

RESUMEN PARADIGMA ORIENTADO A OBJETOS
FACULTAD REGIONAL BUENOS AIRES
UNIVERSIDAD TECNOLÓGICA NACIONAL

PARADIGMAS DE PROGRAMACIÓN – *K2032*

Trabajos Prácticos

PAZ PORTILLA, José Miguel 2028244
jpazportilla@frba.utn.edu.ar

6 de diciembre de 2023

Índice

1. Introducción a Objetos ->Video 17 Youtube	1
1.1. Problema: La Golondrina Pepita	1
1.1.1. Resolución: La Golondrina Pepita	1
1.1.2. Wollok: La Golondrina Pepita	1
1.2. Problema: La entrenadora de aves Emilia	2
1.2.1. Resolución: La entrenadora de aves Emilia	2
1.2.2. Wollok: La entrenadora de aves Emilia	2
1.3. Emilia entrena a Pepita	2
1.3.1. Wollok: Emilia entrena a Pepita	2
1.4. Conceptos clave de la programación orientada a objetos	3
1.5. Otra ave: El hancón pepote	3
1.6. Wollok: El hancón pepote	3
1.7. Wollok-Test: Emilia entrena a Pepote	4
1.8. Diseño de métodos para favorecer el polimorfismo entre objetos	5
1.9. Otro entrenado Ramiro	5
1.10. Resolución: Otro entrenado Ramiro	5
1.11. Wollok: Otro entrenador Ramiro	5
1.12. Wollok-Test: Ramiro entrena a Pepita y Pepote, con buen y mal humor	6
2. Práctica: La Feria ->Video 18 Youtube	9
2.1. Wollok: Jugadores de la feria	9
2.2. Wollok: Juegos de la feria	9
2.3. Wollok: Premio de la feria	10
2.4. Wollok-Test: Julieta juega en la feria	10
3. Clases como plantillas para crear objetos ->Video 19 Youtube	12
3.1. Wollok: Clase Golondrina	12
3.2. Wollok-Test: Clase Golondrina	12
4. Colecciones y Bloques ->Video 20 Youtube	13
4.1. Lista	13
4.2. Wollok: Clase Vaca	13
4.3. Wollok: Clase Cabra	14
4.4. Wollok: Clase Corral	14
4.5. Wollok-Test: Prueba en lista de vacas	14
5. Herencia ->Video 21 Youtube	16
5.1. Wollok: Clase Tanque	16
5.2. Wollok: Clase Arma	16
6. Herencia vs Composicion ->Video 22 Youtube	19

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 características 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 mediante 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 Figura 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 tanto la energia final resulta 100+50-20+50=180J
18          */
19     }
20 }

```

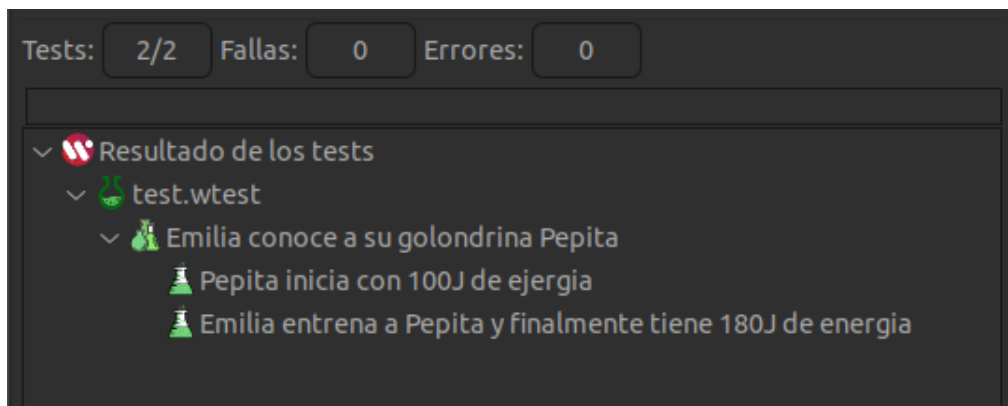


Figura 1. Test unitario de pepita siendo entrenada por emilia.

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 interesa 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 vuela. 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

```
1 object pepote
2 {
3     var volado = 0
4     var comido = 0
5
6     method vola (kilometros)
7     {
8         volado = volado + kilometros
9     }
10    method come(gramos)
11    {
12        comido = comido + gramos
13    }
14    method energia () = 255 + comido **2 - volado / 5
15 }
```

1.7. Wollok-Test: Emilia entrena a Pepote

```
1 import pepote.*
2 import emilia.*
3
4 describe "Emilia conoce a su Halcón Pepote"
5 {
6     test "Pepote inicia con 255J de ejergia"
7     {
8         assert.equals(255, pepote.energia())
9     }
10    test "Emilia entrena a Pepote y finalmente tiene 353J de energia"
11    {
12        emilia.entrena(pepote)
13        assert.equals(353, pepote.energia())
14        /*Ya que luego de comer 5g
15        * luego de volar 10km
16        * y volver a comer 5g, sus atributos quedan
17        * volado 10km y comido 10g, por lo tanto
18        * su energia es 255 + 10**2 - 10/5 =
19        * = 255 + 100 - 2 = 353
20        */
21    }
22 }
```

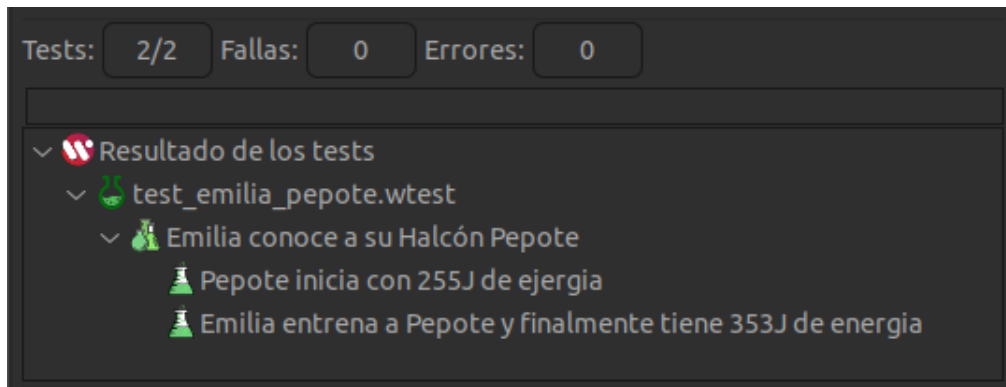


Figura 2. Test unitario de pepote siendo entrenado por emilia.

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 entiende 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
11         horasDormidas
12     method horasDormidas (_horasDormidas)
13     {
14         horasDormidas = _horasDormidas
15     }
16
17     //Retorna true si horasDormidas es mayor o igual a 8, sino retorna
18         false
19     method estaDeBuenHumor () = horasDormidas >= 8
20
21     method entrena (ave)
22     {
23         //Si estaDeBuenHumor es true, asigna 15 a la constante distancia,
24             sino asigna 30
25         const distancia = if ( self.estaDeBuenHumor() ) 15 else 30
26         //Hace volar a ave la distancia que establecio previamente
27         ave.vola(distancia)
28     }
29 }

```

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
12         ramiro un dia que tiene mal humor termina con 40J de energia"
13     {
14         assert.equals(100,pepita.energia())
15         ramiro.entrena(pepita)
16         /*
17          * Ramiro hace volar a pepita 30km, su energia disminuye en 30*2=60J
18          *
19          * Por lo tanto luego de ser entrenada por ramiro termina con
20          * 100-60=40J
21          */
22         assert.equals(40,pepita.energia())
23     }
24     test "Pepote inicialmente tiene 255J de energia, si es entrenada por
25         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
41   ramiro un dia que tiene buen humor termina con 70J de energia"
42   {
43     ramiro.horasDormidas(8)
44     assert.equals(100, pepita.energia())
45     ramiro.entrena(pepita)
46     /*
47      * Ramiro hace volar a pepita 15km, su energia disminuye en 15*2=30J
48      * Por lo tanto luego de ser entrenada por ramiro termina con
49      * 100-30=70J
50      */
51     assert.equals(70, pepita.energia())
52   }
53   test "Pepote inicialmente tiene 255J de energia, si es entrenada por
54   ramiro un dia que tiene buen humor termina con 252J de energia"
55   {
56     ramiro.horasDormidas(8)
57     assert.equals(255, pepote.energia())
58     ramiro.entrena(pepote)
59     /*
60      * Ramiro hace volar a pepote 15km, entonces ha volado 15km y
61      * por ende su energia es 255 - 15/5 = 252J
62      */
63     assert.equals(252, pepote.energia())
64   }
65 }
```

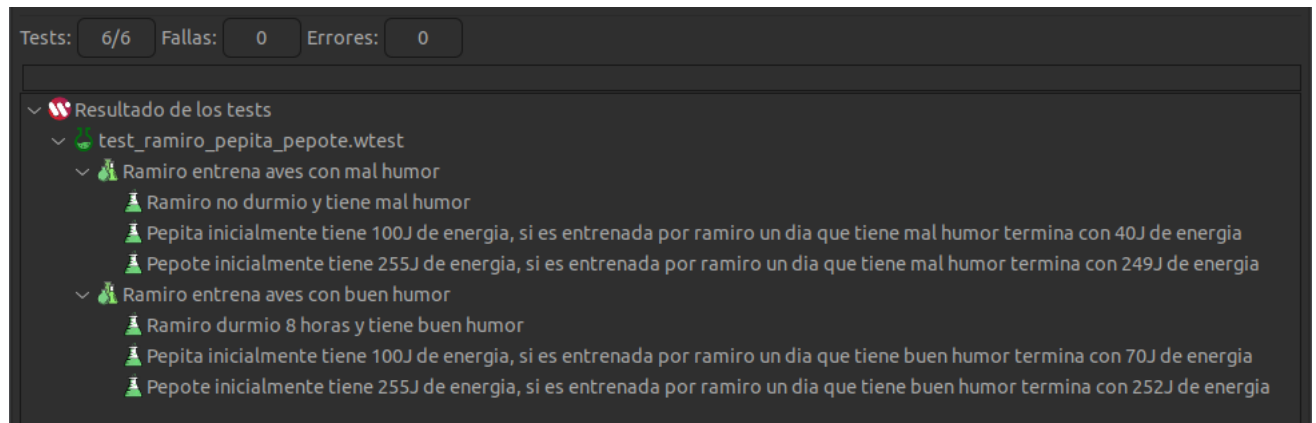


Figura 3. Test unitario de ramiro entrenado con buen y mal humor a pepita y pepote.

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

```
1  import juegos.*
2  import premios.*
3
4  object julieta
5  {
6      //Cantidad de Tickets del año anterior, puede ser modificado
7      var property tickets = 15
8      //Esta completamente descansada
9      var cansancio = 0
10
11     method punteria() = 20
12     method fuerza() = 80 - cansancio
13
14     //Cada vez que juega algun juego, cambian el valor de sus atriburos
15     //tickets y cansancio
16     method jugar ( juego )
17     {
18         tickets = tickets + juego.ticketsGanados (self)
19         cansancio = cansancio + juego.cansancioQueProduce ()
20     }
21     method puedeCanjear ( premio ) = tickets >= premio.costo()
22 }
23 object gerundio
24 {
25     method jugar() { }
26     method puedeCanjear ( premio ) = true
27 }
```

2.2. Wollok: Juegos de la feria

```
1  object tiroAlBlanco
2  {
3      method ticketsGanados (jugador) = ( jugador.punteria() / 10 ).roundUp()
4      method cansancioQueProduce () = 3
5  }
6
7  object pruebaDeFuerza
8  {
9      method ticketsGanados (jugador) = if (jugador.fuerza() > 75) 20 else 0
10     method cansancioQueProduce () = 8
11 }
12
13
14 object ruedaDeLaFortuna
15 {
```

```

16     var property aceptada = 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 }

```

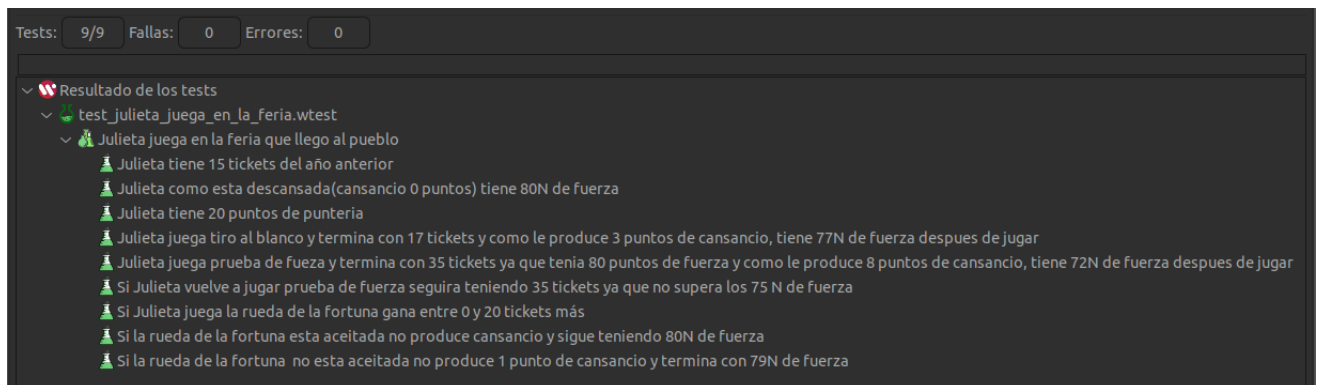


Figura 4. Test unitario julieta juega en la feria.

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. Las clases definen atributos y proveen métodos. 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

```
1  class Golondrina
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     method energia() = energia
16 }
```

3.2. Wollok-Test: Clase Golondrina

```
1  import golondrina.*
2  describe "Clase Golondrina"
3  {
4      const pepita = new Golondrina()
5      const pepito = new Golondrina(energia = 180)
6      test "La energia de la golondrina pepita es 100"
7      {
8          assert.equals(100, pepita.energia())
9      }
10     test "La energia de la golondrina pepito es 180"
11     {
12         assert.equals(180, pepito.energia())
13     }
14 }
```

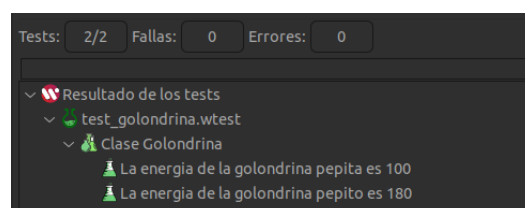


Figura 5. Test unitario clases golondrina.

4. Colecciones y Bloques -> Video 20 Youtube

4.1. Lista

Las listas se representan 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

```
1  class Vaca
2  {
3      var property estaContenta = true
4      var property litrosDeLeche = 100
5      method ordeniar()
6      {
7          litrosDeLeche = litrosDeLeche - 20
8      }
9  }
```

4.3. Wollok: Clase Cabra

```

1  class Cabra
2  {
3      var property estaContenta = true
4      var property litrosDeLeche = 50
5      method ordeniar()
6      {
7          litrosDeLeche = litrosDeLeche - 30
8      }
9  }

```

4.4. Wollok: Clase Corral

```

1  import vaca.*
2  import cabra.*
3
4  class Corral
5  {
6      const property ordeniables = []
7
8      method lecheDisponible() = ordeniables.filter{ordeniable=>ordeniable.
          estaContenta()}.sum{ordeniable=>ordeniable.litrosDeLeche()}
9      method todasContentas() = ordeniables.all{ordeniable => ordeniable.
          estaContenta()}
10     method ordeniar()
11     {
12         ordeniables.forEach{ ordeniable => if(ordeniable.estaContenta())
            ordeniable.ordeniar() }
13     }
14 }

```

4.5. Wollok-Test: Prueba en lista de vacas

```

1  import vaca.*
2  describe "test de Lista de vacas"
3  {
4      const rosita = new Vaca()
5      const petunia = new Vaca(estaContenta = false)
6      const vacas = [rosita, petunia]
7      test "Hay 2 vacas"
8      {
9          assert.equals(2,vacas.size())
10     }
11     test "Agrego una nueva vaca, por ende ahora hay 3 vacas"
12     {
13         vacas.add(rosita)
14         assert.equals(3,vacas.size())
15     }
16     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
29     elementos pero la lista de vacas sigue teniendo 2 vacas"
30 {
31     const tristes = vacas.filter{vaca=>!vaca.estaContenta()}
32     tristes.remove(petunia)
33     assert.equals(0,tristes.size())
34     assert.equals(2,vacas.size())
35 }
36 test "rosita y petunia tienen 100L de leche"
37 {
38     assert.equals(100,rosita.litrosDeLeche())
39     assert.equals(100,petunia.litrosDeLeche())
40 }
41 test "Si ordeño a las vacas tristes, petunia queda con 80L en la lista
42     de vacas original y rosita con 100L"
43 {
44     const tristes = vacas.filter{vaca=>!vaca.estaContenta()}
45     assert.equals(1,tristes.size())
46     tristes.forEach{ vaca => vaca.ordeniar() }
47     assert.equals(100,rosita.litrosDeLeche())
48     assert.equals(80,petunia.litrosDeLeche())
49 }
50 }
```

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

```
1  import arma.*
2
3  class Tanque
4  {
5      const armas = []
6      const tripulantes = 2
7      var property salud = 1E3
8      var property prendidoFuego = false
9
10     method emiteCalor() = prendidoFuego or tripulantes > 3
11
12     method sufrirDanio(danio)
13     {
14         salud = salud - danio
15     }
16
17     method atacar(objetivo)
18     {
19         armas.anyOne().dispararA(objetivo)
20     }
21 }
22
23 class TanqueBlindado inherits Tanque
24 {
25     const blindaje = 200
26
27     override method emiteCalor() = false
28
29     override method sufrirDanio(danio)
30     {
31         if(danio < blindaje)
32             super(danio - blindaje)
33     }
34 }
```

5.2. Wollok: Clase Arma

```
1  class Misil
2  {
3      const potencia
4      var agotada = false
5
6      method dispararA(objetivo)
7      {
8          agotada = true
9      }
10 }
```

```

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

```
1 //Clase Abstracta, nunca de instancia, solo sirve para que hereden de ella
2 class Personaje
3 {
4     const property fuerza = 100
5     const property inteligencia = 100
6     var property rol
7
8     method potencialOfensivo() = 10 * fuerza + rol.potencialOfensivoExtra()
9
10    method esGroso() = self.esInteligente() or rol.esGroso(self)
11    method esInteligente()
12 }
13
14 class Humano inherits Personaje
15 {
16     override method esInteligente() = inteligencia > 50
17 }
18
19 class Orco inherits Personaje
20 {
21     override method esInteligente() = false
22     override method potencialOfensivo() = super() * 1.1
23 }
24
25 //Roles
26 object guerrero
27 {
28     method potencialOfensivoExtra() = 10
29     method esGroso(personaje) = personaje.fuerza() > 50
30 }
31
32 object brujo
33 {
34     method potencialOfensivoExtra() = 0
35     method esGroso(personaje) = true
36 }
37
38 class Cazador
39 {
40     var property mascota
41     method potencialOfensivoExtra() = mascota.potencialOfensivo()
42     method esGroso(personaje) = mascota.esLongeva()
43 }
44
45 class Mascota
46 {
47     const property fuerza = 20
48     const property edad = 5
49     const property tieneGarras = true
50
51     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().
                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 }

```

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.

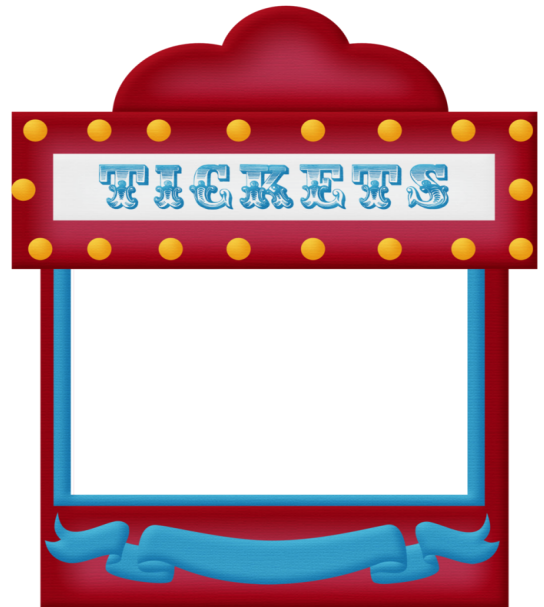
Prueba de Fuerza: Este juego está equipado con un balancín y una plomada instalada en una corredera vertical, que termina en una campana. Los chicos golpean el balancín con una maza de madera y, si hacen sonar la campana, reciben 20 tickets (y si no, nada). Para poder hacer sonar la campana hace falta 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:

- Modelar a Julieta y los juegos mencionados, de forma tal que podamos hacer que Julieta juegue y nos informe cuántos tickets tiene.
- 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.
- 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.