Justificaciones de Diseño (2.B)

En la clase Mensaje:

* Se consideró quitar el atributo booleano fueEnviado dado que esa información ya se puede obtener si la fechaYHoraFueEnviado es NULL, pero se decidió dejarlo por simplicidad. Lo mismo para fueRecibido y fueLeido.
* Se agregaron los atributos fechaYHoraFueRegistradoEnElServidorCentral y fueRegistradoEnElServidorCentral para mayor trazabilidad.

Se modeló la clase Archivo como clase concreta para evitar el code smell de obsesión primitiva.

Se modeló la clase Extension como clase concreta para que sea extensible en tiempo de ejecución.

Para permitir que un usuario pueda ver todos los mensajes anteriores, se creó el método obtenerMensajesAnteriores(Persona) que recorre la lista de mensajesEnviados y retorna una lista de mensajes que la Persona ya había leído previamente.

Para el envío de mensajes, decidí utilizar el patrón Adapter para bajar el acoplamiento con las clases que se encarguen de la implementación real del envío de mensajes. De esta manera, si en un futuro quedasen deprecadas, serían fácilmente reemplazables por otras clases que implementen la interface IAdapterPersona o IAdapterBot. Además, el patrón nos brinda:

* Mayor mantenibilidad debido al nulo acoplamiento entre el Cliente y las clases con la implementación real del envío de mensajes.
* Mayor cohesión de la clase Cliente debido a la delegación de cierto comportamiento en el adapter.
* Mayor facilidad de testeo ya que se podría implementar un mock del AdapterConcreto.

También soluciona/evita de forma directa los siguientes code smells:

* Herencia rechazada.
* Clases alternativas con diferentes interfaces.
* Clase de biblioteca incompleta.

Además, el Notificador implementa una interface INotificador para que, en caso de quedar deprecada o por simple preferencia, se pueda cambiar en tiempo de ejecución por otra clase que implemente la misma interface. De esta manera, se logra bajar el acoplamiento entre el cliente y las clases que implementen la interface INotificador.

También se siguió el principio de segregación de interfaces dado que se crearon 2 interfaces cliente específicas (IAdapterPersona y IAdapterBot) en lugar de una sola interface de propósito general.

Para que un usuario pueda salir de un grupo, se creó el método quitarUsuario(Persona). Es reusable dado que este mismo método es el que se utiliza cuando un administrador desea eliminar un usuario.

Se modeló la clase Bot para que pueda ser extensible en tiempo de ejecución.

Modelo de Dominio (Punto 2.C)

Para crear un bot, simplemente hay que llamar al método estático crearBot() de la clase Bot. El bot ya viene con comandos predeterminados pero sus acciones son personalizables por los usuarios.

Modelo de Dominio (Punto 2.D)

>> Bot

public void procesarTexto(String texto){

String primeraPalabra = texto.obtenerPrimeraPalabra();

List<Parametro> parametros = texto.obtenerParametros();

Comando comando = this.comandos.find(c -> c.obtenerNombre() == primeraPalabra);

comando.ejecutarAcciones(parametros);

}

>> Comando

public String ejecutarAcciones(List<Parametro> parametros){

if(this.parametros.containsAll(parametros)){

return this.acciones.forEach(a -> a.ejecutar());

}

else{

return “los parametros no son correctos”;

}

}

Arquitectura (Punto 1.1)

+mostrarMenuPrincipal(String JSON): String

Arquitectura (Punto 1.2)

La alternativa 2 es menos mantenible dado que se tiene una única base de datos en lugar de 2 bases de datos específicas. Además, se agrega el Front-End Render Server.

Arquitectura (Punto 1.3)

Performance: representa el desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones.

En la alternativa 1, ayuda el hecho de tener 2 bases de datos de uso específico en lugar de una sola de uso general. Por este motivo, todas las consultas con respecto al análisis de los datos se pueden realizar de manera aislada sin afectar todos los otros procesamientos. En la alternativa 2, esto no ocurre dado que todo el procesamiento se realiza en una única base de datos.