Universidad Nacional de General Sarmiento

Programación II – Comisión 3

Trabajo Práctico Integrador – Segunda parte

Docentes:

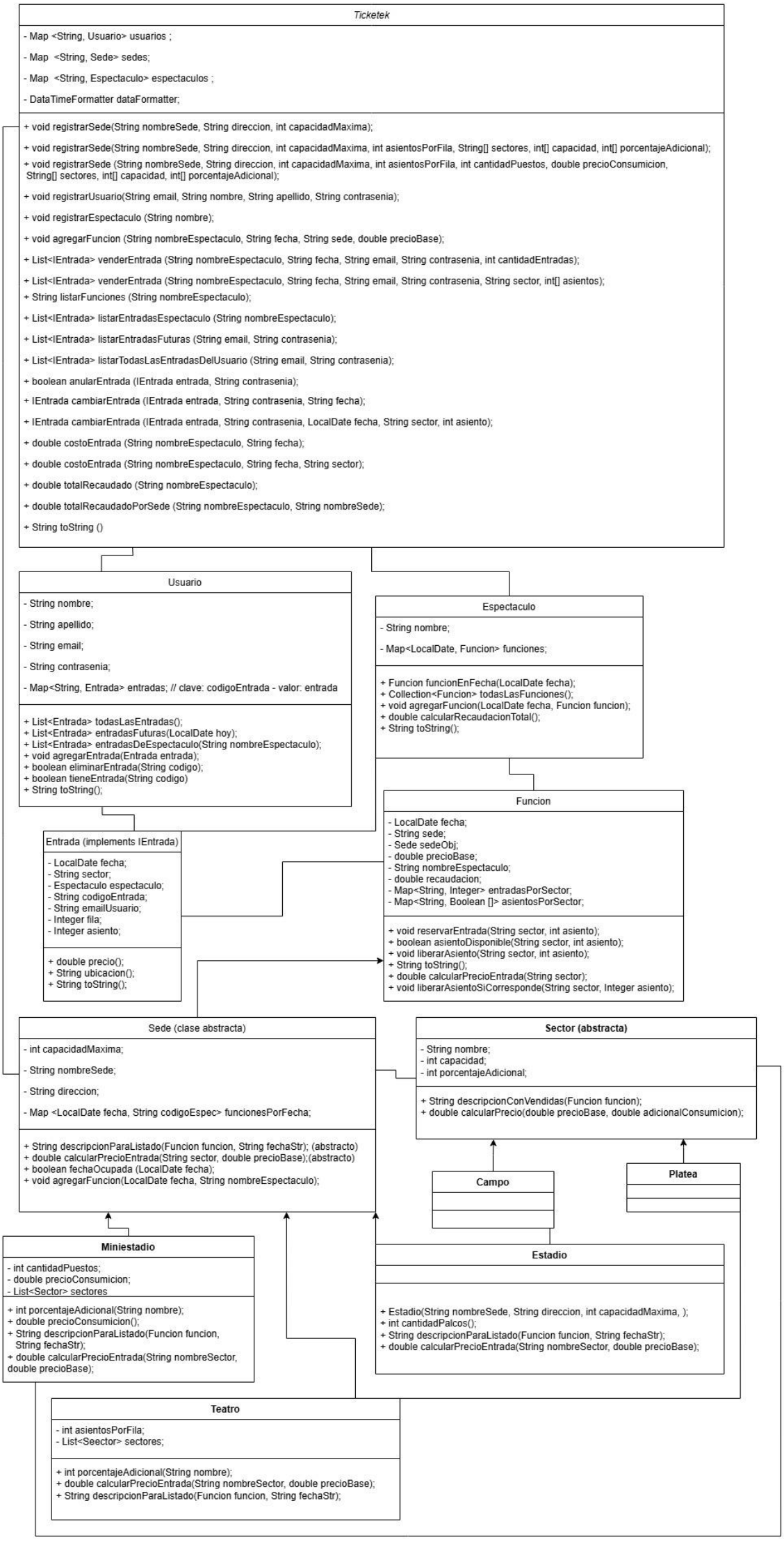
- Nores, José
- Gabrielli, Miguel Ángel

Integrantes:

- Cordua, Kevin Nicolas Daniel
- Chazarreta, Jeremías Vladimir

2025C1

Diseño


Ticketek
- Map <String, Usuario> usuarios ;
- Map <String, Sede> sedes;
- Map <String, Espectaculo> espectaculos ;
- DataTimeFormatter dataFormatter;
+ void registrarSede(String nombreSede, String direccion, int capacidadMaxima);
+ void registrarSede(String nombreSede, String direccion, int capacidadMaxima, int asientosPorFila, String[] sectores, int[] capacidad, int[] porcentajeAdicional);
+ void registrarSede (String nombreSede, 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, String fecha, String sede, double precioBase);
+ List<IEntrada> venderEntrada (String nombreEspectaculo, String fecha, String email, String contrasenia, int cantidadEntradas);
+ List<IEntrada> venderEntrada (String nombreEspectaculo, String fecha, String email, String contrasenia, String sector, int[] asientos);
+ String listarFunciones (String nombreEspectaculo);
+ List<IEntrada> listarEntradasEspectaculo (String nombreEspectaculo);
+ List<IEntrada> listarEntradasFuturas (String email, String contrasenia);
+ List<IEntrada> listarTodasLasEntradasDelUsuario (String email, String contrasenia);
+ boolean anularEntrada (IEntrada entrada, String contrasenia);
+ IEntrada cambiarEntrada (IEntrada entrada, String contrasenia, String fecha);
+ IEntrada cambiarEntrada (IEntrada entrada, String contrasenia, LocalDate fecha, String sector, int asiento);
+ double costoEntrada (String nombreEspectaculo, String fecha);
+ double costoEntrada (String nombreEspectaculo, String fecha, String sector);
+ double totalRecaudado (String nombreEspectaculo);
+ double totalRecaudadoPorSede (String nombreEspectaculo, String nombreSede);
+ String toString ()
Usuario
- String nombre;
- String apellido;
- String email;
- String contrasenia;
- Map<String, Entrada> entradas; // clave: codigoEntrada - valor: entrada
+ List<Entrada> todasLasEntradas();
+ List<Entrada> entradasFuturas(LocalDate hoy);
+ List<Entrada> entradasDeEspectaculo(String nombreEspectaculo);
+ void agregarEntrada(Entrada entrada);
+ boolean eliminarEntrada(String codigo);
+ boolean tieneEntrada(String codigo)
+ String toString();
Espectaculo
- String nombre;
- Map<LocalDate, Funcion> funciones;
+ Funcion funcionEnFecha(LocalDate fecha);
+ Collection<Funcion> todasLasFunciones();
+ void agregarFuncion(LocalDate fecha, Funcion funcion);
+ double calcularRecaudacionTotal();
+ String toString();
Funcion
- LocalDate fecha;
- String sede;
- Sede sedeObj;
- double precioBase;
- String nombreEspectaculo;
- double recaudacion;
- Map<String, Integer> entradasPorSector;
- Map<String, Boolean []> asientosPorSector;
+ void reservarEntrada(String sector, int asiento);
+ boolean asientoDisponible(String sector, int asiento);
+ void liberarAsiento(String sector, int asiento);
+ String toString();
+ double calcularPrecioEntrada(String sector);
+ void liberarAsientoSiCorresponde(String sector, Integer asiento);
Entrada (implements IEntrada)
- LocalDate fecha;
- String sector;
- Espectaculo espectaculo;
- String codigoEntrada;
- String emailUsuario;
- Integer fila;
- Integer asiento;
+ double precio();
+ String ubicacion();
+ String toString();
Sede (clase abstracta)
- int capacidadMaxima;
- String nombreSede;
- String direccion;
- Map <LocalDate fecha, String codigoEspec> funcionesPorFecha;
+ String descripcionParaListado(Funcion funcion, String fechaStr); (abstracto)
+ double calcularPrecioEntrada(String sector, double precioBase);(abstracto)
+ boolean fechaOcupada (LocalDate fecha);
+ void agregarFuncion(LocalDate fecha, String nombreEspectaculo);
Sector (abstracta)
- String nombre;
- int capacidad;
- int porcentajeAdicional;
+ String descripcionConVendidas(Funcion funcion);
+ double calcularPrecio(double precioBase, double adicionalConsumicion);
Campo
Platea
Miniestadio
- int cantidadPuestos;
- double precioConsumicion;
- List<Sector> sectores
+ int porcentajeAdicional(String nombre);
+ double precioConsumicion();
+ String descripcionParaListado(Funcion funcion, String fechaStr);
+ double calcularPrecioEntrada(String nombreSector, double precioBase);
Estadio
+ Estadio(String nombreSede, String direccion, int capacidadMaxima,);
+ int cantidadPalcos();
+ String descripcionParaListado(Funcion funcion, String fechaStr);
+ double calcularPrecioEntrada(String nombreSector, double precioBase);
Teatro
- int asientosPorFila;
- List<Seector> sectores;
+ int porcentajeAdicional(String nombre);
+ double calcularPrecioEntrada(String nombreSector, double precioBase);
+ String descripcionParaListado(Funcion funcion, String fechaStr);

IREP de clases

IREP de Ticketek

- usuarios es un diccionario Map<String, Usuario> tal que:
  - ∀ e ∈ claves(usuarios): e ≠ null ∧ e ≠ "" // e = email
  - ∀ u ∈ valores(usuarios): u ≠ null // u = Usuario
  - ∀ e1, e2 ∈ claves(usuarios): e1 ≠ e2
  - 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 Map<String, Sede> 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
- espectaculos es un diccionario Map<String, Espectaculo> 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 Funcion
    - ∀ f ∈ claves(e.getFunciones()): f.equals(fun.fecha()) donde fun = e.getFunciones().get(f)
    - ∀ f1, f2 ∈ claves(e.getFunciones()): f1 ≠ f2
- dateFormatter usa el patrón "dd/MM/yy"

IREP de Usuario

- nombre, apellido, email, contrasenia ≠ null ∧ ≠ ""
- email contiene '@'
- |contrasenia| ≥ 4
- entradas es un diccionario Map<String, Entrada> 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

IREP de Entrada

- fecha ≠ null
- espectaculo ≠ null
- codigoEntrada ≠ null ∧ ≠ ""
- emailUsuario ≠ null ∧ ≠ ""
- Si sector ≠ null ∧ sector ≠ "" ∧ sector ≠ "CAMPO" (ignorando mayúsculas/minúsculas) ⇒ asiento ≠ null
- Si sector = "CAMPO" (ignorando mayúsculas/minúsculas) ⇒ (asiento = null ∨ asiento = -1)

IREP de Espectaculo

- 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
- Para todo f ∈ valores(funciones): f.recaudacion() ≥ 0
- La suma de todas las recaudaciones (calcularRecaudacionTotal()) ≥ 0

IREP de Función

- 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 (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 (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

IREP de Platea

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

IREP de Campo

- nombre ≠ null ∧ nombre ≠ “” (siempre es "CAMPO")
- capacidad > 0
- porcentajeAdicional ≥ 0

IREP de Miniestadio

- nombre ≠ null ∧ nombre ≠ ""
- direccion ≠ null ∧ direccion ≠ ""
- capacidadMaxima > 0
- cantidadPuestos > 0
- precioConsumicion > 0
- sectores ≠ null ∧ |sectores| > 0
- Para cada **s** ∈ sectores:
    - s ≠ null
    - s.getCapacidad() > 0
    - s.getPorcentajeAdicional() ≥ 0
- El mapa de funciones por fecha (heredado de Sede/Espectaculo):
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claves(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo LocalDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2

IREP de Estadio

- nombre ≠ null ∧ nombre ≠ “”
- direccion ≠ null ∧ direccion ≠ “”
- capacidadMaxima > 0
- El mapa de funciones por fecha (heredado de Sede/Espectaculo):
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claves(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo LocalDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2

IREP de Teatro

- nombre ≠ null ∧ nombre ≠ “”
- direccion ≠ null ∧ direccion ≠ “”
- capacidadMaxima > 0
- asientosPorFila > 0
- sectores ≠ null ∧ |sectores| > 0
- Para cada s ∈ sectores:
    - s ≠ null
    - s. getCapacidad() > 0
    - s. getPorcentajeAdicional() ≥ 0
- El mapa de funciones por fecha (heredado de Sede/Espectaculo):
    - funcionesPorFecha ≠ null
    - ∀ f ∈ claver(funcionesPorFecha): f ≠ null // f = fecha de la función, tipo localDate
    - ∀ f1, f2 ∈ claves(funcionesPorFecha): f1 ≠ f2

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();
    List<LocalDate> fechas = new ArrayList<>();
    for (Funcion f : espectaculo.todasLasFunciones()) {
        fechas.add(f.fecha());
    }
    fechas.sort(LocalDate::compareTo);
    for (LocalDate fecha : fechas) {
        Funcion funcion = espectaculo.funcionEnFecha(fecha);
        String fechaStr = fecha.format(dateFormatter);
        sb.append(funcion.sedeObj().descripcionParaListado(funcion, fechaStr));
        sb.append(str:"\n");
    }
    return sb.toString();
}
```

En la clase **Ticketek**, se utiliza el concepto de **StringBuilder** para construir cadenas de texto de manera eficiente, especialmente útil cuando se deben concatenar múltiples fragmentos de texto dentro de bucles.

En este método, se crea un objeto **StringBuilder** llamado **sb** para ir armando el texto que representará la lista de funciones de un espectáculo. Primero, se obtienen todas las fechas de las funciones del espectáculo y se ordenan. Luego, para cada fecha:

- Se obtiene la función correspondiente y se formatea la fecha.
- Se llama al método **descripcionParaListado** de la sede de la función, pasando la función y la fecha formateada, para obtener la descripción textual de esa función.
- Se agrega esa descripción al **StringBuilder** usando .append(...).
- Se agrega un salto de línea con .append("\n").

Al finalizar el recorrido, se convierte el contenido del StringBuilder a un String usando .toString() y se retorna.
De esta manera, se obtiene un único string con todas las funciones listadas, separadas por saltos de línea, evitando la creación de múltiples objetos intermedios y mejorando la eficiencia en la construcción de la cadena de texto.

Iterador

```java
@SuppressWarnings("UseOfIteratorForSimpleLoop")
@Override
public List<IEntrada> listarEntradasEspectaculo(String nombreEspectaculo) {
    List<IEntrada> entradas = new ArrayList<>();
    for (Usuario usuario : usuarios.values()) {
        // Usamos Iterator para recorrer las entradas del usuario
        Iterator<Entrada> it = usuario.todasLasEntradas().iterator();
        while (it.hasNext()) {
            Entrada entrada = it.next();
            if (entrada.espectaculo().nombre().equals(nombreEspectaculo)) {
                entradas.add(entrada);
            }
        }
    }
    return entradas;
}
```

En la clase **Ticketek**, se utiliza el concepto de **Iterador** para recorrer todas las entradas compradas por todos los usuarios y filtrar aquellas que corresponden al espectáculo solicitado.

Para cada usuario registrado, se obtiene la lista de sus entradas **(usuario.todasLasEntradas())** y se crea un iterador sobre esa colección. El iterador **(Iterator<Entrada> it)** permite recorrer cada entrada de manera secuencial usando los métodos **hasNext()** (para verificar si hay más elementos) y **next()** (para obtener el siguiente elemento).

De esta forma, se procesan todas las entradas de cada usuario sin exponer la estructura interna de la colección y de manera segura, ya que el uso de Iterador ayuda a evitar errores de concurrencia o modificación durante la iteración. Además, permite recorrer cualquier colección que implemente la interfaz Iterable, haciendo el código más flexible y desacoplado de la implementación concreta de la colección.

En cada iteración, si la entrada corresponde al espectáculo buscado, se agrega a la lista de resultados. Así, el método devuelve todas las entradas del espectáculo solicitado, utilizando Iterador para garantizar una iteración eficiente y segura.

Foreach

```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();
    List<LocalDate> fechas = new ArrayList<>();
    for (Funcion f : espectaculo.todasLasFunciones()) {
        fechas.add(f.fecha());
    }
    fechas.sort(LocalDate::compareTo);
    for (LocalDate fecha : fechas) {
        Funcion funcion = espectaculo.funcionEnFecha(fecha);
        String fechaStr = fecha.format(dateFormatter);
        sb.append(funcion.sedeObj().descripcionParaListado(funcion, fechaStr));
        sb.append(str:"\n");
    }
    return sb.toString();
}
```

En la clase **Ticketek**, utilizamos el concepto de **Foreach** (estructura de control que permite recorrer todos los elementos de una colección de manera sencilla y directa, sin necesidad de usar índices o variables de control) para procesar las funciones de un espectáculo.

En este método:

- Primero, se obtiene el espectáculo correspondiente al nombre recibido como parámetro.
- Luego, se utiliza un foreach para recorrer todas las funciones del espectáculo **(for (Funcion f : espectaculo.todasLasFunciones()))**, y se agregan sus fechas a una lista.
- Después de ordenar las fechas, se utiliza otro foreach para recorrer cada fecha **(for (LocalDate fecha : fechas))**, obteniendo la función correspondiente y su descripción formateada.
- Cada descripción se agrega al StringBuilder, junto con un salto de línea.

Gracias al uso de foreach, el código es más claro y legible, ya que permite recorrer todas las funciones y fechas sin preocuparse por índices ni detalles internos de la colección. Esto facilita la construcción de la lista de funciones, reduce la posibilidad de errores y hace que el código sea más fácil de mantener y entender.

Herencia y Polimorfismo

```java
public class MiniEstadio extends Sede {
    private final int cantidadPuestos;
    private final double precioConsumicion;
    private final List<Sector> sectores;
    public MiniEstadio(String nombre, String direccion, int capacidadMaxima, int cantidadPuestos,
    double precioConsumicion, List<Sector> sectores) {
        super(nombre, direccion, capacidadMaxima);
        this.cantidadPuestos = cantidadPuestos;
        this.precioConsumicion = precioConsumicion;
        this.sectores = sectores;
    }
    public int getCantidadPuestos() { return cantidadPuestos; }
    public Sector getSectorPorNombre(String nombre) {
        for (Sector s : sectores) {
            if (s.getNombre().equalsIgnoreCase(nombre)) return s;
        }
        return null;
    }
    public int porcentajeAdicional(String nombre) {
        Sector s = getSectorPorNombre(nombre);
        return s != null ? s.getPorcentajeAdicional() : 0;
    }
    public double precioConsumicion() { return precioConsumicion; }
    @Override
    public String descripcionParaListado(Funcion funcion, String fechaStr) {
        StringBuilder sb = new StringBuilder();
        sb.append(str:" - (").append(fechaStr).append(str:") ").append(nombreSede()).append(str:" - ");
        for (int i = 0; i < sectores.size(); i++) {
            sb.append(sectores.get(i).descripcionConVendidas(funcion));
            if (i < sectores.size() - 1) sb.append(str:" | ");
        }
        return sb.toString();
    }
```

```java
@Override
public double calcularPrecioEntrada(String nombreSector, double precioBase) {
    Sector s = getSectorPorNombre(nombreSector);
    if (s != null) {
        return s.calcularPrecio(precioBase, precioConsumicion);
    }
    return precioBase + precioConsumicion;
}
public static void validarDatos(String nombreSede, String direccion, int capacidadMaxima, int asientosPorFila,
int cantidadPuestos, double precioConsumicion, String[] sectores, int[] capacidad, int[] porcentajeAdicional) {
    Teatro.validarDatos(nombreSede, direccion, capacidadMaxima, asientosPorFila, sectores, capacidad,
    porcentajeAdicional);
    if (cantidadPuestos <= 0) throw new RuntimeException(message:"La cantidad de puestos debe ser mayor a cero");
    if (precioConsumicion <= 0) throw new RuntimeException(message:"El precio de consumición debe ser mayor a cero"
}
@Override
public Map<String, Integer> capacidadPorSector() {
    Map<String, Integer> map = new HashMap<>();
    for (Sector s : sectores) {
        map.put(s.getNombre(), s.getCapacidad());
    }
    return map;
}
```

Mediante el concepto de **herencia**, la clase **MiniEstadio** extiende a la clase abstracta Sede. Esto significa que **MiniEstadio** hereda todos los atributos y métodos definidos en Sede,

como nombre, direccion, capacidadMaxima y la gestión de funciones, permitiendo reutilizar código común a todas las sedes.

Además, **MiniEstadio** puede agregar atributos y métodos propios que son específicos de este tipo de sede, como cantidadPuestos, precioConsumicion y la lista de sectores. También puede sobrescribir métodos definidos en la clase base para adaptar su comportamiento, por ejemplo, redefiniendo descripcionParaListado, calcularPrecioEntrada y capacidadPorSector para que funcionen de acuerdo a las reglas particulares de un miniestadio.

Gracias a la herencia, un objeto de tipo **MiniEstadio** puede ser tratado como un objeto de tipo Sede, lo que permite que métodos que reciben o manipulan instancias de Sede puedan trabajar también con instancias de **MiniEstadio**, **Teatro**, **Estadio**, etc. Por ejemplo:

Sede sede = new MiniEstadio(...);

Mediante el **polimorfismo**, aprovechamos que las subclases de Sede (como **Estadio**, **Teatro** y **MiniEstadio**) pueden ser tratadas como objetos del tipo base Sede, pero cada una puede definir su propio comportamiento para ciertos métodos. Esto se logra a través de la redefinición (override) de métodos abstractos o concretos en la clase base.

Por ejemplo, los métodos **descripcionParaListado** y **calcularPrecioEntrada** están definidos en la clase **Sede** (como abstractos o con una implementación base), pero cada subclase los implementa de manera diferente según sus necesidades:

- En **Estadio**, **descripcionParaListado** muestra la cantidad total de entradas vendidas y la capacidad máxima, mientras que en **Teatro** y **MiniEstadio** se detallan los sectores, asientos y otros datos relevantes para cada tipo de sede.
- El método **calcularPrecioEntrada** también varía: en **Estadio** simplemente devuelve el precio base, mientras que en **Teatro** y **MiniEstadio** se calcula el precio según el sector, porcentaje adicional, o incluso consumiciones extra.

Gracias al polimorfismo, el sistema puede invocar estos métodos sobre una referencia de tipo **Sede** sin preocuparse por el tipo concreto de la sede. Por ejemplo, al listar funciones o calcular precios, el método adecuado se ejecuta automáticamente según el tipo real del objeto:

Sede sede = new MiniEstadio(...);

String descripcion = sede.descripcionParaListado(funcion, fechaStr); // Llama a la versión de MiniEstadio

double precio = sede.calcularPrecioEntrada("Platea", precioBase); // Llama a la versión de MiniEstadio

Esto permite que el código sea más flexible, extensible y fácil de mantener, ya que se pueden agregar nuevas subclases de Sede con comportamientos específicos sin modificar el código que utiliza el tipo base.

Sobreescritura, sobrecarga e interfaz

En la clase **Ticketek** se utiliza el concepto de sobrecarga de métodos, que ocurre cuando existen varios métodos con el mismo nombre pero distinta lista de parámetros (tipo, cantidad u orden). Esto permite ofrecer distintas formas de realizar una acción similar, adaptándose a diferentes necesidades del sistema.

Por ejemplo, para la venta de entradas, existen dos métodos venderEntrada sobrecargados:

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

public List<IEntrada> venderEntrada(String nombreEspectaculo, String fecha, String email, String contrasenia, String sector, int[] asientos)

Ambos métodos permiten vender entradas, pero uno está pensado para estadios (entradas generales) y el otro para teatros o miniestadios (entradas numeradas por sector y asiento).

La sobreescritura ocurre cuando una clase que implementa una interfaz o hereda de una superclase redefine un método, manteniendo la misma firma pero cambiando su implementación.

En **Ticketek**, esto se ve reflejado en la implementación de los métodos definidos en la interfaz **ITicketek**. Cada método sobrescrito está marcado con la anotación @Override, indicando que se está proporcionando una implementación concreta para el contrato definido por la interfaz.

Por ejemplo:

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

```
@Override // Estadio
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:

```
// Implementación principal del sistema Ticketek
public class Ticketek implements 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

```
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.tieneEntrada(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.funcionEnFecha(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.tieneEntrada (ent.codigo()))**, el método **tieneEntrada** internamente utiliza **.containsKey(codigo)** sobre el HashMap de entradas del usuario, lo cual 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.funcionEnFecha(ent.fecha());** internamente, esto accede a un **HashMap<LocalDate, Funcion>** mediante .get(fecha), que 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.

Universidad Nacional de General Sarmiento

Programación II – Comisión 3

Trabajo Práctico Integrador – Primera parte

Docentes:

- Nores, José
- Gabrielli, Miguel Ángel

Integrantes:

- Cordua, Kevin Nicolas Daniel
- Chazarreta, Jeremías Vladimir

2025C1

Introducción

En el siguiente informe se plasmará la resolución de un problema concerniente al sistema de entradas de Ticketek, el sistema necesita el desarrollo de un sistema para gestionar la venta de entradas para los espectáculos públicos que organiza. Por lo que lo haremos mediante el desarrollo de TADs, la interfaz y el modelado del diagrama de herencias correspondiente.

TADs

Posibles TADs:

- Ticketek
- Usuario
- Entrada
- Sede
- Funcion
- Espectáculo
- SedeConPlatea
- Teatro
- Miniestadio
- Estadio

TAD Ticketek:

Datos

- tipo
- usuarios
- sedes

Operaciones

- Ticketek (precio_base, tipo) // constructor
- registrar_sede (capacidad, direccion, nombre) // registra las sedes
- registrar_usuario (nombre, mail, contrasenia) // registra usuarios nuevos
- registrar_espect (nombre) // registra los espectáculos
- vender_entrada (codigo, nombre_espect, fecha, ubicacion) // permite vender entradas a los usuarios
- lista_sedes (direccion) // despliega una lista con las sedes de Ticketek
- lista_entradas_futuras (mail, contrasenia) // despliega una lista con las entradas de un cliente para fechas que no llegaron aun
- lista_todas_entradas (mail, contrasenia) // despliega una lista con todas las entradas del usuario
- anular_entrada (codigo, contrasenia) // permite al usuario anular una entrada
- cambiar_entrada (nombre_espect, codigo, direccion) // permite al usuario cambiar su entrada por otra
- calcular_costo_entrada (mail, contrasenia, codigo): int // calcula el costo de una entrada
- consultar_valor_entrada (direccion, ubicacion): int // permite al usuario ver el costo de una entrada
- calcular_total_recaudado (codigo): int // calcula cuánto recaudó un espectáculo en total

TAD Usuario:

Datos

- nombre_apellido
- mail
- contrasenia
- entradas
- nombre_espect
- fecha
- ubicacion

Operaciones

- usuario (nombre_apellido, mail, contrasenia, nombre_espect, fecha, ubicacion)
- comprar_entradas (nombre_espect, fecha, ubicacion) // permite al usuario comprar entradas
- entradas_pasadas (mail, contrasenia) // muestra las entradas de fechas anteriores
- entradas_futuras (mail, contrasenia) // muestra las entradas de fechas a futuro

TAD Entrada:

Datos

- codigo
- ubicaciones
- funcion
- anulada
- nombre_espect

Operaciones

- entrada (codigo, ubicaciones, funcion, anulada, nombre_espect)

TAD Sede:

Datos

- capacidad
- direccion
- nombre_sede
- precio_base

Operaciones

- sede (capacidad, direccion, nombre_sede, precio_base)

TAD Espectáculo:

Datos

- cantidad
- nombre_espect
- codigo
- funciones

Operaciones

- espectaculo (cantidad, nombre, codigo)
- registrar_funcion (cantidad, nombre_espect, código) // registra las funciones que tiene un espectáculo

TAD Función:

Datos

- fecha
- precio_base

Operaciones

- funcion (fecha, sede, precio_base)

TAD Estadio:

Datos

- capacidad
- direccion
- nombre_sede
- precio_unico

Operaciones

- estadio (capacidad, direccion, nombre_sede, precio_unico): int
- obtener_precio_unico (precio_unico): int // calcula el precio de la entrada para un estadio teniendo en cuenta sus características

TAD SedeConPlatea:

Datos

- capacidad

- direccion
- nombre_sede
- precio_base
- plateas
- cant_asientos_por_fila
- nro_asiento

Operaciones

- ∉ sede_con_platea (capacidad, direccion, nombre_sede, cant_asientos_por_fila, nro_asiento) //constructor
- ∉ obtener_fila (cant_asientos_por_fila, nro_asiento): int // obtiene la fila
- ∉ obtener_precio_platea (obtener_fila()): int // obtiene el precio teniendo en cuenta la platea del tipo de sede

TAD Miniestadio:

Datos

- capacidad
- direccion
- nombre_sede
- precio_base
- plateas
- precio_aniadido
- cantidad_puestos
- cant_asientos_por_fila
- nro_asiento

Operaciones

- miniestadio (capacidad, direccion, nombre_sede, precio_base, precio_aniadido, cantidad_puestos, cant_asientos_por_fila, nro_asiento)
- obtener_precio_platea (obtener_fila(), precio_aniadido): int // obtiene el precio dependiendo de la platea
- precio_entrada_unidad_mini (precio_aniadido, cantidad_puestos): int // calcula el precio de la entrada para un miniestadio teniendo en cuenta sus características

TAD Teatro:

Datos

- capacidad
- direccion
- nombre_sede

- precio_base
- plateas
- cant_asientos_por_fila
- nro_asiento

Operaciones

- teatro (capacidad, direccion, nombre_sede, cant_asientos_por_fila, nro_asiento)
- precio_entrada_unidad_teatro (precio_base): int // calcula el precio de la entrada para un teatro teniendo en cuenta sus características

Interfaz

(cabe aclarar que estaría bueno utilizar @throws en todos los métodos para para indicar qué excepciones puede lanzar)

Interfaz para Ticketek

```
// Constructor

// Texto dando la bienvenida a la página, se despliega el menú con
// las opciones que puede ejecutar el usuario. Se genera un error en
// caso de que la página no cargue correctamente

Ticketek(int precio_base, Tipo [] tipo);

// Registra una nueva sede en el sistema, se genera un error si la
// sede ya existe o los datos son inválidos

public void registrar_sede (int capacidad, String direccion, String
nombre);

// Registra un usuario nuevo, en caso de que ya exista un usuario
// asociado con el correo electrónico ya registrado, se genera un
// error, ya que debe ser único. Genera error si el usuario ya existe
// o los datos son inválidos

public void registrar_usuario (String nombre, String mail, String
contrasenia);

// Registra un nuevo espectáculo, se genera un error en caso de que
// dos espectáculos distintos tomen lugar en el mismo día y en la
// misma sede, o en el caso de que más de un espectáculo contenga el
// mismo valor de código, o los datos son inválidos

public void registrar_espect (String nombre);
```

```
// Vende entradas al usuario para un espectáculo, teniendo en cuenta
//  el  espectáculo,  la  sede,  números  de  asientos  y  también  su
//  contraseña. Si la sede es del tipo mini estadio o teatro se tiene
// en cuenta el sector. Genera error si el espectáculo no existe, la
// fecha es inválida o no hay disponibilidad

public void vender_entrada (int codigo, String nombre_espect, Fecha
fecha, String ubicacion);

// Despliega una lista con todas las sedes disponibles. Genera error
// si no hay sedes registradas

public Map <String, sede> lista_sedes (String direccion);

// Despliega una lista con las entradas compradas por el usuario,
// para funciones futuras de fechas que aún no han llegado. Genera
// error si el usuario no existe o la contraseña es incorrecta

public  Map  <String,  Entrada>  lista_entradas_futuras  (String  mail,
String contrasenia);

//  Despliega  una  lista  con  todas  las  entradas  compradas  por  el
// usuario, tanto las de fechas pasadas como las próximas.  Genera
// error si el usuario no existe o la contraseña es incorrecta

public Map <String, Entrada> lista_todas_entradas (String mail, String
contrasenia);

// Permite al usuario anular una entrada ya existente, indicando el
// código de la entrada y la contraseña del usuario. Se genera un
// error si la entrada no existe, ya está anulada o la contraseña es
// incorrecta

public void anular_entrada (int codigo, String contrasenia);

// Permite al usuario cambiar una entrada que ya compró por otra,
// para un espectáculo para otra sede, conociendo el código de la
// entrada. Se genera un error si la entrada no existe, ya está
// anulada o la sede destino no existe

public void cambiar_entrada (String nombre_espect, int codigo, String
direccion);

// Calcula el costo de una entrada en específico autenticando al
// usuario que la compró. Genera error si el usuario no existe, la
// contraseña es incorrecta o la entrada no existe

public int calcular_costo_entrada (String mail, String contrasenia,
int codigo);
```

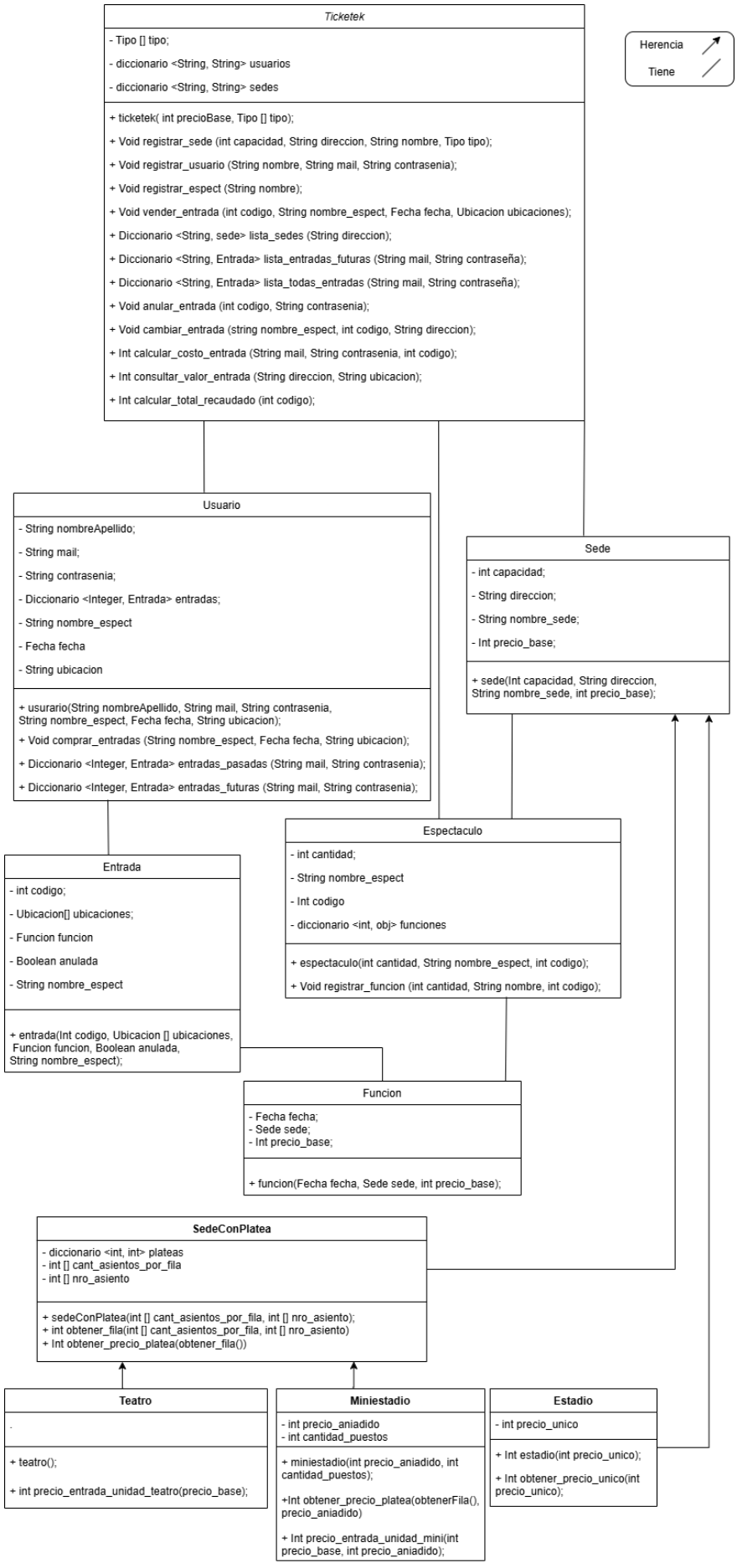
```java
// Permite al usuario consultar el valor base de una entrada a un
// espectáculo indicando la sede y el sector. En caso de ser en
// estadio, ese valor será ignorado. Genera error si la sede no
// existe o la ubicación es inválida

public int consultar_valor_entrada (String direccion, String
ubicacion);

// Calcula el total recaudado por un espectáculo.  Genera error si
// el código del espectáculo no existe

public int calcular_total_recaudado (int codigo);
```

Diseño


Ticketek
- Tipo [] tipo;
- diccionario <String, String> usuarios
- diccionario <String, String> sedes
+ ticketek(int precioBase, Tipo [] tipo);
+ Void registrar_sede (int capacidad, String direccion, String nombre, Tipo tipo);
+ Void registrar_usuario (String nombre, String mail, String contrasenia);
+ Void registrar_espect (String nombre);
+ Void vender_entrada (int codigo, String nombre_espect, Fecha fecha, Ubicacion ubicaciones);
+ Diccionario <String, sede> lista_sedes (String direccion);
+ Diccionario <String, Entrada> lista_entradas_futuras (String mail, String contraseña);
+ Diccionario <String, Entrada> lista_todas_entradas (String mail, String contraseña);
+ Void anular_entrada (int codigo, String contrasenia);
+ Void cambiar_entrada (string nombre_espect, int codigo, String direccion);
+ Int calcular_costo_entrada (String mail, String contrasenia, int codigo);
+ Int consultar_valor_entrada (String direccion, String ubicacion);
+ Int calcular_total_recaudado (int codigo);
Herencia
Tiene
Usuario
- String nombreApellido;
- String mail;
- String contrasenia;
- Diccionario <Integer, Entrada> entradas;
- String nombre_espect
- Fecha fecha
- String ubicacion
+ usurario(String nombreApellido, String mail, String contrasenia, String nombre_espect, Fecha fecha, String ubicacion);
+ Void comprar_entradas (String nombre_espect, Fecha fecha, String ubicacion);
+ Diccionario <Integer, Entrada> entradas_pasadas (String mail, String contrasenia);
+ Diccionario <Integer, Entrada> entradas_futuras (String mail, String contrasenia);
Sede
- int capacidad;
- String direccion;
- String nombre_sede;
- Int precio_base;
+ sede(Int capacidad, String direccion, String nombre_sede, int precio_base);
Entrada
- int codigo;
- Ubicacion[] ubicaciones;
- Funcion funcion
- Boolean anulada
- String nombre_espect
+ entrada(Int codigo, Ubicacion [] ubicaciones, Funcion funcion, Boolean anulada, String nombre_espect);
Espectaculo
- int cantidad;
- String nombre_espect
- Int codigo
- diccionario <int, obj> funciones
+ espectaculo(int cantidad, String nombre_espect, int codigo);
+ Void registrar_funcion (int cantidad, String nombre, int codigo);
Funcion
- Fecha fecha;
- Sede sede;
- Int precio_base;
+ funcion(Fecha fecha, Sede sede, int precio_base);
SedeConPlatea
- diccionario <int, int> plateas
- int [] cant_asientos_por_fila
- int [] nro_asiento
+ sedeConPlatea(int [] cant_asientos_por_fila, int [] nro_asiento);
+ int obtener_fila(int [] cant_asientos_por_fila, int [] nro_asiento)
+ Int obtener_precio_platea(obtener_fila())
Teatro
.
+ teatro();
+ int precio_entrada_unidad_teatro(precio_base);
Miniestadio
- int precio_aniadido
- int cantidad_puestos
+ miniestadio(int precio_aniadido, int cantidad_puestos);
+Int obtener_precio_platea(obtenerFila(), precio_aniadido)
+ Int precio_entrada_unidad_mini(int precio_base, int precio_aniadido);
Estadio
- int precio_unico
+ Int estadio(int precio_unico);
+ Int obtener_precio_unico(int precio_unico);