Segunda Práctica (P2)

Implementación del diseño de una estructura de clases

Competencias específicas de la segunda práctica

- Interpretar diagramas de clases del diseño en UML.
- Implementar el esqueleto de clases (cabecera de la clase, declaración de atributos y declaración de métodos) en Java y en Ruby, y relacionarlas adecuadamente a nivel de implementación.
- Implementar algunos de los métodos simples de las clases.

A) Programación y objetivos

Tiempo requerido: Dos sesiones, S1 y S2 (cuatro horas).

Comienzo: **semana del 1 de octubre.**

Planificación y objetivos:

Sesión	Semana	Objetivos
S1	1 de octubre	• Ubicar las clases y asociaciones implementadas en la primera práctica dentro del diagrama de clases UML dado. • Identificar y definir las nuevas clases. • Declarar los atributos básicos de cada clase en Java. • Declarar los atributos de referencia (implementan asociaciones entre clases) de cada clase en Java. • Declarar e implementar los métodos constructores y consultores de cada clase en Java. • Declarar en Java otros métodos que aparezcan en el diagrama de clases UML. • Conocer y utilizar el patrón de diseño Singleton.
S2	8-11 de octubre o 19 de octubre	• Los mismos objetivos anteriores en lenguaje Ruby
La práctica se desarrollará tanto en Java como en Ruby en equipos de 2 componentes. El examen es individual.		

Evaluación:

El examen de las prácticas 1 y 2 será en la semana del 15 al 18 de octubre (cada grupo en su sesión) para todos los grupos excepto para los grupos A1, C2 y D1 que lo tendrán el 26 de octubre.

El examen lo realizará **cada grupo en su hora** y aula de prácticas y se resolverá sobre el código que se haya desarrollado para las prácticas 1 y 2 **en el ordenador del aula**. Para ello, cada estudiante deberá acudir a clase con su proyectos Java y Ruby en un pen drive o haberlos dejado en su cuenta de usuario de los ordenadores del aula.

No habrá entrega del código previa al examen, sino que se pedirá que cada estudiante suba a

una tarea creada en PRADO a tal efecto los proyectos Java y Ruby con los cambios solicitados durante el examen.

Enlaces interesantes

http://groups.diigo.com/group/pdoo_ugr/content/tag/Java
https://groups.diigo.com/group/pdoo_ugr/content/tag/ruby

SESIÓN 1

B) S1. Tareas en Java

1) **Implementa el diagrama de clases *DCQytetet*** proporcionado en el fichero *DiagramaClases.pdf*. Incluye todas las clases en el paquete ModeloQytetet. Sigue las siguientes directrices para cada clase:

   a) Declara todas la clases de visibilidad *public.*

   b) Declara todos los atributos básicos teniendo en cuenta, además de su tipo primitivo, su visibilidad (si son *private*, *package*, *protected* o *public*), y su ámbito (si son de instancia o de clase - *static* -).

   c) Declara todos los métodos teniendo en cuenta: parámetros, valor de retorno, visibilidad y ámbito de acceso. Evita errores de ejecución en los métodos que devuelven algún valor, comentando el método o bien lanzando una excepción con el siguiente código:

```
throw new UnsupportedOperationException("Sin implementar");
```

   d) Identifica los atributos de referencia a partir de las asociaciones existentes entre las clases y decláralos.

   e) Implementa los constructores de todas las clases prestando atención a que los objetos queden correctamente inicializados con valores en todos sus atributos. Los atributos pueden inicializarse dentro de los constructores o cuando se declaran. Los constructores deben tener visibilidad *package*, salvo las clases *singleton*, tal y como se describe en el apartado g).

   f) Implementa el método *toString* para todas las clases. Ten en cuenta que si la navegabilidad es en doble sentido (como la asociación entre *Jugador* y *TituloPropiedad*) no puedes poner en los dos métodos *toString* correspondientes el atributo de referencia. ¿Por qué? ¿Cómo lo puedes solucionar?

   g) Ten en cuenta también en la implementación si aparece en el diagrama la palabra ***<<singleton>>*** y ***<<enumeration>>***. Asegúrate de conocer bien qué significa cada unos de esos estereotipos. Crea las clases singleton usando un código parecido al que aquí aparece (suponiendo que *MiClase* es la clase *singleton*). Se dice que una clase es un *singleton* cuando sólo puede tener una instancia. Para conseguirlo utilizamos el patrón de diseño *singleton*:

```
public class MiClase {

   private static final MiClase instance = new MiClase();

    // El constructor privado asegura que no se puede instanciar
    // desde otras clases
   private MiClase() {
```

```
        // La implementación que corresponda.
    }
    public static MiClase getInstance() {
        return instance;
    }
    }
```

Cuando necesitemos acceder a la instancia de `MiClase`:

```
MiClase mc= MiClase.getInstance();
```

Más información: http://en.wikipedia.org/wiki/Singleton_pattern

NOTA: Observa que la clase *Qytetet* es singleton y por tanto deberás realizar los cambios necesarios en el código para ello.

h) **Implementa todos los métodos consultores y modificadores básicos** que aparecen en el diagrama de clases en color rojo.

i) Implementa los siguientes métodos de la clase Qytetet:

- ***inicializarJugadores(nombres : string [2..MAXJUGADORES]) : void***.- Crea un jugador a partir de cada uno de los nombres que están representados en las cadenas de caracteres del contenedor suministrado en el argumento de entrada. El jugador debe tener el valor *7500* en su atributo *saldo*. Los límites impuestos al tamaño del contenedor *nombres* son precondiciones, de forma que el método no tiene que comprobar que éstas se cumplan.

- ***inicializarJuego(nombres : string [2..MAXJUGADORES]) : void***.- Llama a los métodos ***inicializarJugadores***, ***inicializarTablero*** e ***inicializarCartasSorpresa***. Este método se terminará de implementar en la práctica 3.

j) **Implementa** en *PruebaQytetet* **el método de clase *getNombreJugadores()*** para leer desde la interfaz de texto el número de jugadores y sus nombres, devolviendo los nombres en una lista. Para ello puedes usar la clase Scanner y sus métodos de instancia *nextInt* y *nextLine* para leer un entero y un string respectivamente. La creación de una instancia de Scanner y lectura de un entero se puede hacer de la siguiente manera:

```
private static final Scanner in = new Scanner (System.in);

String s = in.nextLine();
```

k) Por último, revisa las clases *Sorpresa*, *TipoSorpresa*, *Tablero*, *Casilla*, *TipoCasilla* y *TituloPropiedad* desarrolladas en la práctica anterior, y asegúrate de que concuerdan con lo que se indica en el diagrama de clases proporcionado.

2) **Pruébalo todo:** Cambia el método *main* de la clase *PruebaQytetet* que hiciste en la práctica 1. Añade un bucle para visualizar a todos los *jugadores* y a la única instancia de la clase *Qytetet*.

3) Realiza en Java los **ejercicios de autoevaluación** que se proponen al final del guión.

SESIÓN 2

C) S2. Tareas en Ruby

1) Sigue lo dispuesto para Java implementando el diagrama de clases, teniendo en cuenta lo indicado en la práctica 1 para adaptar el código a las particularidades de Ruby y otras que vemos en esta práctica por primera vez:

- Para lanzar una excepción en Ruby de código no implementado se usa el código

```
raise NotImplementedError
```

- El patrón *Singleton* en Ruby ya está implementado en el módulo *Singleton* y lo único que tenemos que hacer es usarlo, por ejemplo, si la clase *MiClase* es un singleton tenemos que añadir:
  - Al fichero donde está definida la clase singleton:

```
require "singleton"
```

  - En *MiClase* incluir el módulo *Singleton* de la siguiente forma:

```
include Singleton
```

  - Donde necesitemos acceder a la instancia de *MiClase*:

```
mc = MiClase.instance
```

- La lectura de datos por consola se puede hacer usando el método *gets*, que lee y devuelve un string con lo que introduzcamos hasta pulsar intro. Por ejemplo, la siguiente línea guarda en *cadena* el string leído y la segunda línea guarda en *numero* su conversión a entero, siendo *0* si se introduce un carácter no numérico:

```
cadena=gets
numero=gets.chomp.to_s
```

2) **Pruébalo todo**: utiliza el método *main* de la clase *PruebaQytetet* para crear y mostrar *jugadores, tablero, cartas sorpresa* y a la única instancia de la clase *Qytetet*.

3) Realiza en Ruby los **ejercicios de autoevaluación** que se proponen a continuación.

NOTA: Asegúrate de que todos los métodos resaltados en color rojo en el diagrama de clases están implementados (en Java y en Ruby) al finalizar la práctica.

Ejercicios de autoevaluación

Haz una copia de los proyectos antes de introducir los cambios que se proponen, ya que la práctica siguiente partirá de lo realizado hasta ahora. Trabaja sobre estas copias.

1. Realiza los cambios que consideres oportunos en el código para incluir las siguientes supuestas modificaciones en el diagrama de clases:
   a) En la clase Dado solo se pueden crear cuatro instancias como máximo, un dado para cada

jugador.

b) El número mínimo de jugadores será 3.

c) Se añade "*Escapandose*" como otra forma de salir de la cárcel.

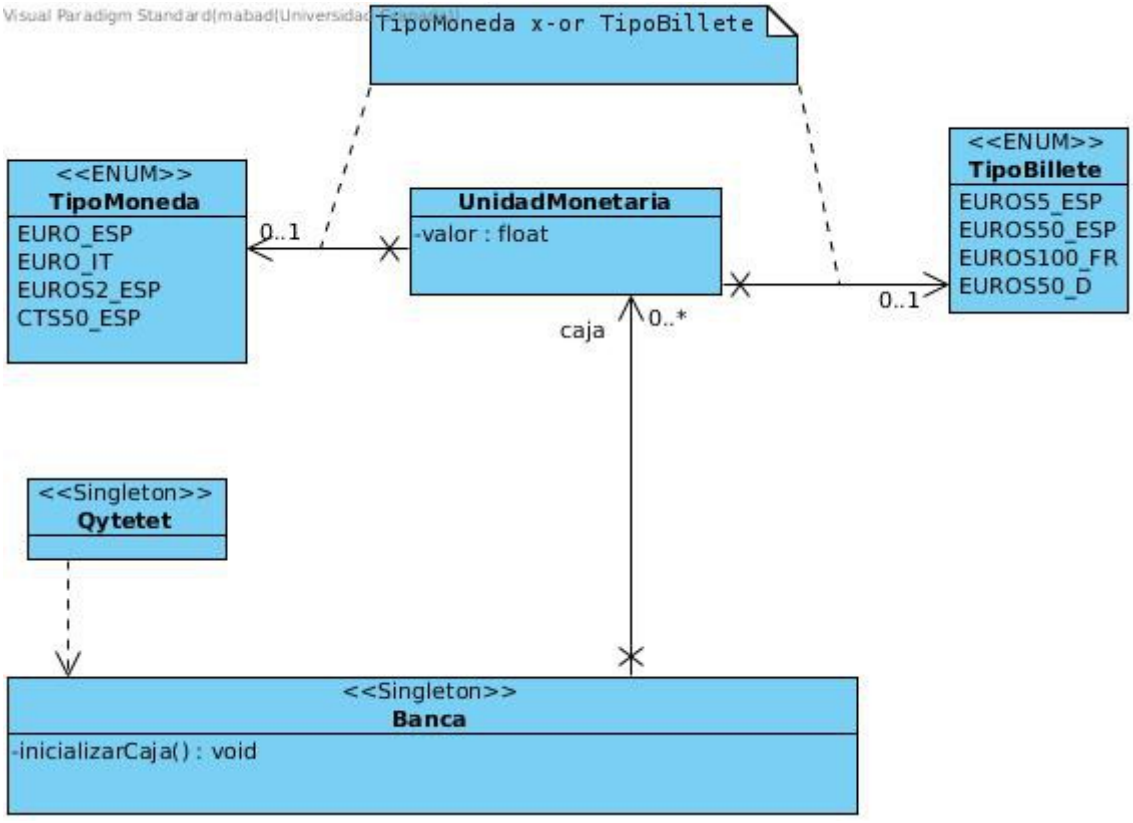
d) El mazo no es un *ArrayList* sino un *HashMap*.

e) La composición de Tablero a Casilla pasa a ser simplemente una agregación.

f) Se desea añadir al juego las fichas, de tal forma que cada jugador tiene asociada una ficha para jugar. Las fichas tienen una forma y un color, y estarán colocadas en una casilla en cada momento. En una casilla puede haber varias fichas. Modifica el diagrama de clases y el código para incluir este nuevo elemento y sus relaciones con los demás.

Ejercicio tipo examen I (Ruby)

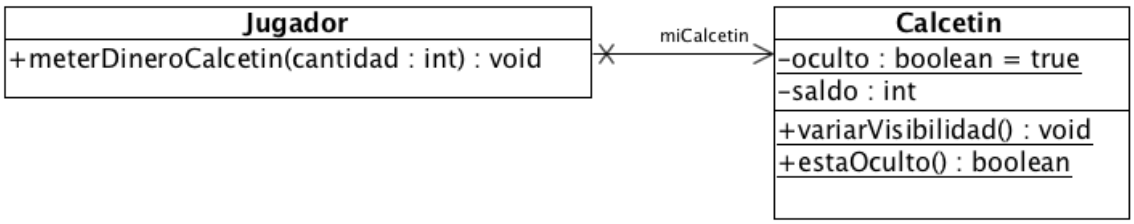
1. Añade al proyecto todo lo que aparece en el diagrama de clases de la figura.
2. Define dos constructores para la clase **UnidadMonetaria**, según se trate de una moneda o un billete.
3. Define el método **to_s** para la clase **UnidadMonetaria** y un consultor para el atributo **valor**.
4. Implementa el método privado ***inicializarCaja*** de **Banca** de tal forma que produzca 5 unidades monetarias de cada tipo y las meta en la **caja**. Este método debe ser llamado desde el constructor.
5. Define un consultor y un modificador para el atributo **caja**.
6. En la clase PruebaQytetet define un atributo de clase de nombre **banca** que apunte al único objeto de la clase Banca.
7. En la clase PruebaQytetet define un método de clase de nombre ***obtenerMonedas*** que recorra la **caja** de la **banca** y devuelva un Array sólo con las monedas de la caja.
8. Llama al método ***obtenerMonedas*** y luego muestra las monedas con el método ***to_s***.



Ejercicio tipo examen II (Ruby)

Los jugadores van a tener calcetines donde meter el dinero que van ganando.

1. Crea la clase **Calcetin** y el/los atributos de referencia que sean necesarios siguiendo el siguiente diagrama e inicializándolo/s en el constructor correspondiente. Sólo el atributo *saldo* de **Calcetin** tiene consultor y modificador. Implementa el constructor para inicializar correctamente el saldo.



2. Implementa el método *estaOculto* de **Calcetin** para que muestre el valor del atributo *oculto* y el método *variarVisibilidad* que cambia el valor de *oculto* de verdadero a falso y viceversa.
3. Implementa el método *meterDineroCalcetin* de **Jugador** para que reste *cantidad* a su *saldo* y se la sume al *saldo* del *Calcetin.*
4. Añade al final del constructor de **Jugador** una instrucción para que meta 300€ en el calcetín usando *meterDineroCalcetin*.
5. Implementa el método *to_s* de **Calcetin** para mostrar el saldo.
6. En el método *to_s* de **Jugador**, añade información de calcetín o no según esté oculto o visible.
7. Implementa o modifica si ya lo tienes el método *to_s* de **Qytetet** para que muestre únicamente la lista de jugadores.
8. Modifica el método *inicializarJugadores* de **Qytetet** para que:
    a. Tenga visibilidad pública.
    b. Reciba, además de la lista de nombres, una lista de calcetines.
    c. Inicialice los jugadores correctamente. Para ello, puedes usar el siguiente formato de bucle de Ruby:

```
for i in 0..lista.length-1
  #El elemento en la posición i se accede con lista.at(i)
end
```

9. En la clase **PruebaQytetet**:
    a. Crea tres calcetines con saldos 10, 100 y 200 respectivamente.
    b. Crea tres jugadores usando los calcetines.
    c. Imprime por pantalla los tres jugadores.
    d. Varía la visibilidad de los calcetines.
    e. Vuelve a imprimir los jugadores.

10. Prueba tu código y asegúrate de que funciona sin errores.