**Práctica Laboral e Investigativa I**

**Universidad Central Marta Abreu de Las Villas**

**Facultad de Matemática, Física y Computación**

**«Central Telefónica»**

Amalia Rodríguez Sánchez

Rocio de la Caridad Hernández Caballero

**Resumen**

En el transcurso de esta práctica laboral, se abordó el desafío de desarrollar un sistema integral para automatizar las operaciones de una central telefónica, facilitando la gestión de llamadas nacionales e internacionales. Para resolver esta problemática, se diseñó una solución que incluía clases bien estructuradas y una interfaz de usuario intuitiva, alineada con los requisitos establecidos. Este enfoque no solo optimizó el procesamiento de datos relacionados con abonados y tarifas, también permitió calcular costos y generar informes detallados sobre la actividad telefónica. Las pruebas realizadas confirmaron la eficacia del sistema, que se estableció como herramienta valiosa para la central telefónica.

**Introducción**

En un mundo donde la comunicación es un pilar esencial de la interacción humana y empresarial, las centrales telefónicas se encuentran ante el imperativo de modernizar sus operaciones. La creciente demanda de servicios de llamadas nacionales e internacionales plantea un desafío significativo: la necesidad de gestionar eficientemente un volumen cada vez mayor de tráfico telefónico. Este contexto exige la implementación de sistemas que no solo faciliten la conexión entre abonados, sino que también optimicen la administración de tarifas y permitan la generación de informes detallados sobre la actividad telefónica.

La problemática central radica en la automatización de procesos que, hasta ahora, han sido realizados manualmente. Esta dependencia de métodos tradicionales puede dar lugar a errores humanos y retrasos en la gestión, lo que afecta la calidad del servicio ofrecido a los usuarios. Además, es crucial contar con un sistema que proporcione un seguimiento preciso de las llamadas y un control efectivo sobre los costos asociados a cada tipo de comunicación. En este sentido, se hace evidente la necesidad de diseñar una solución que contemple tanto la gestión integral de datos de los abonados como el cálculo dinámico de tarifas basadas en el tipo y duración de las llamadas.

Para abordar esta complejidad, se requiere un diseño robusto y flexible que permita procesar información en tiempo real. Este enfoque debe garantizar no solo una experiencia fluida para los usuarios, sino también una administración eficiente por parte del personal de la central telefónica. El presente informe detalla el enfoque adoptado para desarrollar un sistema integral que responda a estas necesidades, así como los resultados obtenidos durante las pruebas realizadas, destacando su eficacia y potencial para transformar las operaciones de la central telefónica en cuestión.

**Desarrollo**

***Excepciones***

En el desarrollo de nuestro programa, implementamos un sistema de manejo de errores para gestionar situaciones inesperadas de manera efectiva. Este enfoque nos permite anticipar y controlar problemas que podrían surgir durante la ejecución, garantizando que el programa se mantenga estable. Al ofrecer mensajes claros y comprensibles al usuario en caso de fallos, evitamos confusiones y mejoramos la experiencia general. A continuación, se presentarán las clases que responden a las excepciones, proporcionando una visión detallada de su implementación y propósito en el sistema.

**PhoneNumberAlreadyExistsE:** es lanzada por el método **addPhoneNumber()** en la clase **Client**. Su función es evitar que se agregue un número de teléfono que ya existe en el listado de números del cliente.

**PhoneNumberDoesNotExistE**: es lanzada por el método **removePhoneNumber()** en la clase **Client**. Su función es prevenir la eliminación de un número de teléfono que no se encuentra en el listado de números del cliente.

**ClientAlreadyExistsE:** es lanzada por el método **addClient()** en la clase **Central**. Su función es evitar que se intente añadir un cliente que ya está presente en el listado de clientes de la central telefónica.

**TaxAlreadyExistsE**: es lanzada por el método **addTax()** en la clase **Central**. Su función es evitar que se agregue una tarifa que ya existe en el listado de tarifas de la central telefónica.

**InvalidMonthE**: es lanzada por el método **getMonthEarning()** en la clase **Central**. Su función es verificar que el mes proporcionado como parámetro, del que se desea calcular su ganancia, sea un entero válido, es decir, que esté entre 1 y 12.

**InvalidOperationE**: es lanzada por el método **getMonthEarning()** en la clase **Central**. Su función es verificar que la operación proporcionada como parámetro sea un entero válido, es decir, que esté entre 1 y 3. Este rango responde al tipo de llamada del que se desea calcular la ganancia mensual: 1 para llamadas nacionales, 2 para internacionales y 3 para el total de llamadas del mes.

**WrongPeriodFormatE**: es lanzada por los métodos **getHotCountries()**, **getHotProvince()** y **getClientOwe()** en la clase Central. Su función es verificar que el período proporcionado como parámetro tenga el formato correcto de (mes-mes). Para realizar esta validación, se apoya en el método auxiliar **invalidPeriod()**, que retorna verdadero si el período no es válido.

**InvalidCredentials**: es lanzada por el método **verifyClient** en la clase **Central**. Esta excepción fue creada con el propósito de utilizarse en la clase de la interfaz de usuario **ClientLoginPanel** para verificar que el cliente esté registrado en la central. Si el cliente no está registrado, se lanza la excepción y se solicita que cree una cuenta. Al gestionar esta situación, garantizamos que el usuario reciba un mensaje claro y pertinente, facilitando así un proceso de inicio de sesión más intuitivo y eficiente.

***Clases de la Interfaz Gráfica de Usuario***

A continuación, se explicarán las clases de la interfaz gráfica de usuario que son fundamentales, ya que son el punto de interacción directa entre el usuario y el sistema. Estas clases están diseñadas para ofrecer una experiencia intuitiva y accesible, permitiendo a los usuarios realizar diversas operaciones de manera eficiente. La correcta implementación de estas clases no solo mejora la usabilidad del programa, sino que también facilita la comprensión y gestión de las funcionalidades disponibles.

**ClientLoginPanel:** da la bienvenida al usuario en la central telefónica y le brinda la opción de registrarse con su cuenta para acceder a sus datos. Si el usuario no está registrado, se lanza la excepción **InvalidCredentials**, lo que da paso Cuenta con un apartado que dirige al **ClientRegisterPanel** para que el usuario pueda crear una cuenta en la central telefónica. Además, si el usuario desea acceder a todos los datos de la central como administrador, este panel facilita la transición al **AdminPanel**. De esta manera, **ClientLoginPanel** actúa como un punto de entrada versátil, adaptándose a las necesidades del usuario y garantizando un acceso adecuado a las funcionalidades del sistema.

**ClientRegisterPanel:** está diseñada para aquellos usuarios que no están registrados en la central telefónica y desean crear una cuenta. Este panel solicita todos los datos necesarios, incluyendo el código del usuario y su tipo de cliente, y verifica que esta información sea correcta. Una vez validados los datos, se agrega al cliente a la central, permitiéndole disfrutar de todas las opciones y funcionalidades disponibles como cliente. De esta manera, **ClientRegisterPanel** facilita el proceso de registro, asegurando que los nuevos usuarios puedan acceder rápidamente a los servicios ofrecidos por la central telefónica.

**AdminLoginPanel**: se encarga de gestionar el proceso de inicio de sesión del administrador de la central telefónica. Ofrece al administrador la posibilidad de ingresar su código, el cual está establecido como "admin". En caso de que se introduzca un código diferente, el acceso será denegado. Este panel es fundamental para asegurar que solo personas autorizadas puedan acceder a toda la información sensible de la central telefónica, protegiendo así la integridad y seguridad de los datos.

**ClientPanel:** muestra al cliente, una vez registrado, sus números de teléfono y su historial de llamadas recientes. Además, ofrece las opciones de agregar o eliminar un número de teléfono, manejando correctamente las excepciones **PhoneNumberAlreadyExistsE** y **PhoneNumberDoesNotExistE**, asegurando así que las operaciones se realicen sin errores.

Una de las funcionalidades más importantes que proporciona este panel es la capacidad de realizar una llamada. Para ello, se presenta una ventana que solicita toda la información necesaria para la llamada, según el paquete DAYOS, y se verifica que todos los datos ingresados sean correctos antes de proceder. Una vez validados, la llamada se guarda en el listado de llamadas de la central telefónica, garantizando un registro preciso y eficiente de las interacciones del cliente.

**AdminPanel**: se erige como la más crucial de todas, ya que proporciona una visión completa de toda la información relacionada con la central telefónica. Este panel otorga al administrador la capacidad de analizar todos los clientes, sus números de teléfono y el historial de llamadas, incluyendo detalles relevantes sobre cada interacción.

Además, cuenta con un panel titulado "Ganancias en un mes", que se conecta con el método **getMonthEarning()** de la clase Central. En este espacio, el administrador puede ingresar el mes del año que desea examinar y el tipo de llamadas para obtener la ganancia mensual correspondiente. También incluye otro panel que permite al administrador introducir un número de teléfono deseado y utilizar el método **getClientOwe()** de la clase Central para conocer lo que debe pagar ese cliente.

Por otro lado, dispone de un botón titulado "Ver estadísticas", que dirige al **StatsPanel**, donde se pueden explorar otras funcionalidades importantes de la central telefónica. De esta manera, el **AdminPanel** actúa como un centro integral para la gestión y análisis de datos dentro de la central, facilitando a los administradores el acceso a información crítica para la toma de decisiones.

***Clases principales***

Finalmente, ahondaremos en las clases principales de nuestro programa, imprescindibles para su funcionamiento ya que son las encargadas de gestionar y manejar todos los datos recopilados. Estas clases no solo organizan la información, sino que también aseguran que el programa opere correctamente, respondiendo a los requerimientos de la central que lo necesite.

**Call:** se encarga de representar las llamadas dentro del sistema y cuenta con dos constructores. El primero permite inicializar las llamadas utilizando el código *DAYOS*, que se empleará para su almacenamiento. El segundo constructor está diseñado para facilitar la integración con la interfaz de usuario, desde la cual se obtendrán los datos necesarios. Esta clase es esencial pues sus métodos permiten la obtención de los datos específicos de las llamadas.

**Client:** se utiliza para gestionar la información de los clientes dentro del sistema. Su propósito principal es permitir la manipulación y el almacenamiento de datos esenciales del cliente (nombre de usuario, contraseña y el tipo de cliente), así como el manejo de la lista de números de teléfono asociados a cada uno. Además, garantiza la validez de estos datos.

**Tax:** maneja la información relacionada con las tarifas. Su funcionamiento se centra en almacenar, verificar y proporcionar cuando sean requeridos los datos de las tarifas que deben aplicarse según la localidad.

**PhoneNumber:** representa los números telefónicos. Su función es la de devolver datos sobre cada número, así como la validación de estos.

**Central:** clase principal del programa. En ella se encuentran todas las funciones que responden directamente a los requisitos planteados en la orientación de este trabajo.

*Parámetros*

private String name: nombre de la central (4 números, representando los 2 primeros la provincia y los últimos 2 la localidad).

private String address: dirección de la central.

private ArrayList<Client> clients: lista de clientes que alberga la central.

private ArrayList<Tax> taxes: lista de las tarifas según la localidad.

private ArrayList<Call> calls\_history: lista de las llamadas que se han realizado.

*Funciones principales*

public void addClient(Client client) throws ClientAlreadyExistsE: añade un cliente (parámetro *client*) a la lista de clientes de la empresa. Luego de ello, se ordena la lista en orden ascendente utilizando la función *sort* con un comparador personalizado, lo cual fue necesario pues esta no posee integrado un criterio para ordenar objetos tipo **Client**. En caso de que el cliente exista, se eleva el error **ClientAlreadyExistsE**.

public void addClient(Client client) throws ClientAlreadyExistsE {

if (!clients.contains(client)) {

clients.add(client);

clients.sort(new Comparator<Client>() {

@Override

public int compare(Client o1, Client o2) {

return o1.getUsername().compareTo(o2.getUsername());

}

});

} else {

throw new ClientAlreadyExistsE();

}

}

public float getMonthEarning(int operation, int month) throws InvalidMonthE: devuelve cuánto recaudó la central en un mes (parámetro *month*) de acuerdo a un criterio (parámetro *operation*). En dependencia de si *operation* es igual a 1, 2 o 3 se devolverán las ganancias generadas por llamadas nacionales, internacionales o en total, respectivamente. Como *month* es ingresado por el usuario manualmente, se comprueba si este es válido. En caso de no serlo, se eleva el error  **InvalidMonthE.** Finalmente, el valor de la ganancia obtenida se redondea a 2 decimales utilizando el método *setScale* de la clase **BigDecimal.**

public float getMonthEarning(int operation, int month) throws InvalidMonthE {

if (month < 1 || month > 12) throw new InvalidMonthE();

float earning = 0;

for (Call call : calls\_history) {

String receiver\_country\_code = call.getReceiverCountryCode();

String sender\_country\_code = call.getSenderCountryCode();

int call\_month = call.getMonth();

if (call\_month == month) {

switch (operation) {

case 1:

if (receiver\_country\_code.equals(sender\_country\_code)) {

earning += getCallCost(call);

}

break;

case 2:

if (!receiver\_country\_code.equals(sender\_country\_code)) {

earning += getCallCost(call);

}

break;

case 3:

earning += getCallCost(call);

break;

}

}

}

BigDecimal rounded\_earning = new BigDecimal(earning);

rounded\_earning = rounded\_earning.setScale(2, RoundingMode.HALF\_UP);

return rounded\_earning.floatValue();

}

public float getClientOwe(String phone) throws ClientDoesNotExistsE: devuelve cuanto debe pagar un cliente determinado dado su número telefónico (parámetro *phone*). En caso de que el cliente no se encuentre en la lista de clientes de la central, se eleva el error **ClientDoesNotExistsE.** Finalmente, el valor obtenido se redondea a 2 decimales utilizando el método *setScale* de la clase **BigDecimal.**

public float getClientOwe(String phone) throws ClientDoesNotExistsE {

float owe = 0;

boolean client\_not\_found = true;

for (Call call : calls\_history) {

if (call.getSenderPhone().equals(phone)) {

owe += getCallCost(call);

client\_not\_found = false;

}

}

if (client\_not\_found) {

throw new ClientDoesNotExistsE();

}

BigDecimal rounded\_owe = new BigDecimal(owe);

rounded\_owe = rounded\_owe.setScale(2, RoundingMode.HALF\_UP);

return rounded\_owe.floatValue();

}

public ArrayList<Call> getOverPays(float max\_ax): devuelve una lista con las llamadas cuyo costo sobrepase una tarifa establecida como máxima (parámetro *max\_ax*).

public ArrayList<Call> getOverPays(float max\_ax) {

ArrayList<Call> overpays = new ArrayList<>();

for(Call call: calls\_history) {

if (getCallCost(call) > max\_ax) {

overpays.add(call);

}

}

return overpays;

}

public ArrayList<Map.Entry<String, Integer>> getHotCountries(String period, int n) throws WrongPeriodFormatE: devuelve una lista de países a los cuales se han realizado más de cierta cantidad de llamadas (parámetro *n*) en un período determinado (parámetro *period)*. Se decidió que esta lista fuera de **Map.Entry<String,Integer>**, la cual representa el país y la cantidad de veces que fue llamado, para la posterior utilización de estos resultados en la realización del gráfico de países más llamados que se muestra en el **StatsPanel**. Como *period* es ingresado por el usuario manualmente, se comprueba si este es válido. En caso de no serlo, se eleva el error  **WrongPeriodFormatE.**

public ArrayList<Map.Entry<String, Integer>> getHotCountries(String period, int n) throws WrongPeriodFormatE {

int counter = 0;

if (invalidPeriod(period)) throw new WrongPeriodFormatE();

String[] s = period.split("-");

int start = Integer.parseInt(s[0]);

int end = Integer.parseInt(s[1]);

Map<String, Integer> map = new HashMap<>();

for(Call call: calls\_history){

if (call.getMonth() >= start && call.getMonth() <= end){

if(map.containsKey(call.getReceiverCountryCode())) {

map.put(call.getReceiverCountryCode(), map.get(call.getReceiverCountryCode()) + 1);

} else {

map.put(call.getReceiverCountryCode(), 1);

}

}

}

ArrayList<Map.Entry<String, Integer>> hotCountries = new ArrayList<>();

for(Map.Entry<String, Integer> entry: map.entrySet()) {

if(entry.getValue() >= n) {

hotCountries.add(entry);

}

}

return hotCountries;

}

public ArrayList<Map.Entry<String, Integer>> getProvinces(String period) throws WrongPeriodFormatE: devuelve una lista de provincias a las cuales se han realizado llamadas en un período determinado (parámetro *period)*. Se decidió que esta lista fuera de **Map.Entry<String,Integer>**, la cual representa la provincia y la cantidad de veces que fue llamada, para la posterior utilización de estos resultados en la realización del gráfico de provincias más llamadas que se muestra en el **StatsPanel**. Como *period* es ingresado por el usuario manualmente, se comprueba si este es válido. En caso de no serlo, se eleva el error  **WrongPeriodFormatE.**

public ArrayList<Map.Entry<String, Integer>> getProvinces(String period) throws WrongPeriodFormatE {

if (invalidPeriod(period)) throw new WrongPeriodFormatE();

String[] s = period.split("-");

int start = Integer.parseInt(s[0]);

int end = Integer.parseInt(s[1]);

Map<String, Integer> map = new HashMap<>();

for(Call call: calls\_history){

if (call.getMonth() >= start && call.getMonth() <= end){ if(map.containsKey(call.getReceiverLocationCode())) {

map.put(call.getReceiverLocationCode(), map.get(call.getReceiverLocationCode()) + 1);

} else {

map.put(call.getReceiverLocationCode(), 1);

}

}

}

return new ArrayList<>(map.entrySet());

}