Programación II - TP1 1er Cuatrimestre 2018

Fecha de presentación: 15/5/18
Fecha de entrega por mail: 29/5/18

Requerimientos técnicos:
Grupos de 1 o 2 personas
Se debe utilizar al menos una vez iteradores y stringbuilder (Tecnologías java).
Además de pasar el junit suministrado en el TP, la cátedra testeará los ejercicios con otro junit adicional, por lo que se recomienda armar un junit propio para probar, antes de entregar el TP.
Se debe escribir el Irep del ej1 en el informe de no más de una página.

Ejercicio 1 Diseño

a) Se desea modelar el juego del Jenga para dos jugadores.
La dinámica del juego es la siguiente

Un ***Jenga*** se inicializa con n niveles {0…n-1} con n > 0 y con tres piezas por nivel.
Los jugadores quitan piezas de manera alternada hasta que el Jenga se cae.

El juego es automático, no se va a realizar una interfaz de usuario, se simulan las jugadas de cada jugador (¡esto no significa que se deba diseñar un TAD jugador solo por este requerimiento!)

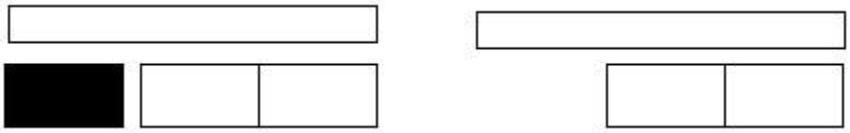
Cada vez que un jugador quita una pieza, esta pieza se agrega en la cima(nivel n-1) del Jenga.
El agregado es automático, de manera que cuando se completa un nivel, el Jenga crea un nivel nuevo de manera transparente para el usuario.

Solo se puede quitar una pieza de los niveles 0...n-2.
El jenga debe mantener la altura de manera consistente y forzar el invariante del juego.

Según la pieza que se quite existe una diferente probabilidad(%) de perder (i.e. que la torre se caiga). Asumiendo un porcentaje máximo de 100 para cada regla.

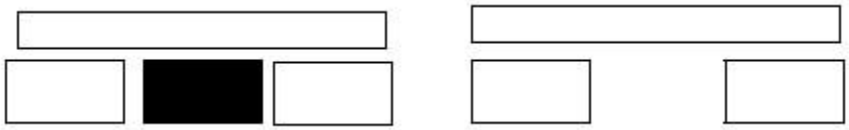
Ejemplo para un Jenga de 10 niveles

Para el caso 1: Pieza del costado, quedan dos



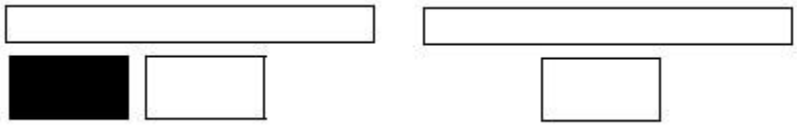
Regla: 1% * cantidad de niveles arriba de esa pieza
Para el ejemplo: 10%

Para el caso 2: Pieza del medio, quedan dos



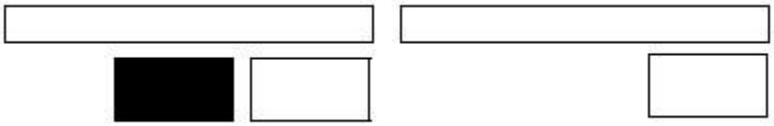
0%

Para el caso 3: Pieza del costado, queda una



Regla: 5% * cantidad de niveles arriba de esa pieza
Para el ejemplo: 50%

Para el caso 4: Pieza del medio, quedan una



Regla 100%

Agregar otras reglas al Irep si hace falta.

Modelar al TAD Jenga

Escribir el Irep
Justificar los TAD o clases soporte. Es obligatorio que al menos haya otro TAD que no sea una clase de java, que no sea una "cascara" que tan solo enmascare una estructura*.

Implementar el TAD

Debe ser consistente con el Irep elegido
Debe respetar la interfaz propuesta por la cátedra
Debe cumplir satisfactoriamente el junit de la cátedra

*Por ejemplo
Class Jenga
ArrayList<?> estructura
…

No se considera una solución de diseño válida, porque modelar este problema en un solo Tad. Tiene mucho acoplamiento, como se explicó en las teóricas.

Utilizar lineamientos de diseño que consideren las nociones de *Cohesión* y *Acoplamiento* explicadas en la teórica.
Se puede utilizar como guia el siguiente ejemplo:
https://prog2-ungs.github.io/codigo/PPTLSv2.pdf

Caso de uso de ejemplo1 "modo automático"

```
Jenga unJenga = new Jenga(20,"jug0", "jug1")  // Jenga de 20 niveles
0..19

While unJenga.ganador() <>
      unJenga.jugar()  // juegan los dos jugadores
      system.out.println(unJenga)

system.out.println(unJenga.ganador())
```

- jugar() debe mantener el Jenga consistente.
- "`system.out.println(unJenga)`" invoca a `unJenga.toString()`, que debería generar un string consistente en un "resumen" de todos los niveles.

Caso de uso de ejemplo2 "modo semi automático"

```
Jenga unJenga = new Jenga(20,"jug0", "jug1")  // Jenga de 20 niveles
0..19
Nivel ni
While unJenga.ganador() <> ""

      //unJenga.quitar(2,0)    quita la pieza 0 del nivel 2*

      ni = unJenga.primerNivelPosible()**
      if (ni ¡= null)
            unJenga.quitar( ni , unJenga.piezaRecomendada(ni))***

      system.out.println(unJenga)

system.out.println(unJenga.ganador())
```

*Si la pieza o el nivel no existen, debe arrojar una excepción
La interfaz mínima es la descrita en los ejemplos 1 y 2; los jUnit utilizaran dicha interfaz.
No es necesario agregar más métodos, en caso de hacerlo justificar y consultar en la clase de consulta del TP.

**¿Cuantas piezas tiene que tener un nivel para ser elegible para quitar piezas?

**Notar, que internamente, luego de quitar una pieza hay que agregarla en el último nivel o agregar un nivel nuevo. ¿Quien se debería encargar de esto?; hablar de esto en Irep

b) Acomplamiento y Cohesión

En la práctica a veces se sacrifica algo del diseño por cuestiones de implementación.
Explicar, si existe en su código, un ejemplo de *Acoplamiento* o de de falta de *Cohesión*.

Va un ejemplo del código fuente de la cátedra:
"La pieza no será unTAD para nuestro código fuente.
Sera representada por un Integer por carecer de otras propiedades relevantes (además de su ID) para el modelo del problema.
Notar que la pieza del Jenga tiene tres dimensiones pero elegimos no modelar ese aspecto"

-¡No significa que si modelan la pieza este mal!
-¡Dar un ejemplo distinto para el ejercicio b)!

Ejercicio 2 Árboles

Implementar 3 de los siguientes métodos del TAD ABB<Integer> extends AB<Integer>. Implementar también los métodos auxiliares o privados.

Sea abb una instancia de ABB<Integer>

a) boolean balanceado(): que devuelve verdadero si abb está balanceado y falso en caso contrario.

b) void rebalancear(): que dado cualquier abb, lo modifica para que balanceado() devuelve verdadero.

c) Integer iesimo(int i): que devuelve el iesimo nodo, considerando el recorrido ***inorden***.
Debe hacerse sin utilizar estructuras auxiliares.
Ayuda: Justificar respecto de la cantidad de veces que se recorre cada nodo.

d) boolean Irep(): que devuelve verdadero si la cantidad de nodos "alcanzables" difiere de la cantidad de nodos reales.
Ayuda: Modificar la implementación de ABB, agregar la variable int cantNodos como variable de instancia del ABB.
Luego, comparar la cantidad de nodos alcanzables con esa variable.

En todos los ítems se debe justificar la complejidad de la solución elegida.

Apéndice I: Condiciones de entrega y tutorial de cómo instalar Junit

https://prog2-ungs.github.io/tp/entregas

Apéndice II: Test obligatorio ej1

```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;


public class TestEj1 {


	private Jenga jenga1, jenga2;

	@Before
	public void setUp() {
		jenga1 = new Jenga(10,"j0","j1");
		jenga2 = new Jenga(10,"jugador0","jugador1");


	}


	@Test
	public void test1() {

		int alturaInicial = jenga1.altura();

		jenga1.Jugar();
		jenga1.Jugar();
		jenga1.Jugar();

		//System.out.println(alturaInicial +","+ jenga1.altura());
		// deberia cambiar la altura
		assertTrue(alturaInicial != jenga1.altura());

	}

	@Test
	public void test2() {

		int nivel = jenga2.primerNivelPosible();

		jenga2.quitar(nivel,0);
		jenga2.quitar(nivel,1);
		jenga2.quitar(nivel,2);
```

```
            System.out.println(jenga2.ganador());
            // deberia haberse caido el jenga!
            assertTrue(!jenga2.ganador().equals(""));

      }

}
```

Apéndice III: Test obligatorio ej2

```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class TestE2 {

      ABB abBalanceado, abVacio, abDesbalanceado;

      @Before
      public void setUp() throws Exception {
            abVacio = new ABB();

            abDesbalanceado = new ABB();
            abDesbalanceado.insertar(5);
            abDesbalanceado.insertar(3);
            abDesbalanceado.insertar(1);

            abBalanceado = new ABB();
            abBalanceado.insertar(8);
            abBalanceado.insertar(3);
            abBalanceado.insertar(10);
            abBalanceado.insertar(1);
            abBalanceado.insertar(6);
            abBalanceado.insertar(4);
            abBalanceado.insertar(7);
            abBalanceado.insertar(14);
            abBalanceado.insertar(9);
      }

      @Test
```

```
	public void testBalanceado()
	{
		assertTrue(abBalanceado.balanceado());
		assertTrue(abVacio.balanceado());
		assertFalse(abDesbalanceado.balanceado());
	}

	@Test
	public void testRebalancear()
	{
		abDesbalanceado.rebalancear();
		assertTrue(abDesbalanceado.balanceado());
	}

	@Test
	public void testIesimo()
	{
		assertEquals(abBalanceado.iesimo(0), new Integer(1));
		assertEquals(abBalanceado.iesimo(5), new Integer(8));
		assertEquals(abBalanceado.iesimo(8), new Integer(14));

		boolean thrown = false;
		try {
			abBalanceado.iesimo(88);
		} catch (Exception e) {
			thrown = true;
		}
		assertTrue(thrown);
	}

	@Test
	public void testIrep() {
		assertTrue(abVacio.irep());
		assertTrue(abDesbalanceado.irep());
		assertTrue(abBalanceado.irep());

		abBalanceado.romperIrep();
		assertFalse(abBalanceado.irep());
	}
}
```