto) Existe el modelo Airplane con asociaciones correctas.
   iii) (1 pto): Existe un modelo Flight con asociaciones correctas.
   iv) (1 pto): Existe una migración para crear Flight que contiene una columna para la foreing_key de Airplane.
   v) (1 pto): Existe un modelo Seat con asociaciones correctas.
   vi) (1 pto): Existe una migración para crear Seat que contiene una columna para la foreing_key de Airplane.
   vii) (1 pto): Existe un modelo Ticket con asociaciones correctas.
   viii) (1 pto): Existe una migración para crear Ticket que contiene una columna para la foreing_key de Flight.
   ix) (1 pto): En la migración para crear Ticket también existe una columna para la foreign_key de seat. Si se decidió resolver esa dependencia sólo guardando un identificador de seat (por ejemplo el string “A12”) también está correcto.
   x) (2 ptos): Están considerados todos los atributos de tipo correcto que se mencionaron en el enunciado.

migración de Airplane:

```
class CreateAirplanes < ActiveRecord::Migration[5.2]
 def change
   create_table :airplanes do |t|
     t.string :model
     t.string :patent

     t.timestamps
   end
 end
end
```

modelo de Airplane:

```
class Airplane < ApplicationRecord
 has_many :flights
 has_many :seats, dependent: :destroy
end
```

migración Flight:

```
class CreateFlights < ActiveRecord::Migration[5.2]
 def change
   create_table :flights do |t|
     t.integer :number
     t.string :arrival_airport
     t.string :departure_airport
     t.datetime :flight_date
     t.belongs_to :airplane, index: true

     t.timestamps
   end
 end
end
```

modelo de Flight

```
class Flight < ApplicationRecord
 belongs_to :airplane, optional: true
 has_many :tickets
end
```

migración de Seat

```
class CreateSeats < ActiveRecord::Migration[5.2]
 def change
   create_table :seats do |t|
     t.string :letter
     t.integer :row
     t.integer :seat_category
     t.belongs_to :airplane, index: true

     t.timestamps
   end
 end
end
```

modelo Seat

```
class Seat < ApplicationRecord
 enum seat_category: { priority: 0, emergency: 1, regular: 2 }
  belongs_to :airplane
 has_many :tickets
end
```

migración de Ticket

```
class CreateTickets < ActiveRecord::Migration[5.2]
 def change
   create_table :tickets do |t|
     t.boolean :assigned_seat
     t.float :price
     t.string :passenger_email
     t.belongs_to :seat, index: true
     t.belongs_to :flight, index: true

     t.timestamps
   end
 end
end
```

modelo Ticket

```
class Ticket < ApplicationRecord
 belongs_to :flight
 belongs_to :seat, optional: true
end
```

c) 10 ptos:
   i) (9 ptos): 3 puntos por cada método correcto.
   ii) (1 pto): base

i) Listar vuelos con información requerida:

```
def is_oversold? flight
 capacity = flight.airplane.seats.count
 tickets = flight.tickets.count
 return capacity < tickets
end

def list_flights
 flights_information = []
 flights = Flight.all.includes(:airplane)
 flights.each do |flight|
   flights_information.push({
     number: flight.number,
     arrival_airport: flight.arrival_airport,
     departure_airport: flight.departure_airport,
     date: flight.flight_date,
     airplane_model: flight.airplane.model,
     total_seats: flight.airplane.seats.count,
     oversold: is_oversold?(flight)
   })
 end
 return flights_information
end
```

ii) Lista de aviones disponibles con capacidad mayor de la de un vuelo *flight*

```
def available_planes flight
 airplanes = Airplane.all
 result = []
 airplanes.each do |airplane|
   if airplane.flights.count == 0 && airplane.seats.count >
flight.tickets.count
     result.push(airplane)
   end
 end
 return result
end
```

iii) Lista de e-mails de pasajeros a los que se debe notificar que su asiento cambió de clasificación:

```
def same_seat? (ticket, new_airplane)
 new_seat = new_airplane.seats.where(letter: ticket.seat.letter, row:
ticket.seat.row).first
 if (new_seat)
   return new_seat.seat_category == ticket.seat.seat_category
 end
 return false
end

def list_emails (flight, new_airplane)
 emails = []
 flight.tickets.each do |ticket|
   if(ticket.seat)
     same_seat = same_seat?(ticket, new_airplane)
     unless(same_seat)
       emails.push(ticket.passenger_email)
     end
   end
 end
 return emails
end
```