



Universidad de Murcia

FACULTAD DE INFORMÁTICA

Documentación AppChat

Tecnologías de Desarrollo de Software 2024/2025

Autores:

Eva Pina Dubovtseva
Subgrupo 2.3
eva.pinad@um.es

Ismel Alejandro Oquendo
Rodríguez
Subgrupo 2.3
ia.oquendorodriguez@um.es

Docente:

Francisco Javier Bermudez Ruiz
fjavier@um.es

Índice

1	Introducción	3
2	Historias de Usuario	3
2.1	Autenticación de usuario	3
2.2	Registro de usuario	3
2.3	Gestión de contactos	4
2.4	Enviar mensajes	4
2.5	Recepción de mensajes	5
2.6	Creación de grupos de contactos	5
2.7	Conversión a usuario premium	6
2.8	Exportar mensajes en PDF	6
2.9	Aplicar descuentos en premium	7
2.10	Aplicación de múltiples filtros de búsqueda	7
3	Diagramas de clases UML	8
3.1	Primera entrega	8
3.2	Utilizado en el proyecto	8
4	Diagrama de secuencia	10
4.1	Añadir contacto a grupo	10
5	Arquitectura de la aplicación	11
5.1	Diseño en capas	11
5.2	Relaciones entre capas	11
5.3	Persistencia de datos	11
5.4	Gestión de dependencias	12
5.5	Características arquitectónicas clave	12
6	Patrones de diseño empleados	13
6.1	Patrones implementados directamente	13
6.1.1	Singleton	13
6.1.2	Factoría	13
6.1.3	Estrategia	15
6.1.4	Pool	16
6.2	Adaptador	17
6.3	Patrones empleados indirectamente	19
6.3.1	GRASP y separación modelo-vista	19
6.3.2	Patrones en Swing	19
6.3.3	Patrones en el Servicio de Persistencia (TDS)	19
7	Manual de uso	20
7.1	Acceso a la aplicación	20
7.2	Registro de usuario	20
7.3	Inicio de sesión	21
7.4	Envío de mensajes	22
7.4.1	Envío por número de teléfono	23
7.4.2	Envío por nombre de contacto	24
7.5	Crear grupo	26
7.6	Gestión del perfil	28

7.6.1	Cambiar imagen	28
7.6.2	Cambiar saludo	29
7.7	Agregar contacto	29
7.7.1	Opción 1	29
7.7.2	Opción 2	30
7.8	Premium y descuentos	31
7.8.1	Conversión a premium	32
7.8.2	Exportar chats a PDF	33
7.9	Búsqueda de mensajes	33
8	Tiempo dedicado	35
9	Observaciones finales	35

1 Introducción

AppChat es una aplicación de mensajería de escritorio desarrollada en Java, inspirada en aplicaciones populares como WhatsApp y Telegram.

El desarrollo de este proyecto se enfoca en aplicar buenas prácticas de ingeniería del software, como son el desacoplamiento de componentes, el uso de interfaces, la gestión de dependencias con Maven, y la implementación de una arquitectura modular que separa claramente las responsabilidades de cada capa del sistema.

A lo largo de este documento, se detallarán los aspectos de diseño, implementación y las decisiones técnicas tomadas durante el desarrollo de *AppChat*.

2 Historias de Usuario

2.1 Autenticación de usuario

- **Como** usuario registrado
- **quiero** poder iniciar sesión en la aplicación
- **para** poder acceder a mis mensajes.

Criterios de aceptación:

1. Autenticación exitosa

- Dado que el usuario ya está registrado, cuando introduce correctamente su número de teléfono y contraseña, entonces se le debe permitir el acceso a la aplicación.

2. Error en autenticación

- Dado que el usuario no introduce los datos correctamente, entonces debe aparecer un mensaje de error que le informe del fallo y la necesidad de reintentarlo.

3. Usuario no registrado

- Dado que el usuario no esté registrado, no se le podrá conceder acceso a la aplicación.

2.2 Registro de usuario

- **Como** nuevo usuario
- **quiero** poder registrarme en la aplicación
- **para** poder empezar a utilizar sus funcionalidades.

Criterios de aceptación:

1. Registro exitoso

- Dado que un usuario ingresa toda la información necesaria para registrarse, cuando presiona el botón “Registrar”, entonces debe ser registrado exitosamente en el sistema.

2. Campos incompletos

- Dado que el usuario no rellena uno de los campos obligatorios, cuando intenta registrarse, entonces debe aparecer un mensaje de error que falta rellenar ese campo.

3. Contraseñas no coinciden

- Dado que el usuario ingresa una contraseña no coincidente en los dos campos requeridos, cuando intenta registrarse, entonces debe aparecer un mensaje de error indicando que las contraseñas no coinciden.

2.3 Gestión de contactos

- **Como** usuario registrado
- **quiero** poder añadir nuevos contactos
- **para** poder comunicarme con ellos.

Criterios de aceptación:

1. Número no existente

- Dado que el usuario ingresa un número de teléfono de contacto que no existe en el sistema, cuando intenta añadir el contacto, entonces debe recibir un mensaje de error indicando que el número no existe.

2. Añadir contacto exitoso

- Dado que el usuario introduce correctamente el número de teléfono y nombre de un contacto existente, cuando presiona “Añadir contacto”, entonces debe añadirse el contacto a su lista de contactos y mostrarse en la lista.

2.4 Enviar mensajes

- **Como** usuario registrado
- **quiero** enviar mensajes a mis contactos
- **para** poder comunicarme con ellos.

Criterios de aceptación:

1. Enviar mensaje a contacto

- Dado que el usuario selecciona un contacto que existe en su lista de contactos, cuando escribe un mensaje y presiona “Enviar”, entonces el mensaje debe enviarse al contacto seleccionado.

2. Enviar mensaje a grupo

- Dado que el usuario selecciona varios contactos en un grupo, cuando presiona “Enviar”, entonces el mensaje debe enviarse individualmente a cada contacto del grupo.

3. Enviar mensaje a número no guardado

- Dado que el usuario intenta enviar un mensaje a un número que no está en su lista de contactos, se le debe indicar que ese número no se encuentra en su lista, y asimismo sugerir que lo añada como contacto.

2.5 Recepción de mensajes

- **Como** usuario registrado
- **quiero** recibir mensajes de otros usuarios
- **para** mantener la comunicación con mis contactos.

Criterios de aceptación:

1. Visualización de mensaje recibido

- Dado que otro usuario envía un mensaje, cuando llega al sistema, entonces debe mostrarse en la ventana de conversación correspondiente con la información del remitente y hora de recepción.

2. Actualización de conversaciones recientes

- Dado que se recibe un nuevo mensaje, cuando este llega al sistema, entonces la conversación debe aparecer como la primera en el panel de conversaciones recientes.

3. Mensaje de usuario no registrado como contacto

- Dado que se recibe un mensaje de un número no guardado en contactos, cuando este llega, entonces debe mostrarse el número y ofrecer la opción de añadirlo como contacto.

2.6 Creación de grupos de contactos

- **Como** usuario registrado
- **quiero** crear grupos de contactos
- **para** poder comunicarme con varios contactos simultáneamente.

Criterios de aceptación:

1. Creación exitosa de grupo

- Dado que el usuario proporciona un nombre para el grupo, cuando selecciona “Añadir grupo”, entonces el sistema debe crear el grupo y añadirlo a la lista de contactos.

2. Nombre de grupo ya existente

- Dado que el usuario intenta crear un grupo con un nombre que ya existe en su lista, cuando selecciona “Añadir grupo”, entonces debe mostrarse un mensaje de error indicando que el nombre ya está en uso.

3. Nombre de grupo vacío

- Dado que el usuario no proporciona un nombre para el grupo, cuando intenta crearlo, entonces debe mostrarse un mensaje solicitando que se introduzca un nombre válido.

2.7 Conversión a usuario premium

- **Como** usuario registrado
- **quiero** poder convertirme en usuario premium pagando una suscripción
- **para** poder acceder a funcionalidades adicionales.

Criterios de aceptación:

1. Convertirse en premium

- Dado que el usuario no es premium, cuando selecciona “Convertirse en Premium” y paga la suscripción, debe recibir la confirmación de su estado premium.

2. Usuario ya premium

- Dado que el usuario ya es premium, cuando selecciona esta opción, debe ver un mensaje indicando que ya tiene una suscripción activa.

2.8 Exportar mensajes en PDF

- **Como** usuario premium
- **quiero** poder exportar mis mensajes en formato PDF
- **para** guardarlos en mi equipo.

Criterios de aceptación:

1. Exportación exitosa

- Dado que el usuario premium selecciona una conversación con un contacto, cuando elige la opción “Exportar PDF”, entonces se debe generar un archivo PDF con los mensajes intercambiados en esa conversación.

2. Funcionalidad exclusiva para premium

- Dado que el usuario no es premium, cuando intenta exportar una conversación, entonces debe aparecer un mensaje indicándole que esta funcionalidad es exclusiva para usuarios premium.

2.9 Aplicar descuentos en premium

- **Como** usuario premium
- **quiero** poder aplicar descuentos
- **para** pagar mi suscripción con un precio menor.

Criterios de aceptación:

1. Descuento aplicable

- Dado que el usuario se registra en un intervalo de fechas específico o ha enviado más de un cierto número de mensajes en el último mes, cuando selecciona “Convertirse en Premium”, debe aplicarse el descuento correspondiente.

2. No apto para descuento

- Dado que el usuario no es apto para un descuento, el sistema debe notificarlo.

2.10 Aplicación de múltiples filtros de búsqueda

- **Como** usuario registrado
- **quiero** combinar diferentes criterios de búsqueda
- **para** realizar búsquedas precisas en mi historial de mensajes.

Criterios de aceptación:

1. Búsqueda combinada

- Dado que el usuario configura múltiples filtros, cuando activa la búsqueda, entonces los resultados deben cumplir simultáneamente con todos los criterios especificados.

2. Visualización de resultados

- Dado que el sistema encuentra mensajes coincidentes con los filtros, cuando muestra los resultados, entonces estos deben aparecer ordenados cronológicamente.

3. Acceso a conversación completa

- Dado que se muestran resultados de búsqueda, cuando el usuario selecciona uno de ellos, entonces debe abrirse la conversación completa posicionada en el mensaje seleccionado.

3 Diagramas de clases UML

3.1 Primera entrega

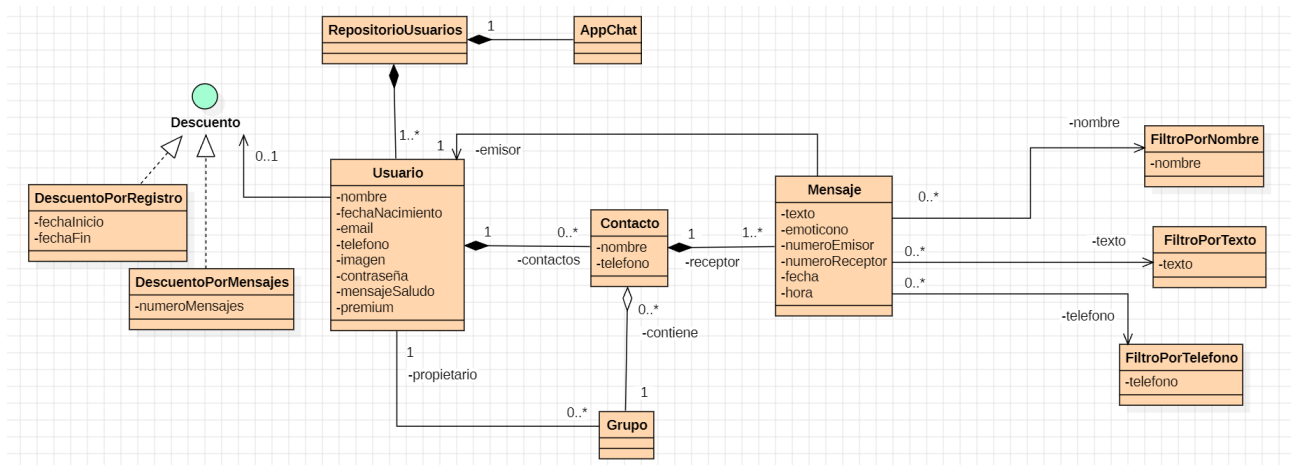


Figura 1: Diagrama de clases AppChat creado en StarUML (primera entrega)

El **modelo inicial propuesto** en la primera entrega presentaba una estructura básica que, aunque funcional, carecía de algunos elementos esenciales para favorecer la extensibilidad y el mantenimiento del código.

3.2 Utilizado en el proyecto

El **diagrama final implementado** es el que fue proporcionado por el profesor. Este incluye una estructura más clara de las relaciones entre clases, una mejor definición de las responsabilidades y un diseño más alineado con los patrones utilizados como Singleton, Factoría, Estrategia, Adaptador y los principios GRASP. Este diagrama corregido ha sido fundamental para implementar correctamente la arquitectura de la aplicación, permitiendo una separación adecuada entre el modelo, la vista y el controlador.

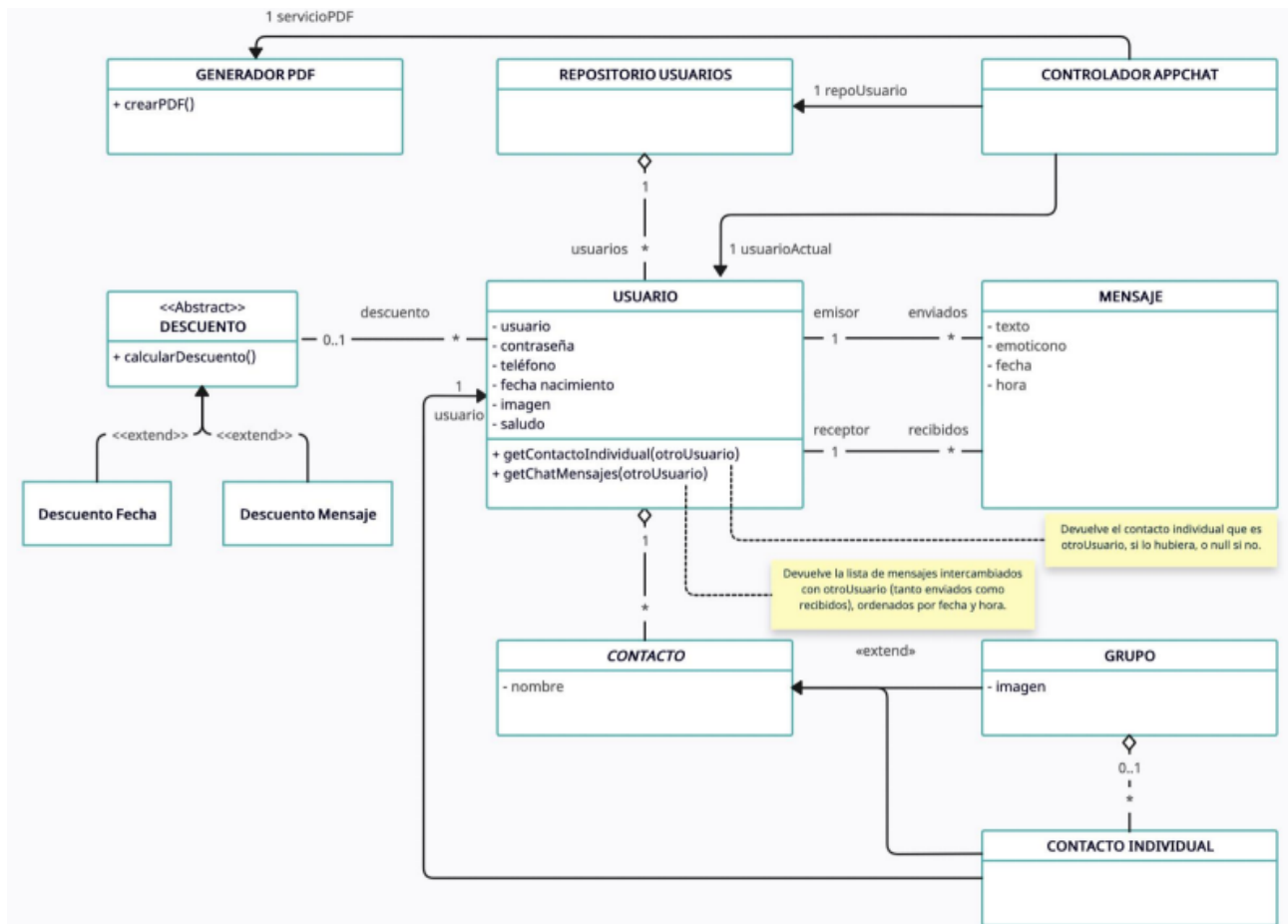


Figura 2: Diagrama de clases AppChat utilizado en el proyecto

Además, decidimos incorporar la clase Chat propuesta por el profesor, a pesar de que el modelo original funcionaba relacionando directamente Usuario y Mensaje. De esta manera, se puede encapsular mejor toda la lógica relacionada con las conversaciones, y facilitando la gestión del historial de mensajes.

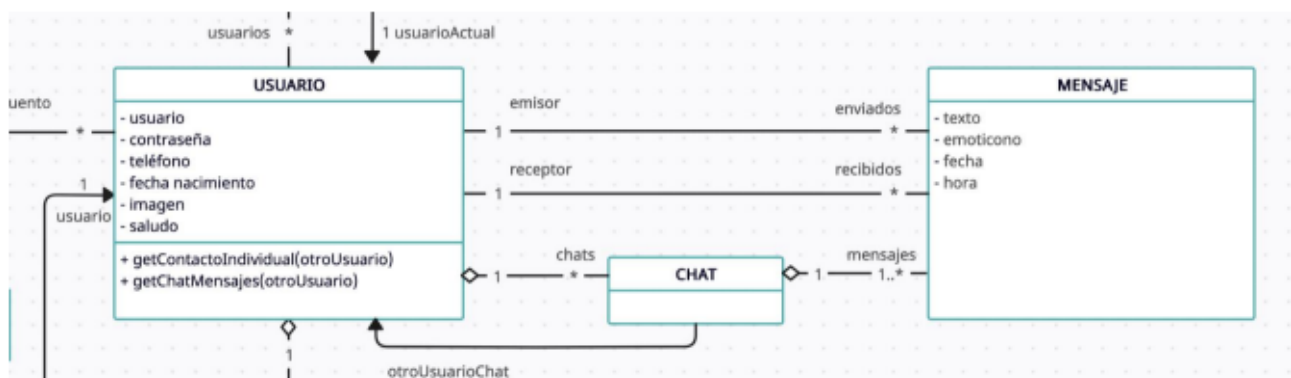


Figura 3: Diagrama de clases AppChat utilizado en el proyecto

La implementación de la clase Chat ha aportado dos beneficios principales: por un lado, mejora la escalabilidad del sistema, permitiendo incorporar fácilmente nuevas funcionalidades

sin modificar otras clases, y por otro lado, optimiza el rendimiento de las consultas, lo que se puede ver favorecido en el caso de tener un gran volumen de mensajes.

4 Diagrama de secuencia

4.1 Añadir contacto a grupo

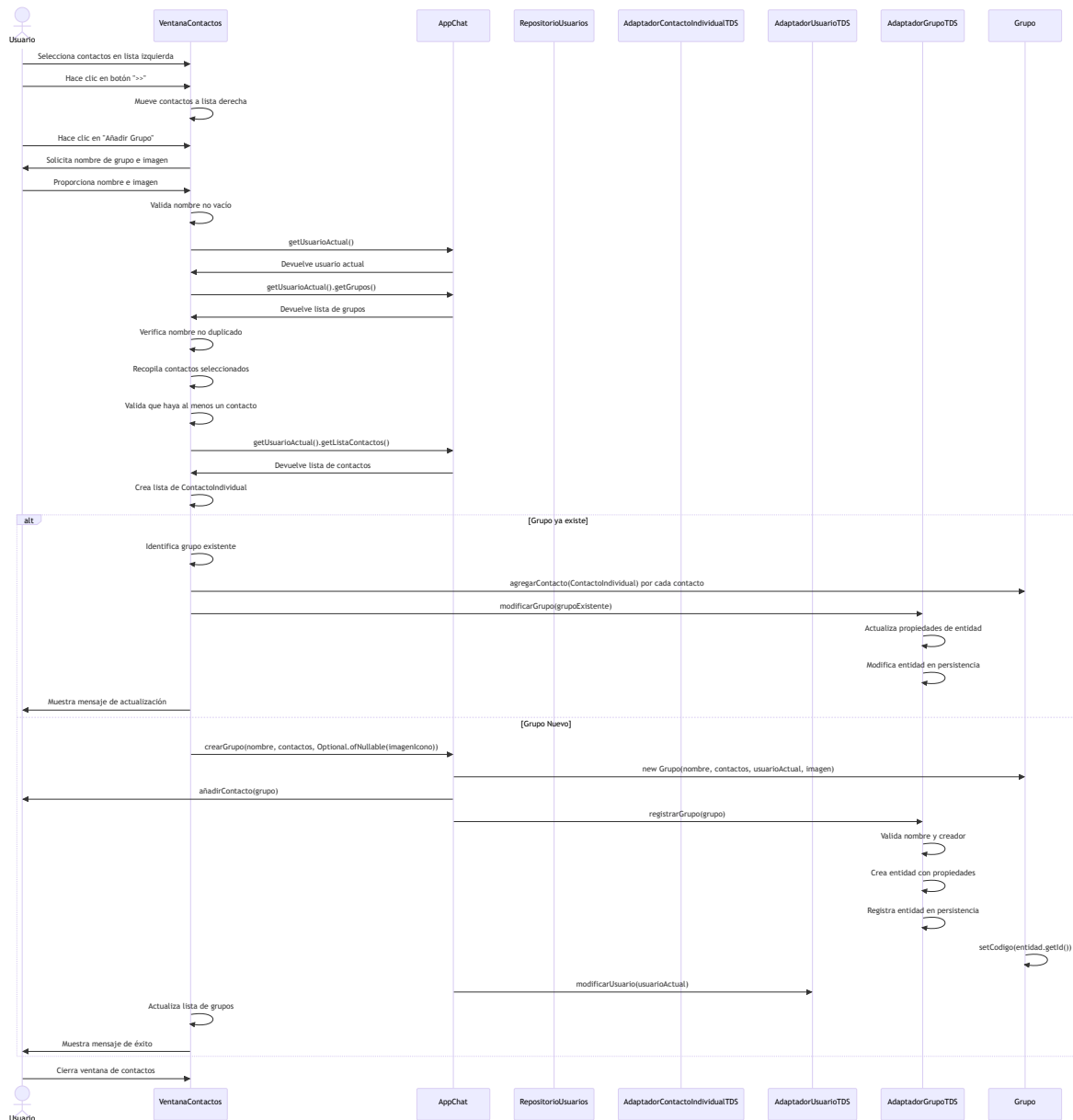


Figura 4: Diagrama de secuencia para añadir un contacto a un grupo

5 Arquitectura de la aplicación

Nuestra aplicación *AppChat* implementa una arquitectura multicapa basada en el patrón Modelo-Vista-Controlador (MVC), siguiendo los principios GRASP para garantizar la mantenibilidad y escalabilidad del software.

5.1 Diseño en capas

La arquitectura se estructura en las siguientes capas principales:

- **Capa de presentación (Vista):** Contiene todas las clases que conforman la interfaz gráfica de usuario, implementadas mediante la biblioteca Swing de Java. Se localiza en el paquete `umu.tds.apps.vista`.
- **Capa de negocio (Controlador):** Centralizada en la clase `AppChat`, dentro del paquete `umu.tds.apps.controlador`, que implementa el patrón Singleton. Se encarga de coordinar todas las operaciones entre la vista y el modelo.
- **Capa de modelo (Modelo):** Contiene las entidades del dominio como `Usuario`, `Mensaje`, `Chat`, `Contacto` y `Grupo` en el paquete `umu.tds.apps.modelo`.
- **Capa de persistencia:** Implementa el patrón DAO (*Data Access Object*) mediante las interfaces e implementaciones en el paquete `umu.tds.apps.persistencia`, utilizando el framework TDS para almacenamiento de datos.

5.2 Relaciones entre capas

El flujo de interacción entre las capas sigue los principios del patrón MVC:

1. La **Vista** captura los eventos del usuario y los transmite al Controlador.
2. El **Controlador** procesa estos eventos, ejecuta la lógica de negocio correspondiente y actualiza el Modelo.
3. El **Modelo** notifica de sus cambios a través de mecanismos de observación.
4. La **Vista** se actualiza para reflejar los cambios en el Modelo.

Esta separación clara de responsabilidades garantiza un alto grado de desacoplamiento entre componentes, facilitando así el mantenimiento y la extensibilidad del sistema.

5.3 Persistencia de datos

El sistema de persistencia se basa en:

- El motor de base de datos relacional H2.
- El servidor de persistencia TDS para la gestión de las operaciones de almacenamiento.
- Un sistema de caché mediante el patrón Pool (`PoolDAO`) que mejora el rendimiento evitando recuperaciones redundantes de objetos.
- El patrón Factoría (`FactoriaDAO`) para la creación de adaptadores específicos, promoviendo la extensibilidad.

5.4 Gestión de dependencias

La aplicación utiliza Maven para la gestión de dependencias externas, incluyendo:

- Las bibliotecas Swing para la interfaz gráfica.
- El framework de persistencia TDS.
- La biblioteca iText para la generación de documentos PDF.

5.5 Características arquitectónicas clave

- **Desacoplamiento:** Las capas están claramente separadas y se comunican a través de interfaces bien definidas.
- **Encapsulamiento:** Cada componente oculta sus detalles de implementación y expone solo las funcionalidades necesarias.
- **Cohesión:** Los componentes tienen responsabilidades bien definidas y enfocadas en una única tarea.
- **Extensibilidad:** La arquitectura permite añadir nuevas funcionalidades con cambios mínimos en el código existente.

Esta arquitectura proporciona una base sólida que facilita la evolución futura de la aplicación, cumpliendo con los requisitos de calidad establecidos para el proyecto.

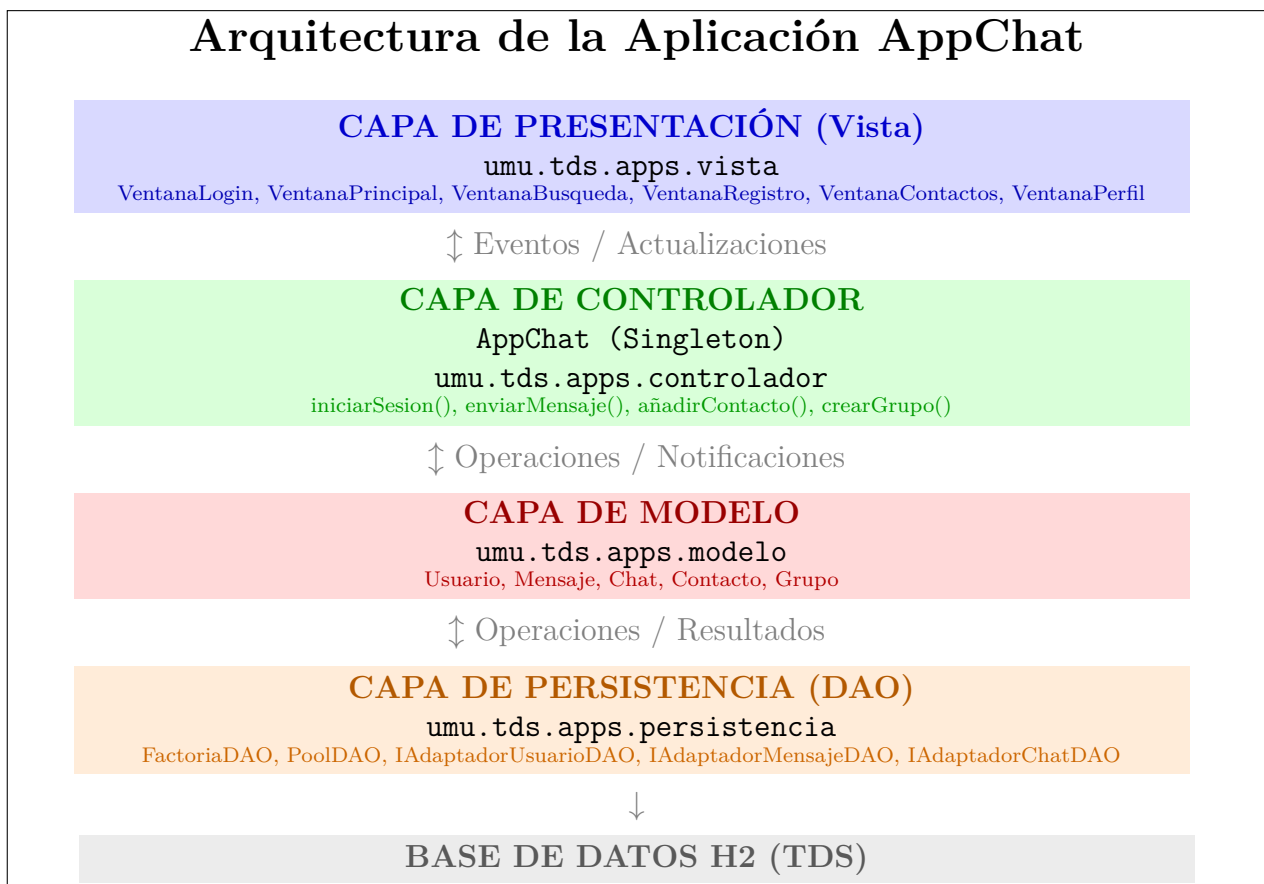


Figura 5: Arquitectura multicapa de AppChat

6 Patrones de diseño empleados

6.1 Patrones implementados directamente

6.1.1 Singleton

El controlador AppChat implementa el **patrón Singleton** para **garantizar que solo exista una instancia** que coordine todas las operaciones entre vista y modelo. De manera similar, **RepositorioUsuarios** también aplica este patrón para ofrecer **un punto único de acceso al almacén de usuarios** del sistema.

La implementación en la clase AppChat muestra claramente este patrón:

```
1 public class AppChat {
2     private static AppChat unicaInstancia = null;
3
4     // Constructor privado
5     private AppChat() {
6         inicializarAdaptador();
7         inicializarRepositorio();
8     }
9
10    // Metodo publico para obtener la unica instancia
11    public static AppChat getUnicaInstancia() {
12        if (unicaInstancia == null) {
13            unicaInstancia = new AppChat();
14        }
15        return unicaInstancia;
16    }
17
18    // Resto de metodos de la clase...
19 }
```

Listing 1: Implementación del patrón Singleton en AppChat

Este enfoque nos proporciona varias ventajas:

- **Control centralizado** de la aplicación
- **Acceso global único** al controlador
- Evita la creación de múltiples instancias que podrían provocar inconsistencias

6.1.2 Factoría

FactoriaDAO nos permite crear diferentes tipos de adaptadores DAO. Este patrón facilita la creación de adaptadores de persistencia de manera flexible sin la necesidad de acoplar el código a implementaciones concretas.

```
1 public abstract class FactoriaDAO {
2     private static FactoriaDAO unicaInstancia;
3
4     public static final String DAO_TDS = "umu.tds.apps.persistencia.
5         TDSFactoriaDAO";
6
7     // Obtener instancia de factoria segun tipo
```

```

7      public static FactoriaDAO getInstancia(String tipo) throws
        DAOException {
8          if (unicaInstancia == null)
9              try {
10                  unicaInstancia = (FactoriaDAO) Class.forName(tipo).
                    newInstance();
11              } catch (Exception e) {
12                  throw new DAOException(e.getMessage());
13              }
14          return unicaInstancia;
15      }
16
17      protected FactoriaDAO() {}
18
19      // Metodos factoria para cada tipo de adaptador
20      public abstract IAdaptadorUsuarioDAO getUsuarioDAO();
21      public abstract IAdaptadorMensajeDAO getMensajeDAO();
22      public abstract IAdaptadorContactoIndividualDAO
        getContactoIndividualDAO();
23      public abstract IAdaptadorChatDAO getChatDAO();
24      public abstract IAdaptadorGrupoDAO getGrupoDAO();
25  }

```

Listing 2: Implementación del patrón Factoría con FactoriaDAO

La implementación concreta TDSFactoriaDAO extiende esta clase y proporciona las implementaciones específicas:

```

1  public class TDSFactoriaDAO extends FactoriaDAO {
2      public TDSFactoriaDAO() {
3      }
4
5      @Override
6      public IAdaptadorUsuarioDAO getUsuarioDAO() {
7          return AdaptadorUsuarioTDS.getUnicaInstancia();
8      }
9
10     @Override
11     public IAdaptadorMensajeDAO getMensajeDAO() {
12         return AdaptadorMensajeTDS.getUnicaInstancia();
13     }
14
15     // Otras implementaciones de metodos...
16 }

```

Listing 3: Implementacion concreta de la Factoria

Este patrón ofrece las siguientes ventajas:

- **Desacoplamiento** entre clases cliente y las implementaciones concretas
- **Flexibilidad para cambiar implementaciones** de persistencia **sin afectar al resto del código**
- Capacidad para **extender con nuevos adaptadores** sin modificar código existente

6.1.3 Estrategia

Este patrón define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Las clase Descuento implementa diferentes estrategias para calcular descuentos, así como la clase BusquedaMensaje.

```
1 public interface Descuento {
2     public double calcularDescuento(double precio);
3 }
4
5 public class EstrategiaDescuento {
6     private Descuento estrategiaDescuento;
7
8     public void setEstrategiaDescuento(Descuento estrategiaDescuento)
9     {
10         this.estrategiaDescuento = estrategiaDescuento;
11     }
12
13     public double calcularPrecioFinal(double precioOriginal) {
14         if (estrategiaDescuento == null) {
15             return precioOriginal; // Sin descuento si no hay
16                                     estrategia
17         }
18         return estrategiaDescuento.calcularDescuento(precioOriginal);
19     }
20 }
```

Listing 4: Interfaz Descuento y EstrategiaDescuento

Las implementaciones concretas incluyen:

```
1 public class DescuentoFecha implements Descuento {
2     private LocalDate fechaRegistro;
3
4     public DescuentoFecha(LocalDate fechaRegistro) {
5         this.fechaRegistro = fechaRegistro;
6     }
7
8     @Override
9     public double calcularDescuento(double precio) {
10         // Aplicamos 20% de descuento si el usuario se registro hace
11             menos de 7 dias
12         long diasDesdeRegistro = ChronoUnit.DAYS.between(fechaRegistro
13             , LocalDate.now());
14
15         if (diasDesdeRegistro < 7) {
16             return precio * 0.8; // 20% de descuento
17         }
18
19         // Sin descuento
20         return precio;
21     }
22 }
23
24 public class DescuentoMensaje implements Descuento {
25     private int mensajesEnviados;
26 }
```



```

24
25 public DescuentoMensaje(int mensajesEnviados) {
26     this.mensajesEnviados = mensajesEnviados;
27 }
28
29 @Override
30 public double calcularDescuento(double precio) {
31     // Diferentes descuentos segun el numero de mensajes enviados
32     if (mensajesEnviados > 300) {
33         return precio * 0.7; // 30% de descuento
34     } else if (mensajesEnviados > 200) {
35         return precio * 0.8; // 20% de descuento
36     } else if (mensajesEnviados > 100) {
37         return precio * 0.85; // 15% de descuento
38     }
39
40     // Sin descuento
41     return precio;
42 }
43 }

```

Listing 5: Implementaciones concretas de estrategias de descuento

Este patrón proporciona:

- **Flexibilidad** para cambiar algoritmos en tiempo de ejecución
- **Encapsulamiento** de algoritmos en clases separadas
- **Facilidad para añadir nuevas estrategias** sin modificar código existente
- Eliminación de condiciones complejas dentro del código cliente

6.1.4 Pool

PoolDAO implementa un patrón de Pool de objetos para **gestionar la caché de objetos persistidos**. Este patrón es crucial para mejorar el rendimiento, evitando múltiples instancias del mismo objeto.

```

1 public class PoolDAO {
2     private static PoolDAO unicaInstancia;
3     private Hashtable<Integer, Object> pool;
4
5     private PoolDAO() {
6         pool = new Hashtable<Integer, Object>();
7     }
8
9     public static PoolDAO getUnicaInstancia() {
10         if (unicaInstancia == null) unicaInstancia = new PoolDAO();
11         return unicaInstancia;
12     }
13
14     public Object getObjeto(int id) {
15         return pool.get(id);
16     } // devuelve null si no encuentra el objeto
17 }

```

```

18 public void addObjeto(int id, Object objeto) {
19     pool.put(id, objeto);
20 }
21
22 public boolean contiene(int id) {
23     return pool.containsKey(id);
24 }
25 }

```

Listing 6: Implementación del patrón Pool con PoolDAO

El uso típico de este pool se ve en los adaptadores, por ejemplo:

```

1 public Usuario recuperarUsuario(int codigo) {
2     // 1) Consultar la cache
3     if (PoolDAO.getUnicaInstancia().contiene(codigo)) {
4         return (Usuario) PoolDAO.getUnicaInstancia().getObjeto(codigo)
5         ;
6     }
7
8     // 2) Recuperar la entidad de la base de datos
9     Entidad eUsuario = servPersistencia.recuperarEntidad(codigo);
10    if (eUsuario == null) {
11        return null;
12    }
13
14    // 3) Crear el objeto Usuario
15    Usuario usuario = new Usuario(/* parametros */);
16    usuario.setCodigo(codigo);
17
18    // 4) Anadir al PoolDAO antes de establecer relaciones para evitar
19    // recursion
20    PoolDAO.getUnicaInstancia().addObjeto(codigo, usuario);
21
22    // 5) Establecer relaciones y propiedades
23    // ...
24    return usuario;
25 }

```

Listing 7: Uso del PoolDAO en un adaptador

Las ventajas de este patrón incluyen:

- **Mejora del rendimiento** al evitar múltiples recuperaciones del mismo objeto
- Resolución de referencias circulares durante la carga de objetos
- **Reducción de consultas** a la base de datos
- **Garantiza la identidad de los objetos** (misma instancia para el mismo ID)

6.2 Adaptador

Los adaptadores de persistencia (AdaptadorUsuarioTDS, AdaptadorMensajeTDS, etc.) siguen el patrón Adapter. **Permiten adaptar la persistencia de datos entre diferentes**

implementaciones, convirtiendo la interfaz de persistencia de TDS a la interfaz esperada por nuestra aplicación.

```
1 public interface IAdaptadorUsuarioDAO {
2     public void registrarUsuario(Usuario usuario);
3     public void modificarUsuario(Usuario usuario);
4     public Usuario recuperarUsuario(int id);
5     public List<Usuario> recuperarTodosUsuarios();
6 }
7
8 public class AdaptadorUsuarioTDS implements IAdaptadorUsuarioDAO {
9     private static ServicioPersistencia servPersistencia;
10    private static AdaptadorUsuarioTDS unicaInstancia = null;
11
12    private AdaptadorUsuarioTDS() {
13        servPersistencia = FactoriaServicioPersistencia.getInstance().
14            getServicioPersistencia();
15    }
16
17    public static AdaptadorUsuarioTDS getUnicaInstancia() {
18        if (unicaInstancia == null) {
19            unicaInstancia = new AdaptadorUsuarioTDS();
20        }
21        return unicaInstancia;
22    }
23
24    @Override
25    public void registrarUsuario(Usuario usuario) {
26        // Convertir Usuario a Entidad de TDS
27        Entidad eUsuario = new Entidad();
28        eUsuario.setNombre("usuario");
29        eUsuario.setPropiedades(new ArrayList<>(Arrays.asList(
30            new Propiedad("nombre", usuario.getUsuario()),
31            new Propiedad("telefono", usuario.getTelefono()),
32            // Mas propiedades...
33        )));
34
35        // Registrar en el servicio de persistencia
36        eUsuario = servPersistencia.registrarEntidad(eUsuario);
37        usuario.setCodigo(eUsuario.getId());
38    }
39
40    // Resto de metodos...
```

Listing 8: Interfaz del adaptador y su implementación

Este patrón proporciona:

- **Separación** entre la lógica de negocio y los detalles de persistencia
- **Conversión** entre el modelo de objetos de nuestra aplicación y el modelo de datos del framework TDS
- **Capacidad para cambiar** la tecnología de persistencia sin afectar al resto de la aplicación

6.3 Patrones empleados indirectamente

6.3.1 GRASP y separación modelo-vista

La estructura de la aplicación está delineada mediante una clara división de los componentes clave según el patrón **MVC (Modelo-Vista-Controlador)**, lo que facilita tanto la organización como el mantenimiento del sistema.

El **modelo** está formado por clases ubicadas en el paquete `umu.tds.apps.modelo`, como **Usuario**, **Chat** y **Mensaje**, que encapsulan la lógica del negocio y los datos de la aplicación. Las **vistas** se representan a través de clases del paquete `umu.tds.apps.vista`, como **VentanaPrincipal** y **VentanaRegistro**. Finalmente, el **controlador** se encuentra en la clase **AppChat**, del paquete `umu.tds.apps.controlador`, y es el encargado de coordinar la interacción entre el modelo y la vista.

Varios principios **GRASP** se ven en el diseño:

En primer lugar, el principio de **Creador** se refleja en la clase **AppChat**, la cual se encarga de crear instancias clave como **Usuario**, **Mensaje** y **Grupo**.

El principio de **Experto** se aplica en la clase **Usuario**, ya que gestiona sus propios mensajes, chats y contactos. En cuanto al Bajo Acoplamiento, se logra gracias a la separación de responsabilidades entre clases y al uso de interfaces DAO, que permiten cambiar implementaciones sin afectar al resto del sistema. Además, el diseño muestra Alta Cohesión, con clases bien enfocadas, como **EstrategiaDescuento**, que se encarga exclusivamente de calcular descuentos.

Por último, el principio de **Controlador** se cumple al tener un controlador en la clase **AppChat**. Esta actúa como el mediador principal, administrando las interacciones de la vista y el modelo.

6.3.2 Patrones en Swing

Swing utiliza varios patrones de diseño:

- **Patrón Observer:** Para manejar eventos de la interfaz de usuario.
- **Patrón Composite:** Para construir interfaces gráficas jerárquicas con componentes.
- **Patrón Decorator:** Para añadir funcionalidades a componentes UI, como puede ser el caso de los bordes.

6.3.3 Patrones en el Servicio de Persistencia (TDS)

El servicio de persistencia proporcionado por los profesores utiliza el patrón Factoría **FactoriaServicioPersistencia** para crear instancias del servicio. Además, la capa de persistencia combina varios patrones, que ya han sido explicados anteriormente:

- **Abstract Factory (FactoriaDAO):** Define una interfaz para crear familias de objetos DAO.
- **Singleton:** Asegura una única instancia de cada adaptador y factoría.
- **Adapter:** Convierte entre objetos del dominio y entidades de persistencia.

- **DAO:** Encapsula la lógica de acceso a datos.
- **Object Pool:** Mantiene un caché de objetos para evitar duplicados.

En definitiva, esta combinación de patrones permite una separación clara entre la lógica de negocio y la persistencia, facilitando el mantenimiento y la extensibilidad de la aplicación.

7 Manual de uso

7.1 Acceso a la aplicación

Para poder ejecutar la aplicación, debe de estar ejecutándose el **servidor de persistencia H2**, tal y como se muestra en la siguiente figura:

```
C:\Users\evapi\Downloads\apache-maven-3.9.9\ServidorPersistenciaH2>java -jar ServidorPersistenciaH2.jar
Objetos distribuidos listos : FactoriaServicioPersistencia
Presionar ENTER para salir (los objetos distribuidos seran destruidos)...
```

Figura 6: Ejecución del servidor de persistencia H2

Posteriormente, ya podremos iniciar la aplicación desde la ventana de login.

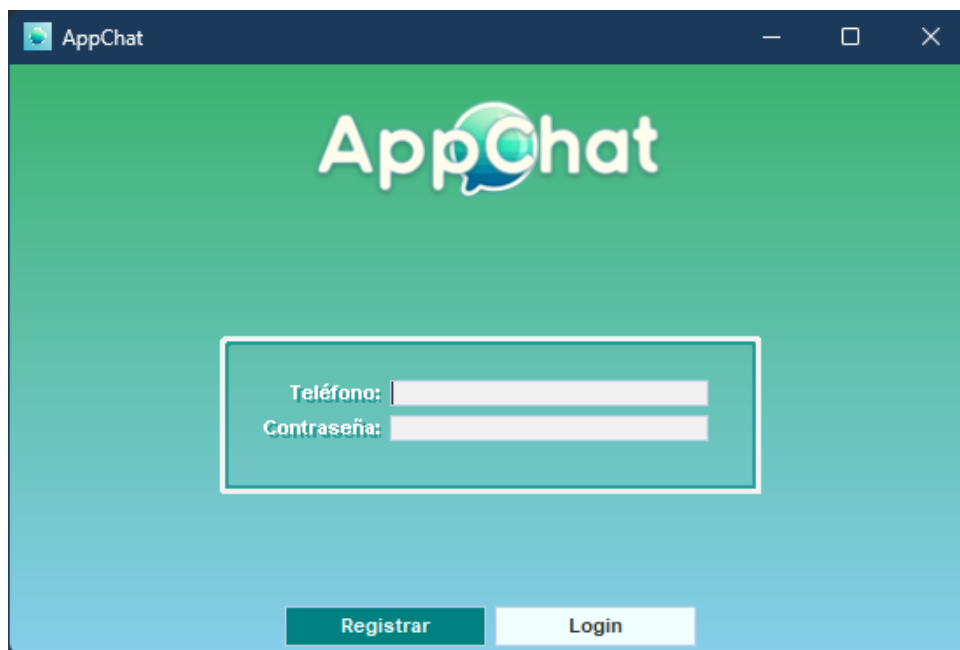


Figura 7: Ventana de login

7.2 Registro de usuario

Si es la primera vez que utiliza la aplicación, debe registrarse como un nuevo usuario empleando el botón “**Registrar**” de la ventana de login.

Para el registro, debe completar todo el formulario de registro con sus datos personales, tal y como aparece en la Figura [8].

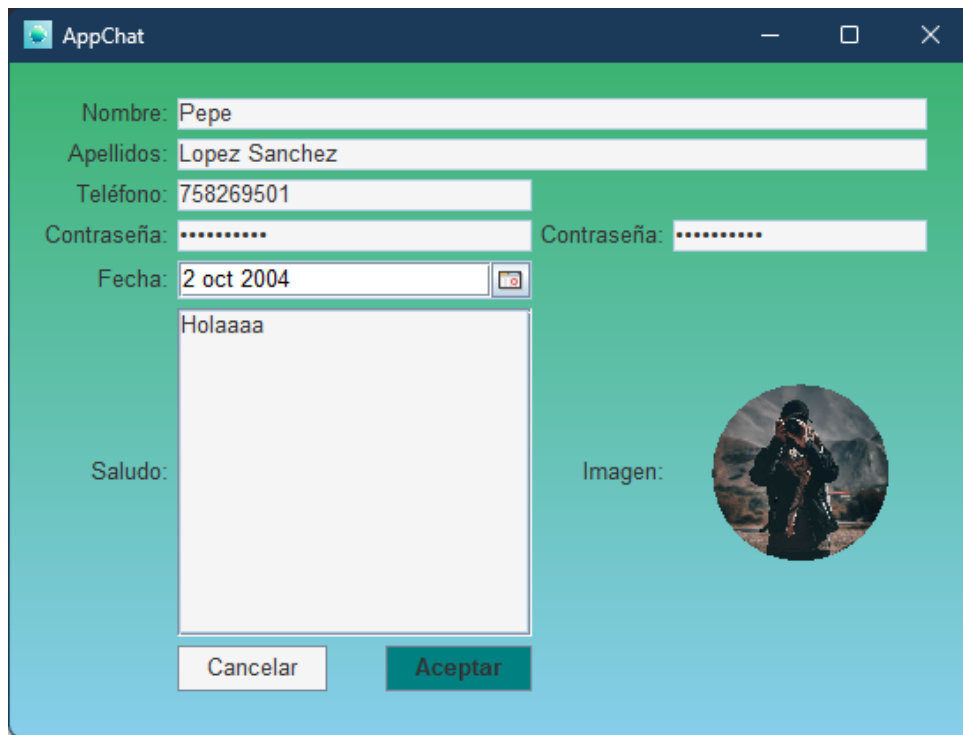


Figura 8: Registro de un usuario en el sistema

Además, se puede establecer una imagen de perfil, haciendo click encima de la imagen predeterminada que aparece. Al realizar esto, se abrirá una ventana como en la Figura [9], indicando dos opciones:

1. **“Introducir enlace”**: Para utilizar una imagen de internet
2. **“Seleccionar archivo”**: Para elegir una imagen almacenada en su equipo

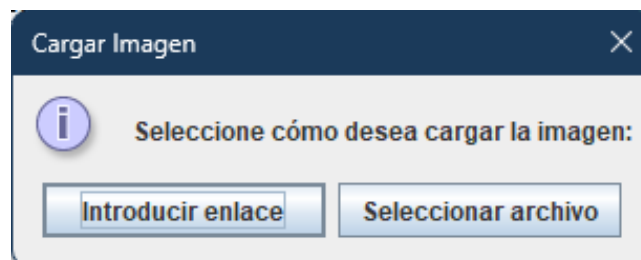


Figura 9: Elegir foto de perfil

Tras completar todos los campos obligatorios y hacer clic en **“Aceptar”**, su cuenta quedará registrada y podrá iniciar sesión.

7.3 Inicio de sesión

Para acceder a su cuenta de *AppChat*, debe introducir el número de teléfono, y su contraseña en la ventana principal, y posteriormente hacer click en el botón **“Login”**.

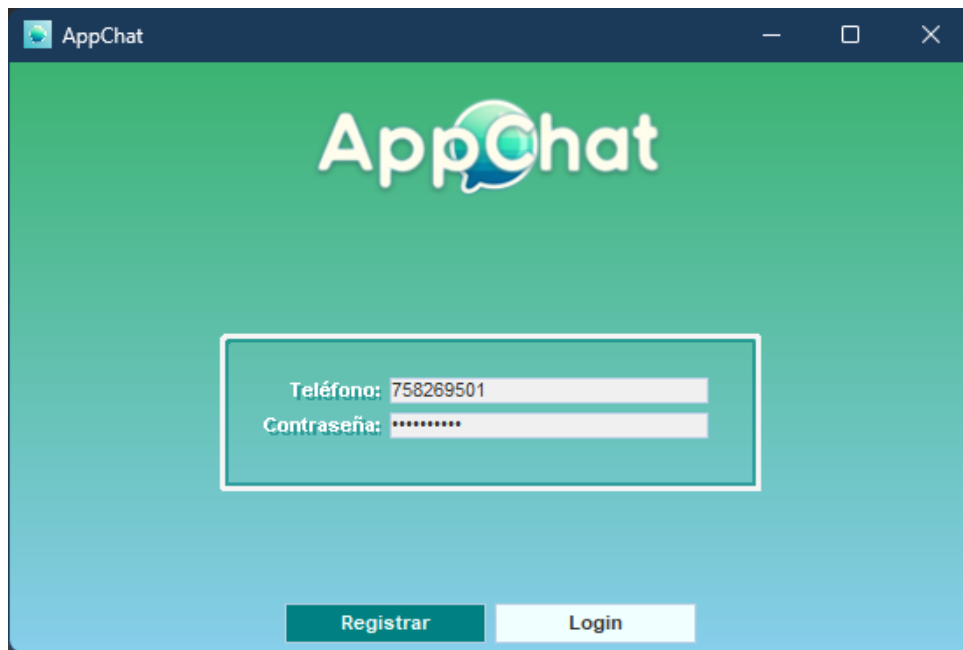


Figura 10: Introducir datos de usuario

Si los datos son correctos, accederá directamente a la pantalla principal, donde ya podrá hacer uso de la aplicación. En caso contrario, la aplicación le informará de que los datos son incorrectos.

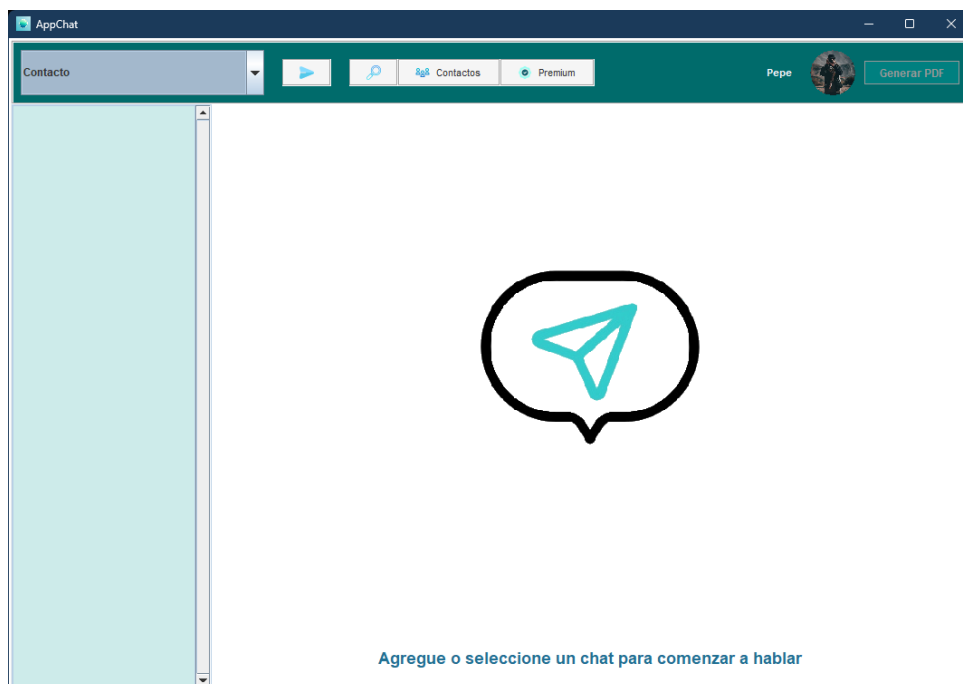


Figura 11: Pantalla principal al iniciar sesión

7.4 Envío de mensajes

AppChat ofrece distintas formas para iniciar una conversación. Si nos situamos en la esquina superior izquierda de la aplicación, se observará un desplegable con dos opciones “**Teléfono**” y “**Contacto**”.

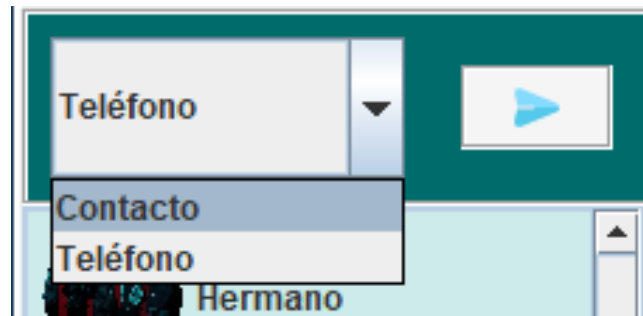


Figura 12: ComboBox para seleccionar número o contacto

7.4.1 Envío por número de teléfono

Al seleccionar la opción de “**Teléfono**” del desplegable, si se pulsa al botón situado justo en la derecha, con un icono de envío, aparecerá una ventana para introducir el número de teléfono al que queremos escribir.

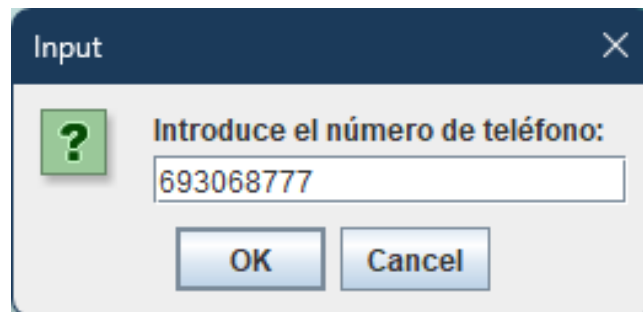


Figura 13: Introducir número de teléfono

Si es un número registrado, se abrirá el chat correspondiente. En el panel inferior del chat, podrá escribir el texto que desee.

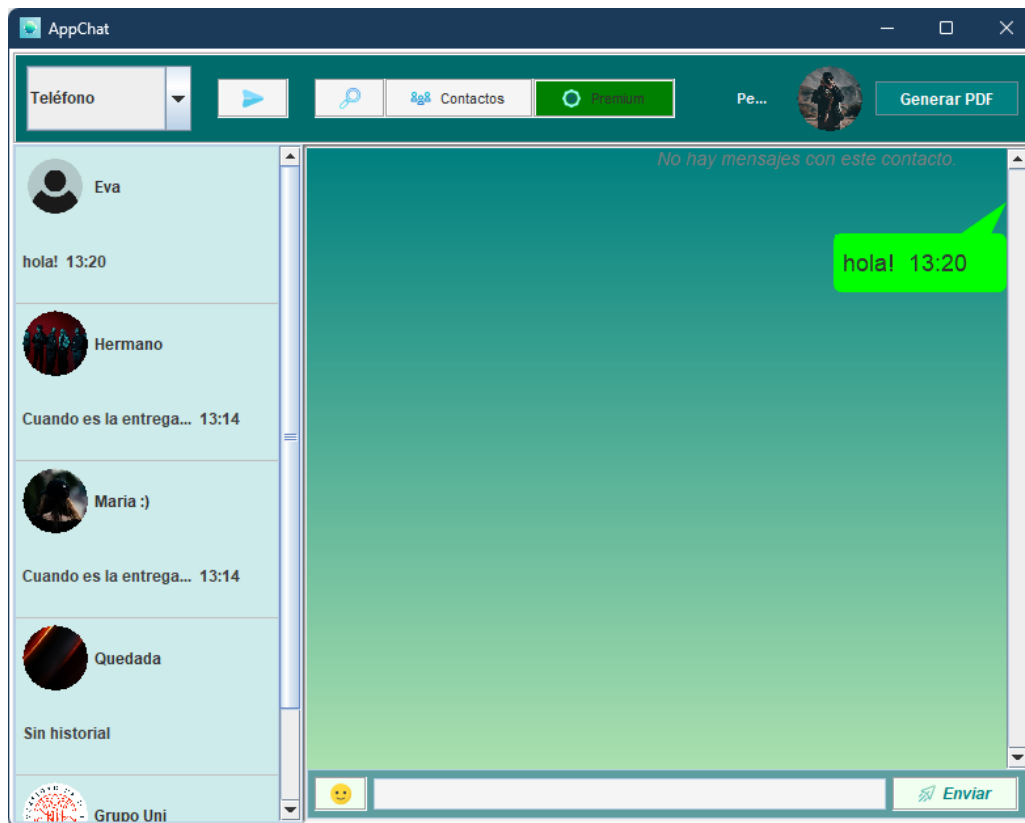


Figura 14: Chat a través de opción 'Teléfono'

7.4.2 Envío por nombre de contacto

La otra opción es empleando un nombre de un contacto, si ya lo tiene agregado. Se repite el proceso anterior, pero seleccionando la opción de “**Contacto**” en el desplegable. Pero esta vez, en la ventana que aparece se introducirá el nombre del contacto, o incluso grupo que haya creado el usuario.

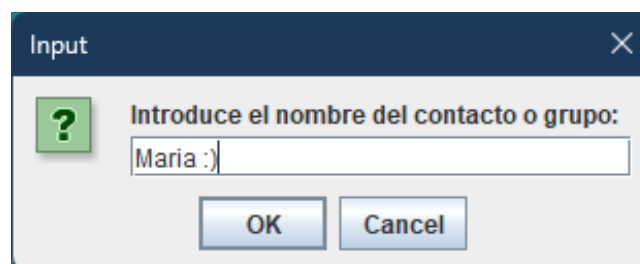


Figura 15: Introducir nombre del contacto

Al igual que antes, se abrirá el chat correspondiente.

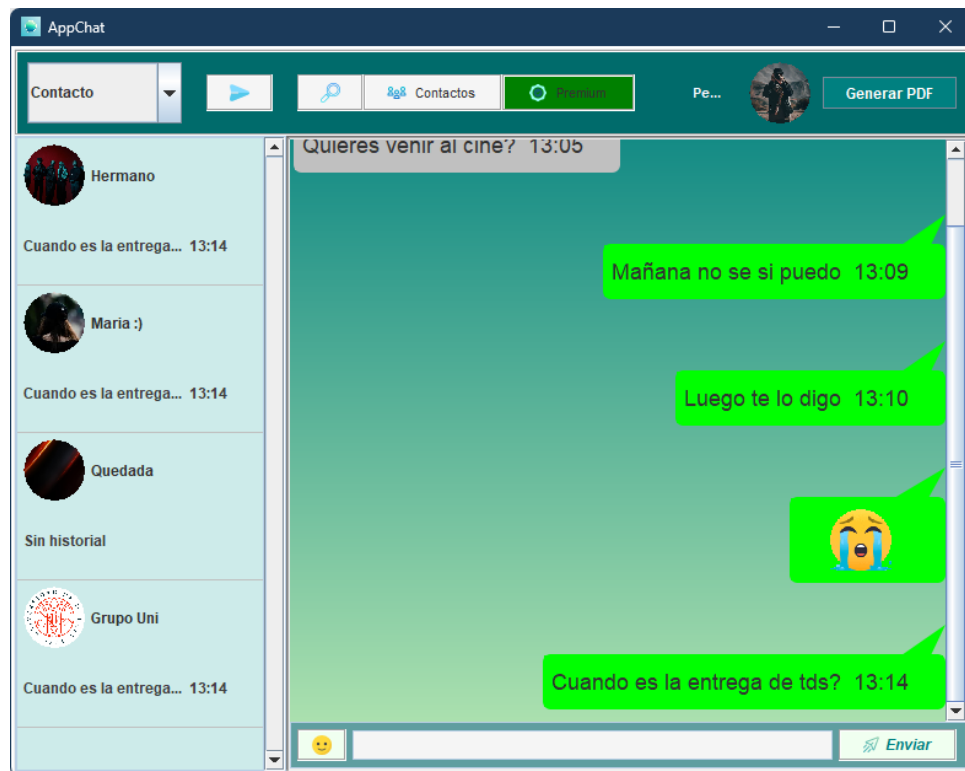


Figura 16: Chat a través de opción 'Contacto'

Por otra parte, para el envío de mensajes, se mencionó el panel inferior. En este, se ubica el campo de texto, para escribir los mensajes, y a la izquierda, un botón de emojis. Para enviar el mensaje del campo de texto, basta con pulsar el botón de enviar de la derecha, o incluso usando la tecla “**Enter**”. Al enviarlo, el mensaje aparecerá en el chat como una burbuja.

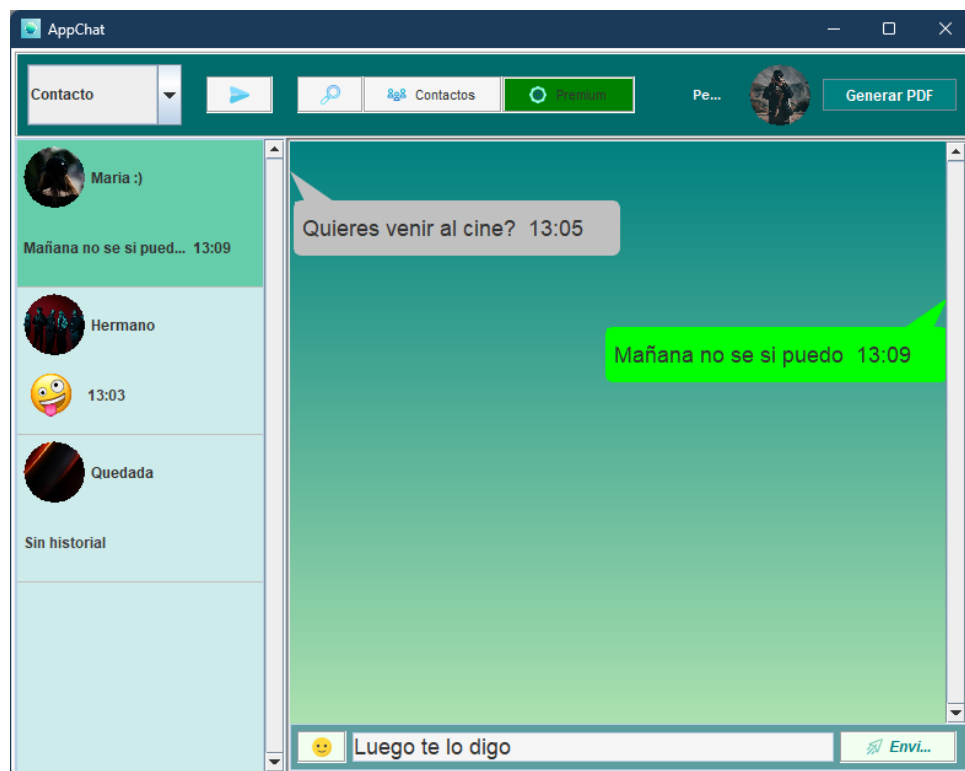


Figura 17: Envío de mensajes al chat

Para el envío de emojis, basta con pulsar el botón de emojis, y seleccionar el emoji que se desee enviar. Automáticamente, el emoji se enviará al chat.

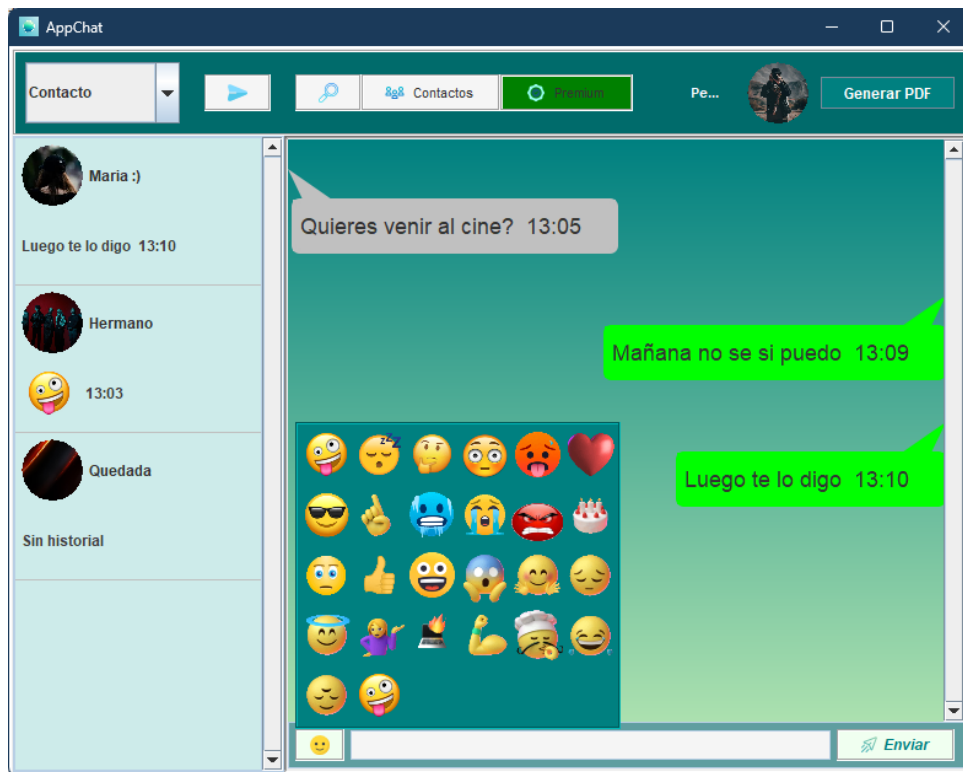


Figura 18: Panel de emojis

7.5 Crear grupo

Para crear un nuevo grupo, debe pulsar el botón “**Contactos**” de la barra superior. Automáticamente, se abre una ventana como la Figura [19].

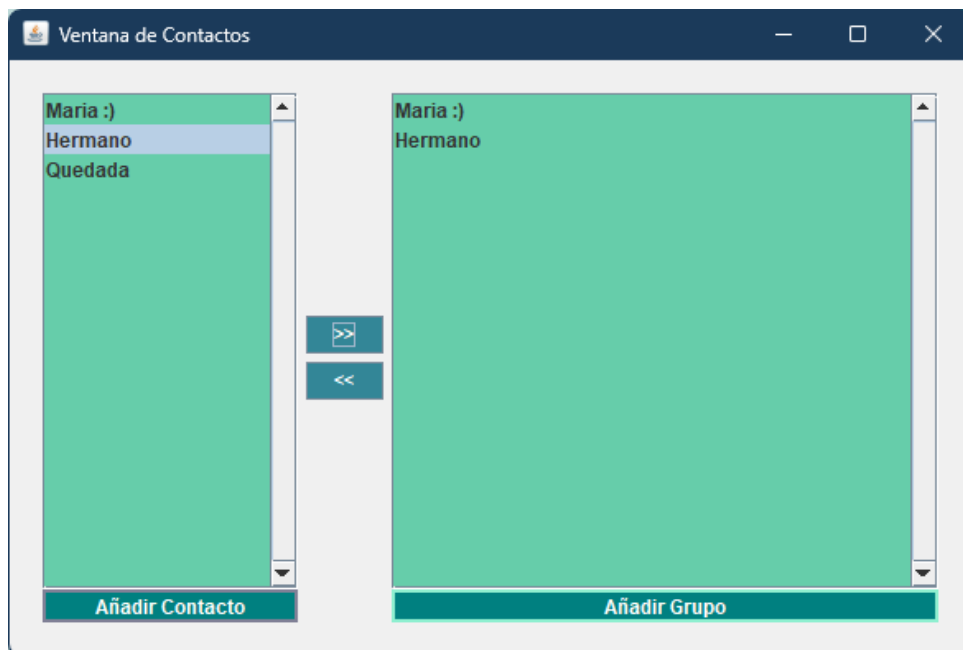


Figura 19: Ventana para crear un grupo

En la parte izquierda de dicha ventana, aparecen los contactos y grupos que tiene el usuario. De esta manera, el usuario selecciona los usuarios que desee incluir con los botones de >>. Una vez finalizado, debe pulsar “**Añadir Grupo**”.

Para la creación del grupo, le aparecerá la siguiente ventana, en la que puede ponerle un nombre, y seleccionar una imagen local.



Figura 20: Crear un grupo

El nuevo grupo creado le aparecerá automáticamente en la lista de mensajes de la izquierda.

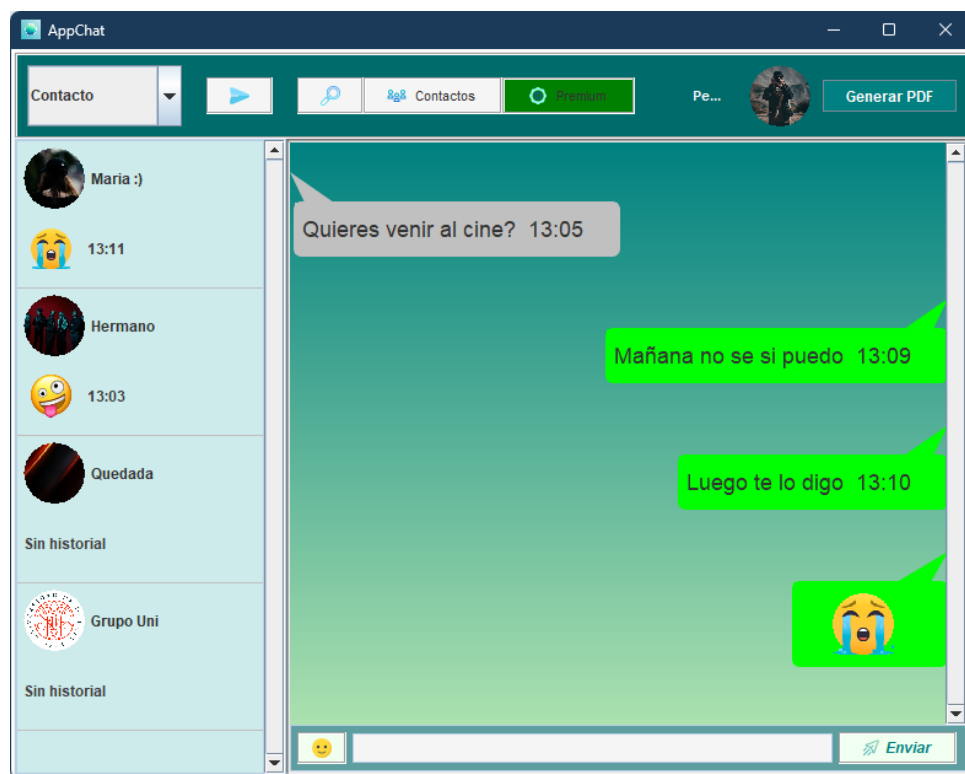


Figura 21: Nuevo grupo agregado

Todos los mensajes que envíe por el grupo, los recibirá por privado cada miembro del grupo. Es decir, funcionaría como una lista de difusión en WhatsApp.

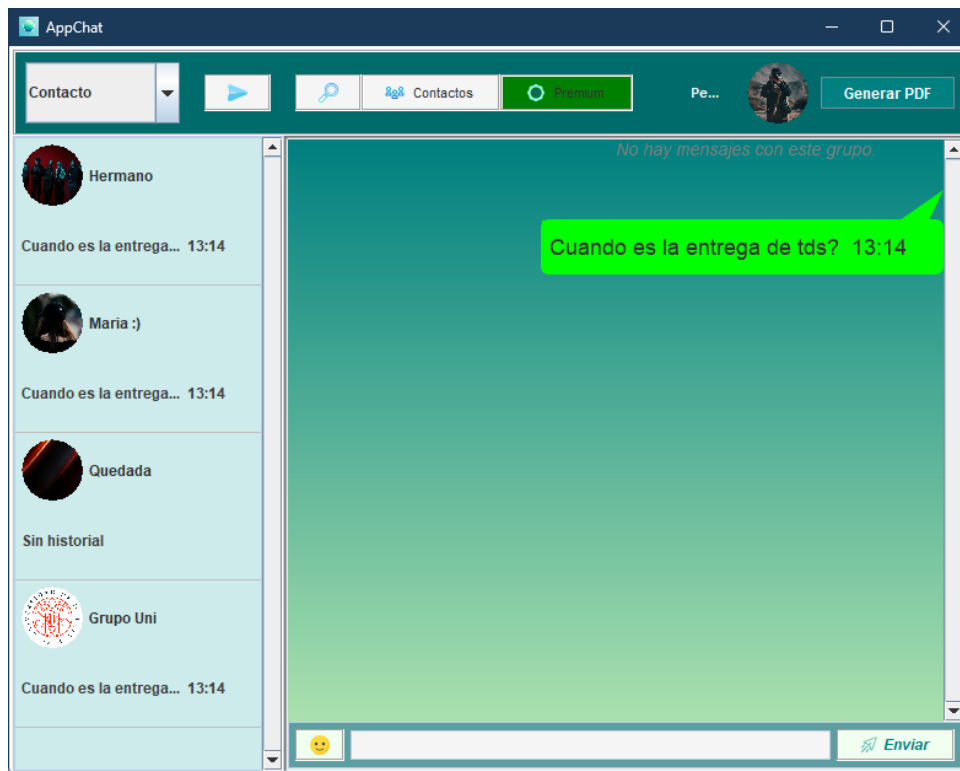


Figura 22: Envío de mensajes por el grupo

7.6 Gestión del perfil

Si pulsamos nuestra foto de perfil de la barra superior, nos aparecerá una ventana de diálogo con distintas opciones, como “**Cambiar foto de perfil**” y “**Ver perfil**”.

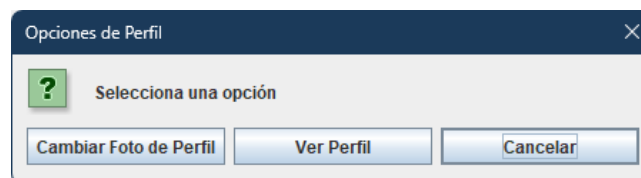


Figura 23: Opciones de gestión del perfil

7.6.1 Cambiar imagen

Si el usuario seleccionar “**Cambiar imagen de perfil**”, podrá cambiar su imagen de perfil de la misma manera que en el registro. Con la opción de introducir una URL de internet, o una imagen local.

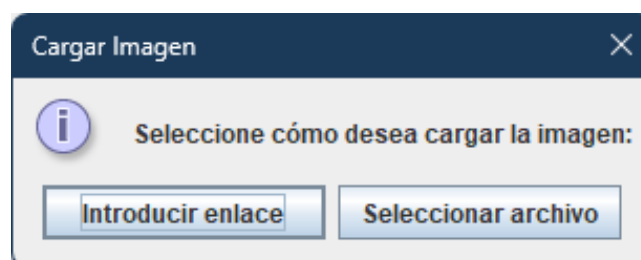


Figura 24: Cambiar imagen de perfil

7.6.2 Cambiar saludo

Por otro lado, si el usuario selecciona “**Ver Perfil**”, se abrirá una ventana con sus datos personales, y donde podrá modificar su estado. Para guardarlo, deberá asegurarse de que utiliza el botón “**Guardar Saludo**”.

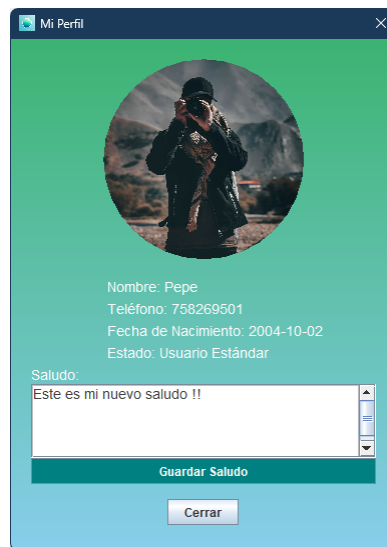


Figura 25: Cambiar saludo del usuario

7.7 Agregar contacto

Existen dos formas de añadir nuevos contactos a la lista.

7.7.1 Opción 1

La primera opción, es usando el botón “**Contactos**” de la barra superior.



Figura 26: Botón 'Añadir Contacto'

Al igual que en la funcionalidad de crear un grupo, aparece la lista con los contactos que el usuario tiene agregado. Si pulsa en el botón de “**Añadir Contacto**”, se le solicitará el nombre y el número del nuevo contacto.

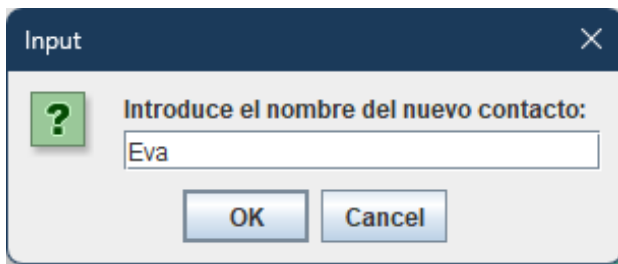


Figura 27: Introducir nombre del nuevo contacto

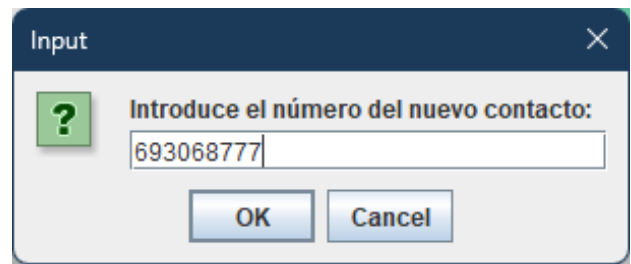


Figura 28: Introducir número del nuevo contacto

Si el usuario existe en el sistema, y no lo tiene agregado de antes, recibirá un mensaje de confirmación.

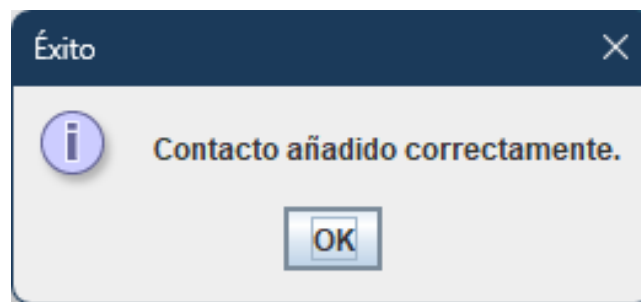


Figura 29: Añadir contacto con éxito

7.7.2 Opción 2

La segunda opción es empleando el botón “+”, cuando se esté conversando con un número no guardado. Tal y como se muestra en la Figura [30].

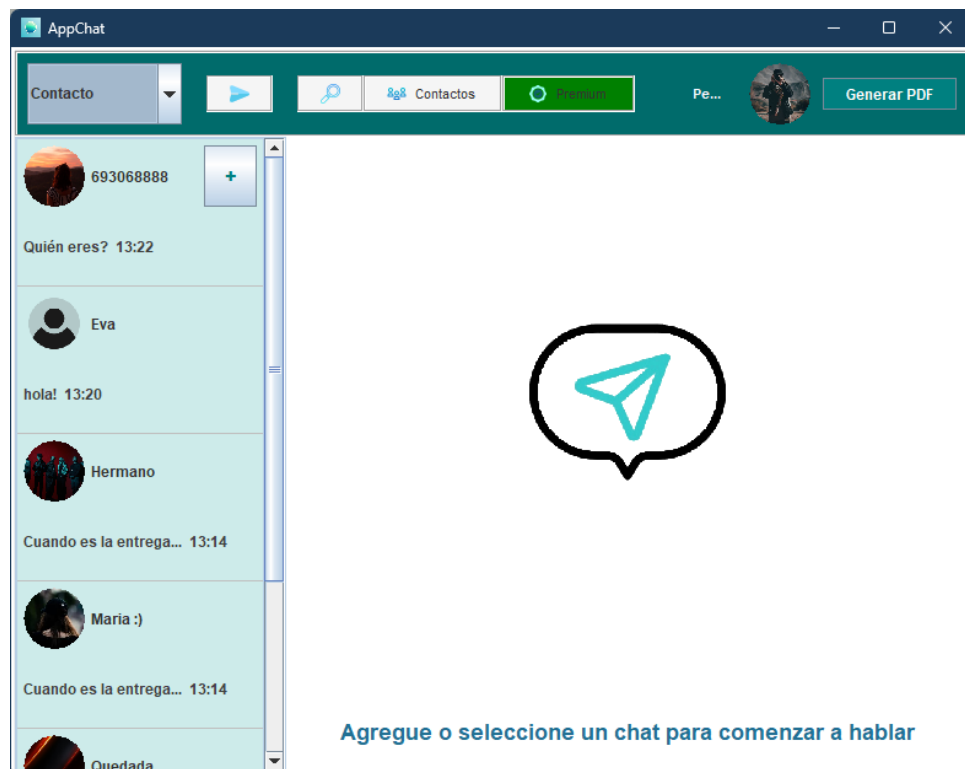


Figura 30: Usar botón '+'

Cuando se hace click en dicho botón, aparece una ventana para poder agregarlo con un nombre. De esta manera, el contacto queda registrado en la lista de contactos.

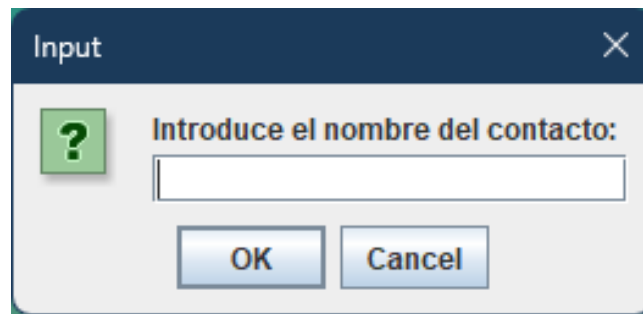


Figura 31: Agregar contacto

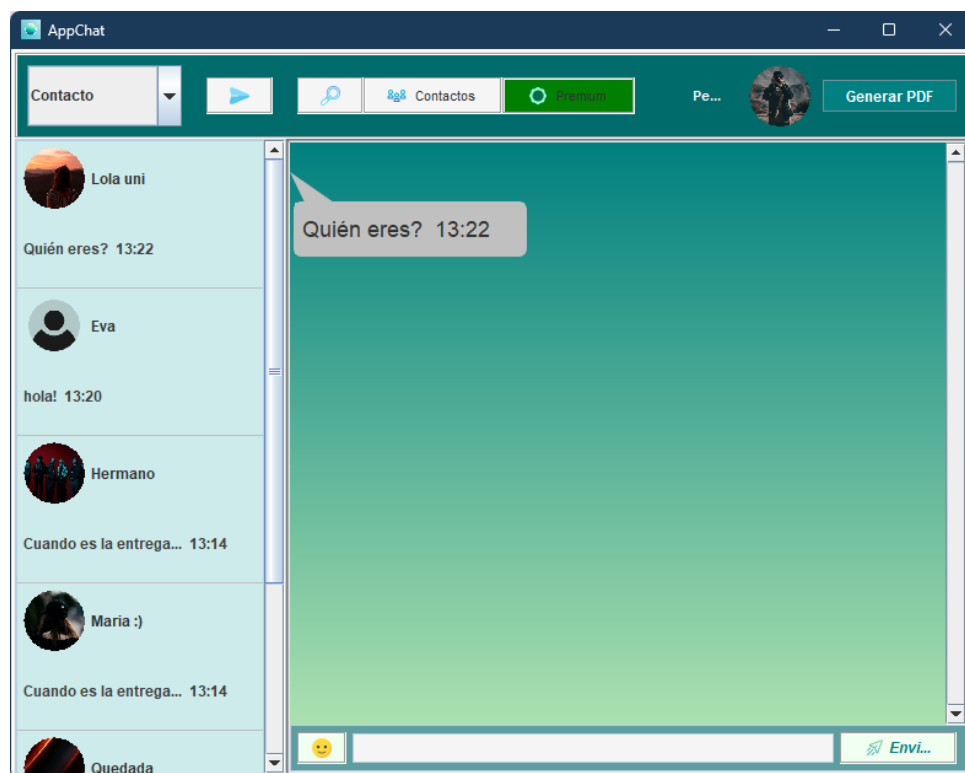


Figura 32: Nuevo contacto agregado

7.8 Premium y descuentos

Para acceder a funcionalidades avanzadas, se puede convertir a un usuario premium. Para ello, en la ventana principal podrá encontrar un botón “**Premium**” situado en la barra superior. Al pulsarlo, se le abrirá una ventana como la Figura [33].



Figura 33: Ventana para pagar el premium

En esta ventana, se indica el precio, así como los posibles descuentos que se puedan aplicar. Actualmente, solo hay dos tipos:

- Descuento por fecha de registro
- Descuento por número de mensajes enviados

Si el usuario es apto para el descuento, se le verá reflejado en el precio final. Y por último, pulsando el botón de “**Activar Premium**”, se abrirá la ventana de la Figura [34] para solicitarle una confirmación al usuario.

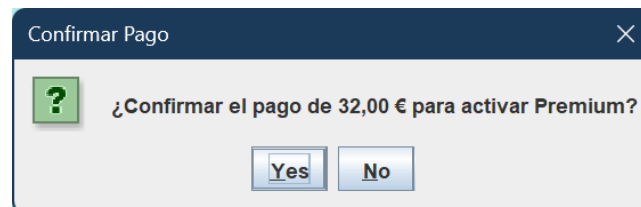


Figura 34: Realizar pago

7.8.1 Conversión a premium

Una vez se haya aceptado la conversión a premium, le aparecerá una ventana indicando el éxito de dicha operación. Asimismo, el botón de “**Premium**” cambia de color, para reflejar dicha funcionalidad.

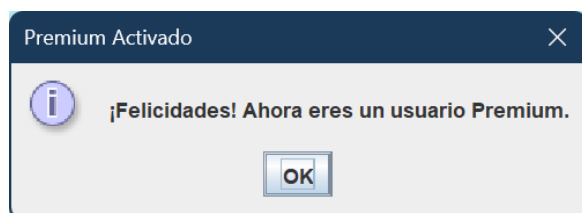


Figura 35: Pago realizado



Figura 36: Usuario Premium activado

7.8.2 Exportar chats a PDF

Para la exportación de chats a PDF, el usuario se debe de haber convertido a Premium. Una vez realizada dicha conversión, se habilitará el uso del botón “**Generar PDF**”.



Figura 37: Botón generar PDF activado

El usuario debe situarse en el chat que quiera exportar, y empleando el botón de “**Generar PDF**”, se le abrirá una ventana como la Figura [38] para que pueda elegir el nombre, y el directorio donde guardar el pdf.

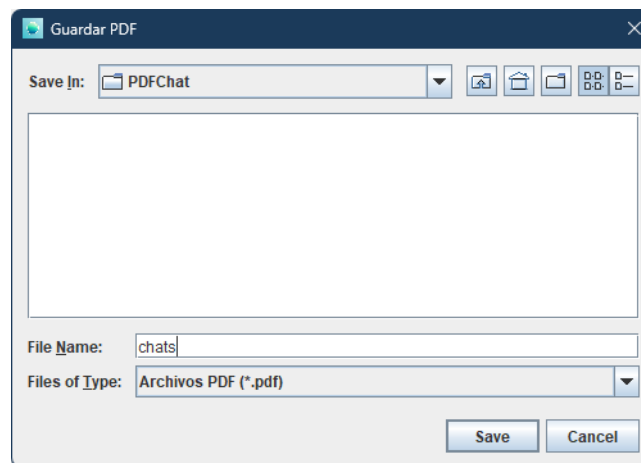


Figura 38: Guardar PDF en el equipo

7.9 Búsqueda de mensajes

AppChat también permite la búsqueda de mensajes específicos en las conversaciones. Si pulsa el botón con icono de lupa de la barra superior, se abrirá una ventana como la Figura [39].



Figura 39: Búsqueda de mensaje

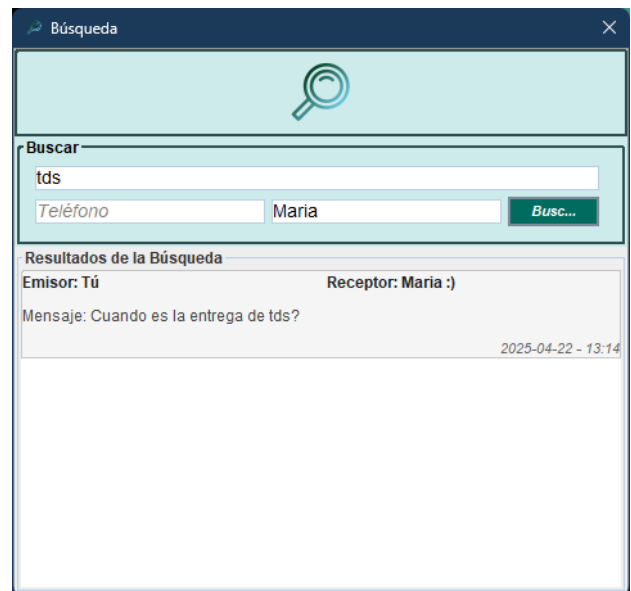


Figura 40: Filtrar por nombre de contacto

En dicha ventana, se pueden apreciar diferentes tipos de filtro, como el texto a buscar, el teléfono del contacto o el nombre del contacto. Todos estos filtros se pueden usar de manera combinada, como en la Figura [40].

Una vez puestos los filtros correspondientes, al pulsar el botón verde de “**Buscar**”, aparecerán abajo los resultados de dicha búsqueda.

Para acceder a dicho mensaje, basta con pulsar dos veces el mensaje correspondiente a buscar, y automáticamente se abrirá el chat donde se ubica dicho mensaje. Asimismo, el scroll se ajusta a la altura correspondiente del mensaje, y se marca con colores distintos para saber que era el buscado.

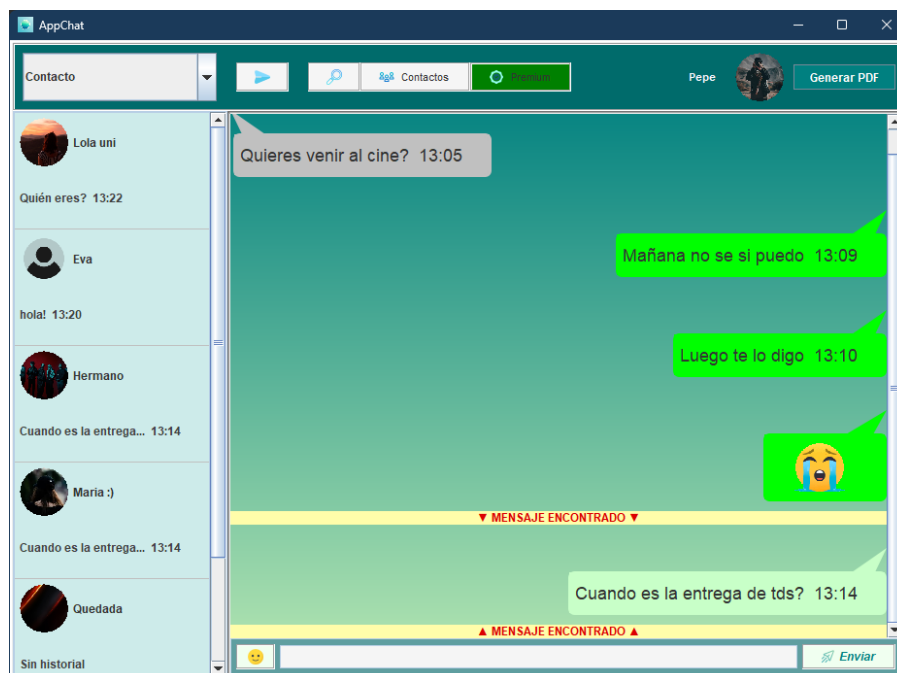


Figura 41: Mensaje buscado en el chat

8 Tiempo dedicado

Componente/Funcionalidad	Eva Pina	Alejandro Oquendo	Total Horas
Interfaces de Usuario			
Ventana Login/Registro	6–8	1–2	7–10
Ventana Principal	8–10	2–3	10–13
Ventana Búsqueda	5–6	1	6–7
Ventana Contactos/Grupos	2–3	5–6	7–9
Ventana Premium/Perfil	4–5	3–4	7–9
Modelo de Datos			
Usuarios y Contactos	3–4	2	5–6
Chat y Mensajes	3–4	1–2	4–6
Grupos	1	4–5	5–6
Patrones de Diseño	4–5	2–3	6–8
Persistencia			
Adaptadores Usuario/Mensaje	4–5	1–2	5–7
Adaptadores Chat	2–3	1	3–4
Adaptadores Grupo	1	3–4	4–5
Controlador			
AppChat (principal)	8–10	1–2	9–12
Funcionalidades			
Envío/Recepción Mensajes	6–7	1–2	7–9
Búsqueda de Mensajes	4–5	1	5–6
Sistema Premium/Descuentos	2–3	4–5	6–8
Exportación a PDF	1	1	2
Otros			
Documentación (Javadoc)	1–2	3–4	4–6
Pruebas y Corrección Bugs	4–5	2–3	6–8
Documentación en LaTeX	6–7	3–4	9–11
TOTAL	75–98	42–58	117–156

Cuadro 1: Distribución de horas por componente

9 Observaciones finales

En primer lugar, realmente pudimos haber entregado el proyecto en la convocatoria de enero, pero decidimos extender el plazo para no entregarlo con prisas y poder revisar posibles fallos, así como hacer la documentación con tranquilidad.

Por otro lado, aunque ya comentamos que la implementación de la clase Chat no era necesaria, decidimos implementarla porque en una aplicación de mensajería real, esta aproximación representa un enfoque más adecuado y profesional.

Respecto a la tecnología empleada, queríamos criticar un poco a Swing. Si bien es útil para aprender, y se pueden hacer cosas chulas si le dedicas tiempo, en nuestra opinión consideramos que deberían enseñarse alternativas más modernas como JavaFX, que ofrecen mayor capacidad de diseño. Con Swing hemos llegado a tener bastantes problemas en cuanto a renderizado, así como ciertas limitaciones, por no decir que de cara a futuro no se suele emplear en entornos profesionales.

En definitiva, esta práctica la hemos disfrutado y sufrido a la vez. Hemos tenido muchos momentos de dificultad y frustración, sobre todo cuando intentábamos pelear con limitaciones de Swing, o con código que no funcionaba. No obstante, ha sido todo un reto, y estamos bastante satisfechos con lo que hemos conseguido lograr, así como con toda la curva de aprendizaje que nos llevamos.