Grado en Ingeniería Informática

Trabajo Final de Grado

Desarrollo de una aplicación móvil para la gestión de partes de trabajo

Autor:
Héctor Valls Mondragón

Supervisor:
Marian Agost
Tutor académico:
José Ribelles

Fecha de lectura: 28 de Noviembre de 2014
Curso académico 2013/2014

Resumen

Este proyecto ha consistido en desarrollar una aplicación móvil para la plataforma Android que automatiza la entrega, recogida y gestión de partes de trabajo por parte de los operarios de la empresa FACSA. Se han llevado a cabo las fases de análisis, diseño e implementación propias de ingeniería de software. Mediante esta aplicación los operarios conocen las tareas a realizar en cada jornada. Además, permite registrar los recursos que se han utilizado en cada parte, como materiales o vehículos.

Palabras clave

Android, Aplicación móvil, Gestión, Partes de trabajo

Keywords

Android, Mobile app, Management, Work order

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Justificación y motivación del proyecto

Este proyecto ha sido desarrollado en la empresa *ADC Sistemas e Infraestructuras*, que pertenece a la división de servicios del grupo empresarial *Grupo Gimeno*.

Esta empresa está especializada en el desarrollo de sistemas de telecontrol del ciclo integral del agua. *ADC* crea sus propias aplicaciones informáticas para la gestión de abastecimientos y el control de estaciones depuradoras. Además, es responsable de la gestión del sistema informático común a todas las empresas pertenecientes a *Grupo Gimeno*.

Figura 1.1: Organigrama de *Grupo Gimeno*

Entre estas empresas se encuentra *FACSA*, que ofrece todos los servicios relativos al ciclo de aguas residuales, desde su captación, potabilización y tratamiento, hasta su

distribución, recogida y depuración.

Este proyecto tiene sentido ligado a las tareas de los operarios de *FACSA*. Estos operarios están divididos en brigadas, que son conjuntos de 2 a 4 trabajadores. Cada brigada dispone de un operario que es el capataz. El tipo de tareas que estos llevan a cabo habitualmente son, entre otras, la instalación de un contador en la vivienda de un nuevo cliente o la reparación de una fuga en un alcantarillado. Para conocer los cometidos de cada día, el capataz de cada brigada de operarios acude a las oficinas de la empresa para recoger los documentos físicos que corresponden a los partes de trabajo de dicha jornada. Posteriormente, el capataz distribuye cada parte de trabajo de forma equitativa entre los operarios de su brigada.

Este proyecto surge de la necesidad de automatizar el proceso de entrega y distribución de partes de trabajo entre los operarios por parte del personal administrativo. Por eso, se pretende desarrollar una aplicación móvil que permita la gestión y entrega de partes de trabajo entre las oficinas y los operarios de la empresa.

1.2. Objetivos del proyecto

El objetivo de este proyecto es el desarrollo de una aplicación móvil que permita a los empleados de *FACSA* gestionar los partes que describen las tareas que deben realizar en cada jornada de trabajo. Además, mediante esta aplicación los operarios serán capaces de registrar los recursos que se han utilizado para completar cada tarea. Estos recursos van desde materiales de construcción, hasta operarios que participan o vehículos que se emplean en dicha tarea.

Mediante este nuevo sistema se pretende:

- Automatizar el proceso de recogida de partes de trabajo por parte de los operarios.
- Facilitar la tarea de documentación del uso de recursos en los partes de trabajo.
- Automatizar el registro y alta de partes de trabajo en el sistema central.

Para desarrollar este sistema, se ha realizado un proceso íntegro de ingeniería de software que a continuación será detallado. En primer lugar, se va a explicar la fase de recopilación de requisitos y análisis; en segundo lugar, se describirá la arquitectura, el diseño a nivel de componentes e interfaz de usuario; finalmente, la fase de implementación y pruebas.

1.3. Descripción del proyecto

En este proyecto se desarrollará una aplicación móvil para tabletas con sistema operativo Android. Esta aplicación intercambiará datos con un servidor, cuya implementación está fuera del alcance de este proyecto. Por esto, se puede decir que la arquitectura que se va a utilizar es cliente/servidor.

Los usuarios que utilizarán esta aplicación son los operarios de la empresa *FACSA*. Cada operario dispondrá de una tableta con esta aplicación instalada, y la llevará consigo durante toda la jornada de trabajo.

Las funcionalidades que permitirá realizar la aplicación a los operarios son las siguientes:

- Sistema de autenticación de usuarios dados de alta previamente para empezar a utilizar la aplicación.
- Crear nuevos partes de trabajo.
- Asignar recursos a estos partes. Estos recursos serán materiales utilizados, vehículos empleados y operarios que han intervenido en el trabajo que describe el parte.
- Recibir partes de trabajo del servidor central de la empresa. De esta manera, los operarios sabrán qué tareas han de llevar a cabo en la ciudad sin necesidad de pasar por la empresa a recoger los partes en formato físico.
- Marcar como finalizados los partes cuya tarea ya ha sido completada.
- Enviar los partes finalizados al servidor.
- Si el operario es un capataz de brigada, podrá distribuir los partes entre los operarios de su brigada. Los usuarios de la aplicación podrán ver en todo momento qué partes tienen asignados.

En el momento de intercambiar información de partes de trabajo con el servidor será necesaria una conexión a Internet. Para utilizar las demás funciones, no será estrictamente necesario. De esta manera, los operarios podrán rellenar información sobre partes de trabajo en zonas con baja o nula cobertura.

Para desarrollar este sistema será necesario diseñar e implementar una base de datos que persistirá la información de vehículos, materiales y partes de trabajo, entre otros, en el dispositivo móvil.

Además, se diseñará una interfaz gráfica atractiva y fiel al aspecto corporativo de la empresa. Esta interfaz será muy intuitiva ya que el objetivo es que el operario invierta el menor tiempo posible usando la aplicación. Este será un beneficio respecto a documentar y registrar recursos de un parte en papel.

Capítulo 2

Planificación del proyecto

2.1. Metodología de trabajo

Este proyecto ha sido llevado cabo mediante el uso de una metodología ágil de desarrollo de proyectos software. Se ha tomado esta decisión por los motivos siguientes:

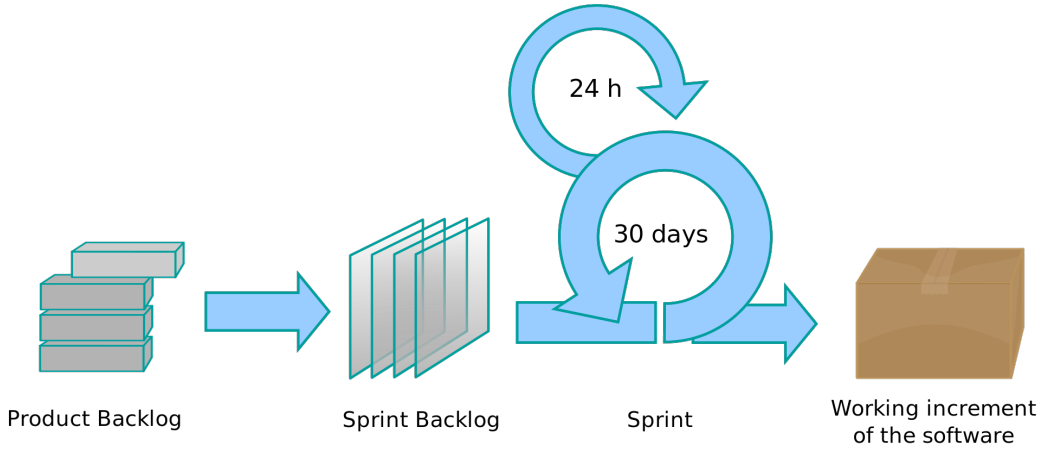
- Efectividad y flexibilidad ante cambios de requisitos.
- Metodología basada en iteraciones de una a cuatro semanas.
- Al final de cada iteración, se dispone de un producto funcional.
- Evitar problemas como retrasos de tiempo y complejidad.

Entre las distintas metodologías ágiles que existen, en este proyecto se ha utilizado la que probablemente sea la más conocida: *SCRUM*. Se ha elegido esta metodología ya que es la habitual entre el equipo de desarrollo de la empresa donde se ha realizado la estancia.

La Figura 2.1 muestra el ciclo de vida de esta metodología, donde:

- El **Product Backlog** es la pila de requisitos que debe implementar el sistema. Se compone de funcionalidades y tareas que debe realizar el programador que están en constante evolución durante todo el ciclo de vida, hasta el cierre del sistema.
- El **Sprint Backlog** es la pila de tareas que se van a desarrollar en el *sprint* actual. Para extraer estas tareas del *product backlog* se pueden llevar a cabo técnicas de priorización de tareas, según las necesidades del usuario y la complejidad de estas. Para llevar el seguimiento del estado de estas tareas, se ha utilizado la técnica Kanban con la herramienta Jira, que se describe posteriormente.

Figura 2.1: Ciclo de vida de *Scrum*



- El **Sprint** es la iteración propiamente dicha, que dura de una a cuatro semanas. En esta iteración, los programadores desarrollan las tareas definidas en el *sprint backlog* y, al finalizar uno de estos, el resultado es un producto funcional que el cliente es capaz de ejecutar y tiene un valor añadido respecto a la iteración anterior.

Además, al inicio y fin de cada iteración se realizan reuniones de planificación de la entrega y construcción del *backlog*, e inspección del incremento integrado en cada *sprint*. La metodología también define que se han de realizar reuniones diarias de 15 minutos aproximadamente para identificar problemas y obstáculos para resolverlos lo antes posible.

El equipo de desarrollo de *SCRUM* esta formado por los siguientes roles:

- **Product Owner**: Decide la funcionalidad del producto y es el responsable de priorizar las tareas a desarrollar en cada iteración. Representa el usuario del sistema.
- **Scrum Manager**: Motiva y coordina al equipo y es responsable de detectar los problemas que pueden surgir durante el proceso.
- **Team Scrum**: Crean el producto en sí y son un equipo multidisciplinar de programadores, *testers*, analistas, etcétera.

Todos los miembros del equipo de desarrollo han de conocer con detalle la visión del *product owner* y han de colaborar regularmente y de manera directa con este.

Aunque este proyecto se ha basado en una metodología ágil, en cada iteración se ha realizado un proceso de ingeniería de software clásico y, por claridad, este proyecto se ha

documentado de esta forma; el apartado de desarrollo de la aplicación se divide en cuatro fases bien diferenciadas:

- *Análisis de requisitos* En esta fase, se estudian las necesidades del cliente y qué espera que el sistema ofrezca. También, se identifican los distintos actores que estarán involucrados así como qué rol van a tomar en el sistema. Como resultado, se obtiene un diagrama de casos de uso, una recopilación de requisitos de usuario formalizada y una serie de prototipos sencillos de la interfaz de usuario.

- *Diseño e implementación* En esta fase tiene lugar el diseño de la arquitectura, que ofrece una visión externa del sistema, con todos los componentes que lo forman, a alto nivel. Además, se realiza el diseño a nivel de componentes, diseño de clases, identificación de patrones de diseño, etcétera. Finalmente, consultando los prototipos obtenidos en la fase anterior, se obtiene el diseño real de la interfaz de usuario y se lleva a cabo la construcción propiamente dicha del sistema.

- *Validación y verificación* Para lanzar el sistema a producción es necesario probarlo en distintos escenarios para asegurar su correcto funcionamiento. Se han llevado a cabo pruebas unitarias, en las que se prueba la lógica interna de un componente independientemente de otros, y pruebas de integración, donde se enlazan los distintos componentes para asegurar que estos cooperan correctamente.

- *Despliegue o implantación* En esta fase, el producto es lanzado en un entorno real de producción. En algunos casos, esta fase puede dividirse en dos: lanzar el producto para un grupo reducido de usuarios y estos reportar errores, en caso de encontrarlos, y finalmente, una vez el sistema funciona correctamente durante un tiempo establecido, realizar el lanzamiento masivo para todos los usuarios.

  En este proyecto la aplicación se ha lanzado en un entorno de usuarios muy reducido pero funcionando en un entorno real. Aun así, sólo se han puesto a disposición del usuario un subconjunto de funcionalidades mientras el resto de ellas se continuaban desarrollando. De esta manera, se ha obtenido rápidamente una retroalimentación en cuanto a fallos de implementación o cambios deseados por el usuario.

Como se ha dicho anteriormente, en este proyecto se han puesto en práctica metodologías ágiles, por tanto, el desarrollo de estas cuatro fases no ha sido completamente lineal. Para cada iteración se ha llevado a cabo un pequeño proceso de ciclo de vida clásico formado por estas fases. De esta forma, al finalizar cada *sprint*, el desarrollador posee una versión parcial funcional del sistema validada por el usuario, evitando riesgos de complejidad o temporales, habituales en el desarrollo clásico de software.

2.2. Planificación temporal

En este apartado se muestran las tareas a realizar para llevar a cabo este proyecto. Para realizar esta planificación se ha tenido en cuenta que el tiempo total del proyecto debía ser de unos 60 días, ya que la estancia en la empresa del alumno era de 300 horas a cinco horas por jornada.

Estas estimaciones se han realizado con ayuda del tutor de prácticas y el supervisor de la empresa, aplicando su juicio por experiencia en el desarrollo de proyectos similares. A continuación se muestran dichas tareas:

- Análisis de requisitos (9 días)
  - Definir casos de uso
  - Definir requisitos funcionales
  - Diseñar prototipos
  - Validar requisitos
- Diseño e implementación (34 días)
  - Arquitectura
    - Documentar arquitectura
    - Validar procesos
  - Diseño a nivel de componentes
    - Diseñar clases del modelo
    - Detectar patrones
    - Refactorizar
  - Implementación
- Pruebas (16 días)
  - Detectar pruebas de unidad
  - Realizar y validar pruebas
  - Realizar pruebas de integración
  - Realizar pruebas del sistema
- Puesta en marcha (4 días)
  - Preparación de la instalación
  - Redactar manual de usuario
  - Validar proceso de instalación
  - Implantación

En la Figura 2.2 se pueden observar estas tareas en un diagrama de Gantt, donde se muestra la duración y la disposición en el tiempo de cada una.

Las tareas que se exponen en esta planificación son las típicas de un desarrollo con un ciclo de vida clásico. En cambio, el proyecto se ha desarrollado mediante una metdología ágil. En cada iteración de SCRUM, se han llevado a cabo un análisis, diseño e implementación de los requisitos de usuario que se pretendía desarrollar en ese periodo. Es decir, las historias de usuario escogidas de la pila del producto para esa iteración.

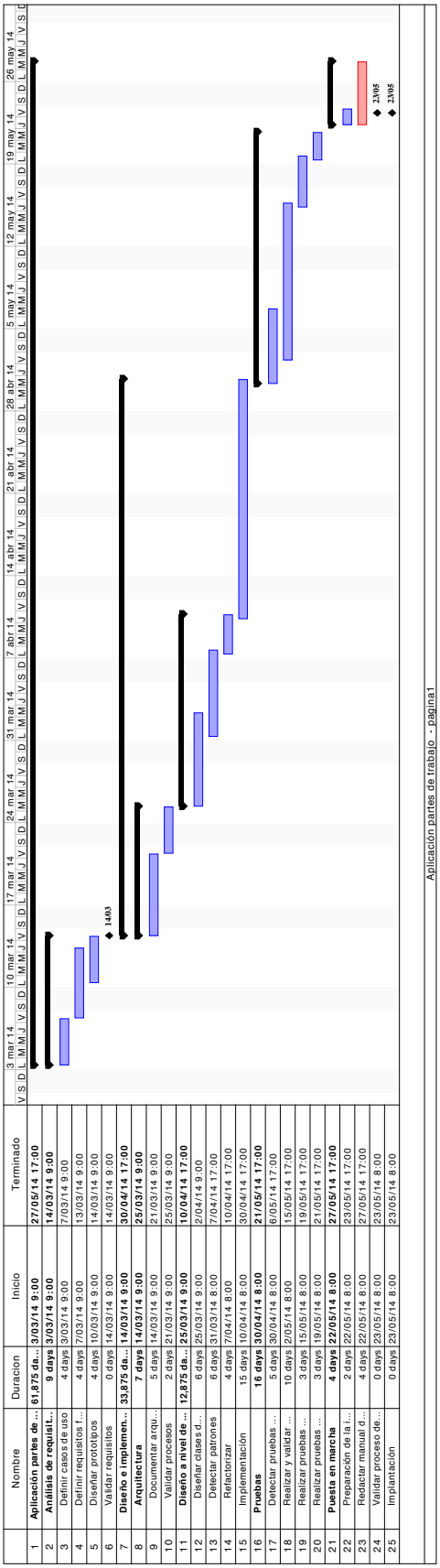
Se han necesitado un total de cinco *sprints* para desarrollar el producto. Además, la pila del producto no ha estado definida completamente desde el principio, sino que se han ido añadiendo historias de usuario a medida que se avanzaba.

2.3. Desviaciones del objetivo inicial

Finalmente, todas las tareas y objetivos definidos al inicio del proyecto se han alcanzado en el tiempo global esperado y los resultados han sido satisfactorios. Sin embargo, respecto a cada tarea individual, ha habido desviaciones temporales, principalmente por la continua refactorización del código, debido a la complejidad de la lógica de negocio, y cambios en los requisitos de interfaz gráfica del usuario. Además, se ha necesitado un periodo de tres días, no planificado, para la formación sobre el desarrollo para la plataforma Android.

En la planificación inicial de las actividades, se describió la tarea de la redacción del manual de usuario y procedimientos. Finalmente, esta tarea se ha desestimado por considerarse innecesaria debido a la claridad de la interfaz de usuario y la familiarización con los conceptos de los futuros usuarios. En lugar de esto, se reunirá a los operarios que vayan a utilizar la aplicación y se les explicará el funcionamiento de esta. Por este motivo, aun habiendo necesitado un periodo no planificado de formación previa, al finalizar el desarrollo se ha dispuesto del tiempo necesario para realizar pruebas en varios escenarios y asegurar el correcto funcionamiento de la aplicación.

Figura 2.2: Diagrama de Gantt



Capítulo 3

Entorno tecnológico

Este proyecto se ha desarrollado sobre la plataforma Android en el lenguaje Java. Además de esta plataforma, se han utilizado un conjunto de herramientas y tecnologías, cada una de ellas para un propósito concreto. Se han utilizado estas herramientas ya que son las habituales en los proyectos Android del equipo de desarrollo de la empresa. Por eso, no se han analizado herramientas alternativas.

Las herramientas y tecnologías utilizadas son las siguientes:

- *Android Annotations*: Este *framework* para proyectos Android pone a disposición del programador un conjunto de anotaciones Java. Estas anotaciones permiten desde la inyección de dependencias entre componentes, hasta la definición de un *listener* para un botón de la interfaz gráfica de forma más sencilla que el modo habitual. En cuanto a la inyección de dependencias, podría decirse que es el *Spring Framework* para proyectos Android.

- Apache Maven: Esta herramienta de línea de comandos permite la creación de proyectos Java con una determinada estructura de paquetes y la gestión de dependencias automática sin necesidad de incluir los ficheros *jar* manualmente en el *classpath*, entre otras muchas funcionalidades.

- *Representational State Transfer* (REST): Es una arquitectura software que define la comunicación distribuida entre un cliente y un servidor. Esta comunicación se realiza mediante el protocolo HTTP utilizando los métodos GET, POST, PUT y DELETE intercambiando elementos de información llamados recursos.

  Una tecnología alternativa a REST podría ser RPC (*Remote Procedure Call*), donde se invoca a un método que se ejecuta en una maquina remota. La ventaja de REST respecto a RPC es el mínimo acoplamiento, ya que en RPC el cliente necesita saber la signatura del método que va a invocar mientras que en REST cada recurso es identificado por una URI única y, mediante los métodos estándar de

HTTP mencionados anteriormente, puede realizar cualquier acción que se desee en el servidor.

- *SQLite* y *ORMLite*: Ya que en la aplicación se necesita persistencia local, el sistema de gestión de bases de datos que se ha usado ha sido SQLite y el *framework* ORMLite. Utilizando ORMLite no hay necesidad de crear las tablas manualmente, sino que se definen mediante anotaciones en las clases Java que las van a representar.
- Subversion: Se ha utilizado esta herramienta para gestionar el control de versiones.
- Jira: Esta herramienta permite el seguimiento y planificación de tareas fácilmente mediante un tablero Kanban virtual. Una alternativa gratuita podría ser Trello, aunque la empresa dispone de una licencia completa de herramientas de desarrollo de software de la compañía Atlassian, donde se incluye Jira.

En las siguientes secciones se explica con profundidad cada una de estas tecnologías y herramientas.

3.1. Android

Android es un sistema operativo basado en el *kernel* de Linux y enfocado a dispositivos móviles, tales como *smartphones* y tabletas. Inicialmente fue desarrollado por la empresa Android Inc. y perteneció a esta hasta el año 2005, que fue absorbida por Google (Ableson, 2010).

En cada lanzamiento de una nueva versión del sistema se agregan mejoras y nuevas funciones, pero como base, el sistema fue implementado para soportar las siguientes características:

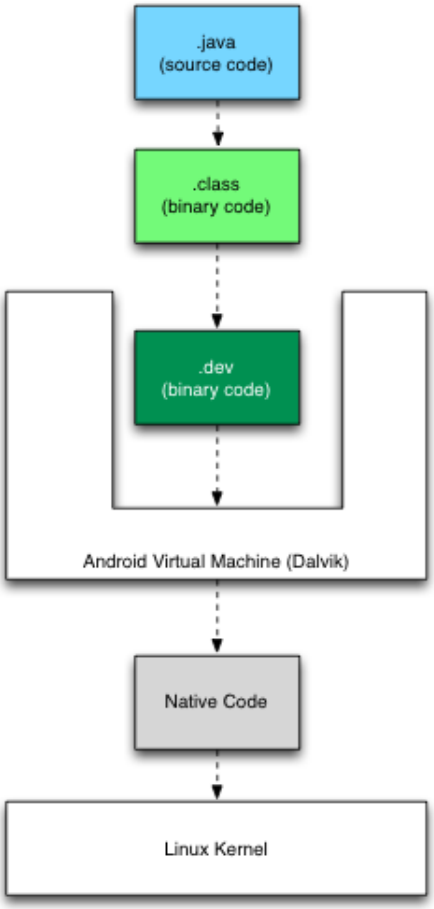
- Estándares de conectividad *GSM/EDGE, UMTS, LTE, Bluetooth y Wi-fi*, entre otros.
- Soporte para los formatos multimedia: MPEG-4 SP, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF y BMP.
- Servicios de mensajería tradicionales SMS y MMS, así como C2DM (*Android Cloud to Device Messaging*), como parte del servicio que ofrece Android de *Push Messaging*.
- Videollamada y búsqueda a través de voz.

A parte de estas, Android ofrece otras muchas funciones y características en función de la versión.

Android Virtual Machine

La mayoría de aplicaciones que corren sobre el sistema Android están escritas en lenguaje Java aunque, internamente, este no dispone de una maquina virtual JVM (*Java Virtual Machine*). Para ejecutar estas aplicaciones, el código se compila a *bytecode* y posteriormente, en lugar de ejecutarse sobre una JVM, se recompila a código capaz de ser ejecutado por la máquina virtual de Android: *Dalvik*.

Figura 3.1: *Android Virtual Machine*



La recompilación de *bytecode* a código que Dalvik es capaz de interpretar, es llevada a cabo por la herramienta *Dex*, incluida en el kit de desarrollo de Android (*SDK*).

Todo este proceso, que va desde la compilación de código Java a la ejecución del código nativo por parte del *kernel*, es transparente al programador.

3.1.1. *Android Annotations*

Android Annotations es un *framework* gratuito y de código abierto que permite al programador desarrollar más rápidamente aplicaciones para el sistema Android, así como facilitar el mantenimiento de estas. Consiste en un amplio conjunto de anotaciones Java de las que el programador puede hacer uso con los siguientes propósitos:

- Inyección de dependencias, tanto de instancias de clases propias como de servicios o recursos del sistema.
- Consumo de una API *REST*, haciendo transparente la conexión de red al programador. En el siguiente apartado se explica con profundidad el consumo de una API *REST* haciendo uso de este *framework*.
- Gestión de *threads* (hilos). En Android, el acceso a la red, por ejemplo, no está permitido en el hilo principal. Por otro lado, la manipulación de elementos de la interfaz de usuario sólo se puede llevar a cabo en el hilo principal. *Android Annotations* permite esta gestión de hilos mediante las anotaciones *@Background* y *@UIThread*.
- Manejo de eventos de usuario. Anotaciones como *@Click*, *@ItemSelect* o *@CheckedChange* permiten al programador capturar eventos que produce el usuario como un *click* sobre un botón o la selección de un elemento de una lista.

Todas estas funciones hacen posible que el programador no se preocupe tanto sobre los detalles de implementación o sintaxis, y se centre en lo que realmente es importante: la lógica de negocio. Simplifica el código significativamente, proporcionando así más mantenibilidad y haciendo más fácil el desarrollo en tiempo de codificación.

AndroidAnnotationProcessor

Internamente, *Android Annotations* contiene un procesador de anotaciones, que actúa en el momento previo a la compilación. Antes de que el compilador *javac* lleve a cabo la compilación a *bytecode* de los ficheros, el procesador *AndroidAnnotationProcessor* procesa todas aquellas clases que han sido anotadas. Por cada una de estas clases, se genera otro fichero *.java* que tiene el mismo nombre, pero con terminación '_'.

Este nuevo fichero generado, contiene el código fuente que habría tenido que escribir el programador si no hubiese utilizado este *framework*. De esta forma, cuando el programador quiera hacer uso de alguna clase anotada, tendrá que emplear la clase generada con terminación '_', en lugar de la original.

3.2. Apache Maven

Maven es una herramienta para la gestión y la construcción de proyectos software escritos en lenguaje Java.

Esta herramienta facilita algunos procesos en el desarrollo como la resolución de dependencias, el nombrado de versiones o el generado de informes. Además, permite la gestión de perfiles como podrían ser *producción* o *desarrollo*. De esta forma, se podría disponer de una configuración distinta para cada perfil.

Toda la configuración del proyecto referente a Maven, ya sean perfiles, dependencias u otros se encuentra en un fichero XML (*pom.xml*) que se encuentra alojado en el directorio raíz del proyecto.

En este proyecto, se han creado dos perfiles: *devel* y *production*.

```
1    <profiles>
2        <profile>
3            <id>devel</id>
4            <properties>
5                <restServerURL>localhost:9003</restServerURL>
6                <developmentMode>True</developmentMode>
7                <defaultUser>1234</defaultUser>
8                <defaultPassword>1234</defaultPassword>
9            </properties>
10       </profile>
11       <profile>
12           <id>production</id>
13           <properties>
14               <restServerURL>xxx.xxx.xxx:xxxx</
                     restServerURL>
15               <developmentMode>False</developmentMode>
16               <defaultUser>user</defaultUser>
17               <defaultPassword>password</defaultPassword>
18           </properties>
19       </profile>
20   </profiles>
```

Como se puede observar, la diferencia entre estos dos perfiles es la dirección del servidor al que apuntan. Por motivos de seguridad y privacidad, la dirección del servidor de producción se ha ocultado. Desde el programa, se podría acceder a estas propiedades en tiempo de ejecución y dependiendo del perfil en que se esté ejecutando serían unas u otras.

Una vez definidos los perfiles necesarios, para compilar el proyecto en un perfil de-

terminado hay que ejecutar el comando *mvn package -P idperfil*, donde *idperfil* es el identificador que se ha asignado a dicho perfil en el *pom.xml*.

3.3. *Representational State Transfer (REST)*

REST es una técnica de comunicación distribuida que intercambia información mediante el protocolo de red *HTTP* (*Hypertext Transfer Protocol*). Posee una arquitectura de cliente-servidor, ya que cada petición *HTTP* es independiente y, ni el cliente ni el servidor, almacenan ningún estado.

Los sistemas intercambian recursos, que los elementos de información, mediante un conjunto de operaciones bien definidas: *GET*, *POST*, *PUT* y *DELETE*. Para comprender estas operaciones, se pueden comparar con las operaciones RPC (*Remote Procedure Call*) o de cualquier interfaz típica de acceso a datos: *get, add, edit, delete*.

En este proyecto, los recursos que se intercambian se codifican en formato *JSON* (*JavaScript Object Notation*). Este formato sólo admite tipos básicos, es decir, texto y números. Es menos extensible respecto al formato *XML* pero superior en interoperabilidad.

A continuación, se muestra un ejemplo de una petición *HTTP* al API *REST*, existente en la aplicación que se va a desarrollar:

- Se necesita obtener todos los usuarios existentes para almacenarlos en el dispositivo, y que estos puedan autenticarse al arrancar la aplicación.
- Existe un servicio *REST* que acepta la operación *GET* en la URL *http://servidor.es/servicios/usuarios* y devuelve los usuarios en formato *JSON*.
- La aplicación envía una petición *GET* a dicha dirección web.
- La aplicación recibiría una respuesta similar a la siguiente:

```
[
    {"idusuario" : "12",
     "idempleado" : "14",
     "login" : "sgarcia",
     "pass" : "5df9f63916ebf8528697b629022993e8"},

    {"idusuario" : "17",
     "idempleado" : "87",
     "login" : "mmoles",
     "pass" : "897a6e87c6a8ea7a897689a76c897687"},

```

```
    {"idusuario" : "85",
     "idempleado" : "35",
     "login" : "dbou",
     "pass" : "e987ca9087ce90a8c7ae098ac87a6e09"},
]
```

Como se observa, se ha obtenido un *array* que contiene cuatro elementos que representan los usuarios con los atributos 'idusuario', 'idempleado', *'login'* y 'pass', codificada en *MD5*. Como este servicio, existirán otros muchos para poder descargar, así como enviar, datos al servidor.

Internamente, *Android Annotations* hace uso de las clases Java que ofrece el paquete *java.net* para realizar las peticiones *HTTP*. Sin embargo, esto es transparente al programador, que sólo ha de codificar como si se tratase de una llamada a un método estándar de una clase. A continuación, se muestra su uso:

```
@Rest(rootUrl = "http://servidor.es/servicios")
public interface ClienteAPIRest {

    @Get("/vehiculos")
    public Vehiculo[] getVehiculos();

    @Get("/materiales")
    public Material[] getMateriales();

    @Get("/partes/{codEmpleado}")
    public ParteTrabajo[] getVehiculos(String codEmpleado);

}
```

Al compilar, el procesador de anotaciones de *Android Annotations* generará una clase que implementa esta interfaz.

Por ejemplo, recuperar todos los vehículos del servidor se llevaría a cabo de la siguiente manera:

```
public class Updater {

    @RestClient
    ClienteAPIRest_ restClient;

    public void update() {

        Vehiculo[] vehiculos = restClient.getVehiculos();

        ...
    }
}
```

Mediante la anotación *@RestClient* se le indica al procesador de anotaciones, que debe inyectar una instancia, siempre *singleton*, de la clase *ClienteAPIRest_* y que se va a usar como interfaz contra un servicio *REST*.

3.4. *SQLite*

La aplicación precisa de un sistema de persistencia local. Para ello, se ha elegido el sistema de gestión de bases de datos relacional *SQLite* por diversos motivos:

1. *SQLite* se integra sin problemas con Android y el consumo de memoria en tiempo de ejecución es mínimo. Es necesario optimizar los recursos de la aplicación, ya que puede que los dispositivos utilizados por los operarios no tengan un hardware de última tecnología.
2. La base de datos no es un proceso independiente con el que el programa principal se comunique, sino que todas las estructuras que la definen (tablas, vistas, etc...) están alojadas en un fichero.
3. Para realizar una copia de seguridad, basta con hacer una copia del fichero de la base de datos.
4. La librería de *SQLite* se integra con el mismo programa, es decir, no existe comunicación entre procesos (programa principal y base de datos), lo que reduce la latencia entre las operaciones sobre los datos almacenados.

Por otra parte, existen desventajas respecto a otros SGBD. Por ejemplo, no existe la gestión de usuarios, por lo que la seguridad delega en el sistema de permisos de ficheros del sistema operativo (Jay, 2010). Además, tampoco permite concurrencia de conexiones ya que el fichero de la base de datos se bloquea mientras un usuario está realizando alguna modificación.

A pesar de estas desventajas, se ha considerado que el SGBD *SQLite* es más que adecuado en el contexto de esta aplicación.

3.4.1. ORMLite

Esta herramienta permite mapear tablas de un modelo relacional de bases de datos a uno objeto-relacional mediante el uso de anotaciones. Utilizando este modelo, la manipulación de los datos se simplifica notablemente, ya que permite leer datos almacenados y tratarlos como instancias de clases del modelo. Esta conversión se realiza automáticamente en base al mapeo.

A continuación, se muestra un ejemplo de una clase mapeada mediante ORMLite.

```
1
2  @DatabaseTable(tableName = "VEHICULOS")
3  public class Vehiculo {
4
5      @DatabaseField(columnName = "MATRICULA", id = true)
6      private String matricula;
7
8      @DatabaseField(columnName = "MARCA")
9      private String marca;
10
11     @DatabaseField(columnName = "MODELO")
12     private String modelo;
13
14     public Vehiculo() {
15         super();
16     }
17 }
```

Este mapeo indica que existirá una tabla llamada 'VEHICULOS' (*@DatabaseTable*), que estará formada por las columnas 'MATRICULA', 'MARCA' y 'MODELO' (*@DatabaseField*). Además, la matrícula será la clave primaria de la tabla.

Una vez mapeado todo el modelo de la base de datos, se persistirán y recuperarán datos mediante una API sencilla que ofrece ORMLite con este propósito, sin necesidad de escribir sentencias *SQL*, que pueden ser propensas a errores.

3.5. Subversion

Los sistemas de control de versiones tienen varios objetivos. El primero es la posibilidad de compartir el código fuente y todos los recursos que componen un proyecto entre los desarrolladores que lo están llevando a cabo, y tener versiones actualizadas en todo momento. El segundo objetivo es el alojamiento de dicho proyecto en una máquina remota, evitando así pérdidas de información si se producen problemas en las máquinas de los programadores. Finalmente, los sistemas de control de versiones permiten generar "ramas" de desarrollo donde, por ejemplo, se podría tener una de producción y otra de desarrollo. Cada vez que una versión de desarrollo se va a lanzar a producción, se fusionaría la rama de producción actual con la de desarrollo, añadiendo a la primera las nuevas funcionalidad.

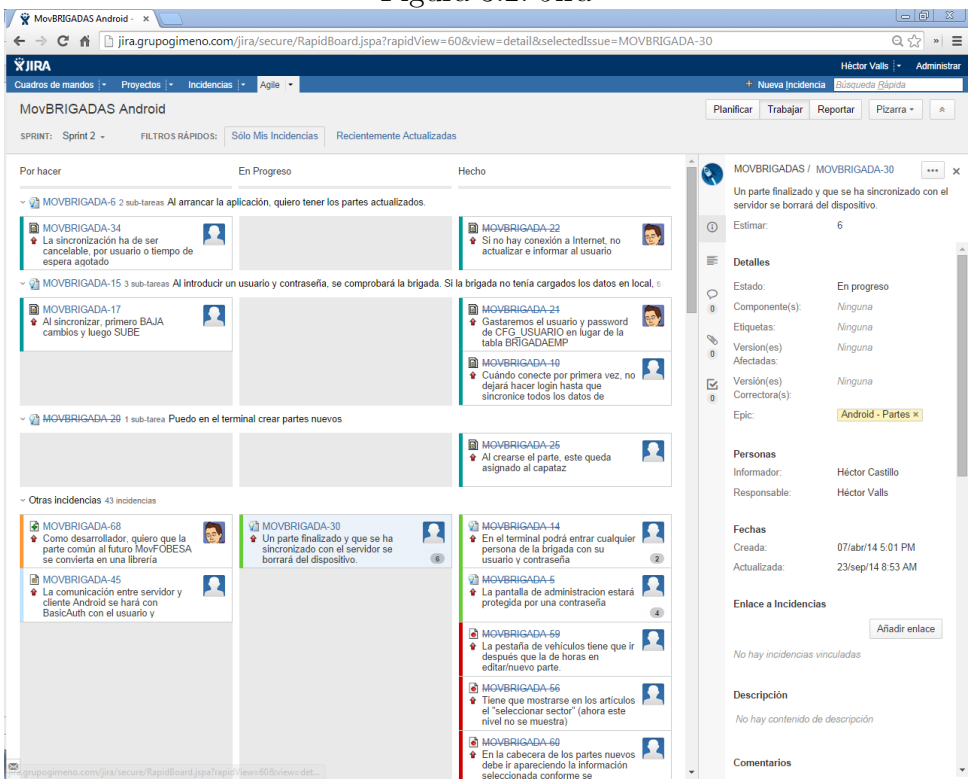
Para desarrollar este proyecto, se ha utilizado el control de versiones Subversion. Cuando se termina de desarrollar una funcionalidad o algún módulo con cierta complejidad, los cambios se suben al repositorio remoto para que el próximo programador que solicite una actualización de su versión sea capaz de ver estos cambios.

3.6. Jira

Jira es una herramienta web de la compañía Atlassian que permite al usuario gestionar las tareas de un proyecto de cualquier ámbito. Está fuertemente enfocado a proyectos que se llevan a cabo utilizando metodologías ágiles, ya que la página principal de la herramienta es un tablero Kanban donde el usuario puede generar tareas (en Jira, incidencias), y clasificarlas según su estado: pendiente, en proceso o resuelta. Además, permite al usuario crear sprints, definir su fecha de inicio y de fin y asignar tareas a cada uno de ellos.

En este proyecto, se han creado sprints a medida que se iba avanzando. Por cada requisito funcional descrito anteriormente, se ha generado una serie de tareas, para finalmente obtener la funcionalidad completa descrita en el requisito. Además, por cada cambio que se pedía en cada revisión, se ha generado una tarea inmediatamente, para así no olvidar ningún detalle. Por esto, las tareas que ha contenido el tablero Kanban no han sido estrictamente historias de usuario, como indica la metodología SCRUM, sino que en algunos casos eran simples detalles de implementación, o cambios mínimos en la interfaz de usuario.

Figura 3.2: Jira



En la Figura 3.2 se puede observar una captura de pantalla de la herramienta Jira, con una serie de tareas en el tablero.

Capítulo 4

Desarrollo de la aplicación

En los siguientes apartados se va a especificar la recopilación de requisitos de usuario, seguido del análisis del entorno y actores involucrados, diseño a nivel de arquitectura, componentes e interfaz gráfica y, finalmente, la implementación propiamente dicha del software.

4.1. Análisis

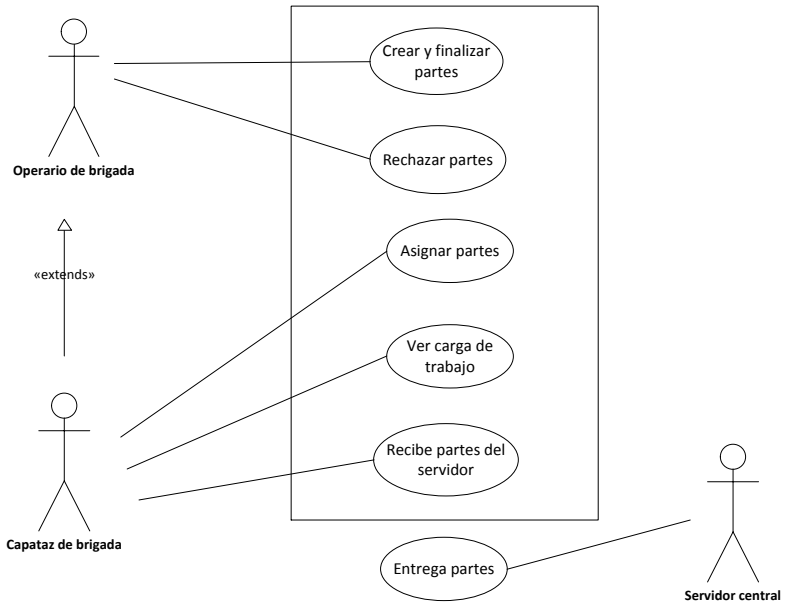
En esta fase, se define formalmente todos los elementos que constituyen el contexto del problema. Para ello, se va a presentar los requisitos de usuario y, como consecuencia de estos, los requisitos de software y hardware. Posteriormente, se mostrará un diagrama de casos de uso para apreciar qué actores realizan qué determinadas acciones sobre el sistema.

4.1.1. Casos de uso

Los casos de uso son una herramienta para el análisis de proyectos software. Haciendo uso de ellos se puede observar gráficamente la interacción entre los actores involucrados y el propio sistema (Pressman, 2005). En este proyecto, se pueden identificar tres actores: un operario de brigada, un capataz de brigada y el servidor central de la empresa. Este último, no interactúa directamente con la aplicación, pero es necesario tenerlo en cuenta para comprender el funcionamiento del software que se está analizando.

La Figura 4.1 muestra el diagrama de casos de uso, con los actores y las acciones que realizan sobre el sistema.

Figura 4.1: Casos de uso



Como se puede apreciar, se muestran los tres actores mencionados anteriormente, relacionados con las acciones que realiza cada uno sobre el sistema. Mediante el uso de la generalización (*extends*) se indica que el actor "Capataz de brigada" es una especialización de "Operario de brigada", es decir, que realiza las acciones de operario y además las que se relacionan con el actor de capataz.

Por otra parte, se puede observar un rectángulo que envuelve a una serie de acciones. Este rectángulo representa el alcance del sistema, es decir, aquellas acciones que se realizan dentro del contexto de la aplicación. El sistema central realiza la acción de entregar partes a los capataces de brigada pero, como no está dentro del rectángulo que representa el alcance, se entiende que es ajena a este.

4.1.2. Requisitos funcionales

El objetivo de los requisitos funcionales de usuario es definir el alcance del sistema, es decir, las características, a nivel funcional, que debe satisfacer el software. La definición formal de estos requisitos es necesaria para comprender el problema a resolver.

A continuación, se muestran las tablas que definen estos requisitos. Cada requisito contiene un identificador, nombre, descripción y una secuencia de pasos que definen su ciclo de vida.

Requisito funcional	
Código	FR01
Nombre	Crear parte
Descripción	El sistema ha de permitir a los operarios crear nuevos partes de trabajo. Cuando se crea un parte en la aplicación, automáticamente este queda asignado al operario que lo ha creado.
Secuencia normal	
1	Seleccionar la opción de crear nuevo parte.
2	Introducir información del parte.
3	Seleccionar opción de finalizar o guardar parte.
Importancia	Media
Comentarios	El sistema también ha de permitir editar y eliminar los partes que un operario crea mediante la aplicación.

Requisito funcional	
Código	FR02
Nombre	Finalizar parte
Descripción	Al terminar el trabajo descrito en el parte, el operario asignado a este ha de poder marcarlo como finalizado. Una vez finalizado, dicho parte aparecerá en la lista de finalizados, aun así, el empleado ha de poder editar este parte mientras no haya enviado los datos al servidor.
Secuencia normal	
1	En la pantalla de edición de parte, seleccionar la opción de finalizar.
Importancia	Alta
Comentarios	Al finalizar un parte, el sistema ha de llevar a cabo un proceso de validación de datos. En caso de que haya errores en la información introducida por el operario, ha de avisar de dicho error e impedir la finalización del parte, mientras no se haya corregido.

Requisito funcional	
Código	FR03
Nombre	Guardar parte
Descripción	El sistema ha de permitir al operario guardar un parte antes de ser finalizado. Posteriormente, el operario podrá editar la información de este y finalizarlo cuando sea necesario.
Secuencia normal	
1	En la pantalla de edición de parte, seleccionar la opción de guardar.
Importancia	Alta
Comentarios	Al contrario que en la opción finalizar parte, cuando se quiera guardar un parte no se realizará una validación de datos. Es decir, el operario podrá guardar un parte no finalizado aunque este contenga errores.

Requisito funcional	
Código	FR04
Nombre	Editar parte
Descripción	El sistema ha de permitir al operario editar un parte guardado así como finalizado. Hasta que no se sincronicen los datos y el parte este accesible para el operario, se ha de poder modificar su información.
Secuencia normal	
1	Seleccionar el parte en cuestión para abrir el modo edición.
Importancia	Media
Comentarios	Una vez editado el parte se podrá guardar o finalizar.

Requisito funcional	
Código	FR05
Nombre	Asignar partes
Descripción	Cuando se sincronizan los datos desde el servidor el capataz de la brigada recibe los partes asignados a esta y es el responsable de asignarlos a cada operario. El sistema ha de permitir realizar este proceso.
Secuencia normal	
1	Seleccionar la opción de ver los partes pendientes de asignar.
2	Seleccionar un parte.
3	Seleccionar la opción de asignar parte.
4	En la lista de empleados, seleccionar al que se desee asignar el parte.
Importancia	Alta
Comentarios	Un capataz de brigada también es un operario y puede realizar las tareas que describen los partes. Por tanto, también debe haber una opción que permita asignar el parte a el mismo capataz sin necesidad de seleccionar la opción de asignar y buscarse a sí mismo en la lista de operarios. Además, mientras no se sincronicen los datos, el capataz ha de ser capaz de modificar la asignación de partes entre los operarios.

Requisito funcional	
Código	FR06
Nombre	Rechazar parte
Descripción	El sistema ha de permitir a un operario rechazar partes que le han sido asignados. En este caso, el capataz de la brigada del operario ha de reasignar dicho parte.
Secuencia normal	
1	Seleccionar la opción de ver los partes asignados.
2	Seleccionar un parte.
3	Seleccionar la opción de rechazar parte.
Importancia	Media
Comentarios	Si el parte es rechazado por un operario, el capataz debe reasignar este parte. En caso de ser un capataz el que rechace el parte, este volverá al servidor central que realizará las acciones necesarias.

Requisito funcional	
Código	FR07
Nombre	Ver carga de trabajo
Descripción	Cuando un capataz de brigada quiera asignar un parte a un operario, la aplicación debe mostrarle la carga de trabajo de cada operario. Esta carga de trabajo consta del número de partes asignados a dicho empleado, y las horas aproximadas para finalizar todas sus tareas.
Importancia	Media
Comentarios	Un parte siempre tiene asignada una incidencia y cada una de estas tiene asignado un número de horas aproximadas de trabajo. La suma de las horas de finalización de cada incidencia de todos los partes asignados a un operario son las horas que se han de mostrar en la carga de trabajo.

Requisito funcional	
Código	FR08
Nombre	Asignar incidencia a parte y responder preguntas
Descripción	Cuando se vaya a finalizar un parte de trabajo el sistema ha de mostrar una serie de preguntas asociadas a la incidencia del parte (que pueden ser obligatorias o no), que el operario ha de responder.
Importancia	Media
Comentarios	Si el operario crea un parte nuevo, ha de ser capaz de elegir la incidencia que corresponda a la tarea que se va a llevar a cabo. Cada incidencia tiene asociadas una serie de preguntas que pueden ser de diferentes tipos. En el apartado de requisitos de datos se explican en más profundidad.

Requisito funcional	
Código	FR09
Nombre	Registrar horas de empleado
Descripción	Cuando se realiza la tarea descrita en un parte de trabajo, el sistema ha de permitir al operario responsable registrar las horas de trabajo de cada uno de los operarios que han intervenido en la tarea. Además, también se ha de poder indicar el tipo de horas (normales, extra, nocturnas...).
Importancia	Alta
Secuencia normal	
1	En la pantalla de edición de parte, seleccionar opción de horas de empleado.
2	Seleccionar opción de registrar horas de empleado.
3	Seleccionar empleado que ha intervenido en el trabajo.
4	Introducir cantidad y tipo de horas.
Comentarios	Cuando se vaya a seleccionar empleados que han participado en la tarea, el sistema ha de permitir seleccionar cualquier empleado de la empresa. Además, ha de poder introducir un empleado externo. Para ello, el empleado responsable del parte, habrá de introducir el nombre de dicha persona manualmente.

Requisito funcional	
Código	FR10
Nombre	Registrar horas de vehículos
Descripción	Cuando se realiza la tarea descrita en un parte de trabajo, el sistema ha de permitir al operario responsable registrar si se ha utilizado algún vehículo, y en caso afirmativo, indicar las horas de funcionamiento.
Importancia	Alta
Secuencia normal	
1	En la pantalla de edición de parte, seleccionar opción de horas de vehículo.
2	Seleccionar opción de registrar horas de vehículo.
3	Seleccionar vehículo que se ha utilizado.
4	Introducir horas funcionamiento.
Comentarios	Además de seleccionar vehículos de la empresa, el sistema ha de permitir indicar el uso de un vehículo externo. En este caso, el empleado responsable del parte, habrá de introducir la matrícula de dicho vehículo manualmente.

Requisito funcional	
Código	FR11
Nombre	Registrar materiales
Descripción	Cuando se realiza la tarea descrita en un parte de trabajo, el sistema ha de permitir al operario responsable registrar los materiales que se han utilizado para llevar a cabo la tarea.
Importancia	Alta
Secuencia normal	
1	En la pantalla de edición de parte, seleccionar opción de materiales.
2	Seleccionar opción de añadir material.
3	Seleccionar material que se ha utilizado.
4	Introducir cantidad utilizada.
Comentarios	Los tipos y jerarquía de materiales se explican con detalle en el apartado de requisitos de datos.

4.1.3. Requisitos de datos

Para satisfacer los requisitos de usuario expuestos anteriormente, es necesario definir las entidades y atributos que se van a almacenar, es decir, los requisitos de datos.

A continuación, se muestran las tablas que representan cada entidad y los propiedades que contendrán cada una de estas:

Requisito de datos	
Código	DR01
Nombre	Empleado
Datos	Para cada empleado, el sistema debe almacenar el nombre y apellidos, empresa a la que pertenece y brigada a la que pertenece (en caso de pertenecer a alguna).
Comentarios	Un empleado puede confundirse con un usuario de la aplicación. Un usuario siempre es un empleado, en cambio, existen empleados que no tienen acceso a al sistema y, por tanto, no son usuarios.

Requisito de datos	
Código	DR02
Nombre	Usuario
Datos	Para cada usuario, el sistema debe almacenar el nombre de usuario (*login*), la contraseña y el empleado asociado a este usuario.
Comentarios	

Requisito de datos	
Código	DR03
Nombre	Parte de trabajo
Datos	Para cada parte, el sistema debe almacenar la dirección de actuación, la fecha, la incidencia, el departamento, la razón social, la descripción del trabajo, el número de póliza, el empleado que al que ha sido entregado (capataz de brigada), empleado que ha finalizado el parte (empleado asignado), empleados que han participado en la tarea y vehículos y materiales que se han utilizado.
Comentarios	

Requisito de datos	
Código	DR04
Nombre	Incidencia
Datos	Para cada incidencia se debe almacenar el nombre y una lista de preguntas asociadas. Estas preguntas pueden ser de seis tipos, según el tipo de respuesta que requieran: texto, numérico, sí/no, fecha, hora o de tipo lista. En las preguntas de tipo lista, el operario habrá de seleccionar una de una serie de opciones definidas por el dominio de la pregunta.
Comentarios	Como se ha dicho anteriormente, de las preguntas también se debe conocer si son obligatorias o no.

Requisito de datos	
Código	DR05
Nombre	Material
Datos	Existen sectores, que contienen familias. A la vez, estas familias contienen subfamilias. Finalmente, cada material pertenece a una de estas subfamilias.
Comentarios	Cuando el operario quiera seleccionar un material, habrá de recorrer esta jerarquía hasta llegar al material propiamente dicho.

Requisito de datos	
Código	DR06
Nombre	Vehículo
Datos	Para cada vehículo, se ha de almacenar la matrícula, marca y modelo. Además, existen vehículos asociados a una brigada. En ese caso, también se ha de almacenar la brigada a la que pertenecen.
Comentarios	En algunos casos, una brigada puede tener un vehículo asignado y es frecuente su uso entre los operarios de esta. Por esto, se debería hacer intuitiva la selección del vehículo de la brigada a la que pertenece el usuario que esta haciendo uso del sistema.

a

Requisito de datos	
Código	DR07
Nombre	Abastecimiento
Datos	Un parte tiene asociado un departamento o abastecimiento. Cada usuario tiene permisos para crear y ser asignado a partes de una serie de abastecimientos. Por tanto, para cada usuario, se necesita conocer a qué abastecimientos tiene acceso.
Comentarios	Cuando se cree un nuevo parte, el sistema pedirá al operario que rellene el campo de abastecimiento. El sistema sólo debe mostrar aquellos a los que el usuario tenga permiso.

Requisito de datos	
Código	DR08
Nombre	Brigada
Datos	Cada brigada contiene un conjunto de operarios como la componen, así como un empleado que es el capataz. Además, tiene un vehículo asignado.
Comentarios	Que cada brigada tenga asignado un vehículo no significa que se vaya a hacer uso de este o que sólo se vaya a utilizar ese, sino que es habitual su uso para esta brigada.

4.1.4. Requisitos de software y hardware

La aplicación que se describe en este proyecto ha sido testada sobre una tableta Samsung Galaxy PT-1000 de 7" con sistema operativo Android 4.2.2. Los requisitos para ejecutar esta aplicación son los siguientes:

- Smartphone o tableta con sistema operativo Android 4.0 o posterior.
- Conexión a Internet. Aunque la aplicación puede correr sin conexión a Internet, en un determinado momento se han de sincronizar los datos con el servidor.
- Almacenamiento interno mínimo para almacenar la base de datos local. El espacio dependerá del volumen de datos relativo a los maestros (incidencias, vehículos, abastecimientos, etc...) que se vayan a descargar.

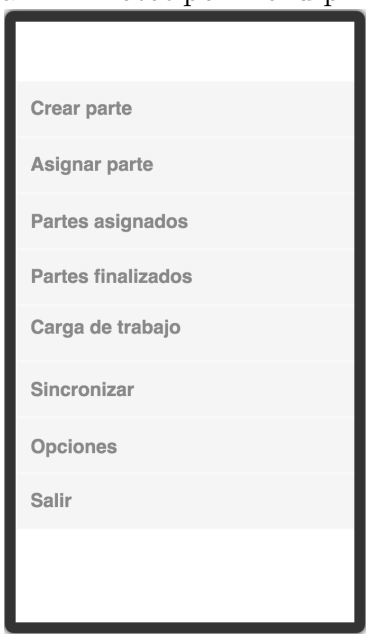
El espacio interno que ocupan los partes que se crean utilizando la aplicación es muy pequeño a comparación del que ocupen los datos sobre maestros. Aun así, cualquier dispositivo actual es capaz de almacenar grandes cantidades de datos, por lo que no debería suponer ningún problema.

- Respecto a la interfaz gráfica de usuario, la aplicación ha sido diseñada para dispositivos de 7" y con una resolución de 600x1024 píxeles. Al tratarse, de una aplicación interna de empresa y no de comercio masivo, se ha desarrollado y probado exclusivamente sobre los dispositivos en los que correrá. En un dispositivo con especificaciones gráficas distintas, la interfaz de usuario podría verse afectada.

4.1.5. Prototipos de la interfaz

Para terminar con la fase del análisis, se han creado prototipos de las interfaces de usuario para las funcionalidades más significativas de la aplicación: menú principal, crear un nuevo parte, asignar partes y ver la carga de trabajo de los operarios. Estos prototipos no representan el aspecto definitivo de las pantallas, sino que su objetivo es plasmar una idea más visual de la aplicación para permitir la verificación de las funcionalidades por parte del cliente, sin entrar en aspectos de diseño.

Figura 4.2: Prototipo: Menú principal



Las figuras 4.2, 4.3, 4.4 y 4.5 muestran los prototipos generados para las pantallas de menú principal, crear parte, asignar partes y carga de trabajo, respectivamente.

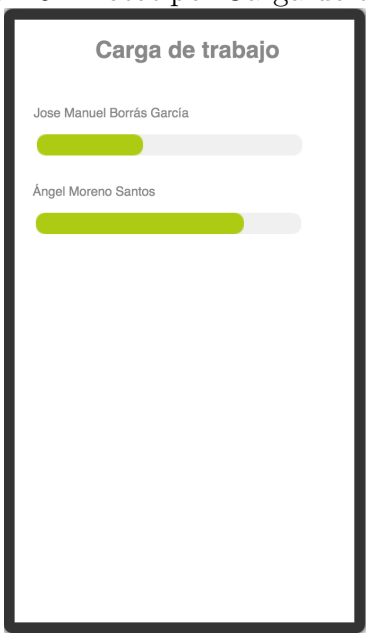
Figura 4.3: Prototipo: Crear parte

Figura 4.4: Prototipo: Asignar partes

4.2. Diseño e implementación

Una vez terminada la fase de análisis, se procede a modelar el sistema a nivel de arquitectura y componentes de software. La etapa de diseño permite a los desarrolladores tener una idea más clara y concreta de cómo será el sistema, reduciendo la abstracción respecto a la fase anterior.

Figura 4.5: Prototipo: Carga de trabajo



4.2.1. Arquitectura de la aplicación

El primer paso en la fase de diseño es definir la arquitectura del sistema a alto nivel. La arquitectura es una especificación de la estructura a nivel global del sistema. Se centra en aspectos de más alto nivel que los algoritmos, funciones o tipos de datos. Estos últimos forman parte del diseño a nivel de componentes.

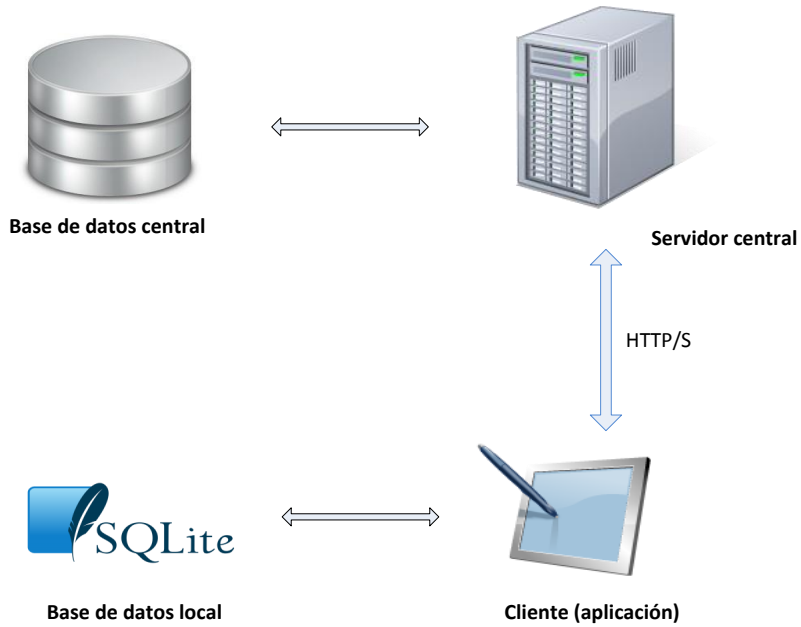
En este caso, la arquitectura presente es de cliente/servidor, ya que existe un cliente (aplicación Android) que solicita y envía datos al servidor central.

La Figura 4.6 muestra todos los módulos que están dentro del alcance del sistema, así como los que no lo están pero cumplen alguna función en todo el ciclo de vida de los partes de trabajo.

En este diagrama encontramos cuatro módulos distintos:

- **Base de datos central:** Esta es la base de datos central de la empresa. Contiene la información de usuarios, empleados, partes, etc... La aplicación que se está describiendo en este documento no realizará ningún acceso a esta.
- **Servidor central:** Esta es la máquina que se encargará de la comunicación entre el cliente (aplicación) y los datos que se encuentran en la base de datos central. Como se ha explicado anteriormente, esta comunicación se realizará mediante un servicio web REST.
- **Cliente:** Este módulo, junto con la base de datos local, componen el sistema que se

Figura 4.6: Arquitectura



esta diseñando en este proyecto. El cliente realizará peticiones y enviará información al servidor central para que esta sea procesada y persistida en la base de datos central.

- **Base de datos local:** Como se ha explicado anteriormente, la aplicación necesita persistir los datos en el dispositivo. Para ello, se ha utilizado el sistema de gestión de bases de datos *SQLite*.

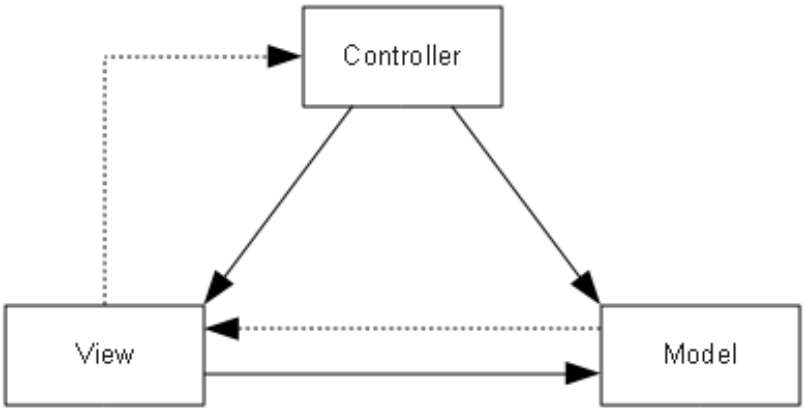
4.2.2. Diagramas de clases

En este apartado se van a mostrar algunos diagramas relativos al diseño a nivel de componentes. Esta etapa del diseño se centra en la organización de las clases e interfaces y cómo se relacionan entre sí. Debido a la complejidad del proyecto, no se pueden mostrar todos los diagramas de clases. Por ello, se mostrarán aquellos que tienen más importancia, ya sea porque se adaptan a un patrón de diseño predefinido o porque simplemente se crea conveniente mencionar.

4.2.2.1. Patrón arquitectónico MVC (Modelo Vista Controlador)

El patrón de diseño o arquitectónico Modelo Vista Controlador es el patrón que engloba toda la aplicación. Todos los componentes del sistema, se pueden ubicar dentro de uno de estos tres módulos. El objetivo de este patrón es separar la lógica de negocio, de la interfaz de usuario y los datos del modelo. A continuación, se definen los tres módulos que componen este patrón:

Figura 4.7: Modelo Vista Controlador



- **Modelo**: Este componente tiene que ver con las representaciones creadas de los datos que va a usar la aplicación. Cada elemento debe tener una representación bien definida y única, con el objetivo de no repetir código y evitar información redundante. Un componente del modelo podría ser la clase *Vehiculo*, ya que es una representación de una entidad de información. La base de datos también se encuentra dentro de este componente.
- **Vista**: Es el componente con el que el usuario interactúa: la interfaz de usuario o GUI. En una implementación pura de MVC, la vista no tiene ningún estado y no realiza acciones, sino que las delega en el controlador.
- **Controlador**: Este componente es el encargado de gestionar la lógica de la aplicación, responder a eventos, peticiones de la vista por parte del usuario, etc.

Gracias a este patrón, la aplicación es más escalable, extensible y capaz de intercambiar alguno de los tres componentes, mientras se cumpla la misma interfaz. De este modo, cambiando la vista se podría obtener una aplicación web o de otro tipo, solo aplicando ligeros cambios. Además, al estar los componentes bien separados, es fácil repartir las tareas de desarrollo y de diseño de interfaz entre los participantes en el proyecto.

4.2.2.2. Patrón DAO (*Data Access Object*)

Data Access Object (DAO) es un patrón de diseño que proporciona una interfaz para acceder a los datos, ya sean en base de datos, un fichero o cualquier otro medio de almacenamiento. Este patrón oculta al programador que usa la interfaz la forma en que se almacenan los datos. Esto permite más abstracción entre la capa de aplicación y la capa de persistencia.

En esta aplicación se han desarrollado cinco interfaces DAO. Cada una de estas permite el acceso a datos de diferente ámbito. Las interfaces son las siguientes:

- **UsuariosDAO:** Mediante esta interfaz se accede a los datos relativos a los usuarios y empleados.
- **PartesDAO:** Mediante esta interfaz se accede a los datos relativos a los partes de trabajo.
- **VehiculosDAO:** Mediante esta interfaz se accede a los datos relativos a vehiculos.
- **MaterialesDAO:** Mediante esta interfaz se accede a los datos relativos a los sectores, familias, subfamilias de materiales y a los propios materiales.
- **MaestrosDAO:** Mediante esta interfaz se accede a los datos relativos a brigadas, incidencias y abastecimientos.

Cada una de estas interfaces tiene una clase concreta que la implementa, y es donde se realiza el acceso a la base de datos local. En la Figura 4.8 se muestra el diagrama de clases correspondiente a estas.

Además, tal como se muestra en la Figura 4.8 existe una clase FakeDAO que implementa todas las interfaces DAO. Esto es porque mediante el patrón factoría se obtiene una implementación real, es decir, una instancia de las clases con terminación DAODB, o una instancia *fake*, que utiliza datos alojados en memoria RAM y no están persistidos en base de datos.

La Figura 4.9 muestra la estructura del patrón factoría adaptado, para el caso del DAO de usuarios. Todas las interfaces DAO tienen esta misma estructura, con su correspondiente clase factoría.

La clase UsuariosDAOFactory contiene una instancia de FakeDAO y otra de UsuariosDAODB. Cuando se llama al método *getUsuariosDAO* de esta, se consulta la propiedad *USE_FAKE_DATA* de un fichero de configuración. Si esta propiedad está a *true*, se devolverá una instancia de FakeDAO, sino una de UsuariosDAODB. Esto es posible ya que ambas clases implementan la interfaz UsuariosDAO.

Figura 4.8: Diagrama de clases DAO

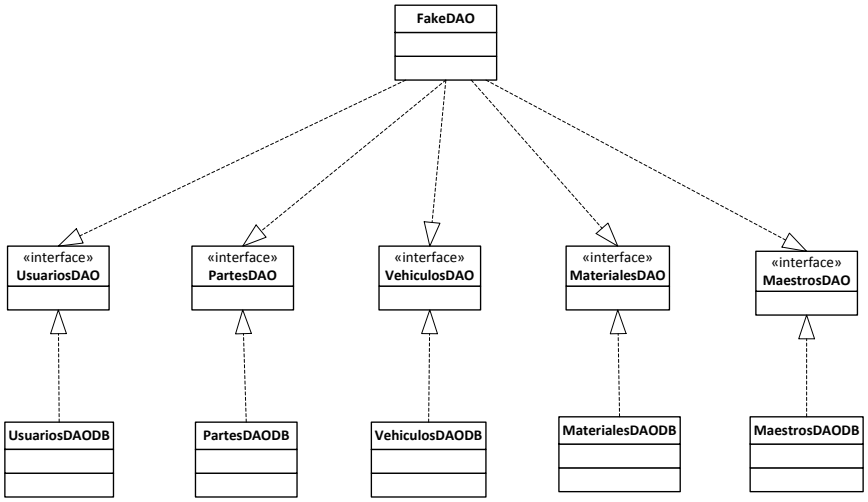
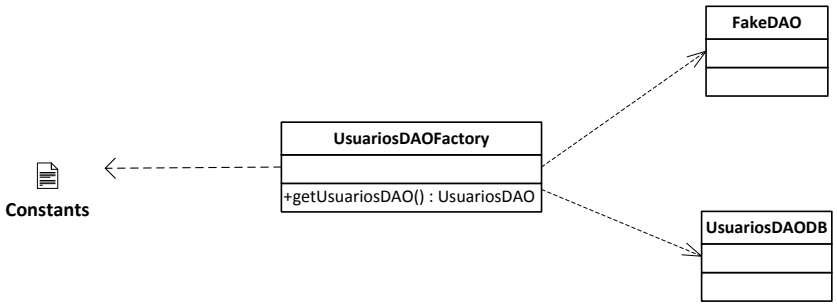


Figura 4.9: Diagrama de clases factoría DAO



4.2.2.3. Patrón Observador

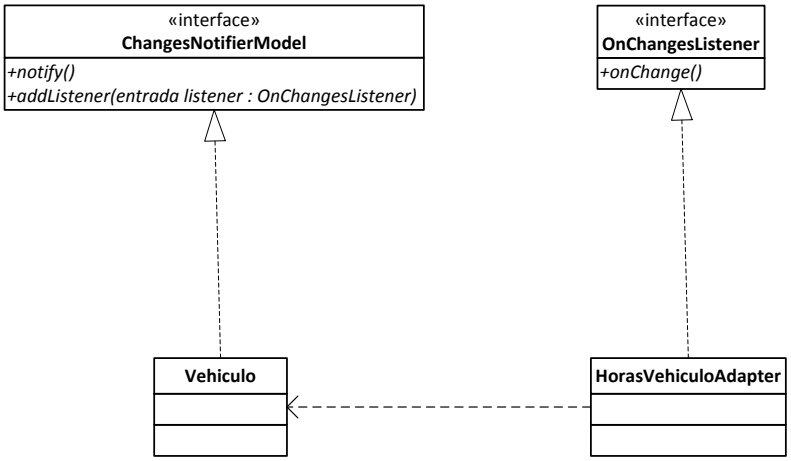
En este patrón, un componente (observador) se suscribe a otro (observado) para ser notificado cuando el segundo cambia su estado, esto permite crear una cadena de notificaciones entre varios componentes automáticamente. Es un patrón de suscripción/publicación (Freeman, 2004). En esta caso, hemos utilizado este patrón para notificar los cambios del modelo de datos a la interfaz gráfica.

La clase HorasVehiculoAdapter contiene los vehículos utilizados en el parte de trabajo que se esté editando y las horas de cada uno. El objetivo de esta clase es pintarlos en una lista (*ListView*, componente de Android) en la interfaz gráfica.

Cuando el operario pulsa el botón para aumentar las horas de un vehículo concreto, se puede hacer una llamada al controlador para que aumentará esta cantidad en el modelo y, posteriormente, repintar la nueva cantidad de horas en la interfaz gráfica. En lugar de esto, utilizando el patrón Observador sólo es necesaria una operación: indicar al controlador que aumente la cantidad en el modelo de datos. Así, sólo cuando el modelo cambie la vista será notificada para repintar los nuevos datos.

Para implementar este patrón, se han desarrollado las clases conforme se muestra en la Figura 4.10.

Figura 4.10: Diagrama de clases patrón Observador



Para cada vehículo que contiene el adaptador de la lista (HorasVehiculoAdapter), este llama al método *addListener()* pasándose como parámetro a sí mismo (*this*), suscribiéndose así a los cambios que experimente cada Vehiculo.

Cada Vehiculo contendrá una lista de observadores (instancias de clases que implementan la interfaz OnChangesListener). Cuando cambie su estado, en este caso, la cantidad de horas de uso, llamará al método *notify()* que a su vez, ejecutará *onChange()* en todos sus observadores.

Al recibir la notificación del cambio en el modelo de datos, el adaptador se encargará de repintar cada el elemento de la lista.

Con esto hemos conseguido dos cosas: realizar solo una operación, sin preocuparnos de la interfaz gráfica sino centrándonos sólo en la lógica de negocio y además, la interfaz gráfica siempre es fiel al modelo de datos: si este no se actualiza, la interfaz gráfica tampoco. De otra manera, en caso de error en el modelo de datos, podrían existir incoherencias entre los datos reales y los que se están mostrando al usuario.

Este mismo patrón se utiliza en las listas de horas de operarios y cantidad de materiales.

4.2.2.4. Patrón *Builder*

Builder es un patrón de diseño creacional, es decir, su cometido es la creación de instancias de alguna clase de forma desacoplada y transparente.

Para cada clase del modelo (Vehiculo, Material, Empleado, etc.) tenemos otra clase con el mismo nombre pero con terminación Builder. Esta clase tiene un método estático *create* que se encarga de crear una instancia del *builder*, y otros métodos *with* para definir los atributos concretos de una instancia de la clase que queremos construir. Finalmente, un método *build* genera la instancia en sí con los atributos correspondientes.

A continuación, se muestra un ejemplo de la clase MaterialBuilder:

```
public class MaterialBuilder {
    private String id = "";
    private String nombre = "";
    private SubfamiliaMaterial subfamilia;

    public static MaterialBuilder create() {
        return new MaterialBuilder();
    }

    public MaterialBuilder withNombre(String nombre) {
        this.nombre = nombre;
        return this;
    }

    public MaterialBuilder withId(String id) {
        this.id = id;
        return this;
    }

    public Material build() {
        Material m = new Material();
        m.setNombre(nombre);
        m.setId(id);
        m.setSubfamilia(subfamilia);
        return m;
    }
}
```

De esta forma, el usuario obtendría una nueva instancia de la clase Material ejecutado la siguiente instrucción:

```
Material nuevoMaterial = MaterialBuilder.create()
        .withId("1")
        .withNombre("Tubo de acero 3mm")
        .build();
```

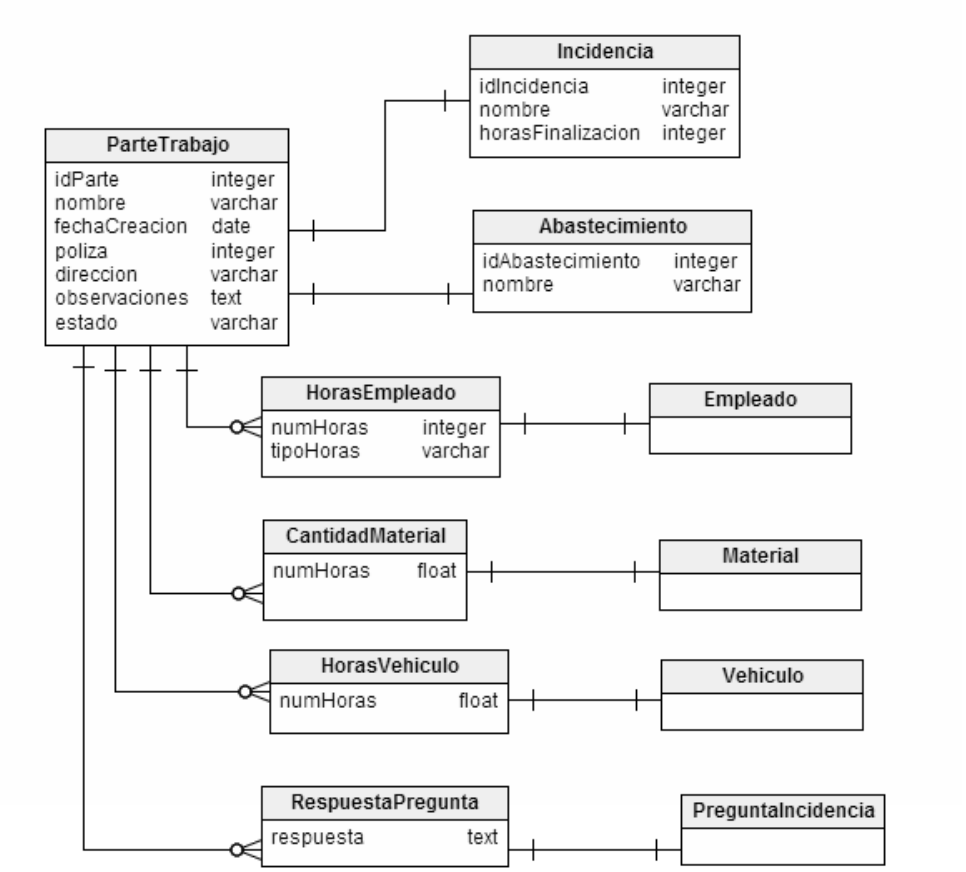
Usando este patrón, nos aseguramos que la creación es transparente al programador que instancia dicha clase, ya que no se utiliza la palabra reservada *new*. En un futuro, se podría cambiar la estructura interna de la clase Material y no afectaría al funcionamiento. Si se añadiesen atributos, se podría ampliar el *builder* en lugar de crear constructores, que acoplan el código y hacen difícil su mantenimiento.

4.2.3. Diseño de la base de datos

Como se ha dicho anteriormente, para persistir los datos en el dispositivo se ha hecho uso de una base de datos mediante el sistema de gestión SQLite. En este apartado se van a analizar los diagramas Entidad-Relación que representan el diseño de dicha base de datos. Cada tabla contiene un identificador de tipo *integer*, que es la clave primaria, y una serie de columnas para cada atributo. A continuación, se muestran los diagramas E-R:

- **Partes de trabajo**: La Figura 4.11 muestra el diagrama entidad relación relativo a los partes de trabajo. Como se puede observar, cada parte de trabajo tiene una clave ajena a una incidencia y otra a un abastecimiento, que a la vez son tablas de la base de datos. Las tablas *HorasEmpleado*, *CantidadMaterial*, *HorasVehiculo* y *RespuestaPregunta* almacenan la cantidad de horas que un operario ha trabajado en el parte, la cantidad de materiales que se han utilizado, los vehículos que se han empleado y las respuestas de las preguntas de la incidencia, en caso de tener alguna.
- **Incidencias**: La Figura 4.12 muestra cómo se almacenan las incidencias. Cada una de ellas puede tener ninguna o varias preguntas de cada uno de los cinco tipos existentes. Cada pregunta de tipo lista tiene asignado un de valores posibles de respuesta. En el resto de tipos, la respuesta del operario será libre, siempre que sea válido el tipo de datos.
- **Brigadas**: En la Figura 4.13 se muestra el diagrama de las tablas que corresponden a la persistencia de las brigadas de operarios. Cada brigada tiene asignado un vehículo y tiene uno o muchos operarios que la componen.

Figura 4.11: Diagrama E-R partes de trabajo



4.2.4. Diagramas de flujo

Un diagrama de flujo se utiliza para representar gráficamente un proceso del sistema. Mediante símbolos estándar definidos e interconectados entre sí se representan los pasos de un proceso.

Los símbolos son los siguientes:

- Punto negro: Punto de entrada.
- Punto negro dentro de círculo: Puntos de finalización.
- Rectángulo: Actividad o acción.
- Rombo: Punto de decisión/Pregunta.

La Figura 4.14 muestra el diagrama de flujo de la actualización de datos al arrancar la aplicación. En la primera ejecución, se descargan los datos, y se muestra la pantalla de

Figura 4.12: Diagrama E-R preguntas de incidencia

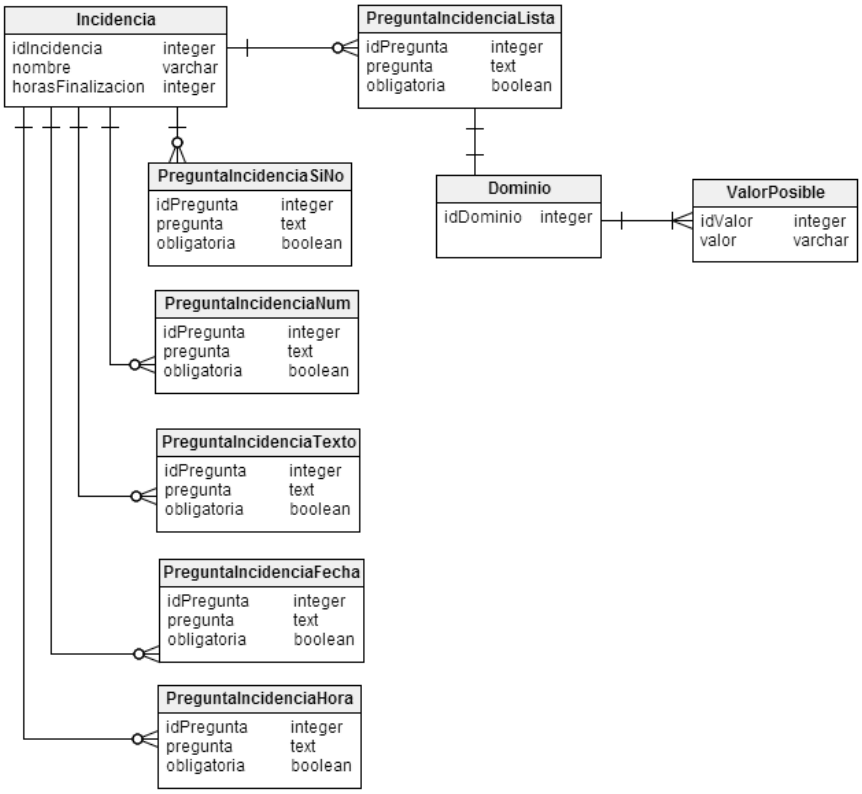
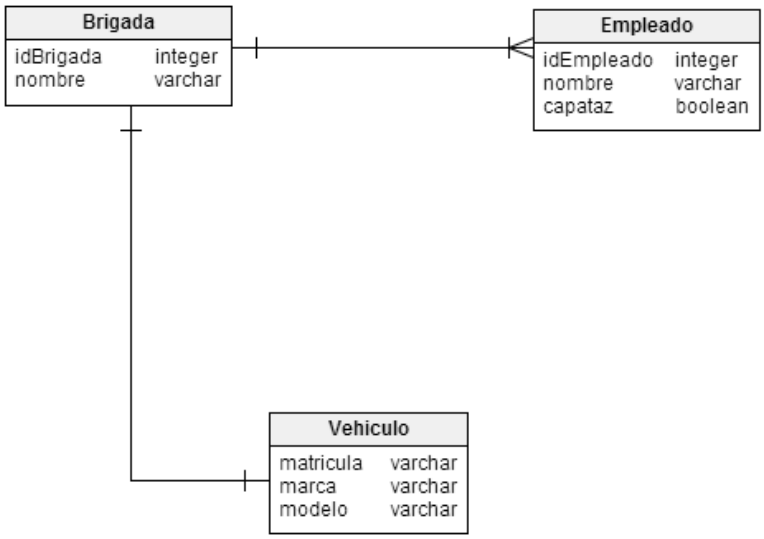
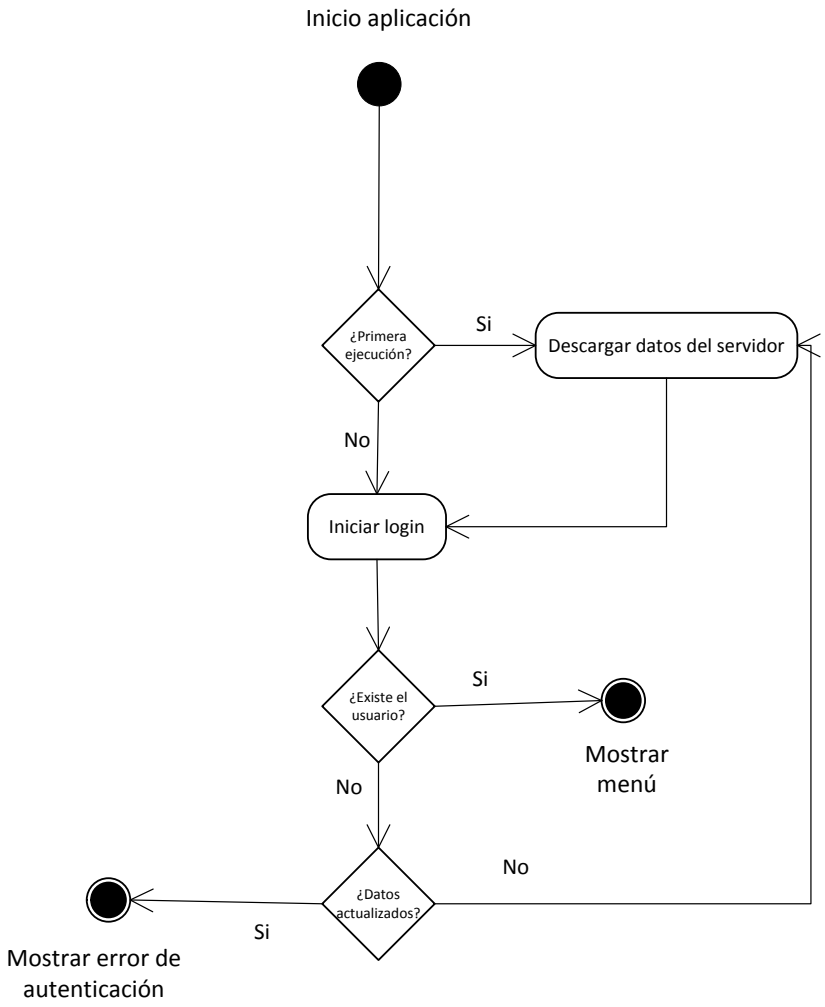


Figura 4.13: Diagrama E-R brigadas de operarios



login al usuario. Si no se trata de la primera ejecución y el usuario introduce un login que no existe, el sistema comprueba si tiene los datos actualizados para mostrar un error, y sino, para descargar de nuevo los usuarios del servidor. Este segundo caso podría darse en el caso de que se dé de alta un nuevo usuario en el sistema central. Este nuevo usuario no estaría almacenado en la base de datos del dispositivo.

Figura 4.14: Diagrama de flujo de actualización de datos



Este es el único proceso que se ha encontrado que ha sido necesario plasmar en un diagrama de flujo para comprenderlo fácilmente y de manera visual.

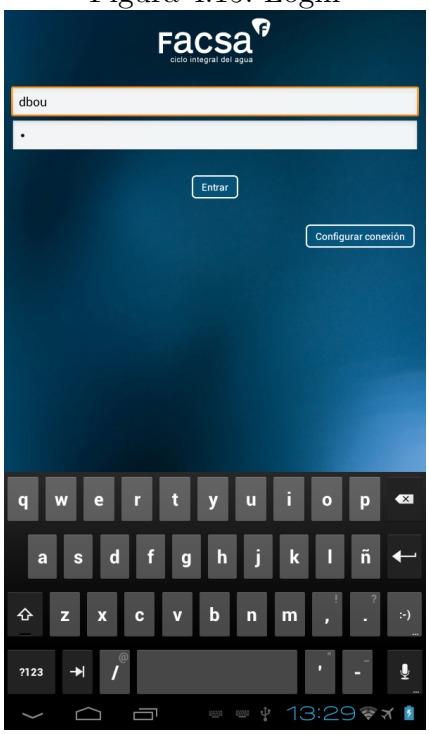
4.2.5. Interfaz de usuario

El kit de desarrollo de aplicaciones de Android ofrece la posibilidad de diseñar la interfaz gráfica mediante código XML. De esta forma, cada pantalla (en Android, *Activity*) se compone de un fichero con extensión *.java*, donde se desarrolla la lógica de negocio, junto con un fichero XML que representa el aspecto gráfico de esta.

4.2.5.1. Login

La Figura 4.15 muestra la pantalla de login. El operario que vaya a utilizar la aplicación estará dado de alta previamente en la base de datos central. Además, los datos de ese usuario estarán almacenados en la base de datos local, ya que la aplicación se ha de poder utilizar sin conexión a Internet, por tanto, el proceso de login no se realiza contra el servidor, sino contra la base de datos del dispositivo.

Figura 4.15: Login



4.2.5.2. Menú principal

La Figura 4.16 corresponde al menú principal de la aplicación. En la figura se muestra el menú que observaría un empleado capataz. En caso de ser un operario, las opciones de "Partes pendientes de asignar"(opción 1) y "Carga de trabajo" (opción 2) no estarían

visibles, ya que estas funciones sólo están permitidas para los capataces, tal como se explica previamente en el apartado de casos de uso.

Figura 4.16: Menú principal

4.2.5.3. Partes pendientes de asignar

Si el empleado capataz selecciona la primera opción del menú principal, se mostrará la pantalla de la Figura 4.17. En esta pantalla se observan todos los partes asignados a la brigada de este capataz que están pendientes de asignación.

Cuando el capataz selecciona un parte, se abre un diálogo que permite elegir entre varias opciones (Ver Figura 4.18) según quiera asignar el parte a un operario o a sí mismo.

4.2.5.4. Carga de trabajo

Esta pantalla permite al capataz ver la carga de trabajo de cada uno de los operarios de su brigada (Ver Figura 4.19) y, de esta manera, regular las horas de trabajo de cada operario y evitar que se produzcan sobrecargas.

Por cada operario, se muestra el número de partes que tiene asignado, las horas estimadas de finalización y una barra que representa el número de horas de trabajo asignado respecto a una jornada laboral de ocho horas.

Figura 4.17: Partes pendientes de asignar

Figura 4.18: Partes pendientes de asignar. Diálogo

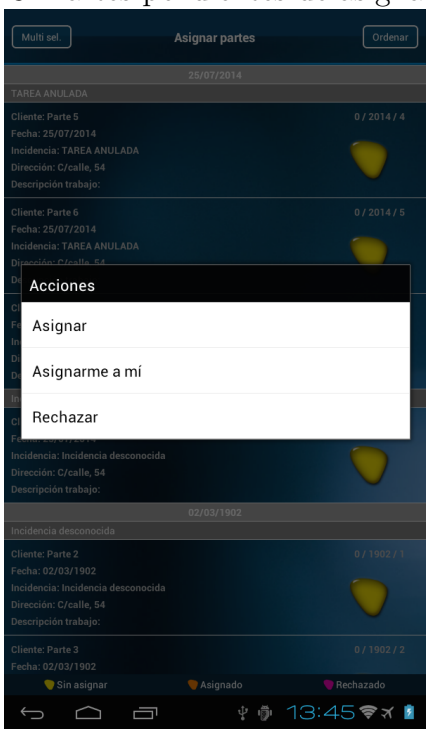
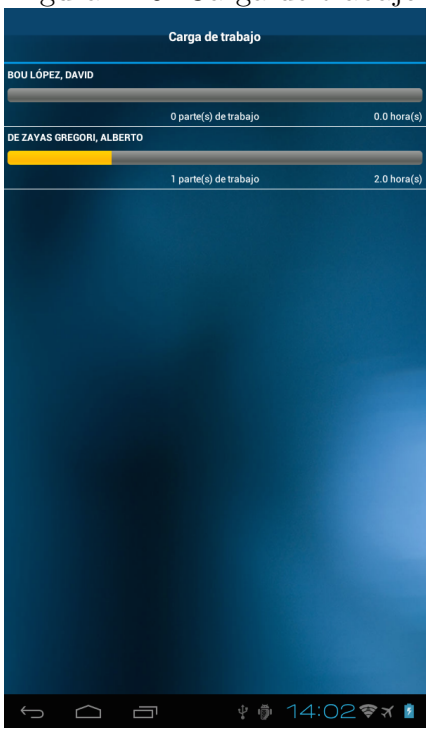


Figura 4.19: Carga de trabajo



4.2.5.5. Crear nuevo parte

La pantalla de crear nuevos partes se divide en varios apartados según el tipo de información que el operario quiera agregar al parte. Estos apartados son los siguientes:

- **Info**: Contiene los datos generales del parte: incidencia, abastecimiento, nombre del cliente, dirección, número de póliza y observaciones. Ver Figura 4.20.
- **Materiales**: En este apartado el operario puede indicar los materiales que se han utilizado para llevar a cabo la tarea y la cantidad de cada uno de estos (Ver Figura 4.21).
- **Horas**: En este apartado el operario puede indicar los operarios que han participado en la tarea, así como la cantidad y tipo de horas dedicadas. Ver Figura 4.22.
- **Vehículos**: En este apartado el operario puede indicar los vehículos que se han utilizado y las horas de cada uno (Ver Figura 4.23).
- **Finalizar**: Este apartado muestra las preguntas asignadas a la incidencia que el operario debe responder antes de finalizar el parte (Ver Figura 4.24).

  Si el usuario pulsa el botón de finalizar y la validación no tiene éxito, se muestra un diálogo con una relación de los errores que se han encontrado (Ver Figura 4.25).

Figura 4.20: Crear parte nuevo. Info

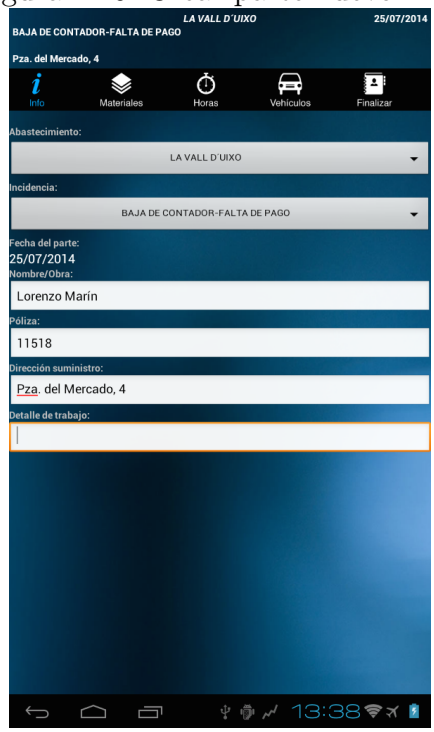


Figura 4.21: Crear parte nuevo. Materiales

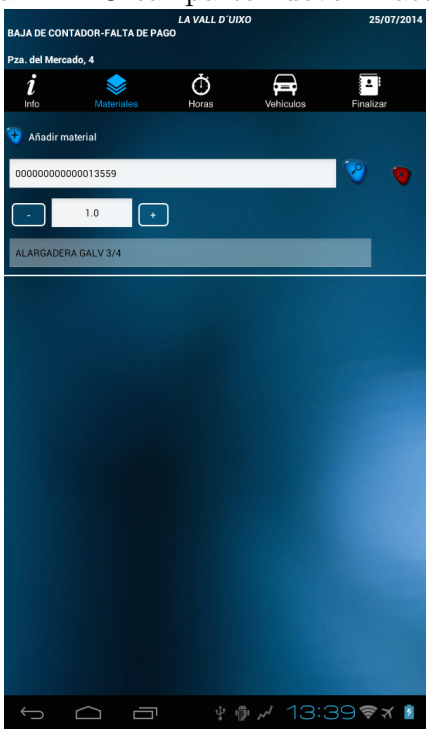


Figura 4.22: Crear parte nuevo. Horas

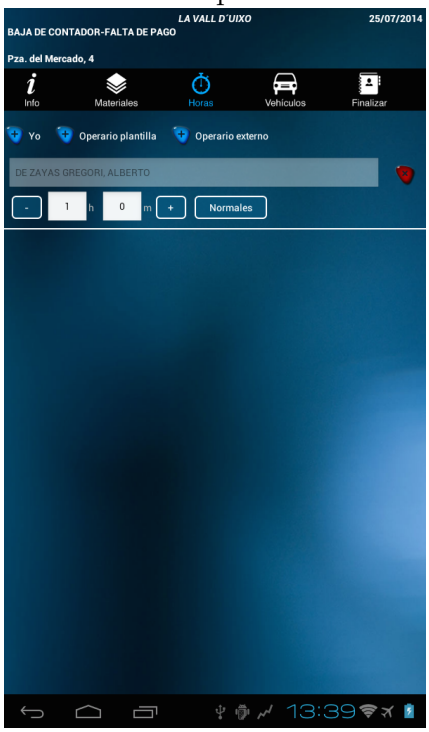
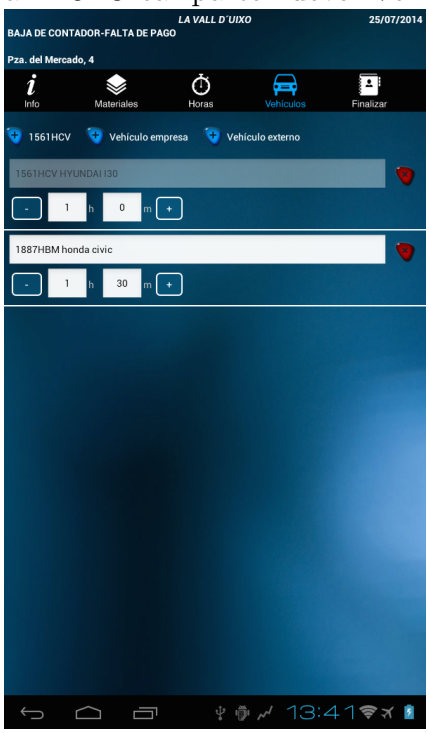


Figura 4.23: Crear parte nuevo. Vehículos



4.2.5.6. Partes asignados

Si el usuario elige esta opción, podrá visualizar los partes que le han sido asignados. La pantalla es similar a la de "Partes pendientes de asignar". Cuando el usuario selecciona

Figura 4.24: Crear parte nuevo. Finalizar

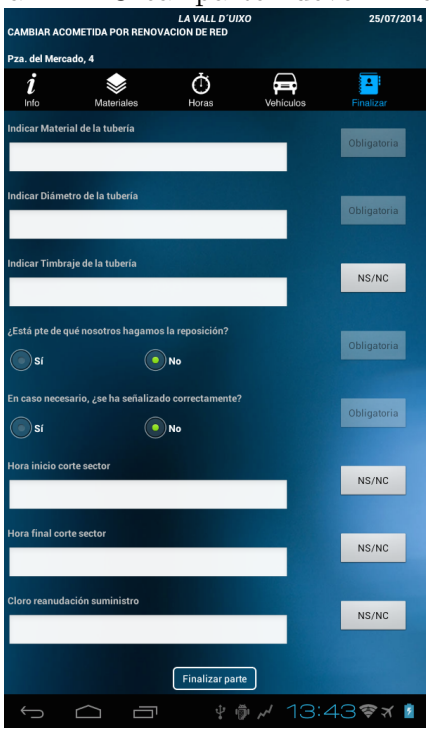
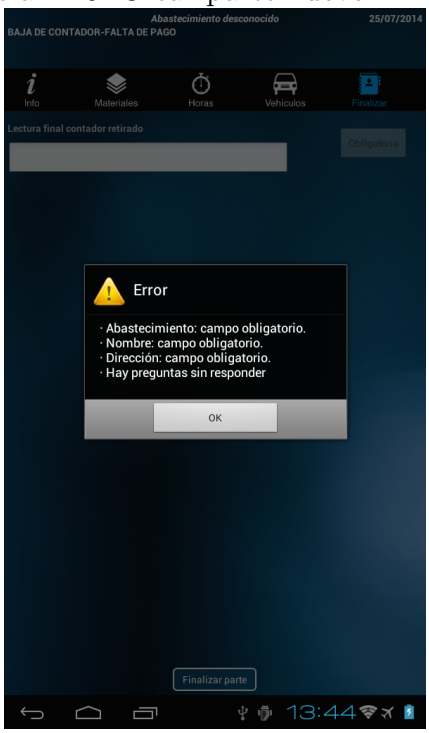


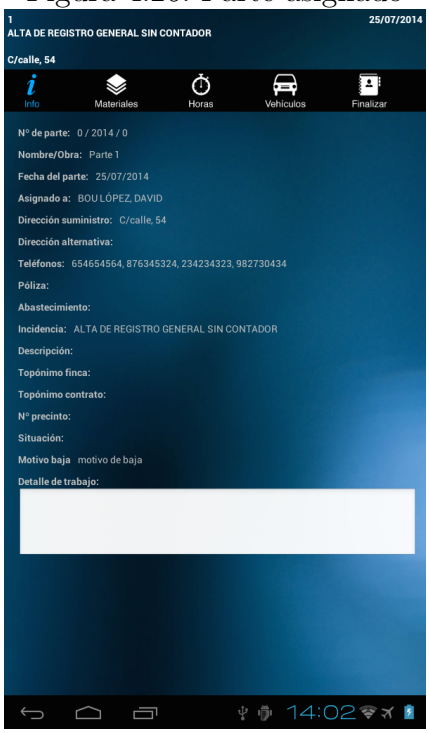
Figura 4.25: Crear parte nuevo. Errores



un parte de la lista se muestra una pantalla semejante a la de crear partes nuevos, con la diferencia de que en esta última el usuario no podrá editar la información general del parte, ya que estos datos ya están rellenados al descargarse del servidor.

La Figura 4.26 muestra el apartado de información general de un parte asignado.

Figura 4.26: Parte asignado



4.2.5.7. Sincronizar

Utilizando esta opción, el usuario envía los partes finalizados hasta ese momento al servidor. Además, en la parte del servidor, se comprueba si ha habido cambios en los maestros (nuevos usuarios, modificación de una incidencia, etc...) y, en ese caso, la aplicación los descarga para mantener la información local actualizada. Ver Figura 4.27.

4.2.5.8. Opciones

Esta opción esta protegida por contraseña, y no es accesible para los usuarios, sino que esta dirigida a los desarrolladores. En la Figura 4.28 se observa que hay dos opciones:

- **Configurar red**: Esta opción permite a los desarrolladores cambiar la dirección URL del servicio REST. Esto podría ser útil para realizar pruebas contra un servidor en desarrollo, en lugar de utilizar el de producción, sin necesidad de recompilar el código.
- **Copia de seguridad**: Esta opción permitiría volver a un estado anterior la base de datos local del dispositivo en caso de algún fallo o corrupción de datos. Como se

Figura 4.27: Sincronizar

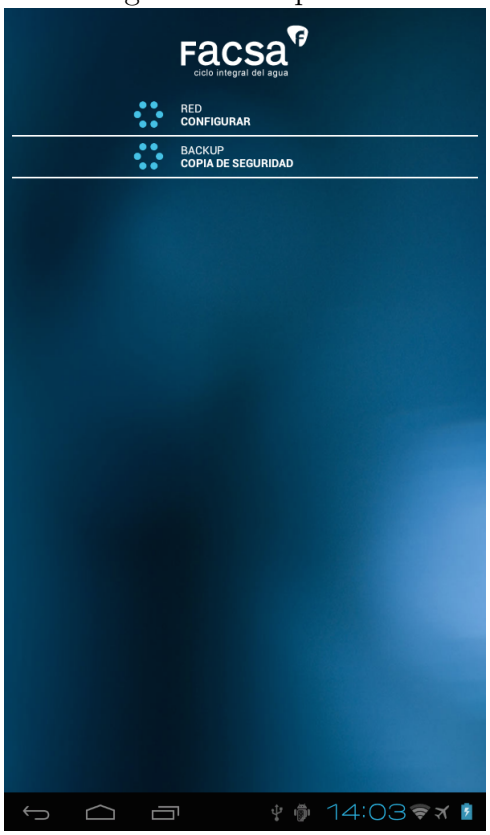
ha mencionado en apartados anteriores, la base de datos local consiste en un fichero con extensión *.db*. Cada cierto tiempo, la aplicación realiza una copia de este fichero y cuando se selecciona la opción de "Copia de seguridad" ofrece reemplazar la base de datos primaria por una de las copias realizadas.

4.3. Validación y verificación

El objetivo de la fase de validación y verificación del sistema es comprobar el correcto funcionamiento de este a todos los niveles. Esta fase es esencial en el proceso de desarrollo de software ya que proporciona información sobre la calidad del sistema que se está desarrollando. Existen varios tipos de pruebas según su alcance o según sea el tipo de componente que se esta probando:

- *Pruebas unitarias*
- *Pruebas de integración*
- *Pruebas de carga*
- *Pruebas de estrés*
- *Pruebas de aceptación*

Figura 4.28: Opciones



El objetivo de todo desarrollo software no es llevar a cabo todos los tipos de pruebas, sino que para cada caso concreto hay que realizar un estudio con el objetivo de definir qué pruebas son útiles y viables, teniendo en cuenta variables como el tiempo o el presupuesto. En este proyecto se han desarrollado pruebas unitarias y de integración.

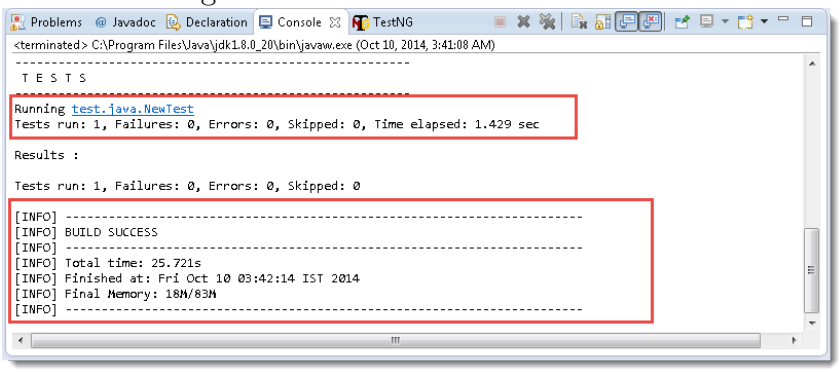
4.3.1. Pruebas unitarias

Las pruebas unitarias son aquellas que se encargan de comprobar el correcto funcionamiento de cada módulo o componente que constituye el sistema. La prueba de cada módulo es independiente de los demás, de esta forma nos aseguramos de que cada uno de los módulos trabaja como se espera por separado.

Para desarrollar las pruebas unitarias en este proyecto se han utilizado las herramientas *JUnit* y *Mockito*. *JUnit* es un *framework* libre y de código abierto que permite definir casos de pruebas unitarias para programas escritos en Java. *Mockito* se ha utilizado para la simulación de comportamientos en componentes complejos. Esto permite desarrollar pruebas unitarias para componentes que tienen una alta dependencia con otros, sin llegar a ser pruebas de integración.

Una vez implementadas las pruebas unitarias, ejecutando el comando *mvn test* en el directorio raíz del proyecto (donde se encuentra el fichero *pom.xml*), Maven se encargará de compilar y ejecutar todas estas y mostrar un informe de los resultados.

Figura 4.29: Maven: Informe de tests



La Figura 4.29 muestra un ejemplo donde se ha ejecutado una prueba y el resultado ha sido satisfactorio.

Antes de poder ejecutar las pruebas, necesitamos indicarle a Maven bajo qué directorio se encuentran las clases de test. En el fichero *pom.xml*:

```
1 <build>
2         <finalName>${project.artifactId}-${project.version}</
             finalName>
3
4         <sourceDirectory>${project.basedir}/src</
             sourceDirectory>
5
6         <testSourceDirectory>${project.basedir}/test</
             testSourceDirectory>
7         ...
8 </build>
```

La etiqueta *targetSourceDirectory* indica que el directorio */test* en el directorio raíz del proyecto contiene pruebas unitarias.

Estas pruebas se han estado ejecutando con mucha frecuencia para asegurar que cada cambio no ha afectado a la lógica implementada hasta el momento. Por esto, las pruebas que se han desarrollado han sido lo más independientes y simples posible, ya que el objetivo es consumir el menor tiempo posible en ejecución de tests unitarios.

4.3.2. Pruebas de integración y sistema

Una vez todas las pruebas unitarias se han aprobado satisfactoriamente, se realizan las pruebas de integración. En estas, se verifica que un subconjunto de componentes del sistema colaboran juntos y funcionan como se espera. Cuando se testea la integración entre todos los componentes que constituyen el sistema, las llamamos pruebas del sistema.

En este proyecto se han llevado a cabo las pruebas del sistema mediante el modelo propuesto por Martin Fowler: **integración continua**. Este paradigma consiste en automatizar las pruebas de integración, de manera que un software se encarga de descargar el código fuente de un repositorio de control de versiones, compilarlo, ejecutar pruebas y generar informes, cada cierto tiempo.

Esta práctica de programación permite detectar errores en el código lo antes posible y conocer el estado del software en todo momento.

Figura 4.30: Jenkins: correo electrónico de error

```
Enviado el: viernes, 19 de septiembre de 2014 9:11
Para:
Hector Valls
See <http://jenkins.grupogimeno.com/jenkins/job/gg-movbrigadas-android/172/changes>

Changes:

[hector] Añadida la funcionalidad de añadir vehiculos.

------------------------------------------
Started by an SCM change
Updating http://svn.grupogimeno.com/svn/grupogimeno/MOVBRIGADAS/gg-movbrigadas-android/trunk
D         movbrigadas.iml
D         project.properties
At revision 11480
Parsing POMs
[workspace] $ java -Djava.awt.headless=true -cp /root/.hudson/plugins/maven-plugin/WEB-INF/lib/maven3-agent-1.2.jar:/opt/devel/tools/apache-maven-3.0.5/boot/plexus-classworlds-2.4.jar org.jvnet.hudso
jenkins/webapps/jenkins/WEB-INF/lib/remoting-2.11.jar /root/.hudson/plugins/maven-plugin/WEB-INF/lib/maven3-interceptor-1.2.jar 46602
<===[JENKINS REMOTING CAPACITY]===>   channel started
Executing Maven:  -B -f <http://jenkins.grupogimeno.com/jenkins/job/gg-movbrigadas-android/ws/pom.xml> clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building movbrigadas-android 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
Downloading: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/maven-metadata.xml
Downloaded: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/maven-metadata.xml (789 B at 4.1 KB/sec)
Downloading: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/android-files-0.1.6-20140919.070834-1.pom
Downloaded: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/android-files-0.1.6-20140919.070834-1.pom (7 KB at 8.3 KB/sec)
Downloading: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-backups/0.1.4-SNAPSHOT/maven-metadata.xml
Downloaded: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-backups/0.1.4-SNAPSHOT/maven-metadata.xml (791 B at 30.9 KB/sec)
Downloading: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/android-files-0.1.6-20140919.070834-1.apklib
Downloaded: http://nexus.grupogimeno.com/nexus/content/groups/public/com/grupogimeno/android/android-files/0.1.6-SNAPSHOT/android-files-0.1.6-20140919.070834-1.apklib (10 KB at 11.7 KB/sec)
[INFO]
[INFO] --- maven-clean-plugin:2.4.1:clean (default-clean) @ movbrigadas ---
[INFO] Deleting <http://jenkins.grupogimeno.com/jenkins/job/gg-movbrigadas-android/ws/target>
log4j:WARN No appenders could be found for logger (org.apache.commons.beanutils.converters.BooleanConverter).
log4j:WARN Please initialize the log4j system properly.
[INFO] 13 errors
[INFO] ------------------------------------------------------------------------
[JENKINS] Archiving <http://jenkins.grupogimeno.com/jenkins/job/gg-movbrigadas-android/ws/pom.xml> to /root/.hudson/jobs/gg-movbrigadas-android/modules/com.grupogimeno.android$movbrigadas/bui
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 25.750s
[INFO] Finished at: Fri Sep 19 09:11:19 CEST 2014
[INFO] Final Memory: 24M/293M
[INFO] ------------------------------------------------------------------------
mavenExecutionResult exceptions not empty
message : Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:2.3.2:compile (default-compile) on project movbrigadas: Compilation failure
cause : Compilation failure
Stack trace :
org.apache.maven.lifecycle.LifecycleExecutionException: Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:2.3.2:compile (default-compile) on project movbrigadas: Compilation failure
        at org.apache.maven.lifecycle.internal.MojoExecutor.execute(MojoExecutor.java:213)
```

Para llevar a cabo esta tarea se ha utilizado el software *Jenkins*. Es un servidor de integración continua, gratuito, de código abierto y uno de los más utilizados en la actualidad.

Jenkins ha sido configurado de manera que cada vez que una nueva versión sea subida

a nuestro repositorio de control de versiones mediante Subversión, se lleve a cabo una tarea concreta (*build*), que será la prueba de integración propiamente dicha. Esta prueba supone recompilar el código, ejecutar pruebas de unidad y, en caso de error, enviar un correo electrónico a los desarrolladores implicados en el proyecto. Para realizar estos procedimientos, *Jenkins* necesita que el proyecto tenga naturaleza en las herramientas de construcción Maven o Ant: en nuestro caso, Maven.

La Figura 4.30 muestra un ejemplo de email de error. En este correo se puede observar que el responsable del último cambio en el código ha sido el usuario *hector*, añadiendo una funcionalidad nueva. El error del que *Jenkins* informa es un error de compilación, mostrando la salida por consola que devuelve Maven.

El modelo de desarrollo de integración continua ha sido esencial en este proyecto, ya que gran parte de las pruebas de integración detectaron fallos de varios tipos. Esto permite a los desarrolladores detectar este cambio rápidamente y encontrar una solución cuanto antes.

Capítulo 5

Conclusiones

En este Trabajo de Fin de Grado se ha llevado a cabo el desarrollo de una aplicación para el sistema operativo Android enfocada en el ámbito de la gestión y administración de partes de trabajo. Este sistema se ha implementado para permitir a la empresa *FACSA* entregar los partes de trabajo de cada jornada a sus operarios mediante un sistema informatizado. Así, los operarios de cada brigada de trabajo podrían conocer las tareas que deben llevar a cabo, recopilando los recursos (vehículos, materiales y operarios) que se han necesitado para completar el trabajo satisfactoriamente y almacenando esta información de forma persistente automáticamente.

Para poder iniciar el desarrollo en sí de la aplicación, ha sido necesario un estudio concreto del sistema operativo en cuestión, ya que, a pesar de que las aplicaciones de este estén basadas en el lenguaje Java, existen características concretas, como la arquitectura, los recursos o los ciclos de vida de cada componente. A parte de la implementación propiamente dicha, la fase de análisis ha sido en realidad muy extensa, ya que los requisitos de usuario eran complejos y la lógica de negocio, a veces, enrevesada. Además para conseguir el aspecto deseado por el cliente ha sido necesario profundizar en aspectos de diseño, componentes gráficos y gestión de estilos y temas en Android.

Todo el sistema está desarrollado sobre el patrón de diseño o arquitectónico Modelo Vista Controlador o MVC, cuyo objetivo es aislar la lógica de negocio del sistema, de los datos persistentes y la interfaz de usuario o GUI. El uso de este patrón es indispensable si se quiere ampliar la funcionalidad del sistema en un futuro. También, el empleo de paquetes para reestructurar la localización de las clases según su cometido es esencial para la reutilización de componentes.

Gracias a las pruebas unitarias realizadas se ha comprobado que todos los componentes funcionan como se espera y en las reuniones periódicas con un representante del cliente, se ha comprobado que todas sus expectativas se han cumplido. Más allá de lo exigido por el cliente, esta aplicación puede ser reutilizada para otra similar aplicando ligeras modificaciones, por ejemplo, para otra empresa del Grupo Gimeno. Además,

teniendo en cuenta que la aplicación va a ejecutarse en terminales relativamente poco potentes, se ha procurado optimizar al máximo los procesos y los recursos de cara a no agotar rápidamente la vida de la batería del dispositivo.

Como trabajo futuro, la empresa ha propuesto varias mejoras y funcionalidades para este proyecto. Entre estas se encuentra la posibilidad de descargar ficheros adjuntos a los partes de trabajo. Estos documentos podrían ser documentos legales como permisos o acreditaciones para poder llevar a cabo una tarea. Otra mejora propuesta es la posibilidad de hacer fotografías para adjuntar a un parte.

Finalmente, como aprendizaje personal, la realización este proyecto me ha proporcionado conocimientos sobre una tecnología actual y con un presente y futuro prometedor a nivel de cuota de mercado como es el desarrollo para la plataforma Android. Además, poner en práctica metodologías ágiles de desarrollo junto con un equipo de desarrolladores me ha permitido poner en práctica y en un entorno empresarial real mis conocimientos teóricos sobre la materia.

Bibliografía

[1] La máquina virtual dalvik. `http://androideity.com/2011/07/07/la-maquina-virtual-dalvik`. [Consulta: 1 de Octubre de 2014].

[2] Ableson, F.; Collins, C.; Sen, R. Android: Guía para desarrolladores.

[3] Freeman, E.; Freeman, E.; Bates, B.; Sierra, K. Head first: Design patterns.

[4] Jay A. Kreibich. Using sqlite.

[5] Roger S. Pressman. Software engineering: A practitioner's approach.