

# Informe de Laboratorio 4: Implementación de juego «Dobble» en Java Dirigido por Eventos



PARADIGMAS DE PROGRAMACIÓN

11/07/2022



## Índice general

<b>1</b>	<b>Introducción</b>	<b>3</b>
1.1	Descripción del Problema . . . . .	3
1.2	Descripción del Paradigma . . . . .	3
<b>2</b>	<b>Desarrollo</b>	<b>5</b>
2.1	Análisis del Problema . . . . .	5
2.2	Diseño de la Solución . . . . .	5
2.2.1	TDA Card . . . . .	6
2.2.2	TDA Dobble . . . . .	6
2.2.3	TDA Player . . . . .	7
2.2.4	TDA DOBBLEGAME . . . . .	7
2.3	Aspectos de Implementación . . . . .	8
2.4	Instrucciones de uso: . . . . .	9
2.4.1	Resultados Esperados . . . . .	9
2.4.2	Posibles errores . . . . .	9
2.5	Resultados y autoevaluación . . . . .	10
<b>3</b>	<b>Conclusiones</b>	<b>11</b>
<b>4</b>	<b>Referencias</b>	<b>12</b>
<b>5</b>	<b>Anexos</b>	<b>13</b>
5.1	Anexo 1: Algoritmo generación de cartas . . . . .	13
5.2	Anexo 2: Tabla de Pruebas y autoevaluación . . . . .	14
5.3	Anexo 3: Diagrama de Análisis . . . . .	14
5.4	Anexo 4: Diagrama de Diseño . . . . .	15

# CAPÍTULO Introducción 1

Este informe corresponde al 4to laboratorio de la asignatura *Paradigmas de Programación*. En este laboratorio se empleara el Paradigma Orientado a Eventos, 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 cualesquiera 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 principalmente la Programación dirigida por eventos, junto a la Programación Orientada a Objetos , Un paradigma de la programación el cual se basa el uso de eventos , donde los eventos determinaron el flujo de el programa. Estos distintos eventos pueden ya ser acciones que realiza el usuario, mensajes de otras aplicaciones,etc. Las principales características de este paradigma son:

- **Eventos:** Como ha sido mencionado anteriormente, los eventos son una acción que puedes provenir de distintos lugares, pero que el programa recibirá y podrá ser manejado por este.
- **Manejador de eventos:** Al recibir la entrada de un evento, el programa llamara a este manejador de eventos, el cual ejecutara una sección de código cuando reciba un evento que le corresponda.

Estas características tendrán sus ventajas y desventajas. La principal ventaja es que va de la mano junto a la creación de GUIs, ya que permitirá al usuario crear los eventos y el programa fácilmente los podrá recibir y ver que hacer con ellos. Pero estará la desventaja

de que si uno no esta acostumbrado a este paradigma, sera difícil acostumbrarse al principio. También, es posible que nuestro código sea mas grande en comparación a otros paradigmas, ya que hay que realizar las estructuras para todas las clases y componentes de la GUI.



## CAPÍTULO Desarrollo 2

### 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 cualesquiera 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 [TDA 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 [TDA 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 ella simplemente bastara con tener un ArrayList de Strings, ya que los elementos ingresados a ella serán 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 fácilmente 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 creará un constructor en el cual se indica el número 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 quiera usar, ya sea `stackMode`, `emptyHandsStackMode` o 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 comenzar a programarlo (Ver Anexo 3).

## 2.3. Aspectos de Implementación

- Estructura del proyecto: Cada TDA será indicado en su propia interface y la implementación de estos en clases proveerán las funciones necesarias a un archivo principal `DobbleUI.java`, el cual podrá ser ejecutado a través de IntelliJ IDEA y ser interactuado con el programa ejecutado por este.





- 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 utilizó 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 usó el JDK de Azul Zulu ya que el código fue compilado en un procesador ARM y este tiene versiones específicas para ARM en MacOS.

## 2.4. Instrucciones de uso:

Para la compilación y ejecución del código dentro de un entorno UNIX, se necesita hacer uso de el IDE IntelliJ IDEA. Dentro de este hay que abrir el proyecto y ejecutar la clase DobbleUI. Una vez sea ejecutado correctamente se podrá observar una interfaz simple. Primero se procede a presionar el botón "Crear Mazo", se rellena con los datos deseados y se presiona "Crear Mazo para jugar". Con el mazo creado se presiona el botón registrar y se rellena el campo de texto con los jugadores que el usuario quiera registrar dentro del juego, presionando registrar por cada uno. Finalmente se presiona el botón "Jugar". En esta sección primero se deberá iniciar el juego, con lo cual se realizará el volteo inicial de cartas. Después, se les mostrara las cartas al usuario y el turno de quien le toca jugar. Este usuario deberá ingresar el elemento que se repite y presionar "Spot it!" para realizar su acción. Cuando el usuario quiera terminar el juego o se acaben las cartas, se presionara "Terminar Juego", con lo cual se mostrara en la parte izquierda el estado final de el juego.

### 2.4.1. Resultados Esperados

Se espera que una vez ejecutado correctamente el programa a través de el IDE indicado, el usuario no tenga problemas usándolo, siempre y cuando este siga las instrucciones de uso.

### 2.4.2. Posibles errores

Es posible que si el usuario no sigue el orden indicado anteriormente, se muestren diferentes errores por la consola de IntelliJ IDEA, ya que el programa espera que el usuario siga el orden indicado.



## 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 interfaz 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 Anexo 4).




## CAPÍTULO

# Conclusiones 3

El Paradigma Dirigido por eventos no fue difícil de comprender a diferencias de los paradigmas anteriores, pero por la falta de tiempo que quedo para aprender sobre este y realizar la implementación de este no se pudo entender por completo. 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

# Referencias 4

1. Java Platform SE 7. (2020). Java™ Platform, Standard Edition 7 API Specification. Retrieved 2022, from <https://docs.oracle.com/javase/7/docs/api/>
2. Java Platform SE 7. (2020). Java™ Platform, Package javax.swing. Retrieved 2022, from <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>
3. González, R. (2022). Paradigmas de Programación - Proyecto semestral de laboratorio. Recuperado 2022, de 
4. Dore, M. (2021, 29 diciembre). The maths behind Dobble - Micky Dore. Medium. Recuperado 2022, de <https://mickydore.medium.com/dobble-theory-and-implementation-ff21ddbb5318>
5. Dore, M. (2021, diciembre 30). The Dobble Algorithm - Micky Dore. Medium. <https://mickydore.medium.com/the-dobble-algorithm-b9c9018afc52>

## 5

## 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

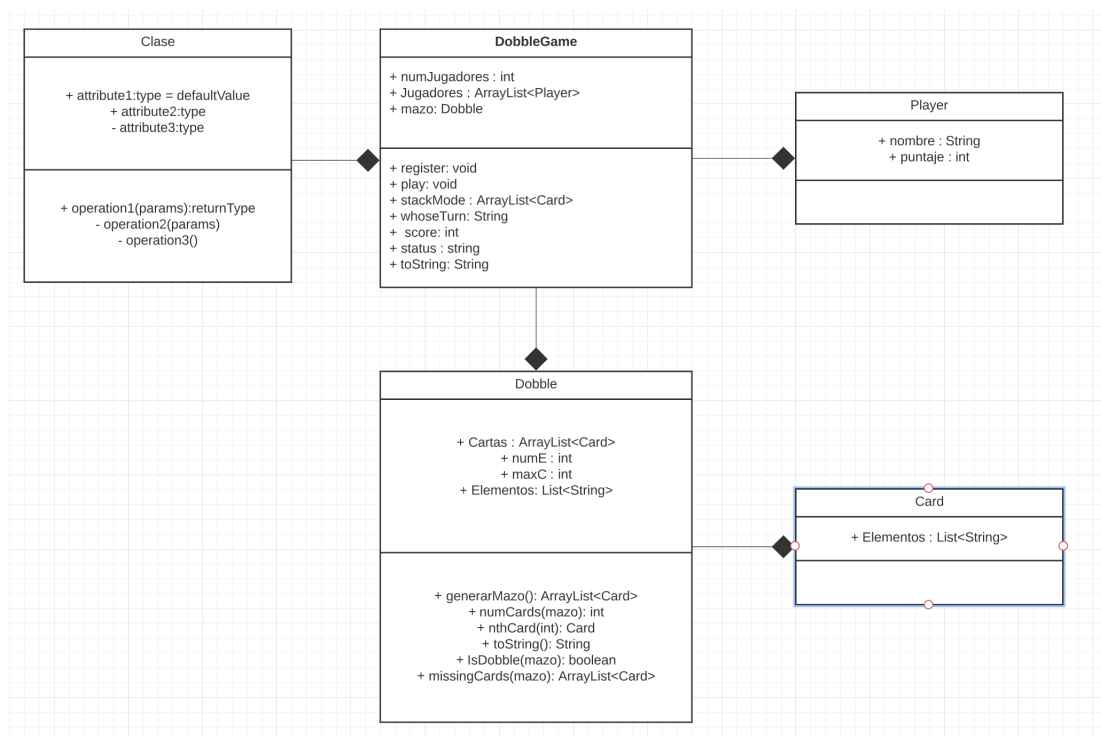
for (i= 1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        card = []
        card.push(i+1)

        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 GUI interactivo	No hubo fallos	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
Play	Se Juega con diferentes mazos	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

