

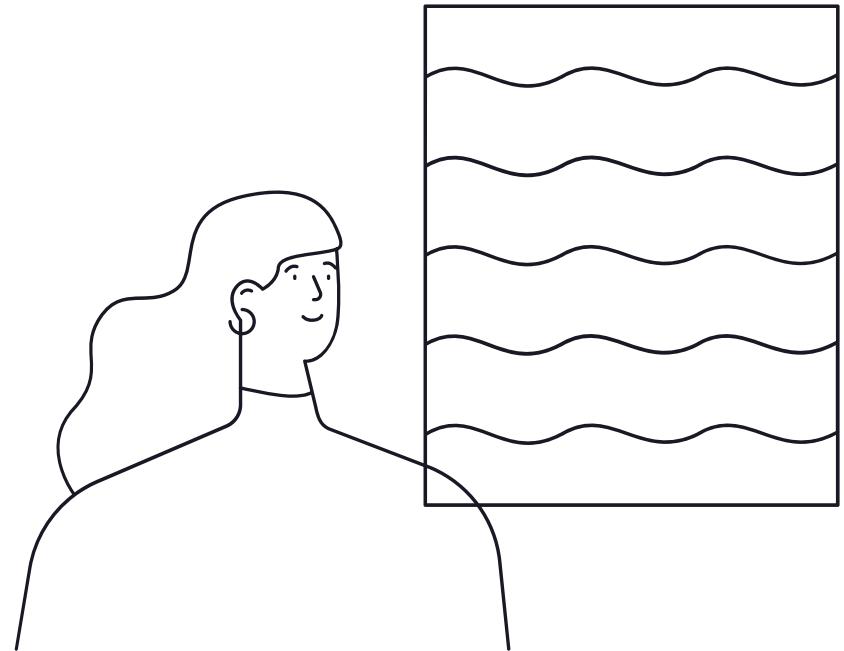
Preparando la I2

Diseño Detallado de Software

Pregunta 1

Refactorizar el siguiente código para hacerlo más mantenible y para disminuir el código repetido, mejorando también las buenas prácticas. Tu respuesta debe contener:

1. Líneas o funciones donde hiciste los cambios. Debes explicar el por qué de los cambios.
2. El código que implementes debe estar ya refactorizado y funcional.



Descargar código:

https://drive.google.com/file/d/1sPnDLL2H_HolvclUcjE1S4AxNtveDb2/view?usp=sharing

Código

```
badd_code
```

```
# He knows the way
class Mando
  attr_accessor :hp
  def initialize
    @hp = 100
    @power_attack = 10
  end
  def shot(enemy)
    attack = (@power_attack * rand).to_i
    enemy.hp -= attack
    p "bang (-#{attack})"
  end
  def walk
    p 'walking'
  end
end
```

```
badd_code
```

```
# A cute baby
class Baby
  attr_accessor :hp
  def initialize
    @hp = 50
    @power_attack = 100
  end
  def the_force(enemy)
    attack = @power_attack * rand
    enemy.hp -= attack
    @hp -= 2
    p "*magic sounds* (-#{attack})"
  end
  def walk
    p 'walking'
  end
end
```

Código

```
Bad_code_p2.rb
```

```
# Someone evil
class Enemy
  attr_accessor :hp

  def initialize
    @hp = 20
    @power_attack = 1
  end

  def attack(someone)
    attack = (@power_attack * rand).to_i
    someone.hp -= attack
    p "bang (-#{attack})"
  end

  def walk
    p 'walking'
  end
end
```

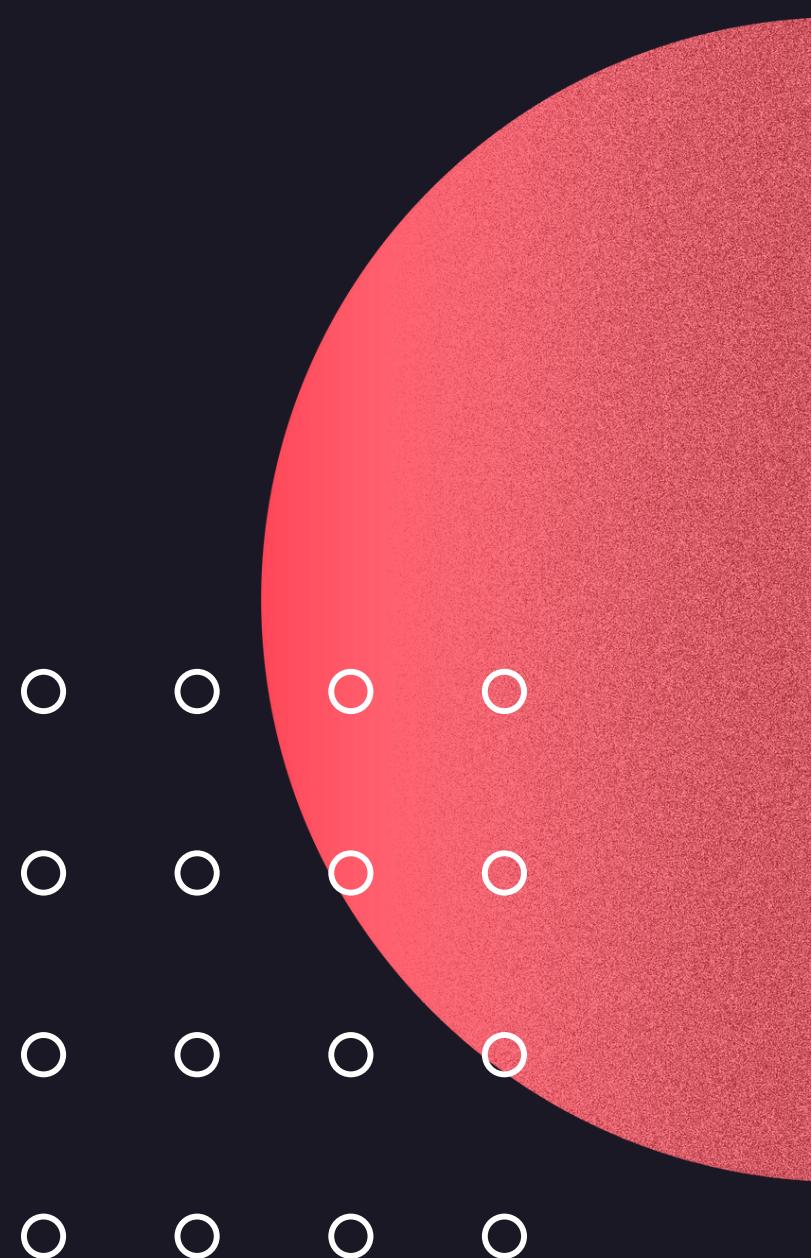
```
Bad_code.rb
```

```
# Runs the simulation
class Main
  def self.run
    mando = Mando.new
    baby = Baby.new
    enemies = [Enemy.new, Enemy.new]

    while enemies.map(&:hp).reduce(0, :+) > 0 || (mando.hp <= 0 || baby.hp <= 0)
      mando.walk if mando.hp > 0
      baby.walk if baby.hp > 0
      enemies[0].walk if enemies[0].hp > 0
      enemies[1].walk if enemies[1].hp > 0
      mando.shot(enemies[0])
      mando.shot(enemies[1])
      if mando.hp < 10
        baby.the_force(enemies[0])
        baby.the_force(enemies[1])
      end
    end
  end
end

Main.run
```

Algunos errores de diseño son:



- Código duplicado.
- Main no escalable.
- Clases no extensibles.
- Código de difícil lectura.

¿Cómo lo podemos solucionar?

- Trabajar con clases abstractas ya que existen elementos en común en las 3 clases presentadas.
- En las clases abstractas indicar aquellos métodos indispensables que deben de ser implementados
- Modularizar responsabilidades dentro de cada clase usando métodos privados, para que cada método se encargue de una lógica en particular
- Si se ponen condiciones o lógicas muy complejas, separar en métodos con nombres descriptivos
- Dividir las acciones a ejecutar en el main para hacer de más fácil lectura y comprensión su ejecución

Solución propuesta

```
  nice_code

class Character
  attr_accessor :hp

  MSG_ERROR = "Method must have been implemented"

  def initialize
    @hp = 0
    @power_attack = 0
  end

  def walk
    p 'walking' if @hp > 0
  end

  private

  # Print an attack effect sound and show the damage done
  def sound_attack(attack); raise MSG_ERROR; end

end
```

```
  nice_code

class Mando < Character
  def initialize
    @hp = 100
    @power_attack = 10
  end

  def shot(someone)
    attack = (@power_attack * rand).to_i
    someone.hp -= attack
    sound_attack(attack)
  end

  private

  def sound_attack(attack)
    p "bang (-#{attack})"
  end
end
```

Solución propuesta

... nice_code

```
class Baby < Character
  def initialize
    @hp = 50
    @power_attack = 100
  end

  def the_force(enemy)
    attack = (@power_attack * rand).to_i
    enemy.hp -= attack
    @hp -= 2
    sound_attack(attack)
  end

  private

  def sound_attack(attack)
    p "*magic sounds* (-#{attack})"
  end
end
```

... nice_code

```
class Baby < Character
  def initialize
    @hp = 50
    @power_attack = 100
  end

  def the_force(enemy)
    attack = (@power_attack * rand).to_i
    enemy.hp -= attack
    @hp -= 2
    sound_attack(attack)
  end

  private

  def sound_attack(attack)
    p "*magic sounds* (-#{attack})"
  end
end
```

Solución propuesta

```
nice_code

# Runs the simulation
class Main
  def self.run
    create_protagonists
    create_enemies(2)

    start_simulation
  end

  private

  def self.create_protagonists
    @mando = Mando.new
    @baby = Baby.new
  end

  def self.create_enemies(n_enemies)
    @enemies = []
    for _ in 1..n_enemies do
      @enemies << Enemy.new
    end
  end
end
```

```
nice_code

def self.start_simulation
  while any_enemy_alive? || protagonists_alive?
    @mando.walk
    @baby.walk
    @enemies.each do |enemy|
      enemy.walk
      @mando.shot(enemy)
      @baby.the_force(enemy) if @mando.hp < 10
    end
  end
end

def self.any_enemy_alive?
  @enemies.map(&:hp).reduce(0, :+) > 0
end

def self.protagonists_alive?
  (@mando.hp <= 0 || @baby.hp <= 0)
end

Main.run
```

Descargar código:

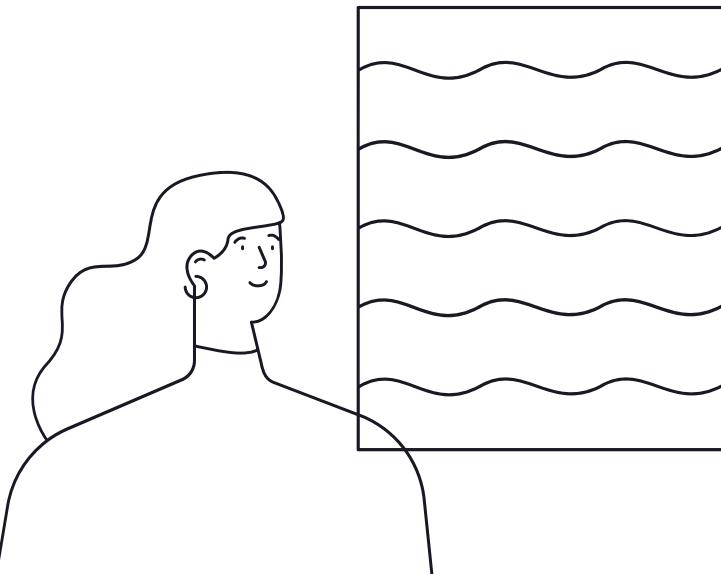
https://drive.google.com/file/d/1_DUfcifMAwiv-HXy0EhE9o5IK4olbIY1/view?usp=sharing

Nota: Esta es solo una solución propuesta que fue diseñada por los ayudantes, ¿Se podría mejorar más? Por supuesto! Te invitamos proponer cambios y hacer refactor al código.

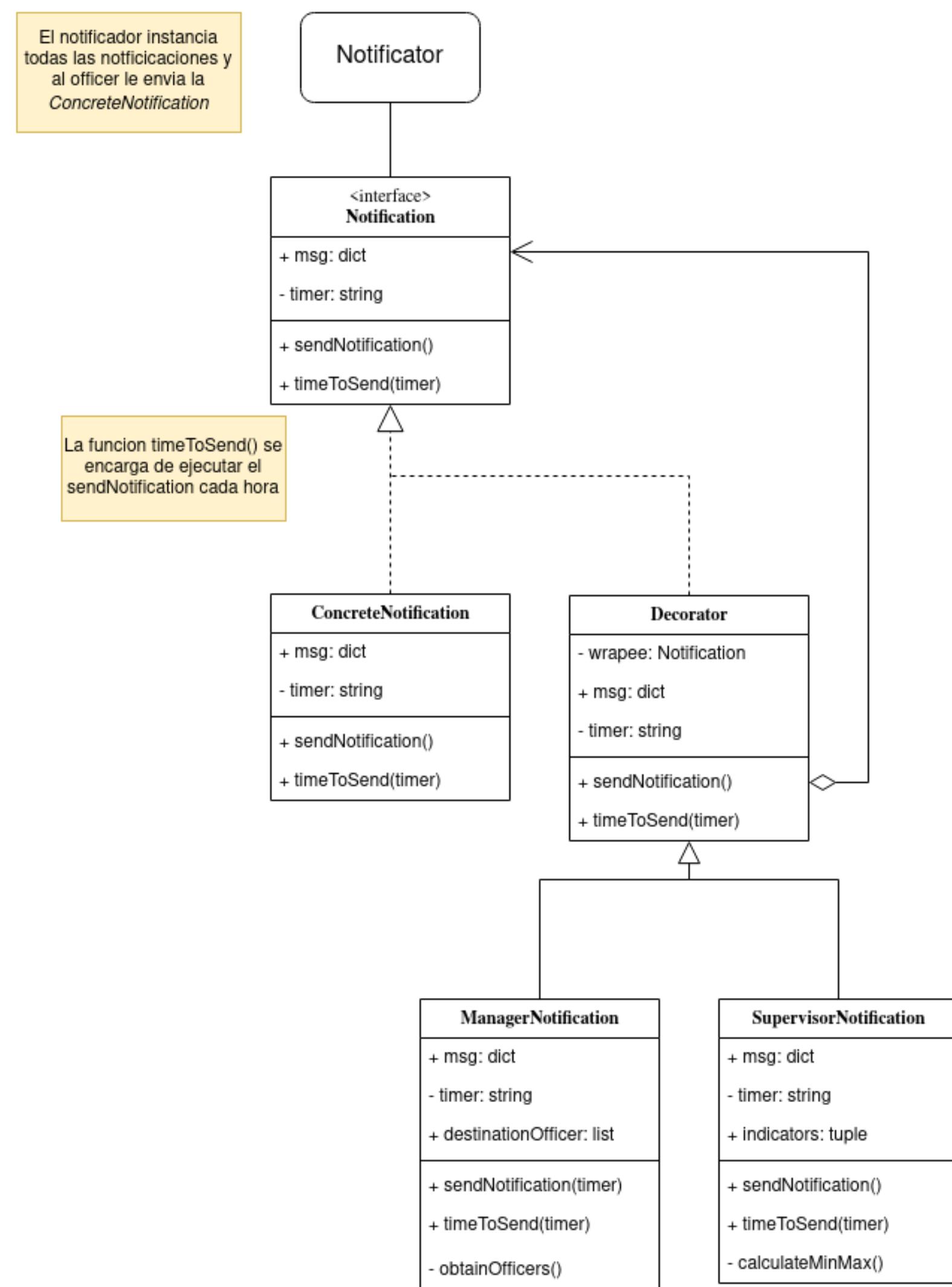
Pregunta 2

Una clase Notificator necesita enviar mensajes de un evento (Event) cada una hora a distintos tipos de usuario. Esta información base es la misma, pero varía lo que se decide mostrar o lo que se calcula dependiendo del tipo de usuario. Hay 3 tipos de usuarios de momento: manager, officer y supervisor. Los 3 usuarios reciben el cuerpo del evento que contiene un indicador decimal que va del -10 al 10. Las diferencias en la notificación son que el supervisor recibe la lista de id's de usuarios officer que recibirán la notificación y el manager recibe el valor máximo y mínimo de los indicadores de eventos enviados de los últimos 100 eventos.

Utilizando algún patrón de diseño, implemente solamente la lógica de la notificación a los usuarios en base al patrón escogido. Puede usar un diagrama de clases o escribir el código. Si decide usar un diagrama, cualquier información faltante le restará puntaje. Si escribe código, puede abstraerse de algunos cálculos a través de funciones no implementadas. Finalmente, justifique el por qué del patrón utilizado.



Solución propuesta



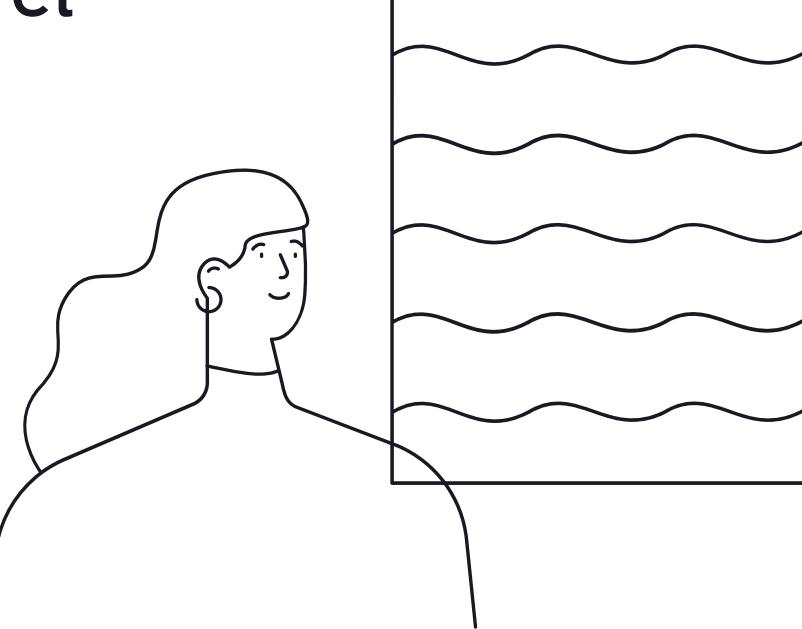
¿Por qué se escogió este patrón?

- Existen distintos usuarios que requieren un mismo mensaje base, pero distinta información adicional dependiendo de quien se trate. Entonces, era necesario un wrapper para dejar cada mensaje con la información que requiere cada tipo de usuario, ademas las funcionalidades específicas para calcular esta información.
- Otro punto a considerar es que eventualmente se quieran agregar otros tipos de usuario, por lo que esto debiese de ser posible sin necesidad de alterar las notificaciones ya existentes.

Pregunta 3

Identifique el o los code smells presentes en el siguiente código. Debe señalar, explicar el por qué de este code smell y las implicancias que podrían significar.

Contexto: Google es un buscador web que además ofrece servicios de correo como Gmail. Facebook es una red social. La aplicación actualmente permite login con Facebook y Google, además de compartir en redes sociales algunos avances de la aplicación. También realiza búsquedas a través de google con el buscador público de la aplicación.



Código



Problema 4

```
public class Client
{
    protected void Login() { ... }
    protected void NewPost() { ... }
    protected void Search() { ... }
    protected void AddFriend() { ... }
}
public class Google : Client
{
    ...
}
public class Facebook : Client
{
    ...
}
```

Solución propuesta

El code smell más evidente es *Refused Bequest*, porque en el uso de la herencia del cliente, según el contexto dado, las funcionalidades no se comparten entre si. En base al contexto, lo único en común que tienen es el Login(), sin embargo el resto de las funciones no se justifican. Por ejemplo Search() en el contexto de buscar con el buscador público no hará sentido en el código de la clase padre. Este tipo de code smell indica o podría implicar una mala utilización de los objetos (herencia) y además falta de diseño inicial.

