# Informe de Laboratorio 3: Implementación de juego «Dobble» en Java

Paradigmas de Programación

**18/06/2022**

Capítulo

# Índice general

Capítulo 1

# Introducción

Este informe corresponde al 3er laboratorio de la asignatura *Paradigmas de Programación*. En este primer laboratorio se empleara el Paradigma Orientado a Objetos, usando el lenguaje de programación `Java`. Este se usara a través de el IDE [Integrated Development Environment] IntelliJ IDEA.

## 1.1. Descripción del Problema

Se pide crear una implementación de Dobble. Este juego consiste en un mazo de cartas con 55 cartas (57 originalmente) donde dos cartas cualquieras comparten **solo** un elemento entre si. Para jugar Dobble existen distintos tipos modos. El modo mas conocido consiste en que los jugadores deben tomar dos cartas del mazo y el primero que identifique cual símbolo se encuentre en ambas cartas toma la primera carta y la guarda. Después, se toma otra carta y se repite el proceso hasta que se acabe el mazo. Una vez que se acaben las cartas, el jugador que tenga la mayor cantidad de cartas ganará. Para representar esto en Java hay que tener en cuenta que la generación de cartas deberá respetar la condición de que solo se repita el símbolo una vez entre cualquier par de cartas.

## 1.2. Descripción del Paradigma

Para crear la representación de «Dobble» se usara la Programación Orientada a Objetos , Un paradigma de la programación el cual se basa el uso de objetos y clases, donde estas clases tendrán sus métodos y datos estarán contenidos dentros de ellas y podrán ser instanciadas al crear objetos. Las principales características de este paradigma son:

- **Herencia**: Nos permite usar el código de una clase ya existente en otra clase. Esto nos permite compartir métodos y datos entre ellos. En el caso de java, una clase solamente podrá heredar una otra clase.
- **Asociación**: Los objetos se podrán comunicar con otros, para poder usar sus métodos y datos.
- **Agregación**: La agregación es una forma de asociación, La cual es una relación unidireccional ya que una clase podrá tener diferentes objetos de una clase, pero la otra clase no podrá tener objetos de ella. Además, no importara si se ven afectadas las instancias de estas clases, ya que no es necesaria que estén siempre disponibles para que el objeto pueda existir.

- **Composición**: La composición es parecida a la agregación, solo que en está las clases serán dependientes entre si, osea que si no esta presenta una en la otra, el objeto no podrá existir.

Estas características tendrán sus ventajas y desventajas. Las ventajas es que debido a la herencias, se podrá ahorrar el tener que crear el mismo código para diferentes clases, ya que al heredar la clase base se tendrá lo necesario, y en el caso de querer cambiar una función, solamente hay que realizar sobreescritura. Además, gracias a la asociación y sus formas, sera fácil hacer uso de diferentes objetos en una misma clase, facilitando la creación de programas. Pero estará la desventaja de que si uno no esta acostumbrado a este paradigma, sera difícil acostumbrarse al principio. Tambien, es posible que nuestro código sea mas grande en comparación a otros paradigmas, ya que hay que realizar las estructuras para todas las clases.

Capítulo 2

# Desarrollo

## 2.1. Análisis del Problema

Para este proyecto se tendrá que tener en cuenta los aspectos fundamentales del juego «Dobble». Este juego, como ya ha sido mencionado, tendrá un mazo de cartas con 57 cartas, donde 2 cartas cualquieras tendrán siempre un elemento en común. Para este hay que tener en cuenta las bases matemáticas del juego, pero en este caso se nos entrega un algoritmo escrito en `Javascript` el cual facilitara la creación del algoritmo en `Java`. El mazo creado sera representado como un Array de Cartas. El usuario podrá pedir diferentes acciones respecto al mazo o a una carta [T D A cardsSet]:

- Verificar si un mazo es valido [dobble?]
- Ver el numero de cartas en el mazo [numCards]
- Buscar que carta hay en x posición [nthCard]
- Encontrar cuantas cartas necesita para un mazo valido a partir de una carta [findTotalCards]
- Encontrar el numero de elementos que necesita para armar un mazo valido a partir de una carta [requiredElemets]
- Ver que cartas faltan de un mazo para armar uno valido [missingCards]
- Transformar el mazo a una representación en strings. [cardsSet→string]

Para permitir al usuario poder jugar el juego hay que tener en cuenta que este podrá [T D A game]:

- Ingresar cantidad de jugadores [register y numPlayers]
- Elegir modo de juego [mode, stackMode, myMode y emptyHandsStackMode]
- Elegir que hacer en su turno [play y pass]
- Terminar el juego cuando quiera
- Ver de quien es el turno y el puntaje [whoseTurnIsIt? y status]

## 2.2. Diseño de la Solución

Para la solución de este problema, se tienen en cuenta todas las acciones que el usuario podrá tomar, ya sea para armar el mazo o para las acciones que podrá tomar al jugar el

juego una vez ya tenga el mazo armado.

### 2.2.1. TDA Card

Este TDA representara cada carta que se generara y en el ella simplemente bastara con tener un ArrayList de Strings, ya que los elementos ingresados a ella seran representados como strings.

### 2.2.2. TDA Dobble

#### Creación de cartas (Constructor Dobble)

El mazo de cartas creado sera representado como una Array de Cartas , y cada carta tendrá elementos, los cuales podrán ser o un numero, o un símbolo introducido por el usuario. El principal problema sera la creación de cartas,para ello se nos entrega el siguiente algoritmo escrito en Javascript. (Ver Anexo 1)

El código de Javascript podrá ser facilmente implementado en Java, a diferencia de los paradigmas anteriores en los cuales era necesario el uso de recursión. En este TDA se generaran los elementos que irán en cada carta y se instanciara un objeto Card, en el cual se ingresaran los elementos generados y se agregara este objeto a el conjunto de cartas en la clase Dobble. Ahora se obtiene un mazo de cartas, pero este esta representando por numero y hay que tener en cuenta que el usuario podrá pedir en la generación de cartas que este se genere con una lista de símbolos ingresada por el usuario (Elements), un numero de elementos en cada carta (numE), un máximo de cartas (maxC) y una función que aleatorizara el ordenamiento de las cartas (rndFn). Para generar el mazo de cartas con los símbolos, se tomara generara el mazo y se remplazara el símbolo que este en la posición i y se remplazaran todos los i que hayan en el mazo con ese símbolo y se repetirá el proceso hasta que no queden mas símbolos o mas cartas. Y para el máximo de cartas generado solo bastara con tomar los maxC elementos de la lista. Y finalmente con todo ya listo, se procede a aleatorizar el ordenamiento de las cartas con el método Shuffle.

#### Pertenencia

Como función de pertenencia tendremos que verificar que el mazo de Dobble entregado por el usuario sea un mazo valido para jugar. Para ello tendremos que verificar que la cantidad de cartas es la correcta, lo cual se puede verificar con la formula $(n * (n-1) + 1)$,

donde n es el numero de elementos. Además habrá que asegurarnos que los símbolos de las cartas solo se repitan una vez en cada par de cartas.

#### Selectores

Nuestra función de selector sera la función `nthCard`, la cual nos permitirá encontrar la carta n (elegida por el usuario). Para esto bastara hacer de el getter de la carta e indicar el índice (la carta que buscamos).

#### Modificadores

La función modificadora en este TDA sera la función `toString`, la cual nos permitirá representar el mazo (Array de cartas) como strings.

#### Otras Funciones

Se tienen otras funciones como `numCards`, `findTotalCards`, `requiredElements` y `missingCards`. En las tres primeras funciones bastara con usar el metodo `size` y para la ultima función, se hará una intersección entre el mazo ingresado y un mazo generado (valido), donde el resultado serán las cartas faltantes.

### 2.2.3. TDA Player

El juego necesitara jugadores para poder ser jugado correctamente. En esta clase los jugadores tendran de datos su nombre como un String y su puntaje representado como un integer.

### 2.2.4. TDA DobbleGame

El TDA DobbleGame permitirá al usuario jugar con el set de cartas anteriormente creado. El usuario podrá registrar usuarios y elegir un modo de juego. Esta clase tendrá de datos, ArrayList de los Jugadores, un int con número de jugadores y un mazo de tipo Dobble (TDA Dobble).

#### Constructor

Para el juego, se creara un constructor en el cual se indica el numero de jugadores y el mazo a utilizar. Los jugadores se podrán registrar usando el método register.

#### Selectores

Se tendrán varias funciones selectoras, ya que hay que poder obtener de quien es el turno actual (whoseTurnIsIt?), obtener el status del juego (status) y conseguir el puntaje de cada jugador (score).

#### Modificadores

Se hará uso de este tipo de función para poder registrar a los usuarios que irán a jugar el juego (register), para poder mostrar el juego en forma de string (game->string).

#### Otras Funciones

El usuario podrá elegir el modo de juego que quera usar, ya sea stackMode, emptyHandsStackMode o un una modalidad creada por uno mismo (myMode). Además podrá accionar en el juego con la función play, y podrá elegir si pasar su turno o seleccionar que símbolo se repite en las dos cartas mostradas.

#### Diagrama de Análisis

Con algunos de los TDAs identificados, se procede a crear un UML para ayudar el proceso de desarrollo antes de comenzarlo en si.(Ver Anexo 3).

## 2.3. Aspectos de Implementación

- Estructura del proyecto: Cada TDA sera indicado en su propia interface y la implementación de estos en clases proveerán las funciones necesarias a un archivo principal `Main.java`, el cual podrá ser compilado y ejecutado a través de consola y se podrá jugar el juego a través de un menú interactivo.

- Bibliotecas utilizadas: Para este proyecto solo se hizo uso de la librería base de Java
- Interprete y compilador utilizado: Se hizo uso de IntelliJ IDEA v.2022.1.2 para la edición de código. Y en el caso del JDK se utilizo Azul Zulu v. 11 para la compilación. El programa fue compilado y ejecutado en MacOS Monterrey 12.4.
- Razones de elección: Cada TDA fue indicado en una interface ya que estas indican un contrato, indicando que es lo que deben hacer, pero no indicando el como. Permitiendo guiar la construcción de cada clase al saber que es lo que debe implementar esta. Se hizo uso de IntelliJ IDEA debido a su fácil uso y modernidad. Además, se uso el JDK de Azul Zulu ya que el código fue compilado en un procesador ARM y este tiene versiones especificas para ARM en MacOS.

## 2.4. Instrucciones de uso:

Para la compilación y ejecución del código dentro de un entorno UNIX, se encuentra en el projecto un archivo .sh ("script.sh"), el cual permitirá la fácil compilación del código. Para hacer uso de el se necesitara darle permiso de ejecutable y ejecutarlo con los siguientes comandos una vez se encuentra dentro de el directorio.

```
chmod +x ./script.sh;
bash ./script.sh;
## o solamente ##
./script.sh;
```

Una vez ejecutado el script correctamente se mostrara un mensaje pidiendo el ingreso de los usuarios que se vayan a registrar. Se procede a ingresar los nombres y una vez terminado se ingresa 0 para terminar el ingreso. Se mostrara un menú donde se podrán elegir diferentes opciones. Para el correcto funcionamiento de el programa primero se deberá elegir la opción 1. para crear el juego, donde se debe indicar el numero de Elementos por carta, el numero de cartas y el numero de jugadores que jugaran. Después, se elige la opción 2 para registrar los jugadores que jugaran en esa partida. Si uno de los jugadores no fue registrado al momento de iniciar el programa, este no se registrara. Finalmente, se podrá hacer uso de la opción 3 o 4 para jugar o ver otras opciones disponibles.

### 2.4.1. Resultados Esperados

Se espera que una vez ejecutado el script, se muestre una entrada preguntando al usuario por los jugadores a registrar. Cuando el usuario termine de registrar usuarios, se espera que se le presente un menú con sus opciones y que ahí pueda crear un juego, indicar los usuarios que participaran en el juego y que pueda realizar acciones respecto al juego.

### 2.4.2. Posibles errores

Es posible que si el usuario intenta registrar jugadores en el juego antes de haberlo creado, se presente errores en el programa. Así mismo, si este intenta realizar acciones en el juego sin crearlo con anterioridad se presentara un error.

## 2.5. Resultados y autoevaluación

Probando el programa creado con los ejemplos demostrados, se puede ver que todos los TDA funcionan correctamente y que el menú interactivo funciona mientras se sigan las instrucciones. La autoevaluación (Ver Anexo 1) se evaluara de la siguiente forma:

1. Nombre de la función
2. Tipo de Prueba
3. Razón de fallos
4. Completado?
5. Evaluación

La columna de evaluación sera evaluada según lo siguiente:

1. 0: No realizado.
2. 0.25: Implementación con problemas mayores (funciona 25 % de las veces o no funciona)
3. 0.5: Implementación con funcionamiento irregular (funciona 50 % de las veces)
4. 0.75: Implementación con problemas menores (funciona 75 % de las veces)
5. 1: Implementación completa sin problemas (funciona 100 % de las veces)

Además, se crea un diagrama de clases UML para este programa ya completado.(Ver Figura 4).

Capítulo 3

# Conclusiones

El Paradigma orientado a objetos fue al principio algo difícil de entender ya que uno se encuentra acostumbrado a la sintaxis más simple de otros lenguajes como `Python` y `Javascript`, las cuales son mas parecidas al lenguaje humano y además no requieren siempre el uso de clases y objetos. Aun con estas complicaciones, este proyecto fue de mucha utilidad ya que permitió aprender sobre un diferente paradigma de programación, hacer uso de tecnologías como git y lenguajes de modelamiento como el UML.

# Capítulo 4
# Referencias


1. Java Platform SE 7. (2020). JavaTM Platform, Standard Edition 7 API Specification. Retrieved 2022, from https://docs.oracle.com/javase/7/docs/api/
2. Gonzaléz, R. (2022). Paradigmas de Programación - Proyecto semestral de laboratorio. Recuperado 2022, de [redacted]
3. Dore, M. (2021, 29 diciembre). The maths behind Dobble - Micky Dore. Medium. Recuperado 2022, de https://mickydore.medium.com/dobble-theory-and-implementation-ff21ddbb5318
4. Dore, M. (2021, diciembre 30). The Dobble Algorithm - Micky Dore. Medium. https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52

Capítulo

# 5 Anexos

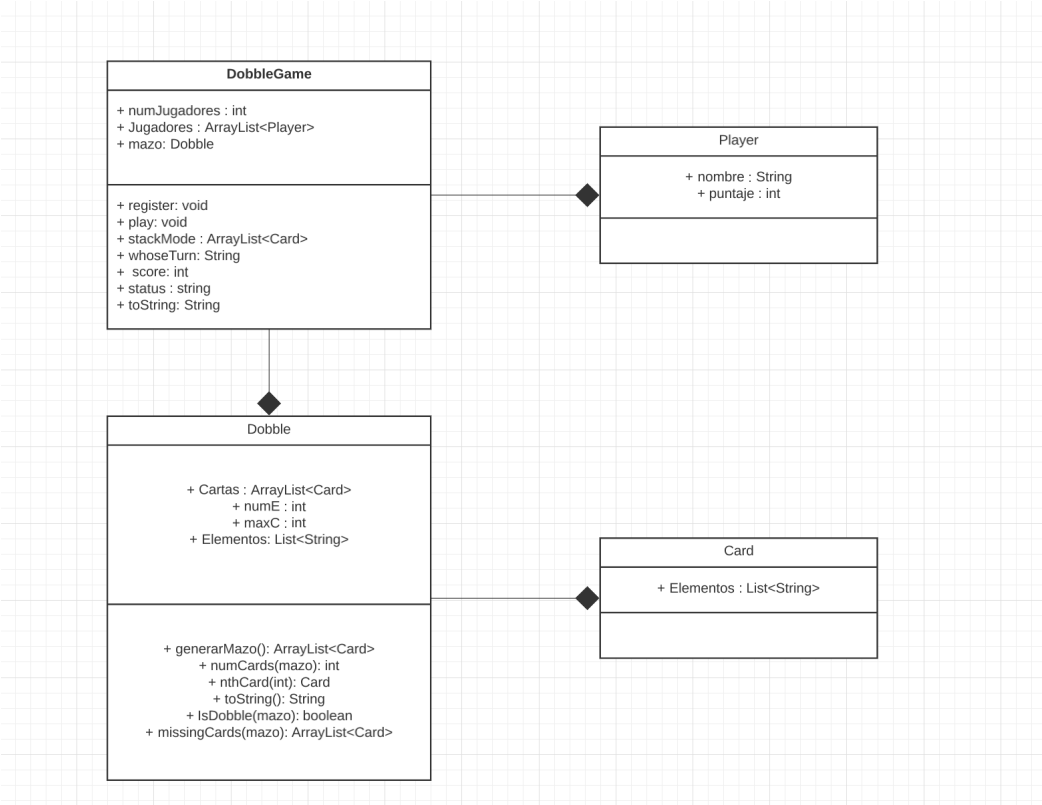
## 5.1. Anexo 1: Algoritmo generación de cartas

```
//Generación Dobble Javascript
//
let i, j, k
let n = 3                 //order of plane, must be a prime number
let numOfSymbols = n + 1  //order of plane + 1let cards = [] //the deck of cards
let card = []; //the current card we are building//to start, we build the first
↪  card
let cards = [];

for (i = 1; i<= n+1; i++) {
    card.push(i)
}
cards.push(card)//then we build the next n number of cards

for (j=1; j<=n; j++) {
    card = []
    card.push(1)

    for (k=1; k<=n; k++) {
        card.push(n * j + (k+1))
    }
    cards.push(card)
}//finally we build the next n² number of cards

for (i= 1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        card = []
        card.push(i+1)

        for (k=1; k<= n; k++) {
            card.push(n+2+n*(k-1)+(((i-1)*(k-1)+j-1)%n))
        }
        cards.push(card)
    }
}
```

## 5.2. Anexo 2: Tabla de Pruebas y autoevaluación

| Función | Tipo de prueba | Razón de fallos | Completado? | Evaluación |
|---|---|---|---|---|
| Clases y Estructuras | Se prueban en menú interactivo | No hubo fallos | Si | 1 |
| Menú Interactivo | | Funciona correctamente en el programa | Si | 1 |
| Constructor de Juegos | Se prueba creando juego en Menú | No hubo fallos | Si | 1 |
| Register | Se registra más de los posibles y repetidos | No hubo fallos | Si | 1 |
| toString | Se llevan a String diferentes Clases | No hubo fallos | Si | 1 |
| equals(Object o) | Se prueba con diferentes clases | No hubo fallos | Si | 1 |

## 5.3. Anexo 3: Diagrama de Análisis



## 5.4. Anexo 4: Diagrama de Diseño