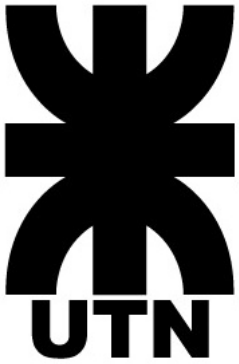

Resumen Paradigma Orientado a Objetos
Facultad Regional Buenos Aires
Universidad Tecnologica Nacional

Paradigmas de Programación – $K2032$

Curso Jueves Noche - Profesor Nicolás Scarcella canal de youtube

Escrito en *Latex* por: PAZ PORTILLA, José Miguel 2028244
jpazportilla@frba.utn.edu.ar

7 de diciembre de 2023

Índice

1. Introducción a Objetos ->Video 17 Youtube

En este paradigma de objetos se vuelve a tener efecto, pero es menos declarativo que funcional y lógico. Se utiliza el lenguaje de programación Wollok. La idea es combinar estructuras de datos y operaciones.

Las caracteristicas de un objeto son:

1. Exponen una interfaz, es un conjunto de operaciones con las que se pueden interactuar con el objetos. Solo se puede interactuar con objetos mediantes mensajes. Los mensajes que un objeto entiende va a ser el resultado de poseer metodos.
2. Pueden llegar a tener estado interno, que son atributos, es decir referencias a otros objetos. Estos atributos pueden cambiar de referencia y apuntar a otros objetos.
3. Tienen una identidad, cada objeto es diferente a cualquier otro, aunque hayan otros que respondan a los mismos mensajes y estado interno.

1.1. Problema: La Golondrina Pepita

- Un ornitólogo nos pide ayuda para estudiar el Consumo de Energía de la golondrina Pepita
- El Volar consume energia de Pepita.
- El Comer recupera la energía de Pepita.

1.1.1. Resolución: La Golondrina Pepita

- La palabra object define un objeto nuevo.
- La palabra var define un atributo que podrá ser cambiado
- La palabra method permite crear métodos.
- El metodo volar() y comer() causan efecto.
- El metodo energia() son solo de consulta.

1.1.2. Wollok: La Golondrina Pepita

```
1  object pepita
2  {
3      var energia = 100
4
5      method vola (kilometros)
6      {
7          energia = energia - kilometros * 2
8      }
9
10     method come (gramos)
11     {
12         energia = energia + gramos * 10
13     }
14
15     //Es un getter, obtiene el valor del atributo energia y lo retorna
```

```
16      //Cuando retorna una expresion se utiliza esa forma de definir metodos
17      method energia () = energia
18      //Se podria haber definido asi
19      /*
20       method energia ()
21       {
22         return energia
23       }
24      */
25  }
```

1.2. Problema: La entrenadora de aves Emilia

- Emilia solo sabe entrenar aves
- La aves deben comer 5g, volar 10km y volver a comer 5g.

1.2.1. Resolución: La entrenadora de aves Emilia

- Emilia no conoce a Pepita, pero como Pepita entiende los mensajes come y vola, entonces podra entrenarla.

1.2.2. Wollok: La entrenadora de aves Emilia

```
1   import pepita.*
2   import pepote.*
3
4   object emilia
5   {
6       method entrena (ave)
7       {
8           ave.come(5)
9           ave.vola(10)
10          ave.come(5)
11      }
12  }
```

1.3. Emilia entrena a Pepita

Emilia puede entrenar a cualquier objeto que entienda los mensajes come y vola. En la Figure 1 se observa que pepita sufre el efecto luego que emilia la entrena, aumentando su energia de $100J$ a $180J$.

1.3.1. Wollok: Emilia entrena a Pepita

```
1   import pepita.*
2   import emilia.*
3
4   describe "Emilia conoce a su golondrina Pepita"
5   {
6       test "Pepita inicia con 100J de ejergia"
```

```
7       {
8           assert.equals(100,pepita.energia())
9       }
10      test "Emilia entrena a Pepita y finalmente tiene 180J de energia"
11      {
12          emilia.entrena(pepita)
13          assert.equals(180,pepita.energia())
14          /*Ya que luego de 5g su energia aumento en 5 * 10= 50J
15           * Luego volar 10km y su energia disminuyo en 10 * 2 = 20J
16           * Finalmente volvio a comer 5g y su energia aumento en 5*10 = 50J
17           * por lo tando la energia final resulta 100+50-20+50=180J
18           */
19      }
20  }
```

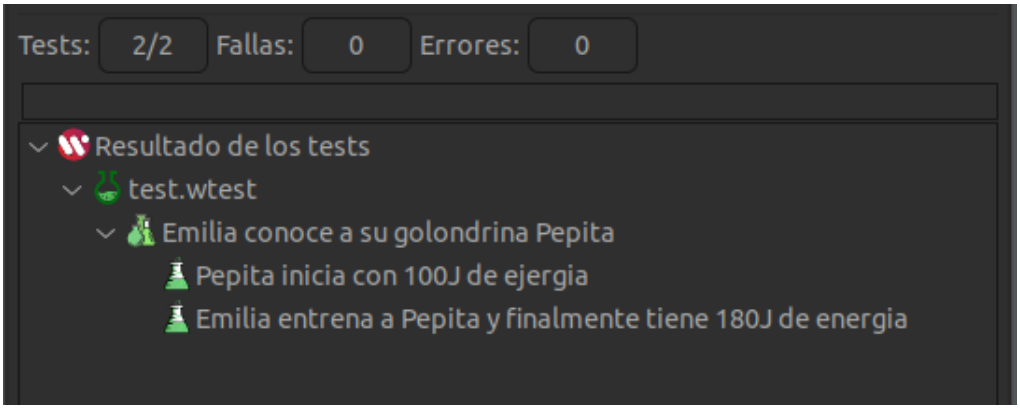

Figura 1. Test unitario de pepita siendo entrenada por emilia.

1.4. Conceptos clave de la programación orientada a objetos

1. Los objetos están encapsulados ya que oculta y protege de los detalles de su implementación. Solo veo la interfaz de un objeto, o sea que mensajes entiende. Solo nos intereza saber que puede hacer. No puedo ver, ni usar los atributos de un objeto, sólo un objeto puede manipular sus atributos.

2. Cuando se delega en un objeto alguna actividad, se le da la responsabilidad al objeto de saber como hacerlo mientras que lo termine haciendo.

3. El polimorfismo implica que un objeto que envia mensajes pueda manipular al menos a dos objetos, siempre y cuando ellos entiendan los mensajes que envia el objeto.

1.5. Otra ave: El hancón pepote

Pepote es otra ave ya que entiende los mismos mensajes que Pepita, es decir come y vola. Para emilia que sabe entrenar aves le es indiferente cual de ellos debe entrenar. Pepote causa efecto sobre su energia al volar y comer, pero lo hace de forma distinta a Pepita. Se observa que su tiene como atributos a comido y volado, ademas su metodo energia no es un getter, sino que devuelve el resultado una operación.

1.6. Wollok: El hancón pepote

```
object pepote
{
    var volado = 0
    var comido = 0

    method vola (kilometros)
    {
        volado = volado + kilometros
    }
    method come(gramos)
    {
        comido = comido + gramos
    }
    method energia () = 255 + comido **2 - volado / 5
}
```

1.7. Wollok-Test: Emilia entrena a Pepote

```
import pepote.*
import emilia.*

describe "Emilia conoce a su Halcón Pepote"
{
    test "Pepote inicia con 255J de ejergia"
    {
        assert.equals(255,pepote.energia())
    }
    test "Emilia entrena a Pepote y finalmente tiene 353J de energia"
    {
        emilia.entrena(pepote)
        assert.equals(353,pepote.energia())
        /*Ya que luego de comer 5g
         * luego de volar 10km
         * y volver a comer 5g, sus atributos quedan
         * volado 10km y comido 10g, por lo tanto
         * su energia es 255 + 10**2 - 10/5 =
         * = 255 + 100 -2 = 353
         */
    }
}
```

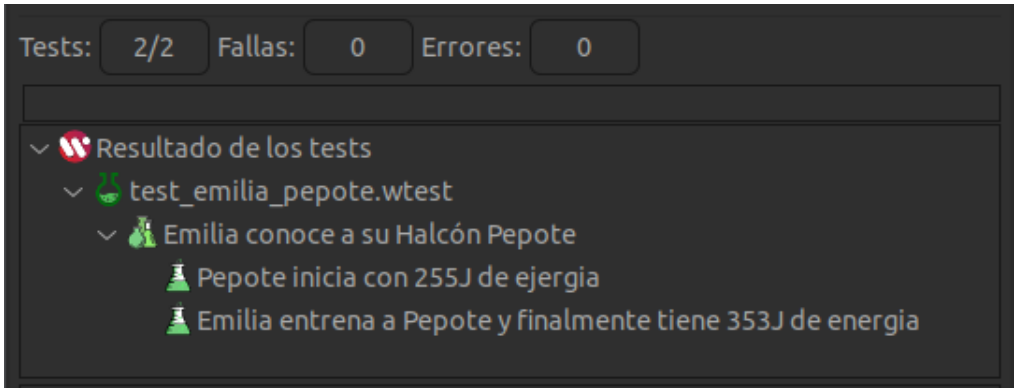

Figura 2. Test unitario de pepote siendo entrenado por emilia.

1.8. Diseño de métodos para favorecer el polimorfismo entre objetos

Si se tiene un objeto Pepaza que entiende los mensajes:

- come()
- volar(kilometros)
- nadar()

Emilia no la podrá entrenar ya que Pepaza no entiene el mensaje vola, sino volar. Además su metodo come no espera un parametro que indique la cantidad que debe comer. Solución cambiar volar por vola y agregar parametro gramos en el metodo come aunque no se use.

Si se tiene otro objeto Pepudo que entiende los mensajes:

- come()
- nada()

Emilia tampoco lo podrá entrenar a Pepudo, ya que no entiende el mensaje vola.

1.9. Otro entrenado Ramiro

Ramiro en otro entrenado de aves. Si esta de buen humor hace volar a las aves 15km, sino 30km. Se sabe que si duerme por lo menos 8 horas entonces esta de buen humor, sino no.

1.10. Resolución: Otro entrenado Ramiro

Por ser entrenador, debe enterder el mensaje entrena que tiene un ave como parametro. Ademas tiene un atributo variable que indica las horas que durmio. Lo inicio en 0 horas dormida y entonces esta de mal humor. Si entreda a Pepita o Pepote los hara volar 30km, pero si en cambien durmio al menos 8 horas, entonces solo hará volar al ave que entrene 15km

1.11. Wollok: Otro entrenador Ramiro

```
1 import pepita.*
2 import pepote.*
3
```

```
4  object ramiro
5  {
6      var horasDormidas = 0
7
8      //Getter->Permite obtener el valor del atributo horasDormidas
9      method horasDormidas () = horasDormidas
10     //Setter->Permite establecer un valor _horasDormidas al atributo
           horasDormidas
11     method horasDormidas (_horasDormidas)
12     {
13         horasDormidas = _horasDormidas
14     }
15
16     //Retorna true si horasDormidas es mayor o igual a 8, sino retorna
           false
17     method estaDeBuenHumor () = horasDormidas >= 8
18
19     method entrena (ave)
20     {
21         //Si estaDeBuenHumor es true, asigna 15 a la constante distancia,
               sino asigna 30
22         const distancia = if ( self.estaDeBuenHumor() ) 15 else 30
23         //Hace volar a ave la distancia que establecio previamente
24         ave.vola(distancia)
25     }
26 }
```

1.12. Wollok-Test: Ramiro entrena a Pepita y Pepote, con buen y mal humor

```
1  import pepita.*
2  import pepote.*
3  import ramiro.*
4
5  describe "Ramiro entrena aves con mal humor"
6  {
7      test "Ramiro no durmio y tiene mal humor"
8      {
9          assert.notThat(ramiro.estaDeBuenHumor())
10     }
11     test "Pepita inicialmente tiene 100J de energia, si es entrenada por
           ramiro un dia que tiene mal humor termina con 40J de energia"
12     {
13         assert.equals(100,pepita.energia())
14         ramiro.entrena(pepita)
15         /*
16          * Ramiro hace volar a pepita 30km, su energia disminuye en 30*2=60J
                .
17          * Por lo tanto luego de ser entrenada por ramiro termina con
                100-60=40J
18         */
19         assert.equals(40,pepita.energia())
20     }
21     test "Pepote inicialmente tiene 255J de energia, si es entrenada por
           ramiro un dia que tiene mal humor termina con 249J de energia"
```

```
22      {
23          assert.equals(255,pepote.energia())
24          ramiro.entrena(pepote)
25          /*
26           * Ramiro hace volar a pepote 30km, entonces ha volado 30km y
27           * por ende su energia es 255 - 30/5 = 249J
28          */
29          assert.equals(249,pepote.energia())
30      }
31  }
32
33  describe "Ramiro entrena aves con buen humor"
34  {
35      test "Ramiro durmio 8 horas y tiene buen humor"
36      {
37          ramiro.horasDormidas(8)
38          assert.that(ramiro.estaDeBuenHumor())
39      }
40      test "Pepita inicialmente tiene 100J de energia, si es entrenada por
            ramiro un dia que tiene buen humor termina con 70J de energia"
41      {
42          ramiro.horasDormidas(8)
43          assert.equals(100,pepita.energia())
44          ramiro.entrena(pepita)
45          /*
46           * Ramiro hace volar a pepita 15km, su energia disminuye en 15*2=30J
                .
47           * Por lo tanto luego de ser entrenada por ramiro termina con
                100-30=70J
48          */
49          assert.equals(70,pepita.energia())
50      }
51      test "Pepote inicialmente tiene 255J de energia, si es entrenada por
            ramiro un dia que tiene buen humor termina con 252J de energia"
52      {
53          ramiro.horasDormidas(8)
54          assert.equals(255,pepote.energia())
55          ramiro.entrena(pepote)
56          /*
57           * Ramiro hace volar a pepote 15km, entonces ha volado 15km y
58           * por ende su energia es 255 - 15/5 = 252J
59          */
60          assert.equals(252,pepote.energia())
61      }
62  }
```

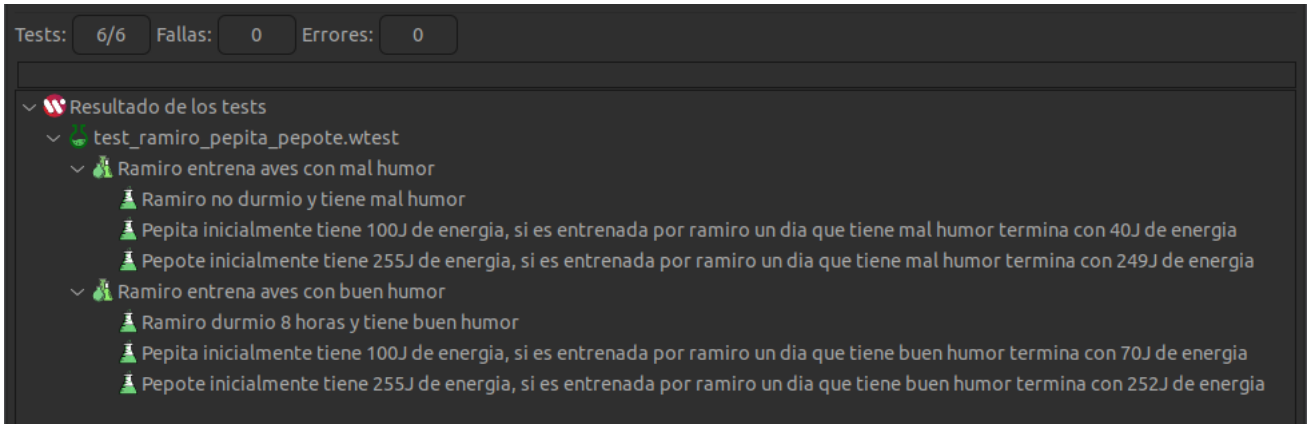

Figura 3. Test unitario de ramiro entrenado con buen y mal humor a pepita y pepote.

2. Práctica: La Feria ->Video 18 Youtube

Julieta cuenta con tickets y ademas no está cansada. Va a jugar los distintos juegos de la feria y al finalizar termina con más tickets y cansada dependiendo de cada juego.

2.1. Wollok: Jugadores de la feria

```
import juegos.*
import premios.*

object julieta
{
    //Cantidad de Tickets del año anterior, puede ser modificado
    var property tickets = 15
    //Esta completamente descansada
    var cansancio = 0

    method punteria() = 20
    method fuerza() = 80 - cansancio

    //Cada vez que juega algun juego, cambian el valor de sus atriburos
        tickets y cansancio
    method jugar ( juego )
    {
        tickets = tickets + juego.ticketsGanados (self)
        cansancio = cansancio + juego.cansancioQueProduce ()
    }
    method puedeCanjear ( premio ) = tickets >= premio.costo()
}
object gerundio
{
    method jugar() { }
    method puedeCanjear ( premio ) = true
}
```

2.2. Wollok: Juegos de la feria

```
object tiroAlBlanco
{
    method ticketsGanados (jugador) = ( jugador.punteria() / 10 ).roundUp()
    method cansancioQueProduce () = 3
}

object pruebaDeFuerza
{
    method ticketsGanados (jugador) = if (jugador.fuerza() > 75) 20 else 0
    method cansancioQueProduce () = 8

}

object ruedaDeLaFortuna
{
```

```
16      var property aceitada = true
17
18      //0.randomUpTo(20) Genera un numero real aleatorio entre 0 y 20
19      //roundUp() Redonde un numero real a uno entero, redondeando para
            arriba
20      method ticketsGanados (jugador) = 0.randomUpTo(20).roundUp()
21      method cansancioQueProduce() = if (aceitada) 0 else 1
22  }
```

2.3. Wollok: Premio de la feria

```
1   object osito
2   {
3       method costo () =    45
4   }
5
6   object taladro
7   {
8       var property costo = 200
9   }
```

2.4. Wollok-Test: Julieta juega en la feria

```
1   import jugadores.*
2   import juegos.*
3
4   describe "Julieta juega en la feria que llego al pueblo"
5   {
6       test "Julieta tiene 15 tickets del año anterior"
7       {
8           assert.equals (15,julieta.tickets())
9       }
10      test "Julieta como esta descansada(cansancio 0 puntos) tiene 80N de
            fuerza"
11      {
12          assert.equals (80,julieta.fuerza())
13      }
14      test "Julieta tiene 20 puntos de punteria"
15      {
16          assert.equals(20,julieta.punteria())
17      }
18      test "Julieta juega tiro al blanco y termina con 17 tickets y como le
            produce 3 puntos de cansancio, tiene 77N de fuerza despues de jugar
            "
19      {
20          julieta.jugar(tiroAlBlanco)
21          assert.equals (17,julieta.tickets())
22          assert.equals (77,julieta.fuerza())
23      }
24      test "Julieta juega prueba de fueza y termina con 35 tickets ya que
            tenia 80 puntos de fuerza y como le produce 8 puntos de cansancio,
            tiene 72N de fuerza despues de jugar"
```

```
25      {
26          julieta.jugar(pruebaDeFuerza)
27          assert.equals (35,julieta.tickets())
28          assert.equals (72,julieta.fuerza())
29      }
30      test "Si Julieta vuelve a jugar prueba de fuerza seguira teniendo 35
            tickets ya que no supera los 75 N de fuerza"
31      {
32          julieta.jugar(pruebaDeFuerza)
33          julieta.jugar(pruebaDeFuerza)
34          assert.equals (35,julieta.tickets())
35      }
36      test "Si Julieta juega la rueda de la fortuna gana entre 0 y 20 tickets
             más"
37      {
38          julieta.jugar(ruedaDeLaFortuna)
39          assert.that(julieta.tickets()>=15 and julieta.tickets()<=35)
40      }
41      test "Si la rueda de la fortuna esta aceitada no produce cansancio y
            sigue teniendo 80N de fuerza"
42      {
43          julieta.jugar(ruedaDeLaFortuna)
44          assert.equals (80,julieta.fuerza())
45      }
46      test "Si la rueda de la fortuna  no esta aceitada no produce 1 punto de
             cansancio y termina con 79N de fuerza"
47      {
48          ruedaDeLaFortuna.aceitada(false)
49          julieta.jugar(ruedaDeLaFortuna)
50          assert.equals (79,julieta.fuerza())
51      }
52  }
```

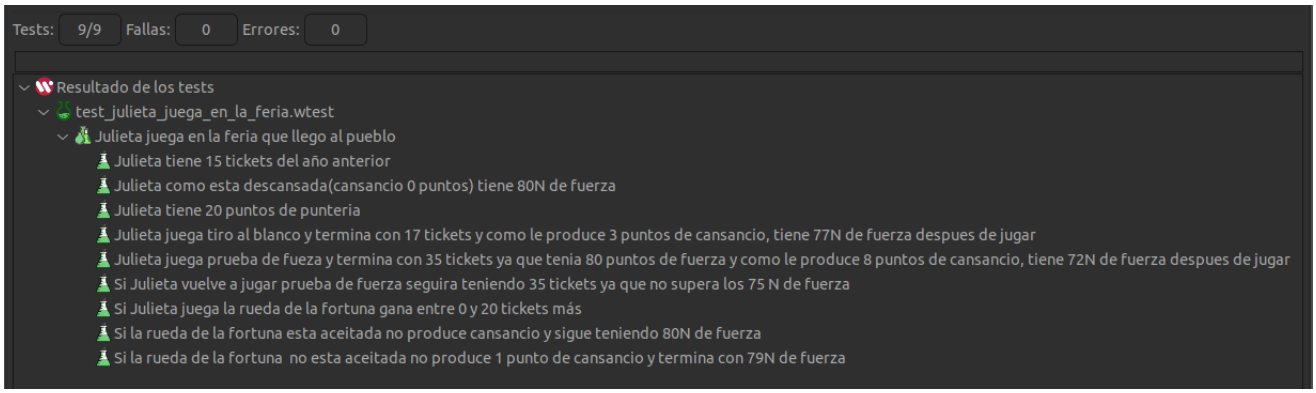

Figura 4. Test unitario julieta juega en la feria.

3. Clases como plantillas para crear objetos ->Video 19 Youtube

Las clases no son objetos, solo son sus moldes a partir de ellos se pueden crear objetos mediante una instanciación. La clases definen atributos y prevee metodos. No se le pueden mandar mensajes a las clases ya que no son objetos. Todos los objetos de una misma clase son polimorficos ya que entienden los mismos mensajes.

3.1. Wollok: Clase Golondrina

```
class Golondrina
{
    var energia = 100

    method vola(kilometros)
    {
        energia = energia - kilometros * 2
    }

    method come(gramos)
    {
        energia = energia + gramos * 10
    }

    method energia() = energia
}
```

3.2. Wollok-Test: Clase Golondrina

```
import golondrina.*
describe "Clase Golondrina"
{
    const pepita = new Golondrina()
    const pepito = new Golondrina(energia = 180)
    test "La energia de la golondrina pepita es 100"
    {
        assert.equals(100,pepita.energia())
    }
    test "La energia de la golondrina pepito es 180"
    {
        assert.equals(180,pepito.energia())
    }
}
```

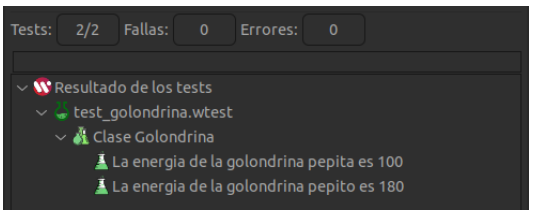

Figura 5. Test unitario clases golondrina.

4. Colecciones y Bloques ->Video 20 Youtube

4.1. Lista

Las listas se representas como: *const lista = new [1,2,3]*

Operaciones de Listas

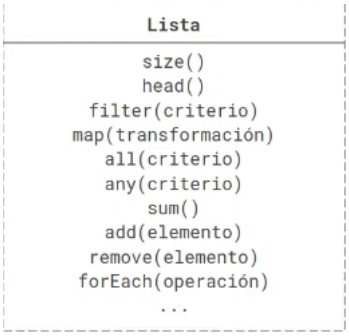

Figura 6. Operaciones de las listas.

De consulta son:

- size()
- head()
- filter(criterio)
- map(criterio)
- all(criterio)
- any(criterio)
- sum()

y de efecto:

- add(elemento)
- remove(elemento)
- forEach(operacion)

4.2. Wollok: Clase Vaca

```
class Vaca
{
    var property estaContenta = true
    var property litrosDeLeche = 100
    method ordeniar()
    {
        litrosDeLeche = litrosDeLeche - 20
    }
}
```

4.3. Wollok: Clase Cabra

```
class Cabra
{
    var property estaContenta = true
    var property litrosDeLeche = 50
    method ordeniar()
    {
        litrosDeLeche = litrosDeLeche - 30
    }
}
```

4.4. Wollok: Clase Corral

```
import vaca.*
import cabra.*

class Corral
{
    const property ordeniables = []

    method lecheDisponible() = ordeniables.filter{ordeniable=>ordeniable.
        estaContenta()}.sum{ordeniable=>ordeniable.litrosDeLeche()}
    method todasContentas() = ordeniables.all{ordeniable => ordeniable.
        estaContenta()}
    method ordeniar()
    {
        ordeniables.forEach{ ordeniable => if(ordeniable.estaContenta())
            ordeniable.ordeniar() }
    }
}
```

4.5. Wollok-Test: Prueba en lista de vacas

```
import vaca.*
describe "test de Lista de vacas"
{
    const rosita = new Vaca()
    const petunia = new Vaca(estaContenta = false)
    const vacas = [rosita, petunia]
    test "Hay 2 vacas"
    {
        assert.equals(2,vacas.size())
    }
    test "Agrego una nueva vaca, por ende ahora hay 3 vacas"
    {
        vacas.add(rosita)
        assert.equals(3,vacas.size())
    }
    test "Hay 1 referencia a una vaca que no esta contenta, que apunta a
        petunia"
```

```
17      {
18          vacas.add(rosita)
19          const tristes = vacas.filter{vaca=>!vaca.estaContenta()}
20          assert.equals(1,tristes.size())
21      }
22      test "Hay 2 referencias a vacas Contentas, ambas apuntan a rosita"
23      {
24          vacas.add(rosita)
25          const contentas = vacas.filter{vaca=>vaca.estaContenta()}
26          assert.equals(2,contentas.size())
27      }
28      test "Si remuevo a petunia de la lista de tristes, se queda sin
            elementos pero la lista de vacas sigue teniendo 2 vacas"
29      {
30          const tristes = vacas.filter{vaca=>!vaca.estaContenta()}
31          tristes.remove(petunia)
32          assert.equals(0,tristes.size())
33          assert.equals(2,vacas.size())
34      }
35      test "rosita y petunia tienen 100L de leche"
36      {
37          assert.equals(100,rosita.litrosDeLeche())
38          assert.equals(100,petunia.litrosDeLeche())
39      }
40      test "Si ordeño a las vacas tristes, petunia queda con 80L en la lista
            de vacas original y rosita con 100L"
41      {
42          const tristes = vacas.filter{vaca=>!vaca.estaContenta()}
43          assert.equals(1,tristes.size())
44          tristes.forEach{ vaca => vaca.ordeniar() }
45          assert.equals(100,rosita.litrosDeLeche())
46          assert.equals(80,petunia.litrosDeLeche())
47      }
48  }
```

5. Herencia ->Video 21 Youtube

Las subclases heredan atributos y metodos de una superclase. Cada subclase tiene una sola superclase. No hace falta tener herencia para tener polimorfismo.

5.1. Wollok: Clase Tanque

```
import arma.*

class Tanque
{
    const armas = []
    const tripulantes = 2
    var property salud = 1E3
    var property prendidoFuego = false

    method emiteCalor() = prendidoFuego or tripulantes > 3

    method sufrirDanio(danio)
    {
        salud = salud - danio
    }

    method atacar(objetivo)
    {
        armas.anyOne().dispararA(objetivo)
    }
}

class TanqueBlindado inherits Tanque
{
    const blindaje = 200

    override method emiteCalor() = false

    override method sufrirDanio(danio)
    {
        if(danio < blindaje)
            super(danio - blindaje)
    }
}
```

5.2. Wollok: Clase Arma

```
class Misil
{
    const potencia
    var agotada = false

    method dispararA(objetivo)
    {
        agotada = true
```

```
9         objetivo.sufrirDanio(potencia)
10     }
11     method agotada() = agotada
12 }
13
14 class MisilTermico inherits Misil
15 {
16     override method dispararA(objetivo)
17     {
18         if (objetivo.emiteCalor())
19         {
20             super(objetivo)
21         }
22     }
23 }
24
25 //Clase abstracta, no sera instanciada
26 class ArmaRecargable
27 {
28     var cargador = 100
29     method recargar()
30     {
31         cargador = 100
32     }
33     method agotada() = cargador <= 0
34 }
35
36 class Metralla inherits ArmaRecargable
37 {
38     const property calibre
39
40     method dispararA(objetivo)
41     {
42         cargador = cargador - 10
43         if (calibre >50)
44             objetivo.sufrirDanio(calibre/4)
45     }
46 }
47
48 //Clase abstracta, no sera instanciada
49 class ArmaRociadora inherits ArmaRecargable
50 {
51     method dispararA(objetivo)
52     {
53         cargador = cargador - self.descargarPorRafaga()
54         self.causarEfecto(objetivo)
55     }
56     //Metodo Abstracto ya que no tiene la logica
57     method causarEfecto(objetivo)
58     method descargarPorRafaga() = 20
59 }
60 class LanzaLlamas inherits ArmaRociadora
61 {
62     override method causarEfecto(objetivo)
63     {
64         objetivo.prendidoFuego(true)
```

```
65      }
66  }
67
68  class MataFuego inherits ArmaRociadora
69  {
70      override method causarEfecto(objetivo)
71      {
72          objetivo.prendidoFuego(false)
73      }
74  }
75
76  class Sellador inherits ArmaRociadora
77  {
78      override method causarEfecto(objetivo)
79      {
80          objetivo.salud( objetivo.salud() * 1.1)
81      }
82
83      override method descargarPorRafaga() = 25
84  }
```

6. Herencia vs Composicion ->Video 22 Youtube

6.1. Wollok: Juego Por La Horda

```
//Clase Abstracta, nunca de instancia, solo sirve para que hereden de ella
class Personaje
{
    const property fuerza = 100
    const property inteligencia = 100
    var property rol

    method potencialOfensivo() = 10 * fuerza + rol.potencialOfensivoExtra()

    method esGroso() = self.esInteligente() or  rol.esGroso(self)
    method esInteligente()
}

class Humano inherits Personaje
{
    override method esInteligente() = inteligencia > 50
}

class Orco inherits Personaje
{
    override method esInteligente() = false
    override method potencialOfensivo() = super() * 1.1
}

//Roles
object guerrero
{
    method potencialOfensivoExtra() = 10
    method esGroso(personaje) = personaje.fuerza() > 50
}

object brujo
{
    method potencialOfensivoExtra() = 0
    method esGroso(personaje) = true
}

class Cazador
{
    var property mascota
    method potencialOfensivoExtra() = mascota.potencialOfensivo()
    method esGroso(personaje) = mascota.esLongeva()
}

class Mascota
{
    const property fuerza = 20
    const property edad = 5
    const property tieneGarras = true

    method potencialOfensivo() = if (tieneGarras) fuerza * 2 else fuerza
```

```
52      method esLongeva() = edad > 10
53  }
54
55  //Zonas
56
57  class Ejercito
58  {
59      const property miembros = []
60      method potencialOfensivo() = miembros.sum{soldado=>soldado.
            potencialOfensivo()}
61      method invadir(zona)
62      {
63          if(zona.potencialDefensivo() < self.potencialOfensivo())
64          {
65              zona.seOcupadaPor(self)
66          }
67      }
68  }
69
70  class Zona
71  {
72      var property habitantes
73      method potencialDefensivo() = habitantes.potencialOfensivo()
74      method seOcupadaPor(ejercito)
75      {
76          habitantes = ejercito
77      }
78  }
79
80  class Ciudad inherits Zona
81  {
82      override method potencialDefensivo() = super() + 300
83  }
84
85  class Aldea inherits Zona
86  {
87      const maximoHabitantes = 50
88      override method seOcupadaPor(ejercito)
89      {
90          if (ejercito.miembros().size() > maximoHabitantes)
91          {
92              const nuevosHabitantes = ejercito.miembros().
93                    sortedBy{uno,otro=>uno.potencialOfensivo() > otro.
                          potencialOfensivo()}.take(10)
94              super(new Ejercito(miembros = nuevosHabitantes))
95              ejercito.miembros().removeAll(nuevosHabitantes)
96          }
97          if (ejercito.miembros().size() <= maximoHabitantes)
98              super(ejercito)
99       }
100 }
```

7. Manejo de Errores ->Video 23 Youtube

7.1. Wollok: Clase Impresora

```
import cabezal.*
import cartucho.*

class Impresora
{
   const cabezal
   const cabezalAux
   var property ocupada

   method trazar(recorrido)
   {

   }
   method mostrarEnPantalla(mensaje)
   {

   }

   method imprimir(documento)
   {
      if (ocupada) throw new NoPuedoImprimirException()

      ocupada = true

      try
      {
         cabezal.eyectar(documento.tinta())
         self.trazar(documento.recorrido())
      }
      catch error: SinCargaException
      {
         cabezalAux.eyectar(documento.tinta())
      }then always
      {
         ocupada = false
      }
   }
}

class NoPuedoImprimirException inherits DomainException
{

}
```

7.2. Wollok: Clase Cabezal

```
import cartucho.*

class Cabezal
```

```
{
    const eficiencia
    const cartucho

    method liberar()
    {

    }

    method eyectar(cantidad)
    {
        cartucho.extraer(1/eficiencia * cantidad)
        self.liberar()
    }
}
```

7.3. Wollok: Clase Cartucho

```
class Cartucho
{
    var property carga = 100

    method extraer(cantidad)
    {
        if(carga < cantidad)
            throw new SinCargaException(carga = carga)
        carga -= cantidad
    }
}

class SinCargaException inherits DomainException
{
    const property carga
}
```

8. Parcial Monetizaciones->Video 24 Youtube

8.1. Wollok: Contenidos

```
import monetizaciones.*
class Contenido
{
    const property titulo
    var property vistas = 0
    var property ofensivo = false
    var property monetizacion

    //Nuevo Setter de Monetizacion
    method monetizacion(_monetizacion)
    {
        if(not _monetizacion.puedeAplicarseA(self))
            throw new DomainException(message = "Este contenido no soporta la
                forma de monetizacion")
        monetizacion = _monetizacion
    }
    override method initialize()
    {
        if(not monetizacion.puedeAplicarseA(self))
            throw new DomainException(message = "Este contenido no soporta la
                forma de monetizacion")
    }

    method recaudacion() = monetizacion.recaudacionDe(self)
    method puedeVenderse() = self.esPopular()
    method esPopular()
    method recaudacionMaximaDePublicidad()
    method puedeAlquilarse(contenido)
}

class Video inherits Contenido
{
    override method esPopular() = vistas > 10E3
    override method recaudacionMaximaDePublicidad() = 10E3
    override method puedeAlquilarse(contenido) = true
}

const tagsDeModa = ["objetos","pdep","serPeladoHoy"]

class Imagen inherits Contenido
{
    const property tags = []
    override method esPopular() = tagsDeModa.all{tag => tags.contains(tag)}
    override method recaudacionMaximaDePublicidad() = 4E3
    override method puedeAlquilarse(contenido) = false
}
```

8.2. Wollok: Monetizaciones

```
object publicidad
{
   method recaudacionDe(contenido) = ( 0.05 * contenido.vistas() +
      if( contenido.esPopular() ) 2000 else 0).
      min(contenido.recaudacionMaximaDePublicidad())

   method puedeAplicarseA(contenido) = not contenido.ofensivo()
}

class Donacion
{
   var property donaciones = 0
   method recaudacionDe(contenido) = donaciones
   method puedeAplicarseA(contenido) = true
}

class Descarga
{
   const property precio
   method recaudacionDe(contenido) = contenido.vistas() * precio
   method puedeAplicarseA(contenido) = contenido.puedeVenderse()

}

class Alquiler inherits Descarga
{
   override method precio() = 1.max(super())
   override method puedeAplicarseA(contenido) = super(contenido) and
      contenido.puedeAlquilarse()

}
```

8.3. Wollok: Usuarios

```
object usuarios
{
   const todosLosUsuarios = []

   method emailsDeUsuariosRicos() = todosLosUsuarios.
      filter{usuario=>usuario.verificado()}.
      sortedBy{uno,otro=>uno.saldoTotal() > otro.saldoTotal()}.
      take(100).
      map{usuario=>usuario.email()}

   method cantidadDeSuperUsuario() = todosLosUsuarios.
      count{usuario=>usuario.esSuperUsuario()}
}

class Usuario
{
   const property nombre
   const property email
   const property verificado = false
```

```
20      const contenidos = []
21
22      method saldoTotal() = contenidos.
23          sum{contenido=>contenido.recaudacion()}
24      method esSuperUsuario() = contenidos.
25          count{contenido=>contenido.esPopular()} >= 10
26
27      method publicar(contenido,monetizacion)
28      {
29          if(monetizacion.puedeAplicarseA(contenido))
30              throw new DomainException(message="El contenido no soporta la
                    forma de monetizacion")
31          contenido.monetizacion(monetizacion)
32          contenidos.add(contenido)
33      }
34  }
```

9. Parcial Super Computadora->Video 25 Youtube

9.1. Wollok: Computadoras

```
import modos.*
class Equipo
{
    var property modo = standard
    var property estaQuemado = false
    method estaActivo() = not estaQuemado and self.computo() > 0
    method consumo() = modo.consumoDe(self)
    method computo() = modo.computoDe(self)
    method consumoBase()
    method computoBase()
    method computoExtraPorOverclock()
    method computar(problema)
    {
        if(problema.complejidad() > self.computo())
            throw new DomainException(message="capacidad excedida")
        modo.realizoComputo(self)
    }
}

class A105 inherits Equipo
{
    override method consumoBase() = 300
    override method computoBase() = 600
    override method computoExtraPorOverclock() = self.computoBase()*0.3
    override method computar(problema)
    {
        if(problema.complejidad() < 5)
            throw new DomainException(message="Error de fabrica")
        super(problema)
    }
}

class B2 inherits Equipo
{
    const microsInstalados
    override method consumoBase() = 50 * microsInstalados + 10
    override method computoBase() = 800.min(100 * microsInstalados)
    override method computoExtraPorOverclock() = 0.3 * microsInstalados

}

class SuperComputadora
{
    const equipos = []
    var totalDeComplejidadComputada = 0

    method equiposActivos() = equipos.
        filter{equipo=>equipo.estaActivo()}

    method estaActivo() = true

```

```
52      method computo() = self.equiposActivos().
53          sum{equipo=>equipo.computo()}
54
55      method consumo() = self.equiposActivos().
56          sum{equipo=>equipo.consumo()}
57
58      method equipoActivoQueMas(criterio) = self.
59          equiposActivos().max(criterio)
60
61      method malConfigurada() =
62          self.equipoActivoQueMas{equipo=>equipo.consumo()} !=
63          self.equipoActivoQueMas{equipo=>equipo.computo()}
64
65      method computar(problema)
66      {
67          const subProblema = new Problema(complejidad = problema.complejidad
                () / self.equiposActivos().size())
68          self.equiposActivos().forEach{equipo=>equipo.computar(subProblema)}
69          totalDeComplejidadComputada += problema.complejidad()
70      }
71  }
72
73  class Problema
74  {
75      const property complejidad = 1
76  }
```

9.2. Wollok: Modos de trabajo

```
1   object standard
2   {
3       method consumoDe(equipo) = equipo.consumoBase()
4       method computoDe(equipo) = equipo.computoBase()
5       method realizoComputo(equipo) {  }
6   }
7
8   class Overclock
9   {   var property usosRestantes
10      override method initialize()
11      {
12          if(usosRestantes < 0)
13              throw new DomainException(message = "Los usos restantes deben ser
                    mayor o igual a cero")
14      }
15      method consumoDe(equipo) = equipo.consumoBase() * 2
16      method computoDe(equipo) = equipo.computoBase() +
17          equipo.computoExtraPorOverclock()
18      method realizoComputo(equipo)
19      {
20          if(usosRestantes == 0)
21          {
22              equipo.estaQuemado(true)
23              throw new DomainException(message = "Equipo quemado!")
24          }
```

```
25
26          usosRestantes--
27      }
28  }
29
30  class AhorroDeEnergia
31  {
32      var computosRealizados = 0
33      method consumoDe(equipo) = 200
34      method periodicidadDeError() = 17
35      method computoDe(equipo) = self.consumoDe(equipo) /
36          equipo.consumoBase() * equipo.computoBase()
37
38      method realizoComputo(equipo)
39      {
40          computosRealizados++
41          if(computosRealizados % self.periodicidadDeError() == 0)
42              throw new DomainException(message = "Corriendo monitor")
43      }
44  }
45
46  class APruebaDeFallos inherits AhorroDeEnergia
47  {
48      override method computoDe(equipo) = super(equipo)/2
49      override method periodicidadDeError() = 100
50  }
```

Práctica 1: Julieta en la Feria

PdeP JN - 2020 - Objetos

Queremos hacer un sistema para registrar la visita de Julieta a la feria que llegó al pueblo.

En la feria hay varios juegos, todos distintos y, cada vez que Julieta juega a uno, gana algún número de tickets, que luego puede cambiar por fabulosos premios. Algunos de los juegos que hay en la feria son:

Tiro al Blanco: En este juego los chicos le disparan a patitos de madera usando un rifle de aire comprimido. La cantidad de tickets que otorga depende de cuántos patos tire (para ser más exactos, se entrega un ticket por cada 10 puntos de puntería del participante, redondeando para arriba). Jugar este juego tensa un poco a los chicos, causandoles 3 puntos de cansancio.

Prueba de Fuerza: Este juego está equipado con un balancín y una plomada instalada en una corredera vertical, que termina en una campana. Los chicos golpean el balancín con una maza de madera y, si hacen sonar la campana, reciben 20 tickets (y si no, nada). Para poder hacer sonar la campana hace falta una fuerza de, al menos, 75 puntos. De más está decir que este juego es agotador (produce 8 puntos de cansancio cada vez que se juega).

Rueda de la Fortuna: Este es un juego puramente de azar. Los participantes hacen girar una rueda gigante que, al detenerse, indica cuántos tickets ganó. La rueda premia un número aleatorio de tickets entre 1 y 20. Cuando está bien aceitada la rueda gira suavemente y sin esfuerzo, pero cuando pasa mucho tiempo sin mantenimiento empieza a hacer fricción y causa un 1 punto de cansancio al girarla.

De Julieta sabemos que es una chica muy fuerte (tiene 80 puntos de fuerza, menos el cansancio que acumule), pero su puntería no es muy buena (apenas 20 puntos). También sabemos que llega a la feria completamente descansada y con 15 tickets que guardó del año anterior.

Se pide:

a. Modelar a Julieta y los juegos mencionados, de forma tal que podamos hacer que Julieta juegue y nos informe cuántos tickets tiene.
b. Extender el programa para saber si Julieta puede canjear alguno de los premios disponibles. La mano viene dura y en la feria quedan solamente dos premios disponibles: Un osito de peluche que cuesta 45 tickets y un taladro rotopercutor Bosch de 750w y mandril de 13mm, cuyo costo el dueño de la feria ajusta todos los días en base al precio del dólar.
c. Agregar al sistema a Gerundio, otro chico más que visita la feria. Geru es el hijo del dueño así que no necesita juntar tickets y puede canjear el premio que quiera cuando quiera. Para pensar: ¿Qué mensajes necesita entender Gerundio?

Por la horda!

Desde hace décadas, la guerra entre las facciones de la Alianza y la Horda se ha extendido por todo el mundo. En esta oportunidad, queremos construir un sistema usando el paradigma orientado a objetos que nos permita modelar la situación actual de la lucha, y quien sabe, ayudar a que ~~nunca~~ termine.

Por el momento, tenemos dos razas distintas de personajes, los orcos (miembros de La Horda) y los humanos (integrantes de La Alianza); aunque en el futuro nuevas razas pueden sumarse a la batalla, el sistema deberá poder extenderse con facilidad. De los personajes nos interesa su fuerza, inteligencia y rol.

Los roles disponibles al momento son guerrero, cazador o brujo. Cada personaje tiene un único rol, pero, si se aburren, pueden cambiarlo por otro haciendo el trámite correspondiente. Los cazadores pueden tener una mascota (un animal salvaje que el cazador supo domar), de la cual conocemos su fuerza, edad y si tiene o no garras.

Nuestros personajes no viven en soledad, es por ello que tenemos distintas localidades donde pueden encontrarse. Por un lado, existen las aldeas, pequeñas pero acogedoras y, por otro lado, ciudades grandes y ricas. Las aldeas tienen una cantidad máxima de habitantes, que depende de su tamaño; sólo las ciudades pueden contener cualquier cantidad de personajes.

En algunas ocasiones los personajes pueden agruparse en ejércitos para atacar una localidad.

Se pide modelar las abstracciones necesarias para soportar los siguientes requerimientos:

1) Obtener el potencial ofensivo de un personaje, el cual se calcula como la fuerza multiplicada por 10 más un cierto extra que depende del rol:
    - Guerrero: Siempre da un extra de 100.
    - Cazador: El extra depende del potencial ofensivo de su mascota. Las mascotas sin garras tienen un potencial ofensivo igual a su fuerza. Las que tienen garras duplican dicho valor.
    - Brujo: No da ningún extra.

    En el caso particular de los orcos, producto de su brutalidad innata, su potencial ofensivo es un 10% más.

2) Saber si un personaje es groso. Esto se da si es inteligente o es groso en su rol.
    Un humano se considera inteligente si su inteligencia es mayor a 50. Los orcos nunca son inteligentes.
    Un personaje es groso en su rol dependiendo de la exigencia del mismo:
    - Guerrero: Es groso si la fuerza del personaje es mayor a 50.
    - Cazador: Es groso si su mascota es longeva. Una mascota es longeva cuando su edad es mayor a 10.

- Brujo: Siempre es groso.

3) Queremos modelar la invasión a una localidad. Cuando esto sucede, el ejército invasor lucha contra los personajes que habitan la zona para ganar control de ella.

   En caso de que el potencial ofensivo total del ejército invasor supere al del defensor, la zona es desalojada y el ejército atacante pasa a ocuparla. Si el ejército es muy grande para la localidad debe dividirse en dos, quedando en la zona un nuevo ejército conformado por los 10 miembros con mayor potencial ofensivo del ejército original.

   Si esto no ocurre, la invasión no tuvo éxito y el ejército defensor permanece a cargo.
   Las ciudades poseen mejores defensas que las aldeas, con lo cual incrementan el potencial ofensivo del ejército defensor en 300.

PdeP: Plataforma de Pago

La plataforma permite subir distintos tipos de contenidos con el fin de monetizarlos, de los cuales conocemos el título y la cantidad de vistas que tuvieron. Además, cada pieza de contenido podría estar marcada como “contenido ofensivo” por su autor (o debido al pedido de otros usuarios). Ahora mismo, los tipos de contenido disponibles son **videos** e **imágenes** (de las cuales también conocemos los tags con los que el autor las etiquetó) pero en el futuro podrían aparecer más.

Los usuarios de la aplicación ingresan su nombre y email los cuales, pasados cierto tiempo, son verificados (hasta que eso ocurra, se los considera “sin verificar”).

Al subir un contenido, los usuarios eligen para el mismo una forma de monetización, la cual determina la forma en que el contenido se cotiza. El usuario puede cambiar en cualquier momento la forma de monetizar cada uno de sus contenidos, pero sólo puede aplicar una a cada uno. Cambiar la forma de monetizar hace que se pierda todo el dinero ganado por ese contenido en la forma anterior.

Las estrategias de monetización posibles son:

- **Publicidad**: El contenido se muestra al lado de un aviso publicitario. El usuario cobra 5 centavos por cada vista que haya tenido su contenido. Además los contenidos populares cobran un plus de $2000.00. Consideramos que un video es popular cuando tiene más de 10000 vistas, mientras que una imagen es popular si está marcada con todos los tags de moda (una lista de tags arbitrarios que actualizamos a mano y puede cambiar en cualquier momento).
  Ninguna publicación puede recaudar con publicidades más de cierto máximo que depende del tipo: $10000.00 para los videos y $4000.00 para las imágenes (incluyendo el plus).
  Sólo las publicaciones no-ofensivas pueden monetizarse por publicidad pero si una publicidad es marcada como ofensiva luego de elegir esta monetización puede conservarla.

- **Donaciones**: El contenido ofrece la posibilidad de hacer una donación al autor. El monto de cada donación depende de cada donador y puede acumularse cualquier cantidad. Todos los contenidos pueden ser monetizados por donaciones.

- **Venta de Descarga**: El contenido puede ser descargado luego de que el comprador pague un precio fijo, elegido por el vendedor. El valor mínimo de venta es de $5.00 y se cobra por cada vista. Sólo los contenidos populares pueden acceder a esta forma de monetización.

Se pide:

1. Calcular el total recaudado por un contenido

2. Hacer que el sistema permita realizar las siguientes consultas:

    a. Saldo total de un usuario, que es la suma total de lo recaudado por todos sus contenidos.

    b. Email de los 100 usuarios verificados con mayor saldo total.

    c. Cantidad de super-usuarios en el sistema (usuarios que tienen al menos 10 contenidos populares publicados).

3. Permitir que un usuario publique un nuevo contenido, asociándolo a una forma de monetización.

4. Aparece un nuevo tipo de estrategia de monetización: El **Alquiler**. Esta estrategia es muy similar a la venta de descargas, pero los archivos se autodestruyen después de un tiempo. Los alquileres tienen un precio mínimo de $1.00 y, además de tener todas las restricciones de las ventas, los alquileres sólo pueden aplicarse a videos.

5. Responder sin implementar:

    a. ¿Cuáles de los siguientes requerimientos te parece que sería el más fácil y cuál el más difícil de implementar en la solución que modelaste? Responder relacionando cada caso con conceptos del paradigma.
        i. Agregar un nuevo tipo de contenido.
        ii. Permitir cambiar el tipo de un contenido (e.j.: convertir un video a imagen).
        iii. Agregar un nuevo estado “verificación fallida” a los usuarios, que no les permita cargar ningún nuevo contenido.

    b. ¿En qué parte de tu solución se está aprovechando más el uso de polimorfismo? ¿Por qué?

PdeP: Parcial de Procesamiento

Nos piden modelar un sistema que permita llevar registro del trabajo de las Super-Computadoras de un laboratorio.

Las **super-computadoras** que tenemos que modelar son conjuntos de **equipos** independientes que se conectan entre sí para hacerse más poderosos. Tenemos, de momento, dos tipos de equipos para conectar: **A105** y **B2**.

Los equipos tipo A105 son los modelos más viejos de equipo. Tienen un consumo eléctrico base de 300 watts y producen 600 unidades de cómputo (base) cada uno.

Los equipos Tipo B2 son más modernos y están pensados para escalar. Cada uno de estos equipos se fabrica para permitir la instalación de microchips que aumentan el poder de cómputo y el consumo del equipo. El equipo tiene un consumo base de 50 watts por cada microchip instalado (más 10 watts para hacer funcionar la placa madre) y produce 100 unidades de cómputo base por microchip, hasta un máximo de 800. La cantidad de micros que cada equipo tiene instalado puede variar y, como se fabrican a pedido, depende de cada equipo.

Además de estos dos tipos de equipos, las super-computadoras también pueden conectarse a super-computadoras más pequeñas, que siempre se consideran activas y cuya capacidad de computo y consumo se calcula a partir de los equipos instalados (ver punto 1).

Cada equipo se conecta a la super-computadora configurado para trabajar en uno de tres posibles **modos**: **Standard, Overclock y Ahorro de Energía**.

El modo de funcionamiento Standard es, como su nombre lo indica, el modo normal de trabajo. Los equipos en este modo consumen y producen sus valores base, sin más ni menos.

Los equipos en modo Overclock se configuran para forzar un mayor desempeño, a cambio de un mayor consumo y corriendo el riesgo de romper el equipo. Los equipos en este modo consumen el doble de energía, pero producen un extra de cómputo que depende del tipo de equipo: Los A105 incrementan su capacidad un 30%, mientras que los B2 la incrementan en 20 unidades por micro. Sin embargo, overclockear es peligroso: al pasar a modo overclock un equipo sólo podrá usarse cierta cantidad de veces antes de quemarse (el número exacto es arbitrario y varía cada vez se overclockea). Cada vez que la super-compudora computa, sus equipos en modo overclock son usados, si esto ocurre las veces necesarias el equipo pasa a estar **quemado**.

Por otro lado, el modo Ahorro de Energía hace que el equipo, no importa su tipo, sólo consuma 200 watts, pero su capacidad de cómputo se ve afectada de forma proporcional a la pérdida de energía (por ejemplo, un equipo con un consumo base de 400 watts pierde la mitad de su energía, por lo tanto producirá la mitad de cómputo).

El modo de cada equipo puede cambiarse a gusto en cualquier momento para adecuar a la super-computadora a una nueva tarea.

Se pide modelar el dominio descripto y desarrollar los siguientes puntos:

1) Dada una Super-Computadora, se quiere poder responder a las siguientes consultas:
    a) **equipos activos:** son los equipos conectados a la SC que no están quemados y tienen una capacidad de cómputo mayor a cero.

    b) **capacidad de computo y consumo:** este es el total de computo y consumo de todos los equipos activos.

    c) **malConfigurada:** esto ocurre cuando el equipo de la SC que más consume NO es el que más computa.

2) **computarProblema:** Dado un problema de complejidad N (o sea, que requiere N unidades de cómputo para ser resuelto), utilizar una SC para computarlo. Cuando esto ocurre la computadora divide el problema en problemas más chicos, tantos como **equipos activos** tenga conectados, de igual complejidad (Si la computadora tiene M equipos, cada subproblema tendrá una complejidad de N/M).

    Cada equipo activo intenta procesar un sub-problema de acuerdo a los siguientes criterios:
    - Un equipo que intenta computar más que su capacidad de cómputo, falla.

    - Por un error de construcción, los equipos A105 no pueden computar problemas de complejidad menor a 5. Si lo intentan, hacen mal el cálculo y fallan.

    - Los equipos en modo overclock pueden quemarse al tratar de computar (ver más arriba). Si el equipo pasa a estar quemado al computar el problema el cómputo falla.

    - Los equipos en modo ahorro de energía corren un monitor de consumo periodicamente y esto causa que fallen 1 de cada 17 intentos de computar.

    Luego de resolver el problema, la computadora incrementa un contador interno que recuerda la cantidad total de complejidad que ha resuelto, con fines de auditoría.

3) Aparece ahora un nuevo tipo de modo: **A Prueba de Fallos**. Este modo es una versión mejor del modo **Ahorro de Energía**, que ofrece la mitad de capacidad de cómputo pero sólo falla por monitorear consumo una vez cada 100 computos.

Maestros PdePizzeros 

Nos pidieron escribir un programa que permita analizar y predecir los resultados de un concurso de pizzeros.

Cada participante del concurso tiene que preparar una pizza que será evaluada por un distinguido jurado. Aquel participante que reciba un mejor puntaje, gana.

La manera en la que cada pizzero prepara una pizza es parecida:

- Comienzan preparando una prepizza, que es una pizza de 500g, con tomate como único ingrediente y cuyo grado de cocción es 0.3.
- Le agrega el ingrediente muzzarella.
- Le dan su toque, que es algo que depende de cada pizzero.
- La cocinan por 10 minutos. Por cada minuto de cocción, la pizza aumenta su grado de cocción en (10/masa de la pizza al comenzar a cocinarla) y también pierde un 1% de su masa inicial (con la que comenzó “de fábrica”).

Por ejemplo, si se cocina una de 500g que tenía 500g inicialmente por 10 minutos, termina con 450g y su grado de cocción aumenta en 0.2.

Queremos representar a los siguientes pizzeros:

- Carla, cuyo toque es agregarle provolone y luego cocinarla durante 1 minuto.
- Facu, cuyo toque depende de el humor con el que se encuentre:
- Si ese día se siente arriesgado, es agregar ananá.
- Si ese día se siente conservador, es agregar orégano.
- los pizzeros de la pizzería “La Marítima”, cuyo toque es agregar su ingrediente favorito (cada pizzero tiene el suyo) y también agregarle el ingrediente del día del local (que se quiere poder cambiar día a día y es el mismo para todos). Además, en La Marítima los pizzeros hacen la prepizza de 650g en vez de 500g.

Hay ciertas características de la pizza que pueden importar al jurado, por ejemplo:

- Que no esté cruda ni quemada. Una pizza está cruda si su grado de cocción es menor a 0.4 y está quemada si su grado de cocción es mayor a 1.0.
- Que tenga cierto ingrediente.
- Que esté cargada, es decir, que tenga más de una vez el mismo ingrediente

El jurado está formado por un grupo de críticos. Todo crítico veta pizzas que estén crudas o que tengan un ingrediente que no les gusta.

Algunos ejemplos de jurados son:

- Zeff: no le gusta el atún, y el puntaje que le pone a la pizza se calcula como la masa final de la pizza sobre la masa inicial con la que se hizo.
- Anton: no le gustan los morrones. Además de vetar las pizzas que suelen vetar los críticos, también veta pizzas con menos de 400g de masa. Su forma de puntuar pizzas es un poco más particular: Parte de un puntaje perfecto (1.0) y a eso le resta 0.5 si la pizza está quemada, le suma 0.2 si la pizza está cargada, y le resta 0.1 por cada 100 gramos por encima de 500g que tenga la pizza.
- Contrera: no le gusta el queso azul, y su ingrediente favorito es el ananá. Este crítico siempre toma a otro crítico de referencia y le lleva la contra. Su puntaje será:

1 - el puntaje del otro crítico + 0.2 si la pizza tiene su ingrediente favorito.

Luego, cada uno tiene un criterio según el cual puntúa la pizza. Nota: el puntaje del criterio siempre tiene como máximo 1.0, si la fórmula aplicada a una pizza pudiese superar ese valor, debe limitarse a 1.0 de todas formas. De la misma manera, nunca puede bajar de 0.0.

Requerimientos:

1. Queremos poder pedir a un pizzero que cocine una pizza.
2. Sobre una pizza, queremos poder saber:

1. Si está cruda.
2. Si está quemada.
3. Si está cargada.

3. Jurados

1. Queremos poder preguntarle a un jurado si vetaría cierta pizza.
2. Queremos poder pedirle a un jurado el puntaje para cierta pizza.

4. Concurso

1. En un concurso pueden participar varios pizzeros y son evaluados por un equipo de críticos.

Dar un ejemplo de cómo se crearía un concurso en el que participan:

- como pizzeros: Facu, Carla y 2 pizzeros de La Marítima (de uno de ellos su ingrediente favorito es el atún y de otro los morrones).
- como jurados: Zeff y Contrera (que está tomando como referencia a Zeff).

1. La forma en la que funciona el concurso es la siguiente:

- Cada pizzero prepara una pizza.
- Se hace una ronda de vetos donde los críticos eliminan a todos los pizzeros cuyas pizzas fueron vetadas.
- En la ronda final, se comparan las pizzas comparando la sumatoria de puntajes que le dieron los críticos. Aquel pizzero cuya pizza sea la mejor puntuada gana el concurso.

1. Queremos, dado un concurso, obtener al pizzero ganador. Siempre tiene que ganar uno sí o sí, por lo que si llegase a haber empate, o si todos los pizzeros fueron vetados, no hay ganador posible y queremos informar esto de alguna manera.

Por ejemplo, en el concurso de ejemplo del punto a:

- Gana Facu si se siente arriesgado y el ingrediente del día en La Marítima son las aceitunas.

- Gana el 2° pizzero de La Marítima si Facu se siente conservador y el ingrediente del día en La Marítima es el ananá.
- No hay ganador posible si Facu se siente conservador y el ingrediente del día en La Marítima es el atún.