Universidad Nacional de General Sarmiento

Programación II – Comisión 3

Trabajo Práctico Integrador – Segunda parte

Docentes:

- Nores, José
- Gabrielli, Miguel Ángel

Integrantes:

- Cordua, Kevin Nicolas Daniel
- Chazarreta, Jeremías Vladimir

2025C1

TADs

TADs:

- Ticketek
- Usuario
- Entrada
- Espectaculo
- Funcion
- Platea
- Campo
- Miniestadio
- Estadio
- Teatro

TAD Ticketek:

Datos

- usuarios: diccionario <String, Usuario>
- sedes: diccionario <String, Sede>
- espectaculos: diccionario <String, Espectaculo>
- dateFormatter

Operaciones

- Ticketek () // constructor
- registrarSede (nombre, direccion, capacidadMaxima) // registra las sedes del tipo // Estadio
- registrarSede (nombre, direccion, capacidadMaxima, asientosPorFila, sectores, capacidad, porcentajeAdicional) // método sobrecargado, registra sedes del tipo Teatro
- registrarSede (nombre, direccion, capacidadMaxima, asientosPorFila, cantidadPuestos, precioConsumicion, sectores, capacidad, porcentajeAdicional) // método sobrecargado, // registra sedes del tipo mini estadio
- registrarUsuario (email, nombre, apellido, contrasenia) // registra usuarios nuevos
- registrarEspectaculo (nombre) // registra los espectáculos
- agregarFuncion (nombreEspectaculo, fecha, sede, precioBase) // añade una función a // un espectáculo
- venderEntrada (nombreEspectaculo, fecha, email, contrasenia, cantidadEntradas): List<IEntrada> // permite vender entradas a los usuarios para funciones en sedes no // numeradas

- venderEntrada (nombreEspectaculo, fecha, email, contrasenia, sector, asientos): List<IEntrada> // método sobrecargado, permite vender entradas a los usuarios para // funciones en sedes numeradas
- listarFunciones (nombreEspectaculo) // despliega un String donde cada fila representa // una función
- listarEntradasEspectaculo (nombreEspectaculo): List<IEntrada> // Busca todas las // entradas vendidas // para un espectáculo
- listarEntradasFuturas (email, contrasenia): List<IEntrada> // despliega una lista con las // entradas de un cliente para fechas que no llegaron aún
- listarTodasLasEntradasDelUsuario (email, contrasenia): List<IEntrada> // despliega // una lista con todas las entradas del usuario
- anularEntrada (entrada, contrasenia): lógico // permite al usuario anular una entrada, // está en O (1)
- cambiarEntrada (entrada, contrasenia, fecha) // método sobrecargado, permite al // usuario cambiar su entrada por otra si la sede es un Estadio
- cambiarEntrada (entrada, contrasenia, fecha, sector, asiento) // permite al usuario // cambiar su entrada por otra si la sede es un Miniestadio o un Teatro
- costoEntrada (nombreEspectaculo, fecha): double // calcula el costo de una entrada para // una sede del tipo estadio, está en O (1)
- costoEntrada (nombreEspectaculo, fecha, sector): double // método sobrecargado, // calcula el costo de una entrada para una sede numerada, está en O (1)
- totalRecaudado (nombreEspectaculo): double // calcula cuánto recaudó un espectáculo // en total
- totalRecaudadoPorSede (nombreEspectaculo, nombreSede): double // calcula cuánto // recaudó un espectáculo en total en una sede determinada, está en O (1)
- toString ()
- generarCodigoEntrada () // Método auxiliar privado para generar códigos únicos de // entrada

IREP

- **usuarios** es un diccionario tal que:
  - ∀ e ∈ claves(usuarios): e ≠ null ∧ e ≠ “” () // e = email
  - ∀ u ∈ valores(usuarios): u ≠ null // u = Usuario
  - ∀ e1, e2 ∈ claves(usuarios): e1 ≠ e2 // e1= email1 e2= email2
  - Para cada usuario u, su colección de entradas es un diccionario Map<String, Entrada> tal que:
    - ∀ k ∈ claves(u.getEntradas()): k ≠ null ∧ k ≠ “” // k = código de la entrada
    - ∀ e ∈ valores(u.getEntradas()): e ≠ null / e = objeto Entrada
    - ∀ k ∈ claves(u.getEntradas()): k.equals(e.codigo()) donde e = u.getEntradas().get(k)
    - ∀ k1, k2 ∈ claves(u.getEntradas()): k1 ≠ k2

- **sedes** es un diccionario tal que:
    - ∀ s ∈ claves(sedes): s ≠ null ∧ s ≠ “” // s = nombre de la sede
    - ∀ v ∈ valores(sedes): v ≠ null // v = objeto Sede
    - ∀ s1, s2 ∈ claves(sedes): s1 ≠ s2 // s1=nombre de sede1 s2= nombre de sede2
- **espectaculos** es un diccionario tal que:
    - ∀ n ∈ claves(espectaculos): n ≠ null ∧ n ≠ “” // n = nombre del espectáculo
    - ∀ e ∈ valores(espectaculos): e ≠ null // e = objeto Espectaculo
    - ∀ n1, n2 ∈ claves(espectaculos): n1 ≠ n2
    - Para cada espectáculo e, su colección de funciones es un diccionario Map<LocalDate, Funcion> tal que:
        - ∀ f ∈ claves(e.getFunciones()): f ≠ null // f = fecha de la función
        - ∀ fun ∈ valores(e.getFunciones()): fun ≠ null // fun = objeto Función
        - ∀ f ∈ claves(e.getFunciones()): f.equals(fun.fecha()) donde fun = e.getFunciones().get(f)
        - ∀ f1, f2 ∈ claves(e.getFunciones()): f1 ≠ f2 // f1 = fecha1 f2= fecha2
- **dateFormatter** usa el patrón “dd/MM/yy”

TAD Usuario:

Datos

- nombre
- apellido
- email
- contrasenia
- entradas: diccionario <String, Entrada>

Operaciones

- usuario (nombre, apellido, email, contrasenia)
- isRegistrado (): lógico
- isEmailValido (): lógico
- isContraseniaValida (): lógico
- agregarEntrada (entrada)
- getEntrada (codigo) // devuelve el objeto Entrada mediante su código
- eliminarEntrada (codigo): lógico
- getEntradas(): diccionario <String, Entrada>
- validarDatos (email, nombre, apellido, contrasenia)
- validarEmail (email)
- validarContrasenia (contrasenia)

IREP

- nombre, apellido, email, contrasenia ≠ null ∧ ≠ “”
- email contiene '@'

- |contrasenia| ≥ 4 // |x| es la longitud del arreglo x
- entradas es un diccionario tal que:
    - ∀ c ∈ claves(entradas): c ≠ null ∧ c ≠ “” // c = código de la entrada
    - ∀ e ∈ valores(entradas): e ≠ null // e = objeto Entrada
    - ∀ c ∈ claves(entradas): c.equals(e.codigo()) donde e = entradas.get(c)
    - ∀ c1, c2 ∈ claves(entradas): c1 ≠ c2 // c1 = codigo1 c2 = codigo2

TAD Entrada:

Datos

- fecha
- sector
- espectaculo
- codigoEntrada
- emailUsuario
- fila
- asiento

Operaciones

- entrada (fecha, sector, espectaculo, codigoEntrada, emailUsuario, fila, asiento)
- esDeCodigo (codigo): lógico
- esDeUsuario (email): lógico
- esDeEspectaculo (nombre): lógico
- esDeFecha (f): lógico
- esDeSector (s): lógico
- esDeFila (f): lógico
- esDeAsiento (a): lógico
- esFutura (hoy): lógico
- codigo ()
- usuario ()
- espectaculo ()
- fecha ()
- sector ()
- asiento ()
- precio (): double
- ubicación ()
- toString()

IREP

- fecha ≠ null
- espectaculo ≠ null

- codigoEntrada, emailUsuario ≠ null ∧ ≠ “”
- Si sector ≠ null ∧ sector ≠ “” ∧ sector ≠ “CAMPO” ⇒ asiento ≠ null
- Si sector = “CAMPO” (ignorando mayúsculas/minúsculas) ⇒ (asiento = null ∨ asiento = -1)

TAD Espectaculo:

Datos

- nombre
- funciones: diccionario <LocalDate, Funcion>
- totalRecaudado
- codigo

Operaciones

- espectaculo (nombre)
- nombre ()
- codigoEspectaculo ()
- getFunciones (): diccionario <LocalDate, Funcion>
- agregarFuncion (fecha, funcion)
- calcularRecaudacionTotal (): double
- toString()
- validarDatos (nombre)
- validarFuncionNoRepetida (fecha)

IREP

- nombre ≠ null ∧ nombre ≠ “”
- funciones ≠ null
- funciones es un diccionario <LocalDate, Funcion> tal que:
  - ∀ f ∈ claves(funciones): f ≠ null // f = fecha de la función
  - ∀ fun ∈ valores(funciones): fun ≠ null // fun = objeto Funcion
  - ∀ f ∈ claves(funciones): f.equals(fun.fecha()) donde fun = funciones.get(f)
  - ∀ f1, f2 ∈ claves(funciones): f1 ≠ f2 // f1 = fecha1 f2= fecha2
- totalRecaudado ≥ 0
- codigo ≠ null

TAD Funcion:

Datos

- fecha

- sede
- sedeObj
- precioBase
- nombreEspectaculo
- recaudacion
- entradasPorSector: diccionario <String, Integer>
- asientosPorSector diccionario <String, Boolean []>

Operaciones

- funcion (fecha, sedeObj, precioBase, nombreEspectaculo)
- sede ()
- precioBase (): double
- recaudacion (): double
- fecha ()
- nombreEspectaculo ()
- sedeObj ()
- reservarEntrada (sector, asiento)
- asientoDisponible (sector, asiento): lógico
- getEntradasVendidas (): int
- getEntradasVendidasPorSector (sector): int
- liberarAsiento (sector, asiento)
- toString()
- validarDatos (nombreEspectaculo, fecha, sede, precioBase)
- descripcionParaListado (sede, fechaStr)
- calcularPrecioEntrada (sector): double
- liberarAsientoSiCorresponde (sector, asiento)

IREP

- fecha ≠ null
- sede ≠ null ∧ sede ≠ “”
- sedeObj ≠ null
- precioBase > 0
- nombreEspectaculo ≠ null ∧ nombreEspectaculo ≠ “”
- entradasPorSector es un diccionario tal que:
    - ∀ k ∈ claves(entradasPorSector): k ≠ null ∧ k ≠ “” // k = nombre del sector
    - ∀ v ∈ valores(entradasPorSector): v ≠ null // v = cantidad de entradas vendidas // en el sector, tipo Integer
    - ∀ k1, k2 ∈ claves(entradasPorSector): k1 ≠ k2
- asientosPorSector es un diccionario tal que:
    - ∀ k ∈ claves(asientosPorSector): k ≠ null ∧ k ≠ “” // k = nombre del sector
    - ∀ a ∈ valores(asientosPorSector): a ≠ null // a = arreglo de booleanos que // representa los asientos del sector
    - ∀ a ∈ valores(asientosPorSector): ∀ i: 0 ≤ i < |a| // los índices válidos // corresponden a asientos existentes

- ∀ k1, k2 ∈ claves(asientosPorSector): k1 ≠ k2
- recaudacion ≥ 0

TAD Platea:

Datos

- nombre
- capacidad
- porcentajeAdicional

Operaciones

- platea (nombre, capacidad, porcentajeAdicional) //constructor
- porcentajeAdicional (): double

IREP

- nombre ≠ null ∧ nombre ≠ “”
- capacidad > 0
- porcentajeAdicional ≥ 0

TAD Campo:

Datos

- nombre
- capacidad
- porcentajeAdicional
- cantidadPuestos

Operaciones

- campo (nombre, capacidad, porcentajeAdicional, cantidadPuestos) //constructor
- porcentajeAdicional (): int
- cantidadPuestos (): int

IREP

- nombre ≠ null ∧ nombre ≠ “”
- capacidad > 0
- porcentajeAdicional ≥ 0
- cantidadPuestos ≥ 0

TAD Miniestadio:

Datos

- capacidadMaxima
- nombre
- direccion
- funcionesPorFecha: diccionario <LocalDate, String>
- cantidadPuestos
- precioConsumicion
- asientosPorFila
- sectores
- capacidad
- porcentajeAdicional

Operaciones

- miniestadio (nombre, direccion, capacidadMaxima, asientosPorFila, cantidadPuestos, precioConsumicion, sectores, capacidad, porcentajeAdicional)
- sectores ()
- capacidadSectores (): int
- porcentajeAdicional (sector): int
- precioConsumicion (): double
- cantidadPuestos (): int

IREP

- nombre ≠ null ∧ nombre ≠ “”
- direccion ≠ null ∧ direccion ≠ “”
- capacidadMaxima, asientosPorFila, cantidadPuestos, precioConsumicion > 0
- sectores ≠ null ∧ |sectores| > 0
- capacidad ≠ null ∧ |capacidad| = |sectores| ∧ ∀ c ∈ capacidad: c > 0
- porcentajeAdicional ≠ null ∧ |porcentajeAdicional| = |sectores| ∧ ∀ p ∈ porcentajeAdicional: p ≥ 0 // |x| es la longitud del arreglo x
- funcionesPorFecha es un diccionario tal que:
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claves(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo // LocalDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2 // f1 = fecha1 f2 = fecha2

TAD Estadio:

Datos

- capacidadMaxima
- nombre
- direccion
- funcionesPorFecha: diccionario <LocalDate, String>
- cantidadPalcos

Operaciones

- estadio (nombre, direccion, capacidadMaxima)

IREP

- nombre ≠ null ∧ nombre ≠ “”
- direccion ≠ null ∧ direccion ≠ “”
- capacidadMaxima > 0
- cantidadPalcos ≥ 0 (si se utiliza)
- funcionesPorFecha es un diccionario tal que:
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claves(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo // LocalDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2 // f1 = fecha1 f2 = fecha2

TAD Teatro:

Datos

- capacidadMaxima
- nombre
- direccion
- funcionesPorFecha: diccionario <LocalDate, String>
- asientosPorFila
- sectores
- capacidad
- porcentajeAdicional

Operaciones

- teatro (nombre, direccion, capacidadMaxima, asientosPorFila, sectores, capacidad, porcentajeAdicional)
- sectores ()
- capacidadSectores (): int
- porcentajeAdicional (sector): int

IREP

- nombre ≠ null ∧ nombre ≠ “”

- direccion ≠ null ∧ direccion ≠ “”
- capacidadMaxima, asientosPorFila > 0
- sectores ≠ null ∧ |sectores| > 0
- capacidad ≠ null ∧ |capacidad| = |sectores| ∧ ∀ c ∈ capacidad: c > 0
- porcentajeAdicional ≠ null ∧ |porcentajeAdicional| = |sectores| ∧ ∀ p ∈ porcentajeAdicional: p ≥ 0
- funcionesPorFecha es un diccionario tal que:
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claves(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo // LocalDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2 // f1 = fecha1 f2 = fecha2

Implementación

StringBuilder

```java
@Override
public String listarFunciones(String nombreEspectaculo) {
    Espectaculo espectaculo = espectaculos.get(nombreEspectaculo);
    if (espectaculo == null) throw new RuntimeException(message:"Espectáculo no encontrado");
    StringBuilder sb = new StringBuilder();
    // Ordenar las fechas
    List<LocalDate> fechas = new ArrayList<>(espectaculo.getFunciones().keySet());
    fechas.sort(LocalDate::compareTo);
    for (LocalDate fecha : fechas) {
        Funcion funcion = espectaculo.getFunciones().get(fecha);
        Sede sede = sedes.get(funcion.sedeObj().nombreSede());
        String fechaStr = fecha.format(dateFormatter);
        sb.append(funcion.descripcionParaListado(sede, fechaStr));
        sb.append(str:"\n");
    }
    return sb.toString();
}
```

Para el método **public String listarFunciones(String nombreEspectaculo)** en la clase **Ticketek**, utilizamos el concepto de **StringBuilder** (es una clase de Java que permite construir y modificar cadenas de texto de manera eficiente, especialmente útil cuando se realizan múltiples concatenaciones, ya que evita crear muchos objetos intermedios como ocurre con el uso repetido del operador + en cadenas) para construir cadenas de texto de manera eficiente, especialmente cuando se necesita concatenar múltiples fragmentos de texto dentro de bucles o procesos repetitivos; implementando un **StringBuilde**r llamado **sb** para construir eficientemente un texto que representa la lista de funciones de un espectáculo, recorriendo todas las fechas de las funciones del espectáculo, ordenadas. En el método para cada función:

- Se obtiene la descripción de la función usando **funcion.descripcionParaListado(sede, fechaStr)**
- Se agrega esa descripción al **StringBuilder** con **.append(...)**
- Se agrega un salto de línea con **.append(“\n”)**

Al final, convertimos el contenido del **StringBuilder** a un **String** con **.toString()** y se retorna; devolviendo un único **String** con todas las funciones listadas, separadas por saltos de línea

Iterador

```java
@Override
public List<IEntrada> listarEntradasEspectaculo(String nombreEspectaculo) {
    List<IEntrada> entradas = new ArrayList<>();
    for (Usuario usuario : usuarios.values()) {
        Iterator<Entrada> it = usuario.getEntradas().values().iterator();
        while (it.hasNext()) {
            Entrada entrada = it.next();
            if (entrada.espectaculo().nombre().equals(nombreEspectaculo)) {
                entradas.add(entrada);
            }
        }
    }
    return entradas;
}
```

El concepto de **Iterador** (un objeto que permite recorrer los elementos de una colección como una lista, conjunto o mapa) uno a uno, sin exponer su estructura interna. Proporciona métodos como **hasNext()** para saber si hay más elementos y **next()** para obtener el siguiente elemento. Es útil para procesar o filtrar datos de manera secuencial y segura) lo utilizamos para recorrer todas las entradas compradas por todos los usuarios y filtrar aquellas que corresponden al espectáculo solicitado. Dentro del método, para cada usuario, se obtiene el mapa de sus entradas (**usuario.getEntradas().values()**) y se crea un **iterador** sobre esa colección.

El iterador nos permite recorrer todas las entradas de cada usuario de manera eficiente y segura, sin exponer la estructura interna de la colección. Así, se pueden procesar todas las entradas y seleccionar solo las que pertenecen al espectáculo buscado. Decidimos implementar Iterador en este método ya que de esa forma evita errores de concurrencia o modificación durante la iteración y permite recorrer cualquier colección que implemente Iterable

Foreach

```java
@Override
public List<IEntrada> listarEntradasFuturas(String email, String contrasenia) {
    Usuario usuario = usuarios.get(email);
    if (usuario == null) throw new RuntimeException(message:"Usuario no encontrado");
    usuario.validarEmail(email);
    usuario.validarContrasenia(contrasenia);
    List<IEntrada> futuras = new ArrayList<>();
    LocalDate hoy = LocalDate.now();
    for (Entrada entrada : usuario.getEntradas().values()) {
        if (entrada.esFutura(hoy)) {
            futuras.add(entrada);
        }
    }
    return futuras;
}
```

Para el método public **List<IEntrada> listarEntradasFuturas(String email, String contrasenia)** en la clase **Ticketek**, utilizamos el concepto de **Foreach** (el cual consiste en una estructura de control que permite recorrer todos los elementos de una colección como por ejemplo un array, lista o map, de manera sencilla y directa, sin necesidad de usar índices o variables de control) para recorrer los objetos del tipo Entrada de manera sencilla y legible de la siguiente manera:

- Se obtiene el usuario correspondiente al email recibido como parámetro.
- Se utiliza un foreach para recorrer todas las entradas que tiene ese usuario:
- Si la entrada corresponde a una fecha futura (usando el método **esFutura(hoy)**), se agrega a la lista de resultados.

Por lo que, de esta manera, gracias a cómo implementamos el **Foreach**, logramos recorrer todas las entradas del usuario de forma clara, sin necesidad de usar índices ni métodos complicados, filtrar fácilmente solo las entradas que corresponden a fechas futuras y evitar errores de manejo de índices que de otra forma podrían llegar a surgir, haciendo el código más seguro y fácil de mantener.

Herencia y Polimorfismo

```java
public class MiniEstadio extends Sede {
    private int cantidadPuestos;
    private double precioConsumicion;
    private int asientosPorFila;    The value of the field MiniEstadio.asientosPorFila is not used
    private String[] sectores;
    private int[] capacidad;
    private int[] porcentajeAdicional;

    public MiniEstadio(String nombre, String direccion, int capacidadMaxima, int asientosPorFila, int cantidadPuestos,
    double precioConsumicion, String[] sectores, int[] capacidad, int[] porcentajeAdicional) {
        super(nombre, direccion, capacidadMaxima);
        this.asientosPorFila = asientosPorFila;
        this.cantidadPuestos = cantidadPuestos;
        this.precioConsumicion = precioConsumicion;
        this.sectores = sectores;
        this.capacidad = capacidad;
        this.porcentajeAdicional = porcentajeAdicional;
    }

    // Métodos específicos de MiniEstadio
    public String[] sectores() { return sectores; }
    public int[] capacidadSectores() { return capacidad; }
    public int porcentajeAdicional(String sector) {
        for (int i = 0; i < sectores.length; i++) {
            if (sectores[i].equalsIgnoreCase(sector)) return porcentajeAdicional[i];
        }
        return 0;
    }
    public double precioConsumicion() { return precioConsumicion; }
    public int cantidadPuestos() { return cantidadPuestos; }
}
```

Mediante la herencia, hicimos que **MiniEstadio** (y a su vez **Estadio** y **Teatro**) reutilice atributos y métodos comunes de la clase abstracta **Sede** (una clase abstracta es una clase que no puede ser instanciada directamente y sirve como base para otras clases. Puede contener métodos abstractos que deben ser implementados por las subclases, así como métodos concretos. Se utiliza para definir un comportamiento común y obligatorio para todas las clases que la extienden), como **nombre, direccion, capacidadMaxima**, y la gestión de funciones y puede agregar atributos y métodos propios (como cantidadPuestos, precioConsumicion, sectores, etc.) que son específicos de los miniestadios. A su vez, como **MiniEstadio** es un subtipo de **Sede**, puede ser tratado como un objeto de tipo **Sede** de la siguiente manera:

Sede sede = new MiniEstadio(...);

Esto permite que métodos que reciben o manipulan objetos de tipo **Sede** puedan trabajar también con instancias de **MiniEstadio, Teatro, Estadio,** etc. Por lo que si se sobrescriben métodos de **Sede** en **MiniEstadio** (polimorfismo), el método sobrescrito será el que se ejecute, aunque el objeto se maneje como Sede. Por ejemplo, en el método **descripcionParaListado** de la clase **Funcion** se usa:

if (sede instanceof MiniEstadio mini) {

// Se accede a métodos específicos de MiniEstadio

}

Sobreescritura, sobrecarga e interfaz

En la clase Ticketek utilizamos el concepto de **sobrecarga** (el cual ocurre cuando hay varios métodos con el mismo nombre pero distinta lista de parámetros, como tipo, cantidad u orden), lo cual nos permite ofrecer varias formas de realizar una acción similar, adaptándose a distintos casos de uso, como por ejemplo con los métodos para registrar sede:

void registrarSede(String nombre, String direccion, int capacidadMaxima);

void registrarSede(String nombre, String direccion, int capacidadMaxima, int asientosPorFila, String[] sectores, int[] capacidad, int[] porcentajeAdicional);

void registrarSede(String nombre, String direccion, int capacidadMaxima, int asientosPorFila, int cantidadPuestos, double precioConsumicion, String[] sectores, int[] capacidad, int[] porcentajeAdicional);

La sobreescritura (la cual ocurre cuando una subclase redefine un método heredado de una superclase o interfaz, cambiando su implementación pero manteniendo la misma firma) la usamos principalmente en la clase **Ticketek** para reescribir métodos de la interfaz **ITicketek**, permitiéndonos personalizar o especializar el comportamiento de métodos heredados para adaptarlos a las necesidades de la subclase como hicimos con los siguientes métodos usando la notación @Override:

```
@Override // Estadio
public void registrarSede(String nombreSede, String direccion, int capacidadMaxima) {
```

```
@Override
public List<IEntrada> venderEntrada(String nombreEspectaculo, String fecha, String email, String contrasenia, int cantidadEntradas) {
```

```
@Override
public String listarFunciones(String nombreEspectaculo) {
```

Respecto a las interfaces, en nuestro proyecto la materia nos brindó las interfaces **ITicketek** e **IEntrada**, que definen el “contrato” público que debe cumplir nuestro sistema de venta de entradas. Esto significa que todas las clases que implementan estas interfaces deben proveer una implementación concreta de los métodos definidos en ellas. Por ejemplo, en nuestra clase **Ticketek** implementamos la interfaz **ITicketek** de la siguiente manera:

```
public class Ticketek implements ITicketek {
    // Implementación de todos los métodos definidos en ITicketek
}
```

La diferencia principal es que la interfaz define el “qué” (qué operaciones debe soportar el sistema, como registrar usuarios, vender entradas, etc.) y la clase define el “cómo” (cómo se realiza cada operación internamente, por ejemplo, utilizando HashMap, validaciones, etc.).

Gracias a la utilización de la interfaz, pudimos separar la definición del comportamiento de la implementación concreta, garantizando que nuestra clase **Ticketek** cumpla con todos los métodos requeridos por el enunciado. Lo mismo ocurre con la interfaz **IEntrada**, que define el comportamiento esperado de una entrada, y nuestra clase Entrada implementa esa interfaz.

En resumen, el uso de interfaces nos permitió estructurar el sistema de manera flexible y cumplir con los requisitos funcionales definidos por la materia

Complejidad

```java
@Override
public boolean anularEntrada(IEntrada entrada, String contrasenia) {
    if (entrada == null || contrasenia == null) throw new RuntimeException(message:"Datos inválidos");
    if (!(entrada instanceof Entrada)) throw new RuntimeException(message:"Tipo de entrada inválido");
    Entrada ent = (Entrada) entrada;
    Usuario usuario = usuarios.get(ent.usuario());
    if (usuario == null) throw new RuntimeException(message:"Usuario no encontrado");
    usuario.validarContrasenia(contrasenia);
    if (!ent.esFutura(LocalDate.now())) return false;
    // Cambia aquí: si no existe, lanza excepción
    if (!usuario.getEntradas().containsKey(ent.codigo()))
        throw new RuntimeException(message:"La entrada no existe");
    boolean eliminado = usuario.eliminarEntrada(ent.codigo());
    // Si maneja asientos(es numerada), libera el asiento
    Espectaculo espectaculo = ent.espectaculo();
    if (espectaculo != null) {
        Funcion funcion = espectaculo.getFunciones().get(ent.fecha());
        if (funcion != null) {
            funcion.liberarAsientoSiCorresponde(ent.sector(), ent.asiento());
        }
    }
    return eliminado;
}
```

El método **anularEntrada** es O(1) en álgebra de órdenes, ya que todas las operaciones principales se realizan en tiempo constante gracias al uso de **HashMap** y acceso directo por clave. es decir, Si todas las estructuras involucradas (usuarios, entradas, funciones, asientos) son **HashMap** o equivalentes.

entonces, **anularEntrada** es O(1) porque:

1. Al buscar un usuario utilizando **Usuario usuario = usuarios.get(ent.usuario());** usuarios es un **HashMap <String, Usuario>** y la operación **.get(email)** es O(1) porque accede directamente a la posición calculada por la función hash.

2. Al verificar la existencia de una entrada mediante **if (!usuario.getEntradas().containsKey(ent.codigo()))**, **.getEntradas** retorna entradas, que es un **HashMap<String, Entrada>** y la operación **.containsKey(codigo)** es O(1).

3. Al eliminar la entrada del usuario mediante **boolean eliminado = usuario.eliminarEntrada(ent.codigo());** el cual eliminarEntrada internamente hace un **.remove(codigo)** sobre el HashMap de entradas, que es O(1).

4. al buscar la función **Funcion funcion = espectaculo.getFunciones().get(ent.fecha());** funciones es un **HashMap<LocalDate, funcion**> y **.get(fecha)** es O(1).

5. Al liberar un asiento utilizando **funcion.liberarAsientoSiCorresponde(ent.sector(), ent.asiento**()); el cual **liberarAsientoSiCorresponde** internamente accede a un HashMap y a un array por índice las cuales ambas son O(1).

Al sumar todas las operaciones principales:

- O(1) (buscar usuario) (Regla 4: O(k) = O(1) para k constante)
- O(1) (verificar entrada)
- O(1) (eliminar entrada)
- O(1) (buscar función)
- O(1) (liberar asiento)

Por álgebra de órdenes, la suma de operaciones O(1) sigue siendo O(1), por lo que el método completo es O(1) en el caso promedio y esto lo podemos deducir gracias a las reglas del algebra de órdenes si utilizamos la regla 2, tenemos que: O(f) + O(g) = O(max{f, g})

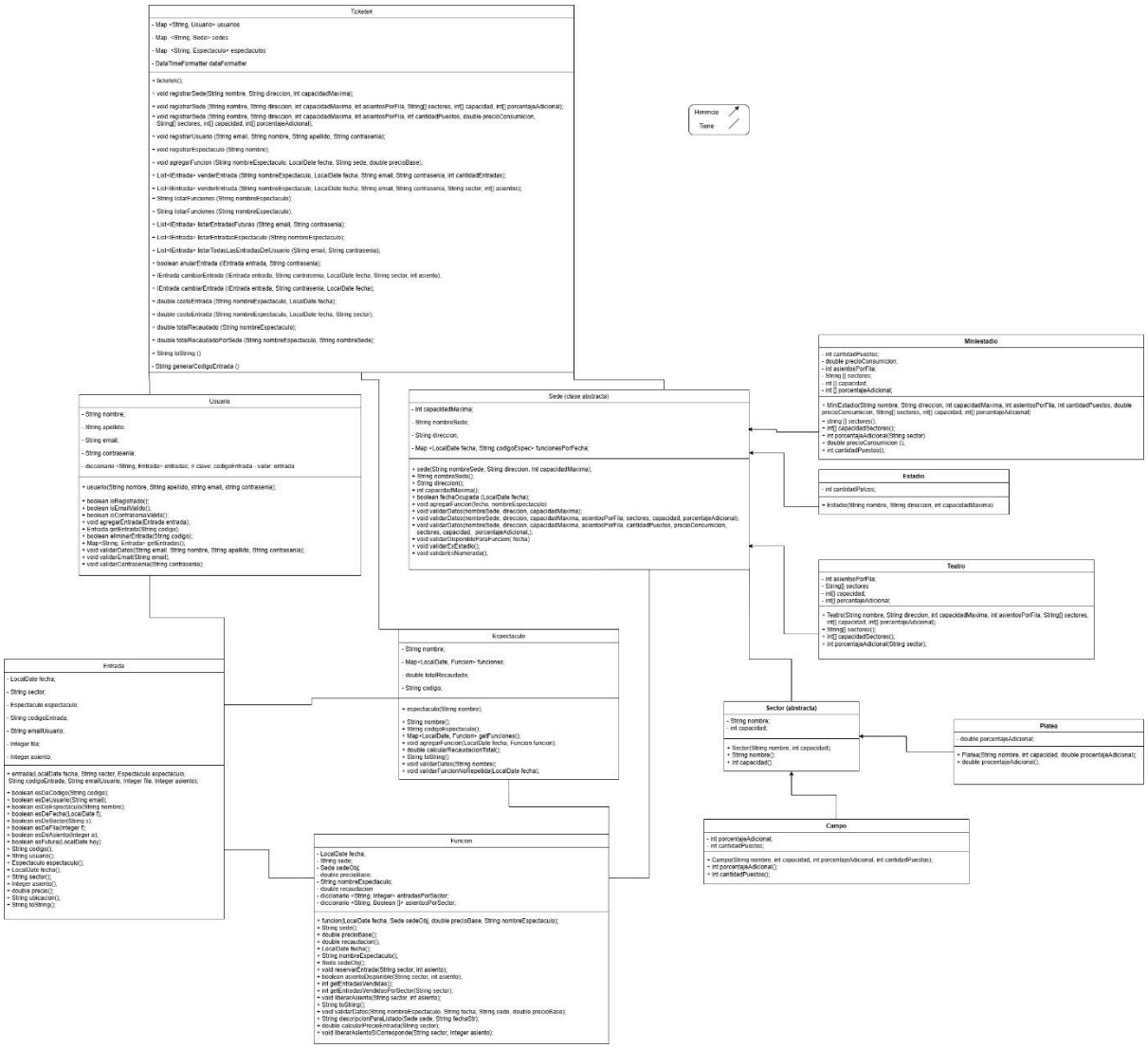
En este caso: O(1) + O(1) + O(1) + O(1) + O(1) = O(max{1, 1, 1, 1, 1}) = O(1)

Multiplicación por constante (Regla 4) Regla 4 (Multiplicación por constante): Si el número de operaciones es constante, su suma sigue siendo O (1):

O(k * f) = O(f) si k es constante

Debido a que todas las operaciones involucradas se ejecutan en tiempo constante O(1), la sumatoria de las mismas conserva dicha complejidad. Aplicando las reglas del álgebra de órdenes, queda demostrado que el método anularEntrada es eficiente y tiene una complejidad O(1). Esto garantiza una ejecución óptima sin importar el tamaño de los datos involucrados.

Diseño


Herencia
Tiene
Ticketek
- Map <String, Usuario> usuarios
- Map <String, Sede> sedes
- Map <String, Espectaculo> espectaculos
- DateTimeFormatter dateFormatter
+ ticketek();
- void registrarSede(String nombre, String direccion, int capacidadMaxima);
+ void registrarSede (String nombre, String direccion, int capacidadMaxima, int asientosPorFila, String[] sectores, int[] capacidad, int[] porcentajeAdicional);
+ void registrarSede (String nombre, String direccion, int capacidadMaxima, int asientosPorFila, int cantidadPuestos, double precioConsumicion,
String[] sectores, int[] capacidad, int[] porcentajeAdicional);
- void registrarUsuario (String email, String nombre, String apellido, String contrasenia);
- void registrarEspectaculo (String nombre);
- void agregarFuncion (String nombreEspectaculo, LocalDate fecha, String sede, double precioBase);
+ List<IEntrada> venderEntrada (String nombreEspectaculo, LocalDate fecha, String email, String contrasenia, int cantidadEntradas);
+ List<IEntrada> venderEntrada (String nombreEspectaculo, LocalDate fecha, String email, String contrasenia, String sector, int[] asientos);
+ String listarFunciones (String nombreEspectaculo);
- String listarFunciones (String nombreEspectaculo);
- List<IEntrada> listarEntradasFuturas (String email, String contrasenia);
+ List<IEntrada> listarEntradasEspectaculo (String nombreEspectaculo);
- List<IEntrada> listarTodasLasEntradasDelUsuario (String email, String contrasenia);
+ boolean anularEntrada (IEntrada entrada, String contrasenia);
+ IEntrada cambiarEntrada (IEntrada entrada, String contrasenia, LocalDate fecha, String sector, int asiento);
+ IEntrada cambiarEntrada (IEntrada entrada, String contrasenia, LocalDate fecha);
+ double costoEntrada (String nombreEspectaculo, LocalDate fecha);
+ double costoEntrada (String nombreEspectaculo, LocalDate fecha, String sector);
+ double totalRecaudado (String nombreEspectaculo);
+ double totalRecaudadoPorSede (String nombreEspectaculo, String nombreSede);
+ String toString ()
- String generarCodigoEntrada ()
Usuario
- String nombre;
- String apellido;
- String email;
- String contrasenia;
- diccionario <String, Entrada> entradas; // clave: codigoEntrada - valor: entrada
+ usuario(String nombre, String apellido, string email, string contrasenia);
+ boolean isRegistrado();
+ boolean isEmailValido();
+ boolean isContrasenaValida();
+ void agregarEntrada(Entrada entrada);
+ Entrada getEntrada(String codigo);
+ boolean eliminarEntrada(String codigo);
+ Map<String, Entrada> getEntradas();
+ void validarDatos(String email, String nombre, String apellido, String contrasenia);
+ void validarEmail(String email);
+ void validarContrasenia(String contrasenia);
Sede (clase abstracta)
- int capacidadMaxima;
- String nombreSede;
- String direccion;
- Map <LocalDate fecha, String codigoEspec> funcionesPorFecha;
+ sede(String nombreSede, String direccion, int capacidadMaxima);
+ String nombreSede();
+ String direccion();
+ int capacidadMaxima();
+ boolean fechaOcupada (LocalDate fecha);
+ void agregarFuncion(fecha, nombreEspectaculo)
+ void validarDatos(nombreSede, direccion, capacidadMaxima);
+ void validarDatos(nombreSede, direccion, capacidadMaxima, asientosPorFila, sectores, capacidad, porcentajeAdicional),
+ void validarDatos(nombreSede, direccion, capacidadMaxima, asientosPorFila, cantidadPuestos, precioConsumicion,
sectores, capacidad, porcentajeAdicional,);
+ void validarDisponibleParaFuncion(fecha)
+ void validarEsEstadio();
+ void validarEsNumerada();
Miniestadio
- int cantidadPuestos;
- double precioConsumicion;
- int asientosPorFila;
- String [] sectores;
- int [] capacidad;
- int [] porcentajeAdicional;
+ MiniEstadio(String nombre, String direccion, int capacidadMaxima, int asientosPorFila, int cantidadPuestos, double
precioConsumicion, String[] sectores, int[] capacidad, int[] porcentajeAdicional)
+ string [] sectores();
+ int[] capacidadSectores();
+ int porcentajeAdicional(String sector)
+ double precioConsumicion ();
+ int cantidadPuestos();
Estadio
- int cantidadPalcos;
+ Estadio(String nombre, String direccion, int capacidadMaxima)
Teatro
- int asientosPorFila;
- String[] sectores
- int[] capacidad,
- int[] porcentajeAdicional;
+ Teatro(String nombre, String direccion, int capacidadMaxima, int asientosPorFila, String[] sectores,
int[] capacidad, int[] porcentajeAdicional);
+ string[] sectores();
+ int[] capacidadSectores();
+ int porcentajeAdicional(String sector),
Entrada
- LocalDate fecha;
- String sector;
- Espectaculo espectaculo;
- String codigoEntrada;
- String emailUsuario;
- Integer fila;
- Integer asiento;
+ entrada(LocalDate fecha, String sector, Espectaculo espectaculo,
String codigoEntrada, String emailUsuario, Integer fila, Integer asiento),
+ boolean esDeCodigo(String codigo);
+ boolean esDeUsuario(String email);
+ boolean esDeEspectaculo(String nombre);
+ boolean esDeFecha(LocalDate f);
+ boolean esDeSector(String s);
+ boolean esDeFila(Integer f);
+ boolean esDeAsiento(Integer a);
+ boolean esFutura(LocalDate hoy);
+ String codigo();
+ String usuario();
+ Espectaculo espectaculo();
+ LocalDate fecha();
+ String sector();
+ Integer asiento();
+ double precio();
+ String ubicacion();
+ String toString();
Espectaculo
- String nombre;
- Map<LocalDate, Funcion> funciones;
- double totalRecaudado;
- String codigo;
+ espectaculo(String nombre);
+ String nombre();
+ String codigoEspectaculo();
+ Map<LocalDate, Funcion> getFunciones();
+ void agregarFuncion(LocalDate fecha, Funcion funcion);
+ double calcularRecaudacionTotal();
+ String toString();
+ void validarDatos(String nombre);
+ void validarFuncionNoRepetida(LocalDate fecha);
Funcion
- LocalDate fecha;
- String sede;
- Sede sedeObj;
- double precioBase;
- String nombreEspectaculo;
- double recaudacion
- diccionario <String, Integer> entradasPorSector;
- diccionario <String, Boolean []> asientosPorSector;
+ funcion(LocalDate fecha, Sede sedeObj, double precioBase, String nombreEspectaculo);
+ String sede();
+ double precioBase();
+ double recaudacion();
+ LocalDate fecha();
+ String nombreEspectaculo();
+ Sede sedeObj();
+ void reservarEntrada(String sector, int asiento);
+ boolean asientoDisponible(String sector, int asiento);
+ int getEntradasVendidas();
+ int getEntradasVendidasPorSector(String sector);
+ void liberarAsiento(String sector, int asiento);
+ String toString();
+ void validarDatos(String nombreEspectaculo, String fecha, String sede, double precioBase);
+ String descripcionParaListado(Sede sede, String fechaStr);
+ double calcularPrecioEntrada(String sector);
+ void liberarAsientoSiCorresponde(String sector, Integer asiento);
Sector (abstracta)
- String nombre;
- int capacidad;
+ Sector(String nombre, int capacidad);
+ String nombre();
+ int capacidad();
Platea
- double porcentajeAdicional;
+ Platea(String nombre, int capacidad, double procentajeAdicional);
+ double procentajeAdicional();
Campo
- int porcentajeAdicional;
- int cantidadPuestos;
+ Campo(String nombre, int capacidad, int porcentajeAdicional, int cantidadPuestos);
+ int porcentajeAdicional();
+ int cantidadPuestos();