IIC 2143 Ingeniería de Software
Interrogación 2 - Semestre 2 /2020

Puedes consultar material de clases o libros, pero tu trabajo debe ser estrictamente individual. Recuerda que estás bajo el código de honor.

Todas las respuestas deben ser entregadas reunidas en forma de un solo archivo en el espacio asignado para la pregunta 1. Asegúrate de que la entrega fue exitosa porque no aceptaremos entregas tardías por ninguna razón.

El puntaje máximo de la prueba es de 100 puntos. Las preguntas no tienen el mismo puntaje. En cada pregunta se indica el puntaje máximo posible. El puntaje asignado no representa el tiempo necesario para resolverlo

Pregunta 1 (32 puntos)

La empresa *ComoNuevos* corresponde a una cadena de venta de automóviles usados que mantiene muchos lugares de venta de autos en las distintas ciudades de Chile. Cada uno de estos *dealer*s locales funciona en forma mas o menos independiente y maneja su propio sistema que llamaremos *SistemaLocal* que es operado por los vendedores de cada tienda de ventas de autos, pero si es necesario también pueden consultar sobre autos disponibles en otros lugares.

A nivel de gestión de *ComoNuevos* se requiere construir un sistema que mantenga la información de la totalidad de autos disponibles para la venta en las distintas ciudades del país, que pueda relacionarse con los sistemas locales de los *dealers* y también poder conectarse con el Registro Civil donde se mantiene la información de la vida de cada auto (esto es para comprobar la propiedad, si hay deudas, etc.)

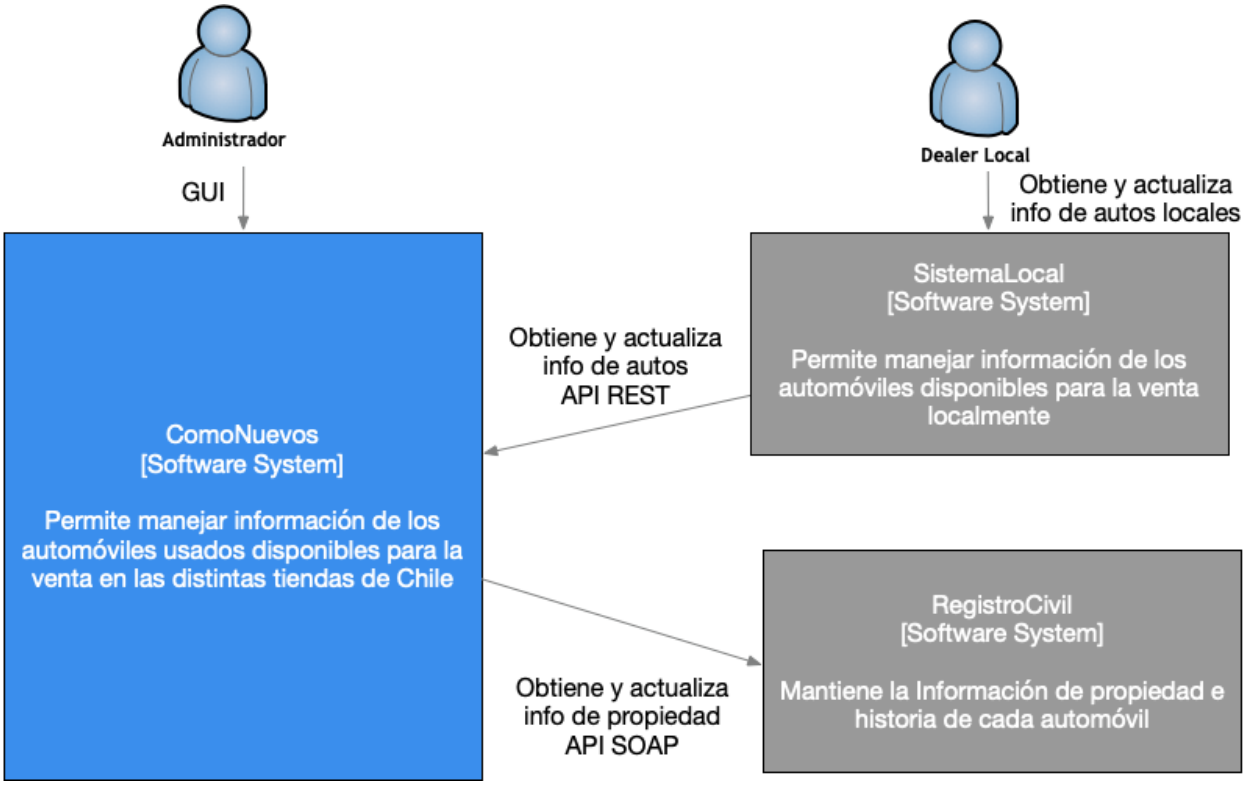
El sistema de *ComoNuevos* se visualiza con un *front-end* orientado a interactuar con el administrador, y un *backend* que se encarga de manejar la información de los autos usados disponibles. El administrador del sistema, aparte de poder acceder a la información de los autos, es responsable de manejar todos los datos de configuración que se almacenan en el sistema. El *backend* está además encargado de interactuar con el Registro Civil. Los distintos sistemas locales se conectarán también con el *backend* de *ComoNuevos* ya sea para extraer nueva información o para actualizar información cuando, por ejemplo, se vende un automóvil.

a) (14 puntos) Dibuja un diagrama de contexto para la arquitectura del sistema *ComoNuevos*. Identifica claramente usuarios y subsistemas externos.

b) (18 puntos) Dibuja un diagrama a nivel de contenedores incluyendo los principales contenedores y a que corresponden los principales intercambios de información entre ellos

Pauta Pregunta 1

a) 14 puntos



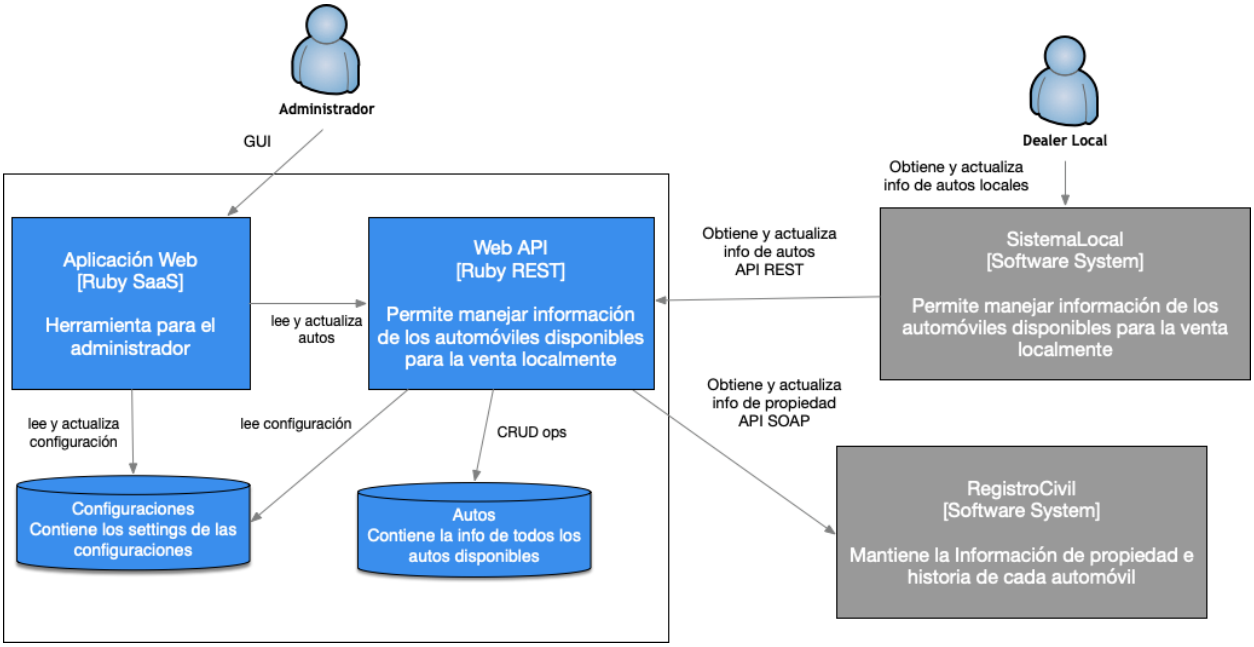
Nota de Pauta

Debe aparecer el rectángulo del sistema, los dos sistemas externos y el administrador a lo menos (puede que olviden al dealer local)

Debe aparecer una breve descripción además del nombre de cada uno los 3 rectángulos

Deben haber líneas con flujos de información entre el sistema con las entidades externas y con el administrador

b) 18 puntos



Nota de Pauta

Debe mantenerse consistente con el anterior pero ahora las líneas desde y hacia las entidades externas y al administrador salen desde contenedores específicos al interior del sistema

A lo menos debería haberse identificado un almacenamiento. Si usó uno solo para las configuraciones y los autos está bien pero deben haber líneas a ese almacenamiento desde la aplicación Web y la Web API

Pregunta 2 (30 puntos)

A continuación, se presenta un código en lenguaje Ruby en el que se han utilizado 3 patrones de diseño distintos de entre los que estudiamos en clases.

```ruby
class C1
  def m1
    raise 'abstract method'
  end
end

class C2
  def m2(f)
    raise 'abstract method'
  end
  def m3
    raise 'abstract method'
  end
end

class C3
  def m4
    raise 'abstract method'
  end
end

class C4 < C1
  @@x = C4.new
  def self.x
    return @@x
  end
  def m1
    return C5.new
  end
  private_class_method :new
end

class C5 < C2
  def initialize
    @s = ''
    @arr = []
  end
  def s=(new_s)
    @s = new_s
    puts @s
    m3()
  end
  def m2(f)
    @arr.push(f)
  end
  def m3
    for f in @arr do
      f.m4
    end
  end
end
```

```ruby
class C6 < C3
  def initialize(msg)
    @msg = msg
  end
  def m4
    puts @msg
  end
end

one = C6.new("software engineering")
two = C6.new("design partterns")

three = C4.x
four = three.m1

four.m2(one)
four.m2(two)

four.s = "I hate"
four.s = "I love"
```

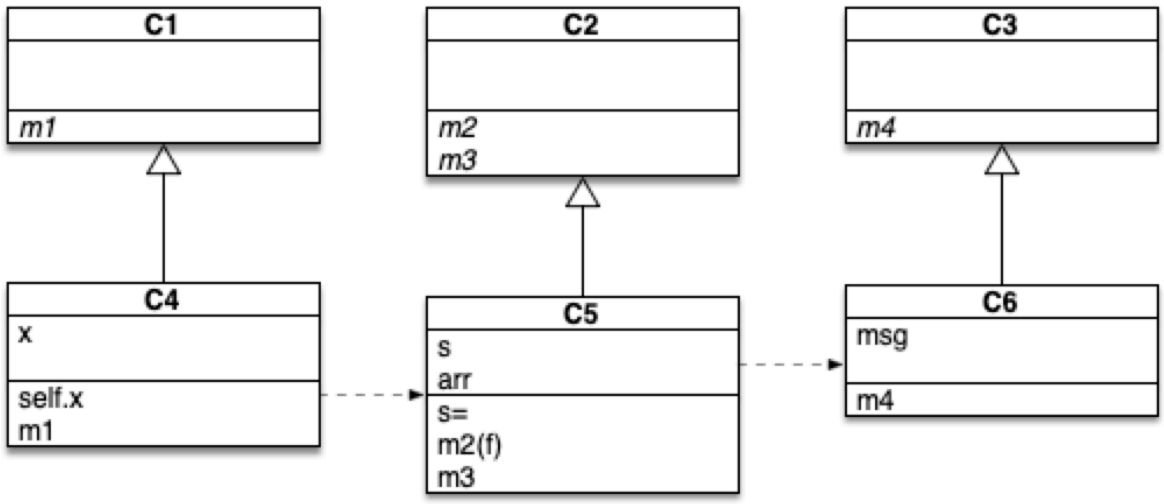
a) (5 puntos) Dibuja un diagrama de clases que corresponda a este código (incluye métodos y atributos)

b) (15 puntos) Identifica cada uno de los 3 patrones utilizados. Para cada uno de ellos indicar el nombre del patrón y los elementos que actúan en él. Por ejemplo, si se tratara del patrón *método plantilla,* decir cual es la clase abstracta, cuales las clases concretas y cual el método plantilla.

c) (10 puntos) ¿Cual es el output que se produce cuando el programa corre? Explícalo haciendo un diagrama de secuencia desde que inicia hasta que termina la ejecución del código

PAUTA PREGUNTA 2

a) 5 puntos



b) 15 puntos

Hay 3 patrones de diseño en el código: singleton, método fábrica y observer

Singleton:

Corresponde a la clase C4. La única instancia de esta clase está referenciada por el atributo estático @@x y es creado solo una vez al comienzo con C4.new. Vemos que además se ha declarado al método new como privado de modo que no se pueda invocar un C4.new desde fuera.

Método Fábrica;

Está en la misma clase C4 y corresponde al método m1. C4 es el creador concreto y C1 es el creador abstracto, m1 en ambas clases es el método fábrica (concreto y abstracto). En nuestro caso el creador concreto ha definido que ese método genere objetos de la clase C5

Observer

El Observer está implementado con las clases C5 (sujeto), la clase C3 (observer abstracto) y C6 (observer concreto). En este caso hay un solo tipo de obeserver concreto que es C6.

C5 provee de un método m2 para registrar nuevos observadores, un método m3 que sería el que notifica a todos los observadores de un cambio en el estado. La clase C5 incluye también un método s= que permite cambiar el string interno s.

Nota de Pauta

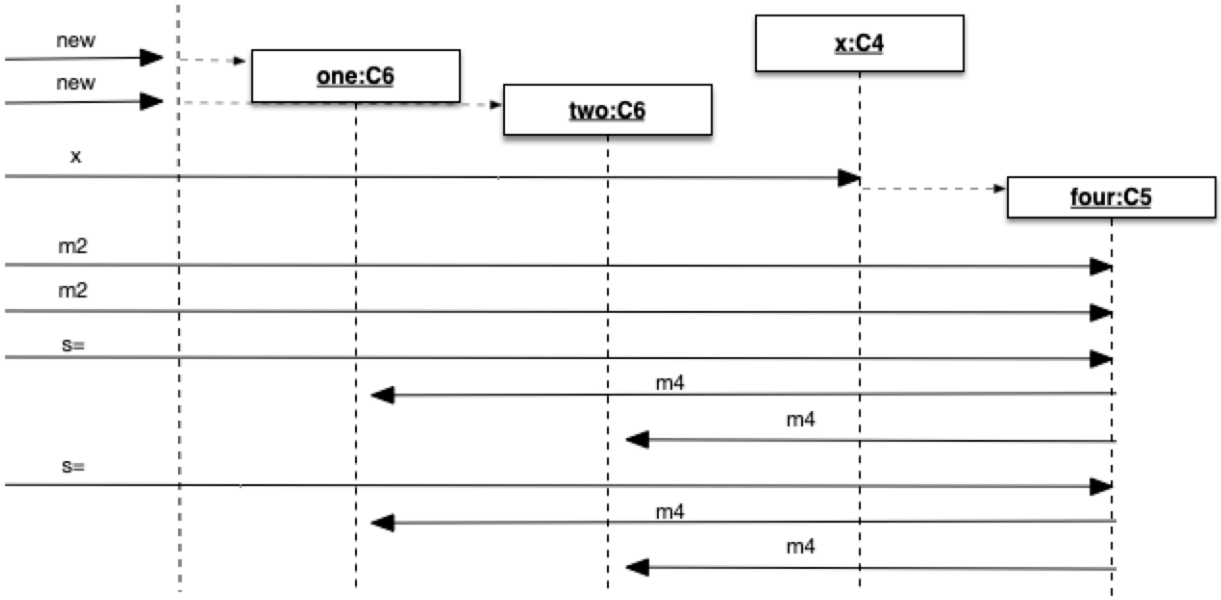
Cada patrón gana 5 puntos, 2 por identificarlos y 3 por explicar como aparece en el código dado.

c) 10 puntos

El output del programa es el siguiente:

```
I hate
software engineering
design partterns
I love
software engineering
design partterns
```

El diagrama de secuencia que muestra como se produce es el siguiente:



Pregunta 3 (26 puntos)

En un Marketplace de autos, distintas tiendas ofrecen sus vehículos, los cuales son vendidos por el sitio. Una vez por semana se ejecuta un proceso de rendición de las ventas, en el cual el sitio reporta las ventas realizadas a cada tienda y también genera un reporte semanal.

Actualmente el proceso se realiza por igual para cada tienda, sin embargo, algunas tiendas han pedido algunos cambios en el proceso. A este proceso lo llamaremos "proceso 2":

- Sólo se deben reportar las ventas por un valor superior a $1000. Esto es porque ventas menores se realizan como una forma de marcar una reserva de un vehículo y no deben considerarse ventas aún.
- Cuando un proceso se aborta, se debe crear un registro local con id remoto -1 y estado `:aborted`
- Tras finalizar el proceso se debe ejecutar una tarea automatizada que enviará el reporte por email y efectuará el pago de manera automática. Para ello basta con llamar un worker que efectúa el pago con:
  `PaymentWorker.perform_async(payment.id, @store.id)`

Por ultimo, el marketplace está en negociaciones con un reconocido sitio de ventas de autos, el cual tendrá un tercer proceso. Aún no es necesario implementarlo, pero es buena idea que lo consideres. Por lo pronto debes saber que el tercer proceso tendrá los mismos 6 pasos que el proceso actual.

El tipo de proceso puede obtenerse al llamar `@store.process_type` - este método retorna 1, 2 o 3, según el tipo de proceso.

a) (14 puntos) Aplicando uno de los patrones de diseño vistos en clases, cambia el código de manera de permitir que tanto el proceso 1 (el actual) como el proceso 2 (el nuevo) puedan ser utilizados. Tu solución también debe considerar que próximamente debe ser fácil hacer los cambios para el proceso 3.

b) (12 puntos) ¿Por qué elegiste ese patrón? Indica qué patrón usaste y describe brevemente 3 ventajas (o desventajas) que ves de aplicar el patrón versus el código original.

```ruby
## abstract_payment_processor.rb
class AbstractPaymentProcessor
  def initialize(store)
    @store = store
  end

  def call
    raise "Abstract method"
  end

end
```

```ruby
## payment_processor.rb
class PaymentProcessor < AbstractPaymentProcessor

  def call
    # 1.- Primero obtenemos las ventas a procesar.
    # Filtramos para dejar sólo las ventas no procesadas de la tienda que
tengan valor
    sales = @store.sales
    sales = sales.select{|sale| sale.unprocessed? && sale.value > 0 }

    # 2.- Hay casos en los que no debemos continuar con el proceso.
    # Actualmente es si no hay ventas o si la compañía tiene el procesamiento
detenido
    if sales.empty? || @store.processing_status == :disabled
      return # abortamos
    end

    # 3.- Iniciamos el proceso. Local y con la tienda
    remote_id = @store.remote_sales_process.start
    payment = Payment.create(remote_id: remote_id, status: :started, store:
@store)

    # 4.- Reportamos las ventas
    sales.each do |sale|
      @store.remote_sales_process.report(remote_id, sale)
      sale.processed!
      payment.sales << sale
    end

    # 5.- Terminamos el proceso con la tienda y local
    resume = @store.remote_sales_process.stop(remote_id)
    payment.update(status: :completed, resume: resume)

    # 6.- Ejecutamos las tareas post proceso
    # Actualmente no hay nada que hacer

  end

end

## otro_archivo.rb

  # ...

  ## Este es el código que efectúa el procesamiento de las ventas
  def make_payment(store)
    processor = PaymentProcessor.new(store)
    processor.call
  end

  # ...
```

PAUTA PREGUNTA 3

El patrón más apropiado para resolver esto es el patrón template. Es importante que:

- transformen el proceso actual en métodos que constituyen los pasos (al menos los suficientes como para poder hacer las diferencias entre el proceso 1 y 2) y un método que es el que hace el llamado a los métodos de los pasos.
- tengan al menos 2 clases, que constituyen las 2 variantes del proceso
- hagan bien el llamado a usar una u otra implementación

En caso de elegir otro patrón, el patrón elegido debe estar correctamente implementado y debe responder con el requisito: el código debe permitir la convivencia del proceso 1 y 2. No es válido si simplemente llenaron de ifs el código.

A continuación un ejemplo de solución usando una implementación "clásica" del patrón template.

```ruby
## abstract_payment_processor.rb
class AbstractPaymentProcessor
  def initialize(store)
    @store = store
  end

  def call
    # 1.- Primero obtenemos las ventas a procesar.
    sales = get_sales

    # 2.- Hay casos en los que no debemos continuar con el proceso.
    if avoid_continue(sales)
      return # abortamos
    end

    # 3.- Iniciamos el proceso. Local y con la tienda
    payment = start_process

    # 4.- Reportamos las ventas
    report_sales(payment, sales)

    # 5.- Terminamos el proceso con la tienda y local
    finish_process(payment)

    # 6.- Ejecutamos las tareas post proceso
    post_process(payment)
  end

  protected

  # estos son los metodos usados por el template

  # Entrega el listado de ventas
  def get_sales
    raise "Not implemented"
  end

  # Dice si el proceso se debe abortar (true)
```

```ruby
  def avoid_continue(sales)
    raise "Not implemented"
  end

  # Inicia el proceso. Retorna un objeto Payment
  def start_process
    raise "Not implemented"
  end

  # Efectua el reporte de las ventas
  def report_sales(payment, sales)
    raise "Not implemented"
  end

  # completa el proceso, usando la información almacenada en el objeto
Payment
  def finish_process(payment)
    raise "Not implemented"
  end

  # efectua tareas post proceso
  def post_process(payment)
    raise "Not implemented"
  end

end



## payment_processor_v1.rb
class PaymentProcessorV1 < AbstractPaymentProcessor

  def get_sales
    # Filtramos para dejar sólo las ventas no procesadas de la tienda que
tengan valor
    sales = @store.sales
    sales = sales.select{|sale| sale.unprocessed? && sale.value > 0 }
    return sales
  end

  def avoid_continue(sales)
    # Actualmente cancelamos es si no hay ventas o si la compañía tiene el
procesamiento detenido
    if sales.empty? || @store.processing_status == :disabled
      return true
    end
    false
  end

  def start_process
    remote_id = @store.remote_sales_process.start
    payment = Payment.create(remote_id: remote_id, status: :started, store:
@store)
    return payment
  end

  def report_sales(payment, sales)
```

```ruby
    sales.each do |sale|
      @store.remote_sales_process.report(remote_id, sale)
      sale.processed!
      payment.sales << sale
    end
  end

  def finish_process(payment)
    resume = @store.remote_sales_process.stop(payment.remote_id)
    payment.update(status: :completed, resume: resume)
  end

  def post_process(payment)
    # Actualmente no hay nada que hacer
  end

end


## payment_processor_v2.rb
class PaymentProcessorV2 < AbstractPaymentProcessor

  def get_sales
    # Filtramos para dejar sólo las ventas no procesadas de la tienda que
tengan valor mayor a 1000
    sales = @store.sales
    sales = sales.select{|sale| sale.unprocessed? && sale.value > 1000 }
    return sales
  end

  def avoid_continue(sales)
    # Actualmente cancelamos es si no hay ventas o si la compañía tiene el
procesamiento detenido
    if sales.empty? || @store.processing_status == :disabled
      # como abortamos, creamos un registro con id remoto -1 y estado
:aborted
      Payment.create(remote_id: -1, status: :aborted, store: @store)
      return true
    end
    false
  end

  def start_process
    remote_id = @store.remote_sales_process.start
    payment = Payment.create(remote_id: remote_id, status: :started, store:
@store)
    return payment
  end

  def report_sales(payment, sales)
    sales.each do |sale|
      @store.remote_sales_process.report(remote_id, sale)
      sale.processed!
      payment.sales << sale
    end
  end
```

```
  def finish_process(payment)
    resume = @store.remote_sales_process.stop(payment.remote_id)
    payment.update(status: :completed, resume: resume)
  end

  def post_process(payment)
    PaymentWorker.perform_async(payment.id, @store.id)
  end

end

## otro_archivo.rb

# ...

## Este es el código que efectúa el procesamiento de las ventas
def make_payment(store)
  processor = if store.process_type == 1
    PaymentProcessorV1.new(store)
  elsif store.process_type == 2
    PaymentProcessorV2.new(store)
  else
    raise "Unknown process type #{store.process_type}"
  end
  processor.call
end

# ...
```

En lo personal yo haría que la clase abstracta ofrezca algunas implementaciones por defecto, por ejemplo (estracto):

```
class AbstractPaymentProcessor
  # ...

  # Dice si el proceso se debe abortar (true)
  # Implementacion opcional
  def avoid_continue(sales)
    return false # por defecto el proceso siempre continua
  end

  # ...

  # efectua tareas post proceso
  def post_process(payment)
    # por defecto no se hace nada post-proceso
  end

end
```

b) 12 puntos

Nota de Pauta

El patrón usado debe estar claramente identificado y debe ser consistente con la implementación – como contraejemplo, si la solución usa un patrón estrategia pero dice que usó un patrón comando, está mal. A priori, la identificación correcta no vale puntaje, pero en caso que el alumno identifique mal su propio trabajo, debiera tener una penalización de hasta un 75% del puntaje de este ítem.

En cuando a las ventajas o desventajas, lo importante es evidenciar un análisis crítico a los cambios hechos Cada ventaja o desventaja justificada vale 4 puntos.

Algunos ejemplos:

- (Ventaja) Al usar el patrón template, es fácil extender un proceso actual para aplicar pequeños cambios (por ejemplo, el proceso v2 podría haber extendido el v1 haciendo sólo pequeños cambios).
- (Ventaja) Tras aplicar el patrón template, cuando haya que aplicar el proceso 3, será muy fácil, pues es cuestión de copiar el proceso v2 y sólo hacer los cambios necesarios.
- (Ventaja) Con el patrón template, las clases ahora tienen sólo métodos muy pequeños que son más fáciles de entender y testear
- (Ventaja) El proceso principal ahora es más claro (ya no depende de los comentarios para entender que se hace en cada paso)
- (Desventaja) Cuando dos implementaciones son muy similares, hay mucha duplicación de código (en la solución de referencia, la clase PaymentProcessorV1 y PaymentProcessorV2 son casi idénticas). Sería bueno cambiar un poco la aplicación del patrón, por ejemplo, haciendo que la clase V2 extienda a la V1.
- (Desventaja) Al usar el patrón template, para hacer el seguimiento de lo que hace el código debo estar revisando 2 archivos, el template abstracto y la implementación en uso.

Pregunta 4 (12 puntos)

Para cada uno de los siguientes métodos, deberás decidir si pertenece al modelo, a la vista o al controlador y dar una razón. Para cada caso, la correcta decisión valdrá 1 punto + 0.5 punto por la justificación (total: 1.5 por método).
Opcionalmente podrías optar por una decisión parcial y clasificar al método entre dos opciones (por ejemplo, “no estoy seguro entre vista y controlador”). Esta decisión parcial + una justificación correcta tendrá un máximo de 1 puntos.

El Contexto

Todos los métodos corresponden a una aplicación para el reparto de comida. En esta aplicación, los usuarios hambrientos pueden elegir cualquier restorán de comida “para llevar” de su área (independiente de si tienen o no despacho), elegir uno o más platos del restorán, pagar en línea y esperar recibir su comida en un plazo no superior a una hora. Los usuarios pueden hacer todo sin registrar una cuenta (pero deben completar todos sus datos cada vez) o pueden registrarse y mantener sus datos, historial de compra, etc. Además, restorantes asociados pueden ingresar con cuentas de restorán y publicar promociones, avisos publicitarios, etc. Los restorantes asociados también mantienen sus páginas con sus menús y los usuarios pueden comentar y valorar los distintos platos.

El Modelo

El modelo consiste principalmente de las siguientes clases:

User	un usuario registrado, puede ser los datos de un cliente o un usuario de tipo “restorán”, el cual puede gestionar los datos de su restorán asociado.
AnonUser	un usuario anónimo que está comprando sin cuenta de usuario.
Address	una dirección (de un usuario o de un restorán)
Restaurant	posee todos los datos de un restorán, incluyendo su dirección y sus platos (food)
Food	corresponde a una plato particular.
Comment	es el comentario efectuado por un usuario. Incluye una valoración de 0 a 10.
Order	es una orden concreta, para un usuario (o usuario anónimo), una dirección y un plato de comida.

```
## a
# Recibe un string.
# Entrega una lista de cafes
def search_food(search_string)
  like_search = "%#{search_string}%"
  sql_query = "name LIKE :like_search_string OR description LIKE
:like_search_string"
  Food.where(sql_query, like_search_string: like_search)
end
```

```ruby
## b
# Recibe los datos del comentario de un usuario y helpers
# Entrega la respuesta a entregar al usuario
def save_comment(data, helpers)
  food = Food.find(data[:food_id])
  comment = food.comments.build( filter_comment_data(data[:comment]) )
  if comment.save
    helpers.redirect_to helpers.view_food_path(coffee)
  else
    @comment = comment
    helpers.render 'new_comment'
  end
end

## c
# Recibe un comentario  y un objeto con helpers de vista
# Entrega un string con las estrellas
# que representan la valoracion que da el comentario
def starts(comment, helpers)
  # el valor va de 0 a 10,
  # pero son 5 estrellas que pueden estar semi-pintadas
  full_starts = Array.new(10,0) # arreglo de 10 ceros
  # vamos a marcar con 1 hasta el valor
  comment.value.times{ |i| full_starts[i] = 1 }
  # ahora lo voy a llevar a 5 estrellas
  # cada estrella tendra un valor:
  # 0 -> vacia
  # 1 -> semi-pintada
  # 2 -> pintada
  starts = (0..4).map{|i| full_starts[2*i] + full_starts[2*i + 1] }
  starts_s = starts.map do |start|
    start_class = case start
    when 0 then 'empty'
    when 1 then 'half'
    else 'full'
    end
    helpers.content_tag('span', '', class: "start #{start_class}")
  end

  helpers.safe_join(starts_s)
end

## d
# Recibe una direccion.
# Entrega un string que representa a la direccion.
def address_string(address)
  out = StringIO.new
  if address.street_type.present?
    out << address.street_type
    out << " "
  end
  out << address.street_name
  out << " "
  out << address.number
  if address.aditional_info.present?
    out << " "
    out << address.aditional_info
```

```ruby
  end
  out.string
end

## e
# Recibe una direccion.
# Entrega una zona para determinar factibilidad de despacho
def zone(address)
  unless address.has_zone?
    # hay direcciones de las que no tenemos guardada la zona
    address.zone = Region
      .where(name: address.region_name)
      .first
      .zones
      .where(comune: address.comune)
      .available
      .first || Zone.closest_to(address.latitude, address.longitude)
    address.save!
  end
  address.zone
end


## f
# Recibe los datos entregados por el usuario al confirmar una orden de
comida.
# Entrega los mismos datos como los espera la clase Order.
def filter_order(data)
  result = {}
  if logged_user?
    result[:user_id] = current_user_id # metodo que entrega el id del usuario
actual
  else
    # a un anonimo igual le creamos una sesion, solo que nunca lo marcamos
como logueado
    result[:anon_user_id] = current_anon_user_id
  end
  result[:food_ids] = data[:foods]
  # en este punto el usuario ya ingreso una direccion
  # no podemos quedarnos simplemente con la del usuario,
  # porque podria haber elegida otra.
  # Si no eligio una direccion especifica, usamos la del usuario
  result[:address_id] = data[:address_id] ||
    current_user.try(:address_id) ||
    current_anon_user.try(:address_id)
  result
end
```

```ruby
## g
# Recibe un restoran y una zona
# Entrega true o false si es posible el reparto
# desde ese restoran a la zona pedida
def available?(restaurant,zone)
  r_zone = restaurant.zone
  distance = r_zone.distance_to(zone)
  # segun configuracion la distancia permitida puede cambiar
  # por dia / hora o otras razones configurables
  if distance > Delivery.max_distance
    false # fuera de distancia maxima
  else
    true
  end
end


## h
# Recibe un restoran.
# Entrega los mejores platos segun la valoracion de los usuarios
def recommendations(restaurant)
  foods = restaurant.foods
  # ordenamos por valoracion. Si no tienen, dejaremos al final
  foods = foods.sort_by{|food| food.average_starts || -1 }
  # el sort_by ordena de menor a mayor, asi que necesitamos invertir
  foods = foods.reverse
  # retornamos
  foods
end
```

PAUTA PREGUNTA 4

Como regla general, justificaciones válidas "por default":

Modelo – lógica de negocios

Vista – genera HTML / genera trozos de vista / se encarga de "cómo se ve" algo

Controlador – Traduce datos del usuario / recibe los datos del usuario y entrega una respuesta adecuada

Aciertos parciales

Para cada método, pertenecen exclusivamente sólo a la vista, al modelo o al controlador. Si, por ejemplo, la respuesta correcta es "vista" y eligen "vista-controlador" (con una justificación razonable), tendrían todo el puntaje parcial (1 puntos).

Es posible que algún método, interpretado de alguna forma distinta a la imaginada por el autor, corresponda a algo distinto a lo señalado a continuación (de ahí que es fundamental la justificación).

a: modelo

b: controlador

c: vista

d: vista (pensando en que se está ajustando para como lo verá el usuario)

e: model

f: controlador

g: modelo

h: modelo

Nota de Pauta

Cada ítem entrega 1 punto por la decisión correcta y 0.5 puntos por la justificación, o 1 punto por un match parcial con una justificación razonable.