Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Bot para el manejo de Checklists en Telegram

Autor:
Dª. Rebeca Rodríguez Feria

Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Bot para el manejo de Checklists en Telegram

Autor:
Dª. Rebeca Rodríguez Feria
Tutor:
D. César Llamas Bello

Resumen

Telegram es un servicio de mensajería instantánea *open source* muy versátil. Permite al usuario personalizarlo desarrollando él mismo distintas funcionalidades. Estas nuevas opciones pueden ser compartidas con el resto de usuarios. Gracias a esto, existen infinidad de nuevas posibilidades con las que mejorar la aplicación y adaptarla a las necesidades del propio usuario.

En el presente trabajo de fin de grado se documenta el diseño y la implementación de un "*bot*" capaz de dar una solución a la gestión de listas de comprobación (*checklists*). Esta aplicación permite la creación de listas, compartición de tareas entre usuarios, marcado de los elementos según se completan, asignación de tareas a usuarios concretos y el seguimiento simultáneo y en tiempo real del estado de finalización de las tareas de la lista. La aplicación dispone de una interfaz de usuario sencilla e intuitiva, y se ha desarrollado utilizando el lenguaje Python.

Abstract

Telegram is a very versatile open source messaging service. The user can personalize Telegram developing himself multiple functionalities. These new options can be shared with every user. Because of this, there are a big amount of new possibilities to improve the application and adapt it to users' needs.

This final grade work explains a bot's design and develop. This bot solves checklists' management in real time. The application's users can create lists, share tasks between groups, check elements of the list when it's completed, assign task to someone. These actions can be followed among few users at the same time. The application has an easy and intuitive user interface. It has been developed by using Python programming language.

Contenido

Capítulo 1. Presentación del proyecto

Resumen del proyecto

Propósito y alcance

La idea de desarrollar un bot para Telegram de checklists surge de la utilidad de este en multitud de chats. Con este bot se busca facilitar la tarea de organizarse dentro de un grupo para conseguir un objetivo común.

Por ejemplo, en un equipo de albañiles uno de ellos va a encargarse de poner a punto todas las salidas de luz de una casa. Para ello, es necesario que el suministro de electricidad de la casa se encuentre desconectado. A través del bot, el usuario puede solicitar que el responsable desconecte la electricidad añadiendo una nueva tarea y asignándosela al susodicho. De esta forma justo en el momento en el que la electricidad se encuentre apagada, la tarea se mostrará como hecha y el trabajador que estaba encargado de las tomas de luz podrá hacer su trabajo.

Un ejemplo más cotidiano es el uso en chats de grupo del hogar (familias o compañeros de piso). Este bot es muy útil para llevar un seguimiento real de la consecución de los productos pertenecientes a la lista de la compra. Si una de las personas va a un supermercado donde uno de los productos está de oferta puede comprarlo y tacharlo como adquirido en el bot de Telegram, así todos los miembros sabrán al instante que ya no es necesario comprar ese artículo.

También es de gran utilidad dentro de grupos creados con el objetivo de organizar algún evento. Un ejemplo de esto sería la organización de un cumpleaños o una boda. Con este bot el encargado de la organización puede crear una lista y compartirla con el resto. Cada miembro podría responsabilizarse de cierta/s tarea/s y marcar en tiempo real el momento en el que la/s finalice. El resto sabrá que el usuario la ha llevado a cabo.

Por otra parte, un usuario puede usar el bot en una conversación consigo mismo para crear *To-Do Lists*. Estas listas se realizan para llevar a cabo un seguimiento de las tareas que uno mismo tiene por hacer. Permiten a una persona saber las cosas que le faltan por hacer aún y las que ya ha hecho. Este tipo de listas se han vuelto virales recientemente y existen diversas aplicaciones en Apple Store y Play Store únicamente para este fin. Con el *CkeckListBot* podemos cubrir todas estas situaciones sin necesidad de instalar más apps.

Suposiciones y restricciones

El equipo consta de dos miembros: el jefe de equipo y un analista-desarrollador. El analista-desarrollador debe cumplir con los requisitos impuestos por el jefe y seguir sus indicaciones para conseguir un producto final de calidad. La aplicación debe estar finalizada el 28 de junio de 2017. Suponemos que los miembros del equipo estarán disponibles durante todo el tiempo que dure el proyecto.

Entregables del proyecto

Se realizarán 6 grandes entregas que coinciden con la finalización de la fase de investigación sobre Telegram, la fase de planificación del proyecto, la fase de análisis, la fase de diseño, la finalización de la implementación de la aplicación y la consecución del proyecto completo. Estos entregables serán revisados y aprobados por el jefe de proyecto. Cada una de estas entregas coincide con un hito.

Hito	Fecha de entrega	Descripción
1	17 de febrero de 2017	Entrega: Documento de investigación acerca de Telegram. Contexto del proyecto a realizar
2	5 de marzo de 2017	Entrega: Documento de planificación
3	26 de marzo de 2017	Entrega: Documento de Análisis
4	23 de abril de 2017	Entrega: Documento de Diseño
5	4 de junio de 2017	Entrega: Código fuente aplicación
6	18 de junio de 2017	Entrega: - Mejoras propuestas y conclusiones - Batería de pruebas - Documentación del código - Manual de usuario

A mayores de estas 6 entregas existirá una séptima, la cual será la entrega final, en la cual se maquetan todos los documentos y se preparan junto con el código para su entrega definitiva. Para esta última entrega, a mayores de todo lo entregado anteriormente se entregará una presentación del proyecto.

Planes de procesos técnicos

Modelo de proceso

El modelo de proceso elegido para el desarrollo del trabajo ha sido *Proceso Unificado (UP)* [1]. Se ha tomado esta decisión por las ventajas de este modelo de proceso. Sus características principales son:

- Dirigido a los casos de uso
- Centrado en la arquitectura
- Es iterativo e incremental

Una ventaja del proceso unificado es que al finalizar cada iteración disponemos de un artefacto el cual podemos enseñar al cliente. El UP se divide en cuatro etapas:

- **Inicio/Concepción**: En esta fase se debe:

    - Establecer los objetivos y los límites del proyecto.
    - Identificar los casos de uso más importantes del sistema.
    - Estimar el costo global y la planificación de todo el proyecto.
    - Especificar los riesgos del proyecto.
- **Elaboración**: Durante esta etapa los objetivos son:
    - Garantizar que la arquitectura, los requisitos y los planes son lo bastante estables.
    - Obtener una visión refinada del proyecto a realizar.
    - Producir un prototipo del proyecto.
- **Construcción**: Los objetivos en esta fase son:
    - Minimizar los costos de desarrollo optimizando los recursos e impidiendo las reconstrucciones y los fragmentos innecesarios.
    - Conseguir una calidad adecuada de forma rápida y práctica.
    - Conseguir versiones útiles (alfa, beta y otros releases de prueba)
    - Completar el análisis, diseño, desarrollo y prueba de toda la funcionalidad necesaria.
- **Transición**: Durante esta fase se debe:
    - Ejecutar el despliegue.
    - Conseguir que el usuario sea capaz de mantener el producto.
    - Conseguir la aceptación por el usuario (*stakeholder*) que lo entregado es completo y consistente con el criterio de evaluación fijado en la visión inicial del proyecto.
    - Obtener un producto final tan rápido y eficiente respecto al coste como práctico.

Las etapas del UP están subdivididas en iteraciones en las cuales se realiza un "subproyecto" mediante desarrollo en cascada. Dependiendo de la etapa a la cual corresponda la iteración se pondrá más énfasis a una de las fases "tradicionales" (requisitos, análisis, implementación o verificación).

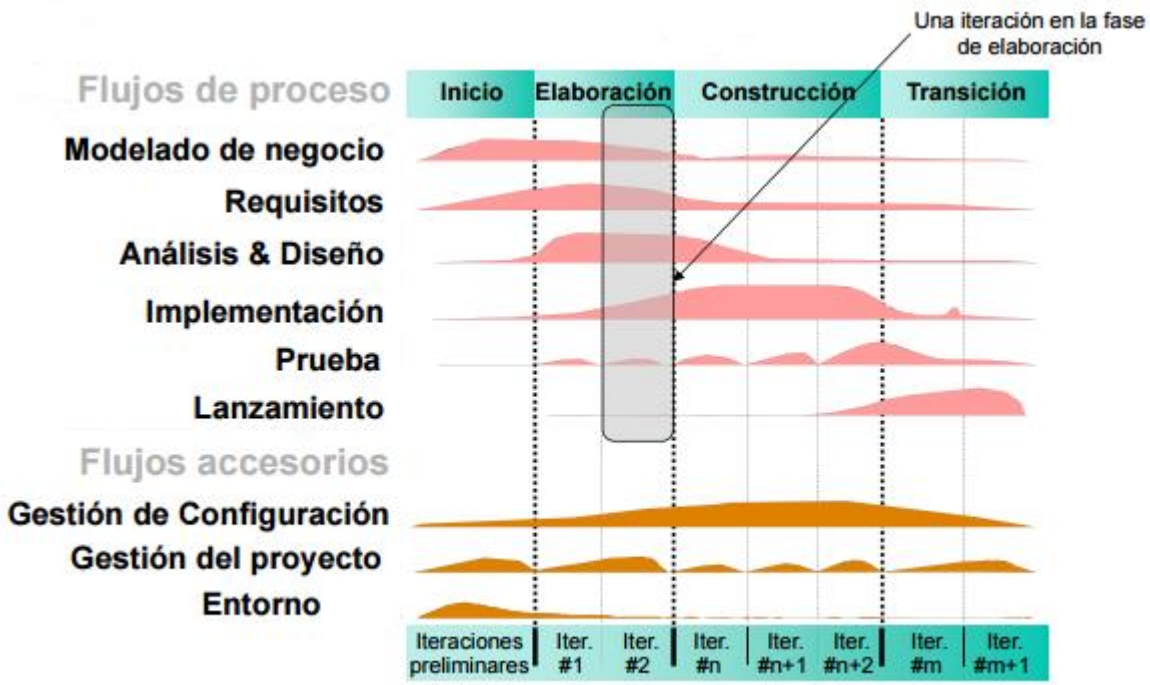

Figura 6. Ejemplo de iteraciones en un proyecto realizado mediante Proceso Unificado.

En la *figura 6* podemos observar un ejemplo del trabajo realizado en cada una de las iteraciones de un proyecto realizado mediante PU. Si nos fijamos en la iteración destacada, la segunda y última dentro

de la fase de elaboración, comprobamos como la mayoría del trabajo realizado pertenece a análisis y diseño. También podemos ver como se comienza a trabajar menos en actividades relacionadas con modelado del negocio y especificación de requisitos y se dedica más tiempo a tareas relacionadas con implementación. En las iteraciones pertenecientes a la etapa de construcción apenas se emplea tiempo en tareas de las fases de modelado de negocio y requisitos, ni tampoco a análisis y diseño a partir de la mitad de la etapa.

Métodos, herramientas y técnicas

El método utilizado durante el desarrollo del proyecto es como ya se ha dicho en el punto anterior: Proceso Unificado. Para organizar mejor las etapas correspondientes a la aplicación de este método se utilizará *Google Calendar*. Se han fijado unos hitos o entregas especificados en el punto **Entregables del proyecto**. El fin de cada etapa no corresponde con estos hitos debido a que, en UP, el cierre de la etapa de elaboración no se corresponde con la finalización del documento de análisis. Esto se debe a que, como se comentó en el punto anterior, todas las tareas del desarrollo en cascada realizan durante cada etapa. Por ejemplo, en la etapa de construcción se realizan tareas de análisis y diseño, aunque sea en menor medida. Se estima que las etapas finalizarán aproximadamente:

- **Etapa de Inicio.** 24 de febrero (Duración aprox. 3 semanas)
- **Etapa de elaboración.** 17 de marzo (Duración aprox. 3 semanas)
- **Etapa de construcción.** 26 de mayo (Duración aprox. 10 semanas)
- **Etapa de transición.** 18 de junio (Duración aprox. 3 semanas)

El entregable final del trabajo ha sido escrito a través del servicio proporcionado por *Microsoft Office 365* debido a su comodidad y la posibilidad de trabajar desde distintos dispositivos a través de la nube.

El desarrollo del código será llevado a cabo a través del entorno de desarrollo *c9.io* debido a las mismas ventajas ofrecidas por Microsoft Office 365. Cloud 9 permite trabajar en el código desde cualquier dispositivo desde el cual accedemos a nuestra cuenta. Utilizo:

- Python como lenguaje de programación. Se ha tomado esta decisión por ser un lenguaje que ya se conoce y permite mucha libertad a la hora de implementar el modelo.

El control de versiones se va a realizar con Bitbucket. Bitbucket está muy bien integrado con c9 lo que facilita el trabajo. Es necesario el uso de un repositorio a pesar de ser un solo miembro en el equipo debido a los posibles cambios constantes en el código. Bitbucket permite realizar un control de versiones sencillo mediante el cual es fácil recuperar una versión anterior funcional. Esta posibilidad es esencial teniendo en cuenta que ciertas modificaciones podrían perjudicar el correcto funcionamiento de otras ya finalizadas.

Organización del proyecto

Plan de iniciación

La formación necesaria para comenzar con las tareas de implementación se ha realizado al final de la etapa de Inicio y principio de la de Elaboración. Esto es así, porque la implementación de la aplicación comienza ya durante la etapa de Elaboración.

Formación impartida	Método	Duración	Fecha	N.º de participantes
Conocimientos básicos Python	Formación online	1 hora	20/02/2017	1
Conocimientos base Python aplicado a Telegram	Formación online	5 horas	23/02/2017 - 24/02/2017	1
Conocimientos avanzados Python para Telegram	Formación online	10 horas	27/02/2017 – 28/02/2017	1

Plan de trabajo

En este punto se presentan todas las actividades necesarias para la consecución del proyecto y su éxito. Existen actividades de duración 0 para marcar los hitos que representan la finalización de cada etapa del desarrollo.

Cada actividad tiene un coste estimado de duración en horas/hombre. Como el proyecto es realizado por un único miembro, los costes estimados no se pueden reducir incrementando los recursos.

Identificador	Actividad	Predecesoras	Coste
1	Reunión inicio proyecto	-	1 hora
2	Planificación general: Estructura trabajo final	1	2 horas
3	Resumen proyecto (español/inglés)	1	1 hora
4	Investigación: ¿Qué es Telegram?	2	7 hora
5	Investigación: Historia aplicación	2	7 hora
6	Investigación: Características y Bots	2	10 horas
7	Investigación: Telegram vs WhatsApp	2	7 horas
8	**Hito:** Entrega documento investigación	3,4,5,6,7	0 horas
9	Resumen del proyecto (Propósito, alcance, suposiciones y restricciones)	8	3 horas
10	Decisión: Entregables del	8	3 horas

	proyecto		
11	Decisión: Modelo de proceso y herramientas	8	5 horas
12	Decisión: Plan de trabajo	10	3 horas
13	Definición de recursos	8	5 horas
14	Plan de administración de riesgos	8	5 horas
15	Otros planes	8	3 horas
16	**Hito:** Entrega documento de planificación	9,11,12,13,14,15	0 horas
17	Especificación requisitos funcionales	16	5 horas
18	Especificación requisitos no funcionales y de información	16	2 horas
19	Diagrama Casos de Uso	17,18	5 horas
20	Especificación de CU	19	10 horas
21	Diagrama de clases	20	15 horas
24	**Hito:** Entrega documento análisis	21	0 horas
25	Diagrama de diseño	24	15 horas
26	Diagrama de actividades	25	10 horas
27	Diagramas de secuencia de CU	25	15 horas
28	Diagrama de despliegue	25	2 horas
29	Prototipo aplicación	26,27,28	5 horas
30	**Hito:** Entrega documento diseño	29	0 horas
31	Pasos iniciales creación Bot	30	1 hora
32	Funcionalidad: Nueva Checklist	31	10 horas
33	Funcionalidad: Añadir Tarea	32	10 horas
34	Funcionalidad: Modificar Tarea	33	7 horas
35	Funcionalidad: Eliminar Tarea	33	7 horas
36	Funcionalidad: Reinicializar Checklist	33	5 horas
37	Funcionalidad: Eliminar Checklist	33	5 horas
38	Funcionalidad: Compartir Checklist	34, 35, 36, 37	7 horas
39	Funcionalidad: Asignar Tarea	38	20 horas
40	Funcionalidad: Clonar Checklist	34,35,36,37	7 horas
41	Control de datos. Persistencia	33	15 horas
42	Funcionalidad: Administración Checklists	39, 41	7 horas
43	Multilenguaje	40,42	7 horas

44	Búsqueda errores y solución de estos. Baterías de pruebas	43	20 horas
45	Limpieza de código	44	5 horas
46	**Hito:** Entrega implementación	45	0 horas
47	Búsqueda de posibles mejoras	46	5 horas
48	Conclusiones	47	5 horas
49	Revisado maquetado documento	48	1 hora
50	**Hito:** Entrega proyecto	49	0 horas
51	Presentación para exposición del trabajo	50	3 horas
52	Preparación defensa	51	2 horas

Recursos

Se han dividido los recursos que se van a utilizar en el proyecto según la división realizada en el libro *Software Project Management* de Hughes y Cotterell y estudiada durante la carrera.

Trabajo

Aquellos recursos relacionados con las personas físicas (miembros del equipo, clientes, *product owner*, etc.)

Nombre del Recurso	Equipo
Descripción	Formado Rebeca Rodríguez Feria, único miembro del equipo. Encargada de planificación, análisis, desarrollo y pruebas del proyecto
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Habilidades técnicas	Conocer el alcance de la aplicación para poder llevarla a cabo, con las herramientas y técnicas adecuadas

Equipamiento

Todo el material informático e infraestructura necesaria para el desarrollo de la aplicación

Nombre del Recurso	Computadora
Descripción	Ordenador utilizado para el desarrollo completo del proyecto
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable final del proyecto

Nombre del Recurso	Móvil y Tablet
Descripción	Dispositivo utilizado como apoyo para el desarrollo de la aplicación. Necesario para comprobar el correcto funcionamiento del bot en Telegram
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Fecha de comienzo de uso	23/02/2016
Tiempo durante el que se precisa	Desde el comienzo de la implementación del bot hasta el entregable final del proyecto

Materiales

Cualquier documento o utensilio no tecnológico necesario para el desarrollo del proyecto

Nombre del Recurso	Documentación online
Descripción	Páginas web tales como blogs, repositorios y foros online
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable final del proyecto

Nombre del Recurso	Python API
Descripción	Documentación necesaria para el desarrollo de aplicaciones en lenguaje Python.
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto
Fecha de comienzo de uso	23/02/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto

Nombre del Recurso	Telepot API
Descripción	Documentación del framework utilizado para el desarrollo del bots. Necesaria para entender el funcionamiento de las llamadas a servidor.
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto
Fecha de comienzo de uso	23/02/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto

Nombre del Recurso	Telegram Bot API
Descripción	Documentación para el desarrollo de bots para Telegram. Contiene todos los cambios actualizados.
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto
Fecha de comienzo de uso	23/02/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto

Espacio

Son los lugares facilitados para el trabajo de los empleados

Nombre del Recurso	Estudio
Descripción	Habitación aislada y sin posibles distracciones utilizada durante todo el desarrollo del proyecto
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable final del proyecto
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable final del proyecto

Servicios

Las aplicaciones informáticas utilizadas para la realización del proyecto.

Nombre del Recurso	Red de Internet
Descripción	Red necesaria para la investigación y desarrollo de la aplicación
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo.
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo.

Nombre del Recurso	Cloud9
Descripción	Entorno de desarrollo de software online
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.
Fecha de comienzo de uso	23/02/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.

Nombre del Recurso	Microsoft Office 365
Descripción	Servicio de creación, edición, lectura y alojamiento de archivos. Permite el acceso a los archivos desde diferentes dispositivos.
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo

Nombre del Recurso	Google Calendar
Descripción	Calendario online que realiza la función de agenda y permite organizar el proyecto.
Informe sobre su disponibilidad	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo de la primera actividad hasta el entregable del plan de desarrollo

Nombre del Recurso	Bitbucket
Descripción	Herramienta utilizada para el control de versiones. Funciona muy bien en conjunción con C9.
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.
Fecha de comienzo de uso	23/02/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.

Tiempo

Nombre del Recurso	Tiempo
Descripción	Periodo transcurrido para la realización de todas las tareas necesarias para la consecución de los hitos y finalmente del proyecto
Informe sobre su disponibilidad	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.
Fecha de comienzo de uso	30/01/2017
Tiempo durante el que se precisa	Desde el comienzo del desarrollo de Software hasta el entregable final del proyecto.

Plan de control

Al finalizar cada etapa del desarrollo del proyecto se procederá a una revisión en detalle del trabajo realizada durante esta. Después de este primer control se pasará a una segunda revisión general en coordinación con el tutor del proyecto (jefe del proyecto).

El objetivo de la primera revisión es de encontrar errores y subsanarlos, corregir errores sintácticos y gramaticales en el documento y dejarlo bien maquetado. En la revisión de la etapa de implementación también se deberán buscar errores que puedan producirse durante la ejecución del código en medio de un uso normal de la aplicación.

La finalidad de la revisión general es la de informar del estado y los avances del proyecto al jefe. Es interesante atender las distintas propuestas e ideas del jefe para mejorar tanto la aplicación como el documento que la acompaña. A cualquier propuesta de cambio del trabajo por parte del jefe se le dará solución lo antes posible y se concertará una reunión extraordinaria. Durante esta revisión extraordinaria se tiene como propósito principal el mostrar el cambio implementado y las modificaciones necesarias para llevarlo a cabo. Esta reunión extraordinaria puede ser sustituida por comunicación vía mail si ambas partes lo consideran adecuado.

Plan de administración de riesgos

Dentro del contexto de desarrollo de Software un riesgo es un percance que consume tiempo y requiere de una solución rápida para que no nos impida cumplir con los plazos establecidos para cada actividad.

A continuación, se detallan los posibles riesgos del desarrollo de este proyecto y un plan de actuación ante cada uno de ellos:

1	**Título: Fallo de personal**	
Fase: Todas	Probabilidad: Baja	Categoría: Proyecto
Descripción: el miembro del equipo, Rebeca Rodríguez Feria, no se encuentra disponible durante un período de tiempo para realizar una tarea que le ha sido asignada.		
Consecuencias: Una tarea no es realizada a tiempo o incluso no se realiza por falta de personal.		
Contexto: Situación que se puede dar en cualquier fase del proyecto por motivos inherentes al mismo.		
Análisis: Puede provocar un alargamiento de la tarea y, por lo tanto, el incumplimiento del plazo de entrega.		
Plan de acción: Reorganización del trabajo asignando más horas de trabajo al día hasta alcanzar con los plazos establecidos.		

2	Título: No disponibilidad de equipo informático	
Fase: Todas	Probabilidad: Baja	Categoría: Proyecto
Descripción: El equipo informático no se encuentra disponible durante un período de tiempo por razones técnicas por lo que no se puede realizar una tarea.		
Consecuencias: Una tarea no es realizada a tiempo o incluso no se realiza.		
Contexto: Situación que se puede dar en cualquier fase del proyecto por motivos inherentes al mismo.		
Análisis: Puede provocar un alargamiento de la tarea y, por lo tanto, el incumplimiento del plazo de entrega.		
Plan de acción: Arreglo del equipo informático y/o intento de disposición de otro equipo y reorganización del trabajo asignando más horas.		

3	Título: Falta de coordinación del equipo con el jefe del proyecto	
Fase: Todas	Probabilidad: Baja	Categoría: Proyecto
Descripción: El equipo, Rebeca Rodríguez Feria, no consigue concertar una cita con el jefe, César Llamas, por lo que no se puede avanzar correctamente con el proyecto.		
Consecuencias: Una tarea no comienza a realizarse a tiempo con lo que todas sus sucesoras se retrasan también.		
Contexto: Situación que se puede dar en cualquier fase del proyecto por motivos propios de la organización y del jefe del proyecto.		
Análisis: Provoca el incumplimiento de los plazos de entrega.		
Plan de acción: Breve reunión informal o revisión por correo electrónico para obtener un primer visto bueno del jefe y poder continuar con el proyecto hasta el momento de la reunión general.		

4	Título: Falta de revisiones efectivas	
Fase: Desarrollo y testing	Probabilidad: Media	Categoría: Proceso
Descripción: El equipo no realiza las revisiones periódicas acordadas a lo largo de todo el proyecto.		
Consecuencias: Mal funcionamiento de la aplicación final y realización de actividades de baja calidad.		
Contexto: Situación que puede darse en la fase de desarrollo debido a una mala organización y una mala y escasa realización de casos de prueba.		
Análisis: Provoca la producción de software de mala calidad.		
Plan de acción: El equipo de desarrollo debe estar concienciado de la necesidad de revisar cada una de las tareas una vez acabadas, aun cuando crea que la implementación no tiene ningún fallo/error.		

5	Título: Secuencia continuada de cambios en la idea inicial	
Fase: Todas	Probabilidad: Media	Categoría:
Descripción: El equipo cambia algún o algunos de los requisitos establecidos inicialmente.		
Consecuencias: Afecta a las tareas relacionadas con los requisitos que han sido modificados.		
Contexto: Puede ocurrir en cualquier fase del proyecto debido a una idea mejor sobre la implementación de la aplicación		
Análisis: Tiene un fuerte impacto ya que puede implicar cambios en las distintas etapas (análisis, diseño e implementación)		
Plan de acción: El equipo debe realizar una reunión extraordinaria con el jefe para informarle del cambio que planea realizar y obtener su aprobación tras el análisis del impacto de llevar a cabo la modificación.		

6	Título: No cumplimiento de entrega de un hito	
Fase: Todas	Probabilidad: Media	Categoría: Proceso
Descripción: No se consiguen los objetivos planteados en la fecha de entrega del hito.		
Consecuencias: Afecta al plazo de entrega del hito actual y siguientes.		
Contexto: Situación que puede darse en cualquier fase del proyecto y se debe a una mala planificación del mismo.		
Análisis: Produce el retraso de la entrega actual y el de los hitos siguientes, por lo que se produce una importante prolongación del tiempo del proyecto.		
Plan de acción: Replanificación del siguiente hito para no retrasar el proyecto completo.		

Plan de costes

El proyecto será llevado a cabo por un único miembro. Rebeca Rodríguez Feria es la analista-desarrolladora del proyecto completo, estudiante de Ingeniería Informática (mención de Ingeniería de Software).

Se supone un sueldo de 7€/hora para un programador junior (1.120€ al mes). Teniendo en cuenta que se ha realizado una media de 4 horas al día, se ha trabajado de lunes a viernes para la consecución del proyecto y se ha estimado el proyecto en 19 semanas, obtenemos un coste por mano de obra de:

4 horas x 7€ x 95 días = 2.660€

Plan de funcionamiento

Para poner en funcionamiento esta aplicación se ha usado el servidor gratuito proporcionado por c9.io durante la etapa de construcción. Al finalizar esta etapa, y con el código finalizado, se ha utilizado una máquina virtual Unix con sistema operativo Ubuntu para mantener corriendo el bot indefinidamente. Esta máquina ha sido proporcionada, de forma gratuita, por la universidad de Valladolid. El servicio prestado por la UVa finaliza en julio de 2017, por esta razón se ha realizado el correspondiente plan de funcionamiento. Cabe destacar que este plan ha sido estudiado y redactado una vez finalizada la etapa de construcción para contar con datos realistas con los que realizar una estimación de presupuesto anual [2].

Si se quisiera mantener la aplicación operativa más allá de julio de 2017, sería necesario un servidor. Teniendo en cuenta el tipo de aplicación desarrollada y la escalabilidad que podría tener, las opciones más coherentes son:

Servidor en la nube.

Es un servidor en Internet encargado de resolver las peticiones que recibe de los distintos usuarios. Este tipo servidor es una abstracción de un servidor físico. Está compuesto por una red de servidores físicos que dan resolución a las diferentes tareas que el servidor abstracto recibe. La deslocalización del servidor en varios permite algunas ventajas. Por ejemplo, si uno de los servidores se cae, otro puede reemplazarle y realizar la tarea que había sido encomendada al primero. Es infinitamente escalable. Siempre se pueden añadir nuevos servidores físicos para ampliar la "nube". Por esta razón los clientes pueden ampliar o reducir los planes contratados fácilmente. Además, al estar abstraído del hardware, es susceptible a fallos de hardware. Por esta razón, permite que nunca haya cortes en el suministro del servicio, ya que si un nodo de la red deja de funcionar se pasa al siguiente.

El primer servidor en la nube aparece en 2006 y es de Amazon. Aunque, hasta 2008 no empezó a extenderse el concepto y su uso. El pago de estos servidores es por tráfico real consumido.

Utilizando la calculadora de presupuesto mensual de Amazon [3], para un servidor en la nube con 2 CPU's, 4 GB de memoria y un rendimiento de operaciones entrada/salida de unos 5MB por segundo. pagaríamos 34,50 dólares. A esta cantidad, hay que sumarle el número de peticiones POST/GET que realizamos al mes.

- **Peticiones POST.** Utilizadas para publicar o subir datos.
- **Peticiones GET.** Utilizadas para obtener datos.

En nuestro caso, el uso de estas peticiones se reduce a las operaciones de contacto con la API Bot de Telegram. Concretamente, las operaciones dentro del *Checklistbot* son: recibir mensaje, enviar y editar mensaje, obtener un miembro de un chat, obtener el número de miembros de un grupo y devolver respuesta a una *callbackquery*. Analizando cada operación, deducimos que todas son operaciones POST a excepción de *glance (*recibir mensaje de usuario)*, getChatMember* y *getChatMembersCount.*

En total, en el código del bot se realizan 75 peticiones. De ellas son:

- 4 llamadas a *glance*
- 20 llamadas a *sendMessage*
- 26 llamadas a *editMessageText*
- 1 llamadas a *getChatMember*
- 1 llamada a *getChatMembersCount*

- 23 llamadas a *answerCallbackQuery.*

En conclusión, solo el 8% del total de las peticiones son GET dentro de la aplicación. Una de estas llamadas es desde la funcionalidad “Asignar tarea” siempre y otra cuando un usuario que la aplicación no tiene almacenado entre sus datos escribe al bot. Las 4 llamadas restantes, correspondientes a la invocación del método *glance,* se realizan cada vez que la aplicación recibe un mensaje de texto, una *callbackquery*, una petición *inline* o una selección de un resultado de una petición *inline*. A priori, podríamos caer en el error de calcular el total a pagar por peticiones aplicando este porcentaje. Sin embargo, es importante tener en cuenta otros factores.

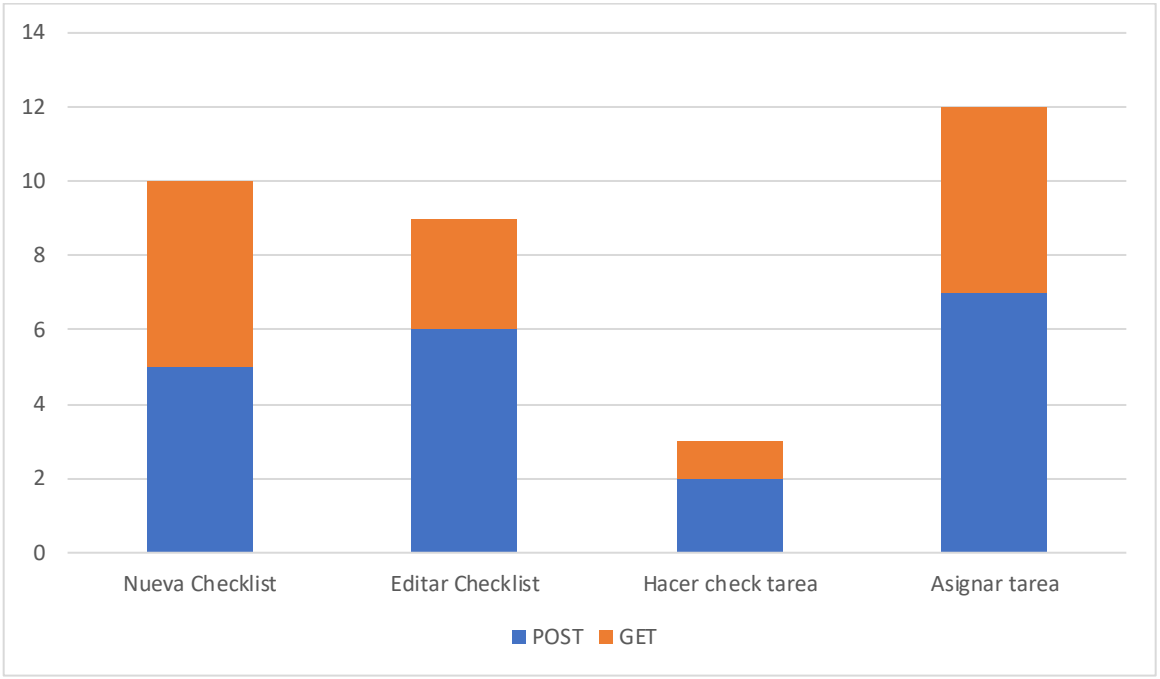

Figura 7. Gráfico de barras: Operaciones POST/GET medias necesarias para realizar algunas de las funcionalidades principales del *checklistbot*.

Observando el gráfico de la *figura 7* podemos ver que, dentro de las funcionalidades principales, las que más peticiones necesitan son “Crear nueva checklist” y “Asignar tarea”. Se ha supuesto que se crea una lista con 3 tareas. Cada tarea a mayores en la creación de una Checklist incrementa directamente el número de peticiones en uno. También se ha supuesto que “Editar Checklist” consume 9 peticiones. Esta cantidad se obtiene como una media de las peticiones que requieren las 5 funcionalidades que engloba (“Añadir Tarea”, “Modificar Tarea”, “Eliminar Tarea”, “Reinicializar Checklist” y “Eliminar Checklist”). Esta cantidad también se incrementaría si no se sigue la secuencia normal descrita en los casos de uso detallados en el Capítulo de **Análisis**.

Las peticiones GET corresponden a la invocación al método *glance* para recuperar el mensaje o *callbackquery* enviado por el usuario. En “Asignar Tarea” una de las peticiones GET corresponde al método *getChatMembersCount.* Las peticiones POST se incrementan bastante cuando se trabaja con *callbackqueries.* Por cada vez que un usuario pulsa un botón, se cuentan dos peticiones POST (*sendMessage/editMessage* y *answerCallbackQuery).* La llamada al método *answerCallbackQuery* es necesaria para que el bot no se mantenga a la espera de la respuesta a la petición por tiempo indefinido. Si no se le llama al dar respuesta a la petición, el botón queda marcado con una “*circle progress bar”.*

Teniendo los datos del gráfico en cuenta y suponiendo que, de media, un usuario crea una lista al día, realiza la acción de editar listas 5 veces al día, hace *check* en tareas unas 15 veces y asigna 3 tareas al día. Obtenemos un total de:

5 POST + 5 GET + 5 x (6 POST + 3 GET) + 15 x (2 POST + 1 GET) + 3 x (7 POST + 5 GET) = 86 POST + 50 GET

Por último, suponiendo en sus primeros meses un uso del bot por parte de 100 usuarios mensuales, obtenemos:

100 usuarios x 30 días x (86 POST + 50 GET) = 258.000 POST + 150.000 GET

Los precios de Amazon por este tipo de peticiones son: 0.005 dólares por cada 1 000 solicitudes PUT, COPY o POST y 0.004 dólares por cada 10 000 solicitudes GET.

Finalmente, el precio total que tendríamos que pagar mensualmente para mantener nuestro bot en un servidor Cloud de Amazon es de 31,94 euros (35,79 dólares). Suponiendo un incremento de usuarios hasta llegar a 1.000 mensuales, el precio total a pagar sería de 42,84 euros (48 dólares). Con 10.0000 usuarios mensuales pagaríamos 151,28 euros (169,50 dólares). En estas tres estimaciones se ha mantenido el servidor seleccionado inicialmente. El incremento del precio se debe única y exclusivamente al incremento de las peticiones. Si mejoramos el servidor contratado inicialmente, el precio aumentaría.

Servidor Virtual privado (VPS)

Aparecen en 2001. Son servidores físicos particionados en varios servidores de tal manera que cada uno de estos servidores funcione como si fuera una máquina independiente. Como cada VPS funciona con su propio sistema operativo, cada usuario en su servidor tiene permisos de *superusuario* y puede instalar lo que desee. Debido a que varios clientes trabajan en el mismo servidor físico, existen algunos problemas de disco, RAM y tiempo de procesamiento.

Los precios mensuales por un servidor virtual privado son fijos al mes, al contrario de los servidores Cloud. Escogiendo como servidor uno lo más similar posible al elegido en Amazon en el apartado anterior, los precios mensuales son los siguientes:

- **DigitalOcean.** 40 dólares/mes. CPU de 2 núcleos, 4 GB de RAM, 60GB de disco duro y 4TB de transferencia [4].
- **Linode.** 20 dólares/mes. CPU de 2 núcleos, 4 GB de RAM, 48GB de disco duro y 3TB de transferencia [5].
- **Ramnode**. 14 dólares/mes. CPU de 2 núcleos, 4 GB de RAM, 80GB de disco duro, 3TB de transferencia [6].
- **Scaleway.** 6 dólares/mes. CPU de 4 núcleos, 4GB de RAM, 100GB de disco duro, 200Mbit/s de transferencia [7].

Como se puede ver los precios varían mucho entre los distintos servicios, pero podríamos fijar una media de 20 euros mensuales. Podríamos mantener estas tarifas en los primeros meses y ampliar el servicio contratado si llegásemos a 10.000 usuarios. Por un plan de 8GB de RAM, alrededor de 100GB de disco duro (200GB en Scaleway) y 4TB de transferencia pagaríamos una media de 40 euros.

Raspberry Pi.

Una Raspberry es un computador de placa reducida. Esto significa que es una computadora completa en solo una placa. El diseño se centra en un sólo microprocesador con la RAM, conectores

de E/S y todas las demás características de un computador funcional en una sola tarjeta que suele ser de tamaño reducido [8].

Su software es open source, por lo que cualquiera puede ver su código y programar nuevo código. Su sistema operativo oficial es una versión adaptada de Debian llamada Raspbian.

Podemos utilizar una Raspberry como servidor físico en el cual mantener el bot desarrollado corriendo. Las características de la nueva Raspberry Pi 3 son:

- Cpu, **ARM Cortex A53,** de cuatro núcleos a 1.2GHz de 64 bits
- 1GB de RAM
- La capacidad del disco duro corresponde al tamaño de la memoria SD elegida por el usuario (hasta 128GB).
- La velocidad de transferencia dependerá de la red a la que conectemos la Raspberry.

El precio total de nuestro "servidor" será de 93 euros. Una Raspberry pi 3 cuesta alrededor de 40 euros y una tarjeta SD SanDisk Extreme Pro 64GB son 53 euros. Este dinero sería lo único que tendríamos que pagar sin contar la conexión a Internet. Para poder utilizar nuestro bot desde otros dispositivos y conectar con la aplicación desarrollada, la Raspberry debe estar conectada a Internet. Para ello Raspberry Pi 3 dispone de un conector Ethernet y conexión Wifi. Suponiendo que contratamos Fibra de 50Mb, los costes de Internet nos supondrían 15 euros al mes.

Conclusiones

Según la opción elegida, para mantener en funcionamiento el bot se pagará durante el primer año:

- Servidor Cloud: 383,28 euros.
- Servidor Virtual Privado: 240 euros
- Raspberry Pi: 273 euros

Cabe destacar que, si escogemos la opción de montar el servidor en una Raspberry, el gasto a partir del segundo año tan solo será el de la conexión a Internet (180 euros). Este dinero no supondría un gasto a mayores ya que la conexión a Internet es un recibo habitual en el hogar. También hay que tener en cuenta que, esta opción no sería viable si la aplicación comienza a ser usada por muchos usuarios. Dentro de las cuotas estudiadas, una RAM de 1 GB no sería suficiente para dar soporte a 1.000 usuarios. Por esta razón, esta posibilidad está mayormente orientada al uso personal del bot y es extensible al círculo de amigos.

El servidor en la nube es la mejor opción para una aplicación profesional. Nos asegura que siempre va a estar disponible y ofrece servidores con las mejores características en cuanto a memoria y velocidad [9]. Sin embargo, desde mi punto de vista, el precio es excesivo para la aplicación desarrollada.

La mejor opción relación servicio/precio sería un **VPS**. Como se ha podido observar en el punto correspondiente, existen servicios que ofrecen este tipo de servidores desde muy poco dinero. Podríamos tener un servidor por 144 euros al año y dar soporte a miles de usuarios.

Plan de cierre

Para el correcto cierre del proyecto es necesaria una reunión final. En esta reunión deben estar presentes las partes interesadas en el proyecto: Rebeca Rodríguez Feria y César Llamas Bello.

Los temas a tratar son:

- El nivel de éxito del proyecto, evaluar el grado de consecución de todos los requisitos fijados.
- Analizar el desempeño de las actividades necesarias para llevar a cabo el proyecto.
- Revisar el proyecto objetivamente junto con su documentación y su presentación para preparar la defensa de este ante el jurado.

La finalidad de la reunión es analizar si el proyecto ha cumplido con los hitos que se propusieron en el documento de planificación, si el tiempo predicho para cada actividad ha diferido del tiempo necesitado realmente, evaluar si el modo de trabajo utilizado ha sido el que mejores resultados podía dar y observar si se han satisfecho los objetivos iniciales fijados para la aplicación. Se pretende que al analizar todos estos puntos se adquieran conocimientos que puedan servir para mejorar el rendimiento en futuros proyectos.

Antes de dar por terminado el proyecto es interesante para futuros proyectos recolectar todos los documentos realizados durante el desarrollo del mismo, así como un histórico de información con las soluciones propuestas a los problemas técnicos que han ido surgiendo y explicaciones de cómo se han hecho las cosas. Esta documentación en nuestro caso nos puede servir como guía para realizar futuros bots para Telegram o aplicaciones en lenguaje Python.

Capítulo 2. Introducción. ¿Qué es Telegram?

Actualmente la tecnología forma parte de nuestras vidas. Nos facilita realizar multitud de tareas y por ello se ha convertido en medio para realizar muchas de ellas. A día de hoy, los dispositivos electrónicos han remplazado a las herramientas tradicionales en múltiples campos. El uso de los ordenadores ha evolucionado y ha aparecido lo que comúnmente es llamado *IoT* (**Internet of Things).** En los últimos años, todos los dispositivos han comenzado a ser "inteligentes". El reloj ha sido reemplazado por el *smartwatch* y la televisión se ha visto sustituida por la *Smart Tv.* Ya no leemos en un libro sino en un *e-book.* La expansión informática está revolucionando todo a pasos agigantados. El afán por informatizar todo cada año es mayor y las personas no reniegan de estos cambios. Los usuarios crecen cada año. Basándonos en un estudio sobre las tecnologías en España realizado cada año por Ditendria, división digital de Tatum, en 2016 en España un 87% de los teléfonos móvil eran *smartphones.* En este estudio también se dice que de media un español utiliza el móvil una media de 3 horas y 23 minutos diarios y, además, el 51% de los españoles usan a diario el móvil para chatear a través de mensajería instantánea. Respaldando estos datos, el Informe de *La Sociedad de la Información* realizado por Telefónica corrobora el enorme uso del smartphone. En este informe se puede ver que el teléfono móvil es el dispositivo preferido para acceder a Internet, un 93,3% de los usuarios lo utilizan para navegar por internet [10]. En este contexto, aparecen centenares de aplicaciones a diario. Algunas se vuelven populares un periodo de tiempo tras el cual son desinstaladas, otras no obtienen apenas descargas y las afortunadas se convierten en básico indispensable en todos los smartphones. En esta última categoría se encuentran las populares aplicaciones de mensajería instantánea. El gran gigante de la mensajería es WhatsApp, del cual se hablará en posteriores apartados. Con muchos menos usuarios, pero un enorme crecimiento en sus primeros años de vida se encuentra Telegram.

Telegram es principalmente una aplicación para móviles y tablets, concretamente un servicio de mensajería instantánea. A través de esta aplicación, dos o más usuarios pueden establecer una conversación a tiempo real mediante mensajes. Esta interacción puede ser a través de mensajes de texto, imágenes, vídeos o notas de voz. Si nos quedamos en esta definición, Telegram es otra aplicación más entre muchas que nunca podrá equipararse con WhatsApp debido a la gran cantidad de usuarios que ya tiene establecidos. Existen algunas características de Telegram que han permitido que destaque, propiciando así su gran crecimiento en poco tiempo. Lo primero que se puede destacar de Telegram es que es de código abierto, por lo que cualquier usuario con conocimientos suficientes, y que así lo quiera, puede aportar algo para mejorar la app.

Otro punto a favor que tiene la app es la importancia que dan a la seguridad de los mensajes enviados a través de ella. El servicio está basado en el protocolo MTProto el cual soporta documentos,

multimedia y grandes archivos. Este protocolo fue diseñado específicamente para Telegram a diferencia de otros servicios similares que usan XMPP. MTProto fue desarrollado bajo un estándar abierto a base de API Java. El protocolo se divide en tres fases: Alto nivel de componente, capa de criptografía y transporte.

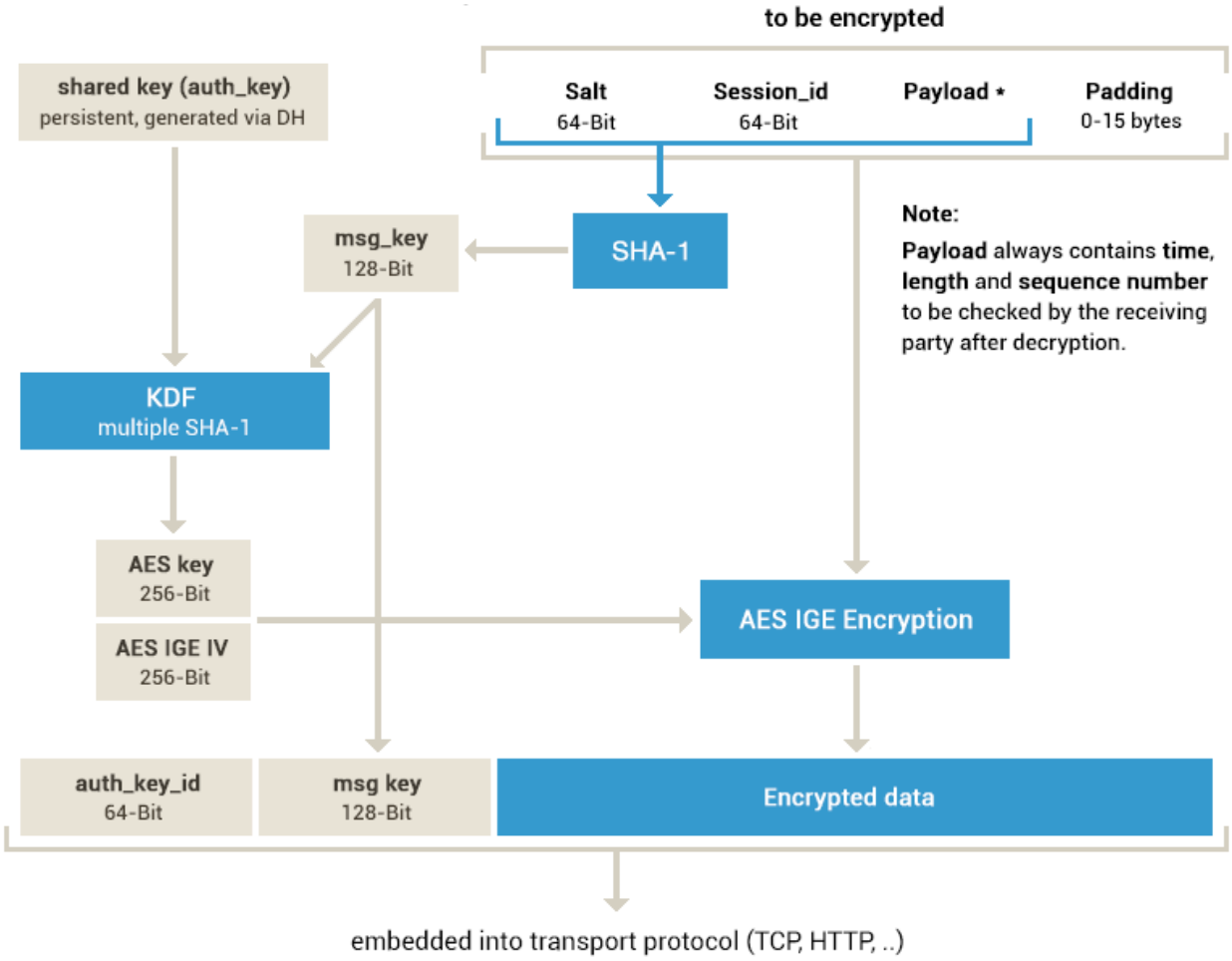

Figura 1. Funcionamiento del protocolo MTProto

En la capa de **alto nivel** se crea una sesión entre el cliente y el servidor, esta sesión se establece con el dispositivo físico concretamente. Una vez establecida una conexión con éxito se le asigna al dispositivo una clave identificadora de usuario. Esta clave es necesaria para desencriptar los mensajes recibidos desde el servidor y generalmente se genera una única vez para cada usuario. La respuesta a un mensaje puede ser enviada por otra conexión diferente, pero nunca en una sesión a la cual no pertenece. Cuando se usa UDP en la conexión, la respuesta puede enviarse mediante una dirección IP diferente a la del mensaje recibido.

Todos los mensajes tienen la misma estructura: el identificador de la sesión, un número secuencial perteneciente a su sesión, la longitud del propio mensaje, el cuerpo del mensaje y una cabecera. También tiene un campo “Salt”. Este campo es común encontrarlo cuando se va a encriptar un mensaje. El *Salt* (comúnmente conocido como semilla) es un número de bits aleatorios que se usan como entrada añadida para generar el encriptado. Este valor añade un plus de dificultad al trabajo de desencriptar el mensaje original si este no es conocido. Esta dificultad añadida se basa en que dos mensajes idénticos no darán nunca el mismo mensaje encriptado debido a los bits aleatorios introducidos. [11]

Este mensaje es **encriptado** mediante cifrado SHA-1 antes de ser enviado como se puede observar en la Figura 1. **SHA** (*Secure Hash Algorithm*) es una familia de funciones Hash creadas por el Instituto Nacional de Estándares y Tecnología. SHA-1 permite entradas de hasta 2^{64} bits y proporciona salidas de 160 bits (20 bytes) independientemente del tamaño de entrada. Estas salidas son únicas y para cada valor de entrada existe un valor de salida diferente. En febrero de 2017, la seguridad de SHA-1 se ha visto comprometida por primera vez debido a que se ha producido la primera colisión. Es decir, dos entradas diferentes han producido una salida idéntica. De momento, Telegram no ha tomado ninguna medida al respecto y mantiene el cifrado SHA-1. Al mensaje encriptado, se le añade una segunda cabecera la cual contiene un identificador del mensaje y el identificador del usuario. La capa de encriptado nos da la seguridad de que si alguien consigue interceptar los mensajes de un chat necesitaría robar el dispositivo físico para obtener la clave identificadora de usuario. Solo con este clave se puede desencriptar los mensajes enviados a un usuario.

En la capa **transporte** el mensaje encriptado se envía a través de uno de los siguientes protocolos: HTTP, TCP o UDP. El protocolo TCP da soporte a HTTP, por esta razón ambos comparten la mayoría de características. El protocolo TCP (***Transmission Control Protocol)*** está orientado a la conexión entre un cliente y un servidor. Este protocolo incorpora control de recepción de mensajes, control en el orden de los mensajes y control en la integridad de los mensajes. Esto se consigue introduciendo algunos campos en la cabecera del mensaje: número de secuencia y *checksums*. Además, TCP permite varias aplicaciones en una misma dirección gracias a los "puertos". En Telegram, cuando se crea una conexión TCP o HTTP se asigna al puerto 80 de la sesión que transmitió el primer mensaje. Por el contrario, el protocolo **UDP (User Datagram Protocol)** está orientado a la no conexión. Suele utilizarse en flujos unidireccionales de información. No nos permite saber si los mensajes se reciben en el mismo orden en el que fueron enviado ni comprobar si estos se han recibido sin errores. En la cabecera del mensaje se introduce tan solo lo necesario para saber quién es el emisor de este [12].

Historia de la aplicación

Desde pequeño **Pável Dúrov** sintió interés por la programación. En la escuela hackeo la red de ordenadores y consiguió cambiar la pantalla de bienvenida. A pesar de que la escuela cambió las contraseñas y le cortó el acceso a la red, Pável consiguió crackear las contraseñas nuevamente. Él siempre quiso ser un referente en Internet. Comparte su afán por la informática con su hermano mayor: Nikolai Durov. Su hermano siempre fue su mentor y referente. Nikolai fue un niño prodigio. Se presentó cuatro años a la Olimpiada Internacional de Informática y todos los años consiguió varias medallas. También participó en la Olimpiada Internacional de Matemática, en la cual obtuvo una medalla cada año. Ganó el ACM *International Collegiate Programming Contest* dos años consecutivos. Se licenció en Matemáticas y Ciencias de la Computación.

En 2006, Pável se licenció en filología. Recién graduado, en septiembre de ese mismo año, Pável lanzó una versión beta de una red para compartir apuntes y libros. Había estado trabajando en esta idea mientras se preparaba para ser interprete y traductor. La web se hizo popular entre sus compañeros de universidad rápidamente. A partir de ese momento Nikolai entró a formar parte del proyecto y comenzó a ayudar a su hermano a mejorar la aplicación. Extendieron la web a todo el campus universitario. Al principio solo se podía acceder mediante invitación. Viendo el potencial de la web, el padre de un antiguo compañero de Pável financió el proyecto en noviembre y registraron el nombre de dominio vkontakte.ru.

En febrero de 2017 el sitio ya había alcanzado 100.000 usuarios. La web empezó a ser utilizada por todo tipo de usuarios y no solo estudiantes universitarios. En Julio de 2007, logró alcanzar la cifra de 1 millón de usuarios y en menos de un año logró 10 millones. A finales de 2008 se convirtió en la red social más popular de Rusia. Después de dos años desde el lanzamiento de su beta, vkontakte ya no se limitaba a la difusión

y la transmisión de apuntes, los usuarios comenzaron a compartir todo tipo de archivos. Las películas y canciones piratas inundaban la red. VKontakte no tomó ningún tipo de represalia ni medida ya que la difusión gratuita de contenido entraba dentro de los ideales de libertad de Pável. Esto incitó un montón de críticas hacia la web. Acusaban a Pável de permitir y facilitar la violación de los derechos de propiedad intelectual.

En diciembre de 2012 se celebraron elecciones. Ganó el partido gobernante. Los rumores de que habían manipulado las elecciones se difundían a través de VKontakte. El gobierno de Rusia pidió a VKontakte que bloqueara a los grupos extremistas y solicitaron a Pável que cerrase las cuentas relacionadas. Pável se negó y por esta razón fue mandado a testificar en varias ocasiones. En abril de 2014, Pável respondió públicamente en sus redes sociales con imágenes provocadoras que no proporcionaría la información que le solicitaba el gobierno. Días más tarde fue despedido como CEO de la compañía debido a una carta de dimisión escrita por él mismo. Pável declaró que la compañía había sido tomada por Putin y finalmente vendió el 12% de las acciones de la compañía que aún conservaba. Ese mismo año se marchó de Rusia.

En 2013 nació Telegram de mano de los hermanos Dúrov. Esta aplicación surge independiente a VKontakte y se presenta como una organización sin ánimo de lucro. Fue desarrollada para cubrir la necesidad de Pável de comunicarse sin ser espiado por la FSB (Servicio Federal de Seguridad de la Federación Rusa). Nikolai desarrolló el protocolo MTProto específicamente para Telegram. Buscaba satisfacer con éxito los objetivos de seguridad y privacidad con los cuales había nacido la app. En agosto se lanza la app para iOS y en abril de 2014 para Android. Desde sus inicios Telegram se presenta como una aplicación de código abierto, gratuita, segura y rápida.

La residencia legal de Telegram no se encuentra establecida en ningún país en concreto. Según Pável esto les obligaría a regirse bajo las leyes de ese país. La "sede" se encuentra en Berlín habitualmente, aunque va cambiando de ubicación a la par que Pável y su equipo cambia de domicilio. El lanzamiento de Telegram no solo puso a los hermanos en el punto de mira del FSB, esta vez fueron muchos los países que se fijaron en ellos. China bloqueó el acceso a Telegram y en Irán están bloqueadas algunas de las opciones de la app (por ejemplo: los mensajes de voz). Las leyes de Rusia estudiaron la implantación de una medida similar a China después de que se averiguó que un grupo terrorista había utilizado Telegram para comunicarse y realizar un ataque en Paris. También entró en controversia el uso de los canales de la aplicación. Se acusó de que existían algunos canales para distribuir mensajes de adoctrinamiento Islámico. Desde entonces, Telegram ha cerrado alrededor de 100 canales que eran utilizados para este fin. En diciembre de 2015, se solicitó a Apple que desencriptara los datos de un iPhone, propiedad del responsable de un atentado terrorista. Pável apoyó públicamente a Apple con la defensa de la privacidad de los datos personales de cualquiera de sus usuarios. Sin embargo, Apple se desentendió de cualquier relación con Telegram*: "una de las aplicaciones más peligrosas relacionadas con el terrorismo es Telegram. Ellos no tienen nada que ver con Apple."* La refinada seguridad de Telegram puede tener partidarios y detractores. Lo que está claro es que, hasta el momento, es inquebrantable. [13 y 14]

Con el fin de corroborar y demostrar la seguridad de la aplicación y aprovechar una de sus principales características: ser open source; se lanzan distintos concursos que ponen a prueba la propia app y a los usuarios. A principios de 2014, Telegram anuncia su primer concurso. Se paga 200.000 dólares a la primera persona que rompa el protocolo de encriptación de Telegram antes de marzo. Este concurso no obtuvo ningún ganador [15]. En noviembre se anunció un nuevo concurso en el que se ofrecían 300.000 dólares por crackear el cifrado de Telegram, esta vez los concursantes no sólo podían controlar el tráfico, sino también actuar como servidor de Telegram y usar ataques activos. El concurso finalizó sin ganadores en febrero de 2015 [16]. En 2016 se anuncia el concurso *BotPrize.*

Este consiste en premiar a los mejores creadores de bots de la comunidad. En 2016 se eligieron 5 ganadores y se les otorgó 50.000$ a cada uno. En 2017 este concurso se repite. [17]

Características

Telegram sobresale frente a otras aplicaciones de mensajería gracias a multitud de características de las que solo ella dispone. Como ya mencionamos antes, su código es abierto. Otra característica destacable es que todos los chats de usuario son almacenados en la nube, por lo que son accesibles desde cualquier dispositivo una vez iniciemos con la misma cuenta de usuario. Además, dispone de otras muchas características interesantes [18]:

Multiplataforma. Telegram se encuentra disponible para múltiples plataformas: iOS, Android, Windows Phone, Ubuntu Touch y una versión de escritorio para Windows, Mac OS X y basados en UNIX. También disponen de una aplicación web con la que se puede acceder a la cuenta sin necesidad de instalaciones. El servicio web se encuentra alojado en la dirección *web.telegram.org* . Todas las versiones llevan disponibles desde, prácticamente, el inicio de la aplicación. Esta característica ya ha sido copiada por otros servicios como WhatsApp.

Mensajes. Los mensajes en Telegram pueden ser modificados una vez son enviados. Podemos editar el texto si nos equivocamos, el mensaje cambiará su estado a "editado" a la hora que hayamos hecho el cambio. También podemos eliminar un mensaje enviado con el consecuente de que también será eliminado en el dispositivo receptor. Esta opción solo está disponible las primeras 48 horas después de haber sido enviado el mensaje.

Chats secretos. Son chats normales, pero con una seguridad extra y algunas opciones especiales. Los mensajes que enviamos a través de estos chats están cifrados end-to-end. Cuando se crea un chat secreto, ambos dispositivos intercambian claves de cifrado (claves de intercambio Diffie-Hellman).

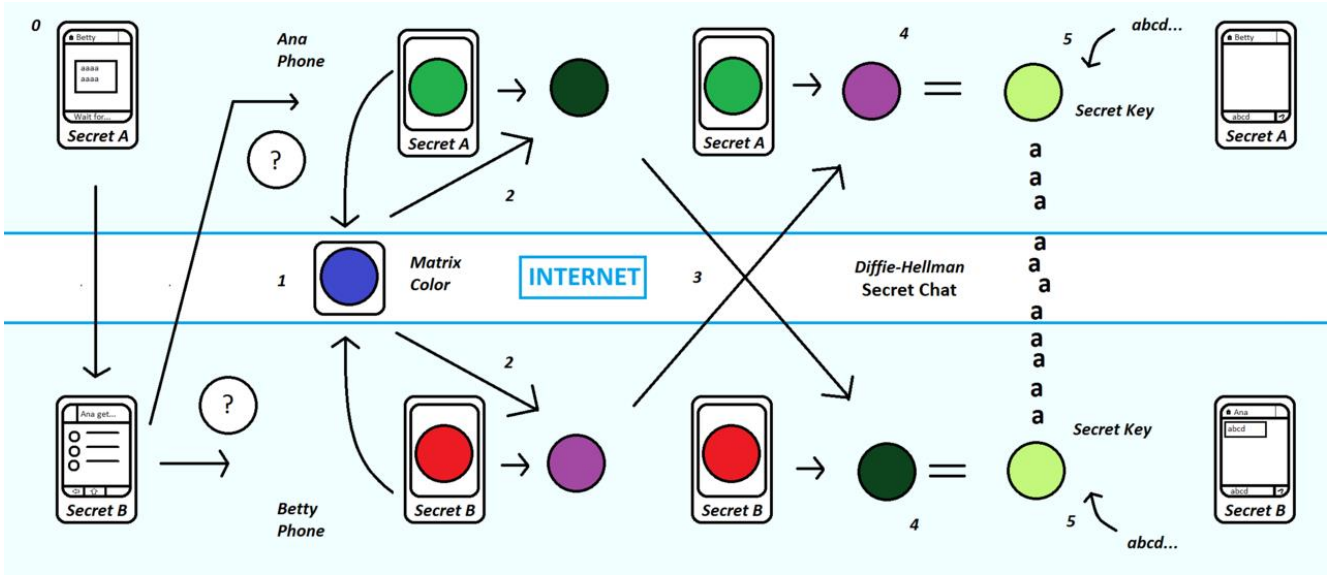

Figura 2. Funcionamiento del protocolo Diffie-Hellman

En la figura 2 podemos observar un esquema de cómo funciona el protocolo Diffie-Hellman. Al crear un chat secreto, los dispositivos obtienen dos números públicos cada uno (color verde claro y rojo respectivamente). Usando sus dos números y el número secreto (color azul) obtienen un resultado tras aplicar una fórmula matemática. Después, se intercambian estos resultados. Utilizando la fórmula matemática, los dos resultados y el número secreto los dos deben llegar al mismo resultado (color

amarillo verdoso final). Si el resultado final es el mismo la conexión es segura y este se usa como clave privada dentro del chat. [19]

Una vez se establece una conexión segura extremo a extremo, se genera una imagen con la clave de cifrado (privada) para el chat. Esto permite que dos usuarios puedan comprobar si su chat es realmente privado fácilmente. Si las imágenes de los dispositivos pertenecientes al chat son iguales, el chat es seguro y ningún ataque intermediario puede tener éxito. Ni siquiera Telegram puede acceder a los mensajes enviados mediante un chat secreto. Los mensajes enviados mediante chats secretos no pueden reenviarse y el emisor puede fijar un tiempo de vida al mensaje. Es decir, el emisor puede marcar el tiempo que quiere que el mensaje permanezca en los dispositivos una vez leído por el destinatario, dos ticks verdes. Al finalizar el tiempo el mensaje se autodestruye y desaparece del dispositivo emisor y receptor. Además, si uno de los participantes hace una captura de pantalla de la conversación, Telegram informa al otro participante. Los chats secretos se crean únicamente en el dispositivo físico. Por lo tanto, no están disponibles si nos conectamos desde otras sesiones (Telegram web, por ejemplo).

Two-Step Verification (Verificación en dos pasos). Se trata de un añadido de seguridad a la hora de acceder a la cuenta de un usuario. Habitualmente al intentar acceder a una cuenta se pide el Pin recibido por SMS, sistema que usan la mayoría de aplicaciones de mensajería instantánea. A esta verificación, Telegram suma (si el usuario lo quiere) la verificación por clave. Activándolo desde las opciones de privacidad, el usuario puede establecer una contraseña que se pedirá cada vez que se intente acceder a su cuenta desde un nuevo dispositivo.

Nick/Apodo. Como es habitual en los servicios de mensajería, al registrarse se pide solo tu número de teléfono. El teléfono es lo único necesario para identificarnos ante el servidor y entre contactos. Telegram añade la opción de añadir un apodo a nuestra cuenta. Gracias a esto, se puede añadir un contacto únicamente con el apodo sin necesidad de compartir el número de teléfono.

Envío de archivos. En aplicaciones análogas se permite a los usuarios enviar archivos multimedia. Telegram amplía la diversidad de archivos que se pueden compartir a través de sus chats. A los "típicos" envíos de imágenes, vídeos, audios y ubicaciones sumamos el envío de pdfs, documentos, apks, etc. Se pueden compartir archivos de hasta 1,5 GB de cualquier extensión conocida.

Grupos, supergrupos y canales. Los **grupos** en Telegram tienen un límite de 200 usuarios. En un grupo todo el mundo puede cambiar el nombre y la foto de este y añadir nuevos contactos. Los contactos pueden añadirse mediante su apodo. En un **supergrupo** se pueden añadir hasta 5000 miembros. Los supergrupos tienen administradores con opciones específicas para administrar el supergrupo. Los administradores pueden añadir nuevos miembros, borrar mensajes del grupo y bloquear contactos. Los administradores también pueden anclar mensajes al chat. Cuando un mensaje es anclado aparece en la parte superior del chat siempre y se puede acceder a él directamente. Además, al anclar un mensaje se avisa a todos los miembros del supergrupo, incluso cuando el grupo se encuentra silenciado. Los supergrupos tienen un historial único, por lo que si el emisor edita o elimina su mensaje este se modifica en todos los dispositivos del grupo. Se pueden mencionar mensajes y *taggear* miembros para facilitar las comunicaciones dentro del chat. Si silenciamos un supergrupo solo nos llegaran notificaciones de los mensajes que nos afecten directamente (menciones y respuestas a nuestros mensajes). Un supergrupo tiene distintos niveles de privacidad, si el chat es público se crea un link mediante el que cualquier persona que disponga de él puede unirse sin invitación previa. Por el contrario, los supergrupos privados requieren que un administrador nos añada al chat. Los **canales** tienen un objetivo difusor. En un canal la comunicación es unidireccional, solo los administradores del canal pueden publicar mensajes. Estos mensajes se publican en nombre del canal

y no con su propio nombre. Los canales tienen un número ilimitado de miembros. Al igual que los supergrupos, un canal tiene dos niveles de seguridad (público y privado).

Privacidad selectiva. Esconder la hora de última conexión o nuestra foto es algo que otras aplicaciones de mensajería permiten. Telegram también nos da esa opción, pero de una forma más amplia. Habitualmente se puede elegir que estas informaciones se muestren a "Todos", "Nuestros contactos" y "Nadie". Telegram añade un filtro mediante el cual podemos seleccionar usuarios concretos a los que no mostrar alguna de estas informaciones. También podemos restringir quién queremos que tenga el poder de añadirnos a los grupos, de esta forma evitamos que desconocidos nos agreguen a grupos.

Temas. En su afán por adaptarse a cada usuario, Telegram lanza en febrero de 2017 una actualización que incluye esta nueva característica. Los temas permiten a cada usuario cambiar toda la interfaz de la aplicación. Cualquier usuario puede modificar los colores todo. Desde el fondo, las notificaciones y la letra hasta el Nick de usuario. Todo es customizable. Por defecto, Telegram añade un tema "nocturno" para facilitar el uso de la app en espacios oscuros. Además, los usuarios pueden compartir sus temas con el resto de la comunidad.

Instant view. Recientemente Telegram ha incorporado la posibilidad de ver páginas web desde la propia aplicación. Los links seguidos desde mensajes enviados a través de la app se abren en una ventana aparte dentro de Telegram. Las webs se muestran con un diseño adaptado, incluso aquellas que no disponen de vista móvil desde el navegador. Para animar a la comunidad a colaborar en la labor de adaptar el mayor número de webs posibles, Telegram ha abierto un nuevo concurso. Cada adaptación equivale a unos puntos, quien obtenga más puntos por realizar un mayor número de adaptaciones y/o las adaptaciones a webs con mayor ponderación/dificultad ganará 10.000 dólares. El segundo puesto será premiado con 5.000 dólares. Además, la mejor adaptación para cada web será premiada con 100$. [20]

Telescope. En abril de 2017, Telegram ha añadido también una nueva forma de comunicarse. Su competidor, WhatsApp, permite el envío de notas de voz o llamadas, además de los habituales mensajes de texto, para comunicarse con otros usuarios. Telegram añade a estas opciones, las cuales ya incorpora, los vídeos. A través de WhatsApp también se pueden enviar vídeos, pero desde Telegram se realiza de una manera más dinámica con *Telescope*. Esta nueva funcionalidad permite enviar vídeos de hasta un minuto a través de los diferentes chats. Estos vídeos interactúan con el usuario de una forma análoga a las notas de voz. Podemos reproducirlos dentro de una conversación sin necesidad de descargarlos y se graban y envían directamente desde Telegram manteniendo pulsado un botón. [21]

Pago mediante bots. Esta característica se encuentra en desarrollo. Telegram ha puesto una versión de prueba en su última actualización. Se pretende conseguir que todas las compras online puedan realizarse a través de la propia app. Con unos cuantos clics, un procedimiento sencillo, rápido e intuitivo. Una vez recibido el enlace con el producto, el usuario tan solo tiene que pulsar en comprar y se abre un popup que muestra el resumen de compra. Puede seleccionar el método de pago y la forma de envío y tras esto pulsar en el botón de pagar. Si el usuario tiene activado en su cuenta la "Verificación de dos pasos" puede guardar sus tarjetas para no tener que introducir la forma de pago en cada compra. [22]

Bots en Telegram

La característica más destacable en Telegram son los bots. Por esta razón, le dedicamos un aparatado completo aparte. Los bots son pequeñas aplicaciones que podemos agregar a Telegram para tener nuevas funcionalidades que no vienen por defecto. Se encuentran en los servidores y los usuarios pueden "llamarlos" para conseguir cierta funcionalidad. Según el tipo de usuario esta característica tendrá un interés u otro. Para un usuario estándar los bots pueden añadir nuevas funcionalidades muy variadas que le ayuden con distintas tareas. Sin embargo, para un usuario interesado en la programación lo más atractivo de los bots es la posibilidad de poder crearlos. Gracias a esta característica un usuario que sepa programar puede personalizar completamente la app. Independientemente del tipo de usuario, podemos ver que los bots son un añadido muy interesante dentro de la aplicación.

Existen bots de todo tipo, pero todos pueden catalogarse dentro de dos grupos: bots "originales" y bots "inline". Para utilizar un bot "original" debemos añadirlo a un chat tal y como hacemos al agregar cualquier contacto a una conversación. Se debe poner una arroba delante del nombre del bot para buscarlo. Una vez dentro de la conversación podemos llamar y usar el bot siempre que lo necesitemos. Los bots "inline" se usan exactamente igual con la única diferencia de que no es necesario agregarle antes a la conversación y podemos hablarles directamente usando su nombre antecedido por un "@".

Los bots pueden agregar juegos a los chats, ser buscadores de imágenes, gifts, información; actuar como alarmas, permitirnos realizar encuestas, informarnos del tiempo, traducir todo lo que escribamos e infinidad de opciones más. La lista de bots disponible es muy numerosa y cada día crece gracias a la posibilidad de que cualquiera con conocimientos suficientes puede crear sus propios bots y luego compartirlos.

Para comenzar a utilizar un bot siempre hay que añadirle al chat o enviarle un mensaje, un bot nunca empezará una conversación por sí mismo. Habitualmente, el mensaje que le enviemos se inicia con un "/". Esta "norma" es un estándar que identifica el mensaje como comando del bot. Si añadimos un bot a un chat grupal, no recibirá ningún mensaje que no comience así. Para que un bot reciba todos los mensajes de una conversación el desarrollador debe desactivar el modo de privacidad, aunque esta práctica es desaconsejada por el equipo de Telegram a no ser que sea estrictamente necesario para que el bot funcione correctamente. Un bot sí recibirá todos los mensajes que los usuarios envíen si estos empiezan una conversación privada con él. Para saber si un bot tiene el modo de privacidad activado, al agregarle a un grupo se debe comprobar la información del grupo. Si debajo del bot aparece el mensaje "no tiene acceso a los mensajes", el bot no recibirá ningún mensaje que no empiece por "/". Por el contrario, si el mensaje es "tiene acceso a los mensajes", el bot recibirá todos los mensajes enviados por el chat y se debe tener cuidado al usarlo. Aunque Telegram no almacene datos de conversaciones, un bot con la privacidad desactivada podría estar proporcionando a alguien todos los mensajes que enviamos a través de los grupos donde se ha añadido al bot. [23]

Actualmente, Telegram solo tiene abierto el código cliente. El código servidor se utiliza a través de API's que utilizan el protocolo MTProto para comunicarse con el servidor. Estas API's se encuentran disponibles en diferentes lenguajes de programación. Pável comunicó que abrirá el código servidor una vez este sea lo suficientemente estable. Las API's más completas se pueden ver desde la web oficial de Telegram. Importándolas en el proyecto, se puede hacer uso de sus funcionalidades simplemente llamando a los correspondientes métodos. De esta forma, un programador solo tiene que preocuparse del código cliente e incorporar las llamadas necesarias a servidor. Todas las API's incorporan métodos para el envío de mensajes y la edición de estos. Gracias a estas API's, el desarrollador del bot no debe responsabilizarse de desarrollar un protocolo de comunicación ni

ocuparse de que su bot sea seguro. Los lenguajes más comunes a la hora de desarrollar el bot y de los cuales Telegram incorpora varios APi [24] son:

- **PHP**. Lenguaje de scripting especialmente destinado al desarrollo del lado del servidor de páginas web. Es un lenguaje sencillo de aprender y trae consigo todas las ventajas del HTML en cuanto a personalización de estilos.
- **PYTHON**. Lenguaje de programación interpretado muy versátil. Una de sus ventajas es la legibilidad del código y el tipado dinámico. Es bastante flexible.
- **JAVA**. Lenguaje de programación interpretado orientado a objetos. Casi todo en java es un objeto/instancia de una clase. Es un lenguaje popularmente extendido.

Los bots son cuentas de usuario especiales que no requieren un número de teléfono. Todos los bots tienen un alias por el cual pueden ser buscados por el resto de usuarios y este siempre debe acabar en "bot". Además, todos tienen un "token" que les identifica de forma única. Las API's de los distintos lenguajes son un *framework* que finalmente resuelve las llamadas a sus métodos haciendo peticiones a la API bot de Telegram. Las peticiones deben tener la siguiente estructura indiferentemente de la API desde la cual se realice:

```
api.telegram.org/bot<token>/METHOD_NAME
```

Estas peticiones se realizan sobre https y la respuesta siempre es un objeto JSON. El JSON siempre tiene un campo de tipo Boolean llamado "ok" y un campo String "descripción". Si "ok" es Verdadero, la petición se ha resuelto con éxito y el resultado de esta se encuentra en el campo "resultado". Si el campo "ok" es Falso, se explica porque ha fallado la petición en "descripción".

A la hora de crear un bot, existe un bot que nos ayudará con el proceso. El desarrollador debe hablar a *@BotFather* y seguir algunos pasos. Es este bot quien nos proporcionará el "token" anteriormente mencionado. Una vez creado el bot y recibida la autorización, la Bot API y las distintas guías oficiales aportan la información necesaria al desarrollador sobre cómo implementar su bot. Este procedimiento se explicará mejor en el capítulo de Implementación.

Telegram vs WhatsApp

WhatsApp, propiedad de Facebook desde abril de 2016, es el servicio de mensajería instantánea más popular del mundo. Desde su compra, WhatsApp permite el enlace con tu cuenta de Facebook, aunque esto supondrá que Facebook tenga acceso a la información de tu lista de contactos (Información requerida para usar WhatsApp). Desde su lanzamiento en 2009, se ha ido mejorando la seguridad de la aplicación. Con el lanzamiento de Telegram en 2013, la seguridad de WhatsApp quedó inevitablemente en duda. En 2014, comienzan a encriptarse los mensajes de texto enviados mediante WhatsApp. Esta medida no fue suficiente, ya que era un encriptado débil y los mensajes interceptados podían ser desencriptados. En abril de 2016 implantan un cifrado end-to-end en los chats [25]. Con esta mejora todos los mensajes (texto, multimedia, voz y vídeo) que se envían están encriptados por lo que ni terceros ni WhatsApp pueden leerlos. Al no poder leer estos mensajes, WhatsApp no almacena estos mensajes, aunque permite al usuario mantener una copia local de sus conversaciones. Con esta mejora, la seguridad con la que se envían los mensajes dentro de la aplicación se igualó con la utilizada por Telegram. WhatsApp tampoco almacena ningún dato de carácter personal del usuario, la lista de contactos está sincronizada sólo en el propio terminal y no en los servidores de la compañía [26].

Actualmente si comparamos su encriptación ambos servicios son suficientemente seguros. Los dos utilizan un cifrado extremo a extremo en sus chats, en Telegram se debe usar un chat privado. Es importante recordar que para que los mensajes en WhatsApp se envíen cifrados ambos dispositivos deben tener instalada una actualización de la aplicación superior a la versión 2.16.39. Esta actualización fue lanzada el 24 abril de 2016 [27]. Ambas medidas de seguridad nos garantizan que los mensajes viajan seguros entre dispositivos, pero no que estos estén seguros en el propio terminal. Ante un ataque al dispositivo se pueden obtener todas las conversaciones. En WhatsApp se guardan copias locales periódicamente por lo que es posible que ante un ataque se puedan obtener incluso conversaciones antiguas ya eliminadas. Estas copias locales son cifradas en los teléfonos más modernos, versiones posteriores a *iOS8* y *Android Lollipop*. En Telegram las conversaciones se guardan en la nube, la compañía asegura que es segura y tienen las claves repartidas en distintos puntos para imposibilitar el acceso y desencriptado de esta.

A pesar de que WhatsApp ha logrado alcanzar el nivel de seguridad de Telegram, se queda atrás en muchos otros puntos. WhatsApp no implementa muchas de las características de Telegram detalladas en el apartado correspondiente **"Características"**. Además, algunas de las características que ambas comparten están mejor optimizadas dentro de Telegram [28]. Según un estudio realizado en mayo de 2017 [29], WhatsApp consume más datos al realizar las mismas acciones.

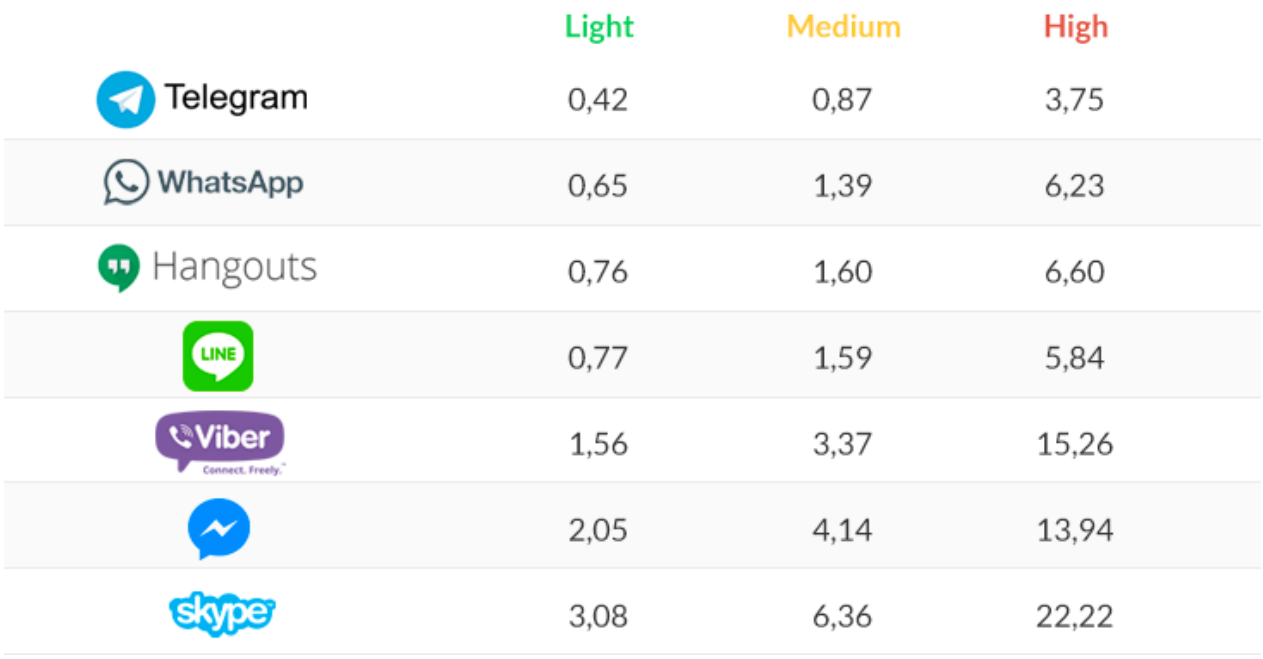

Figura 3. Comparativa de datos consumidos en Megabytes por cada aplicación con conexión 4G.

Los escenarios estudiados son:

- Nivel bajo: 20 mensajes enviados y 20 recibidos, 5 fotos recibidas y 2 enviadas.
- Nivel medio: 40 mensajes enviados y 40 recibidos, 10 fotos recibidas y 20 enviadas
- Nivel alto: 100 mensajes enviados y 100 recibidos, 50 fotos recibidas y 20 enviadas

Como se puede observar en la imagen, Telegram es el servicio de mensajería que menos consume en todos los escenarios. Le sigue WhatsApp. Con un uso intensivo de la aplicación, WhatsApp duplica los datos consumidos con respecto a Telegram. Estas diferencias se deben al elevado grado de optimización de Telegram.

Si sumamos la optimización de Telegram y su escaso consumo de datos a las múltiples características añadidas de las que dispone, podemos ver cómo es un fuerte competidor para WhatsApp. A pesar de su victoria como aplicación, WhatsApp sigue siendo la más usada con mucha diferencia. Este triunfo se suele atribuir a que salió al mercado antes y fue la primera aplicación de mensajería instantánea rápida y sencilla. Actualmente, a pesar de existir una alternativa mejor, muchos usuarios ya se han habituado al uso de WhatsApp y por costumbre y miedo al proceso de adaptación no prueban otras opciones.

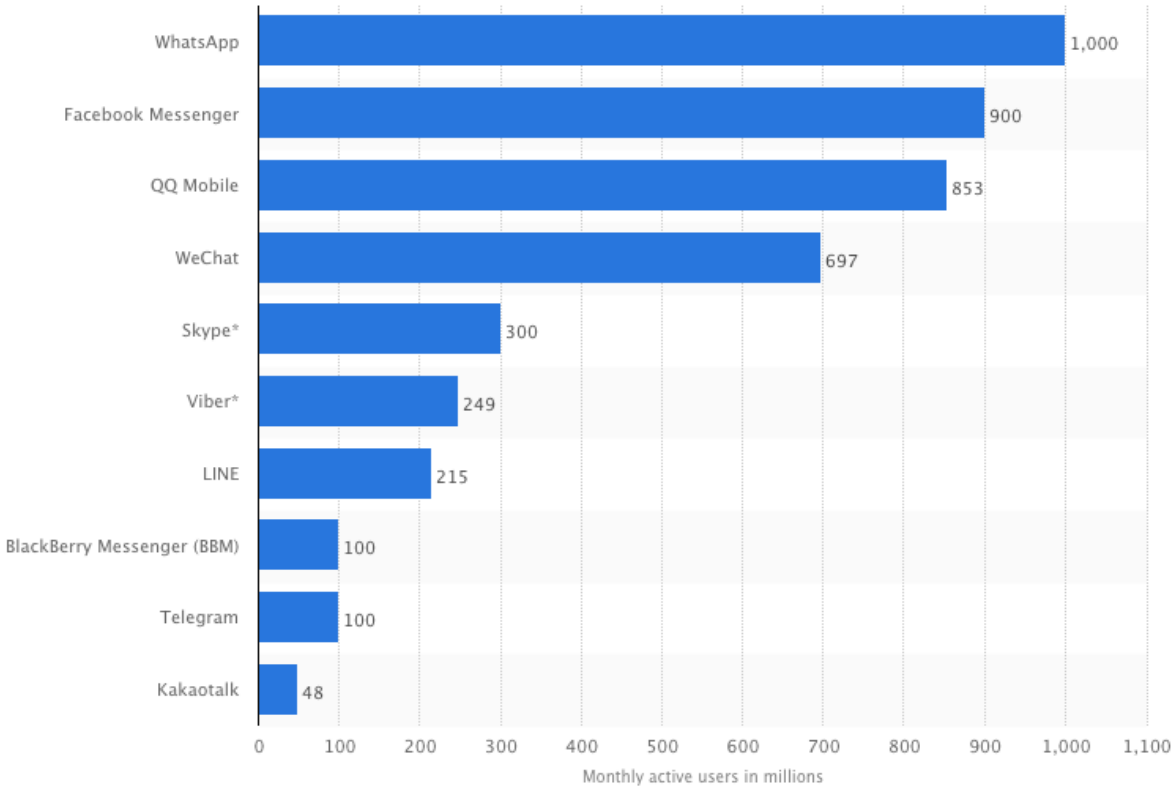

Figura 4. Gráfico de barras: Cantidad de millones de usuarios activos mensuales en diferentes aplicaciones de mensajería instantánea.

Se puede observar en el gráfico que WhatsApp es la aplicación más utilizada, seguida de Facebook Messenger. Estos datos son de febrero de 2016 [30]. En 2017, con la compra de WhatsApp por parte de Facebook, los dos gigantes se han unido y Facebook dispone en sus aplicaciones de alrededor de 2.000 millones de usuarios activos al mes. Esto le aleja mucho de la siguiente aplicación, QQ Mobile, con alrededor de 850 millones.

Telegram disponía de 100 millones de usuarios activos al mes, una cantidad modesta si la comparamos con WhatsApp. A pesar de esto, si observamos la figura 5 podemos augurar un prometedor futuro para Telegram.

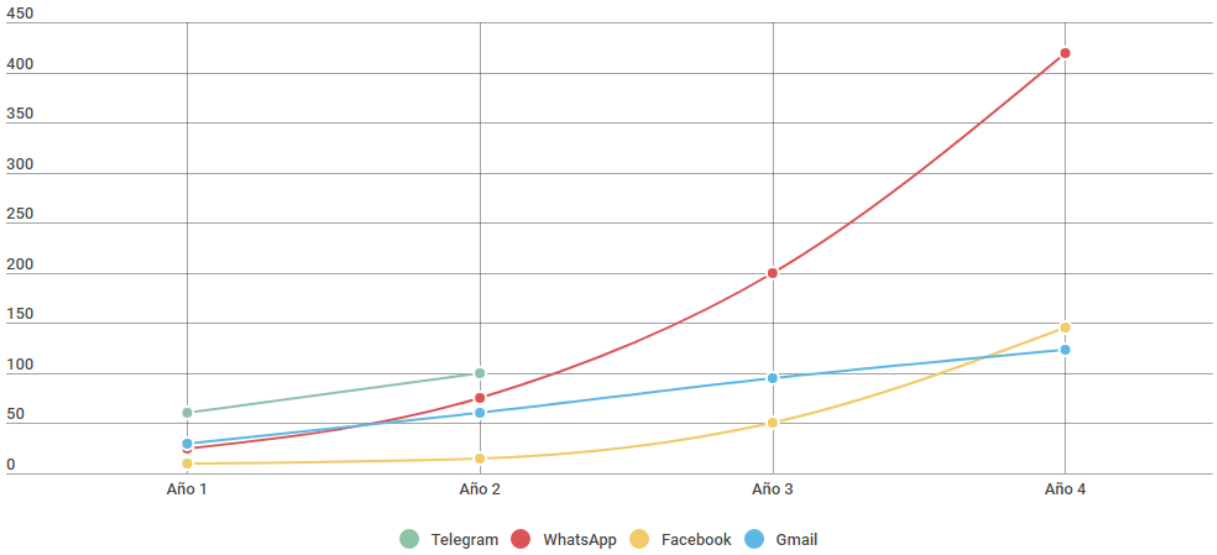

Figura 5. Gráfico de líneas: Cantidad de millones de usuarios activos mensuales cada año en los primeros años de distintas aplicaciones de mensajería.

En la gráfica podemos observar como Telegram ha sido la aplicación que más usuarios ha conseguido en su primer y segundo año de vida. Falta por ver cómo ha crecido en su tercer año, pero todavía no existen datos relacionados con la cantidad de usuarios activos en 2017.

Capítulo 3. Análisis

En este capítulo se lleva a cabo un estudio inicial del problema. Se analizan los requisitos necesarios para desarrollar la aplicación. Aunque a priori el análisis de requisitos parezca una tarea sencilla, es todo lo contrario. Es imprescindible llevar a cabo un estudio profundo de los requisitos que existen antes de comenzar a perfilar la aplicación. Los requisitos son la base de todo proyecto software y es muy importante que estén bien definidos. Hay que identificar todos los requisitos que debe cumplir el sistema. Es habitual que algunos requisitos estén incompletos, sean ambiguos o contradictorios. Un contacto constante entre desarrollador y cliente permite obtener una visión completa de los requerimientos que debe cumplir el sistema. Fijar bien los requisitos desde el principio, permite obtener un producto que cumpla todas las expectativas del cliente. Además, los casos de uso que se identifican durante esta etapa y sobre los que desarrollaremos la aplicación se construyen en base a los requisitos.

Requisitos

Los requisitos software son la descripción de los servicios que proporcionará el sistema a desarrollar y las restricciones de este. Representan los objetivos que se deben cumplir, por lo que es importante identificar los máximos posibles antes de comenzar con el desarrollo del proyecto. Los requisitos se dividen en tres grupos: requisitos funcionales, no funcionales y de información [31].

Requisitos funcionales

Describen las funciones que el sistema debe realizar para cumplir con los objetivos propuestos inicialmente. Los requisitos funcionales pueden ser: cálculos, detalles técnicos, manipulación de datos y otras funcionalidades específicas. En resumen, establecen el comportamiento del sistema.

RF-1. Crear checklist. El sistema deberá permitir al usuario crear una nueva checklist con nombre único (dentro de sus listas) en cualquier momento.

RF-2. Editar checklist. El sistema deberá permitir al usuario modificar una checklist ya existente y de la cual es el creador.

RF-2.1. Eliminar checklist. El sistema deberá permitir al usuario creador de una lista eliminarla en cualquier momento.

RF-2.2. Añadir tareas. El sistema deberá permitir al usuario añadir nuevas tareas a una checklist ya existente y de la cual es el creador.

RF-2.3. Modificar tareas. El sistema deberá permitir al usuario modificar tareas pertenecientes a una checklist ya existente y de la cual es el creador.

RF-2.4. Eliminar tareas. El sistema deberá permitir al usuario eliminar tareas pertenecientes a una checklist ya existente y de la cual es el creador.

RF-2.5. Reinicializar checklist. El sistema deberá permitir al usuario creador de una lista reinicializarla en cualquier momento.

RF-3. Asignar tareas a personas. El sistema deberá permitir al usuario creador de una checklist asignar determinadas tareas a otros usuarios.

RF-4. Marcar tareas hechas. El sistema deberá permitir al usuario creador o a cualquier usuario con el que la checklist haya sido compartida marcar tareas ya realizadas

RF-4.1. Marcar tareas asignadas hechas. El sistema deberá permitir únicamente al usuario asignado a una tarea concreta marcar que ya está hecha.

RF-5. Compartir checklist. El sistema deberá permitir al usuario creador de una lista compartirla en chats grupales.

RF-6. Clonar checklist. El sistema deberá permitir a cualquier usuario, que no sea el creador, con el que la checklist haya sido compartida duplicar la lista. El usuario que obtenga una lista duplicándola, se asignará como su creador.

RF-7. Ver checklists. El sistema deberá permitir a los usuarios ver todas las listas de las cuales son propietarios, es decir, las que han creado o clonado.

RF-8. Generar nombres. El sistema deberá generar y proponer un nombre único al crear una nueva checklist si el aportado por el usuario ya corresponde a una checklist existente.

Requisitos no funcionales

Los requisitos no funcionales describen características de funcionamiento del sistema y no cómo debe comportarse este. Es muy común especificar características de diseño o arquitectura dentro de estos y también temas relacionados con la seguridad, el rendimiento, la disponibilidad y la usabilidad del sistema entre otros.

RNF-1. Lenguaje de programación. Se utilizará Python como lenguaje de programación

RNF-2. Usabilidad. La aplicación será fácilmente utilizable para un 97% de los usuarios acostumbrados al uso de Telegram.

RNF-3. Diseño checklist. La aplicación dispondrá de un diseño intuitivo para mostrar el estado de las tareas de las checklists.

RNF-4. Rendimiento. La aplicación tendrá un tiempo de respuesta de máximo 2 segundos para el 99% de los casos.

RNF-5. Persistencia. La aplicación dispondrá de un modo de recuperar los datos de los usuarios en caso de que la aplicación deba reiniciarse.

RNF-6. Almacenamiento datos persistentes. La aplicación guardará la información de las checklists y de las sesiones en un XML.

RNF-7. Multilenguaje. La aplicación hablará al usuario en el idioma que este haya seleccionado.

Requisitos de información

RI-1. Checklist. El sistema deberá almacenar la información correspondiente a cada una de las checklist. En concreto:

- Nombre.
- Propietario/Creador.
- Lista de tareas.

RI-2. Tarea. El sistema deberá almacenar la información correspondiente a cada una de las tareas. En concreto:

- Descripción.
- Estado.
- Propietario.

El estado de una tarea puede ser hecha o no hecha. Por lo que se representará por una variable de tipo booleano. El propietario de una tarea es la persona a la cual está asignada dicha tarea. Si el campo está vacío significa que cualquiera puede cambiar el estado de la tarea. En caso contrario, existe un propietario, solo el propietario podrá modificar el estado.

RI-3. Sesión. El sistema deberá almacenar la información correspondiente a la sesión que cada usuario establece con el bot. En concreto:

- Identificador del chat
- Lista activa
- Tarea activa
- Mensaje inline activo

Los campos de lista activa, tarea activa y mensaje inline activo representan los valores: lista, tarea y mensaje sobre los cuales está “interactuando” el usuario. Son necesarios para guardar la concordancia entre los sucesivos mensajes y llevar a cabo correctamente las funcionalidades. Si estos campos están vacíos, el usuario, al cual identifica el identificador de chat, no está a mitad de ninguna tarea dentro de la aplicación.

Diagrama de casos de uso

La aplicación debe cubrir mediante los casos de uso que sean necesarios todos los requisitos descritos en el punto anterior: **Especificación de Requisitos.**

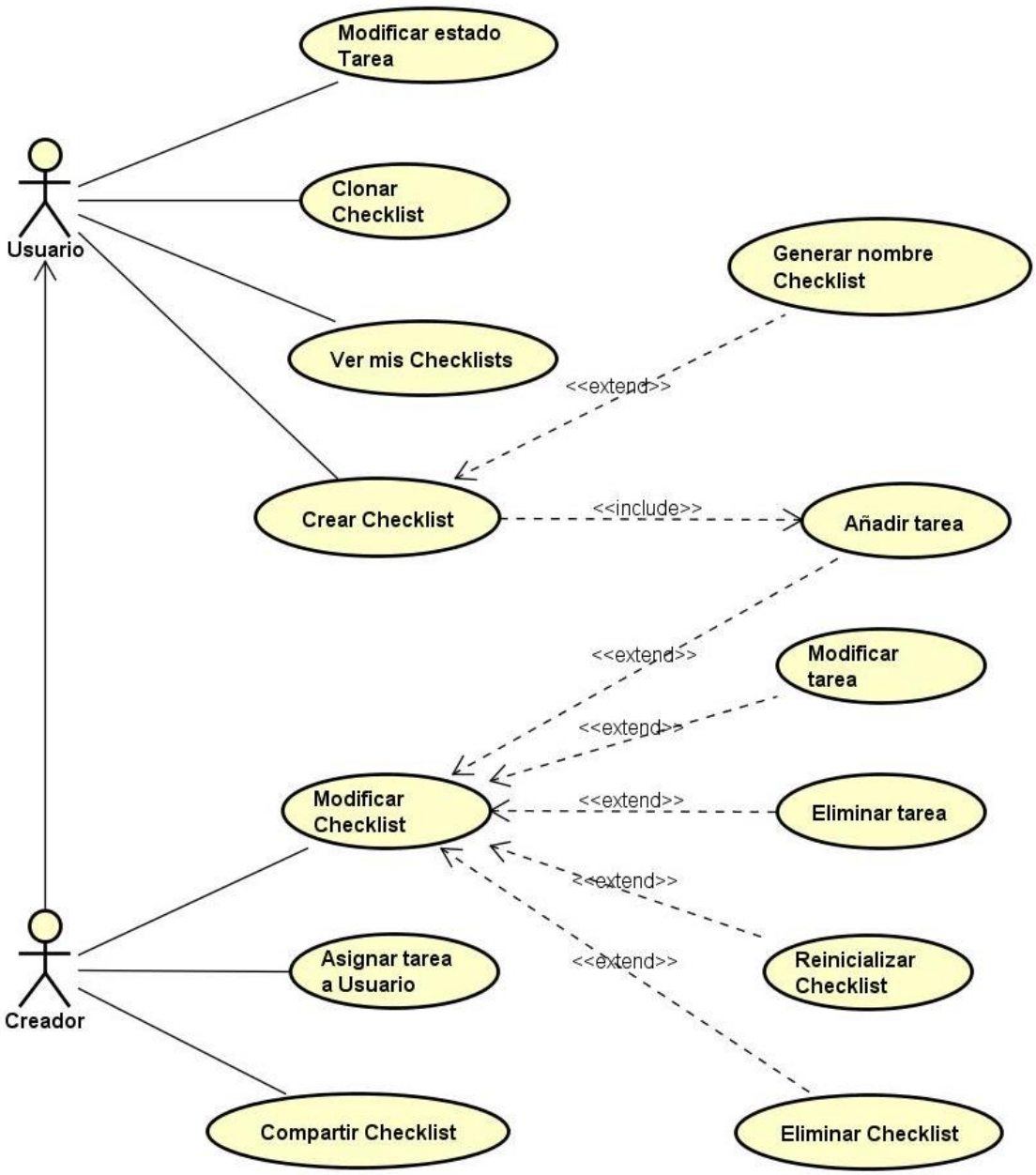

Figura 8. Diagrama de casos de uso del modelo de dominio.

Tal y como se puede observar en la *figura 8*, cualquier usuario puede modificar el estado de una tarea, duplicar una checklist y crear una nueva. Estos casos de uso no requieren que el usuario cumpla ninguna condición antes de ser realizados. Cuando un usuario crea una nueva checklist, se convierte automáticamente en Creador de esta. A mayores de los casos de uso mencionados, los creadores pueden realizar algunos casos de uso especiales. Los casos de uso que solo pueden ser realizados por un creador, y únicamente para su propia checklist, son: Modificar checklist, asignar una tarea a un usuario y compartir checklist. Los casos de uso: añadir tarea, modificar tarea, eliminar tarea, reinicializar checklist y eliminar checklist extienden de modificar checklist. Esto quiere decir que no siempre que se edite una checklist se van a dar todos los casos de uso, sino que se dará uno de ellos según la elección del usuario.

El caso de uso añadir tareas también es incluido por crear nueva checklist. Esta relación es diferente a la que tiene el CU añadir tarea con modificar checklist. Un *include* representa que siempre que se quiera crear una nueva lista se va a realizar el caso de uso añadir tarea. Esto se representa así porque al crear un checklist es obligatorio introducir al menos una tarea. Crear checklist extiende un segundo caso de uso que únicamente se llevará a cabo si el nombre introducido para la lista ya existe. Esto evita la duplicidad de nombres entre listas de un mismo usuario y con ello la imposibilidad de identificar a una frente a otras.

Con estos trece casos de uso cubrimos todos los requisitos funcionales presentados en el apartado anterior. El caso de uso Modificar estado de tarea, abarcaría tanto el requisito **RF-6** como el **RF-6.1.** La manera de actuar del sistema según sea una tarea asignada o no, se explica en el siguiente apartado.

Además de estos CU, existe un proceso de negocio en el cual no es necesaria la interacción del sistema con el usuario. Este proceso, desarrollado íntegramente por el sistema, es **PN-1: Mostrar Checklist**. Este es ejecutado siempre al finalizar los CU: Crear Checklist, Añadir tarea (desde el CU Modificar Checklist), Modificar tarea, Reinicializar Checklist y Clonar Checklist. También puede ser ejecutado dependiendo del resultado obtenido al terminar los CU: Eliminar tarea, Eliminar Checklist, Compartir Checklist y Ver mis Checklists.

Especificación de CU

CASO DE USO - 1	**Crear checklist**
Versión	1.0
Dependencias	
Precondición	El usuario debe haber iniciado una conversación con el bot mediante el envío del mensaje `/start`
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera crear una nueva checklist
Secuencia normal	1. El usuario envía el comando `/new` 2. El sistema solicita un nombre para la nueva checklist 3. El usuario proporciona un nombre 4. El sistema comprueba el nombre y lo asigna a la checklist 5. El sistema realiza el **CU-3: Añadir Tarea** y solicita la confirmación de que la checklist está finalizada 6. El usuario confirma la creación de la checklist enviando el comando `/done` 7. El sistema crea una nueva checklist identificable para el usuario a través de su nombre. Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha creado una nueva checklist con sus tareas
Secuencia alternativa	1.a. El usuario envía el comando seguido de un nombre para la lista: - El sistema avanza hasta el paso 4 4.a. El usuario ha enviado un nombre que ya tiene otra de sus listas: - El sistema realiza el **CU-13: Generar nombre Checklist no existente** y después continúa en el paso 5 6.a. El usuario envía cualquier mensaje que no sea `/done`: - El sistema regresa al paso 5
Comentarios	

CASO DE USO - 2	Modificar checklist
Versión	1.0
Dependencias	
Precondición	El mensaje con la checklist debe disponer del botón de Editar. Este botón solo se muestra al creador de la checklist dentro de su conversación privada con el bot.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera modificar una de sus checklist
Secuencia normal	1. El usuario pulsa sobre el botón de editar de una checklist 2. El sistema muestra unos botones con las opciones de edición posibles 3. El usuario selecciona la opción deseada 4. El sistema realiza el CU necesario para cumplir con la opción seleccionada
Postcondición	El usuario ha editado una de las checklists creadas por él
Secuencia alternativa	4.a. El usuario ha seleccionado “Añadir tarea”: - El sistema realiza el **CU-3: Añadir tarea** 4.b. El usuario ha seleccionado “Modificar tarea”: - El sistema realiza el **CU-4: Modificar tarea** 4.c. El usuario ha seleccionado “Eliminar tarea”: - El sistema realiza el **CU-5: Eliminar tarea** 4.d. El usuario ha seleccionado “Reinicializar checklist”: - El sistema realiza el **CU-6: Reinicializar checklist** 4.e. El usuario ha seleccionado “Eliminar checklist”: - El sistema realiza el **CU-7: Eliminar checklist**
Comentarios	

CASO DE USO - 3	Añadir tarea a checklist
Versión	1.0
Dependencias	
Precondición	El usuario debe haber comenzado la creación de una nueva checklist mediante el **CU-1: Crear Checklist** o solicitado añadir una tarea desde el **CU-2: Modificar Checklist**.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera añadir una tarea nueva a una de sus checklist
Secuencia normal	1. El sistema solicita una tarea 2. El usuario proporciona al sistema la tarea 3. El sistema añade la tarea proporcionada a la lista.
Postcondición	El usuario ha añadido una nueva tarea a una de sus checklists
Secuencia alternativa	3.a. El usuario envía una tarea ya existente dentro de la lista: - El sistema informa al usuario y regresa al paso 1 3.a. Se accede al caso de uso desde el **CU-2**: - El sistema realiza el **PN-1: Mostrar Checklist.**
Comentarios	

CASO DE USO - 4	Modificar tarea de checklist
Versión	1.0
Dependencias	
Precondición	El usuario debe haber solicitado modificar una checklist creada por él mediante el **CU-2: Modificar Checklist**
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera modificar una de las tareas de su checklist
Secuencia normal	1. El sistema muestra una lista de botones con las tareas actuales de la checklist 2. El usuario selecciona la tarea a modificar 3. El sistema solicita la nueva tarea que reemplazará a la anterior 4. El usuario envía la nueva tarea 5. El sistema modifica la tarea de la checklist. Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha reemplazado una de las tareas ya existente en la checklist por una nueva
Secuencia alternativa	3.a. El usuario envía una tarea ya existente dentro de la lista: - El sistema informa al usuario y regresa al paso 3
Comentarios	

CASO DE USO - 5	Eliminar tarea de checklist
Versión	1.0
Dependencias	
Precondición	El usuario debe haber solicitado modificar una checklist creada por él mediante el **CU-2: Modificar Checklist**
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera eliminar una de las tareas de su checklist
Secuencia normal	1. El sistema muestra una lista de botones con las tareas actuales de la checklist 2. El usuario selecciona la tarea a eliminar 3. El sistema elimina la tarea permanentemente. Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha eliminado una de las tareas de su checklist
Secuencia alternativa	3.a. El usuario ha seleccionado la única tarea que tiene la checklist: - El sistema informa al usuario y realiza el **CU-7: Eliminar Checklist**
Comentarios	

CASO DE USO - 6	Reinicializar Checklist
Versión	1.0
Dependencias	
Precondición	El usuario debe haber solicitado modificar una checklist creada por él mediante el **CU-2: Modificar Checklist**
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera reinicializar una sus checklists
Secuencia normal	1. El sistema reinicializa la checklist, es decir, el estado de todas las tareas se establece a "no hecha". Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha reinicializado una de sus checklists
Secuencia alternativa	
Comentarios	

CASO DE USO - 7	Eliminar Checklist
Versión	1.0
Dependencias	
Precondición	El usuario debe haber solicitado modificar una checklist creada por él mediante el **CU-2: Modificar Checklist**
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera eliminar una sus checklists
Secuencia normal	2. El sistema solicita confirmación de la acción 3. El usuario confirma que desea eliminar la lista pulsando el botón correspondiente 4. El sistema elimina la checklist permanentemente
Postcondición	El usuario ha eliminado una de sus checklists del sistema
Secuencia alternativa	2.a. El usuario selecciona el botón "No": - El sistema informa al usuario y el caso de uso queda sin efecto. Se realiza el **PN-1: Mostrar Checklist.**
Comentarios	

CASO DE USO - 8	Modificar estado tarea
Versión	1.0
Dependencias	
Precondición	La checklist debe haber sido compartida con el usuario o creada por él.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera modificar el estado de una de las tareas de la checklist
Secuencia normal	1. El usuario hace clic sobre una de las tareas de la checklist 2. El sistema modifica el estado de la tarea a hecha. Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha modificado el estado de una de las tareas de la checklist. Si estaba sin hacer ahora se muestra como hecha y viceversa.
Secuencia alternativa	1.a. El usuario hace clic sobre una tarea asignada que no está asignada a él: - El sistema informa al usuario y el caso de uso queda sin efecto 1.b. El usuario hace clic sobre una tarea que ya está hecha: El sistema informa al usuario y el caso de uso queda sin efecto
Comentarios	

CASO DE USO - 9	Ver mis Checklists
Versión	1.0
Dependencias	
Precondición	El usuario debe haber iniciado una conversación con el bot mediante el envío del mensaje `/start`
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario desee ver sus checklists.
Secuencia normal	1. El usuario envía el comando `/checklists` 2. El sistema muestra una lista con todas las checklists de las que el usuario es el creador. En el mensaje se puede ver el número de tareas de realizadas y el total de tareas de cada lista. Además, también incluye un comando para cada checklist con el cual ver la lista en detalle.
Postcondición	El usuario ha obtenido una lista con todas sus checklists.
Secuencia alternativa	2.a. El usuario no tiene ninguna checklist: - El sistema informa al usuario y el caso de uso queda sin efecto. 2.b. El usuario pulsa sobre alguno de los comandos recibidos dentro de la lista: - El sistema realiza el **PN-1: Mostrar Checklist**.
Comentarios	

CASO DE USO - 10	Asignar tarea a Usuario
Versión	1.0
Dependencias	
Precondición	El usuario debe ser el creador de la checklist
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera asignar una de las tareas de su checklist a otro usuario
Secuencia normal	1. El usuario envía el comando /assignTask 2. El sistema muestra una lista de botones con todas sus checklists. 3. El usuario selecciona una de las listas 4. El sistema muestra una lista de botones con las tareas de la checklist elegida 5. El usuario selecciona una de las tareas 6. El sistema muestra una lista de botones con los usuarios pertenecientes al grupo 7. El usuario selecciona el usuario deseado 8. El sistema asigna la tarea al usuario indicado
Postcondición	La tarea queda asignada a un usuario específico quien será el único que pueda modificar el estado de esa tarea.
Secuencia alternativa	1.a. El usuario envía el comando en un chat privado: - El sistema informa al usuario y el caso de uso queda sin efecto 2.a. El usuario no tiene ninguna checklist: - El sistema informa al usuario y el caso de uso queda sin efecto 6.a. El sistema no tiene almacenados todos los usuarios pertenecientes al grupo: - El sistema informa al usuario de que puede añadir a un usuario concreto introduciendo una “@” y seleccionando al deseado. El nuevo usuario es almacenado por el sistema dentro del grupo para posteriores ocasiones. Después se realiza el paso 8
Comentarios	

CASO DE USO - 11	Compartir Checklist
Versión	1.0
Dependencias	
Precondición	El mensaje con la checklist debe disponer del botón de Compartir. Este botón solo se muestra al creador de la checklist dentro de su conversación privada con el bot.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera compartir una de sus checklists
Secuencia normal	1. El usuario pulsa sobre el botón de compartir de una checklist. 2. El sistema muestra la pantalla principal de Telegram con los grupos/contactos del usuario. 3. El usuario selecciona uno de los chats. 4. El sistema le redirige al chat seleccionado y un mensaje preparado para enviar con el nombre de la lista. 5. El usuario envía el mensaje 6. El sistema realiza el **PN-1: Mostrar Checklist**
Postcondición	El usuario ha compartido una de sus checklist con un grupo
Secuencia alternativa	3.a. El usuario da atrás: - El caso de uso queda sin efecto.
Comentarios	

CASO DE USO - 12	Clonar Checklist
Versión	1.0
Dependencias	
Precondición	La checklist que se desea clonar debe haber sido compartida con el usuario.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario requiera clonar una checklist
Secuencia normal	1. El usuario pulsa sobre el botón de clonar de una checklist. 2. El sistema comprueba si el nombre de la checklist que se quiere clonar existe entre las checklists del usuario 3. El sistema crea una nueva checklist replica de la primera, pero con las tareas sin realizar y almacena como propietario al usuario *clonador* de la lista. Realiza el **PN-1: Mostrar Checklist.**
Postcondición	El usuario ha clonado una checklist de otro usuario
Secuencia alternativa	2.a. El nombre ya pertenece a una de sus checklists: - El sistema informa al usuario y el caso de uso queda sin efecto
Comentarios	

CASO DE USO - 13	Generar nombre Checklist no existente
Versión	1.0
Dependencias	
Precondición	El usuario debe haber comenzado la creación de una nueva checklist mediante el **CU-1: Crear Checklist.** Además, el nombre con el que se iba a crear la lista ya existía entre sus listas.
Descripción	*El sistema deberá comportarse como se describe en el siguiente caso de uso cuando* un usuario haya introducido un nombre ya existente en el sistema y el sistema necesite generar uno nuevo y único
Secuencia normal	3. El sistema genera tres nuevos nombres compuestos por el nombre introducido por el usuario durante la realización de la precondición y una sucesión de números y caracteres especiales y muestra las tres alternativas mediante botones 4. El usuario selecciona uno de los nombres disponibles 5. El sistema asigna el nombre a la checklist
Postcondición	El usuario ha proporcionado un nombre único a la checklist que está creando
Secuencia alternativa	
Comentarios	

PROCESO DE NEGOCIO - 1	Mostrar Checklist
Versión	1.0
Dependencias	
Precondición	El sistema debe haber solicitado la realización del proceso tras la finalización de unos de los siguientes casos de uso: **CU-1, CU-3, CU-4, CU-5, CU-6, CU-7, CU-8, CU-10** o **CU-12**
Descripción	*El sistema deberá comportarse como se describe en el siguiente proceso de negocio cuando* el sistema necesite generar el mensaje con la checklist que se enviará al usuario
Secuencia normal	1. El sistema recibe algunos valores para saber a qué chat/s debe enviar el mensaje y qué lista debe generar, además de algunos modos de envío. 2. El sistema genera un mensaje con la checklist asociada al identificador. En este mensaje se muestra: la última modificación realizada sobre la checklist, el detalle de la checklist y una lista de botones con las tareas de la checklist. Además, unos botones de opciones adicionales solo visibles en el chat del creador. 3. El sistema envía o edita uno o varios mensajes dependiendo del modo de envío pasado como valor de entrada.
Postcondición	El sistema ha generado un mensaje con la correspondiente checklist y lo ha enviado de la forma solicitada al usuario.
Secuencia alternativa	
Comentarios	

Diagrama de clases

El diagrama de clases presentado a continuación modela el dominio del problema. En él representamos los objetos del mundo real que modelan el dominio en estudio. La representación del modelo de dominio, se realiza con un diagrama de clases UML. Cada clase representa una entidad conceptual. Estas clases tienen los atributos relevantes para su propia definición. También se muestra en este diagrama las relaciones entre las clases. En el modelo de dominio, no se incluye ninguna operación ya que esto es propio de la etapa de diseño. Las operaciones de cada clase no influyen en la representación del modelo del dominio de interés. La descripción de las operaciones de las que dispone cada clase se expondrán en el Capítulo **Diseño**.

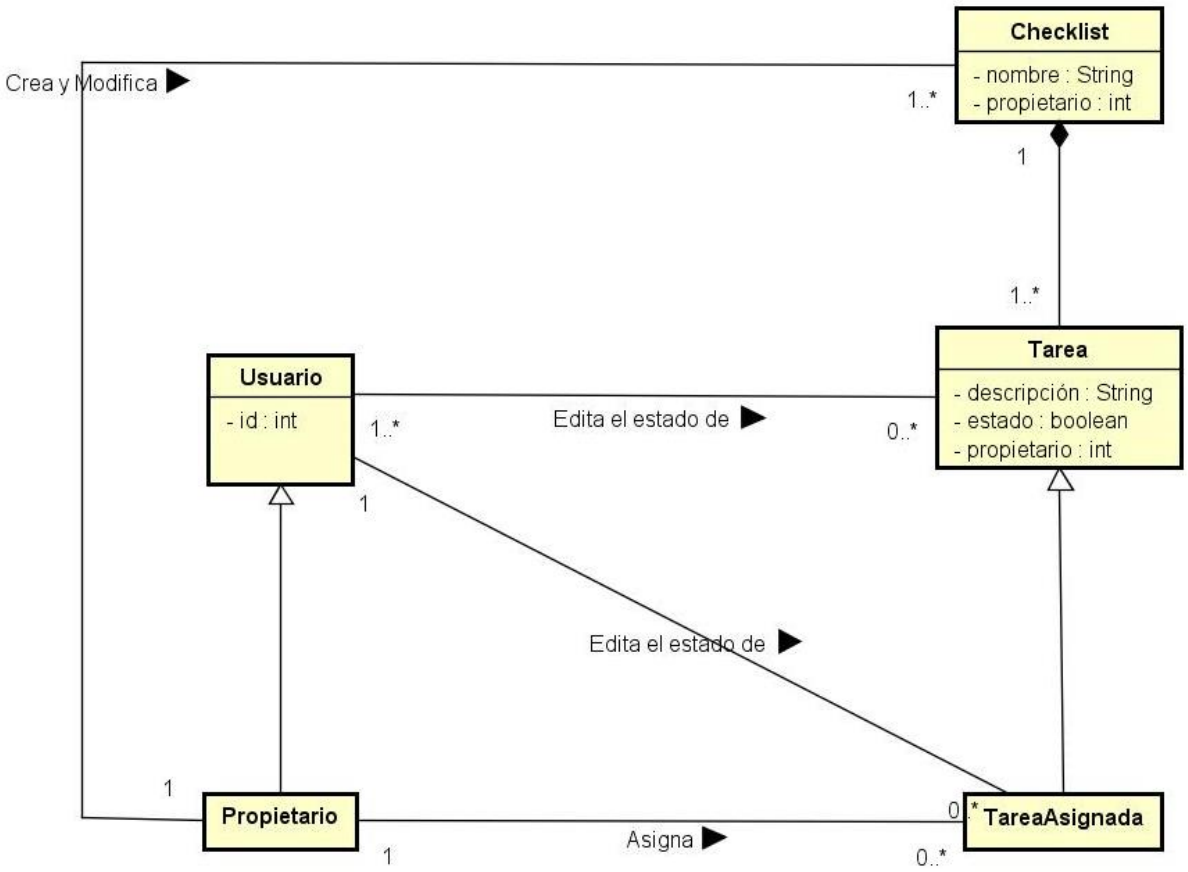

Figura 9. Diagrama de clases del modelo de dominio.

Como se puede observar en la figura 9, se ha modelado el dominio con una serie de entidades. Estas entidades están relacionadas entre ellas de diferentes maneras. Procedemos a explicar en detalle cada clase:

Usuario

Representa a un usuario estándar de la aplicación desarrollada. Sus atributos son:

- **Id:** Identificador unívoco que nos permite distinguir a un usuario específico del resto. Este atributo se corresponde con el identificador del chat que Telegram asigna a cada usuario.

Sus relaciones con el resto de entidades son:

- **Tarea:** Un usuario puede editar el estado de ninguna o muchas tareas diferentes.

- **TareaAsignada:** Un usuario puede editar el estado de ninguna o muchas tareas asignadas diferentes. Está relación tiene como restricción que el usuario debe ser el "usuario asignado" a esa tarea para poder editarla.

Propietario

Representa a un usuario que ha creado alguna lista en el sistema. Cuando un usuario crea una lista, se convierte en propietario de esta instantáneamente. Sus atributos son:

- Los mismos que los de Usuario

Sus relaciones con el resto de entidades son:

- **Usuario:** Propietario hereda de la clase Usuario sus atributos.
- **Checklist:** Un propietario puede crear y modificar una o más checklists. El número concreto de checklists que puede modificar dependerá del número de listas de las que sea propietario.
- **TareaAsignada:** Un propietario puede asignar desde ninguna a muchas tareas distintas a usuarios.

Checklist

Representa una checklist dentro del sistema. Sus atributos, los cuales coinciden con algunos de los descritos en el apartado de Requisitos de información, son:

- **Nombre:** Nombre de la checklist. En un formato entendible para los usuarios y sirve para identificar a la lista por parte de estos. Entre las checklists de un mismo usuario debe ser único.
- **Propietario:** Usuario creador de la checklist. Para poder identificar unívocamente al usuario, se almacena el correspondiente id.

Sus relaciones con el resto de entidades son:

- **Propietario:** Una checklist es creada y modificada por únicamente un Propietario. Este propietario debe coincidir con el usuario del atributo detallado.
- **Tarea:** Una checklist es una composición de tareas. Sin tareas no existe checklist. Si borramos todas las tareas de una checklist del sistema, la checklist también es eliminada. Una checklist debe tener siempre una tarea al menos.

Tarea

Representa una tarea "estándar" dentro del sistema. Sus atributos, coinciden exactamente con los descritos en el apartado de Requisitos de información:

- **Descripción:** Nombre unívoco que identifica a la tarea. Es una pequeña descripción de esta y sirve al usuario para diferenciarla del resto. Este atributo no se puede repetir dentro de tareas de una misma checklist.
- **Estado:** Estado en el que se encuentra la tarea. Puede ser: "Done" o "Not done" (Hecha/No hecha).
- **Propietario:** Usuario al cual ha sido asignado la tarea. Para poder identificar unívocamente al usuario, se almacena el correspondiente id. Este atributo siempre va a estar vacío en las tareas "estándar".

Sus relaciones con el resto de entidades son:

- **Checklist:** Una tarea siempre pertenece a una única checklist.
- **Usuario:** En una tarea “estándar” pueden editar su estado muchos usuarios sin restricción ninguna.

TareaAsignada

Representa una tarea asignada a un usuario dentro del sistema. Sus atributos son:

- Los mismos que los de Tarea. Sin embargo, en TareaAsignada el atributo Propietario siempre va a contener el id de un usuario.

Sus relaciones con el resto de entidades son:

- **Tarea:** TareaAsignada hereda de la clase Tarea sus atributos.
- **Usuario:** En una tarea asignada puede editar su estado únicamente un usuario, el usuario al cual ha sido asignada.
- **Propietario:** Una tarea asignada puede ser asignada por solo un propietario, el propietario de la checklist a la cual pertenece la tarea.

Diagrama de actividades

Los diagramas de actividades son representaciones gráficas de los *workflows (flujos de trabajo).* A través de ellos se pueden mostrar decisiones, iteraciones y concurrencia. Con UML se puede presentar en el diagrama de actividades tanto operaciones computacionales como procesos organizacionales. Comprendiendo estos diagramas, se puede entender el flujo que sigue el programa ya que en ellos representa el orden en el cual las actividades ocurren. Los diagramas de actividades tienen especial interés para explicar las actividades que se producen de manera concurrente en un programa. En este caso, el principal objetivo de los diagramas que se presentan en este apartado es analizar los casos de uso más interesantes. Estos diagramas están englobados dentro de la etapa de análisis porque no se centran en asignar acciones a los objetos, se centran en plasmar qué acciones deben ocurrir y cuáles son sus dependencias [32].

Están construidos con un número de formas conectadas mediante flechas. Las formas que se pueden encontrar en los diagramas mostrados a continuación son:

- Acciones: Rectángulos redondeados
- Decisiones: Rombos
- Inicio/fin de actividades concurrentes: Barras negras
- Inicio del flujo explicado: Círculo negro
- Fin del flujo explicado: Circunferencia con círculo negro en el centro

En este apartado, se van a explicar dos diagramas de actividades. El primero representa el *workflow* del caso de uso **Crear Checklist** y el segundo muestra el *workflow* CU **Editar Checklist.** En ambos se van a contemplar los flujos alternativos a mayores de la secuencia normal. Se han escogido estos casos de uso por ser la base del programa, junto con hacer "check" en una tarea. No se ha incluido el diagrama de actividades de este último caso de uso por no tener una mínima complejidad.

Antes de comenzar la descripción de los diagramas de actividades mencionados, es importante resaltar que en estos no se ha representado la interacción con Telegram. En todos los diagramas debería existir el servidor de Telegram como intermediario entre el bot y el usuario. Es el servidor de Telegram a quien el usuario envía los mensajes al establecer una conversación con el bot. Todos los mensajes que un usuario envía y recibe del bot llevan en la petición https el token de la aplicación @Checklisttfg_bot. De esta forma Telegram puede redireccionar los mensajes enviados por el usuario a la aplicación desarrollada y enviar la respuesta proporcionada por el bot al primer mensaje al usuario emisor de este.

En la figura 11 se empieza explicando el más sencillo de ambos casos de uso: Crear Checklist. El flujo inicial empieza cuando un usuario solicita crear una nueva checklist. El sistema comprueba si el comando se ha enviado sólo o seguido de un nombre para la lista. Si se ha enviado sólo solicita el nombre. En el momento que el bot dispone de un nombre para la nueva checklist, lo comprueba. Si el nombre ya existe entre las listas del usuario, genera tres nuevos nombres a partir del proporcionado. Si el nombre no existe, crea un objeto Checklist y solicita la primera tarea. Una vez el usuario introduce la primera tarea, el bot solicita una más. Dependiendo de lo que el usuario desee, añadir más o no, se finaliza el flujo o se siguen procesando tareas. Si el usuario decide añadir otra tarea, el sistema comprueba que esta no esté repetida dentro de la lista. Si no es así, crea un objeto Task y lo almacena en el XML. Si por el contrario ya existe una tarea con ese nombre en la lista, solicita otra tarea. Cuando el usuario decida no añadir más tareas, el objeto Checklist anteriormente creado es almacenado en XML.

La actividad de añadir la primera tarea se muestra separada del resto porque la primera nunca se comprueba y siempre se añade por ser la única de la lista y no poder existir otra con mismo nombre. Se realizan varias comprobaciones antes de crear la checklist para evitar duplicidades que más tarde dan lugar a errores en la aplicación.

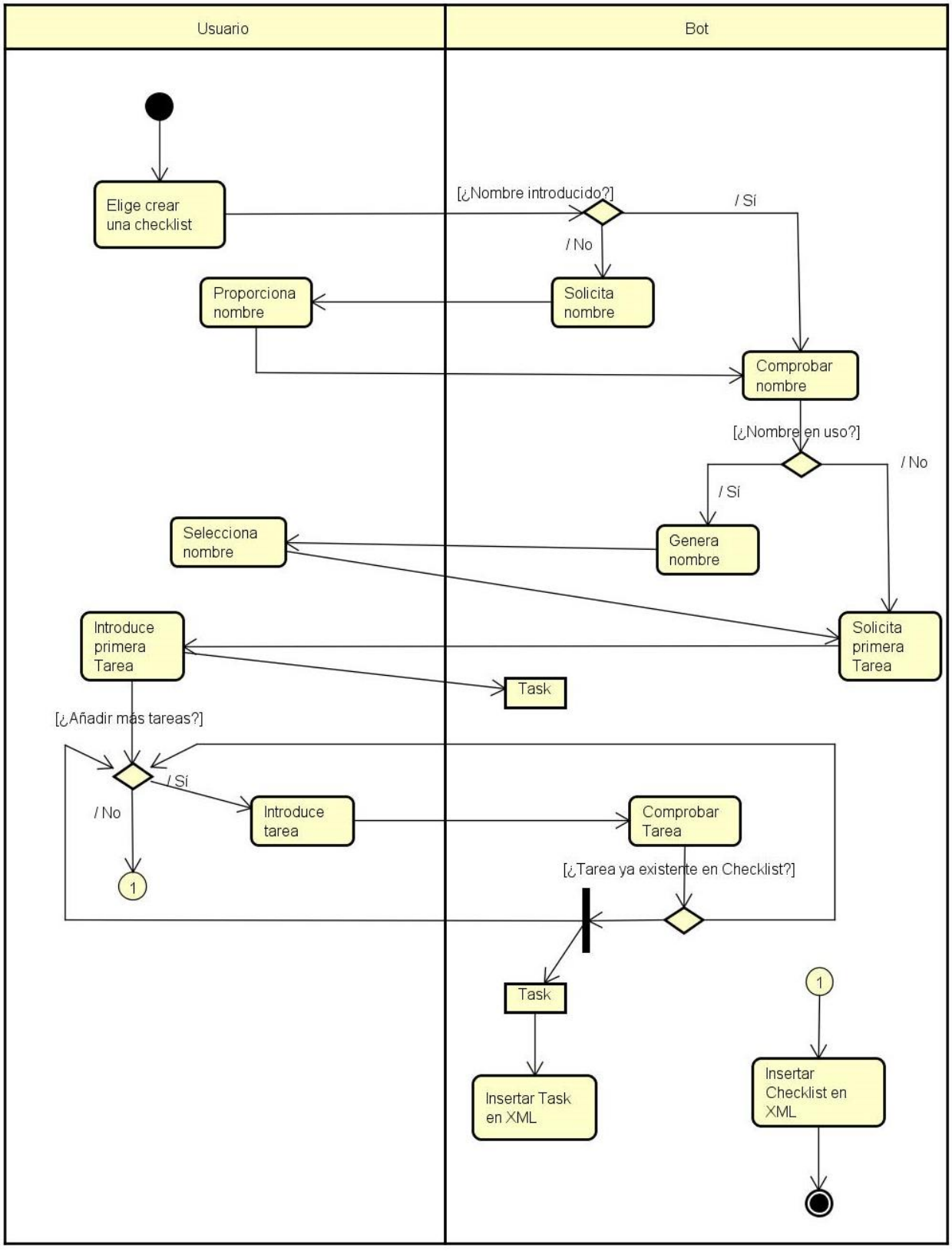

Figura 10. Diagrama de actividades del **CU-1: Crear Checklist**.

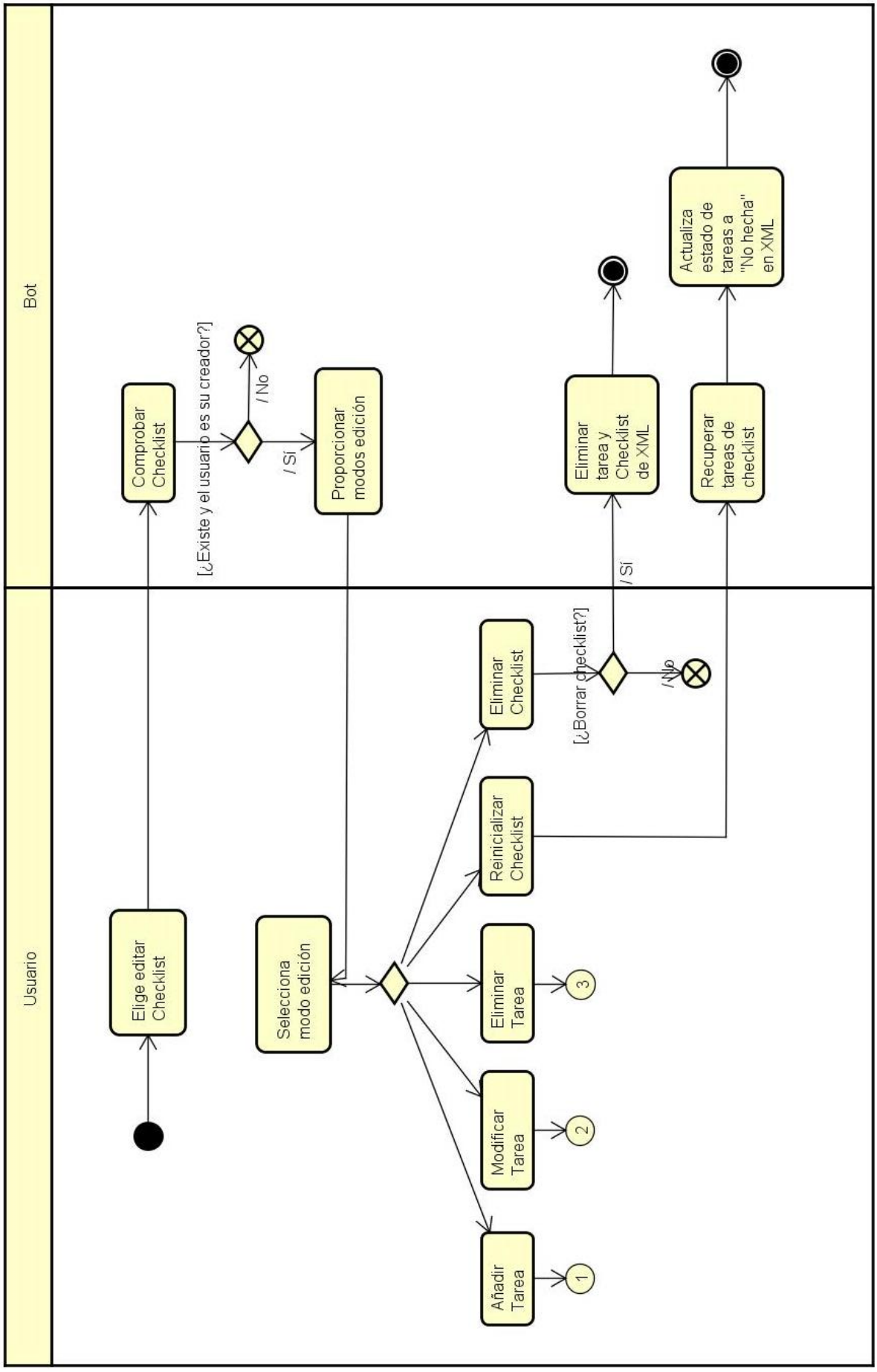

Figura 11. Diagrama de actividades del **CU-2: Editar Checklist**.

En el segundo diagrama de actividades, figura 11, se describe el flujo de control si un usuario decide editar una de sus checklists. El inicio del flujo comienza con una actividad del usuario. En esta primera actividad el usuario decide editar una checklist, en el caso concreto en estudio el usuario pulsa el botón Editar de una checklist. El bot comprueba que la lista existe en sistema y es del usuario que ha iniciado la acción, para evitar incongruencias, y muestra los modos de edición si cumple con los requisitos citados. Si, por el contrario, la checklist no existe en el sistema o el usuario no es su creador el flujo finaliza sin efecto. El usuario selecciona uno de los modos de edición proporcionados por el bot. Los casos de uso: añadir tarea, modificar tarea y eliminar tarea se describen en diagramas aparte para simplificar las figuras. Si el usuario selecciona **Reinicializar Checklist**, el sistema recupera las tareas asociadas a la lista que se está editando y modifica en el XML el estado de cada una de ellas a "Sin hacer". Si el usuario pulsa en el botón de **Eliminar Checklist** el sistema le pide una confirmación y el usuario toma la decisión que prefiera. Si confirma borrar la lista, esta es eliminada del XML correspondiente. Si cancela la acción, el flujo termina sin efecto.

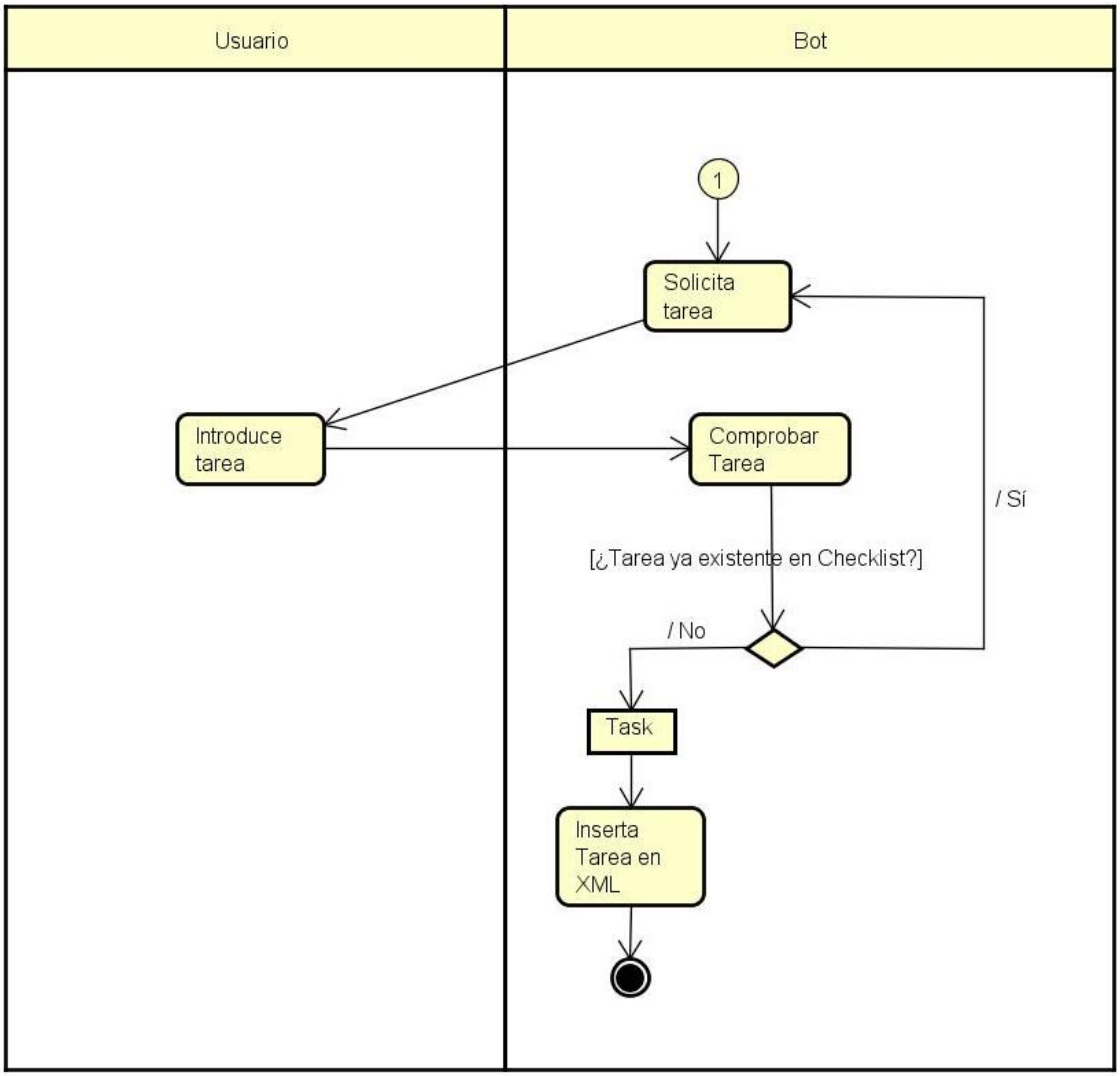

Figura 12. Diagrama de actividades del **CU-3: Añadir tarea**.

El diagrama de actividades de la imagen 12 muestra el flujo entre el usuario y la aplicación en el caso de que el usuario decida añadir una nueva tarea a la checklist. En este escenario, el bot solicita que se envíe la nueva tarea, el usuario se la proporciona y el bot comprueba la tarea proporcionada. Si ya existe una tarea con el mismo nombre en la lista, vuelve a solicitar la tarea. Mientras no se proporcione un nombre no duplicado dentro de la lista, el bot pide una tarea. Una vez el usuario

introduzca una tarea no repetida dentro de la lista, crea una tarea, almacena su información en el XML correspondiente y finaliza el *workflow.*

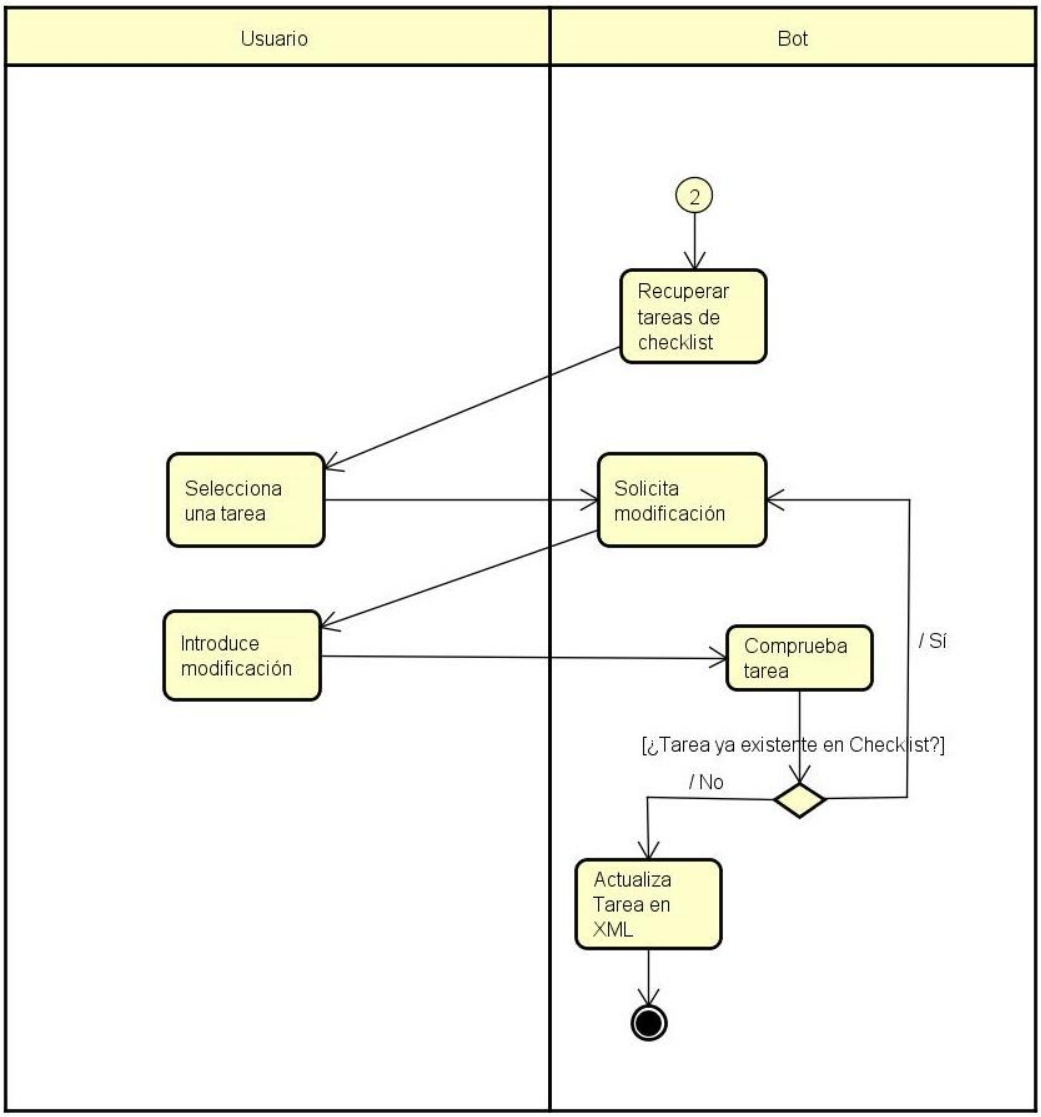

Figura 13. Diagrama de actividades del **CU-4: Modificar tarea**.

En el diagrama superior se puede ver el flujo de actividades entre ambos para la realización del caso de uso modificar tarea. El sistema recupera todas las tareas de la lista y se las muestra al usuario. Entre estas, el usuario selecciona una. La tarea seleccionada será la que se va a proceder a modificar. El bot pide la tarea que sustituirá a la seleccionada y el usuario introduce una nueva. El sistema comprueba que no existe ya una tarea en la checklist idéntica a la proporcionada. Esta comprobación se realiza siempre que se vaya a proceder a añadir una tarea a la lista porque una tarea es identificada dentro de una lista únicamente por su nombre. Si dos o más tareas tuvieran el mismo nombre (dentro de una misma checklist) el bot las trataría de la misma manera. Esto provocaría que al hacer *check* en una se hiciera *check* en las de exacto nombre. El bot accede al mismo bucle que en el diagrama anterior si la tarea ya existe en la lista. Cuando el usuario proporciona una tarea no existente, actualiza el XML añadiéndola y eliminando la reemplazada y finaliza el flujo comenzado en la figura 12.

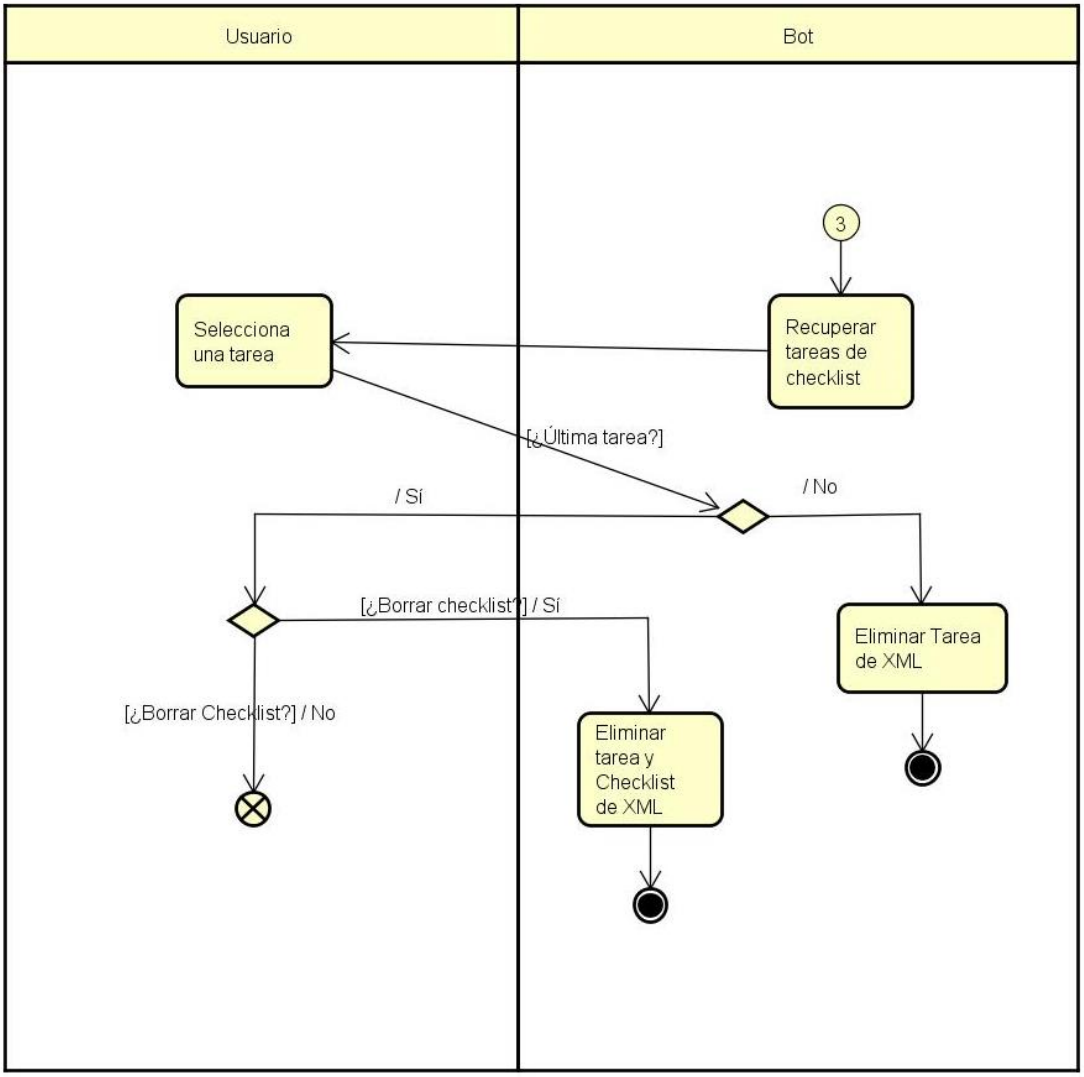

Figura 14. Diagrama de actividades del **CU-5: Eliminar tarea**.

Por último, en la figura 14 se puede observar el flujo respectivo a la realización del caso de uso **Eliminar tarea.** Este diagrama comienza de forma análoga al anterior. El bot proporciona una lista con todas las tareas pertenecientes a la checklist y el usuario selecciona una. Dependiendo de si la tarea es la última de la lista o no, se borra directamente o se pide una confirmación por parte del usuario. Si no es la única tarea de la lista se actualiza el XML borrando la información almacenada referente a la tarea. Si es la última, se pide al usuario una confirmación porque eliminar esta tarea implica borrar también la lista del sistema. Esto es así porque la relación checklist-task es una composición. Su relación de dependencia es fuerte y no puede existir una checklist sin ninguna tarea asociada.

Capítulo 4. Diseño

Durante la etapa de análisis se ha realizado un análisis de cómo debe comportarse la aplicación ante el usuario. En este capítulo se comienzan a fijar detalles más técnicos y relacionados con la implementación del *bot*. En la etapa de diseño se tiene como objetivo el estudio de los componentes necesarios para cubrir con las funcionalidades descritas en la etapa anterior. Se realiza un nuevo diagrama de clases con las clases que finalmente se necesitarán en la aplicación para cumplir con todos los casos de uso. En esta etapa también se especifican las operaciones y atributos del sistema. Es recomendable explicar en detalle el sistema para ahorrar trabajo al programador en la siguiente etapa. Es imprescindible realizar diagramas claros y limpios para facilitar su entendimiento.

Diagrama de clases

El patrón de arquitectura seguido es **MVC (Modelo-Vista-Controlador).** MVC separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario. Para llevar a cabo esta separación, la aplicación se divide en clases pertenecientes a paquetes distintos según la funcionalidad que aportan. Estos paquetes son: modelo, vista y controlador. El objetivo de este patrón es distinguir entre los datos de la aplicación y la interacción con el usuario. En la aplicación @Checklisttfg_bot se ha dividido el controlador en varias clases. El controlador de vista recoge los datos introducidos por el usuario a través de la vista, y encarga a un controlador concreto el procesamiento de estos. El controlador realiza las operaciones necesarias para obtener una respuesta a la petición del usuario. Para llegar al resultado interactúa con las clases del modelo y los gestores de datos. Una vez obtiene el resultado a la petición del usuario, lo devuelve al controlador de vista y este lo envía para que lo muestre la vista [33].

No se ha implementado una capa de vista. Es Telegram quien se encarga de presentar los datos al usuario. El desarrollador de un bot no tiene ninguna decisión acerca de cómo presentar los datos al usuario en cuanto a estilos. Puede utilizar los métodos proporcionados por el framework para elegir entre los diferentes componentes. El desarrollador sí puede decidir que la respuesta al usuario se envíe mediante un mensaje, botones o popups, pero no puede fijar el color de estos, el tipo o tamaño de letra o el estilo del mensaje.

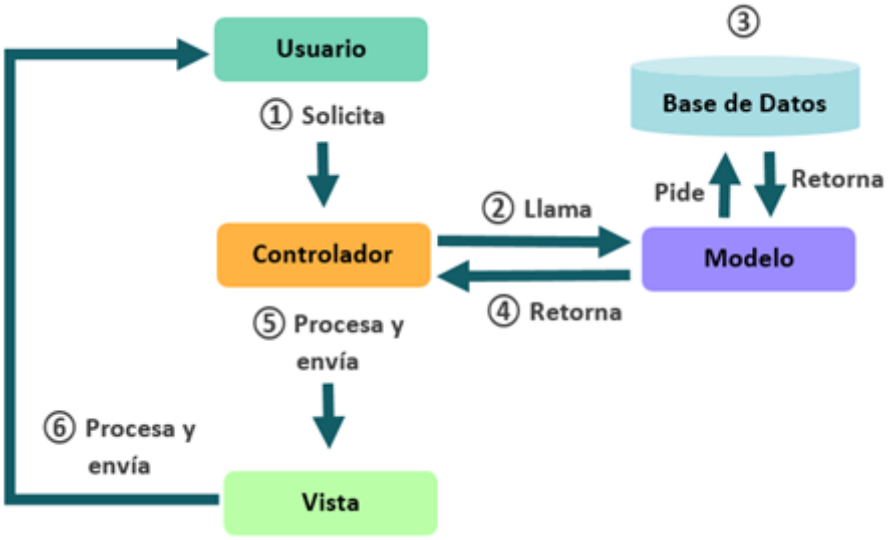

Figura 15. Esquema funcionamiento MVC.

En MVC se separan las clases del modelo en el paquete correspondiente, de las clases de la capa de persistencia (acceso a base de datos) y de la lógica de negocio (capa de controladores).

Se dispone de un controlador de vista, el cual es el encargado de recoger las peticiones de usuario, enviarlas al controlador asociado a esa petición y enviar el resultado obtenido a "la vista" *(Telegram)* para que lo muestre al usuario. La comunicación con el servidor de Telegram se realiza única y exclusivamente a través de este controlador como se puede observar en la figura 16.

Contando con el controlador de vista, existen siete controladores dentro de la aplicación. Inicialmente se pensó en implementar un controlador por cada caso de uso. Debido a la escasez de complejidad de algunos de ellos se optó por agruparlos. Estos cinco controladores son los encargados de toda la lógica de negocio del programa y heredan de la clase **Controller.** Esta clase sigue el patrón de diseño *singleton*, el cual heredan el resto. El patrón *singleton* será explicado en posteriores apartados. Los gestores también cumplen con este patrón de diseño.

En la capa de persistencia se encuentran los gestores de datos. Estas clases son las encargadas del acceso a los datos persistentes de la aplicación. Cada uno de ellos controla un XML. En estas clases se implementan las operaciones necesarias para simular el comportamiento de sentencias SQL. Debido a esto es en las clases *manager* donde se deben localizar los métodos que cubren las operaciones CRUD. CRUD es el acrónimo de *Create, Read, Update* y *Delete*. Estas son las funciones básicas de acceso a base de datos.

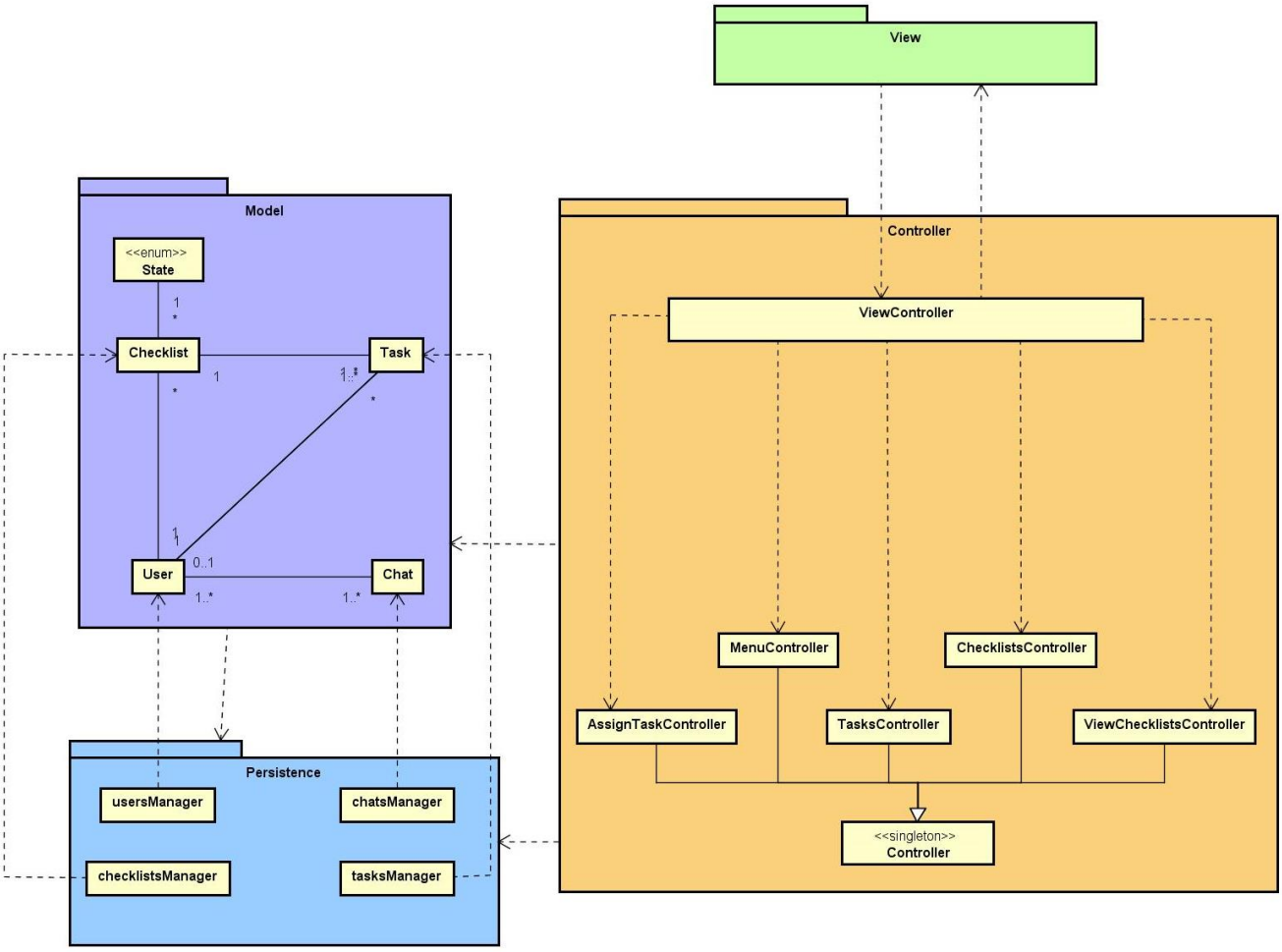

Figura 16. Diagrama de clases de Diseño.

Diagrama detallado del modelo

Partiendo del diagrama de clases UML presentado en el Capítulo de Análisis se ha construido el siguiente diagrama de clases del modelo para el diseño de la aplicación.

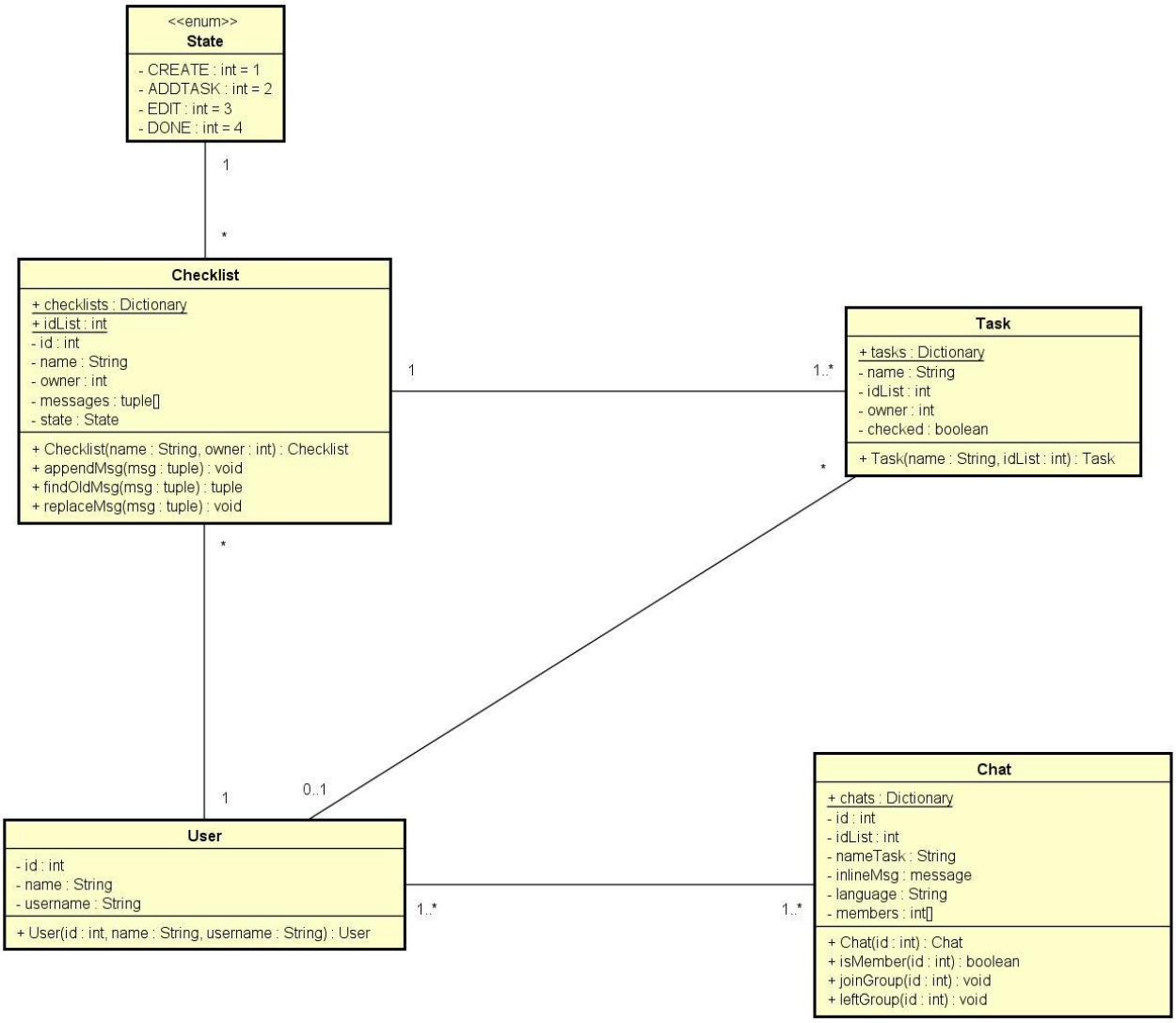

Figura 17. Diagrama detallado de clases del modelo.

Lo primero que destaca si comparamos el diagrama de clases presentado en Análisis con el de Diseño es que han desaparecido algunas clases y han aparecido otras nuevas. Se prescinde de las clases: *Propietario* y *TareaAsignada*. Las nuevas clases son: *Chats y State*. Estas clases no habían sido descubiertas durante la etapa de análisis debido a que solo son necesarias debido a la forma escogida para resolver el problema. En las siguientes páginas se da una explicación detallada del diagrama de clases y una exposición de las razones por las cuales se ha implementado de esta manera el dominio. Es importante informar al lector de que en el citado diagrama de la figura 17 no aparecen los getters y setters de las clases para facilitar su lectura.

Checklist

La clase Checklist tiene dos atributos estáticos. Un atributo estático pertenece a la propia clase y no a las instancias que creamos de ella. No se puede acceder a ellos desde un objeto, se debe acceder a estos desde la propia clase. Sus atributos estáticos son:

- **idList:** Atributo de tipo entero que se usa para dar un valor identificador diferente a cada checklist. Actúa como contador y es incrementado cada vez que se asigna un valor a una instancia de Checklist. De esta forma se evita la duplicidad en el identificador de las checklists.

- **Checklists:** Es de tipo Diccionario y almacena los objetos Checklist mientras esten en estado *CREATE*. Al finalizar la creación de una checklist, y pasar esta a estado *DONE*, se elimina del diccionario. Tiene por *key* el identificador de la checklist.

El resto de atributos pertenecen a cada objeto en particular. A los mencionados en el capítulo de **Análisis** se suman: *id, messages* y *state.* En id se guarda un identificador de la lista y en state el estado de "creación" en el cual se encuentra la lista. Messages es una lista de tuplas[1] con los identificadores de los mensajes de Telegram en los cuales se ha mostrado por última vez la checklist. Se guarda una tupla nueva por cada chat. De esta forma se tienen localizados los mensajes sobre los cuales mostrar toda las funcionalidades disponibles para cada checklist.

Cada instancia dispone de unos métodos propios. Además de los getters y setters de cada atributo, cada objeto de tipo checklist tiene las siguientes operaciones:

- **Relacionadas con el atributo messages:** appendMsg, findOldMsg y replaceMsg. Estas operaciones permiten añadir una nuevo mensaje a la lista de mensajes, buscar el mensaje entre los mensajes de la checklist que tiene el mismo identificador de chat que el mensaje que pasamos como parámetro de entrada y remplazar un mensaje por otro.

Sus relaciones con el resto de clases son:

- **Tarea:** Una checklist tiene al menos una tarea siempre.
- **State:** Una checklist tiene un único estado asociado.
- **User:** Una checklist pertenece a un único usuario, su creador.

Task

La clase Task se mantiene prácticamente idéntica a la presentada en el diagrama de análisis del modelo del dominio. Mantiene los mismos atributos, pero añade uno nuevo: *idList*. Este almacena el identificador de la lista a la cual pertenece la tarea. El atributo *name* es el que en el capítulo de análisis se presentaba como *descripción*. Sus operaciones se reducen a los getters y setters necesarios para acceder a los atributos. Task tiene un atributo público y estático en el que se guardan todos los objetos tasks que se van creando: **Tasks**. Es un diccionario que tiene por clave el id de la checklist y guarda por valor una lista con las tareas pertenecientes a esa lista. Una vez se dé por finalizado el proceso de creación de la checklist, la lista de tareas del valor relacionado al identificador de la lista se elimina.

Sus relaciones con el resto de clases son:

- **Checklist:** Una tarea pertenece a una única checklist siempre.

[1] Tupla. Estructura de datos de Python que representa una colección de objetos, pudiendo estos ser de distintos tipos. Es un tipo de dato inmutable, no se puede asignar un valor a través del índice,

- **User:** Una tarea puede tener asignado uno o ningún usuario para realizarla.

User

La clase User representa lo mismo que representaba en el diagrama de clases del Análisis, un usuario dentro de la aplicación. Aparte del atributo id, el cual ya mencionábamos en ese diagrama, tiene los atributos *name* y *username*. Se ha decidido añadir estos campos para almacenar un nombre/alias legible que mostrar al usuario en los mensajes que envía la aplicación durante el uso del bot. Se guardan ambos valores y no solo uno de los dos porque (1) hay usuarios que no disponen de alias y (2) el nombre puede ser el mismo para dos usuarios mientras que el alias es unívoco. Por estas razones almacenamos los dos valores y la aplicación juega con ambos a la hora de enviar los mensajes. Sus operaciones se reducen a los getters necesarios para acceder a los atributos.

Sus relaciones con el resto de clases son:

- **Checklist:** Un usuario dispone de ninguna o muchas checklists creadas.
- **Task:** Un usuario tiene asignadas para realizar ninguna o muchas tareas.
- **Chat:** Un usuario puede pertenecer a muchos chats, pero obligatoriamiente al menos a uno: el chat privado con el bot.

Chat

La clase Chat se crea para cubrir la necesidad de conseguir una coherencia entre los mensajes enviados por el bot. Todos los atributos de los que dispone ayudan al cumplimiento de esta función y son: *id, idList, nameTask, inlineMsg, language* y *members.*

- **id:** identificador del chat. Telegram asigna un número que identifica unívocamente a cada chat. Este número es negativo en el caso de los chats grupales y positivo en los privados.
- **idList:** identificador de la lista sobre la cual está realizando acciones el usuario.
- **nameTask:** nombre de la lista sobre el cual el usuario quiere realizar modificaciones. Este nombre combinado con el id de la lista nos permite identificar la instancia de task concreta.
- **inlineMsg:** mensaje sobre el cual mostrar los mensajes que envía el bot. Se usa para poder editar el mensaje original en vez de enviar uno nuevo en casa paso. Esto evita un spam innecesario por parte del bot en múltiples ocasiones.
- **language:** lenguaje seleccionado por un usuario dentro del chat. El bot usa este atributo para enviar los mensajes acordes al lenguaje solicitado. Por defecto el valor es 'EN' (English).
- **members:** lista con todos los identificadores de usuario de los miembros de un chat. Este valor estará vacío en chats privados. Es necesario porque la API Bot de Telegram no dispone de ningún método para obtener los miembros de un grupo. Por esta razón este atributo es indispensable para realizar por ejemplo el caso de uso **Asignar tarea**.

Además, tiene un atributo público y estático en el que se guardan todos los objetos chats que se van creando. Este atributo de clase es **Chats.** Es un diccionario que tiene por clave el id de cada chat y guarda por valor el objeto chat con dicho id.

La clase chat dispone de los correspondientes getters y setters asociados a estos atributos. Estos nos permiten acceder a los atributos de un objeto concreto de tipo chat desde fuera de él. A mayores de estos métodos, Chat tiene algunos métodos relacionados con el atributo *members.*

- **isMember:** Devuelve True/False dependiendo de si el id pasado como parámetro de entrada pertenece a la lista de miembros del objeto o no. Es indispensable para no añadir más de una

vez al mismo usuario al chat ni eliminar un usuario de la lista de miembros que no pertenece a esta.

- **joinGroup:** Añade el id introducido como parámetro de entrada a la lista de miembros del chat grupal.
- **leftGroup:** Elimina el id pasado como parámetro de entrada de la lista de miembros del grupo.

Sus relaciones con el resto de clases son:

- **User:** Un chat tiene como miembros a uno o más usuarios. Siempre hay uno al menos porque:

    - En los chats privados al iniciar la conversación con el bot, este crea un chat y un usuario (si este no existía ya) relacionados. Los miembros del chat privado son el propio usuario. Es decir, coinciden identificador de chat y usuario.
    - En los chats grupales, el bot no crea el objeto chat hasta que un usuario le habla. En ese momento el objeto chat se crea con un miembro usuario relacionado.

State

Clase que hereda de Enum. Tiene como atributos de clase los 4 estados en los cuales puede estar una lista. Estos estados son:

- **Create:** Representa una lista recién creada. Desde el momento en el que se proporciona un nombre para crearla hasta el momento que se informa de que se ha acabado de añadir tareas.
- **Addtask:** Representa el estado en el cual se ha solicitado añadir una tarea a una lista que ya estaba creada.
- **Edit:** Representa el estado en el cual se encuentra una lista cuando se solicita modificar el nombre de una tarea.
- **Done:** Representa cualquier otro estado en el que se encuentre una lista que ya ha sido creada.

Estos estados son imprescindibles para que el bot sepa cómo actuar ante los mensajes que le llegan del usuario. Sin contar con los comandos, durante el uso del bot, un usuario solo necesita enviar mensajes cuando: (1) proporciona el nombre del bot, (2) envía nuevas tareas y (3) envía la modificación de una tarea. Estos mensajes pueden contener cualquier texto/emoji y extensión por lo que no es posible para el bot reconocer cómo debe reaccionar frente a estos sin la ayuda de los estados descritos. Sin la clase State, el bot no sabría filtrar que mensajes de los recibidos debe añadir o reemplazar como tarea de una lista.

Sus relaciones con el resto de clases son:

- **Checklist:** Un estado puede estar asociado a ninguna o muchas checklists distintas.

Diagrama detallado de la capa de persistencia

La persistencia es la acción de preservar los datos de un objeto de forma permanente. Es imprescindible que tras el guardado de la información se pueda volver a recuperar para utilizarla nuevamente. En la aplicación se debe manejar la persistencia de los datos de dos maneras.

- **Variables.** La información se puede almacenar mediante variables de diferentes tipos. Para recuperar los objetos se debe tener alguna variable que guarde una referencia al objeto. Los datos almacenados de esta forma se encuentran en memoria RAM. Para guardar información de varias instancias la mejor solución es un array con punteros a cada una de las instancias. Un objeto existe mientras haya alguna referencia a él. Para evitar consumir memoria innecesariamente, se deben borrar las referencias a un objeto cuando deje de ser necesario.

- **Fichero.** Es el método usado para que los datos sobrevivan a la ejecución del programa que los ha creado. Sin los correspondientes ficheros, los datos solo existen en memoria RAM, y se pierden cuando la memoria pierde energía. La causa más común es el apagado del ordenador o del servidor en el cual la aplicación está siendo ejecutada. Para evitar la pérdida de datos, la información se almacena en un medio secundario, no volátil. De esta forma, los objetos pueden ser reconstruidos una vez se corra nuevamente la aplicación. Así se puede proceder a su reutilización. En conclusión, el tiempo de vida de una instancia es independiente del proceso que la creó.

El almacenamiento de los datos en disco es inevitable si se quiere poder recuperar las checklists y sus tareas. Se han definido cuatro controladores en el sistema, uno por cada clase del modelo. Cada uno de estos almacena y recupera información referente únicamente a los objetos de su clase. Debido al requisito no funcional impuesto en el capítulo 3, los datos son guardados en archivos XML. Los archivos XML son idóneos para almacenar objetos con estructura homogénea. Cada controlador edita un XML diferente. Al ser archivos con un esquema fijo es fácil exportarlos a un Excel y poder analizar los datos que contienen. Gracias a la existencia de esta posibilidad, los datos de la aplicación pueden ser fácilmente observables por un usuario administrador sin conocimientos de programación. Además, una vez exportado el XML a Excel las tablas se pueden filtrar por una de las columnas simplificando aún más la tarea de buscar algún dato concreto.

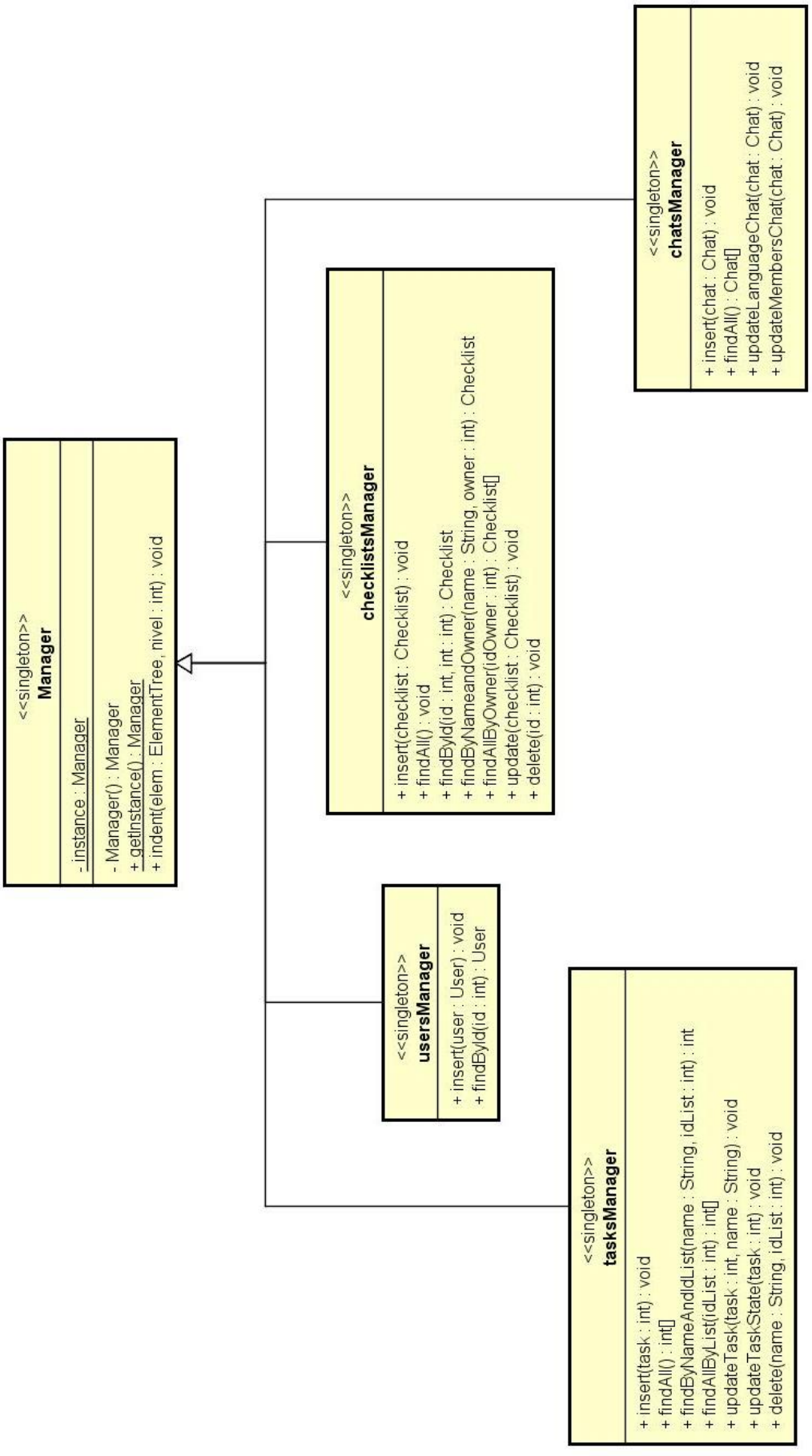

Figura 18. Diagrama detallado de clases del paquete de persistencia.

En la figura 18 se pueden observar los cuatro gestores de datos mencionados anteriormente y sus operaciones. Cada una de estas clases maneja los datos de una clase del modelo tal y como indica su nombre. Los cuatro gestores heredan de un gestor. Se ha añadido un quinto gestor del cual hereda el resto para evitar la duplicidad de código. Todos los gestores requieren la operación *indent* para hacer legible el XML generado y las correspondientes operaciones para implementar el patrón *singleton* explicadas a continuación.

Los gestores deben cumplir con el patrón de diseño *singleton.* El patrón *singleton* es necesario en lenguajes como Java en el que todo archivo es una clase. Para utilizar una *clase* en Java se debe llamar primero a su constructor. Cada llamada al constructor crea un nuevo objeto de la clase. Para impedir que se creen múltiples instancias de una misma clase, se usa el patrón *singleton*.

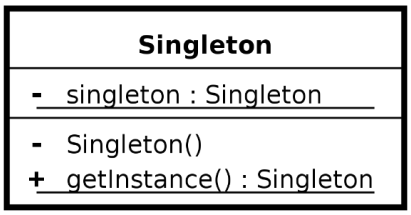

Figura 19. Estructura de una clase con estereotipo *Singleton* en UML.

Una clase que sigue el patrón *singleton* privatiza su constructor y para crear una nueva instancia de él y usarlo desde otra clase se llama al método getInstance(). Esta función es estática, es decir, pertenece a la clase y no al objeto. Lo mismo pasa con el atributo singleton. Cuando se llama a getInstance(), si el atributo singleton es nulo se invoca al constructor de la clase y se almacena el nuevo objeto creado en él. Si el atributo singleton tiene un objeto de tipo Singleton asignado, al llamar a getInstance() se devuelve el valor del atributo estático.

Los gestores deben disponer de métodos análogos para cada una de las clases del modelo que gestionan. Estos métodos se pueden englobar en cuatro grupos según la operación SQL que simulan:

- **Insert:** Los métodos insert se encargan de escribir en el archivo XML correspondiente la información del objeto que reciben como parámetro de entrada.
- **Find:** Las definiciones find buscan dentro de su respectivo fichero XML una o más instancias de la clase. Si no recogen valor de entrada *(findAll())* retornan todos los objetos almacenados en el XML. Si, por el contrario, reciben un valor de entrada filtran los datos devueltos por este valor.
- **Update:** Las operaciones update actualizan el XML de la clase con los nuevos datos de un objeto concreto. Para llevar a cabo la modificación del archivo, primero buscan el objeto que cumple con el parámetro de entrada y luego actualizan la información de este según se indique.
- **Delete:** Los métodos delete eliminan una instancia de un archivo XML. Localizan el objeto que coincide con el valor pasado por entrada y lo eliminan del XML. De esta forma se borran los datos de un objeto definitivamente y ya no pueden ser recuperados.

Los métodos insert() simularían una operación Create y las definiciones find() una operación Read, Se mantiene el mismo nombre para las operaciones Update y Delete de CRUD.

Diagrama detallado de la capa de controladores

El paquete de controladores contiene toda la lógica del negocio. Un controlador es la clase que crea, modifica o elimina los objetos del sistema. También, es el encargado de resolver las peticiones que son realizadas por el usuario a través de la vista. Todas las funciones definidas en un controlador son iniciadas cuando un evento sucede en la vista y esta tiene que mostrar algún cambio para responder a la solicitud. En el caso de estudio, cuando se recibe un nuevo mensaje de usuario el controlador lo recoge y procesa. Para obtener una respuesta a la petición recibida interactúa con el modelo y los gestores. Realiza las operaciones necesarias para llegar a resolver la solicitud del usuario y luego devuelve el mensaje a la "vista". Es decir, envía un mensaje al servidor de Telegram, quien será el que presente el resultado de la petición al usuario. De la tarea de los controladores se puede deducir que actúan de intermediarios entre el modelo y la vista. Su rol dentro de la aplicación es la del "middleware". Las clases del paquete controlador habitualmente son las más extensas y complejas porque en ellas se encuentra la mayor parte de la lógica de la aplicación.

Al igual que los gestores, es recomendable implementar los controladores siguiendo el patrón *singleton.* De esta forma, se garantiza la existencia de un único controlador de cada tipo para toda la aplicación.

Inicialmente, se pensó en diseñar la capa *controller* con un controlador por cada uno de los casos de uso especificados en el capítulo 3. Al analizar la complejidad operacional de cada uno de ellos, se llegó a la conclusión de que algunos de los controladores se podían agrupar. Ejemplos de controladores demasiado simples son: clearChecklistController, removeChecklistController o removeTaskController. Finalmente, siempre que así sea posible, se ha optado por agrupar los controladores por clases del modelo. Siguiendo esta idea se definen seis controladores.

Se puede observar el paquete de controladores en la siguiente página, figura 18. A priori, en esta imagen se puede observar que cinco de los controladores heredan de uno: **Controller.**

Controller

Esta clase es la que define las operaciones necesarias para que el resto de controladores sigan, a través de los mecanismos de herencia, el patrón *singleton*. En esta clase también se implementan algunos métodos que serán útiles para el resto de controladores:

- **message:** Devuelve un diccionario con los parámetros de entrada. Este método es usado para retornar la respuesta a la petición siempre como una variable con la misma estructura al controlador de vista.
- **printChecklist:** Convierte un objeto checklist con sus tareas a un texto con el formato adecuado para ser enviado al usuario. "Imprime" la lista.
- **enableOptions:** Añade a los botones iniciales los necesarios para dar privilegios al usuario creador.
- **checklistKeyboard:** Genera el teclado de botones de tareas que acompaña al mensaje con la checklist.
- **checklistMessage:** Retorna una lista con uno o varios mensajes con la checklist actualizada acompañados del chat al que deben enviarse.

Esta clase dispone del atributo público estático *instance,* necesario para implementar el patrón *singleton* y el método *getInstance()*

Figura 20. Diagrama de clases del paquete de controladores y la relación de herencia entre clases.

ChatsController

Este controlador engloba las operaciones referentes a la clase Chat y su relacionada: User. Es la encargada de resolver peticiones de los mensajes recibidos cuando: un usuario inicia conversación con el bot, se añaden o eliminan miembros en grupos o se selecciona el lenguaje. Los métodos de los que dispone para ello son:

- **checkUser:** Comprueba si los datos de un usuario se encuentran ya almacenados de manera persistente. Si no es así, los guarda.
- **selectLanguage:** Devuelve las opciones disponibles de lenguaje para el bot.
- **saveLanguage:** Guarda la selección de lenguaje de un usuario de forma persistente.
- **noModify:** Mantiene todo en su estado anterior sin guardar las modificaciones que habían sido iniciadas.
- **newChat:** Comprueba si los datos de un chat ya se encontraban almacenados. Si no es así, los guarda.
- **help:** Recupera y retorna la ayuda del bot.
- **messageGroup:** Comprueba si el usuario emisor de un mensaje dentro de un grupo ya se encontraba guardado de manera persistente como miembro de ese grupo. Si no es así le añade.
- **messageNewMember:** Tratamiento de mensaje recibido con campo "new_member". Añade el nuevo miembro al correspondiente grupo en el XML.
- **messageLeftMember:** Tratamiento de mensaje recibido con campo "left_member". Elimina al miembro del correspondiente grupo en el XML.

ChecklistsController

El controlador de checklists es el encargado de resolver todas las peticiones referentes a una lista concreta. Los métodos que pertenecen a esta clase son:

- **generateName:** Devuelve tres nombres únicos y no repetidos a partir del nombre pasado como parámetro de entrada.
- **newChecklist:** Realiza las operaciones necesarias para crear una nueva checklist. Comprueba si el nombre está duplicado y crea el objeto Checklist.
- **doneChecklist:** Confirma la creación de la nueva checklist.
- **editOptions:** Genera y devuelve la lista de botones con las opciones disponibles.
- **clearChecklist:** Reinicializa las tareas de la checklist.
- **cloneChecklist:** Comprueba si el nombre de la checklist a clonar ya existe entre las listas del usuario y si no es así la duplica dentro de sus checklists.
- **requestRemoveListConfirmation:** Retorna un teclado para solicitar la confirmación para eliminar una checklist.
- **removeChecklist:** Elimina la información de una checklist y sus tareas de los XML correspondientes.

TasksController

Esta clase controla todas las peticiones relacionadas con la clase del modelo Task. En el controlador de tareas se pueden encontrar los siguientes métodos:

- **requestNewTask:** Solicita la nueva tarea a añadir a cierta checklist y establece el estado de la lista a "añadiendo nueva tarea".

- **newTask:** Comprueba si la tarea ya existe en la checklist y si no es así crea un objeto Task relacionado a la lista.
- **selectModifyTask:** Retorna una lista de botones con todas las tareas pertenecientes a una checklist concreta.
- **requestModifyTask:** Solicita la nueva tarea que sustituirá a la tarea que se está modificando.
- **modifyTask:** Reemplaza la tarea en modificación por la nueva. Ambas se pasan como parámetro de entrada. Además, elimina la tarea reemplazada del XML e inserta la nueva.
- **undoTask:** Establece el estado de la tarea en modificación a *False.* Es decir, deshace una tarea.
- **selectRemoveTask:** Devuelve un teclado con un botón por cada tarea de una checklist concreta.
- **removeTask:** Elimina la información de una tarea del XML correspondiente.
- **requestRemoveTaskConfirmation:** Retorna un teclado para solicitar la confirmación de eliminar la última tarea de la lista y con ello la lista.
- **checkTask:** Realiza las operaciones y comprobaciones necesarias para marcar una tarea dentro de una checklist. Se comprueba que el usuario que la está marcando sea el asignado, si es que la tarea está asignada, y que la tarea se encuentre sin hacer.

ViewChecklistsController

Controlador de caso de uso. El caso de uso **Ver mis checklists** tiene suficiente lógica asociada para ser un controlador aparte. Además, sus funciones no se pueden englobar dentro de una sola clase del modelo. Por ambas razones se ha decidido crear un controlador para todos los métodos relacionados con recuperar y mostrar la información de una o más checklists. Los métodos de esta clase son:

- **doneTasks:** Calcula y retorna las tareas realizadas dentro de una checklist.
- **all:** Genera un mensaje, el cual devolverá, con datos sobre todas las checklists de un usuario. Este mensaje es el que el usuario obtiene al enviar el comando */checklists*.
- **detail:** Recupera la información de una checklist concreta y retorna un mensaje con el detalle de esta.
- **findResults:** Devuelve las checklists de un usuario en las cuales su nombre coincide con las letras introducidas por el propio usuario en la búsqueda. Obtiene los cinco primeros resultados.
- **shareResult:** Almacena el mensaje en el que se ha enviado el detalle de una checklist resultado de la búsqueda en el XML correspondiente. Al guardar este mensaje, la aplicación lo actualizará al unísono con el resto de mensajes de la misma lista.

AssignTaskController

Controlador de caso de uso **Asignar Tarea.** Este caso de uso es el que más pasos requiere para ser realizado, en consecuente, es el que más métodos asociados tiene. Por esta razón, no se ha englobado junto al controlador de tareas. A pesar de ello, este controlador está íntimamente relacionado con la clase del modelo Task.

- **selectChecklist:** Devuelve una lista de botones con todas las checklists del usuario que envía el comando */assignTask*.
- **selectTask:** Retorna un teclado con un botón por cada una de las tareas referentes a la checklist pasada como parámetro de entrada.
- **selectMember:** Devuelve un teclado con botones. Uno por cada miembro del chat almacenado de forma persistente en la aplicación.

- **assign:** Asigna la tarea al usuario enviado mediante un mensaje a través de la funcionalidad de menciones de Telegram. Comprueba que el usuario no tenga alias, si tiene alias la aplicación no puede acceder a su identificador de usuario por lo que no le asigna a la tarea y devuelve un mensaje informando.
- **assign_CallbackQuery:** Asigna al usuario seleccionado mediante la lista de botones a la tarea seleccionada anteriormente.
- **addMember:** Solicita un usuario el cual se añadirá a los miembros del grupo, si es que no existe ya, y además se asignará a él la tarea mediante la llamada al método assign.

ViewController

A mayores de los seis controladores explicados, existe el controlador de vista mencionado. Este séptimo controlador es el intermediario entre el resto de controladores y la "vista". Toda la lógica de este controlador está destinada a analizar el mensaje recibido del usuario y averiguar qué operación realizar para darle respuesta. Por esta razón, es en este controlador donde se agrupan todas las operaciones de envío de mensajes al usuario. Existe una relación de dependencia entre el *viewController* y los demás controladores. El controlador de vista depende del resto porque invoca sus métodos para resolver las peticiones del usuario.

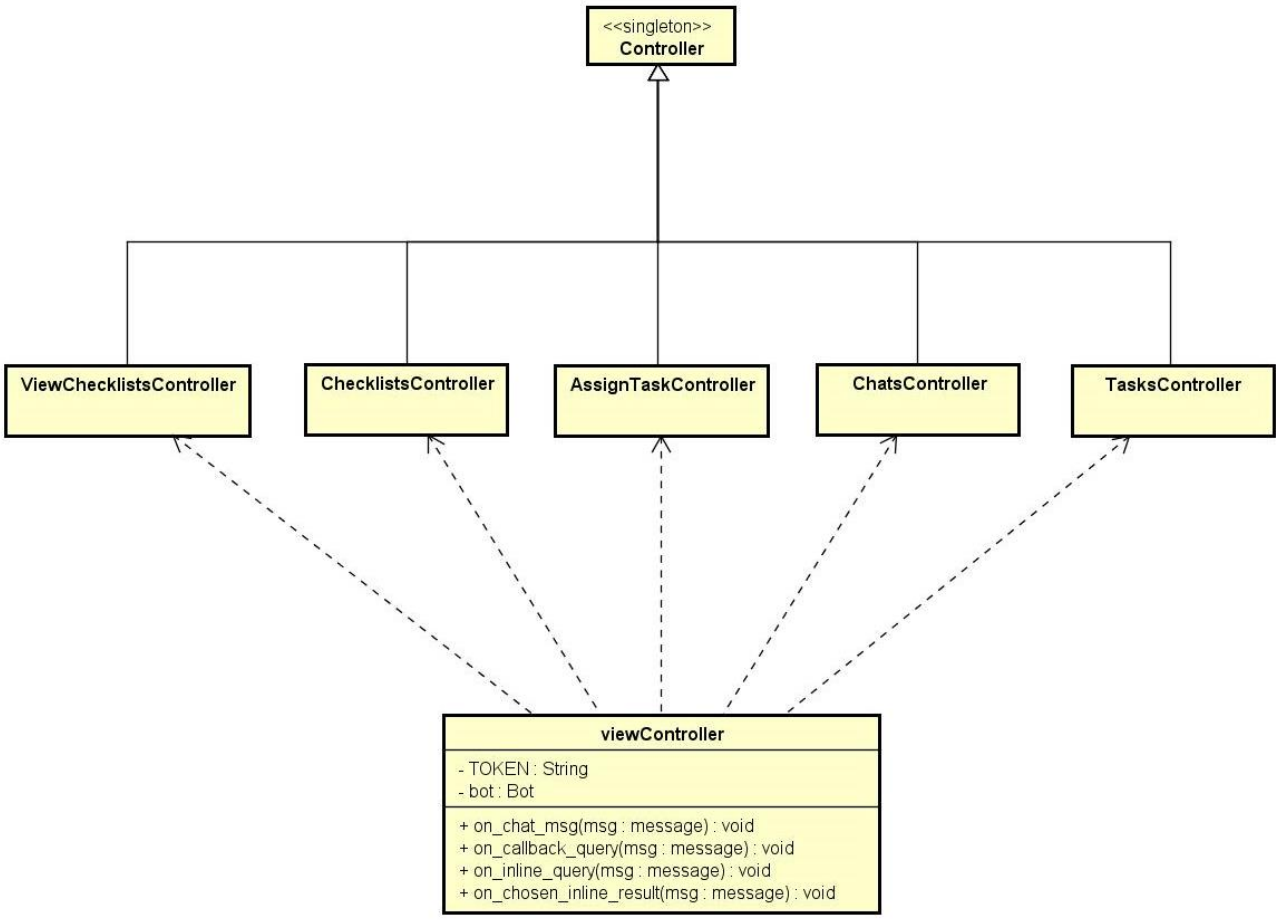

Figura 21. Diagrama de clases del paquete de controladores y la relación de dependencia con el controlador de vista.

Los métodos necesarios en la clase viewController se han descubierto con ayuda de *telepot. Telepot* es uno de los *framework* Python que Telegram recomienda para implementar sus bots. Basándose en la documentación de *telepot* y en el alcance de la aplicación se han considerado imprescindibles los

métodos de tratamiento de todos los tipos diferentes de peticiones que un usuario puede enviar. Estos métodos son:

- **on_chat_msg:** Trata los mensajes enviados por el usuario. Dentro de estos mensajes se incluyen: stickers, fotografías, gifts, etc. Dentro del *ChecklistBot* este método se usa para el tratamiento de todos los mensajes de texto recibidos.
- **on_callback_query:** Se encarga de los eventos recibidos al pulsar un *inlineButton.* Dentro de los campos disponibles en un objeto de tipo *inlineButton* se encuentra el campo: *callback_data*. Este campo es recibido por el método cuando un usuario pulsa el respectivo botón.
- **on_inline_query:** Trata las peticiones enviadas a través del modo *inline*. Un usuario realiza una petición *inline* al escribir el alias del bot seguido de cualquier texto. El modo *inline* presenta una forma distinta de interactuar con un bot. Este método es el encargado de las búsquedas de checklists de un usuario dentro de la aplicación.
- **on_chosen_result:** Es invocado al seleccionar uno de los resultados mostrados por el método anterior: *on_inline_query*. Este método es usado para almacenar el identificador del mensaje enviado al elegir un resultado de una petición *inline* como un nuevo mensaje asociado a la checklist seleccionada.

Diagrama relacional

No siempre el esquema que siguen las tablas de la base de datos donde almacenaremos los objetos de la aplicación coincide con el esquema de las clases del modelo. A través del diagrama relacional se pueden mostrar las tablas y las relaciones que existen entre ellas. Para llegar al diagrama relacional se debe realizar primero el diagrama entidad-relación partiendo del diagrama de clases del modelo. Un diagrama entidad-relación muestra entidades y sus relaciones de forma gráfica, modela la estructura que tendrá la base de datos. Es común que el diagrama de entidad-relación coincida con el diagrama de clases. Las diferencias más habituales que podemos encontrar son: las clases asociativas. Toda relación "muchos a muchos" del diagrama de clases debe ser eliminada en el diagrama entidad-relación porque este tipo de relaciones no se debe presentar en una base de datos. La razón es que esto llevaría a guardar en un mismo campo más de un valor FK, imposibilitando esto la navegación de una tabla a otra a través de una operación CRUD. Existen dos opciones para eliminar las relaciones "muchos a muchos":

- Incluir una clase entre las dos clases con relación muchos a muchos. Se introduce una tercera clase relacionada con las dos existentes. Esta clase estará relacionada con las otras dos uno a muchos. En esta nueva clase se incluyen como atributos dos claves foráneas, una a cada clave primaria de las clases originales.

Figura 22. Solución para diagramas entidad-relación mediante tercera clase a relaciones "muchos a muchos".

- Incluir una clase asociativa. Se mantiene la relación muchos a muchos, pero de ella sale una clase asociativa. Esta clase tiene los mismos atributos que la explicada en el punto anterior.

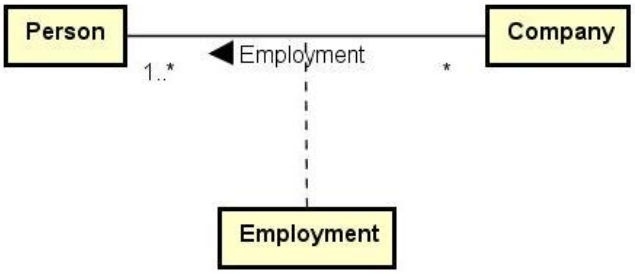

Figura 23. Solución para diagramas entidad-relación mediante clase asociativa a relaciones "muchos a muchos".

Ambas soluciones son idénticas en el momento de implementarlas. En conclusión, es la misma solución con diferente representación dentro del diagrama.

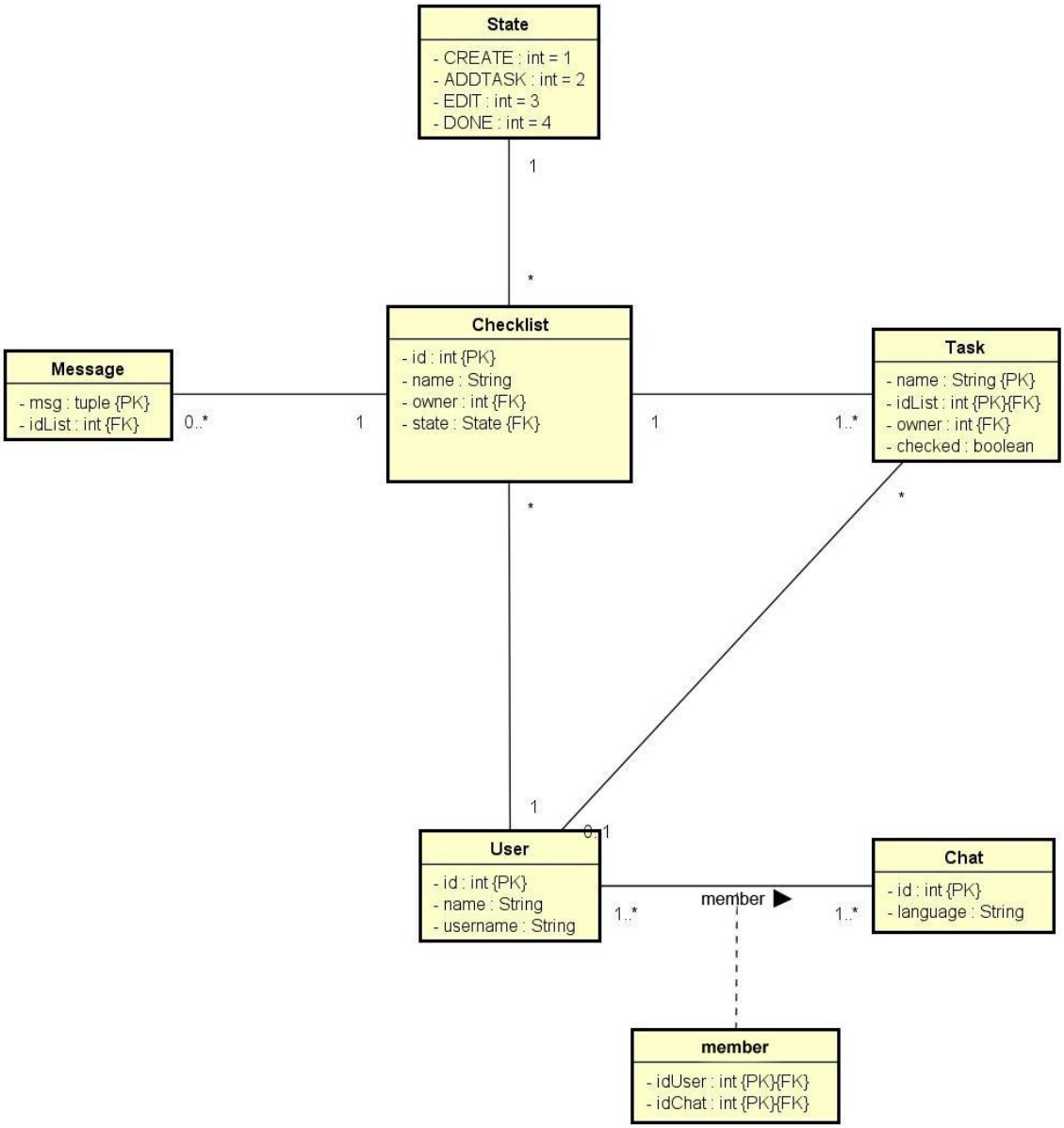

Figura 24. Diagrama entidad-relación de la base de datos.

Observando el diagrama de la figura superior se pueden ver algunos cambios con el diagrama de clases presentado en el apartado **Diagrama detallado del modelo.** El primer cambio, el cual ya se mencionaba antes, es la aparición de una clase asociativa. La clase asociativa creada para solventar el problema de la relación "muchos a muchos" entre *User* y *Chat* es *Session*. Esta clase únicamente tiene dos atributos con las claves de una fila de la tabla usuario y otra de la tabla chat. Además, se ha sustituido el atributo *messages* de la clase Checklist por una nueva entidad Message. Esta tiene una relación uno a muchos con Checklist. En decir, un mensaje pertenece a una única checklist y una checklist tiene uno o más mensajes. Con estos cambios eliminamos la posibilidad de almacenar más de un valor en un campo de la tabla. De él se obtiene el siguiente diagrama relacional:

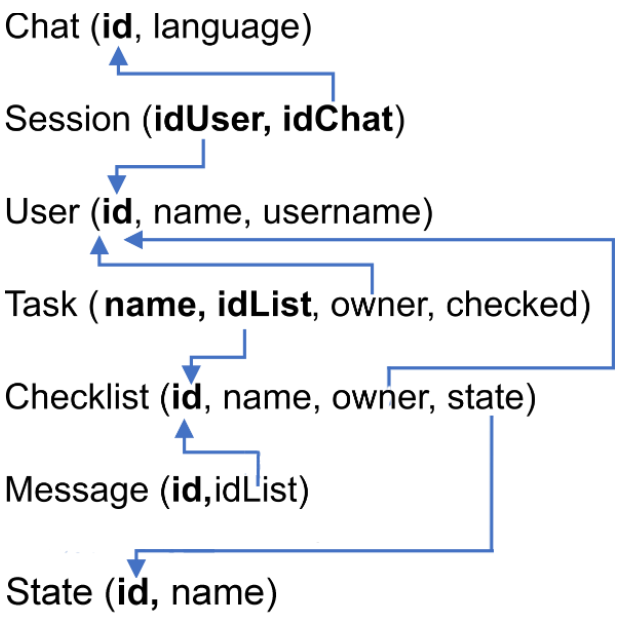

Figura 25. Diagrama relacional de la base de datos.

En letra resaltada se muestran las claves primarias (Primary Keys **(PK)**) de cada tabla. Una *primary key* es un valor único e irrepetible para una fila de una tabla. Cada fila puede ser diferenciada unívocamente del resto mediante su PK. Las flechas muestran la relación entre las claves foráneas (Foraign Keys **(FK)**) y las *primary keys*. El valor del que parte la flecha es la foraign key. Una *FK* guarda el valor *PK* de la fila de otra tabla. Es muy importante que en una base de datos SQL se cumpla la integridad referencial. La integridad referencia significa que la clave externa de una tabla siempre debe referenciar a una fila válida de la tabla a la que se haga referencia. La integridad referencial garantiza que la relación entre dos tablas permanezca sincronizada durante las operaciones de actualización y eliminación. El incumplimiento de la integridad referencial da lugar a confusión en la realización de ciertas operaciones CRUD y finalmente a errores.

Diagramas de secuencia

Un diagrama de secuencia es un diagrama en el cual se modela la interacción entre objetos del sistema para llevar a cabo un caso de uso. La interacción entre los distintos objetos se realiza mediante operaciones. Estas operaciones se organizan de tal forma que permitan distinguir en qué orden se realizan al observar el diagrama. En un diagrama de secuencia se pueden representar operaciones síncronas y asíncronas dependiendo del tipo de línea que acompaña a la operación. Las operaciones asíncronas se representan mediante una línea discontinua.

Los casos de uso que se van a analizar en las siguientes páginas son los estudiados en los diagramas de actividades del capítulo de análisis. De este modo obtendremos dos puntos de vista diferentes para los mismos casos de uso. En el diagrama de secuencia se muestra la misma operativa mostrada en el de actividades, pero de manera más técnica y entrando en detalle de cómo se realiza cada paso del sistema. Las operaciones utilizadas para cada clase en el diagrama son las explicadas en los apartados anteriores del capítulo de diseño.

Para simplificar la lectura de los diagramas de secuencia se ignoran las operaciones relacionadas a la implementación del patrón *singleton.* La dinámica de estas operaciones sería: (1) llamar primero a la clase con *getInstance()* y (2) invocar al objeto específico que nos ha devuelto la clase con el método deseado. Estos dos pasos son necesarios antes de poder llamar a cualquier método de controladores y gestores. En resumen, cualquier llamada a una operación de un objeto controlador/gestor es siempre sobre el objeto *instance*.

En el diagrama de la página siguiente se representan todas las operaciones entre objetos, a través del tiempo, necesarias para llevar a cabo el caso de uso **Crear Checklist.** Tal y como se observa en el diagrama, el caso de uso se realiza con el envío de mínimo tres mensajes por parte del usuario.

El primero contiene el comando "*/new*". Dependiendo de si el mensaje únicamente se compone del comando o del comando y el nombre para la lista, la aplicación entrará en el primer bloque condicional. Si no se ha enviado un nombre para la checklist se le solicitará al usuario. Una vez la aplicación dispone del nombre se llama al método *newChecklist* del controlador de checklists. Se le pasa el nombre de la lista y el identificador del usuario. La aplicación tiene dos bloques diferenciados que dependen de si el nombre dado está duplicado entre las listas del usuario o no. En el primer bloque, si el nombre ya existe, el *checklistsController* genera tres nombres no existentes y los devuelve al usuario en forma de botones. El usuario selecciona uno de ellos, por lo tanto, se invoca al método del *viewController* on_callback_query. Desde este, se llama de nuevo a la función *newChecklist* con el nombre correcto y se crea el objeto Checklist. Si desde el principio el nombre no estaba repetido, se crea el objeto Checklist. Después el bot envía un mensaje solicitando una tarea.

El segundo mensaje que envía el usuario, dentro de los tres mínimos necesarios, contiene el nombre de una tarea. Se llama al método *newTask* del controlador de tareas con el nombre proporcionado y el objeto Checklist creado. Este método comprueba si la tarea ya existe dentro de la lista y, si no es así, crea un objeto Task. Finalmente, devuelve un mensaje al usuario informando del resultado de la operación. Esta operativa se repite en bucle mientras el usuario envíe cualquier texto que no sea el mensaje "/done".

El tercer mensaje, y último, es el comando "*/done".* Al enviar este mensaje, se da por finalizado el procedimiento de creación de una nueva lista. Se realiza la operación *doneChecklist,* durante la cual se insertan los objetos creados en sus respectivos XML y se invoca el método *checklistMessage* que genera el mensaje que muestra la checklist creada con sus botones.

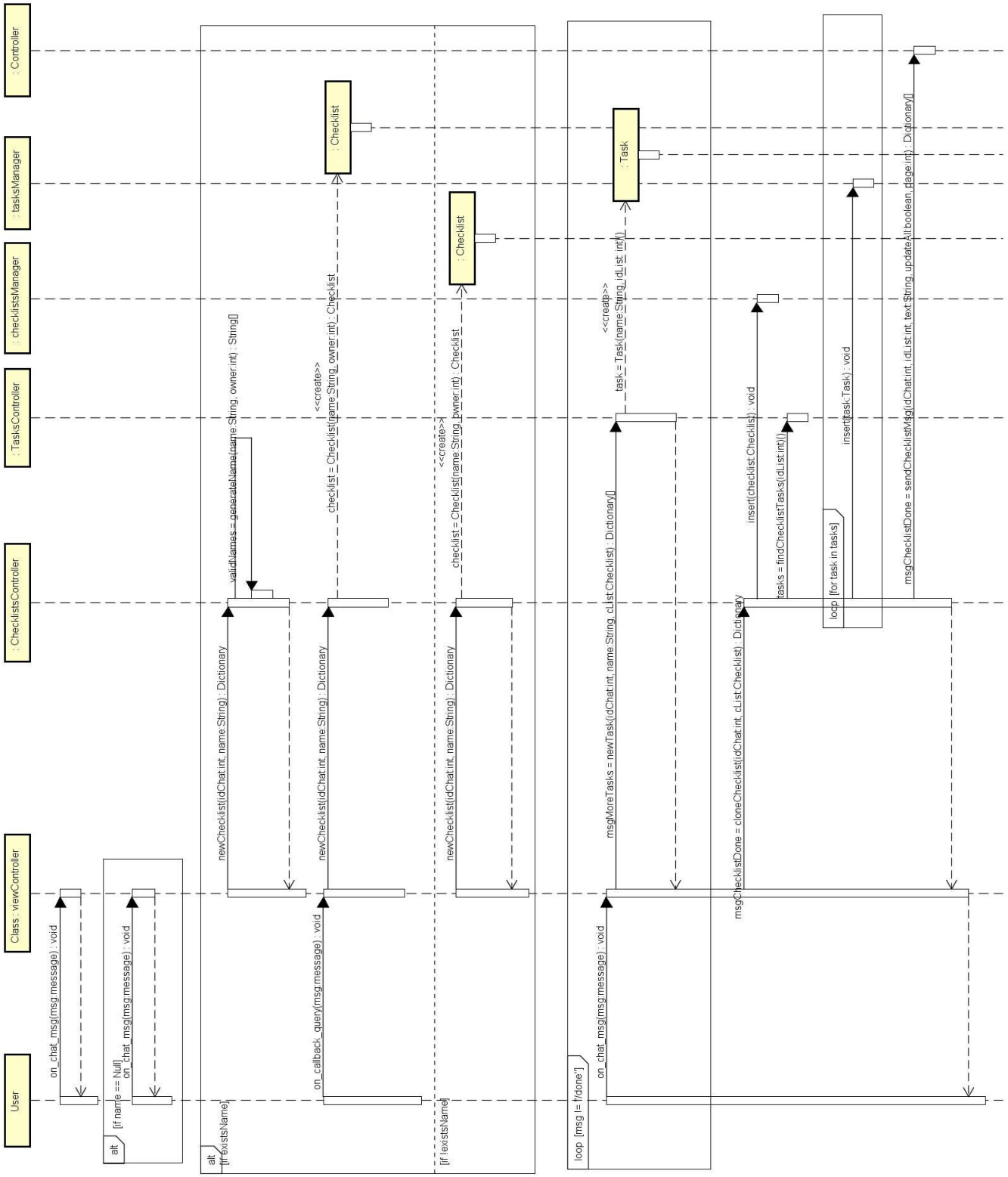

Figura 26. Diagrama de secuencia del CU Crear Checklist.

Se puede ver en el diagrama anterior que un objeto nunca es insertado en el respectivo XML hasta confirmar la operación. Esto se ha diseñado así para evitar inserciones erróneas en los archivos. Por ejemplo, si se comienza la creación de una checklist y no llegar a finalizarse se crearía una checklist con sus tareas recuperable. El usuario debe confirmar el procedimiento de creación mediante el envío del comando *"/done".* Un ejemplo que ilustra mejor la necesidad de que se proceda así es el presentado a continuación. Si un usuario comienza la creación de una lista y al solicitársele la primera tarea envía cualquiera de los comandos reconocidos por el bot, el bot pasa a realizar las operaciones necesarias para dar solución a este comando. Esta casuística deja un objeto *Checklist* almacenado en el XML que no dispone de ninguna tarea asociado en el archivo de tareas. Por lo tanto, al enviar el comando *"/checklists"* el usuario verá esta lista entre sus checklists y podrá recuperarla pulsando sobre su link. Esto retornará un mensaje únicamente con el nombre de la lista y sin botones de tareas.

El siguiente diagrama de secuencia que se va a exponer y explicar es el del caso de uso **Editar Checklist.** Este caso de uso se va a presentar de manera simplificada para reducir la complejidad del correspondiente diagrama.

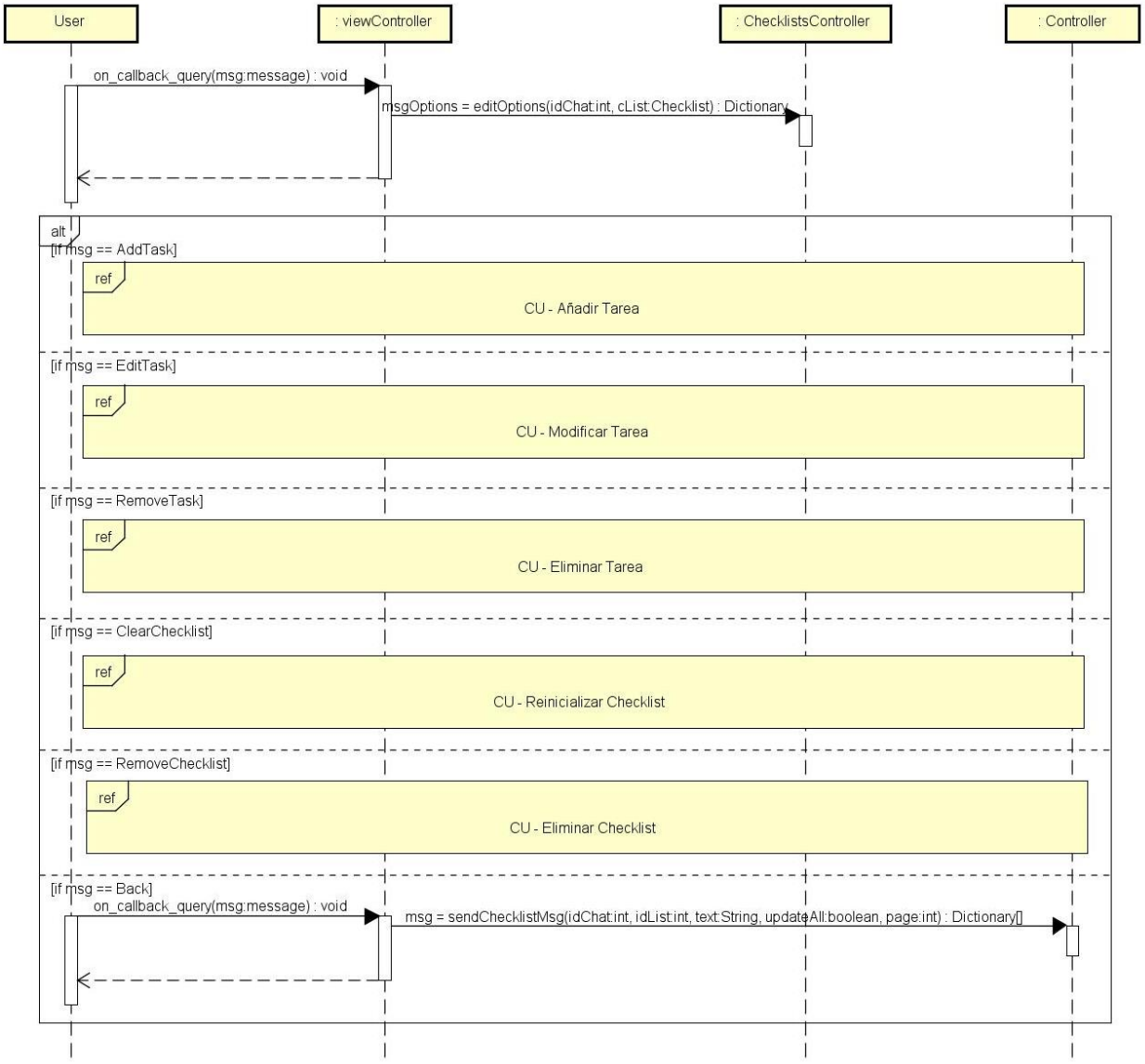

Figura 27. Diagrama de secuencia simplificado del CU Editar Checklist.

En el diagrama de secuencia se puede observar la misma estructura descrita durante la especificación de casos de usos. El diagrama es fácilmente descomponible en los cinco casos de uso que representan las cinco operaciones de edición diferentes. Cada uno de estos casos de uso, tiene un diagrama de secuencia con una estructura diferenciada del resto. Por ejemplo, en el caso de uso **Añadir Tarea** el bot requiere que el usuario envíe mensajes de texto, por el contrario, en el caso de uso **Eliminar Tarea** se interactúa con el usuario únicamente mediante teclados de *inline buttons.* La realización del CU **Modificar Tarea** necesita de una mezcla de ambos métodos de interacción. Debida a la mayor completitud del último caso de uso mencionado se procede a la presentación de su diagrama de secuencia.

En la figura 28 se puede observar el diagrama de secuencia del correspondiente caso de uso. El sistema interactúa como a continuación se expone, si el usuario selecciona la opción modificar tarea de entre todas las disponibles. La primera operación que la aplicación realiza es la obtención de todas las tareas de la lista. Para llevar a cabo esta operación, el *viewController* llama al método *selectModifyTask* del controlador de tareas al recibir el "evento de modificar tarea". Este método retorna todas las tareas de la lista en forma de botones seleccionables. Al eleccionar el usuario uno, se invoca al método *on_callback_query* del controlador de vista y este a su vez invoca al método *modifyTask*. Este método retorna un mensaje solicitando la nueva tarea y además guarda el nombre de la tarea que se va a reemplazar. El usuario envía un mensaje, que el controlador de vista capta mediante la función *on_chat_msg* y manda procesarlo al *tasksController* a través del método *modify.* Este método realiza toda la operativa necesaria para llevar a cabo el reemplazo. Primero, localiza la tarea a modificar apoyandose en el *tasksManager.* Busca la tarea a partir del identificador de la lista a la cual pertenece y el nombre de la tarea. Después, realiza una operación de *update* sobre este elemento en el XML. Le pasa la tarea y el nuevo nombre por el cual se va a reemplazar el antiguo. Por último llama al *controller* para que genere el mensaje de la respectiva checklist actualizada para todos los chats en los que se encuentre compartida. La actualización se realiza en todos los chats mediante una operación de *editMessage* de Telegram por cada chat. Esta operación es invocada desde el controlador de vista al recibir los mensajes actualizados. Si en vez de el usuario enviar un mensaje de texto con el nuevo nombre de la tarea, pulsase sobre el botón "Deshacer" se realizaría el mismo procedimiento sustituyendo la llamada a *updateTask* por *updateTaskState.* El bot recibe la petición de deshacer tarea a través del método *on_callback_query* en vez de *on_chat_msg.*

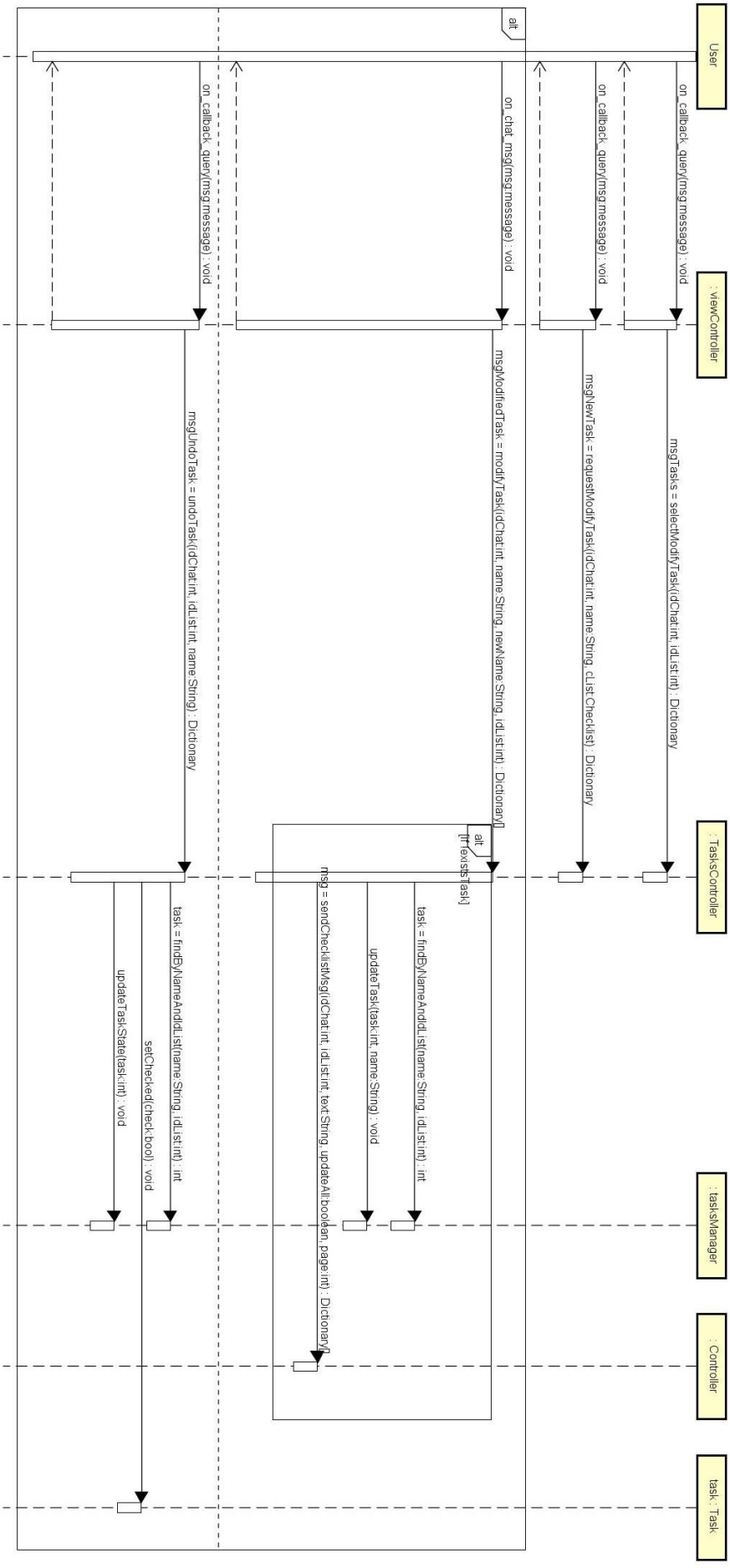

Figura 28. Diagrama de secuencia del CU Modificar Tarea.

Capítulo 5. Implementación

0@BotFather

El primer paso que se debe realizar para comenzar a implementar un bot para Telegram, es común para cualquier bot que se quiera desarrollar. Tal y como se ha mencionado en el apartado **Bots en Telegram** del capítulo introductorio, es necesario un token para poder realizar peticiones a la API Bot. Este token se obtiene a través de @BotFather. @BotFather es un bot desarrollado por Telegram que nos permite crear un nuevo bot. Para ello, debemos comenzar una conversación con este bot y seguir el siguiente procedimiento:

- Enviar el comando `/newbot`.
- @BotFather nos solicitará un nombre para el bot y un nombre de usuario único por el cual los usuarios de Telegram podrán buscar y llamar al bot si desean usarlo. El nombre de usuario o alias debe identificar a nuestro bot unívocamente por lo que @BotFather no aceptará ningún nombre que ya exista en la base de datos de Telegram. En este proyecto, los nombres proporcionados han sido: **ChecklistBot** como nombre público y **checklistTFG_bot** como nombre de usuario. Se debe tener en cuenta que el alias siempre debe acabar en bot. El hecho de que se nos obligue a meter un alias único nos condiciona a la hora de elegir un nombre de usuario sencillo. Por esta razón, el alias seleccionado para el proyecto no es @checklistBot como se hubiera preferido. Aun así, existe una posibilidad de modificar el alias una vez se tenga un bot totalmente operacional contactando con el soporte de Bots de Telegram. Finalmente, BotFather nos proporciona el token mediante el cual podemos acceder al Bot API vía https.

Para configurar las opciones del nuevo bot que acabamos de crear, también se usa a @BotFather. A continuación, una lista con los comandos disponibles y su descripción:

- `/setdescription`: es la descripción de lo que hace nuestro bot. Será el texto que vean los usuarios al iniciar la conversación con el bot bajo el título “¿Qué puede hacer este bot?
- `/setabouttext`: es un breve texto que acompaña a nuestro bot. Será el texto que aparece al compartir el bot acompañando el enlace.
- `/setuserpic`: para establecer la foto de perfil del bot.
- `/setcommands`: las acciones que puede llevar a cabo el bot se piden con comandos. Mediante este comando podemos cambiar la lista de los comandos de nuestro bot. El comando siempre debe empezar con una “/” y estar en minúscula. Debemos especificar los parámetros y una descripción del comando. Se podrá ver la lista de comandos correspondiente al bot escribiendo “/” en una conversación con él o pulsando sobre el icono de comandos.

- `/setjoingroups`: para configurar cómo se puede usar nuestro bot, si únicamente de forma privada o también añadiéndolo a grupos.
- `/setprivacy`: determina qué mensajes recibirá el bot, se recomienda dejar activada esta opción. Si la desactivamos el bot recibirá todos los mensajes, si la activamos solamente recibirá los que comienzan por "/".

Para especificar los comandos con los que vamos a poder interactuar con el bot enviamos a FatherBot el mensaje `/setcommands`. Una vez hemos configurados los comandos del bot debemos empezar a programarlos para que el bot responda adecuadamente a cada uno de ellos. En esta aplicación, los comandos disponibles son:

- `/new`: Crea una nueva checklist. Se puede proporcionar el nombre de esta seguido del comando y separándolo mediante un espacio. Si solo se envía el comando, sin el nombre, el bot pedirá al usuario el nombre con el que desea identificar a su lista. Este nombre debe ser único dentro de las checklists del usuario. Si el nombre ya existe entre sus listas, @checklistTFG_bot proporcionará al usuario tres posibilidades nuevas. Este comando no se puede usar en chats grupales.
- `/checklists`: No disponible en chats grupales. Muestra las checklists del usuario que envía el comando. Envía una lista con todas las listas del usuario. Si el usuario no tiene ninguna lista, le informa. En esta lista se muestra de cada checklist:
  - el nombre
  - el baremo de tareas hechas
  - un comando para ver la checklist al detalle
- `/assignTask`: Permite asignar una tarea a un usuario específico. Este comando no se puede utilizar en chats individuales. Si es llamado desde un grupo, muestra una lista de botones con las checklists del usuario que ha enviado el comando. A través de 3 pasos el usuario puede asignar la tarea que desee a un miembro del grupo. Si el usuario no tiene ninguna lista, se le informa.
- `/language`: Este comando no se muestra entre los disponibles al introducir "/" o pulsar sobre el botón de comandos. Esta decisión ha sido tomada en base a reducir el número de comandos mostrados siempre al usuario y en que no es un comando que sea necesario para cumplir con las funcionalidades principales del bot. Su uso se reduce a ocasiones muy concretas en las cuales el usuario desea modificar el idioma que seleccionó al iniciar el bot. Este comando, permite al usuario modificar el idioma en el que el bot interactúa con él. Los idiomas disponibles son: español e inglés.
- `/help`: Este comando no se muestra entre los disponibles al introducir "/" o pulsar sobre el botón de comandos. Esta decisión ha sido tomada en base a simplificar el número de comandos mostrados siempre al usuario y en que no es un comando que sea necesario tener siempre disponible. Su uso se reduce a las primeras veces que el bot es utilizado por un nuevo usuario. Este comando se muestra si un usuario envía algo que el bot no puede "entender" y para lo cual no dispone de una respuesta programada. Al enviar el comando, @checklistTFG_bot muestra una ayuda sobre cómo usar el bot y sus comandos.

Del diseño a la implementación

Al desarrollar el *bot,* se han respetado la mayor parte de las decisiones tomadas en el capítulo anterior. El paquete con las clases del modelo es idéntico al descrito en diseño. Sin embargo, los controladores y gestores se han implementado de diferente manera. No se han implementado con clases debido a que, al contrario que en Java, en Python no se obliga a que todos los .py sean clases. Aprovechando las opciones que Python ofrece, los controladores y gestores se implementan como módulos. Esto permite crear "una sola instancia" y todas las llamadas a cada gestor son al "mismo objeto": el módulo. Un módulo es un archivo que contiene definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo .py agregado. Las definiciones de un módulo pueden ser importadas a otros módulos. Luego, los métodos del módulo pueden ser llamados desde la clase o módulo desde el que ha sido importado.

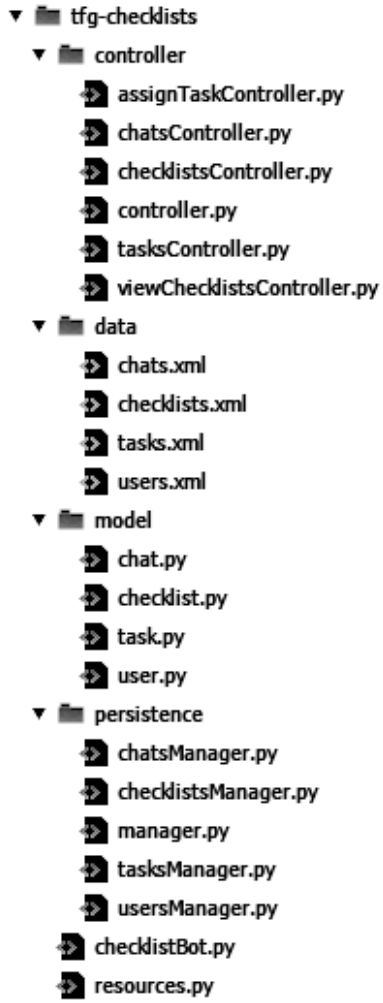

Figura 29. Estructura de la aplicación *ChecklistBot*.

La estructura del proyecto es la mostrada en la figura 29. Se observa que los archivos Python creados finalmente corresponden a las clases presentadas en diseño. Además, la distribución de estos archivos en carpetas es análoga a la de las clases en paquetes. Como se ha explicado en el párrafo anterior, los archivos .py de las carpetas *controller* y *persistence* son módulos. Se muestra la estructura de uno de estos módulos a continuación:

```
from telepot.namedtuple import InlineKeyboardMarkup, InlineKeyboardButton

from model.task import Task
from model.checklist import Checklist
from model.chat import Chat
from resources import Resources
import tasksManager, checklistsManager

def message(text, params, markup):

def printChecklist(cList):

def enableOptions(_id, cList, keyboard):

def checklistKeyboard(_id, cList, page, options = False):

def checklistMessage(_id, idList, text, updateAll, page = 0):
```

Figura 30. Estructura del módulo *Controller*.

Como se puede ver en la figura inmediatamente superior, un módulo no comienza con una definición de clase. En Java todo archivo .java debe comenzar con el habitual *public class.* Lo mismo sucede en C#, todos los archivos .cs deben comenzar definiendo la clase. Sin embargo, en Python no es obligatorio. Por esta razón, no es necesario implementar el patrón *Singleton* para conseguir un único punto de acceso a una "clase" y sus métodos. Debido a esto, los únicos archivos que implementan clases son los que representan clases del modelo. Es necesario que estos archivos sean clases para poder crear distintas instancias de ellos. En las entidades del modelo, se requiere crear objetos con atributos y propiedades independientes del resto de objetos de su misma clase. La estructura de una clase en Python es la mostrada en la imagen 31.

```
from enum import Enum

class States(Enum):
    CREATE = 1
    ADDTASK = 2
    EDIT = 3
    DONE = 4

class Checklist:
    idList = 1
    Checklists = {}

    def __init__(self, name, owner):

    #--------GETTERS------------
    def getId(self):

    def getName(self):

    def getOwner(self):

    def getMessages(self):

    def getState(self):

    #--------SETTERS------------
    def setId(self, idList):

    def setName(self, name):

    def setOwner(self, owner):

    def setMessages(self, messages):

    def setState(self, state):

    #----------MÉTODOS-------------#
    def appendMsg(self, msg):

    def findOldMsg(self, msg):

    def replaceMsg(self, msg):
```

Figura 31. Estructura de la clase C*hecklist*.

Se ha escogido la clase Checklist por ser la más completa y la que más detalles tiene para explicar. En el archivo *checklist.py* se definen dos clases: States y Checklist. Estas clases se han agrupado en un archivo porque están íntimamente relacionadas. La definición de varias clases en un mismo archivo es algo que otros lenguajes no permite.

El constructor de un objeto en Python siempre se define en el método __init__. Este método es heredado de la clase *object* y se sobrescribe en las clases desarrolladas por el programador. El atributo idList está definido fuera del constructor porque es de clase y se accede a él mediante: *Checklist.idList.* En otros lenguajes de programación, *idList* estaría acompañado de *static.* Lo mismo sucede con *checklists*, al cual se accede mediante *Checklist.Checklists*

En Python no existe una manera de definir los métodos y atributos como privados o públicos, en conclusión, se puede acceder a cualquier atributo de un objeto directamente. Aun así, en la aplicación desarrollada se accede a los atributos de un objeto mediante los correspondientes getters y setters. De esta forma, seguimos con el diseño ideado y además cumplimos con las buenas prácticas de programación.

Los métodos de objetos en Python siempre deben recibir al menos el parámetro *self. Self* es una palabra reservada en Python que representa al propio objeto. Es equivalente a *this* en C# y Java. Las únicas operaciones que no requieren del parámetro de entrada *self* son las de clase. Para definir una operación de clase en Python se usan los decoradores. El decorador @staticmethod encima de un método describe que el método a continuación es de clase. Un método de clase se invoca de forma análoga a un atributo de clase *Checklist.classMethod*.

En la clase *States* se puede observar el mecanismo de herencia de Python. *States* hereda de *Enum*. Todos los objetos instanciados de tipo *States* disponen de todas las operaciones propias de un tipo enumeración *(Enum).* En una enumeración se puede conseguir el valor o el "número" que representa ese valor. En código los objetos *States* se comparan mediante el número, mientras que en el archivo XML el valor que se guarda de *States* es la cadena de texto descriptiva.

La persistencia, como se describió en el capítulo cuatro, se logra a través de archivos y variables. Las instancias de chat son almacenadas en un diccionario. La clave para recuperar el objeto es el identificador del chat. Por cada clave del diccionario, se guarda en el *value* un objeto de tipo Chat. Se ha decidido almacenar en RAM un diccionario con los objetos Chat porque algunos de sus atributos no son necesario que persistan, pero sí es imprescindible que estén disponibles durante la ejecución del bot. Estos atributos son: *idList, nameTask y inlineMsg.* Estos atributos se asignan y recuperan en el transcurso de una operación por parte del usuario. La pérdida de sus valores no provoca un trastorno en el funcionamiento de la aplicación. Si el valor de uno de ellos se extraviase en mitad de una operación del usuario, el sistema sería capaz de recuperarlo al reenviar el usuario la petición (pulsando de nuevo el botón). Existe una excepción a esto y es el escenario en el cual el usuario está realizando la creación de una nueva lista. En este caso, al perder los valores no se podría continuar el caso de uso y el usuario tendría que empezarlo de nuevo. Los objetos de tipo *Task* y *Checklist* también son almacenados en un diccionario en el transcurso del caso de uso Crear Checklist. Una vez el caso de uso se da por finalizado, los objetos son eliminados de memoria y se escribe toda su información en XML.

Los archivos XML generados por la aplicación son: *checklists.xml, tasks.xml, chats.xml* y *users.xml.* No se han creado tantos archivos como tablas definidas en el apartado **Diagrama relacional** porque debido a que no se ha utilizado una base de datos SQL, guardar varios datos en un mismo campo no produce ningún error ni incongruencia entre las tablas. La tabla *Session*, es el equivalente en el archivo

chats.xml al campo *members.* Dentro de él existen tantos campos *id*, como datos de usuarios pertenecientes al grupo de los cuales dispone el bot información. Es decir, en cada campo *id* dentro de *members* se guarda una clave de un usuario del archivo *users.xml.* Los datos que habría en la tabla *Message* de una base de datos SQL se almacenan en el correspondiente campo *msg* del archivo *checklists.xml.* Tras estos cambios, los DTD[2] que siguen los citados XML son [34]:

```
<!ELEMENT checklists (checklist)* >
<!ELEMENT checklist (id, name, owner, state, msg+)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT owner (#PCDATA)>
<!ELEMENT state (CREATE|ADDTASK|EDIT|DONE)>
<!ELEMENT msg (#PCDATA)>
```

Figura 32. DTD del archivo c*hecklists.xml*.

```
<!ELEMENT tasks (task)* >
<!ELEMENT task (name, idList, owner, checked)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT idList (#PCDATA)>
<!ELEMENT owner (EMPTY | #PCDATA)>
<!ELEMENT checked (True|False)>
```

Figura 33. DTD del archivo tasks.*xml*.

```
<!ELEMENT chats (chat)+>
<!ELEMENT chat (id, language, members)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT language (ES|EN)>
<!ELEMENT members (EMPTY | idM+)>
<!ELEMENT idM (#PCDATA)>
```

Figura 34. DTD del archivo chats.*xml*.

```
<!ELEMENT users (user)+>
<!ELEMENT user (id, name, username)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT username (EMPTY | #PCDATA)>
```

Figura 35. DTD del archivo users.*xml*.

Los archivos XML referentes a chats y usuarios siempre contienen al menos un valor, porque el archivo se crea al insertar un valor y estos nunca se borran. Por el contrario, en checklists y tareas sí se eliminan valores pudiendo darse el caso de quedarse el XML únicamente con el objeto root (raíz) vacío. Debido a esto, las dos primeras DTD tienen un asterisco (*) acompañando al valor inicial y las

[2] Document type definition. Descripción de estructura y sintaxis de un documento XML

otras dos un símbolo de suma (+). Estos caracteres representan una situación similar en el resto de elementos que acompañan. El valor #PCDATA representa que el elemento al que acompaña es una cadena de texto alfanumérica. El carácter "|" representa que el elemento puede tomar única y exclusivamente los valores dados entre paréntesis. Por último, la palabra EMPTY simboliza que ese elemento está vacío. La unión de EMPTY con otro valor mediante "|" permite representar que el elemento puede estar vacío o no (y tomar el otro valor). Un ejemplo de XML generado por la aplicación y que cumple con la definición de tipo de documento mostrada es:

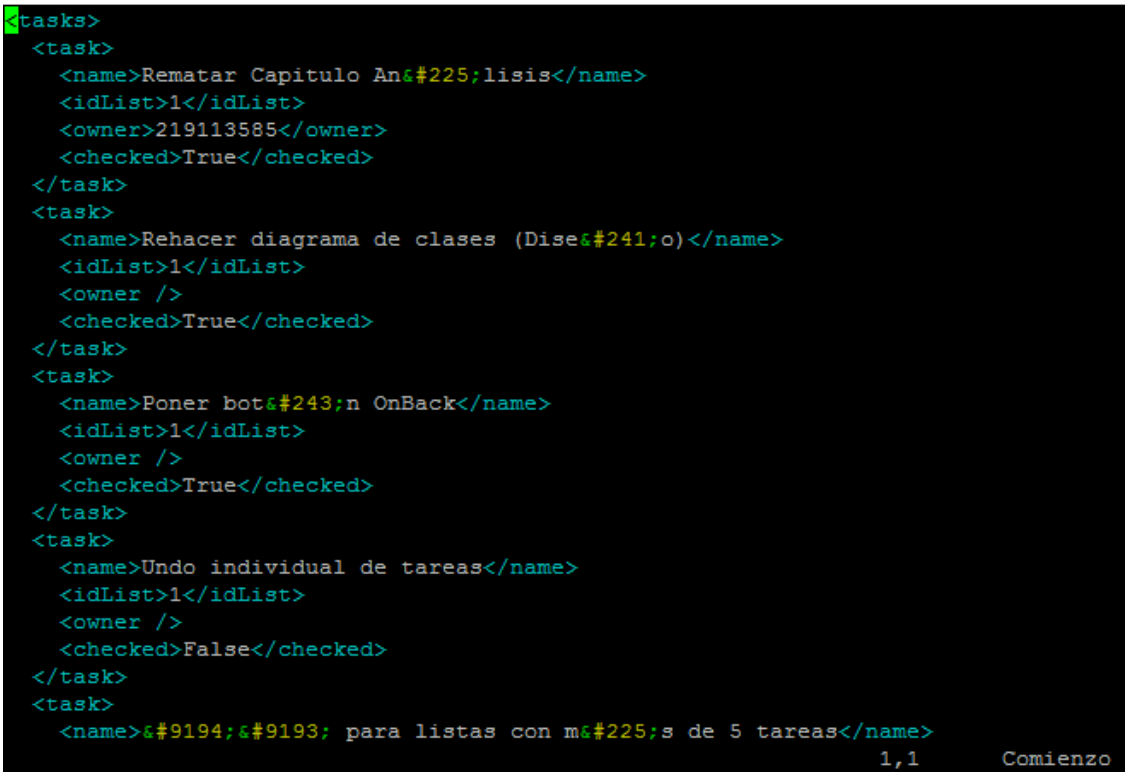

Figura 36. Ejemplo de archivo *tasks.xml* de la aplicación *ChecklistBot*.

En la figura superior se puede observar que el campo *owner* no siempre está completado y puede ser vacío (EMPTY) y que el elemento *checked* siempre toma el valor True o False. Este archivo XML puede ser exportado a Excel para facilitar su lectura. Para exportar un XML a Excel, (1) se abre el programa, (2) se abre el archivo deseado y (3) se selecciona la opción "Como tabla XML" como forma de abrir el archivo. Se obtiene una tabla similar a la siguiente:

	name	idList	owner	checked
2	Rematar Capitulo Análisis	1	219113585	True
3	Rehacer diagrama de clases (Diseño)	1		True
4	Poner botón OnBack	1		False
5	Undo individual de tareas	1		False
6	¿/new en grupos?	1		False
7	◂◂ ▸▸ para listas con más de 5 tareas	1		True
8	Psicología	5		False
9	Cambios	5		False
10	Lengua	5		False
11	Matemáticas	5		False
12	Investigación: No funciona /assign	1		True
13	Rebeca	6		True
14	Silvia	6		True
15	Papa	6		True
16	Mama	6		True
17	Guillermo	6		True
18	Violeta	6		True
19	Send msg nuevo al clonar en vez de editarlo	1		True

Figura 37. Ejemplo de archivo *tasks.xml* exportado en Excel.

En la figura 37 se puede observar fácilmente que tareas son de la misma checklist. Todas las tareas con mismo valor en la columna idList pertenecen a la misma lista. También es fácil ver si la tarea está o no asignada.

Otra diferencia con el diseño, y quizás la más destacable, es la separación de la clase *viewController* en tres distintas. Cada una de estas clases maneja exclusivamente un tipo de las peticiones que el usuario puede enviar. Es decir, las funciones pertenecientes a la clase *viewController* descrita en el capítulo de diseño se implementan finalmente como clases separadas. Esta decisión ha sido tomada para poder implementar hilos y paralelismo dentro del bot. Este cambio mejora el rendimiento de la aplicación y evita la secuencialidad. Así el bot puede resolver varias peticiones de diferentes usuarios concurrentemente.

```
class MsgHandler(telepot.helper.ChatHandler):⌨

class CallbackHandler(telepot.helper.CallbackQueryOriginHandler):⌨

class InlineHandler(telepot.helper.InlineUserHandler, telepot.helper.AnswererMixin):⌨

TOKEN = '350435553:AAFeISB-ZAk6CwsRo3goXA7vz2mOpHqrd_s'

bot = telepot.DelegatorBot(TOKEN, [
    pave_event_space()(
        per_chat_id(), create_open, MsgHandler, timeout=300),
    pave_event_space()(
        per_callback_query_origin(), create_open, CallbackHandler, timeout=120),
    pave_event_space()(
        per_inline_from_id(), create_open, InlineHandler, timeout=10),
])

MessageLoop(bot).run_as_thread()

# Keep the program running.
while True:
    time.sleep(10)
```

Figura 38. Código del archivo *checklistBot.py* que en diseño sería el controlador de vista.

En la figura superior se puede ver la estructura del archivo *checklistBot*. Se han omitido todos los *imports* del fichero. En este archivo se agrupan las tres clases que reciben y manejan cada uno de los tipos de petición existentes. Para esto, el objeto bot es creado como un *DelegatorBot* y luego es corrido en bucle como un hilo. Se crea un hilo nuevo por:

- Cada identificador de chat relacionado a un grupo de peticiones de tipo mensaje.
- Cada callback query original y todas sus posteriores peticiones relacionadas.

- Cada identificador de chat del usuario que inicia una inline query.

De esta forma, cuando un usuario envía un mensaje el bot puede responder al mensaje original sin necesidad de especificar un identificador de chat porque ese objeto de tipo *MsgHandler* solo está asociado al identificador del usuario emisor del mensaje. De forma análoga, al pulsar un usuario un botón el bot crea un objeto de tipo *CallbackHandler.* Este manejará todas las peticiones relacionadas pudiendo así editar el mensaje original donde se encuentra el primer botón pulsado sin necesidad de especificar el identificador de ese mensaje.

Todos los hilos tienen un tiempo de vida máximo fijado en el atributo *timeout.* Si el hilo no es cerrado explícitamente con *close* al acabar la operativa necesaria, se cierra al trascurrir los segundos especificados en su creación.

Se ha añadido una nueva clase a la aplicación a mayores de las descritas en diseño. Esta clase es: *resources*. Se ha implementado, nuevamente, como un módulo debido a que solo es necesario un único punto de acceso a sus variables y funciones. Este módulo contiene todos los literales usados en la aplicación y proporciona uno o varios de ellos dependiendo de la/s clave/s con la que se llame a sus funciones. Tiene dos atributos:

- **Resources_ES:** Diccionario con todos los literales en español.
- **Resources_EN:** Diccionario con todos los literales en inglés.

Ambos diccionarios tienen las mismas claves, la diferencia está en los *values* almacenados. Las funciones del módulo son:

- **getResource:** Devuelve el literal asociado a una *key* en el idioma solicitado. Ambos valores se pasan mediante parámetros de entrada.
- **getConcatResources:** Retorna la concatenación de todos los literales solicitados en el idioma pedido.
- **getCompleteResource:** Devuelve como cadena de texto el literal y el o los parámetros pasados por entrada.

Este módulo es muy útil para llevar a cabo modificaciones en los mensajes enviados por el bot. En lugar de modificar el literal en todos los sitios utilizados, únicamente se realiza el cambio en *resources.* Esto es posible, gracias a que todas las cadenas de texto de la aplicación se encuentran aisladas en un único archivo independiente del resto.

Para terminar con el capítulo de **Implementación,** se procede a comentar los métodos que se consideran de mayor interés dentro de la aplicación.

El primer método elegido pertenece al controlador de tareas y es *newTask.* Este método implementa toda la lógica necesaria para crear una nueva tarea en la aplicación. Lo primero que se comprueba en él, es si la tarea que se va a crear ya existe en la checklist. Para ello, se intenta recorrer la lista de tareas de la correspondiente checklist y, si esto falla, comprobar si ya existe en el archivo *tasks.xml* una tarea con ese nombre. El acceso a la lista de tareas de una checklist falla si la checklist no se encuentra en el proceso de creación. Los objetos de tipo Task solo se guardan en RAM hasta que se confirma la creación de la lista y pasan a almacenarse en memoria no volátil. Tal y como se especifico es el apartado de **Especificación de CU**, este método es invocado tanto desde el caso de uso “Crear Checklist” como desde el caso de uso “Editar Checklist”. Por esta razón, tras la primera bifurcación entre estar o no duplicada en la checklist, la operativa se bifurca de nuevo en dos dependiendo del estado de la lista. Este estado puede ser *CREATE* o *ADDTASK*. Se diferencia entre estas dos ramas

porque si el estado es *ADDTASK* se insertará en el archivo XML en ese instante y no en el array de tareas. Por el contrario, si la lista se encuentra en creación la tarea es añadida al array. Esta diferenciación se debe a que si añadimos una tarea al XML durante el proceso de creación y finalmente nunca se confirma su creación (mediante el envío del comando /done), la tarea se insertaría en su XML sin existir la checklist asociada en el correspondiente XML.

```
def newTask(_id, nameTask, cList):
    inList = False
    try:
        tasks = Task.Tasks[cList.getId()]
        for task in tasks:
            if task.getName() == nameTask:
                inList = True
                break

    except KeyError:
        if tasksManager.findByNameAndIdList(nameTask, cList.getId()) != None:
            inList = True

    if inList == False:
        if len(nameTask) > 50:
            return controller.message(Resources.getConcatResources(['Error_TaskLength','Checklist_AddTask'],
                Chat.Chats[_id].getLanguage()), None, None)

        else:
            task = Task(nameTask, cList.getId())

            if cList.getState() == States.ADDTASK.name: #Añade tarea a una lista ya creada
                tasksManager.insert(task)
                return controller.checklistMessage(_id, cList.getId(), Resources.getCompleteResource('Checklist_NewTask', [nameTask],
                Chat.Chats[_id].getLanguage()), True)

            elif cList.getState() == States.CREATE.name: #Añade tarea a una lista en creacion
                try:
                    Task.Tasks[cList.getId()].append(task)
                except KeyError:
                    Task.Tasks[cList.getId()] = [task]

                return controller.message(Resources.getConcatResources(['Checklist_AddTask','Checklist_Done'],
                    Chat.Chats[_id].getLanguage()), None, None)

    else:
        if cList.getState() == States.ADDTASK.name: #Añade tarea a una lista ya creada
            return controller.message(Resources.getConcatResources(['Task_Duplicate','Checklist_AddTask'],
                Chat.Chats[_id].getLanguage()), nameTask, None)

        elif cList.getState() == States.CREATE.name: #Añade tarea a una lista en creacion
            return controller.message(Resources.getConcatResources(['Task_Duplicate','Checklist_AddTask', 'Checklist_Done'],
                Chat.Chats[_id].getLanguage()), nameTask, None)
```

Figura 39. Código del método *newTask* perteneciente al módulo *tasksController*.

Para terminar de comprender el método que se acaba de explicar y los siguientes se incluye la definición de la función *message.* Esta función genera y retorna un diccionario con los parámetros de entrada que se le pasan. Las tres claves escogidas coinciden con los parámetros con los que se llama a los métodos de enviar y editar del bot definidos por Telegram.

```
def message(text, params, markup):
    msg = {
        'text': text,
        'params': params,
        'markup': markup
    }
    return msg
```

Figura 40. Código del método *message* perteneciente al módulo *controller*.

Todos los métodos de la aplicación retornan un objeto de tipo diccionario generado con el método de la figura 40. De este modo, la llamada al método de envío/edición de mensajes al usuario por parte del *viewController* es siempre idéntico y el mensaje finalmente enviado depende únicamente de los *values* que se guardan en el diccionario devuelto. Esto permite un mayor nivel de abstracción dentro de la aplicación.

Los siguientes métodos escogidos pertenecen a la clase *Controller.* Se han escogido porque son los métodos que generan el mensaje con la checklist. Estos métodos son invocados al finalizar prácticamente todos los casos de uso de la aplicación. La lógica que implementan estos métodos, junto con *enableOptions* y *printChecklist,* cubre el proceso de negocio **Mostrar Checklist.**

```
def checklistMessage(_id, idList, text, updateAll, page = 0):
    cList = checklistsManager.findById(idList)
    if _id != 0:
        cList.appendMsg((_id, None))

    #Se actualizan/editan todos los msg
    if updateAll:
        result = {}
        params = printChecklist(cList)
        for msg_id in cList.getMessages():
            if msg_id[0] == cList.getOwner():
                markup = checklistKeyboard(msg_id[0], cList, page, True)
            else:
                markup = checklistKeyboard(msg_id[0], cList, page)

            result[msg_id] = message(text, params, markup)

        return result

    #Se realiza un único envío de msg
    else:
        if cList.getOwner() == _id:
            markup = checklistKeyboard(_id, cList, page, True)
        else:
            markup = checklistKeyboard(_id, cList, page)

        if markup == None:
                return

        return message(text, printChecklist(cList), markup)
```

Figura 41. Código del método *checklistMessage* perteneciente al módulo *controller.*

El método *checklistMessage* es el que es llamado desde el resto de controladores cuando se quiere generar el mensaje con la checklist actualizada. Desde él se llama a los otros tres métodos necesarios para realizar el proceso de negocio mencionado. Primero, se busca la checklist en el archivo XML de la cual se ha pasado el identificador. Después, dependiendo del valor del atributo updateAll hace una u otra operativa. Si updateAll es True, se desea actualizar todos los mensajes asociados a esa lista. Se genera el texto con la checklist y sus tareas y estados. Luego, se recorren los correspondientes mensajes añadiendo un teclado de botones u otro dependiendo de si el mensaje está destinado al creador o no. Finalmente, se retorna un diccionario con el resultado de todos los mensajes. Si updateAll es False, se desea generar únicamente un mensaje que tiene como receptor el chat del cual se pasa el identificador como parámetro de entrada. Al igual que en la otra rama, se genera diferente teclado si el chat receptor es el del creador de la checklist. Se retorna un diccionario con los parámetros necesarios para realizar el envío del mensaje. Si el teclado generado es nulo, se devuelve nulo. Esto se explica en el siguiente método: *checklistKeyboard*.

```
def checklistKeyboard(_id, cList, page, options = False):
    auxKeyboard = []
    tasks = tasksManager.findAllByList(cList.getId())
    _min = 0
    _max = len(tasks)

    #si se llama desde un inlineQuery no tenemos el chat_id para mandar msg en idioma correcto
    #cogemos ingles por defecto
    if _id == 0:
        language = 'EN'
    else:
        try:
            language = Chat.Chats[_id].getLanguage()
        except KeyError:
            language = 'EN'

    if len(tasks) > 5:
        if page * 4 >= len(tasks):
            return
        _min = page * 4
        _max = (page + 1) * 4

    for task in tasks[_min:_max]:
        auxKeyboard.append([InlineKeyboardButton(text=str(task.getName()),
                                callback_data= 'Check-' + str(cList.getId()) + '-' + str(task.getName()))])

    if len(tasks) > 5:
        auxKeyboard.append([InlineKeyboardButton(text=Resources.getResource('Previous', language),
                                callback_data= 'Previous-' + str(cList.getId()) + '-' + str(page - 1)),
                            InlineKeyboardButton(text=Resources.getResource('Next', language),
                                callback_data= 'Next-' + str(cList.getId()) + '-' + str(page + 1))])

    if options:
        return enableOptions(_id, cList, auxKeyboard)
    else:
        auxKeyboard.append([InlineKeyboardButton(text=Resources.getResource('Clone', language),
                                callback_data= 'Clone-' + str(cList.getId()))])
        return InlineKeyboardMarkup(inline_keyboard=auxKeyboard)
```

Figura 42. Código del método *checklistKeyboard* perteneciente al módulo *controller*.

La figura 42 muestra el código de *checklistKeyboard.* Se llama desde *checklistMessage* para generar el correspondiente teclado que acompaña al mensaje de texto de la checklist. En él, primero se genera una lista de botones con las tareas pertenecientes a la lista. Se llama a este método una vez por cada mensaje para que los botones estén en el idioma seleccionado dentro del chat. Cuando existen más de cinco tareas en una checklist, solo se muestran cuatro y se añaden unos botones de "atrás" y "siguiente" para navegar entre los botones de tareas. Por defecto, no se muestran las opciones de creador de la lista a no ser que se pase el valor True específicamente al llamar a la función. Si *options* es True, se llama al método *enableOptions* para que añada los botones de "Compartir" y "Editar". Este método retorna un objeto de tipo *InlineKeyboardMarkup* con los botones que se han ido añadiendo durante su ejecución. Este objeto es enviado en el campo *reply_markup* al invocar a los métodos de envío de mensajes de Telegram. El teclado retornado se compone de botones (objetos de tipo *InlineKeyboardButton).* Estos objetos tienen los atributos:

- Text: texto mostrado al usuario en el botón
- Callback_data: Dato que recibe la función *on_callback_query* al ser pulsado el botón

Capítulo 6. Batería de pruebas

Al alcanzar los objetivos propuestos en la realización de una aplicación software, obtenemos, a priori, un producto que cumple con los requisitos fijados por el cliente y expuestos en la etapa de análisis. Para asegurar que el proyecto cumple con todos los requisitos, tanto funcionales como no funcionales, este debe pasar una serie de pruebas. Las pruebas elegidas deben intentar cubrir todos los posibles escenarios que pueden darse dentro de la aplicación. La aplicación debe comportarse como se especifica en cada una de las pruebas para lograr superar la batería de pruebas.

Las baterías de pruebas son un modo de asegurar la calidad del producto software **(QA - Quality Assurance).** Existen multitud de tipos de pruebas, pero todas se pueden clasificar dentro de uno de estos tres grupos (Caja Blanca, Caja Negra, Caja Gris) según la accesibilidad que se tenga sobre los elementos del sistema a evaluar. Las pruebas de caja negra solo tienen en cuenta las entradas que reciben y las salidas que producen. Por el contrario, en pruebas de caja blanca se tiene en cuenta el funcionamiento interno que da lugar a salidas concretas. Las pruebas de caja gris son pruebas intermedias en las que el *tester* dispone de conocimientos limitados del funcionamiento interno del programa que está probando. También, pueden clasificarse según el nivel al que se llega, en: Unitarias y de Integración. En las pruebas unitarias se prueba una funcionalidad, componente o método de manera independiente, y en las de Integración se prueba el funcionamiento de este en conjunto con el resto del sistema. Y, por último, si la clasificación se basa en la ejecución del producto las pruebas pueden ser diferenciadas en: funcionales y no funcionales. Esta separación es análoga a la realizada sobre los requisitos [35].

En este proyecto, podríamos diferenciar entre pruebas de caja negra para todas las relacionadas con llamadas al servidor de Telegram y pruebas de caja blanca para las realizadas sobre el resto de funciones del bot desarrollado. Las pruebas especificadas y realizadas en las siguientes páginas son, en su mayoría, **pruebas de caja blanca, integradas** y **funcionales**. Para pasar estas pruebas es recomendable acompañar su realización a la lectura del Anexo I.

Seleccionar lenguaje al iniciar bot por primera vez		
Descripción	Al iniciar una conversación con el bot, el usuario selecciona el lenguaje deseado.	OK
Comportamiento esperado	El bot crea un elemento *chat* y uno *user* referente a ese usuario en los XML, almacena el lenguaje seleccionado y escribe al usuario en ese lenguaje siempre	

No seleccionar lenguaje al iniciar bot por primera vez		
Descripción	Al iniciar una conversación con el bot, el usuario no selecciona ningún lenguaje.	OK
Comportamiento esperado	El bot crea un elemento *chat* y uno *user* referente a ese usuario en los XML, almacena el lenguaje por defecto (inglés) y escribe al usuario en ese lenguaje siempre	

Seleccionar lenguaje mediante comando		
Descripción	El usuario sigue el procedimiento normal de selección de lenguaje mediante el envío del comando.	OK
Comportamiento esperado	El bot actualiza el elemento *chat* referente a ese usuario en el XML, almacena el nuevo lenguaje seleccionado y escribe al usuario en ese lenguaje siempre	

Inicia bot por segundas o posteriores veces		
Descripción	Inicia una conversación con el bot un usuario que ya había iniciado conversación con él en anteriores ocasiones.	OK
Comportamiento esperado	El bot no crea un nuevo elemento *chat* ni uno *user* referente a ese usuario en el XML, porque ya existe uno almacenado. Escribe al usuario siempre en el lenguaje seleccionado anteriormente.	

Visualizar ayuda mediante comando		
Descripción	El usuario sigue el procedimiento normal para visualizar la ayuda del bot mediante el envío del comando.	OK
Comportamiento esperado	El usuario recibe un mensaje con la ayuda del bot.	

Crea checklist		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist.	OK
Comportamiento esperado	El bot crea un nuevo elemento *checklist* y varios elementos *tasks* en los XML correspondientes. El usuario recibe un mensaje con la lista que acaba de crear.	

Crea checklist con nombre ya existente		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist, pero proporciona un nombre que ya tiene otra de sus listas.	OK
Comportamiento esperado	El bot le proporciona otros nombres alternativos durante el procedimiento y finalmente crea un nuevo elemento *checklist* y varios elementos *tasks* en los XML correspondientes. El usuario recibe un mensaje con la lista que acaba de crear.	

Crea checklist con nombre "comando"		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist, pero proporciona como nombre un comando del bot.	OK
Comportamiento esperado	El bot pasa a dar respuesta al comando enviado y deja sin efecto la creación de la nueva checklist.	

Crea checklist con tarea "comando"		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist, pero proporciona como tarea un comando del bot.	OK
Comportamiento esperado	El bot pasa a dar respuesta al comando enviado y deja sin efecto la creación de la nueva checklist.	

Crea checklist con tareas repetidas		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist, pero proporciona varias tareas idénticas.	OK
Comportamiento esperado	El bot informa al usuario y no añade las tareas duplicadas a la lista. Finalmente crea un nuevo elemento *checklist* y varios elementos *tasks* en los XML correspondientes. El usuario recibe un mensaje con la lista que acaba de crear.	

Crea checklist con tareas de longitud mayor a la permitida		
Descripción	El usuario sigue el procedimiento normal de creación de una checklist, pero proporciona tareas de longitud mayor a la permitida.	OK
Comportamiento esperado	El bot informa al usuario y no añade las tareas que exceden la longitud a la lista. Finalmente crea un nuevo elemento *checklist* y varios elementos *tasks* en los XML correspondientes. El usuario recibe un mensaje con la lista que acaba de crear.	

Crea checklist desde chat grupal		
Descripción	El usuario sigue el procedimiento normal de creación de una *checklist*, pero desde un chat grupal.	OK
Comportamiento esperado	El bot informa al usuario y deja sin efecto la creación de una nueva checklist.	

Ver mis checklists		
Descripción	El usuario sigue el procedimiento normal para visualizar sus checklists.	OK
Comportamiento esperado	El bot genera y envía al usuario un mensaje con una lista de todas las checklists que tiene creadas.	

Ver mis checklists desde chat grupal		
Descripción	El usuario sigue el procedimiento normal para visualizar sus *checklists*, pero desde un chat grupal.	OK
Comportamiento esperado	El bot informa al usuario y deja sin efecto la creación de una nueva checklist.	

Añadir tarea a una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción "Añadir tarea".	OK
Comportamiento esperado	El bot crea un nuevo elemento *task* en el XML correspondiente. El usuario recibe un mensaje con la lista que actualizada. El mensaje también es editado en todos los chats en los que ha sido compartida la lista.	

Añadir tarea duplicada a una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Añadir tarea” pero añade una tarea repetida.	OK
Comportamiento esperado	El bot informa al usuario y solicita una nueva tarea hasta que el usuario no envíe una que ya existe. Entonces, crea un nuevo elemento *task* en el XML correspondiente. El usuario recibe un mensaje con la lista que actualizada. El mensaje también es editado en todos los chats en los que ha sido compartida la lista.	

Modificar tarea de una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Modificar tarea”.	OK
Comportamiento esperado	El bot actualiza el elemento *task* en el XML correspondiente con el nuevo nombre de la tarea y el estado “sin hacer”. El usuario recibe un mensaje con la lista que actualizada. El mensaje también es editado en todos los chats en los que ha sido compartida la lista.	

Modificar tarea duplicada de una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Modificar tarea” pero modifica la tarea a una que ya existe en la lista.	OK
Comportamiento esperado	El bot informa al usuario y solicita una nueva tarea hasta que el usuario no envíe una que ya existe. Entonces, actualiza el elemento *task* en el XML correspondiente con el nuevo nombre de la tarea y el estado “sin hacer”. El usuario recibe un mensaje con la lista que actualizada. El mensaje también es editado en todos los chats en los que ha sido compartida la lista.	

Eliminar tarea de una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Eliminar tarea”.	OK
Comportamiento esperado	El bot elimina el elemento *task* del XML correspondiente. El mensaje es editado en todos los chats en los que ha sido compartida la lista, incluyendo el del usuario creador.	

Confirmar eliminar última tarea de una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Eliminar tarea” cuando la checklist tiene una única tarea.	OK
Comportamiento esperado	El bot informa al usuario y elimina el elemento *checklist* y *task* de los XML correspondientes al confirmar el usuario la acción. El mensaje es editado en todos los chats en los que ha sido compartida la lista, incluyendo el del usuario creador, informando del borrado de esa checklist.	

No confirmar eliminar última tarea de una checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Eliminar tarea” cuando la checklist tiene una única tarea.	OK
Comportamiento esperado	El bot informa al usuario y el procedimiento queda sin efecto al no confirmar el usuario la acción. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Reinicializar checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Reinicializar checklist”.	OK
Comportamiento esperado	El bot actualiza todos los elementos *tasks* de esa lista en el XML correspondiente al estado “sin hacer”. El mensaje es editado en todos los chats en los que ha sido compartida la lista, incluyendo el del usuario creador.	

Confirmar eliminar checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Eliminar checklist”.	OK
Comportamiento esperado	El bot actualiza todos los elementos *tasks* de esa lista en el XML correspondiente al estado “sin hacer”. El mensaje es editado en todos los chats en los que ha sido compartida la lista, incluyendo el del usuario creador.	

No confirmar eliminar checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, escogiendo la opción “Eliminar checklist”.	OK
Comportamiento esperado	El bot informa al usuario y el procedimiento queda sin efecto al no confirmar el usuario la acción. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Volver atrás al editar checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para editar sus *checklists*, pero escoge la opción de volver atrás.	OK
Comportamiento esperado	El procedimiento queda sin efecto al seleccionar ninguna opción de edición. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Compartir checklist ya creada		
Descripción	El usuario creador sigue el procedimiento normal para compartir sus *checklists*.	OK
Comportamiento esperado	El bot actualiza el elemento *checklist* en el XML correspondiente añadiendo un nuevo elemento *msg* con el id del mensaje donde se encuentra la lista compartida. El bot envía un mensaje con la checklist al chat seleccionado. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Marcar tarea		
Descripción	El usuario sigue el procedimiento normal para marcar una tarea de una *checklist*.	OK
Comportamiento esperado	El bot actualiza el elemento *task* en el XML correspondiente al estado "hecha". El mensaje de esa checklist se actualiza en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Marcar tarea ya hecha		
Descripción	El usuario sigue el procedimiento normal para marcar una tarea de una *checklist*.	OK
Comportamiento esperado	El bot informa al usuario y el procedimiento queda sin efecto. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Marcar tarea asignada		
Descripción	El usuario al cual está asignada la tarea sigue el procedimiento normal para marcar una tarea de una *checklist*.	OK
Comportamiento esperado	El bot actualiza el elemento *task* en el XML correspondiente al estado "hecha". El mensaje de esa checklist se actualiza en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Marcar tarea asignada sin ser el asignado		
Descripción	El usuario sigue el procedimiento normal para marcar una tarea de una *checklist*.	OK
Comportamiento esperado	El bot informa al usuario y el procedimiento queda sin efecto. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Asignar tarea seleccionando usuario		
Descripción	El usuario sigue el procedimiento normal para asignar una tarea de una *checklist*.	OK
Comportamiento esperado	El bot actualiza el elemento *task* en el XML correspondiente al *owner* seleccionado. El chat grupal recibe un mensaje informando de la asignación. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Asignar tarea a usuario con alias		
Descripción	El usuario sigue el procedimiento normal para asignar una tarea de una *checklist* pero proporciona un usuario con alias.	OK
Comportamiento esperado	El bot informa al usuario y el procedimiento queda sin efecto. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Asignar tarea a usuario sin alias		
Descripción	El usuario sigue el procedimiento normal para asignar una tarea de una *checklist* y proporciona un usuario sin alias.	OK
Comportamiento esperado	El bot actualiza el elemento *task* en el XML correspondiente al *owner* seleccionado. Crea un nuevo elemento *user* (si el usuario no estaba ya almacenado) y un nuevo elemento *id* dentro de *members* en el correspondiente elemento *chat* para los respectivos XML. El chat grupal recibe un mensaje informando de la asignación. El mensaje de esa checklist queda sin cambios en todos los chats en los que ha sido enviado, incluyendo el del usuario creador.	

Checklist con más de cinco tareas al crear		
Descripción	El usuario sigue el procedimiento normal para crear una *checklist* y añade más de cinco tareas.	OK
Comportamiento esperado	El bot crea los elementos *checklist y tasks* necesarios en los XML correspondientes. El usuario recibe un mensaje con la checklist que incluye botones de navegación entre páginas de botones de tareas.	

Checklist con más de cinco tareas al añadir		
Descripción	El usuario añade una tarea a una *checklist* ya creada sobrepasando las cinco tareas.	OK
Comportamiento esperado	El bot crea el elemento *task* necesario en el XML correspondiente. El mensaje de esa checklist se actualiza en todos los chats en los que ha sido enviado, incluyendo el del usuario creador, añadiendo los botones de navegación entre páginas de botones de tareas.	

Navegación entre páginas de botones de tareas		
Descripción	El usuario pasa a la página siguiente y se le muestran las cuatro (o restantes) tareas siguientes; pasa a la página anterior y se le muestran las cuatro tareas anteriores	OK
Comportamiento esperado	Los botones de tareas del mensaje de esa checklist se actualizan solamente en el chat en el que se ha pulsado uno de los botones de paso de páginas.	

Navegación entre páginas de botones de tareas en la primera página		
Descripción	El usuario pasa a la página anterior estando en la primera página.	OK
Comportamiento esperado	El procedimiento queda sin efecto y no se modifica el mensaje de esa checklist en el chat.	

Navegación entre páginas de botones de tareas en la última página		
Descripción	El usuario pasa a la página siguiente estando en la última página.	OK
Comportamiento esperado	El procedimiento queda sin efecto y no se modifica el mensaje de esa checklist en el chat.	

Clonar checklist		
Descripción	El usuario sigue el procedimiento normal para clonar una checklist.	OK
Comportamiento esperado	El bot crea un nuevo elemento *checklist* y varios elementos *tasks* en los XML correspondientes. Estos son idénticos a los elementos del original con la diferencia de que el propietario es el usuario clonador y la lista está en su estado "inicial". El bot informa al usuario mediante ventana emergente de que la checklist se ha clonado. El usuario puede visualizar la nueva checklist con el comando /checklists.	

Clonar checklist duplicada		
Descripción	El usuario sigue el procedimiento normal para clonar una checklist teniendo ya una con el mismo nombre.	OK
Comportamiento esperado	El bot informa al usuario mediante ventana emergente de que la checklist no se ha clonado y el procedimiento queda sin efecto.	

Después de realizar múltiples pruebas se ha localizado un error al crear una checklist con nombre o tarea empezada por el carácter "<". También, se da si se añade o se modifica una tarea con esta condición. Cuando esto sucede la checklist queda inservible porque no se genera y envía nunca el mensaje de la lista al usuario. Este error es originado al devolver Telegram el error *"Button data invalid"* cuando se intenta generar el mensaje de esa checklist. Se ha investigado si existían más caracteres inválidos para restringir los nombres de checklists/tareas, pero no se ha encontrado información al respecto. Debido a que es un error muy concreto y localizado, no se encuentra controlado.

Capítulo 7. Mejoras propuestas

Se han descubierto dos vías principales para el futuro desarrollo del proyecto *checklistBot.* La primera es la modificación del mecanismo de persistencia de la aplicación y la segunda la ampliación de las funcionalidades proporcionadas por el bot.

Modificación capa de persistencia

Aprovechando el estudio realizado en la etapa de diseño referente a la persistencia de la aplicación, se podría reemplazar el sistema de almacenado de datos en archivos XML por una base de datos MySQL. Los diagramas entidad-relación y relacional se han desarrollado cumpliendo con especificaciones y requerimientos MySQL (por ejemplo: integridad referencial). El paso del guardado de datos en XML por un servidor SQL debería ser dado una vez la aplicación tenga una cantidad considerable de usuario. En el supuesto de que eso sucediera, el almacenaje en XML se quedaría corto y afectaría al rendimiento de la aplicación. Los tiempos de respuesta en operaciones de insertado, recuperado, actualizado y eliminado en bases de datos son mejores que en archivos. Esto se debe a que las bases de datos están concretamente preparadas y optimizadas para este tipo de operaciones. No se ha llevado a cabo esta estructura de la aplicación desde el principio del proyecto debido a las dificultades que esto conllevaba. Para disponer de una base de datos MySQL es necesario un segundo servidor donde alojar la base de datos. Además, la persistencia mediante archivos es suficiente para un uso cotidiano de la aplicación por un grupo pequeño de personas.

Los cambios necesarios para implementar la nueva base de datos a la aplicación son:

- Realización del script basado en el diagrama relacional (Anexo II).
- Volcado de datos de los XML a las tablas.
- Cambios en los módulos gestores para adaptarlos a la nueva base de datos.

Nuevas funcionalidades

Comando /tasks

La funcionalidad de asignar una tarea a un usuario concreto es una opción muy interesante si el creador de una checklist quiere realizar un control de quien realiza ciertas tareas. Permite restringir la realización de una tarea concreta a una persona específica. Al asignar la tarea, el bot envía un mensaje informando de la asignación realizada. A partir de ese mensaje, la aplicación no envía más información acerca de esa asignación ni de ninguna otra. Únicamente se puede conocer si una tarea está asignada intentando marcarla. Si no se es la persona a la cual está asignada se muestra un mensaje que informa de ello y de a quien está asignada. Sin embargo, si se es la persona a la cual esta asignada simplemente se marcará como hecha. Del detalle de cómo opera la aplicación cuando una tarea es asignada, se puede concluir que no existe forma para los usuarios de saber que tareas tienen asignadas. De este vacío de información, se ha creado esta nueva funcionalidad. El comando */tasks* es análogo al comando */checklists*. Cuando un usuario envía este nuevo comando obtiene una lista con todas las tareas que tiene asignadas y aún no ha realizado. En este mensaje se mostrará para cada tarea: la lista a la que pertenece y un link para obtener esta lista en un mensaje. Este mensaje con la lista se comporta igual que cuando la lista es compartida en un chat. Es decir, se actualiza simultáneamente al resto. Desde ahí se puede marcar la tarea hecha y el cambio se mostrará automáticamente en el resto de mensajes asociados a esa checklist.

Gracias a este comando, un usuario puede realizar un seguimiento de las tareas que otros usuarios le han asignado. Esta nueva funcionalidad podría mejorarse para obtener también las tareas que ya han sido realizadas y un baremo de tareas hechas frente al total. Para añadir esto, la mejor idea sería añadir un botón en la parte inferior del mensaje que permita obtener el resultado total. Un primer prototipo de la estructura de este mensaje sería el siguiente:

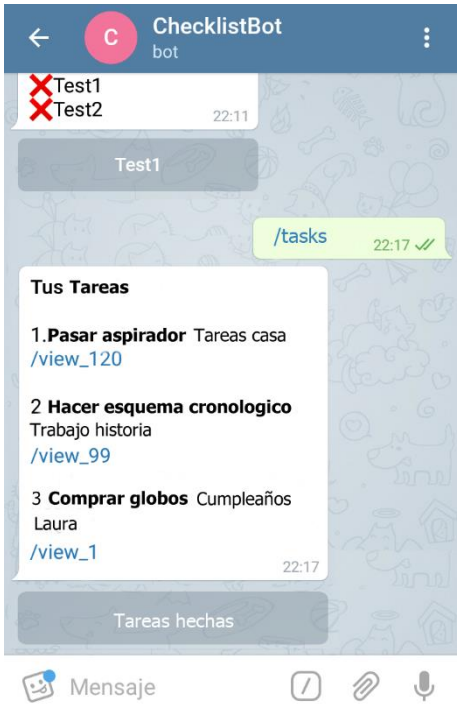

Figura 43. Pantalla ejemplo envío comando */tasks* y respuesta del bot.

Al pulsar en el botón de “Tareas Hechas” se mostrarían las tareas asignadas que ya han sido realizadas. Para no saturar el mensaje con las tareas realizadas, sería conveniente mostrar

únicamente las realizadas en la última semana, por ejemplo. Para introducir esta restricción haría falta un nuevo atributo en la clase Task que guardase la fecha en la cual una tarea ha sido realizada. Luego el mensaje de tareas realizadas se obtendría comparando la fecha de realización de cada tarea con la actual. Si la diferencia es menor de siete días la tarea sería mostrada en el mensaje al usuario.

Mejora de la funcionalidad clonar

La aplicación final dispone de la posibilidad de clonar una checklist que otro usuario ha compartido. Actualmente, la checklist se copia dentro de las listas del usuario que inicia la operativa de clonar, si y solo si, el nombre que tiene la checklist no se encuentra ya entre las listas del usuario. Esto impide duplicar la lista:

- Si el usuario que inicia la operativa es el propio creador de la checklist.
- Si el usuario que desea clonar la checklist ya tiene una suya con el mismo nombre.

Por la primera de las razones, el botón de clonar no se muestra al propietario de la checklist. Si se clonara la checklist con un nuevo nombre, este problema no existiría y un usuario siempre podría duplicar una lista. Para ello, se podría solicitar un nuevo nombre al usuario que desea clonar la lista o generar uno nuevo añadiendo el sufijo "_copy" y/o algún número al final de la lista. Esta evolución de la funcionalidad "Clonar", tendría asociada la adición de, al menos, un nuevo método en el *checklistsController* para cambiar el nombre de una lista. Este método podría usarse para añadir una nueva opción de edición a las listas: "Modificar nombre Checklist".

Usando la opción de clonar una checklist con un nuevo nombre, un usuario podría reutilizar una checklist como "plantilla" para la realización de tareas recurrentes en sucesivos días.

Enlace de listas

Esta idea se basa en añadir el botón "Tareas Hechas" a las checklists que el usuario seleccione como "enlazadas". Cuando una checklist se crea enlazada a otra, la realización de una de las tareas de la primera implica que: (1) la tarea realizada es eliminada de la primera checklist y (2) la tarea es añadida a la segunda checklist. Esto propone una nueva forma de ver el nivel de consecución de una checklist. En lugar de ver las tareas realizadas y no realizadas en una misma lista, todas las tareas no realizadas estarían en una checklist y las realizadas en otra diferente de la primera. Una de las ventajas de esta idea es ver más fácil y claramente las tareas que quedan por hacer. El mensaje con la checklist quedaría menos cargado de información y se generarían botones únicamente para las tareas de esa primera checklist, es decir, para tareas aún sin realizar.

Otra ventaja de esta nueva forma de ver las checklists, es la de poder "proteger" las tareas que ya se han realizado al compartir una checklist. Actualmente, cuando un usuario comparte una de sus checklists, el chat en el que es compartida recibe la lista completa. En el mensaje de la lista, el resto de usuarios pueden ver tanto las tareas sin hacer como las ya realizadas. Si las tareas se dividieran en dos listas, el propietario de la lista puede únicamente compartir la checklist con las tareas que están aún sin realizar y no mostrar al usuario receptor las que él mismo ya ha realizado antes de compartir la lista.

Un ejemplo práctico en el que una tarea realizada en una lista supone una tarea de "otra" checklist es el de la lista de la compra. Una vez se compra algo y se marca esa tarea como hecha, esta desaparecería de la checklist con nombre "Lista de la compra" y pasaría a pertenecer a la checklist "Gastos". Podríamos pasar a otra persona la "Lista de la compra" sin que él supiera los "Gastos" que ya se han realizado.

Podrían llegar a enlazarse varias listas a una misma checklist destino, lo que conllevaría que toda tarea realizada en cualquier de las primeras pasase a formar parte de la checklist enlazada. Esta checklist donde se agrupan todas las tareas realizadas de distintas checklists podría tener utilidad en algunos casos. Continuando con el ejemplo anterior, un usuario crea y realiza una "Lista de la compra" diferente cada semana. La realización de una tarea en cada checklist "Lista de la compra" supone finalmente un gasto. Toda tarea realizada en cualquiera de las listas es un gasto monetario que se apuntaría por igual en la checklist "Gastos". Esta nueva funcionalidad permitiría un mayor control en estos casos. Se podría realizar una nueva checklist enlazada "Gastos" cada mes, lo que permitiría al usuario ver un cómputo global de compras por mes. Esta mejora casa muy bien con la anterior. Implementando ambas se podría clonar cada día "Lista de la compra" con los productos básicos, como por ejemplo el pan, y cada compra de este producto o cualquier otro de los introducidos en cada checklist se añadiría a la checklist destino "Gastos"

Capítulo 8. Conclusiones

Inicialmente, se planteó un bot que cubriera con el manejo de checklists entre varios usuarios de manera simultánea y a tiempo real. Durante la primera etapa de concepción dentro del proceso unificado, se amplió el alcance del proyecto posibilitando la asignación de tareas a usuarios concretos. A través de la mitad final, surgieron nuevos añadidos como la posibilidad de reinicializar una lista para facilitar su reutilización o la de clonar las listas de otro usuario. La idea inicial y las mejoras que han ido surgido durante el desarrollo de la aplicación han sido alcanzadas con éxito. La funcionalidad de clonar tiene aún que mejorarse y completarse mediante la mejora propuesta en el apartado anterior, pero el resto de objetivos del proyecto se han logrado correctamente. En general, creo que se ha completado el proyecto satisfactoriamente.

Por otro lado, la realización de este proyecto ha sido una experiencia muy enriquecedora para mí, la autora. Después de años sin usar Python, he podido volver a retomar su aprendizaje y conocer conceptos más avanzados de este lenguaje. La realización del análisis y diseño en UML, el cual está orientado a lenguajes como C# o Java, y la transición de estos diagramas al código me ha permitido observar en detalle las diferencias entre estos lenguajes. Con esto, he podido ver las ventajas e inconvenientes de Python, un lenguaje interpretado muy versátil.

El desarrollo del proyecto ha sido también un gran reto. Me ha obligado a organizarme desde el principio en base a la consecución del proyecto día a día a pesar de tener lejos la fecha de entrega. Ha sido el primer trabajo de esta magnitud que he realizado sola y no en grupo, por lo que, además me ha servido para aprender a buscar soluciones y resolver situaciones por mí misma.

A mayores, he podido adentrarme en el mundo de los bots. He descubierto la amplia variedad de posibilidades que ofrecen. La finalización de este proyecto me lleva a pensar en nuevas ideas que podrían ser viables para un bot para Telegram. Me ha gustado realizar una aplicación de este estilo porque está destinada a todo tipo de usuarios y es importante tener muy presente en todo momento la interfaz de usuario y la forma en la que interactúa el bot. He tenido en cuenta las opiniones de los *testers* de la aplicación para, finalmente, hacerla lo más intuitiva y sencilla posible. Con este objetivo, se han reducido las interacciones del usuario mediante mensajes de texto a las mínimas posibles. La mayoría de operaciones se realizan a través de botones.

En conclusión, considero que se han alcanzado todos los objetivos fijados inicialmente e incluso algunos que han aparecido según se avanzaba en el desarrollo.

Anexo I. Manual de usuario

En las siguientes páginas, se proporciona un manual que facilita a un usuario la comprensión y utilización del bot @checklisttfg_bot.

Iniciar bot

Para comenzar a utilizar la aplicación es necesario tener instalada la aplicación Telegram en el dispositivo móvil u ordenador. Telegram se puede descargar para dispositivos Android desde "Google Play" y desde "Apple Store" para móviles Apple. El instalador de la versión de escritorio se encuentra en el enlace: https://desktop.telegram.org/. También se puede hacer uso del bot a través de la web que Telegram proporciona para ser usado de forma online: https://web.telegram.org/. Una vez el usuario ha accedido a su cuenta en Telegram, debe empezar una conversación con el bot para acceder a todas las funcionalidades que este proporciona. Para iniciar un chat con el bot se deben seguir los pasos mostrados en las imágenes a continuación.

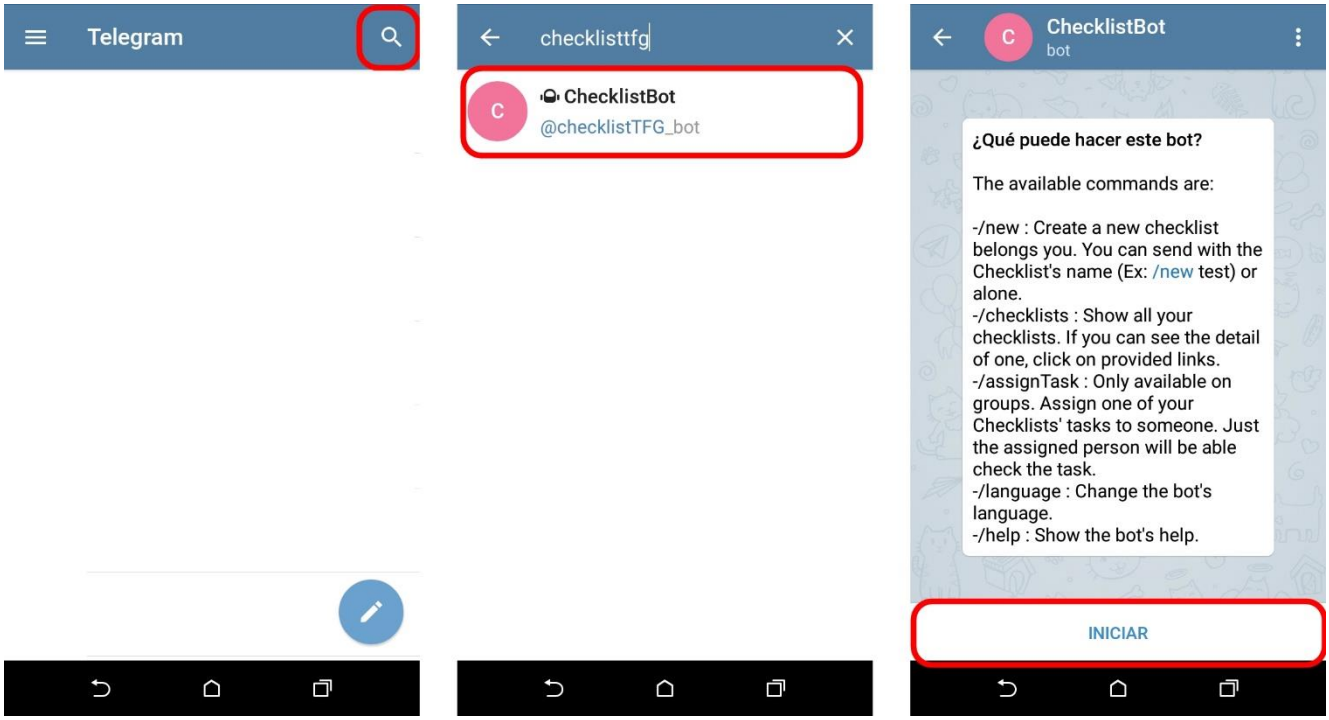

Figura 44. Pasos para iniciar conversación con el bot.

Cuando iniciamos conversación con el bot aparece una breve ayuda de uso del bot, tal y como muestra la tercera imagen de la figura 44. Al iniciar el bot por primera vez, este solicita el lenguaje en

el que el usuario quiere que el bot le escriba. El lenguaje seleccionado puede ser modificado posteriormente enviando el comando /language. La ayuda del bot también puede volver a consultarse enviando el comando /help en cualquier momento. Es posible hacer uso de la aplicación sin necesidad de instalar nada a mayores de Telegram.

Crear nueva checklist

Si el usuario quiere crear una checklist, dentro del chat con el bot debe escribir el comando /new. En el propio chat, Telegram proporciona todos los comandos disponibles para este bot pulsando en el botón:

Este acceso directo ahorra al usuario escribir a mano cada comando. Se encuentra en la parte inferior derecha de la pantalla, junto con los botones de adjuntar archivo y grabar audio.

Al enviar /new, el bot responde solicitando un nombre para la lista. Al proporcionar un nombre, el bot pide las tareas que el usuario quiere introducir en esa lista. El usuario puede terminar y confirmar la creación de la checklist enviando /done. Este comando es enviado en cada mensaje que el bot envía durante la creación de una nueva lista para permitir al usuario pulsarlo en vez de escribirlo. Al enviar /done, el bot genera un mensaje con la lista acompañado de un botón por cada tarea introducida. En la figura 20 se muestra un ejemplo con todos los pasos necesarios para la creación de una lista.

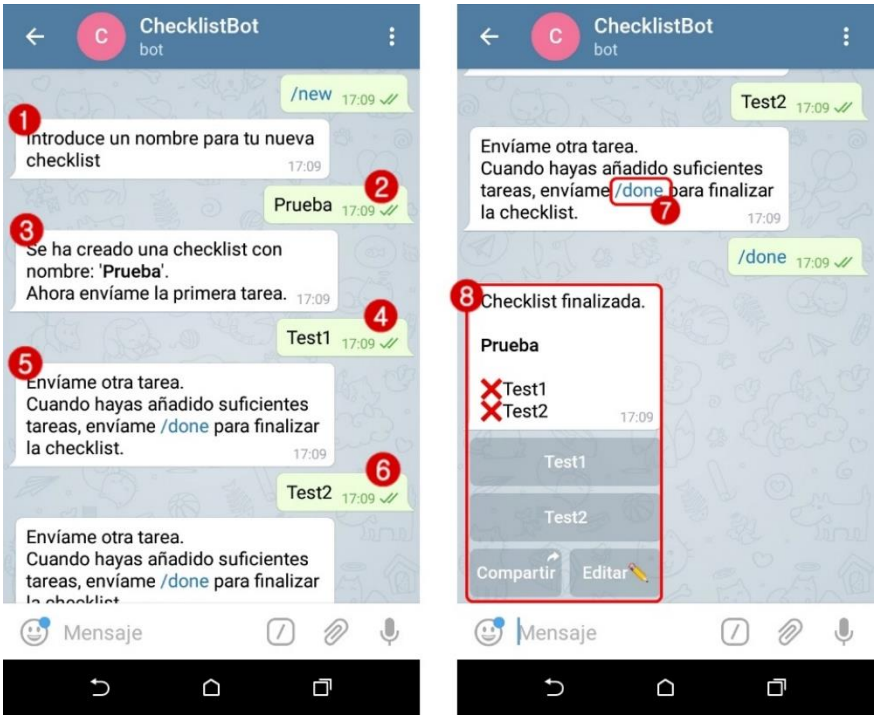

Figura 45. Pasos para crear una nueva checklist.

Si el usuario ya dispone entre sus checklists de una con el nombre proporcionado, se muestran al usuario una serie de nombres alternativos basados en el inicial. Una vez el usuario seleccione uno de estos, el bot procede de la manera habitual.

Marcar hecha una tarea

Para marcar el estado de una tarea a “hecha”, el usuario debe pulsar en el botón correspondiente a la tarea entre los botones que acompañan el mensaje de una checklist. Automáticamente la tarea se marcará como hecha en el mensaje y se informa de la actualización mediante un mensaje en la parte superior del chat.

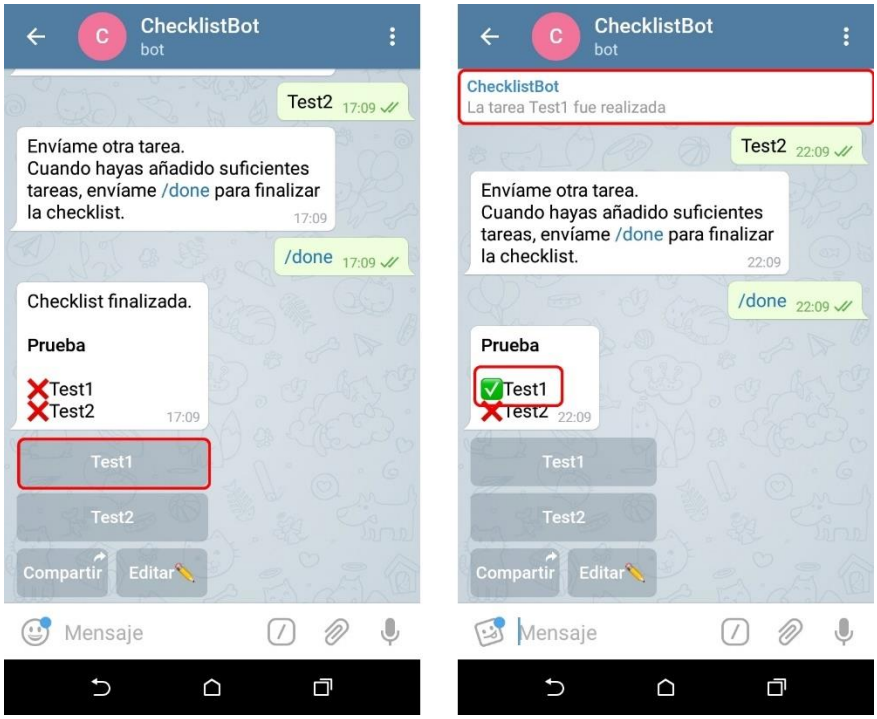

Figura 46. Pasos para marcar una tarea de una checklist.

Si un usuario pulsa el botón de una tarea que ya se encuentra hecha, la aplicación informará de ello al usuario mediante un popup y no se producirá ningún cambio.

Si un usuario pulsa sobre el botón de una tarea que se encuentra asignada a otro usuario, la aplicación informará de ello al usuario a través de un popup y la tarea permanecerá sin hacer.

Editar checklist

Para editar una lista, el usuario debe pulsar en el botón 

La opción de editar sólo es realizable por el creador sobre su propia checklist. Por esta razón, este botón aparece dentro de la lista de botones que acompañan el mensaje de una checklist, si y solo si, el mensaje ha sido enviado al chat privado entre el creador y el bot. En cualquier otro caso, este botón no será visible.

Al pulsar en el botón de editar, se muestran una serie de opciones de edición. Para seleccionar una de ellas, se debe pulsar sobre la opción deseada. Dependiendo del botón pulsado se procederá a un tipo de modificación sobre la checklist diferente. Cada una de estas modificaciones sigue unos pasos distintos. En la figura 21 se pueden ver las opciones disponibles.

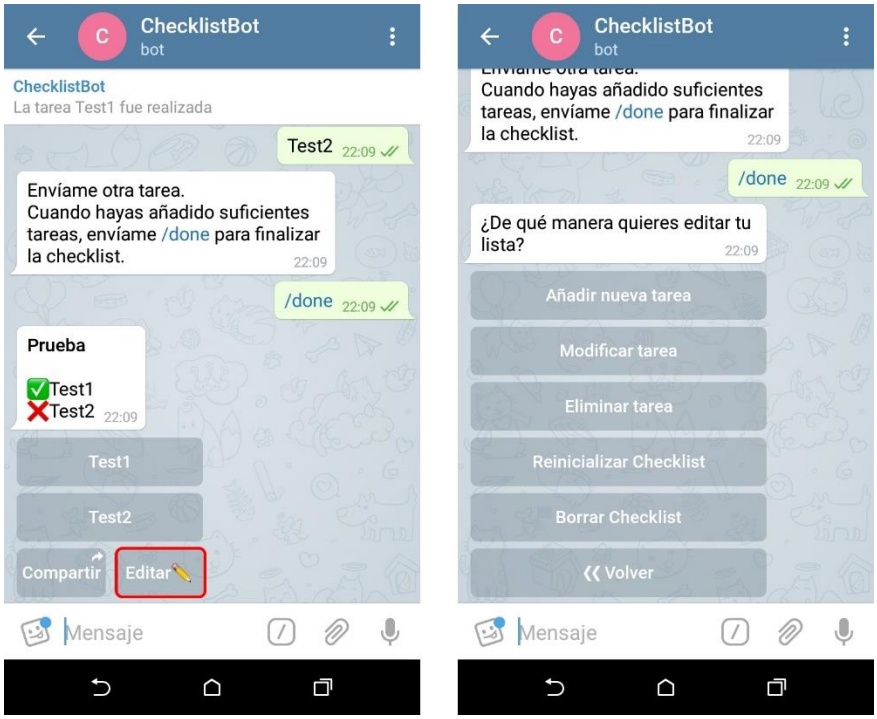

Figura 47. Pasos para editar una checklist.

Si finalmente el usuario desea no realizar ninguna modificación sobre la checklist, debe pulsar el último botón: **"Volver"** y se le mostrará la checklist de nuevo.

Añadir tarea

Para insertar una nueva tarea a una lista ya creada el usuario debe seleccionar la opción: **"Añadir nueva tarea"** de las mostradas en el paso anterior**.**

El bot solicitará la nueva tarea. Cuando el usuario envíe un mensaje con la tarea, la aplicación la añadirá a la correspondiente checklist y la mostrará actualizada. Se incluye un texto explicativo de la modificación realizada en la parte superior del mensaje.

Se puede ver en la segunda imagen de la figura inferior el procedimiento mencionado: (1) el bot solicita la tarea, (2) el usuario envía la nueva tarea y (3) el bot actualiza la checklist.

Siempre que el usuario esté añadiendo una nueva tarea a una lista, se comprobará si la tarea ya existe en esa checklist. Si es así, no se añadirá y se mostrará un mensaje informativo solicitando de nuevo la tarea. El bot también actúa así en la creación de una nueva checklist.

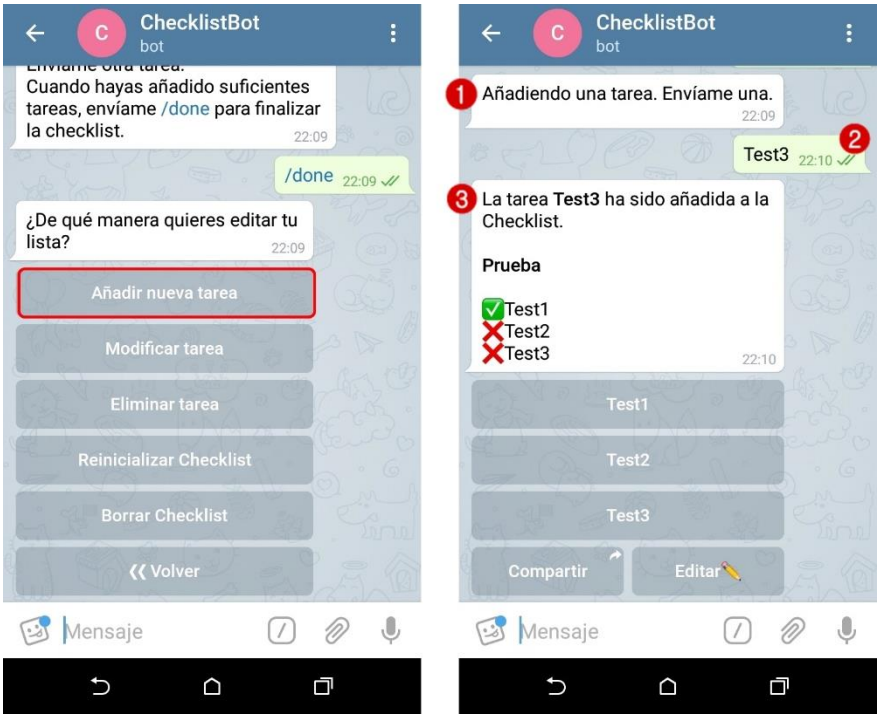

Figura 48. Pasos para añadir una tarea.

Modificar tarea

Para modificar una tarea ya existente dentro de una checklist por otra nueva, el usuario debe pulsar sobre el botón: **“Modificar tarea”**

El bot generará una lista de botones con todas las tareas actualmente disponibles en la lista. Al seleccionar una, el bot pedirá que se envíe la modificación para esa tarea. Cuando el usuario envíe la nueva tarea, el bot mostrará la checklist con la modificación realizada. La nueva tarea aparecerá en el lugar que ocupaba la tarea modificada y estará en estado “sin hacer”. La modificación realizada se informará en la parte superior del mensaje.

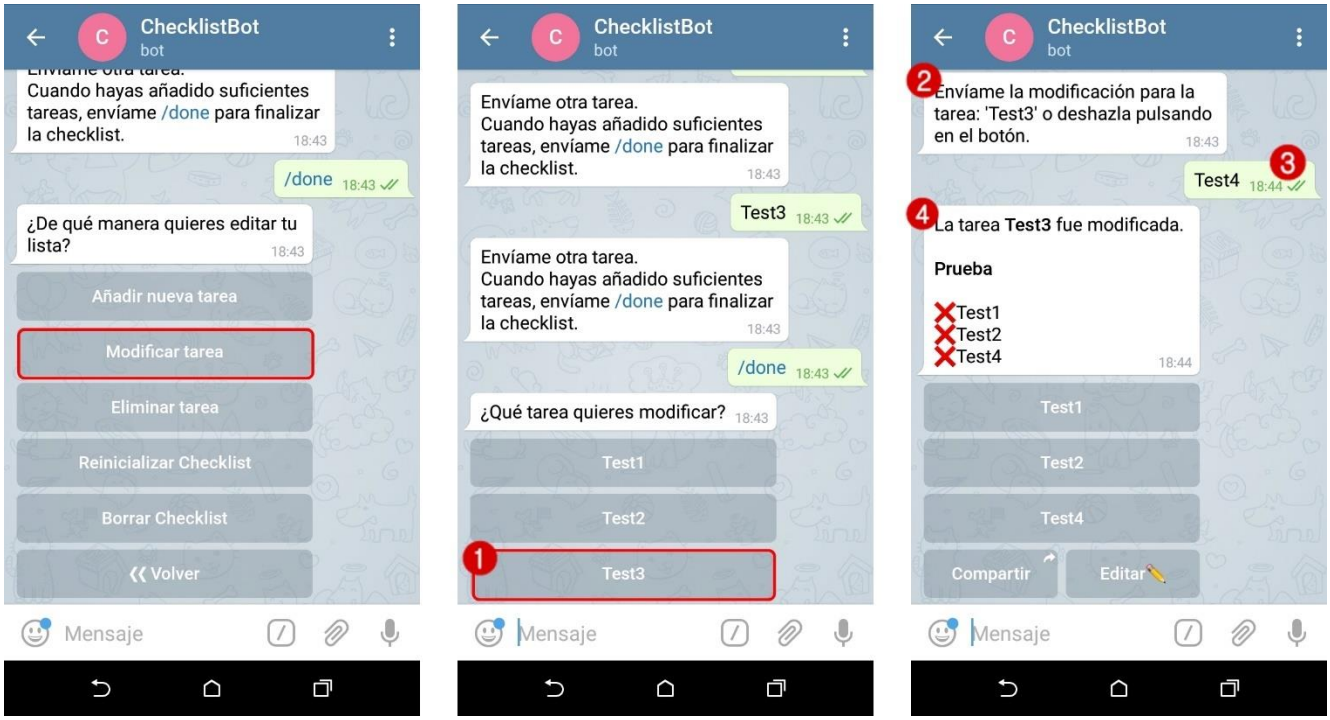

Figura 49. Pasos para modificar una tarea.

Cuando una tarea es modificada, conserva el estado de asignación que tenía antes de la modificación. Si quiere desasignarse, se deben seguir los pasos descritos en el apartado “Asignar Tarea”. Desde la opción de modificar tarea, además de modificar el nombre, también se puede deshacer únicamente la tarea conservando el nombre actual de esta. La diferencia radica en el paso dos. Se han omitido la primera y segunda imagen de la figura 49 para evitar repetir imágenes, porque son los mismos. Por esta razón, la figura 50 muestra un paso dos sin un paso uno anterior. Nuevamente, la modificación realizada se muestra en la parte superior del mensaje.

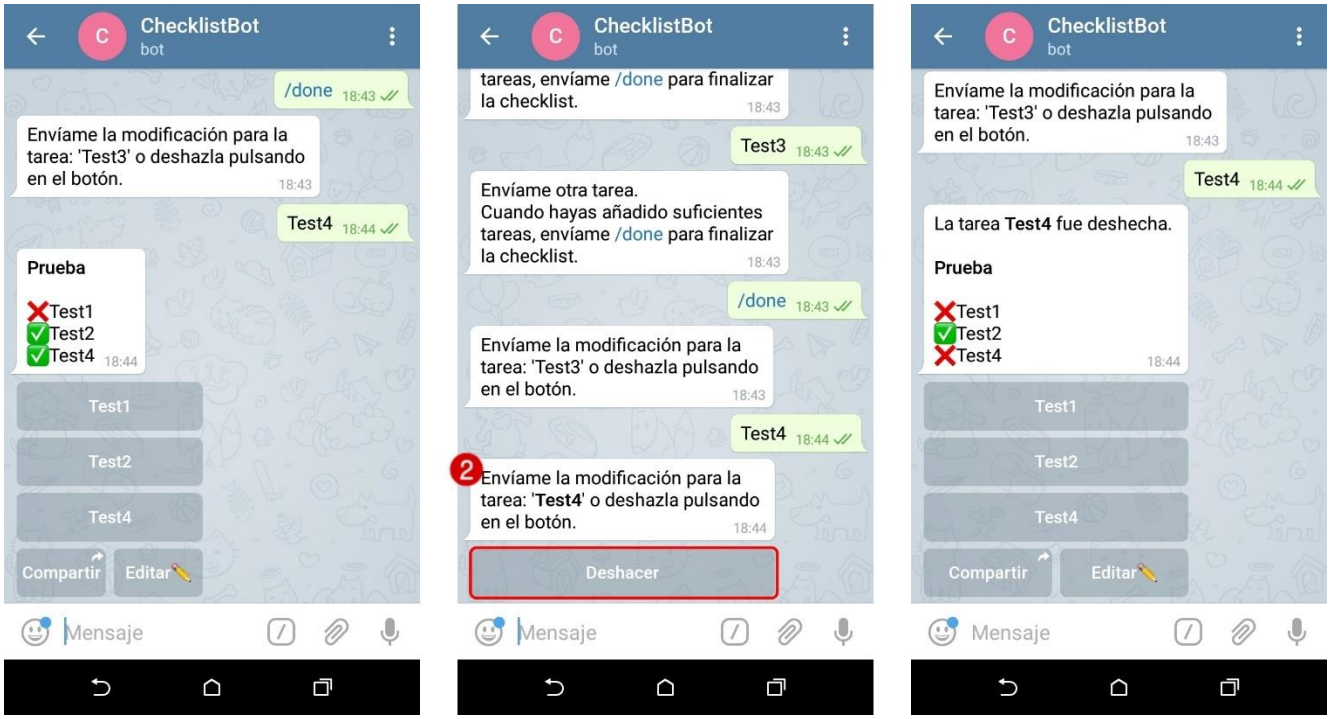

Figura 50. Pasos para deshacer una tarea.

Eliminar tarea

Si el usuario quiere eliminar una tarea de una de sus checklist, debe seleccionar dentro de las opciones de edición el botón: **"Eliminar tarea"**

El bot generará una lista de botones con todas las tareas actualmente disponibles en la lista. Al pulsar en una de ellas, el bot la eliminará de la checklist y la mostrará actualizada.

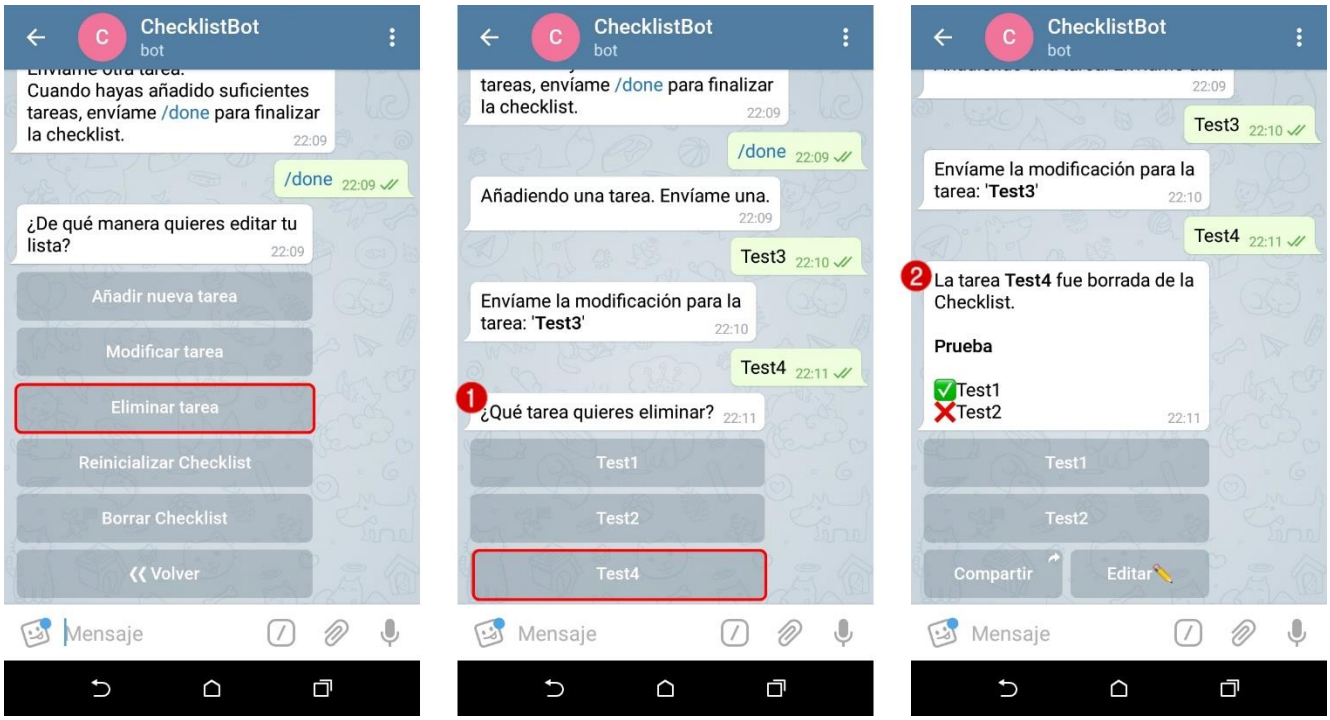

Figura 51. Pasos para eliminar una tarea.

Si la lista únicamente dispone de una tarea, el bot informará al usuario y solicitará una confirmación de que desea borrar la tarea. Cuando la última de las tareas de una checklist es eliminada, la checklist se borra con ella. Si el usuario confirma la acción, la lista será borrada del sistema permanentemente. En caso contrario, la lista no es modificada.

Reinicializar checklist

Cuando el usuario desee inicializar una checklist al estado inicial, es decir, marcar todas las tareas sin hacer, debe pulsar en el botón “Editar” y a continuación el botón: **“Reinicializar checklist”.** El bot modificará el estado de todas las tareas a “sin hacer” y mostrará la lista actualizada al usuario.

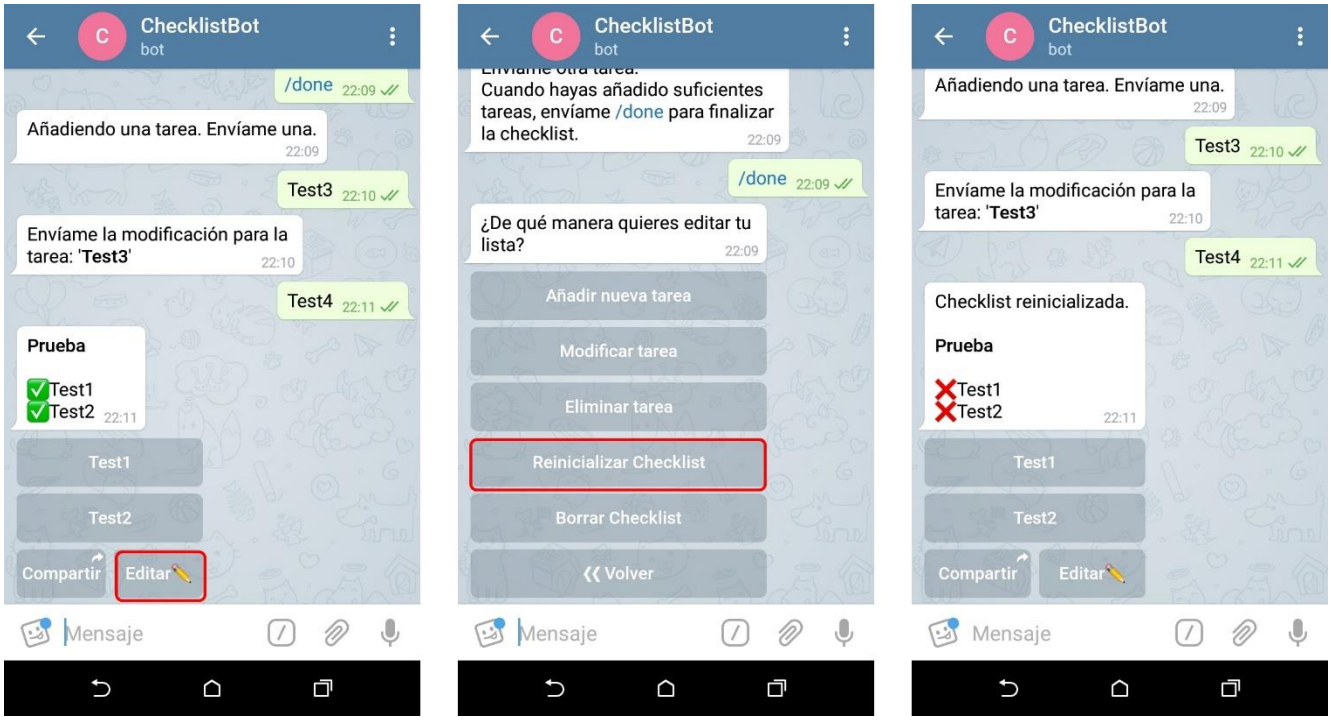

Figura 52. Pasos para reinicializar una checklist.

Eliminar checklist

Para eliminar una checklist de la aplicación, el usuario debe seleccionar de entre las opciones de edición el botón: **“Borrar checklist”**

Por seguridad, esta acción requiere de una confirmación por parte del usuario. Si el usuario confirma, la lista es borrada permanentemente del sistema y no existe forma de recuperarla.

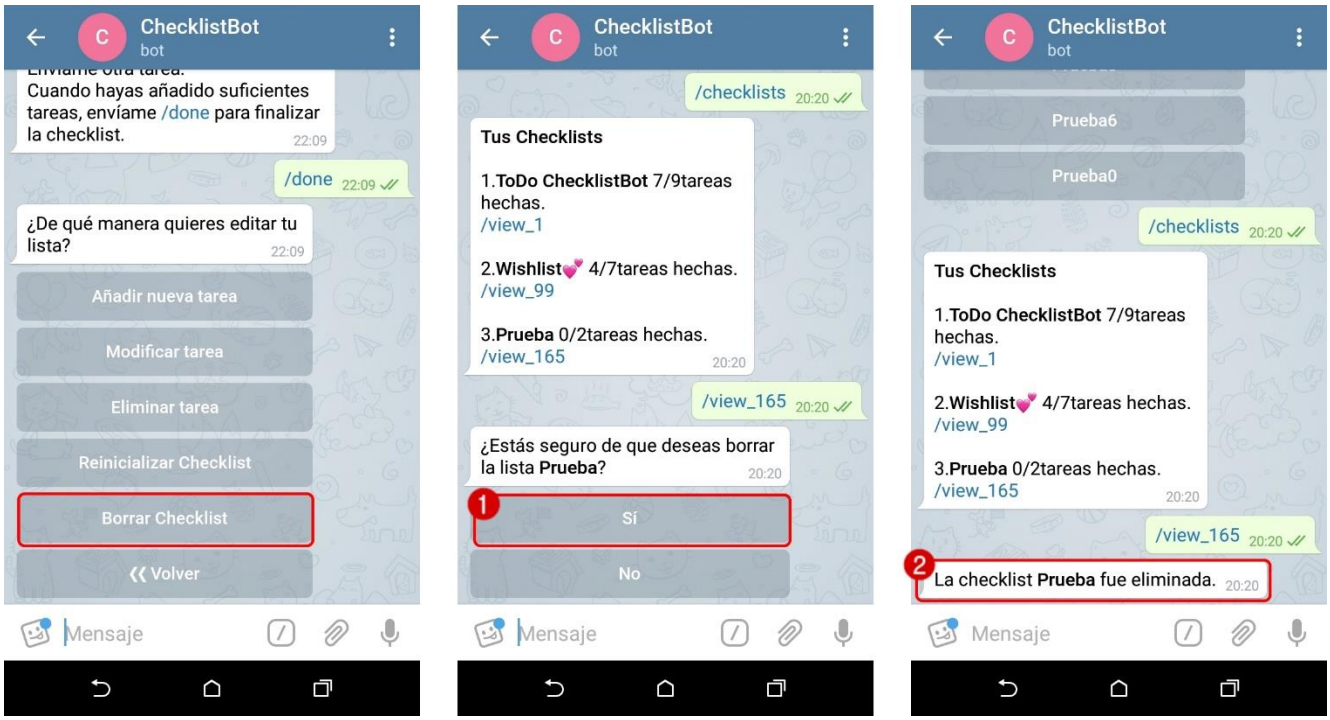

Figura 53. Pasos para eliminar una checklist.

Compartir checklist

Un usuario puede compartir sus checklists con otros usuarios. Para hacer esto debe pulsar sobre el botón 

Este botón aparece disponible en el mensaje con la checklist, únicamente dentro del chat privado entre el creador y el bot. La checklist siempre es compartida en el estado actual de la lista. El procedimiento a seguir para compartir una lista es el siguiente:

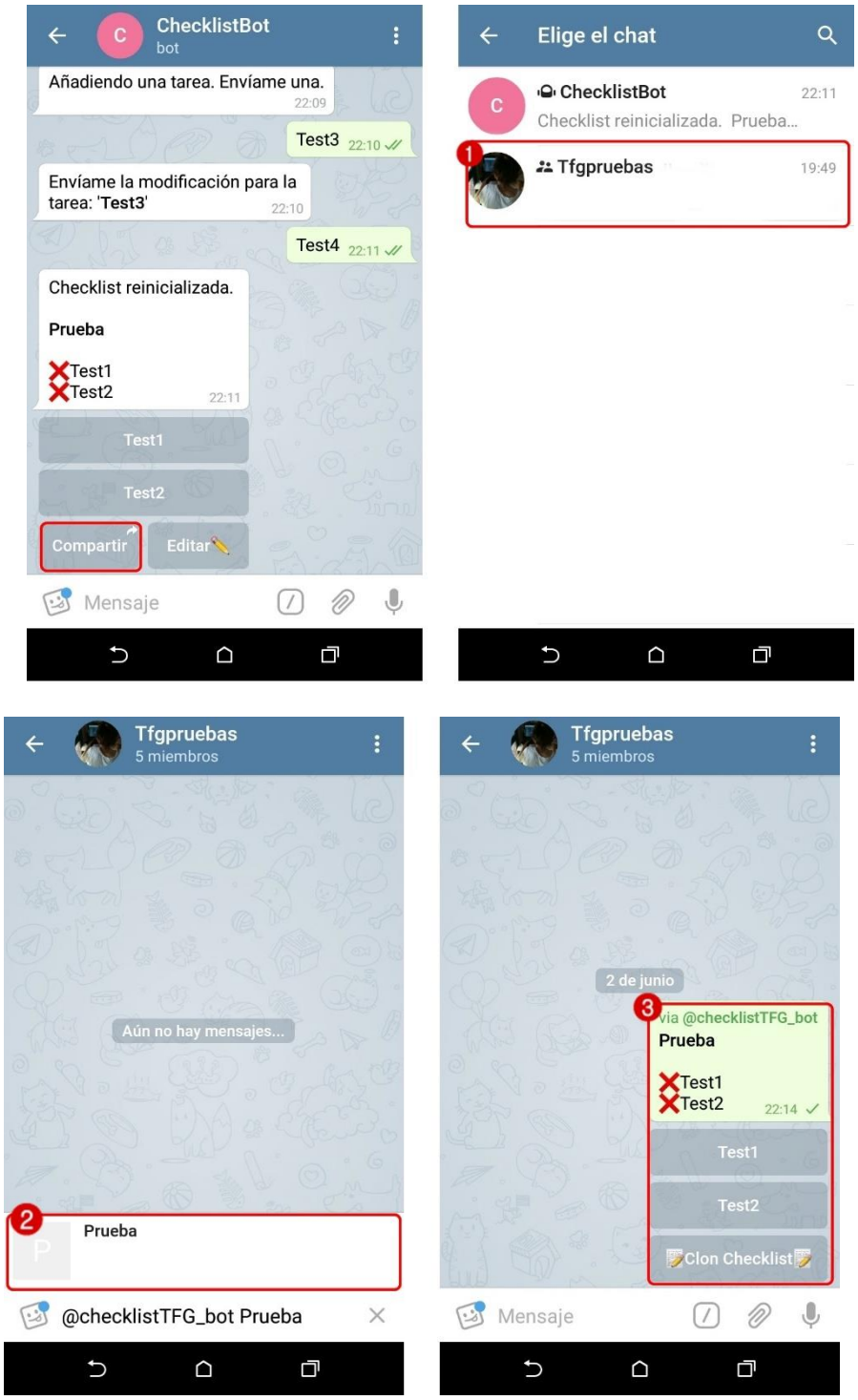

Figura 54. Pasos para compartir una checklist en otro chat.

Dependiendo de si el creador comparte la checklist en un chat privado entre él y otro usuario o un grupo donde el bot esté añadido podrá realizar más o menos funciones con la lista compartida.

Cuando una checklist es compartida dentro de un grupo privado un chat grupal al cual no pertenece *Checklistbot* los usuarios podrán marcar realizadas las tareas en conjunto y la lista original se actualizará simultáneamente. Todas las listas compartidas se modifican y sincronizan en tiempo real con la original.

Si se comparte una checklist dentro de un grupo del cual *Checklistbot* es miembro, el creador podrá asignar tareas de la lista a cierto miembro del grupo. Esta funcionalidad será explicada en próximas páginas.

Clonar checklist

Cualquier usuario con el cual haya sido compartida una checklist puede duplicarla y obtener una copia de la original de la cual será el propietario. De esta forma, obtendrá los privilegios de creador dentro de la lista clonada. Esta lista se comporta de manera independiente a la original, no son actualizadas simultáneamente. Para clonar una checklist, se deben seguir los pasos mostrados a continuación:

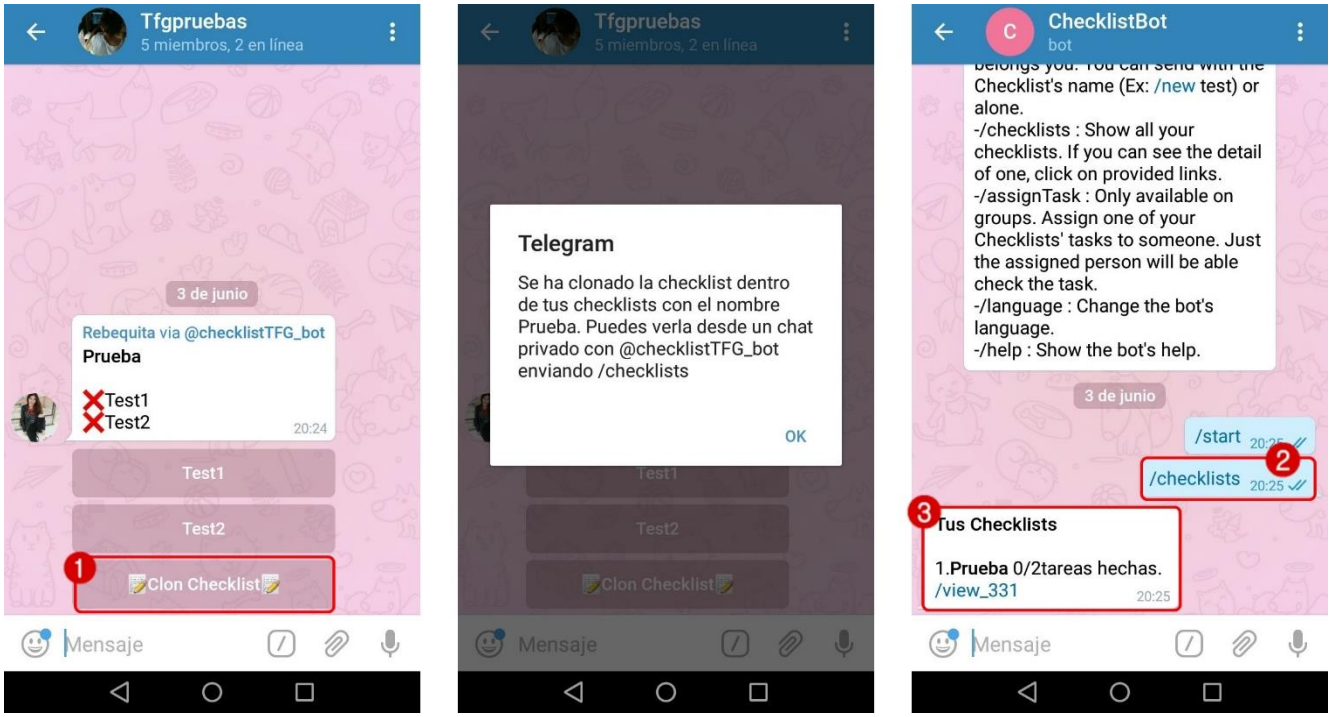

Figura 55. Pasos para clonar una checklist y visualizar el resultado.

Si la checklist que se quiere clonar tiene un nombre que ya existe entre nuestras checklists no será duplicada para evitar futuros problemas con la duplicidad de nombres. El bot nos informará de que la acción no se va a llevar a cabo.

Buscar checklist

Un usuario puede buscar entre sus listas una en concreto escribiendo el nombre del bot. Introduciendo @checklistfg_bot se le mostrarán cinco de sus listas. Puede filtrar los resultados hasta encontrar la lista que busca escribiendo parte del nombre de la lista o el nombre completo. Los resultados mostrados se actualizan al mismo tiempo que el usuario escribe.

Esto es una alternativa al botón de compartir de la propia checklist si el propietario escribe dentro del chat donde quiere compartir la lista.

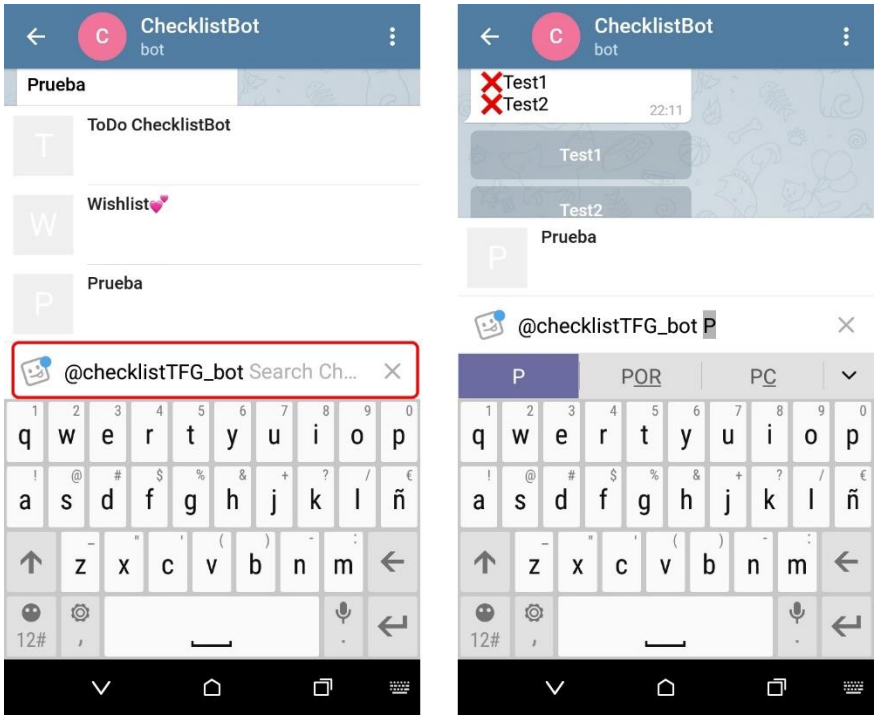

Figura 56. Búsqueda de checklists a través de la zona de texto.

Si un usuario desea recuperar alguna de sus checklists, la mejor opción es mediante el comando */checklists*. Cuando se busca a través de la entrada de texto, el mensaje generado no dispone de los botones especiales para la administración de la checklist por parte del propietario.

Ver checklists

Mandando el comando /checklists dentro de un chat privado con el bot, un usuario puede recuperar una lista con todas sus checklists. Este comando no está disponible dentro de cualquier otro chat que no sea el individual con el bot.

En el mensaje respuesta enviado por el bot se muestra un listado con el nombre de las checklists del usuario, el baremo de tareas realizadas y un link para recuperar la lista y verla en detalle. Pulsando un link se recupera la checklist como se puede ver en la figura X.

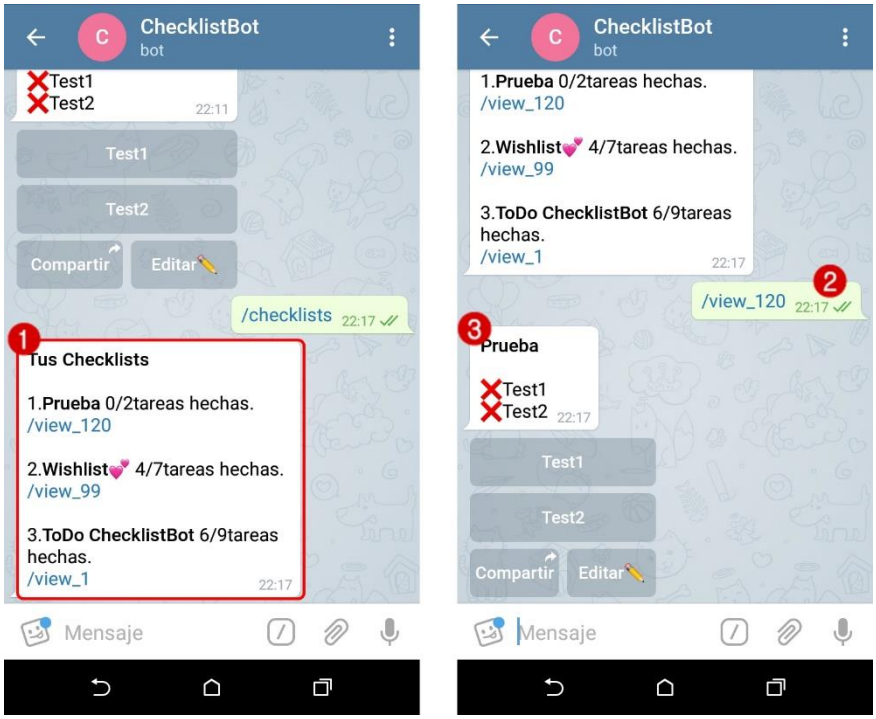

Figura 57. Búsqueda de checklists a través de la zona de texto.

Asignar tarea

Si un usuario envía el comando */assigntask* dentro de un chat grupal podrá asignar cualquiera de las tareas de una de sus checklists a un miembro del grupo. Al enviar el comando, el bot responde con una lista de botones con todas las checklists del usuario que envía el comando. Al seleccionar una de ellas, el bot proporciona todas las tareas pertenecientes a la checklist elegida. Por último, al seleccionar la tarea, el bot solicita el miembro al cual se va a asignar. El bot proporciona una lista de botones con los miembros pertenecientes al grupo que tiene almacenados en sus datos.

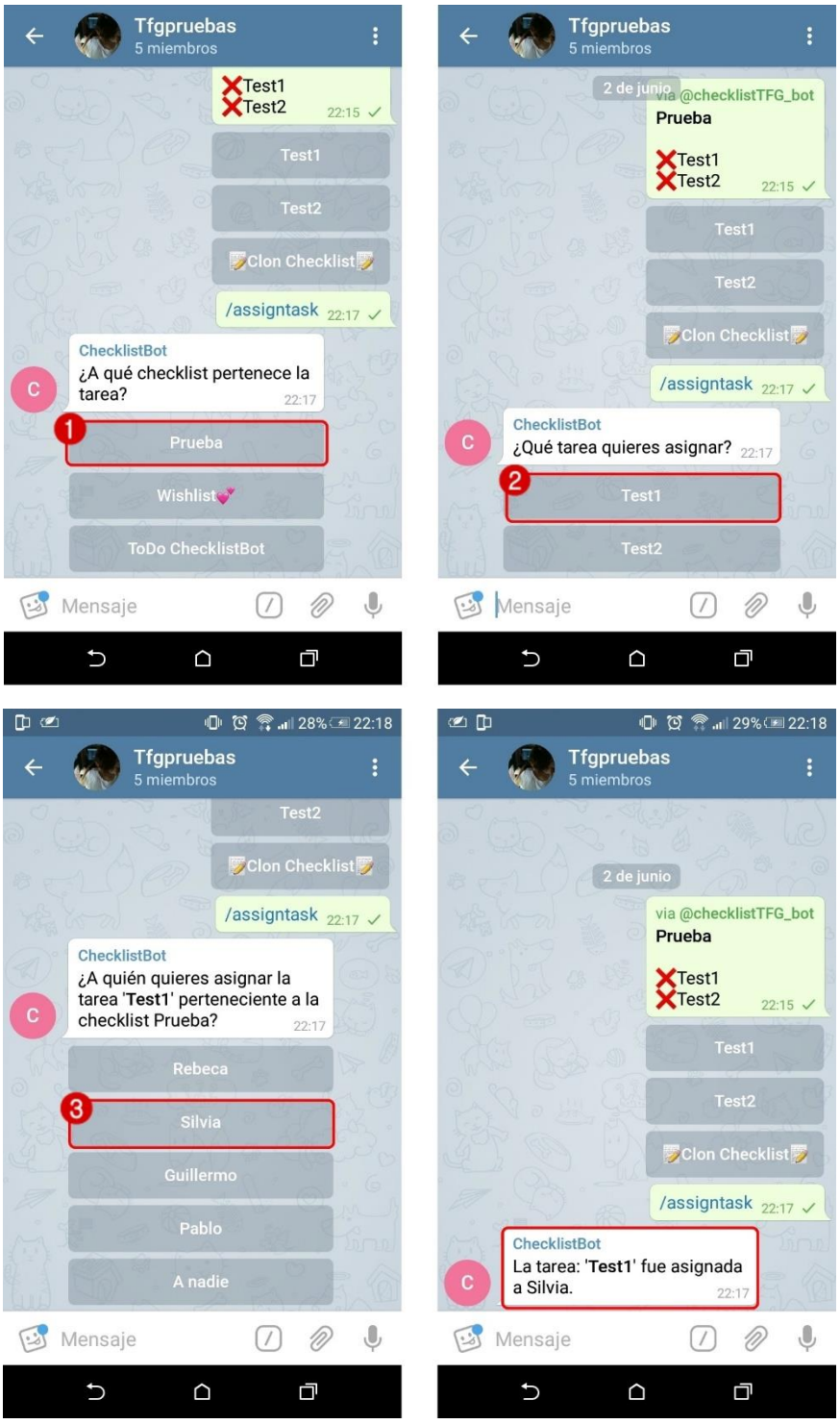

Figura 58. Pasos para asignar una tarea de una checklist a un usuario.

Cuando el bot no dispone de información sobre todos los miembros del grupo, aparece un mensaje adicional acompañado de un botón para añadir nuevos miembros manualmente. Si el usuario al cual se desea asignar la tarea no aparece entre los botones, debe pulsar sobre el botón citado. El bot solicitará el nuevo usuario, para proporcionarlo se debe escribir @ y seleccionarlo. De esta forma, Telegram genera una mención sobre ese usuario y al enviar el mensaje el bot recibe los datos del usuario mencionado. Esta información será almacenada para poder mostrarlo en la lista de botones en posteriores ocasiones.

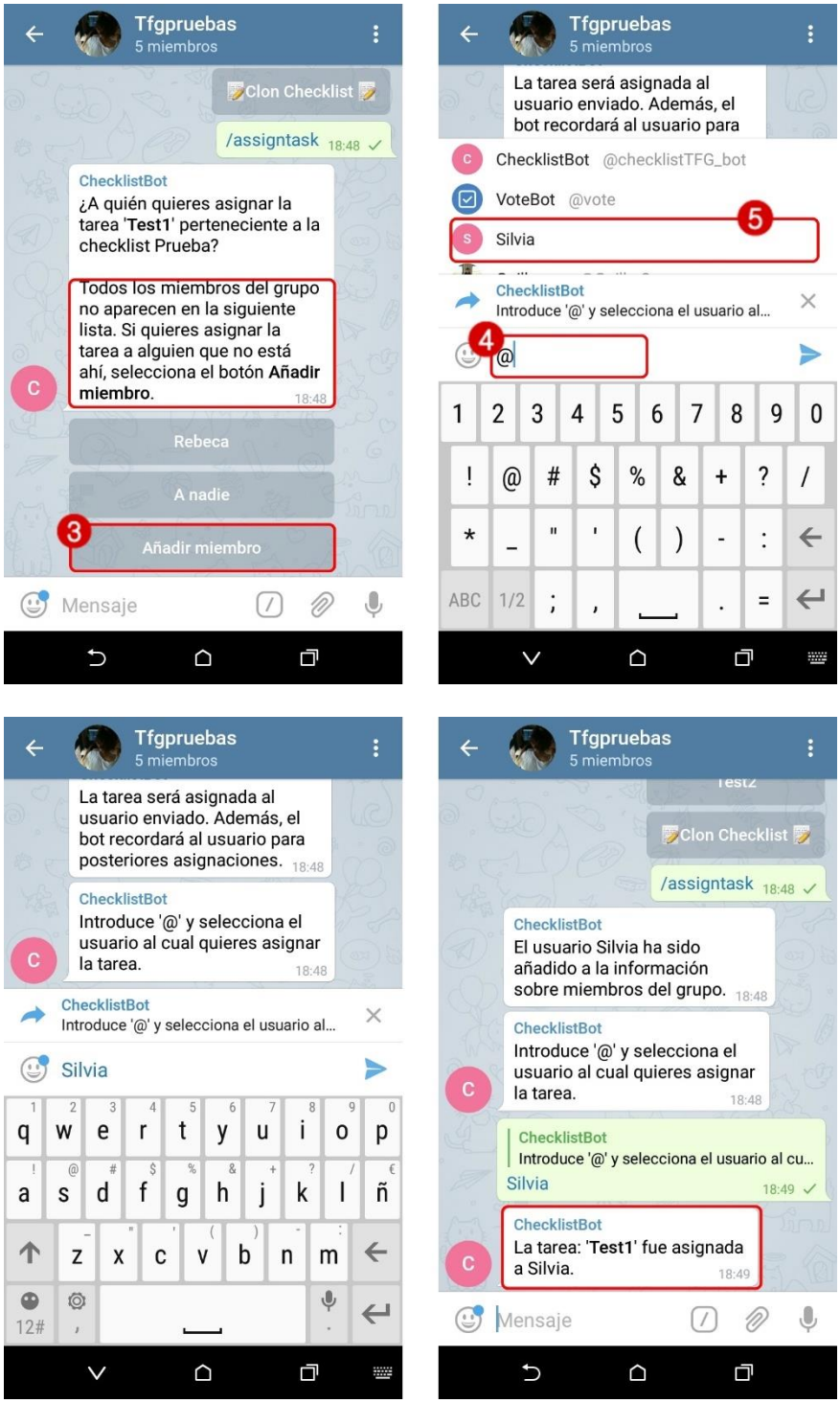

Figura 59. Pasos para asignar una tarea de una checklist a un usuario que no aparece en la lista de botones proporcionada por el bot.

No se puede asignar una tarea a un usuario con alias. Cuando un usuario tiene un alias asignado a su cuenta, sus datos son privados. Para que el usuario citado aparezca en los botones proporcionados por el bot, este debe escribir algo en el chat comenzado por “/”.

Cuando una tarea es asignada, sólo el usuario al cual ha sido asignada puede marcarla.

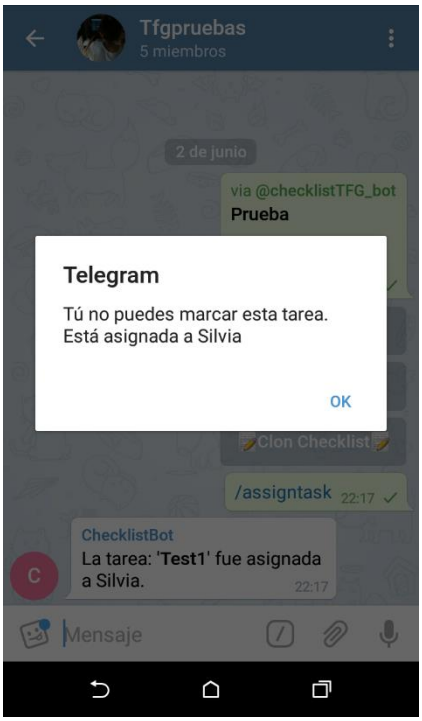

Figura 60. Popup informativo al intentar marcar una tarea asignada a otro usuario.

Anexo II. Script MySQL

```
drop table if exists User;
drop table if exists Chat;
drop table if exists Member;
drop table if exists State;
drop table if exists Checklist;
drop table if exists Task;
drop table if exists Message;

create table User (
    id int,
    name char(100),
    username char(32),
    constraint pk_user primary key (id),
        );

create table Chat (
    id int,
    language char(2),
    constraint pk_chat primary key(id)
        );

create table Member (
    idChat int,
    idUser int,
    constraint pk_member primary key (idChat, idUser)
    constraint fk_chat foreign key (idChat) references Chat(id),
    constraint fk_user foreign key (idUser) references User(id),
        );

create table State (
    id int,
    name char(6),
    constraint pk_state primary key (id),
        );

create table Checklist (
    id int,
    name char(30),
    owner int
    state int,
    constraint pk_checklsit primary key (id)
    constraint fk_state foreign key (state) references State(id),
    constraint fk_owner foreign key (owner) references User(id),
        );

create table Task (
    name char(50),
    idList int,
    owner int,
    checked char(5),
    constraint pk_task primary key (name, idList),
    constraint fk_checklist foreign key (idList) references Checklist(id),
    constraint fk_owner foreign key (owner) references User(id),
        );

create table Message (
    id int,
    idList int,
    constraint pk_state primary key (id),
    constraint fk_checklist foreign key (idList) references Checklist(id),
        );
```

Anexo III. Contenido del CD

El CD del presente trabajo final de carrera contiene la siguiente información:

- *memoria.pdf:* versión electrónica en PDF del presente documento.
- */sources:* código fuente del software desarrollado. Los archivos .py se encuentran localizados en sus correspondientes carpetas siguiendo el patrón MVC.
- */diagramas:* diagramas que han aparecido en el presente documento en formato imagen.

Índice figuras

Bibliografía

[1] Autor: Wikipedia. Año: 2017. https://es.wikipedia.org/wiki/Proceso_unificado#Inicio [Consultado: 27/06/2017]

Información general acerca del desarrollo de software mediante la metodología de Proceso Unificado.

[2] Autor: Eldinnie Año: 2017. https://github.com/python-telegram-bot/python-telegram-bot/wiki/Where-to-host-Telegram-Bots#vps [Consultado: 27/06/2017]

Lista con diferentes opciones donde "hospedar" el bot una vez acabado su desarrollo.

[3] Autor: Amazon. Año: 2017. https://aws.amazon.com/es/pricing/ [Consultado: 27/06/2017]

Precios ofrecidos por Amazon para alquilar un servidor en la nube en su dominio.

[4] Autor: DigitalOcean. Año: 2017. https://www.digitalocean.com/pricing/#droplet [Consultado: 27/06/2017]

Precios ofrecidos por *DigitalOcean* para alquilar un servidor virtual privado en su dominio.

[5] Autor: Linode. Año: 2017. https://www.linode.com/pricing [Consultado: 27/06/2017]

Precios ofrecidos por *Linode* para alquilar un servidor virtual privado en su dominio.

[6] Autor: Ramnode. Año: 2017. https://www.ramnode.com/vps.php [Consultado: 27/06/2017]

Precios ofrecidos por *Ramnode* para alquilar un servidor virtual privado en su dominio.

[7] Autor: Scaleway. Año:2017. https://www.scaleway.com/virtual-cloud-servers/ [Consultado: 27/06/2017]

Precios ofrecidos por *Scaleway* para alquilar un servidor virtual privado en su dominio.

[8] Autor: Wikipedia. Año: 2017 https://es.wikipedia.org/wiki/Raspberry_Pi [Consultado: 27/06/2017]

Información relativa a Raspberry Pi. Características hardware de cada versión e historia de la pequeña computadora.

[9] Autor: Aitor Ortuondo Año: 2013 https://blog.guebs.com/2013/10/02/servidor-cloud-versus-vps/ [Consultado: 27/06/2017]

Comparativa entre servidor virtual privado y servidor en la nube. Ventajas e inconvenientes.

[10] Autor: Ditendria. Año: 2016. http://www.ditrendia.es/wp-content/uploads/2016/07/Ditrendia-Informe-Mobile-en-Espa%C3%B1a-y-en-el-Mundo-2016-1.pdf [Consultado: 27/06/2017]

Análisis realizado en 2016 y centrado en España acerca del uso de los dispositivos móviles.

[11] Autor: Wikipedia. Año: 2017. https://es.wikipedia.org/wiki/Sal_(criptograf%C3%ADa) [Consultado: 27/06/2017]

Información general acerca del término *salt* dentro de criptografía.

[12] Autor: Telegram Año: 2017. https://core.telegram.org/mtproto [Consultado: 27/06/2017]

Información técnica y detallada relativa al protocolo MTProto. Desarrollado y utilizado únicamente en Telegram.

[13] Autor: Eva Hartog. Año: 2016. https://themoscowtimes.com/articles/how-telegram-became-the-durov-brothers-weapon-against-surveillance-52042 [Consultado: 27/06/2017]

Historia de la aplicación Telegram y de su creador Pavel Durov. Explica cómo y por qué surge Telegram.

[14] Autor: Wikipedia. Año: 2017. https://es.wikipedia.org/wiki/Telegram_Messenger [Consultado: 27/06/2017]

Información general y complementaria relativa a Telegram. Usada para contrastar y completar la obtenida mediante el artículo anterior [13].

[15] Autor: Telegram. Año: 2014. https://telegram.org/crypto_contest [Consultado: 27/06/2017]

Bases del concurso criptográfico lanzado por Telegram a principios del 2014. Enlaza los resultados del concurso: Ningún ganador.

[16] Autor: Telegram. Año: 2014. https://telegram.org/blog/cryptocontest [Consultado: 27/06/2017]

Bases del concurso criptográfico lanzado por Telegram a finales del 2014. Enlaza los resultados del concurso: Ningún ganador.

[17] Autor: Telegram. Año: 2016. https://telegram.org/blog/botprize [Consultado: 27/06/2017]

Bases del concurso de bots lanzado por Telegram a principios del 2016. Enlaza los resultados del concurso: se escogieron 4 bots ganadores.

[18] Autor: Wikipedia. Año: 2017. https://es.wikipedia.org/wiki/Anexo:Caracter%C3%ADsticas_de_Telegram_Messenger [Consultado: 27/06/2017]

Información general acerca de las características disponibles en la aplicación de mensajería Telegram.

[19] Autor: Wikipedia. Año: 2017 https://es.wikipedia.org/wiki/Diffie-Hellman [Consultado: 27/06/2017]

Explicación técnica relativa al funcionamiento del protocolo de seguridad *Diffie-Hellman*. Este protocolo es usado por Telegram para asegurar la privacidad de los chats secretos.

[20] Autor: Telegram. Año: 2017. https://telegram.org/blog/instant-view-contest-200K [Consultado: 27/06/2017]

Información detallada de la nueva característica añadida en mayo de 2017 a Telegram: Instant views (vistas instantáneas de webs). Abre un nuevo concurso, se pueden ver las bases desde el enlace.

[21] Autor: Telegram. Año: 2017. https://telegram.org/blog/video-messages-and-telescope [Consultado: 27/06/2017]

Información detallada de la nueva característica añadida en mayo de 2017 a Telegram: Telescope (video mensajes dentro de los chats).

[22] Autor: Telegram. Año: 2017. https://telegram.org/blog/payments [Consultado: 27/06/2017]

Información detallada de la nueva característica añadida en mayo de 2017 a Telegram: Pagos a través de los bots.

[23] Autor: Telegram. Año: 2017. https://core.telegram.org/bots [Consultado: 27/06/2017]

Información general acerca de los bots: cómo comenzar la creación de uno, sus distintos modos de funcionamiento, cómo son las peticiones a servidor, entre otros datos.

[24] Autor: Telegram. Año: 2017. https://core.telegram.org/bots/samples [Consultado: 27/06/2017]

Frameworks recomendados por Telegram para el desarrollo de los bots. Catalogados según lenguaje de programación.

[25] Autor: WhatsApp. Año: 2016. https://faq.whatsapp.com/es/general/28030015 [Consultado: 27/06/2017]

Información relativa al cifrado extremo a extremo implementado en los chats de WhatsApp. Disponible solo para versiones actuales (posteriores a la lanzada en abril de 2016)

[26] Autor: Ana Martínez. Año: 2016. http://www.abc.es/tecnologia/moviles/aplicaciones/abci-whatsapp-cifra-servicio-y-blinda-completo-201604051842_noticia.html [Consultado: 27/06/2017]

Artículo sobre el nuevo cifrado de WhatsApp end-to-end y la importancia de este.

[27] Autor: ApkHere. Año: 2017. http://es.apkhere.com/app/com.whatsapp [Consultado: 27/06/2017]

Historial de versiones de WhatsApp. Disponibles para descargar los distintos archivos *apk*.

[28] Autor: Santiago Luque. Año: 2015. http://www.androidpit.es/10-motivos-por-los-que-telegram-es-mejor-que-whatsapp [Consultado: 27/06/2017]

Comparativa de WhatsApp y Telegram. Razones por las cuales es más ventajoso el uso de Telegram que de WhatsApp.

[29] Autor: EL1OT. Año: 2017. http://tlgram.net/2017/05/telegram-consume-menos-datos-whatsapp/ [Consultado: 27/06/2017]

Comparativa sobre el consumo de datos en los servicios de mensajería instantánea más populares

[30] Autor: Samuel Fernández. Año: 2016. https://www.xatakamovil.com/mercado/tres-graficas-que-muestran-por-que-google-querria-comprar-a-telegram [Consultado: 27/06/2017]

Artículo acerca de los datos destacables de Telegram. Muestra el crecimiento de Telegram en sus primeros años y una comparativa de usuarios activos entre los distintos servicios de mensajería.

[31] Autor: Wikipedia. Año: 2017. https://es.wikipedia.org/wiki/Ingenier%C3%ADa_de_requisitos#Especificaci.C3.B3n_de_requisitos_del_software [Consultado: 27/06/2017]

Información general relativa a la especificación de requisitos de un proyecto software. Su división según su tipo y las características de cada uno.

[32] Autor: Omar Lino. Año: 2012. http://analisisuba2012s.blogspot.com.es/2012/01/diagrama-de-actividades.html [Consultado: 27/06/2017]

Información general relativa a la realización de diagramas de actividades. Explicación de para qué sirven y cómo entenderlos y realizarlos.

[33] Autor: Miguel Ángel Álvarez. Año: 2014. https://desarrolloweb.com/articulos/que-es-mvc.html [Consultado: 27/06/2017]

Descripción del patrón MVC con ejemplos explicativos y fáciles de entender.

[34] Autor: w3schools. Año: 2017. https://www.w3schools.com/xml/xml_dtd_intro.asp [Consultado: 27/06/2017]

Información general sobre lo que son las DTD y cómo se realizan. También se describe cómo se pasa de DTD a XML.

[35] Autor: Jonatan Fiestas. Año: 2014. http://blog.elevenpaths.com/2014/11/qa-pruebas-para-asegurar-la-calidad-del.html [Consultado: 27/06/2017]

Explicación de la importancia de la realización de pruebas en software y categorización en los diferentes tipos de pruebas.

[Api Python 3] https://docs.python.org/3.7/c-api/ [Consultado: 27/06/2017]

Documentación del lenguaje de programación Python (versión 3). Utilizada durante todo el desarrollo de la aplicación.

[Api Bot Telegram] https://core.telegram.org/bots/api [Consultado: 27/06/2017]

Documentación de Telegram para el desarrollo de un bot. Describe los métodos disponibles para llamar al servidor de Telegram y las clases con sus atributos. Necesaria para comprender los mensajes recibidos por Telegram y el usuario final. Utilizada durante todo el desarrollo de la aplicación.

[Api Telepot] http://telepot.readthedocs.io/en/latest/ [Consultado: 27/06/2017]

Documentación del *framework* Telepot. Necesario por ser el intermediario entre el código de la aplicación y el servidor de Telegram. Utilizada durante todo el desarrollo de la aplicación.