ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Ingeniería en Informática

ANDROID: GESTOR DE CONEXIÓN Y BATERÍA

Realizado por
DANIEL PAVÓN PÉREZ
(48898093R)

Dirigido por
Dr. JOSÉ RAMÓN PORTILLO FERNÁNDEZ

Departamento
MATEMÁTICA APLICADA I

Sevilla, (Septiembre de 2012)

Resumen

Los teléfonos móviles de hoy en día evolucionan cada vez más rápido. Tras la inclusión de pantallas táctiles, tecnologías inalámbricas de todo tipo, y sensores de posición e inclinación, no es de extrañar que su consumo en batería sea muy superior. Para controlar todos estos factores de una forma óptima, y no malgastar energía ya sea por inacción del usuario o por defectos de fábrica, este proyecto desea facilitar al usuario las herramientas que necesite para modificar todos estos parámetros, y así conseguir reducir su consumo de batería sin perder conectividad.

Índice general

Índice de cuadros

Índice de figuras

Índice de código

CAPÍTULO 1

Introducción

1.1– Contexto del proyecto

En la última década, el avance de los denominados *smartphones* ha sido imparable. Aquellos terminales móviles con la única, sencilla y evidente cualidad de realizar llamadas comienzan a ser parte del pasado. Gracias a nuevas optimizaciones en computación, miniaturización y refrigeración, el mercado de los móviles ha desarrollado exponencialmente su calidad, variedad y funcionalidad.

Figura 1.1: Evolución de la telefonía móvil.

Con un ritmo vertiginoso, hemos pasado de disponer de terminales con aparatosas antenas y tamaños incómodos a verdaderas joyas de la ingeniería, con multitud de capacidades y de utilidades en ordenadores que caben en la palma de una mano. Al disponer hoy en día tanto de botones como de pantallas táctiles, así como multitud de tecnologías inalámbricas

y de sensores funcionando en ocasiones al unísono, se ha hecho evidente la necesidad de crear robustos sistemas operativos que controlen su funcionamiento.

Como en todo mercado que se precie, son muchas las compañías que entran en pugna por conseguir el mayor número de usuarios. La posición de veteranía la ocupa Nokia, una marca inconfundiblemente identificada con los teléfonos móviles, que sin embargo, está palideciendo ante una competencia feroz de nuevos y completos sistemas operativos, en todo tipo de terminales.

Entre ellos destacan RIM y sus dispositivos Blackberry, que suelen disponer de teclado QWERTY en vez de pantallas táctiles; Apple con sus IPhone, muy apreciados y conocidos en todo el mundo; Google con su sistema operativo Android, en una gran variedad de terminales, tanto de baja como de alta gama; y el nuevo Windows Phone de Microsoft, que lucha por abrirse paso entre los anteriores.

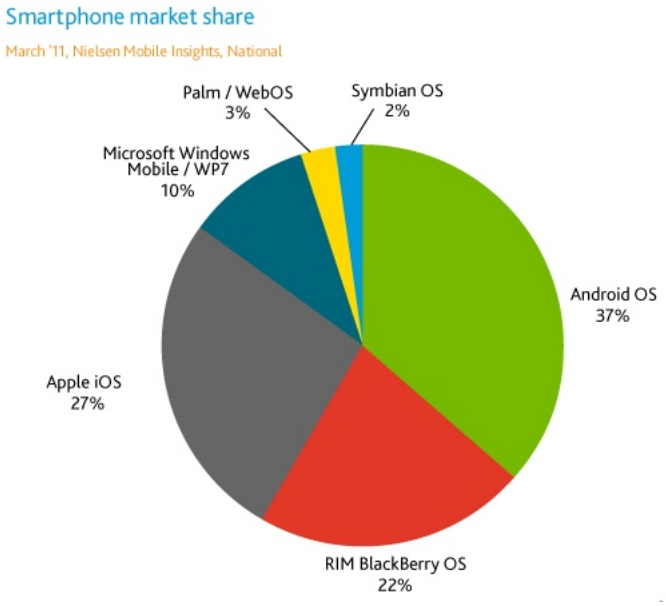

Figura 1.2: Ventas globales de smartphones en USA durante 2011 según NMI

En todos ellos es destacable la importancia que han comenzado a cobrar los sistemas operativos, los cuales deben presentar características de fiabilidad y velocidad ante el exigente escrutinio de los usuarios, sin perder funcionalidades por el camino. No es tarea fácil, pero actualmente destacan en tal hazaña dos sistemas operativos: iOS (Apple) y Android

(Google).

Ambos se pueden englobar en un tipo de terminal similar de ciertas características: una pantalla táctil que ocupa casi todo el teléfono, pocos o casi ningún botón físico, capacidades multimedia de vídeo y sonido, varios sensores y capacidades inalámbricas de todo tipo, con gran énfasis en el acceso a Internet mediante las mismas.

Cabe destacar que mientras iOS se encuentra únicamente en los terminales creados por Apple, de forma cerrada y renovándose cada cierto tiempo tanto en hardware como en software, Android apuesta por una solución de código abierto, utilizable por todo tipo de fabricante que esté interesado. De ésta forma, cuando Apple realiza una actualización de sistema o de hardware, las mejoras son prácticamente instantáneas y aplicables a casi todos sus modelos, debido a su misma arquitectura, mientras que Android se encuentra fragmentado en múltiples terminales, fabricantes y versiones.

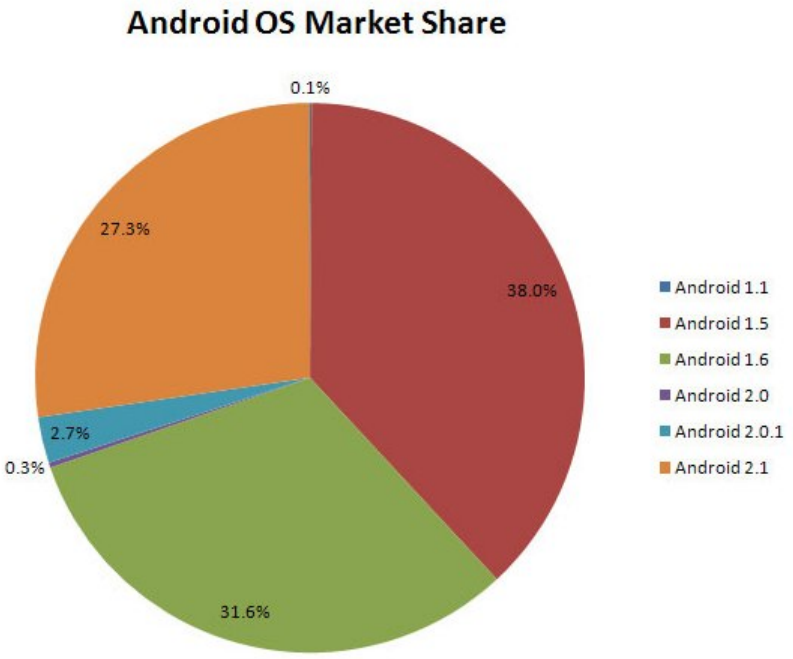

Figura 1.3: Fragmentación de Android a inicios de 2010

Esto, aunque a priori podría parecer una desventaja sin sentido, es una consecuencia de la destacable libertad que Google proporciona tanto a fabricantes, desarrolladores y usuarios. A los primeros les brinda co-

laboración y un sistema operativo adaptable y robusto; a los segundos, completa libertad para subir sus aplicaciones al *Android Market* (*Google Play*, actualmente) sin apenas coste; y a los últimos, la resultante variedad de terminales y aplicaciones.

En cualquier caso, son muchos los esfuerzos de Google por evitar tal cantidad de ralentizaciones en la cadena de actualizaciones. Durante éstos años cada operador de móvil y cada fabricante colocaban su versión modificada de Android y se desentendían de futuras actualizaciones, obligando de buen gusto a que los usuarios caigan en la espiral de obsolescencia programada y compren otro terminal, aunque sus necesidades se concentrasen en la falta de nuevo software.

Con el paso de los años, Android ha ido madurando y podemos observar cómo en mayor o menor medida van extendiéndose las nuevas actualizaciones de sistema, con un gran número de terminales que ya traen de serie, al menos, la versión Gingerbread 2.3.3 [1].

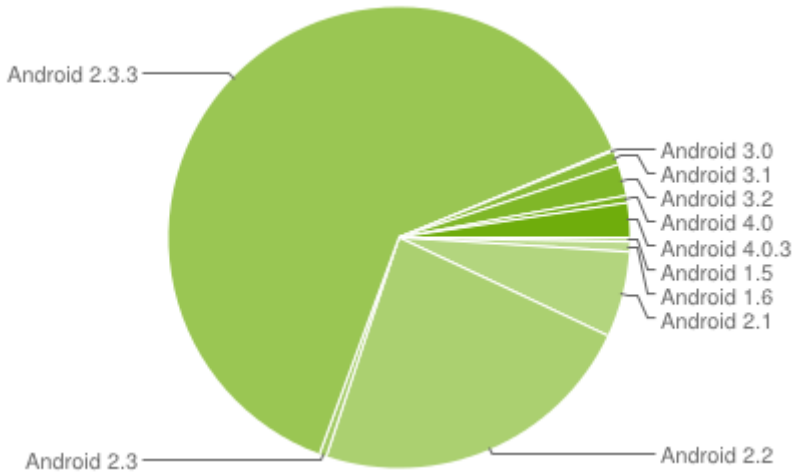

Figura 1.4: Versiones de Android a Abril de 2012

Por todo esto, en ocasiones recae en el mercado de aplicaciones la responsabilidad y la posibilidad de darle el control al usuario en multitud de aspectos, que por ahora, el sistema operativo no contempla. En el caso que nos ocupa, las conexiones inalámbricas y sus periodos de conexión varían de una versión a otra, sin un criterio homogéneo, resultando en distintas pautas de consumo de batería. Es en esos puntos donde radica el núcleo y la raíz de éste proyecto, como veremos a continuación.

1.2– Idea y causas

Al analizar las propiedades más identificativas de los *smartphones*, no dejan de dar la sensación de que todo son ventajas. Con unos sistemas operativos usables y universales, ya no sufrimos los típicos problemas con las opciones e interfaz de los antiguos móviles. Además, nos permiten estar conectados a internet e informados de las últimas novedades, y, por qué no, incluso nos permiten matar el tiempo con juegos muy funcionales y mejorados.

Pero por supuesto, siempre existe un problema. En éste caso: la batería, que se trata sin duda del mayor lastre que acarrean estos terminales. La explosión computacional que han recibido estos años no se ha visto acompañada de grandes avances en el desarollo de nuevas energías portátiles.

Por ello, es muy importante en el día a día de uso de nuestro teléfono cuidar siempre el gasto de la misma, casi tanto como su gasto en las diversas tarifas de datos. No en vano, cada terminal está terminantemente supeditado a la batería, ya que sin la misma se convertiría en un teléfono fijo, si es que aún tenemos la suerte de que se encienda.

Entonces, no sólo se hace evidente la necesidad de administrar tan valiosa energía durante el transcurso del día, sino también poder evitar apagados accidentales al llegar al temido cero por ciento, los cuales podrían causar un grave impacto en las futuras prestaciones de la batería.

Ante ésta necesidad, surge la idea de éste proyecto. Android controla con bastante buen hacer el uso de la batería. Regula el brillo de la pantalla, el tiempo de encendido o apagado de las redes inalámbricas, e incluso nos muestra un desglose con el gasto producido y sus causas.

Sin embargo, éstas opciones a veces pecan de escasas, o de fijas e inamovibles, según la versión de Android que estemos utilizando. Por ejemplo, en el caso de la versión Froyo 2.2, las redes Wi-Fi se desactivan cuando el usuario entra en estado ausente, para activarse de vez en cuando (en intervalos completamente desconocidos y arbitrarios), y en cambio, en GingerBread 2.3 no ocurre, y el receptor Wi-Fi seguirá activo indefinidamente.

La misión de la aplicación a desarrollar tiene la principal misión de proporcionarle al usuario el máximo control de éstos aspectos, forzando al sistema operativo a comportarse tal y como la persona decida, inde-

pendientemente de la versión de la que se disponga, e incluyendo nuevos aspectos de control allá donde sea necesario.

Con esta aplicación, se pretende entonces conseguir que el usuario consiga la mayor conectividad posible a internet minimizando siempre el gasto derivado en batería, y obtener una automatización de los procesos correspondientes según establezca el usuario. De ésta forma, aunque se desentienda del terminal, éste estará siempre disponible y actualizado para cuando el usuario vuelva y necesite información.

1.3– Estructura del documento

De aquí en adelante, el documento consta de cinco partes diferenciadas:

- En los primeros capítulos se definirán los objetivos clave que el proyecto debe cumplir, sus antecedentes y condiciones iniciales, y por último sus metas y cualidades principales a modo de introducción al proyecto en su conjunto.
- Análisis temporal o planificación: en ésta sección se detallará el procedimiento seguido para comenzar, preparar y desarrollar el proyecto en tiempo y en coste esperado. También se especifican las decisiones sobre formación tomadas con la intención de asimilar una buena base de conocimiento de la plataforma Android antes de comenzar las fases de diseño e implementación del proyecto real. En definitiva, se compone de un resumen general de los hitos a conseguir durante el desarrollo de los dos siguientes apartados.
- Proyecto formativo – *ToDo Bars*: en éste capítulo se explicará la decisión de comenzar con un proyecto formativo de menor envergadura para descubrir el funcionamiento interno de Android antes de comenzar con un proyecto real a mayor escala. Este procedimiento permite aislar los errores debidos al aprendizaje a una primera aplicación de prueba, eliminando la posibilidad de que comprometan la estabilidad final del proyecto. Se detallarán los conocimientos asimilados, así como los errores cometidos y sus causas y soluciones, a fin de evitarlos posteriormente y plasmar la experiencia acumulada.
- Proyecto final – *CommAndroid*: aquí se definirá el proyecto de fin de carrera desde su inicio. Una vez realizada la planificación, y habiendo asimilado los conocimientos necesarios para un buen devenir del mismo, se detallarán todos los pasos y decisiones tomadas en

las distintas fases de análisis de requisitos, diseño, implementación y pruebas.

- Tras las dos secciones principales encontraremos ciertos capítulos a modo de epílogo que tratarán diversas conclusiones obtenidas mediante el desarrollo del proyecto, así como comparaciones con otros productos similares.

Finalmente, se termina con un glosario de los términos utilizados en el documento, bibliografía y algunos anexos que podrían resultar de utilidad para el programador o usuario futuro.

CAPÍTULO 2

Definición de objetivos

Tras una ligera introducción al contexto de la aplicación y especialmente al Sistema Operativo que la va a contener, ha llegado el momento de definir con precisión y exactitud sobre qué trata este proyecto y qué pretende conseguir.

Desde un punto de vista general, el proceso creativo se concentrará principalmente en conseguir dos objetivos reconocibles: primero, cuidar el gasto de batería en los momentos de ausencia del usuario, y segundo, que ese ahorro no perjudique la conectividad del terminal.

Para ello, se dividirán los prerrequisitos del proyecto en tres objetivos concisos que se irán detallando durante todo el documento, especificando aquellas acciones y decisiones tomadas para conseguirlos. Estos tres objetivos principales son los siguientes:

- Minimizar el gasto en batería del terminal cuando el usuario no esté utilizándolo de forma activa, es decir, cuando el móvil se encuentre en estado de suspensión, esté o no su propietario cerca. Esto incluye cualquier tipo de redes inalámbricas, sensores, configuración o cualquier elemento que esté consumiendo batería sin confirmación o necesidad expresa del usuario.

- Maximizar la conectividad del dispositivo. Esto es, proporcionarle siempre que sea posible un flujo de datos actualizado, pero no continuo ya que queremos reducir el gasto de energía. De ésta forma,

cuando el usuario tome el móvil y lo haga volver de su estado de suspensión, dispondrá de datos actualizados de la última fracción de tiempo designada por ésta aplicación. Esta información podría considerarse obsoleta para el usuario, que podría necesitar datos en tiempo real, pero incluso los datos de la última conexión automática serán siempre más actuales que los disponibles de la última conexión manual. Según el periodo de conexión escogido por el usuario, éste podrá decidir si desea más frecuencia de actualizaciones o más ahorro de batería.

- Garantizar el control del usuario sobre todas éstas medidas de ahorro, de forma que pueda modificarlas a su antojo y decida según sus criterios cómo debe comportarse su sistema operativo y por extensión su terminal.

Al mismo tiempo, se puede observar la posibilidad de incluir otros objetivos que, si bien no son estrictamente necesarios, ayudarían en gran medida a mejorar la experiencia de usuario, su comodidad, y la utilidad de la aplicación. Según el tiempo de desarrollo y los detalles de implementación, estos añadidos podrían cumplirse de forma completa, parcial, o ser desechados.

- Gestionar las conexiones a redes inalámbricas con un criterio adecuado.
    - Controlar los accesos a redes Wi-Fi para que su fin último sea la conexión a Internet sin perder tiempo ni energía. Se conectarán a la mejor red entre las conocidas, aportando configuraciones adicionales si son necesarias. De no conseguir acceso a Internet, volverá a intentar con otra red Wi-Fi conocida. Si no encuentra alguna, desconectar el receptor y esperar.
    - Cuidar y limitar las conexiones de satélite *3G* debido a su variable coste tarifario.
    - Controlar o limitar dichas redes cuando el usuario se encuentre en movimiento, si el usuario lo desea.
- Informar al usuario sobre todos los datos relativos a sus conexiones inalámbricas y a su gasto de batería, con datos claros y detallados.
- Creación de modos de configuración ya predefinidos que ayuden al usuario a elegir un perfil acorde a su uso del terminal. Posibilitar también opciones sensibles a la localización del usuario. Si es necesario, simular dichas ubicaciones con la detección de redes Wi-Fi conocidas cercanas.

Entonces, quedan ya definidos los objetivos generales que la aplicación implementará en mayor o menor medida, intentando optimizar siempre el uso de la batería y la comodidad para el usuario final. Todas estas metas volverán a aparecer paulatinamente durante el documento, evaluando los logros a conseguir por el sistema.

CAPÍTULO 3

Análisis de antecedentes y aportación realizada

El contexto inicial del proyecto no deja de ser algo típico. Tras pasar mucho tiempo oyendo hablar de Android y de su apertura al desarrollo de terceros, consigo finalmente un terminal adecuado en coste y prestaciones. Poco tiempo después, tras probar algunas opciones de programación del sistema y usar sus capacidades multimedia, me asaltan algunas dudas:

- ¿Bajo qué criterios se suspenden y reactivan las redes inalámbricas mientras no se usa el terminal?
- ¿Puedo controlar cuando y cómo lo hace?
- ¿Cuánta batería se gasta, si se mantiene activo?
- ¿Existe alguna forma de definir conexiones cada cierto tiempo?

Tarde o temprano, cualquier usuario común llegaría a una situación parecida. Por mucho que los creadores del sistema se esfuercen en brindar múltiples posibilidades al usuario, siempre existen áreas en las que se quedan cortos. Por ello, se necesitan un buen número de versiones o de pruebas antes de conseguir contentar a todo el mundo. En ese tiempo, algunos usuarios podrían lamentar la falta de opciones. En el caso anterior, por ejemplo, se haría necesario más control sobre cuándo, cómo y con qué configuraciones se conecta su móvil a ciertas redes.

De ésta forma, una necesidad desde el rol de usuario se transforma en un posible proyecto como desarrollador.

Aún así, siempre es conveniente realizar una búsqueda de información al respecto antes de ponerse manos a la obra. Tras contrastar ciertos datos, se puede observar que muchas de las cuestiones propuestas anteriormente tienen ya solución para algunas versiones actuales de Android, pero las anteriores suelen quedarse sin arreglo. Es en ésas situaciones cuando las aplicaciones ajenas a Google pueden cargar con la responsabilidad de arreglar o cubrir éstos problemas.

Por ejemplo, el tratamiento automático de Android para ahorrar batería ha sido modificado en nuevas iteraciones, desconectando o no las redes inalámbricas si el usuario no está usando el terminal. Algunas versiones las dejan encendidas, y otras las suspenden temporalmente. [2]

También merecen mención las diferencias en el tratamiento de las IP estáticas. En versiones antiguas sólo se admite una configuración (obligando a cambiarla para dos casos distintos que la necesiten), pero en las nuevas versiones más allá de Ice Cream Sandwich (Android 4.0) parece haberse arreglado dicha eventualidad.

Necesidades como estas son las que ocasionan este proyecto, que poco a poco se van transformando en una idea sólida para crear una aplicación que resulte, sobre todo, útil.

En otros sistemas, existen ciertas trabas que impiden el acceso a nuevos programadores no establecidos, pero las características de Android son óptimas. Por ahora, es un sistema idóneo para la creación de aplicaciones sin apenas coste económico ni dificultades de desarrollo, ya que existe documentación bastante detallada por la red.

Además, la formación específica recibida en la Universidad de Sevilla proporciona una comodidad adicional: el lenguaje Java (usado en Android) es ampliamente conocido para cualquiera de sus titulados recientes de Ingeniería Informática. Y no sólo eso, ya que también repite su aparición el entorno de desarrollo Eclipse (si se desea usarlo), reduciendo de forma extrema la dificultad de instalación y adaptación.

Todas éstas peculiaridades forman un buen caldo de cultivo para la creación de éste y otros proyectos similares. Por ello, aunque el desarrollo en Android me es aún desconocido, la curva de aprendizaje y adaptación es mucho menor que en otros proyectos alternativos, convirtiéndolo en una opción óptima para la elección del Proyecto de Fin de Carrera e intereses futuros.

CAPÍTULO 4

Análisis temporal y costes de desarrollo

4.1– Planificación general

En todo proyecto que se precie, es necesaria una correcta planificación y gestión de los recursos y el tiempo. Con una adecuada estructuración, suelen ser mucho menos propensos a errores e imprevistos, y posiblemente serán terminados en los plazos y términos acordados. En cambio, una planificación errática o errónea puede causar no sólo retrasos contínuos y aumento de costes, sino incluso la cancelación del mismo por incumplimiento de contrato o insatisfacción del cliente, causando un grave perjuicio a ambas partes.

Durante el desarrollo de esta aplicación, no existe como tal un cliente que nos defina sus inquietudes y necesidades, con lo que pesa sobre el creador la definición de los requisitos con usuarios potenciales, mejorando la aplicación de forma iterativa con su ayuda.

Además, en un caso como éste, la planificación que se va a realizar no necesita gestionar ningún tipo de asignación de tareas al personal, al tratarse de un proyecto unipersonal. Entonces, el uso de diagramas específicos como *Gantt*, *Pert* o *Roy* se ven ligeramente mermados en su utilidad final.

No obstante, son buenos indicadores y esquemas para realizar una planificación temporal útil y adecuada, y se utilizarán en repetidas ocasiones en este capítulo. En las siguientes secciones del documento, se

realizará tanto una planificación temporal como de costes del conjunto completo del proyecto, y justo después, una recopilación y análisis de los resultados finales, para evaluar el desvío que pudiera existir.

De todas formas, éste proyecto dispone de una característica especial. Dada la falta de experiencia en el entorno Android como desarrollador, he decidido realizar un paso previo. Se trata de otro proyecto de menor envergadura que tenía en mente, que ha de servir como aprendizaje antes de dar paso al proyecto completo.

A efectos prácticos, se trata de tiempo utilizado en formación, y podría reflejarse como tal en los diagramas siguientes, pero para aportar mayor completitud a los datos, se desglosará cada sección. Con esta distribución de los datos se pretenden dos objetivos:

- Primero, mostrar la relación y antelación de las fases de desarrollo de diseño e implementación del primer proyecto con la fase de diseño del segundo. Así, queda evidente que lo aprendido será utilizado en el segundo, creando un proyecto mucho más optimizado y pensado para la plataforma. Con esta ventaja, obtendremos una aplicación más robusta, fácil de mantener y menos sensible a cambios.

- Segundo, poder documentar con detalle qué se esperaba aprender en un principio, y qué conocimientos se adquieren al finalizarlo. Esta información podría ser de especial relevancia para el lector si desea adentrarse en el desarrollo en Android, al tratarse de una interesante recopilación de los errores más habituales al comenzar y como evitarlos.

Con dos proyectos ya en mente, de aquí en adelante se irán mencionando uno y otro con sus respectivos nombres provisionales. El primero, que será usado como proyecto de formación, y cuyo funcionamiento no será detallado a nivel de código (pero sí su aprendizaje conseguido), se denomina *ToDoBars*. Es un gestor de tareas de los que podemos usar diariamente en nuestro correo u ordenador, e intenta simplificar y mostrar de forma gráfica el tiempo restante para realizar alguna acción o deber.

El segundo, es el proyecto presentado como Proyecto de Fin de Carrera, y su nombre en clave es *CommAndroid*. Su deber es el ya explicado en capítulos anteriores, dar más control al usuario sobre la gestión de las redes inalámbricas del terminal, con el objetivo de ahorrar batería sin perder conectividad.

4.2– Estimación de tiempo y coste

Para el análisis temporal de ambos proyectos usaremos dos herramientas gráficas bastante relacionadas entre sí, conocidas respectivamente como *Diagrama de Gantt* y *Método Pert*.

El primero se basa en una gráfica que muestra el avance y relación entre las tareas y el tiempo que necesitan. Se estructuran ambas métricas en dos ejes, y se le asigna un rectángulo a cada tarea con una longitud relativa a su duración. Además, el diagrama nos permite establecer vinculos entre cada uno de los eventos, formando una cadena en el que podemos observar cuales son los primeros hitos a conseguir antes de poder comenzar los siguientes, y cuales de ellos son críticos en la resolución del proyecto.

El segundo, se trata de una representación muy similar al diagrama de Gantt, y de hecho, están íntimamente relacionados ya que se pueden obtener el uno del otro. En el caso de *Pert*, obtenemos un grafo que relaciona todas estas tareas de la misma forma que Gantt, pero menos centrado en su aspecto temporal (en vez de usar un eje, cada tarea tiene sus marcas de tiempo) y más centrado en sus dependencias. Con éste gráfico se hace mucho más evidente el denominado *camino crítico* formado por aquellos eventos que bajo ningún concepto deben retrasarse, ya que perjudicarían la fecha final de entrega.

En nuestro caso, ambos resultan bastante útiles para la planificación y visualización temporal del proyecto. Al tratarse a priori de un desarrollo clásico en cascada (cada fase del proyecto desemboca en la siguiente), su representación en un el *diagrama de Gantt* es muy intuitiva y harían honor a dicho nombre con una sucesión descendente.

Sin embargo, aunque ambos desarrollos encajen en primer lugar con un modelo en cascada clásico, tienen también propiedades de proyectos de ciclo de vida iterativo, ya que siempre se buscará información con la que mejorarlo e intentar en todo momento implicar a los usuarios finales en el desarrollo y análisis. En estos casos, tras varias iteraciones, se consiguen añadir funcionalidades como en un ciclo de vida incremental, pero al mismo tiempo se modifican aspectos en ciertos casos de uso o interfaces gráficas que necesiten cambios para su correcta utilización por el usuario final.

Todas estas modificaciones continuas, algunas debidas a mejoras, otras a fallos en el análisis de requisitos, y otras a extensiones de la aplicación, son posibles causas de futuros retrasos en el devenir del proyecto.

Sea como sea el ciclo de vida del mismo, hay una serie de etapas (que se pueden consultar como capítulos en éste documento) en cada desarrollo software, independientemente de si se ejecutan una sola vez o en múltiples ocasiones reiterativas. Cada una de ellas tiene sus responsabilidades claramente delimitadas, y cada subsección en la que el proyecto podría dividirse suele poseer éstas fases:

- Análisis de requisitos: Es la primera fase de un proyecto y la más importante. Durante ésta etapa se deben documentar de la forma más precisa, completa y correcta los requisitos y necesidades del cliente. Cada aspecto del sistema debe quedar exhaustivamente redactado y carente de ambigüedad para evitar en lo posible cambios futuros.
- Diseño: Se trata del procedimiento encargado de planificar la estructura y funcionamiento interno del sistema. Se deben definir entre otras cosas, las estructuras de datos a utilizar, y la arquitectura general del proyecto, manteniendo siempre una cohesión (clases muy ligadas pero ligeras) y acoplamiento (clases poco ligadas pero pesadas) a niveles aceptables.
- Implemetación: Es el proceso real de construcción de la aplicación a nivel de código. Ya sea creando desde cero los sistemas, o mediante la unión de componentes de terceros, se seguirán los documentos de diseño a rajatabla, cuidando siempre la calidad y robustez de la aplicación.
- Pruebas: Es la etapa que evalúa la corrección del proyecto, no sólo en cuanto a su estabilidad frente a errores, si no también a su rendimiento y cercanía con los requisitos del cliente.

Este orden descendiente no es modificable, ya que en cada fase se estructura todo el conocimiento necesario para la etapa posterior, comenzando desde arriba con una necesidad del cliente, y terminando en la última con un producto estable y correcto con los requerimientos iniciales.

El proyecto a desarrollar comienza su andadura a finales octubre de 2011, tras la elección del departamento y tipo del Proyecto de Fin de Carrera, con lo que los sucesivos diagramas marcan tal fecha como inicio real, aunque no inmediata. Durante un mes, es necesario recopilar tanta información sobre Android y sobre los usuarios como sea posible, para poder entonces realizar una estimación más realista.

Por ello, la fecha de esta estimación se percibe en los gráficos como el momento en que ambos divergen. Es a partir de entonces cuando se debe

comparar este esquema con el producido al final del desarrollo, realizado con datos de tiempo reales.

Dicho esto, pasamos a la estimación inicial de los plazos de desarrollo de esta aplicación, y su subdivisión en varias fases dependientes entre sí, gracias al programa de código libre Gantt Project [3]:

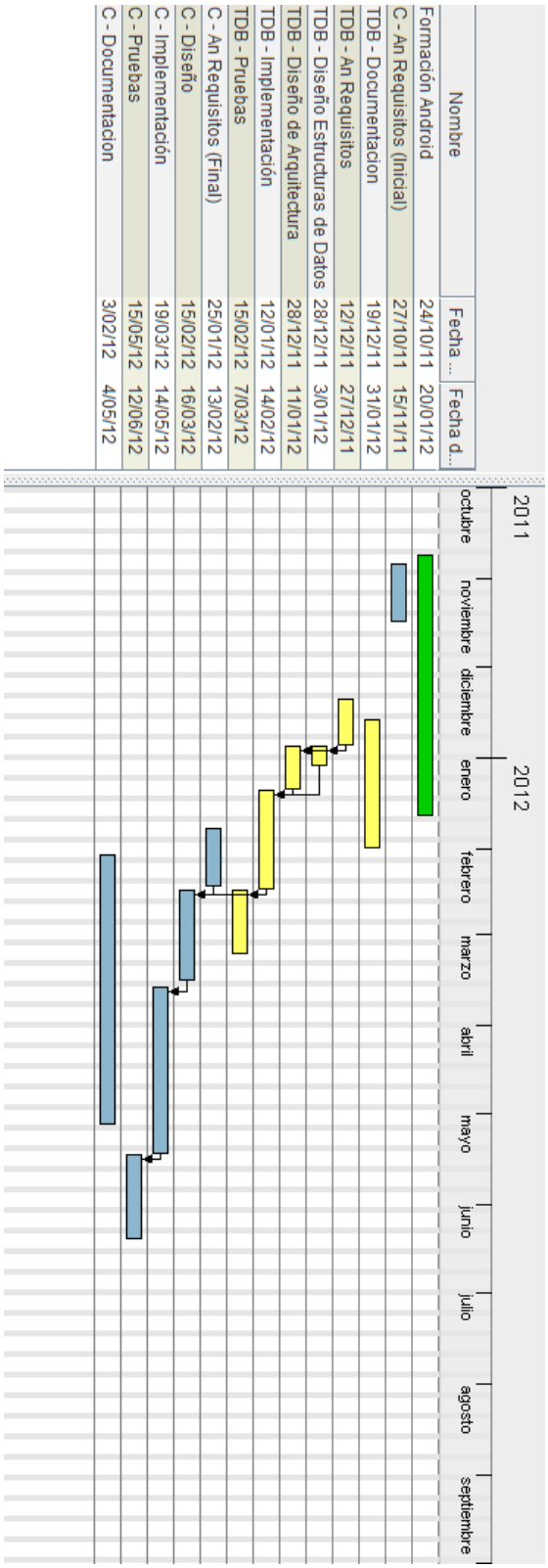

Figura 4.1: Estimación inicial. Esquema Gantt.

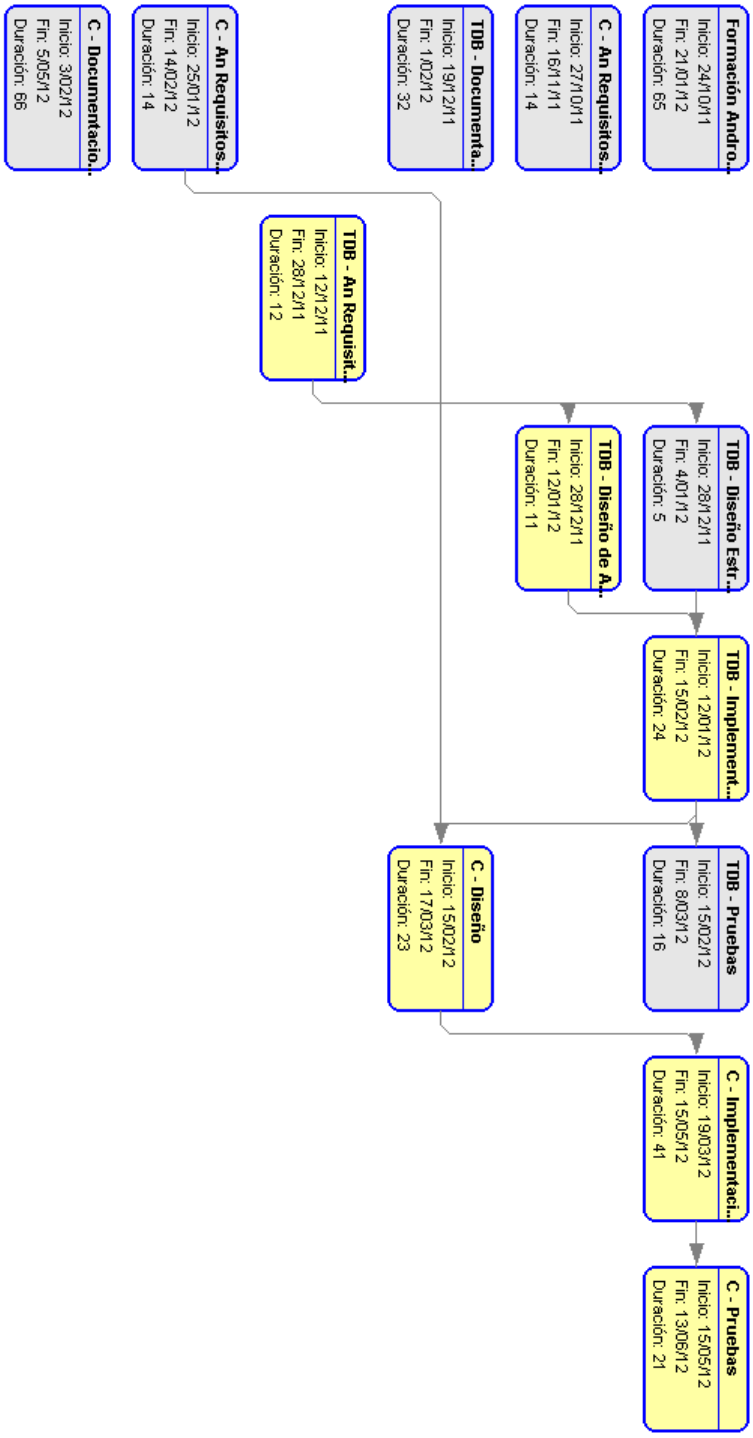

Figura 4.2: Estimación inicial. Esquema Pert.

A primera vista se puede observar la importancia de una buena formación sobre el sistema operativo Android durante gran parte del desarrollo. Se trata de la primera tarea a efectuar, antes incluso que esta estimación, con lo que al ser datos reales figurarán en ambos cronogramas.

Otra de las características del gráfico que salta a la vista es el desglose en dos proyectos distintos, como se explicó en la sección anterior. Una vez realizadas las encuestas a usuarios y las posibles detecciones de necesidades y problemas, para efectuar un diseño limpio, correcto y mantenible se hace necesario un mayor conocimiento de la plataforma.

Por ello aparece un proyecto de menor envergadura en el diagrama, de color amarillo. Sin él, el desarrollo del producto final se realizaría antes, y en apariencia este Proyecto de Fin de Carrera acabaría en un tiempo inferior. Sin embargo, no sólo obtendríamos un producto de menor calidad y más propenso a fallos a causa del aprendizaje, sino también unas fases de diseño e implementación más longevas y accidentadas.

Tras esta decisión de crear dos proyectos distintos, se estima que el desarrollo total de ambos acabaría aproximadamente por Junio, a tiempo para la entrega final en la convocatoria oficial de dicho mes. Según el diagrama, esta estructura supondría una duración total de unos 6 meses, con algunas fases solapadas que no deberían obstaculizarse entre sí.

En cuanto a la duración exacta en horas, el reglamento de un Proyecto de Fin de Carrera en la ETSII de la Universidad de Sevilla estipula una duración de 540 horas para un estudiante de Ingeniería Informática, con lo que deberá superar tal cantidad sin tampoco excederse de forma importante, ya que implicaría una planificación incorrecta.

La distribución de dichas horas debería ser aproximadamente de la siguiente forma:

Cuadro 4.1: Distribución en horas estimada del primer proyecto.

	Tiempo estimado (h)
TDB: Investigación	30:00:00
TDB: Análisis	25:00:00
TDB: Diseño	30:00:00
TDB: Implementación	40:00:00
TDB: Pruebas	15:00:00
Total:	**140:00:00**

Cuadro 4.2: Distribución en horas estimada del segundo proyecto.

	Tiempo estimado (h)
C: Investigación	50:00:00
C: Análisis	40:00:00
C: Diseño	85:00:00
C: Implementación	130:00:00
C: Pruebas	35:00:00
Total:	**340:00:00**

Cuadro 4.3: Estimación del tiempo total en horas.

	Tiempo estimado (h)
Proyecto1: ToDoBars	140:00:00
Proyecto2: Commandroid	340:00:00
Documentacion	60:00:00
Total:	540:00:00

Una vez tenemos una meta temporal, se pueden definir aproximadamente los costes del proyecto. En este caso el material no supone un coste relevante, ya que el entorno de desarrollo proporciona todas las herramientas y conocimiento necesario sin coste alguno. En todo caso, podrían añadirse como material los diversos ordenadores usados para el desarrollo así como los terminales de prueba, pero estos dispositivos fueron adquiridos con otras intenciones y por tanto no sería completamente riguroso incorporarlos. De hecho, si se tratara de un proyecto profesional, todo mobiliario o equipo suele estar incorporado en material de oficina y no es específico de un solo proyecto, excepto los terminales de prueba, si fuera estrictamente necesario adquirirlos.

Entonces, el coste del proyecto se reduce al más importante, el esfuerzo realizado por los trabajadores del mismo. En este caso el perfil del desarrollador único simplifica los cálculos, con lo que bastaría aplicar el coste en el mercado de ese perfil de trabajador. Para aproximar los resultados, podemos estimar al alza y a la baja el salario según el rol que ocupa dicho trabajador y consultar el término medio.

En el caso de un programador, el coste medio que le cuesta a la empresa puede rondar los 19 euros la hora, y un analista 23. Suponiendo que ambos trabajan durante un mes 40 horas a la semana, a 8 horas por día, obtenemos unos salarios que le suponen a la empresa unos 3000 y 3700 euros respectivamente. Eliminando impuestos, ambos cobrarían en cada paga 2000 y 2500 euros.

Así, si calculamos el esfuerzo económico en un punto intermedio, podemos establecer un coste por hora del trabajador a la empresa de unos 21 euros, cuyo trabajo sería distribuido a lo largo de unos tres meses para cumplir las 540 horas de este proyecto, con un resultado total de 11340 euros.

Este cálculo podría extenderse con otras métricas más profesionales como el modelo COCOMO, que realiza unas estimaciones iniciales en base a la cantidad de líneas de código que supondrá el proyecto. Sin embargo, en un caso como este, no poseemos datos específicos o de calidad para el modelo.

Se puede realizar una estimación sobre las líneas de código, pero estaría ampliamente influenciada por la inexperiencia en la plataforma Android. También existirían ciertas imprecisiones al definir cuál de los tres modelos de COCOMO es mejor para adaptar este proyecto. El modelo *semiacoplado* es para proyectos de mediana envergadura, el *empotrado* para sistemas muy restrictivos, y el *orgánico* para equipos pequeños y proyectos medianos o menores, pero de alta experiencia. Entonces, su uso no sería muy preciso, y al contar ya con una aproximación de las horas a realizar, el cálculo previo parece el más adecuado y directo.

4.3– Resultados finales

En esta sección se compararán los resultados obtenidos tras el desarrollo de ambos proyectos con las estimaciones iniciales de tiempo y coste. Por ello, se recomienda encarecidamente al lector que si desea una lectura en orden cronológico, continúe con los próximos capítulos relativos a cada proyecto y entonces vuelva a esta sección.

Tanto en esta sección como en el capítulo de conclusiones se tratarán diversas causas de posibles retrasos, adelantos o imprecisiones en las estimaciones, explicando variables del desarrollo que hayan influenciado en tales casos, con lo que puede adelantar acontecimientos al lector que aún no haya alcanzado capítulos posteriores.

Dicho esto, pasemos a comparar los datos de forma directa analizando el diagrama Gantt de reparto de tareas y horas durante estos meses:

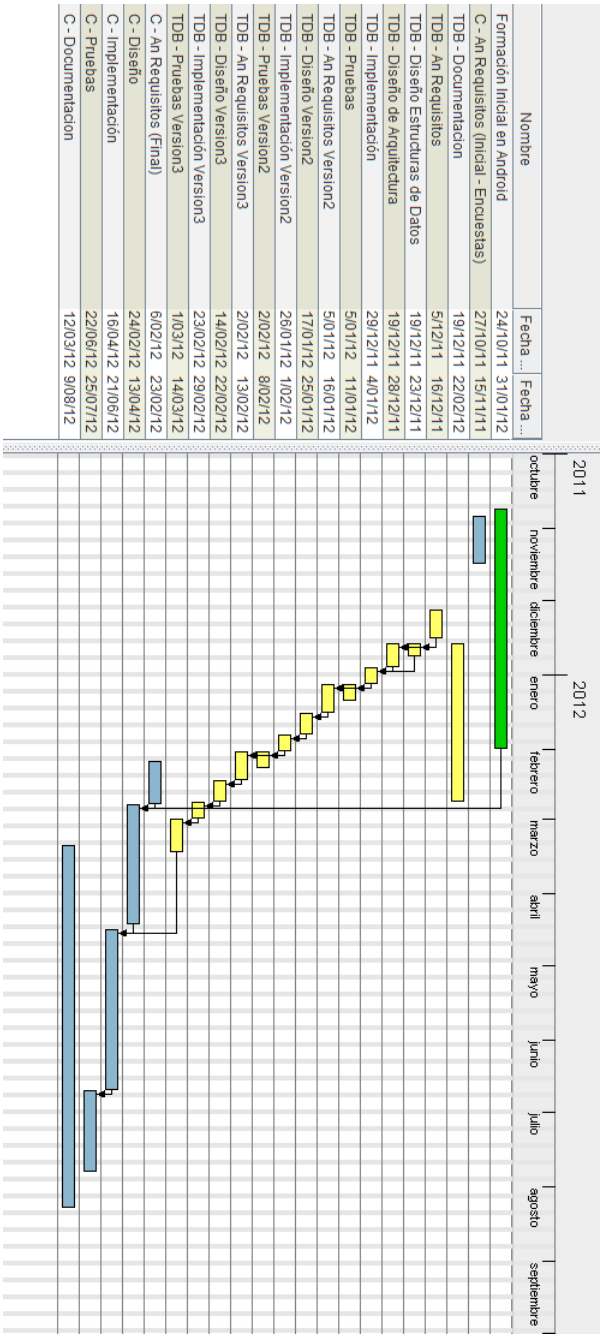

Figura 4.3: Resultados finales. Esquema Gantt

La primera anomalía con respecto al gráfico Gantt estimado, es la decisión de dividir el desarrollo del primer proyecto en varias iteraciones. Se puede observar la repetición de las distintas fases dependientes entre sí y cómo forman una *escalera* de pendiente bastante acusada que muestra una buena velocidad.

Este concepto se acerca ligeramente a las nuevas tendencias de desarrollos ágiles con mayor evaluación de los usuarios y el mercado, reduciendo al mínimo el periodo de creación. Algunos de ellos, como por ejemplo, el Método *Lean Startup* [4] [5] que aboga por minimizar el tiempo de desarrollo para aprender con mayor velocidad y de forma directa, usando esa experiencia para mejorar todo el proceso, así como también las metas que pretendemos conseguir con el producto y sus opciones en el mercado.

Para poder comparar mejor ambos gráficos, ya que no se aprecia ninguna diferencia de forma tan evidente, podemos superponerlos y observar los retrasos (o adelantos) producidos en el proceso.

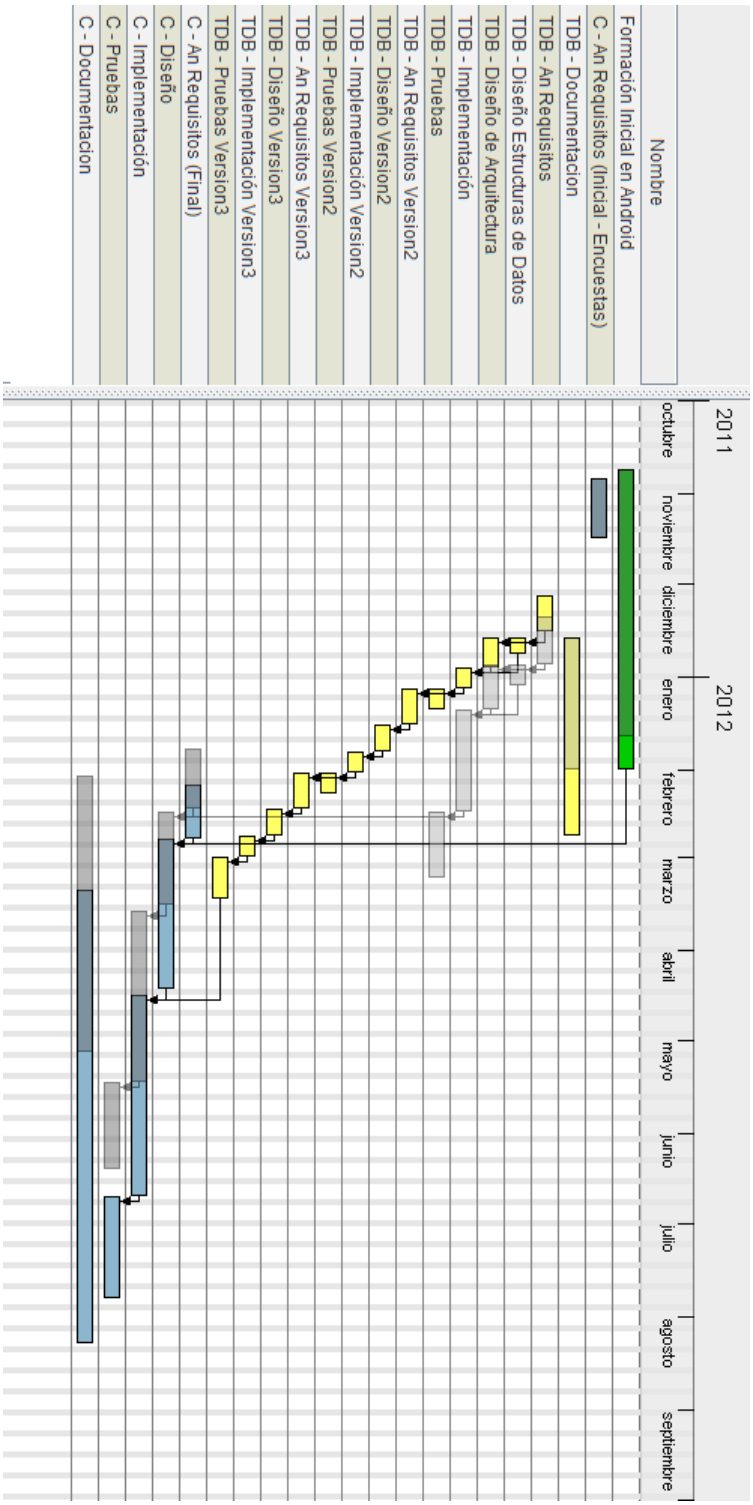

Figura 4.4: Comparativa temporal. Esquema Gantt

En éste gráfico se pueden observar mucho mejor las distinciones entre ambos. Aunque no existe diferencia durante noviembre y octubre, esto se debe a la realización a finales de noviembre de la estimación temporal. Es a partir de ahí cuando toca juzgar ambos gráficos.

Se ve con claridad el mayor ritmo de modificaciones que demuestra el primer proyecto formativo. Al tratarse de un producto fácilmente evaluable por los usuarios y basarse en un proceso de aprendizaje, la velocidad de mejoras aumenta. Durante todo su desarrollo se descubren nuevas y mejores formas de cumplir los requisitos y las necesidades de los clientes, y las fases de diseño e implementación mejoran en su eficacia.

Sus desventajas ya se mencionan durante el capítulo correspondiente, y es que a consecuencia de los cambios contínuos y de la inexperiencia inicial sobre la plataforma, el código resultante es mucho más difícil de mantener y mejorar, a pesar de experimentar un gran incremento en recursos y conocimiento. Sin embargo, estos problemas no se extenderán al segundo proyecto, que contará con unos inicios más sólidos y un diseño más estable, a costa de empezar con un ligero retraso, de dos semanas, según el gráfico.

Sobre este segundo desarrollo pesan varios inconvenientes que se pueden observar en el gráfico. El ligero desfase al iniciarse no supone problema, pero sí es incómodo en cuanto se suma a los retrasos producidos en cada etapa.

Entrando en detalle, el proceso de diseño, aún con el aprendizaje adquirido, supera ampliamente la duración estimada. Aunque se espera una mayor velocidad gracias al conocimiento de la plataforma, las necesidades de esta nueva aplicación son radicalmente distintas a las del primer proyecto. Entonces, a pesar de haber evitado retrasos mayores, existen ciertas variables que afectan negativamente a este proceso.

Muchas de ellas influyen también en la implementación, aunque no se ve de forma directa en el gráfico. Estas se tratan con detalle en el capítulo dedicado al segundo proyecto, pero en esencia son debidos a limitaciones de las herramientas de desarrollo (SDK) con respecto al control de redes inalámbricas, o comportamientos variables del sistema operativo y los drivers disponibles en el mercado. Así, los periodos de investigación necesarios se mantienen en el tiempo, ya que surgen nuevos problemas a pesar de un mejor manejo del sistema.

Todas estas variables deben documentarse en éste documento y por ello también se ve aumentado el periodo de las diversas modificaciones, pero no la cantidad de horas. Por ejemplo, durante todo el proceso de

creación de *Commandroid*, el desarrollo ha sido más lento debido a la necesaria investigación así como un cuidado extremo en cada paso del diseño, sopesando cada una de las opciones.

Entonces, aunque las barras de tiempo del diagrama de Gantt sean contínuas y uniformes, la distribución de horas durante esos periodos no lo son. Todas estas diferencias se apreciarían mejor si, en vez de mostrar únicamente las fechas, consultásemos también las distintas distribuciones de horas que estimadas con respecto a las finales.

Cuadro 4.4: Distribución final de horas dedicadas al primer proyecto.

	Tiempo estimado	**T. Final**	**Error relativo**
TDB: Investig.	30:00:00	35:00:00	16,67 %
TDB: Análisis	25:00:00	20:00:00	25,00 %
TDB: Diseño	30:00:00	25:30:00	17,65 %
TDB: Implem.	40:00:00	47:50:30	19,60 %
TDB: Pruebas	15:00:00	20:00:00	33,33 %
Total:	**140:00:00**	**148:20:30**	**5,96 %**

Cuadro 4.5: Distribución final de horas dedicadas al segundo proyecto.

	Tiempo estimado	**T. Final**	**Error relativo**
C: Investig.	50:00:00	45:00:00	11,11 %
C: Análisis	40:00:00	34:00:00	17,65 %
C: Diseño	85:00:00	93:20:00	9,80 %
C: Implem.	130:00:00	165:00:00	26,92 %
C: Pruebas	35:00:00	33:30:00	4,48 %
Total:	**340:00:00**	**370:50:00**	**9,07 %**

Cuadro 4.6: Distribución final de horas al completo.

	Tiempo estimado	**T. Final**	**Error relativo**
ToDoBars	140:00:00	148:20:30	5,96 %
Commandroid	340:00:00	370:50:00	9,07 %
Documentacion	60:00:00	55:00:00	9,09 %
Total:	**540:00:00**	**574:10:30**	**6,33 %**

Con esta nueva información, ya tenemos una mejor perspectiva para constrastar las diferencias entre la estimación y los resultados. Mientras que en el diagrama Gantt se podía observar que algunos periodos de tiempo aumentaban y otros permanecían estables, ahora estos últimos muestran una mayor actividad.

Algunas de las estimaciones se han quedado cortas, y otras, en cambio, han sido mayores de lo que finalmente ha sido necesario. En general, el error relativo medio de ambos proyectos ha sido del 22 por ciento para el primero y del 14 por ciento para el segundo.

El ejemplo más claro está en la implementación de *Commandroid*. Cada uno de los distintos dispositivos inalámbricos posee distintas configuraciones y gestiones según el terminal que usemos. Algunos de ellos poseen políticas de energía muy restrictivas y otros más leves, con lo que la aplicación ha tenido que lidiar con estas diferencias e intentar ofrecer una experiencia de usuario similar para todos. Al mismo tiempo, se ha necesitado un buen ritmo de investigación para cada una de estas cuestiones, y todo ahorro conseguido en este periodo por la formación previa, se consume con la búsqueda de información de tantas nuevas variables.

Esto, sumado al retraso inicial debido a la aplicación previa y a las diversas convocatorias de exámenes a realizar, provoca un desplazamiento sobre la fecha esperada, terminando el proyecto en Agosto, y no en Junio. Se trata entonces de un retraso de dos meses con respecto a lo deseado, y de unas 35 horas extra.

Por ello, el coste de realización de este proyecto al completo aumenta. Aunque el desarrollo de la primera aplicación podría haberse obviado de los cálculos, no deja de ser tiempo de formación que un profesional necesitaría antes de comenzar el proceso, con lo que en cierto modo es parte importante del proyecto al completo. Sobretodo si tenemos en cuenta la más que probable reutilización de código. Dicho esto, el coste ascendería esta vez a unos 12054 euros, que supone un aumento del 6 por ciento.

CAPÍTULO 5

Proyecto formativo: ToDoBars

Antes de entrar de lleno en el desarrollo del Proyecto de Fin de Carrera de nombre *CommAndroid*, decidí dedicar un tiempo prudencial en formarme todo lo posible sobre el diseño de aplicaciones para el sistema operativo Android.

Tras consultar algunas referencias de su página oficial para desarrolladores [6], diversas webs [22] [23], y dos libros dedicados a Android [7] [8], decidí poner en práctica dichos conocimientos creando alguna aplicación, que es la mejor forma de ponerlos a prueba y comprobar realmente cuánto se ha aprendido.

En vez de escribir código aleatoriamente y probar por separado cada una de las nuevas aptitudes aparentemente adquiridas, la mejor opción fue crear un proyecto personal de menor envergadura que el proyecto de fin de carrera para ir cometiendo los primeros errores y aprender de ellos.

Aunque a priori pueda parecer un consumo de tiempo importante en perjuicio del proyecto final, el aprendizaje es más rápido y el segundo proyecto se torna más sólido, evitando ciertos errores de bulto que convertirían cualquier aplicación en una muy difícil de mantener. Entonces, cada error que aparece y se corrige durante la creación del primer proyecto, representa una mejoría del segundo, no sólo en el código sino también en la fase de diseño. Así conseguiríamos prever los posibles problemas originados por tales decisiones y atajarlas mucho antes de que estos se produzcan.

La decisión estaba tomada, con lo que sólo era necesario escoger una buena idea para comenzar y llevarla a cabo. Durante mi breve uso de la plataforma, una de las primeras aplicaciones que deseaba instalarme para usar de forma diaria era un gestor de tareas. Aunque a veces no se recomienda su uso por depender exclusivamente de tales programas en vez de la memoria de cada persona, en mi caso prefiero desconfiar directamente de mi memoria y evitar así saltarme ningún evento de forma accidental.

Para cubrir esta necesidad existen multitud de aplicaciones en el *Android Market* o ahora llamado *Google Play*. Tras comprobar unas cuantas, me decanté por una con no demasiadas posibilidades pero simple y eficiente. En todas las que instalé, percibí dos estructuras muy parecidas. Una dividía la lista de tareas en varias categorías, que se podían ver con botones en la parte superior de la pantalla. La otra estructura, mostraba un balance semanal o diario con las fechas de inicio y fin de las tareas, más adecuado para personas con una agenda muy ocupada.

Curiosamente, en ninguna de los dos tipos de aplicaciones ni durante mi búsqueda fui capaz de encontrar una aplicación que mostrara de buena forma el tiempo restante de cada una. Colocar sólo la fecha me parecía lo lógico, y al mismo tiempo, me preguntaba si era posible mejorarlo.

Entonces recordé uno de los elementos gráficos de los que dispone Android (y prácticamente cualquier sistema operativo): la *ProgressBar* o barra de progreso.

Figura 5.1: Barra de progreso de Android

¿Y si representáramos el tiempo restante de forma gráfica? Si es un elemento tan útil y común, ¿por qué no aplicarlo a un gestor de tareas? Si la idea es mostrar el tiempo restante de la forma más clara o al menos visible, podríamos ignorar por un momento las cifras y colocarlo de forma gráfica.

Así es entonces como comienza la idea del proyecto denominado *ToDoBars*. En las siguientes secciones se detallaran aspectos de varias fases del desarrollo de la aplicación, de forma más resumida e iterativa que el proyecto principal, y otras dos secciones que son la clave de éste capítulo: qué se esperaba aprender, y qué se ha aprendido finalmente.

5.1– Objetivos del aprendizaje

El hecho de que la programación en Android se realice principalmente mediante el lenguaje Java trae consigo ciertas ventajas y desventajas. Como factor negativo, siempre podemos alegar que el lenguaje Java es más lento que muchos de sus congéneres, debido a su naturaleza como lenguaje interpretado.

Esa eventualidad se convierte también en una de sus ventajas, ya que gracias a la interpretación que realiza la Máquina Virtual de Java (JVM), el código realizado es fácilmente portable a cualquier plataforma que la posea. No es de extrañar, pues, que haya sido comúnmente utilizado en el mercado de telefonía móvil ya desde los Nokia más clásicos.

También hay que considerar otra ventaja. Y es que al contar con un lenguaje ampliamente conocido, ya posee grandes conjuntos de librerías y optimizaciones de las que carecería un lenguaje nuevo exclusivo para la plataforma.

Sin embargo, aunque tanto el entorno de desarrollo como el lenguaje sean familiares, el sistema operativo, como es comprensible, posee ciertas peculiaridades de vital importancia.

Una de estas primeras cosas que hay que conocer de Android, es el ciclo de vida de sus aplicaciones. Como se repite y una y otra vez durante todo este documento, el uso de cpu y por consiguiente, el gasto de batería de un terminal móvil es una de las cosas que más se deben cuidar. Por ello, el sistema operativo Android dispone absoluto control sobre qué aplicaciones están activas y cuales no.

Entonces, cuando el sistema se encuentre con limitaciones importantes, como baja batería, baja memoria disponible, o mucha cpu en uso, puede poner en descanso ciertos procesos, o en casos extremos, detenerlos. De hecho, si un proceso no responde en un determinado tiempo, Android le permite al usuario esperar o eliminarlo, ya que seguramente ha entrado en algún bucle infinito esperando una condición que podría no cumplirse.

Para que Android pueda utilizar estos recursos correctamente, o mejor dicho, para que nuestra aplicación no sufra de los designios aparentemente aleatorios del sistema, debemos implementar ciertos requisitos a nuestras aplicaciones. Así, cuando el sistema operativo desee pausarlas, eliminarlas, o simplemente cuando el usuario abandone nuestra aplicación, producirá una llamada a nuestra implementación de dicho ciclo de vida, lo

que nos permitiría guardar en disco todo tipo de información que pudiera perderse.

De esta manera, cuando el usuario vuelva a la aplicación no notará ningún perjuicio en sus datos o elecciones, y podrá reanudar su uso de manera normal. Este ciclo de vida se compone de varios estados:

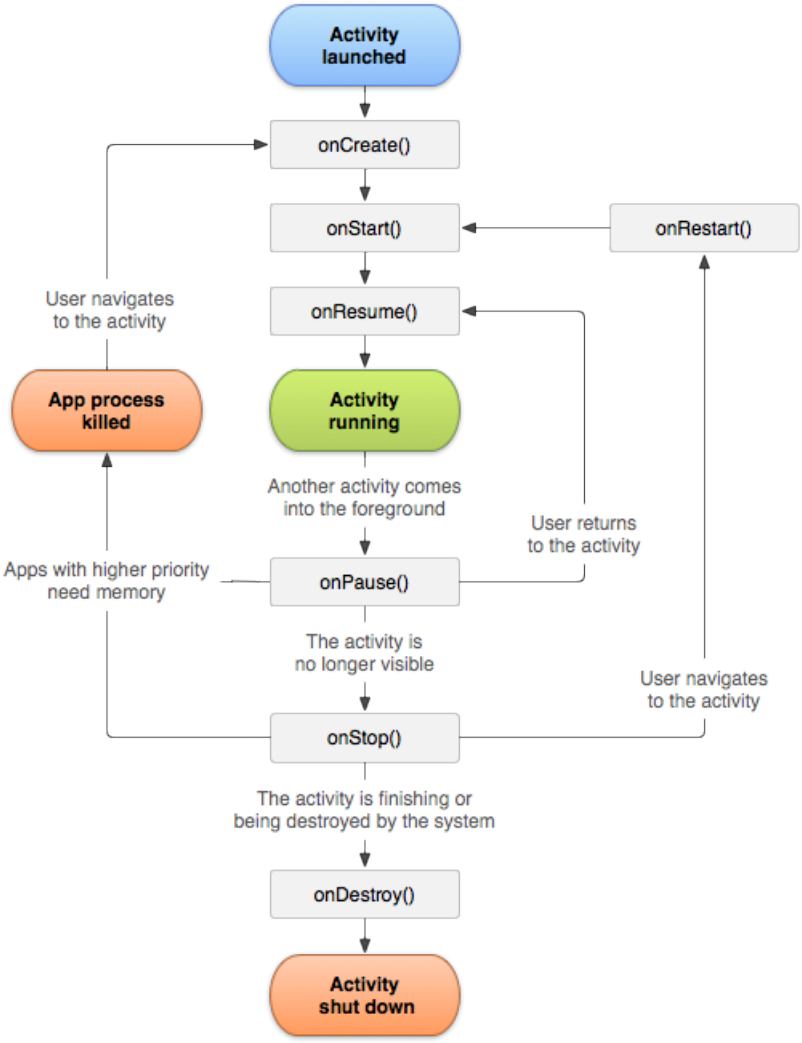

Figura 5.2: Ciclo de vida en Android

Dichos estados, de cara a la aplicación son realmente tres: pausada, detenida o en activo. Sin embargo, entre esos estados de larga duración ocurren una serie de cambios que se pueden ver en el gráfico 5.2. Es nuestra responsabilidad como desarrollador usar cada uno de ésos métodos

descritos (onPause(), onStop(), etc) para que el usuario no note cambio alguno cuando la aplicación se cierre y se vuelva a abrir.

Durante todas estas páginas, se ha hablado indistintamente de proyecto, aplicación o sistema. Sin embargo, en Android, cada aplicación debe dividirse en al menos, cuatro elementos bien definidos, de los cuales, los tres primeros son esenciales.

- *Activity*: Supone el punto de enlace entre el usuario y la aplicación, o lo que comúnmente se conoce como Interfaz de Usuario. Se encarga principalmente de recoger los eventos de pulsaciones del usuario y transformar la interfaz o cambiar de Activity cuando sea necesario. En ocasiones, para aplicaciones no muy complejas, sólo son necesarios estos elementos.
- *Service*: En el caso de aplicaciones más complejas, en cambio, podríamos necesitar un Service. Estos elementos no tienen interacción directa con la interfaz, y su área de trabajo es distinta. De hecho, tienen su propio ciclo de vida simplificado, estando activos o inactivos pero no en pausa. Se encargan de ejecutar código en un segundo plano complementándose con las Activity.
- *Intents*: Son paquetes de información que se envían constantemente en la plataforma. Algunos los genera el sistema, e informan del estado del terminal o de nuevos eventos. El resto pueden crearse desde cualquier aplicación, para establecer comunicaciones entre varios Activity y Service indistintamente. Han de recibirse con una clase especial denominada BroadcastReceiver.
- *Content Providers*: La menos necesaria en comparación con las anteriores, sirve para compartir información de una aplicación con otra externa, como por ejemplo para funciones de copiar y pegar, o de búsqueda personalizada.

Estos elementos son básicos en cualquier aplicación del sistema operativo Android, por lo que su creación y uso son competencias esenciales que se han de adquirir en éste primer proyecto de formación. Como mínimo, se deberá conseguir soltura manejando el ciclo de vida de la aplicación, y saber crear y mantener una Activity. Si este proyecto no lo necesitase, quedaría pendiente el aprendizaje y uso de Services e Intents para el segundo proyecto.

Además de este conocimiento esencial, existen otras dos aptitudes que se deberían de conseguir con éste proyecto previo. En cada Activity que se

ha mencionado, se gestiona una o varias interfaces de usuario. Para crear dichas interfaces, es necesario usar unos elementos especiales colocados en un documento de tipo XML.

Estos documentos tienen estructura de árbol entre sus elementos, por lo que cada uno de ellos dispone de un elemento padre (excepto la raíz del árbol) y/o de elementos hijos (excepto los últimos, que serían sus hojas).

Para diseñar tal interfaz, se debe conocer cómo interaccionan entre sí todos estos objetos y se reparten el espacio de la pantalla. Cada terminal dispone de resoluciones y pantallas de distinto tamaño, debido a la cantidad de móviles que poseen Android, pero existen herramientas para utilizar un tipo de métricas estándares que se vean igual en todos éstos dispositivos.

Entonces, se hace necesario el aprendizaje de los denominados *Layouts*. Existen multitud de ellos. Algunos permiten realizar un desplazamiento de una lista de elementos sin tener que implementar el gesto táctil correspondiente, otros colocan varios elementos en línea, otros permiten posicionarse entre sí al estilo de *HTML* con sus comandos float, etc.

Además, existen otros elementos en tiempo de ejecución que intervienen en la interfaz. Se trata de un tipo de menús especiales que proporciona Android, y puede ser de gran utilidad conocerlos y aprender su uso. Por ejemplo, existe un menú que se abre con el botón *MENU* del terminal, hay otro contextual que depende del área en el que se ha pulsado durante ciertos segundos, otros especiales para preguntar información al usuario (*Dialogs*) y también aquellos que alerten de posibles errores.

Finalmente, existe un elemento que suele pasar desapercibido pero es clave para la aplicación. Se trata del denominado *Manifest*. Este *manifiesto* hace las veces de *contrato* entre desarrollador, usuario y sistema operativo. En él, el creador estipula los accesos a información o controles privilegiados que su aplicación necesita. Con ésto, el usuario, cuando instale la aplicación conocerá qué permisos necesita y a qué datos sensibles necesita acceder.

Por la parte del sistema operativo, el desarrollador especifica qué Intents o paquetes de información del sistema capturará su aplicación, así como el nombre del componente interno que lo ha de recibir, para que Android le remita ésta información siempre que se disponga de los permisos adecuados.

De este primer proyecto, pues, se espera aprender todos éstos elemen-

tos y combinarlos para conseguir una aplicación útil y correcta, además de aprender el funcionamiento de cada uno de ellos, para evitar posibles antipatrones de diseño, arquitecturas erróneas, implementaciones absurdas, alta concentración de código, o en definitiva, cualquier tipo de eventualidad que perjudique la calidad del proyecto de fin de carrera.

5.2– Primera iteración

Al tratarse este desarrollo de un proceso más bien formativo, las siguientes secciones tratarán los aspectos de análisis de requisitos, diseño, implementación y pruebas a una menor escala que el proyecto real, para evitar así entorpecer la claridad de las explicaciones y de la documentación.

En cierto modo, en este apartado encontraremos la primera contradicción entre la planificación inicial y la creación de la aplicación, ya que el ciclo de vida del proyecto es distinto al planificado. Se dijo en aquella situación que se trataba de un ciclo de vida en cascada con leves toques de uno iterativo, sin embargo, como veremos en esta sección y en su propio título, será necesaria la aparición de iteraciones.

Con este nuevo tipo de desarrollo, las cuatro fases definidas anteriormente se repetirán en el tiempo una y otra vez. En cada paso, se mejorarán detalles de la aplicación gracias a la inclusión de usuarios en todo el proceso. No en vano, la importancia de la interfaz en un gestor de tareas es vital, ya que la más mínima incomodidad se perpetúa en el tiempo y erosiona la comodidad del usuario, y ya que éste tipo de aplicación requiere o debería prestarse a un uso sostenido, dichos problemas se magnifican.

Entonces, aunque la inclusión de usuarios finales ha reestructurado los planes de desarrollo iniciales, y posiblemente hayan propiciado un ligero retraso en el proyecto, existe una ventaja que se podrá comprobar de aquí en adelante. Desde los primeros prototipos de la interfaz a los últimos, se puede apreciar una mejoría extrema en las utilidades y comodidad de uso de la aplicación. Esto será vital no sólo para su posible comercialización, si no para extrapolar dichos conocimientos de uso de la interfaz al proyecto de fin de carrera *Commandroid*.

5.2.1. Análisis de Requisitos

La idea general del proyecto ha sido explicada durante capítulos anteriores. En definitiva, se trata de un gestor de tareas que representa de forma gráfica el tiempo restante para realizarlas. Para poder evaluar los requisitos de una forma óptima, se presentará la interfaz a posibles usuarios de la aplicación, ya sea con un prototipado en papel, o un prototipo no funcional para Android, y se recogerán las sugerencias sobre utilidades o mejoras que puedan surgir.

Como desarrollador, mi intención es cumplir con unos objetivos y requisitos iniciales para la aplicación (dejando de lado la formación). Estos son:

- Objetivo 1: Brindar la posibilidad al usuario de registrar y gestionar en el sistema tareas o eventos que necesita realizar y recordar.
- Objetivo 2: Usar recursos gráficos para que se visualice lo mejor posible el tiempo restante.
- Requisito de información 1: Cada tarea poseerá atributos esenciales como su nombre, fecha final, fecha inicial, descripcion, o algunos más optativos como su localización.
- Requisito de información 2: La lista de tareas debe poseer algún tipo de permanencia en memoria, que evite pérdidas de datos por cambios en su ciclo de vida. Además, se debe permitir al usuario editar, borrar, crear o reordenarlas según necesite.
- Requisito de interfaz: Diseñar una apariencia de lista para las tareas, que destaque de forma gráfica el tiempo restante disponible con mayor hincapié que la propia fecha de finalización, cumpliendo con el objetivo 2. Opcionalmente se mostrará con distinto estilo gráfico si se ha acabado el tiempo para realizar alguna de ellas.
- Requisito funcional 1: Implementar un sistema de avisos al usuario cuando quede menos de cierto tiempo para un evento, como por ejemplo, un día, mes o semana antes.
- Requisito funcional 2: Cuando llegue la fecha de finalización de una tarea, avisar al usuario de tal eventualidad, mostrando una animación mientras desaparece de la lista.
- Requisito funcional 3: Proporcionar un botón o acción para que el usuario marque como realizadas las tareas, incluso aquellas que ago-

ten su tiempo restante, para aquellos casos en que el usuario no tuviera el terminal cerca para marcarla como terminada.

- Requisito no funcional 1: Incluir un pequeño sistema de fidelización del usuario. Como si se tratara de un juego, la aplicación evaluará el porcentaje de éxito sobre el total de tareas conseguidas o *perdidas*.

- Requisito no funcional 2: Añadir soporte para idiomas. Como mínimo, inglés y español al principio.

Una vez aclarados los objetivos y requisitos básicos que la aplicación debe cumplir, es el momento de diseñar un prototipo muy básico de interfaz, para ir viendo cómo responden los usuarios e ir guiando la creación de la aplicación.

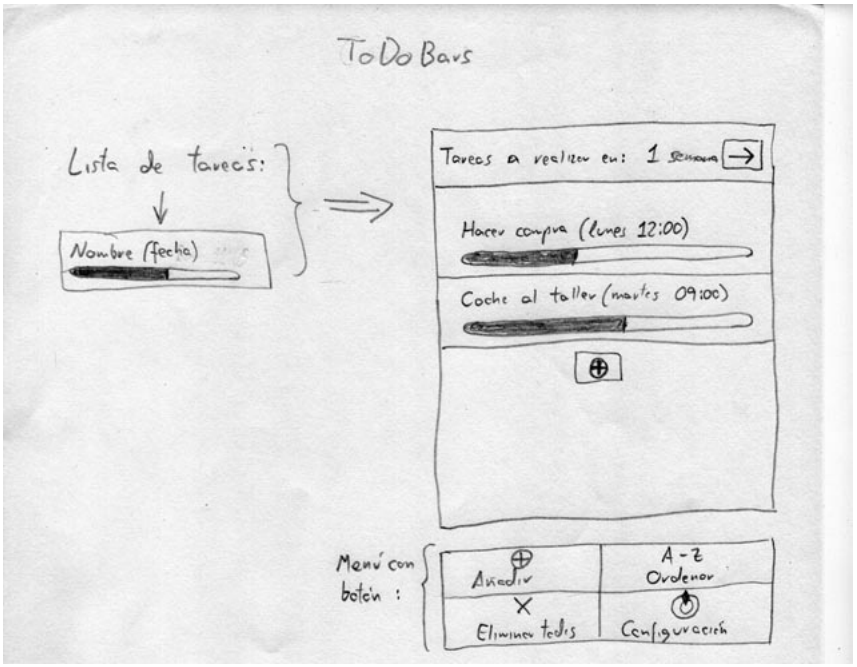

Figura 5.3: Primer prototipo de TodoBars

En la imagen se puede observar cómo cada una de las tareas requerirá de dos textos sobre una barra de progreso. Cada una de ellas formará una unidad completa, y entre todas, una lista de *cajas* que irán mostrandose de forma vertical al usuario. Para una correcta visualización, sería recomendable no utilizar botones de desplazamiento, sino permitir al usuario usar un gesto táctil para moverla arriba o abajo.

Además, se pueden ver dos nuevos elementos para la interfaz que implementarán ciertas opciones. La primera se trata de la sección superior, o selección del denominado *horizonte*. Cuando se definieron los objetivos se mencionó una unidad de tiempo: una semana, un día, o un mes, entre otros. Esa cantidad será usada a modo de referencia para calcular el desplazamiento de la barra de progreso. Si la barra está al máximo, se entiende que la tarea finaliza más tarde de ese lapso de tiempo. Si en cambio no está completa, se puede observar un porcentaje de 0 a 100 del tiempo que queda en referencia a esa marca de tiempo. Si estuviera a la mitad, por ejemplo, quedaría medio mes, medio día o media semana, según el caso.

Esta decisión implica volver a colocar las barras si el usuario modifica el *horizonte* con el botón de arriba a la derecha. Dicho botón cambiará las unidades de tiempo a otra posibilidad preestablecida.

El otro elemento a tener en cuenta es el menú inferior que aparecerá cuando el usuario pulse la tecla MENU de su terminal. En él reunimos las opciones más importantes de edición y consulta, como añadir una nueva tarea, ordenarlas o borrarlas. Como última opcion se accederá a un futuro menú de configuración si fuera necesario crearlo.

Con estos elementos básicos, y una Activity extra a modo de formulario para obtener los datos de una nueva tarea, podemos crear la primera versión de la aplicación. Será algo rudimentaria, pero mediante su evaluación con los usuarios podremos mejorarla paulatinamente con mejores resultados.

5.2.2. Diseño

Para conseguir unos resultados parecidos a los de la interfaz anterior, serán necesarias dos Activity distintas. La primera, que se llamará TaskActivity, se encargará de mostrar la lista de tareas, y la segunda: EditTaskActivity será llamada cuando haya que crear una nueva.

Además, hace falta una estructura de datos en la que almacenar las tareas. Por ahora, crearemos un objeto denominado Task que recoja la fecha de finalización y el nombre de la tarea. En sucesivas versiones, se aumentarán el número de atributos de esta clase para cubrir más posibilidades, pero no será así en esta primera versión.

Para gestionar estas tareas, almacenarlas y leerlas de la memoria, se

creará otra clase denominada TaskManager. Será una clase de tipo Singleton, es decir, que sólo podrá existir una instancia suya, para evitar duplicados innecesarios. En definitiva, hará de puente entre las peticiones de la interfaz en TaskActivity.

Con estos trazos podemos idear ya un primer diseño, concreto y sencillo. En la siguiente imagen que lo ilustra, se han evitado colocar los atributos menos relevantes de las clases, a fin de evitar complejidad innecesaria en la representación.

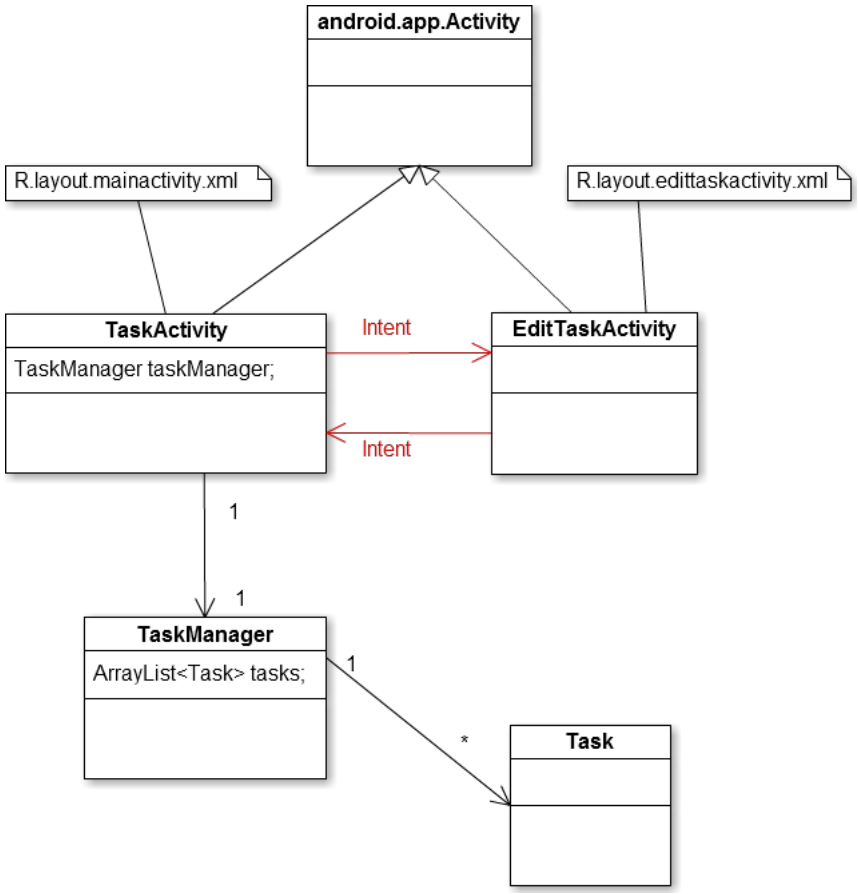

Figura 5.4: ToDoBars: Primer diseño

Usando aproximadamente un esquema similar, se procede entonces al desarrollo e implementación del primer prototipo. Durante la creación del código observaremos pequeñas diferencias al diseño, normalmente debidas a las clases que gestionan los botones de la interfaz, pero se irá entrando en detalle en las próximas secciones e iteraciones.

5.2.3. Implementación y pruebas

Ya tenemos el esqueleto básico de la aplicación diseñado. Se procede entonces a crear ambos Activity y de instanciar también la lista de tareas en TaskManager.

Para que la lista no se elimine o desaparezca una vez la aplicación pase a suspensión o sea cerrada, hemos de usar algún sistema de guardado permanente de archivos. Aunque en un primer momento se ha utilizado para probar el sistema de caché de Android, no es una idea recomendable, ya que se suele eliminar cada cierto tiempo o si el sistema está en condiciones adversas.

Existen multitud de opciones para almacenar esta información, ya sea usando bases de datos SQLite, o alguna conexión a Internet, pero se optará por una solución más sencilla: la lista de tareas será *serializada* con las librerías de Java y se pedirá a Android la ubicación y creación de un archivo en la memoria interna.

Sólo queda entonces preparar la Activity para que en cualquier situación de cierre o pausa se guarde la información mediante el método escogido, dejándole esta responsabilidad a la clase TaskManager. Además, TaskActivity (la principal) no sólo debe guardar los datos sino cargarlos (si los hay) cuando vuelva a estar activa.

Con éste sistema, y probándolo mediante opciones de depuración y con algunas pruebas unitarias para guardado de objetos de cualquier tipo, se observa que los métodos creados funcionan perfectamente.

Pasamos entonces a implementar la interfaz del programa, que ha de mostrar la lista de tareas y asignarles una zona táctil para realizar acciones con ellas, como borrarlas o moverlas. Para ello, inicialmente, y debido a la falta de experiencia, se itera una a una y se crea en tiempo de ejecución cada vista (View en el código de Android).

Durante la creación de éste paso se ve casi desde el principio lo improductivo que va a ser. Tener que recorrer en múltiples ocasiones la lista de forma manual, y definir un estilo de apariencia creando cada botón, cada barra y cada texto, y aplicar el estilo uno a uno, todo se convierte rápidamente en un problema.

Realmente funciona, pero estaba claro que ese código iba a ser muy difícil de mantener. Dudando de que toda aplicación que tenga una lista de

objetos necesite una implementación en tiempo de ejecución de la interfaz, procedo a buscar información en foros y diversos blogs que me acaban llevando a la solución.

Para realizar todo este proceso de creación de la interfaz gráfica, existe una opción muy cómoda y sencilla llamada *ArrayAdapter*. Heredando de esta clase se puede crear el mismo código de una forma mucho más óptima y mantenible, sobre todo si se utilizan recursos externos como XML para especificar el aspecto, y evitar hacerlo en código.

De esta forma, se consigue mostrar la lista de tareas, con dos textos y la barra de progreso de Android. A todas ellas, se le añade un sistema de gestión táctil, para que añada un *ContextMenu* o menú contextual. Este tipo de menús, se ejecutan cuando se pulsa una zona durante un tiempo, y proporciona algunas opciones relacionadas al elemento pulsado, que en nuestro caso permitirá borrarlo y moverlo.

Llegados a éste punto, y desarrollando rápidamente el sistema de horizonte para un mes, una semana, un día y seis meses, tenemos ya el primer prototipo:

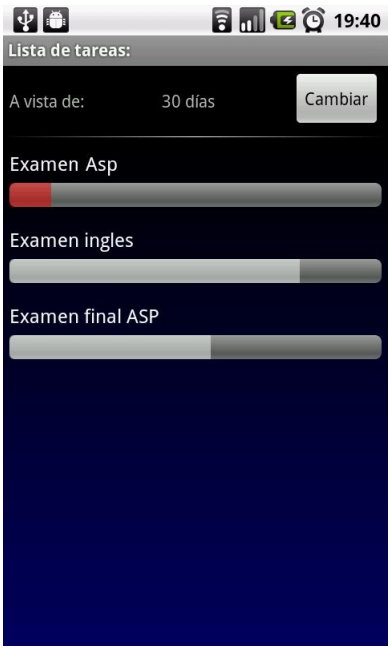

Figura 5.5: ToDoBars: Primera interfaz

Podemos observar en la imagen 5.5 el botón de selección de horizonte, y el estilo seguido para cada una de las barras de progreso, que por ahora

cambiarán de color según el tiempo restante.

Llegados a éste punto, conviene hacer un balance de lo conseguido hasta el momento:

Los objetivos 1 y 2 se han cumplido, ya que el usuario puede ahora gestionar las tareas, y su apariencia gráfica permite atisbar el tiempo restante. Los requisitos de información también presentan ya un estado avanzado. Aunque los atributos que identifican una tarea pueden ser motivo de discusión actual, pueden ser mejorados más adelante, estableciendo siempre algún tipo de retrocompatibilidad. Además, los datos ya se guardan y cargan correctamente, sin pérdidas.

El requisito de interfaz queda pendiente hasta tener un aspecto gráfico completamente definido. Por ahora, es capaz de representar el tiempo restante sin muchas trabas o conflictos.

Sin embargo, los requisitos no funcionales y los funcionales siguen aún sin realizar. De hecho, seguramente alguna de las sugerencias de los usuarios irán por esos caminos. No en vano, son ya necesarios los avisos, las notificaciones, y opciones para cancelar o completar tareas para usar la aplicación con normalidad.

El único requisito no funcional que sí se ha cumplido es el soporte para idiomas. Aunque no se aprecien en las capturas anteriores y sucesivas, desde un primer momento la aplicación soporta el ingles y el español. Esto será un problema en la segunda iteración, ya que al no poseer experiencia previa en Android, se ha realizado de una forma muy mejorable, como veremos más adelante.

En cualquier caso, teniendo ya este primer prototipo probado y funcional, procederemos a evaluarlo con los usuarios para anotar cualquier tipo de sugerencia sobre el estilo gráfico, la usabilidad de la aplicación, o nuevas opciones.

5.3– Segunda iteración

5.3.1. Análisis de requisitos y prototipado

Al volver a contactar con los futuros o posibles usuarios de la aplicación, echan en falta ciertos aspectos o mejoras:

- Más opciones para la selección de horizonte. Con un sólo botón se hace molesto seleccionar un tiempo en concreto.
- Mejores colores para la interfaz. Tanto para las actividades acabadas (en la primera version: letras en rojo, barra en negro), como el fondo.
- Sistema de alarmas y notificaciones, o habría que estar mirando la aplicación constantemente.
- Un listado de tareas ya completadas.
- Un listado de tareas futuras pero sin fecha concreta.
- Insertar un aviso antes de cualquier opción de eliminar tareas, para evitar pulsaciones accidentales.

Algunas de estas sugerencias ya fueron previstas durante la primera iteración, especialmente las relativas al aspecto gráfico de la interfaz, pero es especialmente positivo contar con puntos de vista adicionales.

Por ejemplo, durante el desarrollo se pasó por alto la necesidad de añadir confirmación antes de borrar datos. A priori es difícil pensar en otra barrera que retrasa las acciones del usuario, pero en este caso un sistema de protección contra pulsaciones no deseadas es en realidad una buena idea.

Otras de estas opciones, serán tenidas en cuenta pero por ahora no se implementarán, ya que podrían perjudicar o alterar las funciones normales de la aplicación. Es el caso del listado de tareas futuras, que por ahora se va a incluir en el diseño, pero aún no en el código, ya que está sujeta a cambios. Temporalmente se considerará como la cuarta *pestaña* de la aplicación, pero si surge alguna opción o utilidad nueva que haya que incluir, esa posición puede peligrar en favor de mejores candidatos.

De colocarse ahí finalmente, ha de ser de una forma clara y que no moleste en absoluto la visualización del resto de tareas, pero en realidad, casi todas tendrán una fecha límite, por lo que la utilidad de esa sección sería baja. Si no tienen fecha, siempre tendríamos la opción de colocarlas con fecha exageradamente lejana. Sea como sea, y se use o no, la sugerencia queda registrada por ahora.

El resto de peticiones son todas abarcables o estaban ya planificadas, con lo que las modificaciones en la fase de análisis de requisitos inicial son mínimas. La mayoría de estos nuevos aspectos formarán parte importante

sólo de las fases de diseño e implementación, pudiendo significar que la primera fase del proyecto se ha realizado con un buen nivel de corrección.

Entonces, sólo queda orientar el aspecto de la interfaz con un nuevo prototipo en papel, y observar si los usuarios aceptarían tales cambios, incluso antes de programarlos:

Figura 5.6: Segundo prototipo de TodoBars

En este nuevo diseño de la imagen 5.6, se ha optado por colocar ya cuatro pestañas o secciones diferenciadas según su objetivo (su orden pueden variar según la versión):

- La primera mostrará un resumen general de la primera tarea de la lista o la que se seleccione como *tarea actual*. Tendrá los botones necesarios para cancelarla y marcarla como realizada. Como sugerencia para futuras versiones, podríamos colocar tanto geolocalización de tareas como botones para cambiar la actual.

- La segunda pestaña se trata del historial. Aquí se moverán aquellas tareas que bien porque el usuario las haya marcado como completas, o bien porque haya pasado su tiempo, ya no necesitan estar en la lista principal. De forma opcional se podría mostrar un porcentaje de las completadas con respecto al total.

- En tercera posición colocaremos la lista principal de tareas. Si es posible se les incorporarán iconos que muestren la importancia del evento y que al pulsarlos los marque como realizados. La selección de horizonte ahora dispondrá de dos botones para avanzar y retroceder, y estará en la parte baja de la pantalla.

- La cuarta aún está por definir, pero en el prototipo alberga una posible lista de tareas con fecha indeterminada. Si surge una idea o utilidad mejor para esta pestaña, será modificada.

Con estas funciones ya tenemos cubierto el diseño de la interfaz. De forma obligatoria se implementarán las dos más importantes: la lista principal, y el historial. En cambio, la pestaña general y la de eventos futuros se dejarán implementadas de forma mínima o vacías hasta que estén correctamente definidas y evaluadas todas las posibilidades con los usuarios.

5.3.2. Diseño

Lo primero que hemos de aplicar en el diseño es precisamente el cambio más evidente que se ha realizado en la interfaz: las pestañas.

A efectos prácticos son 4 zonas que deberán reconocer las pulsaciones de la pantalla táctil que realice el usuario sobre ellas y cambiar a la sección correspondiente.

En el diseño e implementación que necesitan para ser creadas, se dispone de cierta libertad. Android nos otorga varios tipos de interfaces para contener esas cuatro vistas, ya sea *TabHost*, *Viewflipper* con sus posibles animaciones, o el reciente *ViewPager* de las nuevas versiones de Android, entre otros.

La implementación es bastante distinta según escojamos uno u otro, y en cierto modo también el diseño, con lo que es importante tener una decisión clara desde el primer momento.

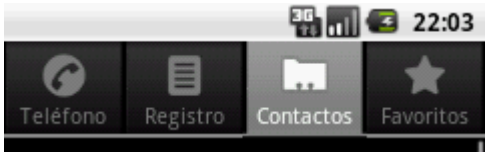

Figura 5.7: Apariencia de TabHost

TabHost parece la opción más propicia y sencilla, pero tras estudiarla detenidamente durante un tiempo, la gestión de las pestañas no coincide con mis intereses. El manejo a nivel de código no es cómodo, y en ocasiones la mejor opción para usar la clase pasa por crear cuatro clases Activity, una por cada pestaña. Esto tendría serias repercusiones en el código ya creado y también complicaría la gestión del ciclo de vida de la aplicación, con lo que ha sido descartada, a falta de más información o experiencia.

ViewPager es demasiado nueva, y para incorporar compatibilidad con versiones antiguas de Android, prácticamente sería necesario tomar el código de la página oficial e incluirlo en el proyecto. No parece una opción cómoda, ya que la integración de código externo puede acarrear problemas que no compensan, aunque mejore en apariencia.

En cambio, Viewflipper parece bastante versátil. Su control parece el más sencillo, y no requiere de una arquitectura muy compleja ni de cambios drásticos de diseño, e incluso se pueden colocar animaciones gráficas de transición.

También se puede alternar fácilmente entre las diversas vistas, ya sea pasándolas una a una de forma iterativa (como en ciertas aplicaciones con movimientos táctiles), o colocando directamente una vista según su posición. En este caso, la segunda opción será muy útil para detectar la pestaña pulsada y aplicar los cambios.

Aún así, no todo son ventajas, ya que realmente no dispone de una implementación de las pestañas típicas. Sin embargo, esto nos brinda la opción de crearlas manualmente, con el estilo personalizado que nos apetezca, y asignarles la tarea de cambiar la vista del ViewFlipper.

En cualquier caso, para todas esas opciones, conviene distribuir las hojas de estilo XML en distintos archivos. Como resultado, tendremos las cuatro pestañas distribuidas en cuatro archivos distintos que definirán su interfaz, y una principal que tenga las imágenes o textos de las pestañas, y una referencia a cada uno de esos archivos.

De esta forma, si fuera necesario hacer alguna modificación, sólo habría que buscar la vista correspondiente y realizar el cambio únicamente en ese archivo, dificultando la propagación de errores.

Dos de esas pestañas son las que reciben la mayor atención en el diseño y en la implementación. Una de ellas ya está creada, y muestra la lista general de tareas. En la otra se enfocaran los nuevos esfuerzos, volviendo a crear una lista de tareas en TaskManager y creando otro ArrayAdapter

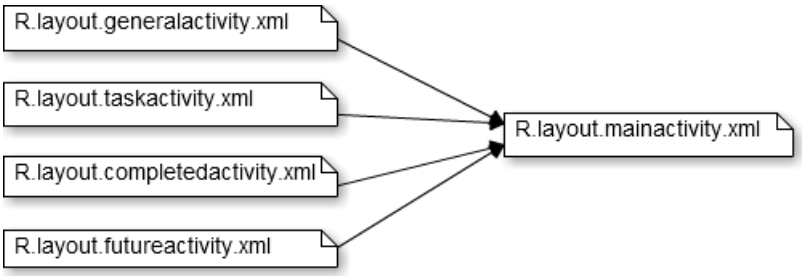

Figura 5.8: Distribución de los archivos Layout

personalizado que se encargue de mostrarla. La nueva lista de tareas, entonces, se irá rellenando con aquellas que acaben el tiempo restante o que sean completadas por el usuario.

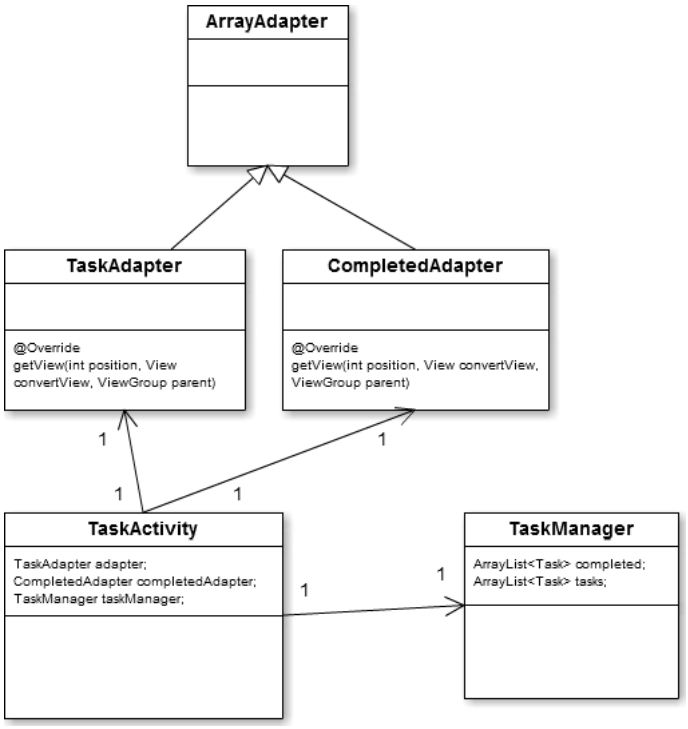

Figura 5.9: Diseño lógico de las listas de tareas

Este nuevo ArrayAdapter se encargará de mostrar en pantalla ciertas tareas con un estilo diferente (definido en un XML). Estos eventos serán los completados o fallidos, es decir, los que conforman la pestaña del historial. Para implementarla, sólo es necesario heredar de la clase ArrayAdapter y redefinir el método getView. En él, se puede usar el XML mencionado o

crear las vistas con código normal.

Sólo restaría aplicarles los menús contextuales y ajustar la Activity principal TaskActivity para que los implemente y le de sentido a las diversas opciones y al menú principal del botón MENU.

Dejando ya de lado el aspecto gráfico, los requisitos y entrevistas con los usuarios evidencian otra necesidad clara: hay que crear un sistema de alarmas y notificaciones.

Para crearla, necesitaremos uno de los conocimientos que se espera obtener de cara a realizar el proyecto *CommAndroid* más tarde, ser capaces de manejar Intents. En este caso, primero tenemos que hacer uso de la clase de Android llamada AlarmManager, que nos permite indicarle al sistema que nos avise cuando se llegue a una hora determinada.

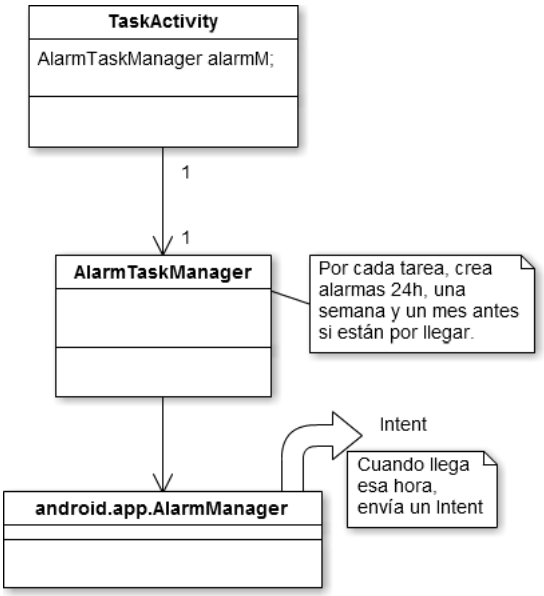

Figura 5.10: Sistema de alarmas. Preparación.

Como muestra la imagen 5.10, una vez se alcanza dicha hora, Android lanza un Intent concreto que le hayamos indicado. Ese paquete de información ya tiene preparado un receptor, *BroadcastNotifier* que debe ser clase hija de BroadcastReceiver para recibirlo, y ser declarada en el archivo *manifest*.

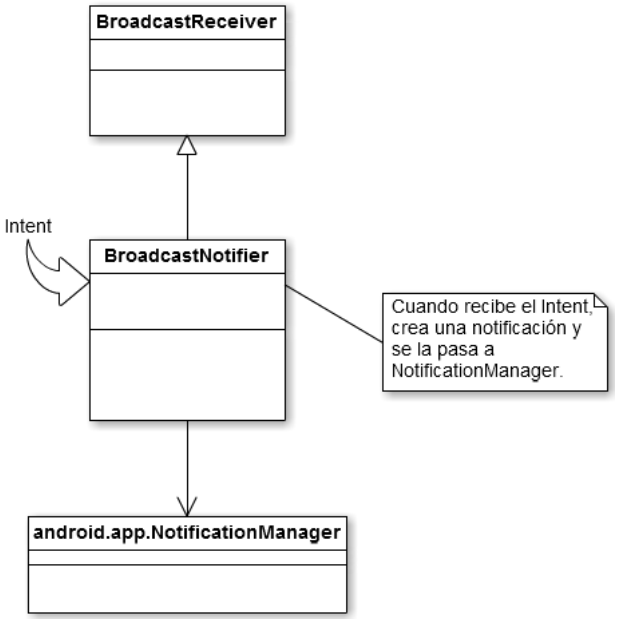

Figura 5.11: Sistema de alarmas. Notificación.

Dentro del Intent se incorporan datos para reconocer qué tarea es la que ha alcanzado la correspondiente marca de tiempo. Con esa información, generamos una notificación típica de Android y se la pasamos al encargado de mostrarla: NotificationManager.

Hay que tener cuidado con estas alarmas. Si el terminal se apaga, Android no parece recordarlas, con lo que es nuestra responsabilidad volverlas a colocar en activo en cuanto sea posible. Por ejemplo, podría implementarse una solución una estrategia de Intents similar, pero capturando el que lanza Android justo al arrancar.

Con los cambios más drásticos pasados por la fase de diseño, sólo queda entonces llevarlos a la práctica.

5.3.3. Implementacion y pruebas

Esta vez, en la fase de implementación cae una gran responsabilidad. No sólo hay que acatar lo decidido durante la etapa de diseño, añadiendo una nueva lista de tareas y un sistema de notificaciones o alarmas, sino también mejorar el aspecto y el código de la aplicación.

De hecho, nada más empezarla, se descubre un fallo importante de la primera versión. Aunque desde un primer momento se ha tenido en cuenta el soporte para idiomas, se implementó de forma algo rústica, por decirlo de alguna manera.

Y es que con un poco de investigación es fácil descubrir que Android ofrece soporte para idiomas de forma casi automática. Esto quiere decir, que sólo es necesario colocar las cadenas de texto de forma organizada en unos archivos XML en una carpeta concreta del proyecto. De esta forma, el sistema operativo reconoce su configuracion de lenguaje, y busca en la aplicación si existe. De no existir, carga el archivo básico con el idioma estándar que posea la aplicación.

Sin embargo, durante la primera versión todo esto se ha realizado de forma manual. La propia aplicación preguntaba al SO el lenguaje seleccionado y cambiaba las cadenas de texto a mostrar. Al principio era un metodo viable y no se hacía muy costoso, pero en cuanto la aplicación aumenta de tamaño es imposible mantener ese código de forma óptima.

Gracias a esta experiencia, en futuros proyectos no volverá a ocurrir tal decisión. El proyecto de fin de carrera *CommAndroid* comenzará con los conocimientos que se están adquiriendo en este capítulo, y como mínimo se dispondrá de cierto aprendizaje sobre Intents, Activities, y otros recursos como soporte para idiomas o interfaces gráficas.

Existe también otro detalle que es necesario mejorar. Android tiene una peculiaridad en el manejo de las aplicaciones que afecta al ciclo de vida de las mismas, y es fácil pasarla por alto. Cuando el usuario cambia la orientación del terminal de vertical a horizontal o viceversa, el sistema operativo destruye sorprendentemente la aplicación y la vuelve a crear con la nueva estructura.

Si hemos cuidado la gestión de los datos y su almacenaje, no debería existir ningún problema con esa información permanente. Sin embargo, los formularios que hayan sido rellenados y las opciones de visualización (como la selección de horizonte) sí se pueden perder, con lo que el usuario notaría cambios si mueve el terminal para usarlo de forma distinta.

Se trata de un detalle a evitar, en la medida de lo posible. Para ello, tras algo de investigación, se descubre que al destruirse o regenerarse, la aplicación recibe un *Bundle*. Estos paquetes de información, que suelen ir bajo al amparo de un Intent, nos permiten traspasar datos sencillos (o complejos pero bien especificados) entre aplicaciones. En este caso, la aplicación al destruirse generará un Bundle con esa información, y al volver

a ser creada, Android se lo devuelve para que se regenere correctamente.

Durante esta fase de generación de código, también se hace vital la reestructuración de la interfaz. El modelo de la primera versión era algo arcaico y simple, creado para poder realizar la primera iteración y evaluación con los usuarios tan pronto como fuera posible. Por ello, recae en esta segunda versión la responsabilidad de mejorarla y hacerla lo más usable posible al mismo tiempo que añade muchas nuevas opciones.

En todo este proceso, se descubren nuevas estructuras para las interfaces (normalmente definidas en archivos XML). Entre ellas destaca *RelativeLayout*. Este elemento nos permite distribuir objetos de la interfaz gráfica de una forma determinada y controlada por la pantalla. Sus propiedades nos permiten establecer interrrelaciones entre los diversos elementos, para que no exista duda ninguna sobre cómo se reparten el espacio disponible. Por ejemplo, podemos definir que un icono este bajo, sobre, a la izquierda, o derecha de algo, así como su alineación con otros elementos, o incluso si debe estar pegado a algún borde de la pantalla.

Con esta estructura, se mejora sustancialmente la apariencia de la interfaz. Antes se había abusado del elemento *LinearLayout*, y aunque fue útil para mostrar listas de forma rápida y repartir de forma igualitaria el espacio de la interfaz, tiene importantes desventajas, y palidece ante RelativeLayout en este tipo de ocasiones.

Gracias a estos descubrimientos y a la programación realizada, surge una nueva interfaz, que a pesar de tener bastante definidas sus funciones, aún necesitaría un poco de estilo y cuidado artístico, más difícil y lento de conseguir.

En la interfaz de la figura 5.12 se observan ya algunos de los cambios mencionados anteriormente. Aparece por ejemplo la casilla para marcar tareas como completadas, que las envía al historial con una animación, y representa su importancia con una forma o color.

La nueva pestaña destaca como novedad por sí misma, e incluye también un pequeño texto que nos indica el porcentaje de eventos realizados, para intentar atraer con más fuerza al usuario hacia la aplicación, incitándole a mejorar los resultados (haciendo trampas o no, es su decisión).

También se ha procedido a implementar el sistema de notificaciones tal y como fue diseñado en la fase anterior. No ha supuesto muchos problemas y parece funcionar a la perfección, excepto el detalle que se mencionó sobre

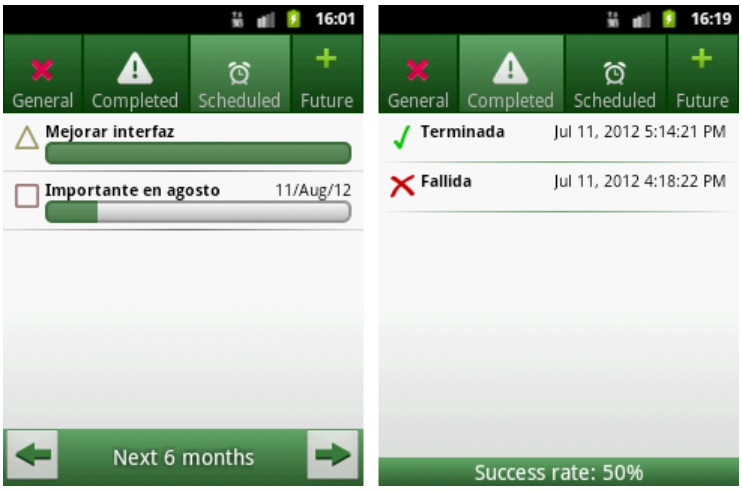

Figura 5.12: Segunda interfaz.

las alarmas al apagar el móvil. Este aspecto será probablemente corregido para la tercera versión. En la imagen 5.13 se puede observar el aspecto de una notificación de un evento llamado *fallida* que no llegó a completarse a tiempo.

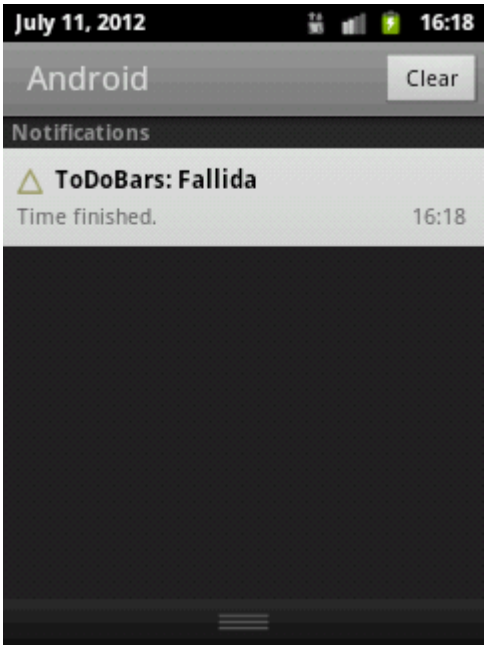

Figura 5.13: Notificacion de tiempo.

Tras haber realizado las pruebas correspondientes a cada una de estas novedades, no se ha detectado ningún error, con lo que a excepción de los detalles mencionados antes, la próxima iteración se compondrá en su mayoría de novedades.

5.4– Tercera iteración

5.4.1. Análisis de requisitos y prototipado

Con los avances de la segunda versión, se han cubierto en mayor o menor medida los requisitos funcionales y no funcionales definidos al comienzo del proyecto. Tras añadir el sistema de notificaciones y mejorado el soporte para idiomas, la aplicación va mostrando un aspecto más finalizado.

Sin embargo, aún necesita rellenar las dos pestañas restantes: la de información general y la cuarta, que por ahora se trata de tareas futuras.

Antes de proceder a diseñarlas, hemos de consultar con los usuarios los posibles cambios o mejoras que aún se puedan realizar a la aplicación, especialmente aquellos no relacionados con el aspecto como iconos o colores (siempre que no sean una molestia).

Además de preguntarles por la posible utilidad de la cuarta sección, también se les preguntará si ven correctamente el tiempo restante de las tareas sin ambigüedades, ya que algunos usuarios no estaban muy convencidos con la representación de la primera versión.

Para evaluar esto, se les mostrará una imagen con tres posibilidades de representación de las tareas. Deberán escoger el aspecto más adecuado y describir el por qué.

Los resultados rechazan ampliamente la opción B, cuyo aspecto cambiante nunca despertó mucha simpatía ni siquiera al pensarlo, pero difieren con las opciones A y C. La opción A es el estado actual. La opción C no parece errónea, sin embargo representa la fecha límite en el lado izquierdo, y realmente esa fecha se acerca en forma de barra desde la derecha a la izquierda. Es posible que una solución intermedia satisfaga a todos, pero por si acaso, se dejará como está y se volverá a repetir esta cuestión en el futuro, con más posibilidades de elección.

Figura 5.14: Encuesta sobre interfaz.

Sobre la utilidad de la cuarta pestaña, los usuarios no parecen muy conformes con otra lista de tareas que lo haría todo más confuso. Uno de ellos ha sugerido la posibilidad de incorporar un calendario a la cuarta pestaña, para poder ver el día actual y las tareas más cercanas.

Se trata de una idea interesante ya que proporcionamos otro punto de vista para cumplir el objetivo principal de la aplicación: mostrar lo mejor posible el tiempo restante para la realización de las tareas.

Hechos estos pasos, parece que los cambios a realizar son menores de lo esperado, más allá de mejorar si es posible el aspecto gráfico. Será entonces responsabilidad de la siguiente etapa incorporar el citado calendario e implementar la pestaña de datos generales.

5.4.2. Diseño

Esta vez, en esta fase no dispondremos de tanta carga de trabajo como en las anteriores. Hay que implementar dos vistas distintas, pero una de ellas sólo necesita ciertos botones y mostrar ciertos datos sencillos, y la otra no posee una lógica muy compleja detrás. El calendario reúne ciertas dificultades gráficas más que de arquitectura. Hay que disponer los días de forma adecuada, en un formato concreto de columnas y filas, sin faltar a la realidad. Sin embargo, no va a mostrar una lista de información, ni ha de acceder de forma exhaustiva a las estructuras de datos principales de la aplicación.

Sí las consultará para mostrar un icono en el día que tenga eventos, pero no mostrará todos los de la lista, sino sólo los que venzan en ese mes

concreto.

Se dispondrán de dos botones como en el resto de la aplicación que permitan pasar de un mes al siguiente. Lo normal es que Android proporcionase este tipo de calendarios de forma automática. Sin embargo, para las versiones a las que va destinada principalmente esta aplicación, no parecen dar esta posibilidad, con lo que es menester de cada desarrollador incluirlo en su creación.

Tras algo de investigación, la mejor opción es usar un *GridView* (que coloca una malla de elementos) mejor que una *TableLayout* (más enfocada a tablas de texto). Si usáramos el segundo, habría que implementar manualmente las vistas, pero con *GridView* podemos recurrir de nuevo a un *ArrayAdapter* sobre una lista como *ArrayList* para la apariencia. De esta forma, creamos una lista de los días que debe tener un mes, y los desplazamos según su día de inicio incluyendo elementos en la lista que no se van a ver. Finalmente, el estilo que muestra cada día se almacenará en un XML para que sea fácil de modificar sin involucrar apenas código.

Como la mayor parte de la lógica de la aplicación ya está implementada y explicada (excepto algunos botones, alertas o comparadores de menor relevancia), podemos pasar a la fase de implementación.

5.4.3. Implementación y pruebas

Con la mayoría de los objetivos ya cubiertos, pasamos a crear las dos pestañas restantes. Para la primera se dispondrán de dos botones, uno que marque la tarea actual como finalizada, otro para borrarla, y otros para navegar de una a otra.

La pestaña de calendario será implementada como se especificó en la etapa de Diseño. Se creará un GridView de 7 columnas que se desplazarán ciertos días según cómo empiece ese mes. Para hacerlo, se volverá a usar un ArrayAdapter personalizado como los que se usaron en la primera y segunda iteración, de forma similar a lo mostrado en la figura 5.9.

A ese ArrayAdapter se le pasará una lista de los días del mes correspondiente, creada para tal efecto, y cada uno de ellos llevará un icono si existe una tarea que termine ese día. Para conseguir el desplazamiento del mes, se incluirán días *fantasma* que no se vean y no cuenten en absoluto, ya que es la forma más rápida y sencilla para implementarlo.

Figura 5.15: Sección general de ToDoBars.

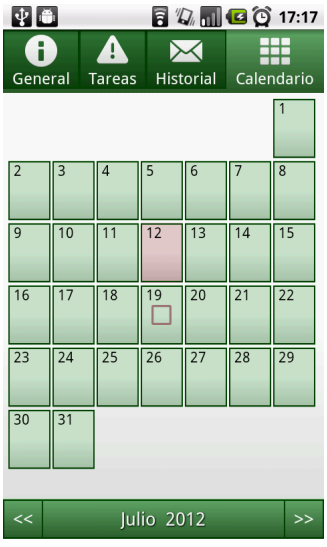

Figura 5.16: Calendario de ToDoBars.

Con esta implementación se cumple una de las sugerencias de los usuarios que han sido de gran ayuda para definir una aplicación mejor de lo esperado. En general, se puede observar una mejoría en la interfaz gráfica, pero aún existe un buen margen de maniobra para darle más calidad.

Casi todas las pestañas anteriores también mejoran su apariencia, y de hecho, cambian su ubicación, ya que la lista principal de tareas pasa a la segunda posición, dejando a historial en la tercera pestaña, ya que su uso será menor. En las figuras de este capítulo podemos observar cómo ha quedado finalmente los dos añadidos anteriores y las secciones de las listas de tareas en sus nuevas posiciones.

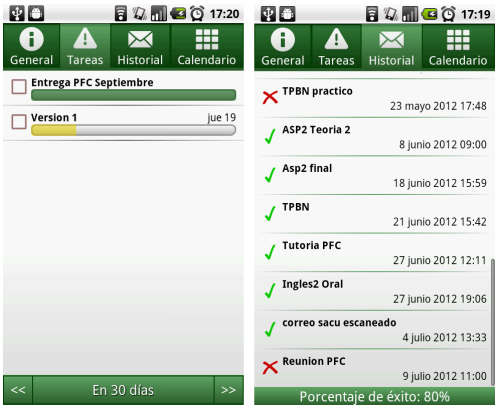

Figura 5.17: Tareas e historial de ToDoBars.

A falta de ciertos detalles, ya se podría decir que la aplicación está acabada. Sus objetivos principales han sido cumplidos, al igual que los objetivos de formación. Algunas mejoras que habrían de incluirse antes de una salida al mercado de aplicaciones de Android serían:

- Reestructurar la interfaz. Aunque los usuarios coinciden en que la mejoría general de la interfaz ha sido importante, el concepto de *horizonte* y su control sigue provocando algunas dudas. Sería una buena idea separar las tareas en varias secciones si les queda menos de un día, menos de una semana, y menos de un mes, en vez de mostrarlas todas.

- La selección de fecha podría hacerse de manera manual y no depender de la que posea el sistema operativo. En algunas versiones se usa un calendario cómodo, pero en otras se selecciona manualmente día, mes y año.

- Utilizar un Service que gestione las alarmas y las reactive al arrancar el sistema operativo para evitar cualquier posible pérdida de avisos.
- Añadir un sistema de rutinas. Algunos usuarios informan de la necesidad de reactivar ciertas tareas una vez cumplidas. Tomar una pastilla a cierta hora, pasear al perro, u actividades rutinarias podrían implementarse con alguna modificación a la importancia de cada tarea.
- En el calendario, añadir la opción de que al pulsar sobre un día aparezca una lista con las actividades que vencen ese día. Mostrar el icono es buena idea pero no suficiente.
- Crear un menú de configuración para que los usuarios decidan por ejemplo el aspecto de la interfaz. Así podrían cambiar el color de fondo de la aplicación de claro a oscuro o viceversa, o también modificar los colores de las barras a su criterio. Otras opciones podrían ser para importar o exportar datos, y/o cambiar avisos y sonidos.
- Para extender funcionalidad se podría crear un Widget que muestre información de la aplicación en el escritorio del usuario de forma más directa y diaria.

Durante el desarrollo se ha conseguido una excelente curva de aprendizaje que ha resultado en una formación rápida, eminentemente práctica, y sobre todo útil de cara a realizar el proyecto de fin de carrera con la mejor base posible en el tiempo requerido. Todos los aspectos formativos quedarán reflejados en la siguiente sección.

5.5– Resultados

Una vez completada la aplicación, al menos hasta un nivel aceptable para su salida al mercado de aplicaciones *Google Play*, llega el momento de hacer balance de lo aprendido y conseguido hasta el momento.

Aunque el desarrollo de esta aplicación previa podría considerarse como un retraso del proyecto final, las mejoras y ventajas de tal decisión se podrán observar en todas y cada una de las secciones posteriores en este documento, donde se explicará qué ideas, código y formación han sido reutilizados del trabajo anterior.

De hecho, gran cantidad de ese conocimiento adquirido ya se ha podido ver en acción en cada una de las mejoras para las distintas versiones de

la aplicación. Algunos se hacen evidentes para todo primer desarrollo en Android, pero otros han necesitado una aplicación práctica donde ponerlos a prueba para observar su utilidad.

En definitiva, la formación adquirida y utilizable en proyectos futuros se puede resumir en estos elementos:

- Activity: La fachada de toda aplicación, y el vínculo entre la misma y el usuario. Al igual que el elemento *Service*, están sujetos a los ciclos de vida que Android les imponga, debiendo cerrarse o *descansar* cuando se les exija.

- Vistas o *View*: Son los objetos gráficos de una aplicación. Los hay de todo tipo: botones, menús desplegables, selectores de opciones, etc. Para añadirles funcionalidades cuando se pulsen o modifiquen su estado, se usan las interfaces de tipo *Listener*

- Interfaces de tipo *Listener*: Cada clase que las implemente, necesitará ser utilizada para poder controlar el comportamiento de una vista ante un cambio, como por ejemplo, pulsar un botón o escribir texto.

- Layout: Son los patrones que definen la posición de los elementos gráficos en una aplicación. Por ejemplo, *LinearLayout* nos permite colocar una lista vertical u horizontal de objetos en pantalla, mientras que *RelativeLayout* nos da recursos para colocarlos donde queramos, definiendo incluso relaciones de posición entre ellos.

- ViewGroups: Se tratan de estructuras que agrupan varias vistas o Layouts añadiéndoles cierto comportamiento especial. Por ejemplo, *ScrollView* permite bajar o subir una lista de elementos mediante gestos táctiles, y *TabHost* y *ViewFlipper* permiten alternar entre varias pestañas de contenido.

- Ciclos de vida: Las aplicaciones y sus servicios en segundo plano tienen un tiempo de vida limitado. El sistema operativo puede cerrarlas o pausarlas cuando necesite más memoria disponible o si han terminado de ejecutarse, y para ello, debemos prepararlas convenientemente.

- *ArrayAdapter*: Se trata de una clase que nos permite enlazar un elemento de una lista de objetos (como un *Array*), con su correspondiente apariencia gráfica (un *ListView* o *GridView*) y comportamiento en cuanto a pulsaciones (añadiéndoles algún *Listener*). A menos que necesitemos una personalización extrema, se trata de un

sistema cómodo y útil que nos evita crear el aspecto de cada elemento manualmente. Además, dispone también de una vista automática para listas vacías.

- Archivos de idioma: La tentación de realizarla manualmente es instantánea nada más empezar, pero Android nos proporciona una carpeta donde colocar las traducciones. Cada cadena de texto tiene un identificador, y según el lenguaje que posea el sistema operativo, se buscará esa cadena en el archivo correspondiente. De ésta forma, añadir un nuevo lenguaje es tan sencillo como copiar otro archivo de lenguaje, traducir, y cambiarle el nombre al archivo, sin alterar el código en ningún momento.

- Permisos: Para cuidar la seguridad, los permisos de acceso de una aplicación están convenientemente categorizados y notificados, tanto al crearla como al instalarla. Así, el usuario sabe en todo momento qué secciones de su información se están usando o se van a usar.

- Manifest: Supone el *contrato* entre el desarrollador y el sistema operativo. En él se estipulan los permisos que la aplicación necesita para ejecutarse, cada Service o Activity que la componen, y las clases que reciban datos del sistema.

- Receivers: Son clases que reciben datos del sistema operativo periódicamente. Para que Android las reconozca y les envíe información, deben figurar en el Manifest y tener los permisos adecuados.

- Iconos: La gestión de iconos y de todo recurso gráfico se realiza como en el caso de las cadenas de texto. Para su correcta visualización en todos los dispositivos independientemente de su tamaño o resolución, existen diversas carpetas donde colocar gráficos según su longitud, que el sistema irá utilizando según necesite. Todos poseen un identificador para asignarlos a nivel de código.

- Intents: Son los mensajes que discurren por el sistema operativo contínuamente. Algunos se pueden consultar y recibir mediante Receivers, pero otros se pueden crear de forma personalizada y exclusiva para nuestra aplicación. Transmiten tipos básicos como números y booleanos, pero también objetos completos si los preparamos para ello.

- Bundle: Es un tipo de objeto que empaqueta otros elementos dentro de un Intent. Se pueden utilizar para estructurarlos o para recibir datos empaquetados de algún Intent específico de Android.

- Parcelable: Se trata de la interfaz que un objeto ha de implementar si quiere poder ser enviado dentro de un Intent. Por ejemplo, si un

Service envía un objeto de tipo *Tarea* a una Activity, la segunda debe tener un Receiver, y la clase *Tarea* debe tener métodos de tipo Parcelable que la preparen para su envío.

- Serializable: Es un sistema de permanencia de datos que permite guardar objetos en un archivo en disco. Si un objeto ha de ser *serializado*, debe implementar ciertos métodos para su creación y reconversión. Parcelable es el sistema de serialización para Android, más rápido y optimizado para la plataforma, y Serializable su antecesor en el lenguaje de programación Java.

- AlarmManager: Es una clase de Android que se encarga de gestionar el sistema de alarmas. Si nuestra aplicación necesita realizar una actividad a una hora concreta, antes que utilizar algún sistema clásico de reloj de Java, podemos usar éste sistema para que envíe un Intent que podamos tratar como nos convenga, escapando también del problema del ciclo de vida limitado.

- Notificaciones: Android posee un sistema en la barra superior que nos va notificando de los eventos que ocurren en nuestro terminal, ya sean de batería, wifi, llamadas entrantes o mensajes entrantes. Como desarrollador, se pueden crear notificaciones al usuario cuando llegue cierta hora o ocurra algo en especial, si usamos la clase *NotificationManager*.

- *Log* y *Logcat*: Son los sistemas de impresión de texto para detección de errores. Nos permite enviar texto a un sistema de monitorización y consultar en todo momento el estado de nuestra aplicación y de los errores que pudieran encontrarse.

Todos estos elementos y otros similares conforman una buena base de conocimiento para realizar un buen diseño de una aplicación, que evite constantes modificaciones, sea más robusta, y consiga cumplir sus funciones de una forma óptima y sencilla.

Además, la experiencia adquirida a nivel de código siempre es útil, ya que las horas utilizadas de forma práctica con el lenguaje de programación, el entorno de desarrollo, y la resolución de problemas, combinadas con el tiempo de investigación dedicado a cada uno de los elementos anteriores, han sido mucho más eficientes y directas que en una hipotética formación exclusivamente teórica.

Y no sólo eso, ya que el código realizado, excepto aquél propenso a errores o a mejorarse debido al conocimiento de nuevas técnicas, puede ser reutilizado en el siguiente proyecto, con un coste de tiempo irrisorio.

Es el momento entonces de comenzar el desarrollo del proyecto definitivo *Commandroid*, que dará al usuario la capacidad de controlar el comportamiento de diversas redes inalámbricas según el estado de la batería. Durante los siguientes capítulos se podrán observar referencias a estos conocimientos adquiridos y su mayor o menor impacto en la mejora del proyecto.

CAPÍTULO 6

Commandroid: Análisis de Requisitos

6.1– Introducción: Usuarios en el desarrollo.

Los procedimientos de desarrollo que se usarán para este proyecto van a ser muy parecidos al caso anterior. Dispondremos de un pequeño grupo de usuarios al que se intentará involucrar lo máximo posible en la evaluación y desarrollo de la aplicación.

En este caso, el impacto de los mismos sobre el sistema no será tan fuerte. No porque su participación se reduzca, sino porque en este nuevo proyecto, la interfaz juega un papel secundario y menos relevante.

Aunque una interfaz limpia y sencilla de utilizar es algo básico para cualquier aplicación que desee tener un mínimo éxito, ya no comercial sino de satisfacción de sus usuarios, ahora el peso debe recaer sobre la utilidad que tendrá esta nueva aplicación.

De hecho, en los objetivos definidos anteriormente, los usuarios van a ser clave. El peso de la calidad del desarrollo va a caer íntegramente en su evaluación por parte de los clientes. En aquella sección se definió la meta principal de proporcionar herramientas al usuario para poder controlar el gasto de batería y la conectividad que deseen.

Para poder conseguirla, es necesario consultar con cada uno de ellos, analizar sus pautas de uso del terminal, informarse de sus necesidades al respecto, y preguntarles durante el desarrollo sobre posibles mejoras del

sistema.

Entonces, como primer paso se ha decidido utilizar un sistema de encuestas para obtener datos que nos puedan guiar por buen camino. Las preguntas serán sencillas, y encaminadas a conseguir una percepción general del problema, sin influencia alguna por el rol de desarrollador o por la idea original.

Las cuestiones a responder son las siguientes:

- Datos Generales: Nombre, modelo de teléfono móvil y versión de Android.
- *¿Cuánto tiempo usas tu móvil Android al día?*
- *De ese tiempo, ¿cuánto tiempo permaneces conectado a internet?*
- *¿Cuánto suele durar la batería del móvil en el peor caso? ¿Y en el mejor caso?*
- *¿Posees tarifa de datos o tarifa "3G"?*
- *En caso afirmativo, ¿usas también la conexión Wifi? ¿Durante cuánto tiempo?*
- *Una vez dejas de utilizar el teléfono: ¿Dejas encendido el Wifi?*
- *En tu móvil, cuando entra en suspensión, ¿se apaga el Wifi automáticamente?*
- *¿Sueles usar alguna conexión inalámbrica cuando estás de camino a algún sitio? ¿Con qué frecuencia?*
- *¿Te sueles conectar a puntos Wifi de acceso libre? ¿Con qué frecuencia?*
- *¿Usas la tecnología Bluetooth? ¿Con cuánta frecuencia?*
- *¿Usas la geolocalización GPS? ¿Con cuánta frecuencia?*
- *¿Tienes instalada alguna aplicación externa que gestione redes inalámbricas? Si es así, ¿cuál es su nombre?*
- *Aproximadamente, ¿cuántas aplicaciones tienes instaladas que requieran una conexión frecuente a internet (correo, redes sociales, etc)?*
- *¿Cuántas veces al día las consultas?*

- *¿Apagas el móvil por la noche?*
- *En caso negativo, ¿colocas el móvil en silencio por la noche?*
- *¿Has tenido algún problema con Android en cuanto al Wifi o a la batería? Descríbelo.*
- *¿Crees que las opciones de configuración de redes inalámbricas son accesibles y sencillas? ¿Cómo las mejorarías?*

Como puede observarse, las preguntas están enfocadas a conseguir datos precisos sobre el uso promedio del terminal y de sus redes inalámbricas disponibles. Tras recopilar las respuestas y simplificarlas a unos niveles más numéricos y analizables, obtenemos una tabla de la siguiente forma:

Cuadro 6.1: Encuesta a usuarios. Primera parte.

	A	**B**	**C**
Telefono:	ZTE Blade	HTC Wildfire	ZTE Blade
Version de Android:	2.3.4	2.2.1	4.0.4
Tiempo de uso:	1h	Constante.	1h
Tiempo conectado:	1h	3 o 4h	30min
Bateria mejor caso:	1semana	2.5 dias	2.5 dias
Bateria peor caso:	1dia	Casi 1 dia	1 dia
Tarifa 3G:	No	No	Si
3G y Wifi:	-	-	Si
Tiempo wifi:	-	-	15min
Dejar encendido wifi:	No	Si	No
¿Wifi se apaga sólo?	No	No	No
Wifi en movimiento:	Si, libre.	Poco	Poco
Wifi acceso libre:	Si	Si	Si
Frecuencia wifi libre:	A veces.	A veces	A veces
Bluetooth	1vez	No	No
GPS:	1vez	No	No
App para bateria:	No.	No	BatteryDrain
Nº apps con internet:	3 o 4	4	4 o 5
Consultas al día:	3	Cada hora	4 o 5
Apagado de noche:	Si	No	No
En silencio:	-	No	No

Cuadro 6.2: Encuesta a usuarios. Segunda parte.

	D	**E**	**F**
Telefono:	Galaxy Mini	HTCWildfireS	Nexus One
Version de Android:	2.3.4	2.3	Miui 2.3
Tiempo de uso:	2:30h	Constante	Constante
Tiempo conectado:	2:30h	Permanente	Permanente
Bateria mejor caso:	1semana	2dias	60h wifi
Bateria peor caso:	3dias	12h	36h 3G
Tarifa 3G:	No	Si	Si
3G y Wifi:	-	No, solo 3G	Si, a veces
Tiempo wifi:	-	-	
Dejar encendido wifi:	Si	No	Si
¿Wifi se apaga sólo?	No	No	Wifi Si.
Wifi en movimiento:	No	No (3G si)	No (3G si)
Wifi acceso libre:	Si	No	No
Frecuencia wifi libre:	1 por semana	No	No
Bluetooth	No	No	A veces
GPS:	No	No	Viajes
App para bateria:	No	No	No
Nº apps con internet:	10	3	10
Consultas al día:	1 o 2 al dia	Constante	Cada hora
Apagado de noche:	No	Si	No
En silencio:	Si	-	Si

Cuadro 6.3: Encuesta a usuarios. Tercera parte.

	G
Telefono:	ZTE Blade
Version de Android:	2.2.2
Tiempo de uso:	2h
Tiempo conectado:	1:30h
Bateria mejor caso:	4dias
Bateria peor caso:	1dia
Tarifa 3G:	No
3G y Wifi:	-
Tiempo wifi:	-
Dejar encendido wifi:	Si
¿Wifi se apaga sólo?	Wifi Si
Wifi en movimiento:	No
Wifi acceso libre:	Si
Frecuencia wifi libre:	Poco
Bluetooth	No
GPS:	1 vez
App para bateria:	No
Nº apps con internet:	3 o 4
Consultas al día:	Cada 2 h
Apagado de noche:	No
En silencio:	Si

Si polarizamos un poco los resultados y los clasificamos, se pueden observar aproximadamente dos grupos. El primer grupo, que estaría compuesto por los usuarios A, C, D y G realizan un uso moderado del móvil y sus conexiones. Aún con diferencias, comprueban el móvil cada cierto tiempo y sólo encienden las redes inalámbricas (normalmente Wifi) para comprobar si existe nueva información.

El segundo grupo, en cambio, realiza un uso intensivo del terminal. Se suelen conectar a las redes inalámbricas la mayor parte del día y necesitan información actualizada con alta frecuencia.

Siguiendo los objetivos de la aplicación, la misión a cumplir es ahorrar el máximo posible de batería con los recursos que podamos, pero siempre sin perder ni un ápice de conectividad para que los usuarios no pierdan notificaciones importantes.

Para ello, es necesario tener el mente ambos grupos de usuarios y cumplir sus expectativas, creando modos de conexión lo suficientemente comprensivos para ambos tipos de uso. Si automatizamos de alguna forma las conexiones, es posible que durante los periodos de inactividad se pueda ahorrar batería, por ejemplo, brindándole la posibilidad al usuario de crear un plan de ahorro nocturno, que desconecte las redes inalámbricas.

Al mismo tiempo, hay que cuidar las interacciones con los diversos sistemas operativos que utilizan nuestros usuarios. Algunos de ellos poseen versiones oficiales, otros versiones modificadas de código libre, y todos en general, de distintos tipos de fabricantes.

Esta mezcla de dispositivos merece especial atención en lo relativo al tratamiento de la batería, ya que cada terminal posee un hardware distinto, que a su vez está controlado por un driver que bien pudiera ser oficial o creado por un tercero, y luego una tercera capa compuesta por el sistema operativo.

Entre los tres elementos se pueden dar multitud de configuraciones distintas, algunas de ahorro extremo, y otras de gasto contínuo para que los usuarios no experimenten problema alguno en su uso, excepto por el evidente incremento en gasto de batería. De hecho, algunas de las preguntas realizadas indagan sobre estos sistemas. La cuestión sobre la desactivación automática de las redes WiFi será clave durante este documento, ya que las respuestas de los usuarios G y F, y la experiencia previa en otras versiones de A y C, denotan un posible sistema de ahorro que desactiva dicha red en cuanto el usuario se ausenta, perdiendo la conexión sin control por parte del usuario.

Este tipo de necesidades, problemas o falta de control causan cierta impotencia al usuario, y es exactamente lo que queremos evitar en la realización de este proyecto. Para definir mejor estas metas, y transformarlas en documentación más exhaustiva y numerada, se volverán a tratar los objetivos de capítulos anteriores de forma más completa a continuación.

6.2– Definición de objetivos

Es hora entonces de redefinir el contenido del capítulo 2 y concretar los objetivos de una forma similar a la utilizada en el proyecto anterior antes de definir requisitos más específicos. Para ello, se dividirán en diversas categorías y se numerarán de cara a posteriores menciones en los capítulos de diseño e implementación.

6.2.1. Minimizar el gasto de batería del terminal

Descripción

El sistema deberá cuidar en todo momento el gasto de batería que cause tanto su funcionamiento como el uso de las redes inalámbricas, sensores y altavoces del dispositivo. El sistema debe proporcionar tantas configuraciones como sean posibles para que el usuario sea capaz de controlar todos los aspectos de estas políticas de ahorro de energía.

Subobjetivos

- A: Permitir la creación de perfiles de ahorro por franjas temporales. El sistema debe proporcionar herramientas para que el usuario pueda preparar órdenes a ejecutar en las horas del día que desee. Estos comandos deben permitir la modificación del estado y configuración tanto de redes inalámbricas como de otros dispositivos internos que consuman energía.

- B: Preparar el sistema para que también pueda activar dichos perfiles ante condiciones cambiantes, tales como modificaciones en el estado de la batería, conexión inalámbrica o los sensores del dispositivo.

6.2.2. Maximizar la conectividad del dispositivo

Descripción

El sistema deberá proporcionar al usuario las herramientas que necesite para garantizar periodos de conexión estables y de calidad el tiempo que le sea necesario. De ser posible, debe proporcionar alternativas a las configuraciones estándares del terminal que refuercen dichas conexiones.

Subobjetivos

- A: Permitir conexiones cíclicas. Este tipo de conexiones, cumpliendo con los objetivos iniciales de ahorro de batería deberán permitir al usuario una configuración que se repita en el tiempo y que alterne pequeños periodos de conexión, en los que gastaríamos menos energía al emplear menos tiempo, con periodos de desconexión en los que se ahorrará batería con respecto a una conexión contínua.

- B: Dar poder al usuario para evitar políticas de ahorro de su sistema operativo o fabricante que perjudiquen la conectividad del terminal. El sistema deberá proporcionar opciones para evadir, evitar o modificar dichos perfiles de ahorro a los que el usuario no pudiera acceder, manteniendo conexiones activas si lo considerara oportuno, evitando desconexiones forzadas por el fabricante.

- C: De forma opcional, gestionar las conexiones inalámbricas y sus configuraciones para que se conecten a la mejor red posible de la forma que sea necesaria, y mejoren u optimicen las funcionalidades del sistema operativo a tal efecto.

6.2.3. Garantizar control e información al usuario sobre el uso de batería y conexiones.

Descripción

La aplicación deberá proporcionar información detallada al usuario sobre el uso del terminal para estas dos variables, así como herramientas de control sobre el comportamiento del sistema en su ausencia, con el fin de cumplir los dos objetivos anteriores.

Subobjetivos

- A: Informar sobre el gasto de batería y estado del terminal. El sistema debe tener opciones para consultar no sólo el estado actual del consumo de energía y el estado de las redes inalámbricas sino también sus valores históricos hasta cierto punto.
- B: Proporcionarle al usuario control sobre todos los modos definidos anteriormente. Debe existir un sistema de perfiles de ahorro que se activarán en ocasiones a unas horas determinadas, y en otras, debido a ciertas variables del estado del terminal. Todas las opciones de ahorro o de conectividad comentadas con anterioridad estarán disponibles para estos perfiles.

6.3– Requisitos del sistema

Una vez definidos los objetivos de la aplicación con mayor detalle, pasamos a concretar los requisitos de la misma. Existen varios tipos de requisitos según su influencia y necesidad.

Por ejemplo, los requisitos de información son aquellos que especifican exactamente qué datos que se van a almacenar en el sistema, mientras que los requisitos funcionales se encargan de detallar los casos de uso que la aplicación debe implementar para que un agente externo o usuario realice diversas tareas. Son también necesarios los requisitos no funcionales, que mostrarán los aspectos de rendimiento o calidad que el sistema debe poseer para cumplir correctamente con el contrato del cliente.

6.3.1. Requisitos de información

Información del estado de la batería

El sistema deberá almacenar los datos que nos proporcione el sistema operativo en todo momento sobre el consumo y estado de la batería del terminal. Algunas de estas variables pueden ser: el nivel de energía, la temperatura, si está en carga o descarga, o la salud de la misma.

Posee relación con los objetivos de la sección 6.2: 1-B, 3, 3-A, 3-B.

Información del estado de las redes inalámbricas

La aplicación deberá guardar la información relativa al estado actual de las diversas redes inalámbricas, con especial prioridad por el estado de las redes Wi-Fi, y en segundo lugar las redes de datos móviles. Algunos de sus datos pueden ser: la existencia o no de acceso a internet, la dirección IP del dispositivo, o el tipo de conexión.

Posee relación con los objetivos de la sección 6.2: 1-B, 3, 3-A, 3-B.

Información del estado de los sensores extra

El sistema deberá almacenar el estado de los sensores que el fabricante del terminal nos proporcione. Pueden tratarse de todo tipo de datos, como: estado de la pantalla, brillo, estado de los altavoces, y sensores de posición o inclinación.

Posee relación con los objetivos de la sección 6.2: 1-B, 3, 3-A, 3-B.

Perfiles de ahorro definidos por el usuario

La aplicación debe almacenar todos aquellos datos que sean necesarios en la definición de un perfil de ahorro que determine el comportamiento del terminal cuando no lo esté usando activamente. Debe ser capaz de almacenar un sistema de horarios, variables para desactivar o activar el perfil, causas que lo hagan ejecutarse, y acciones que debe ejecutar en tal caso.

Se trata de un requisito esencial que posee relación todos o casi todos los objetivos de la sección 6.2: 1, 1-A, 1-B, 2, 2-A, 2-B, 2-C, 3, 3-B.

Configuracion específica de apoyo a las redes inalámbricas

El sistema deberá almacenar todas aquellas configuraciones que sean necesarias para mejorar el uso y acceso a redes inalámbricas para mejorar la conectividad a internet del terminal.

Posee relación con los objetivos de la sección 6.2: 2-C, 3, 3-A.

Configuracion general de la aplicación

La aplicación deberá guardar la información relativa al comportamiento general de la aplicación que no se contemplen en los requisitos de información anteriores. Algunos de estos datos pueden ser: selección de estilo de la interfaz y los tiempos o ciclos de activación de los perfiles o del guardado de información.

Posee relación con los objetivos de la sección 6.2: 3, 3-A, 3-B.

6.3.2. Requisitos funcionales y casos de uso.

En la sección de requisitos funcionales suele tener gran impacto el modelado de negocio del cliente. En este caso, no existe un negocio como tal sobre el cual mejorar las tecnologías existentes y optimizar el flujo de negocio con nuestro sistema.

La aplicación que nos ocupa está totalmente dirigida a usuarios finales, con lo que sólo un tipo de actor o factor humano interviene en la aplicación. Por ello, el diagrama de caso de uso se centrará en el usuario, y tendrá mayor utilidad para mostrar gráficamente sus interacciones con el sistema, en vez de plasmar la interacción del sistema con más de un actor como suele representarse en éste tipo de gráficos.

Cabe destacar que en esta sección no se muestran aquellos algoritmos que el sistema debe ejecutar en un segundo plano, ya que no poseen vínculos directos con la acción del usuario.

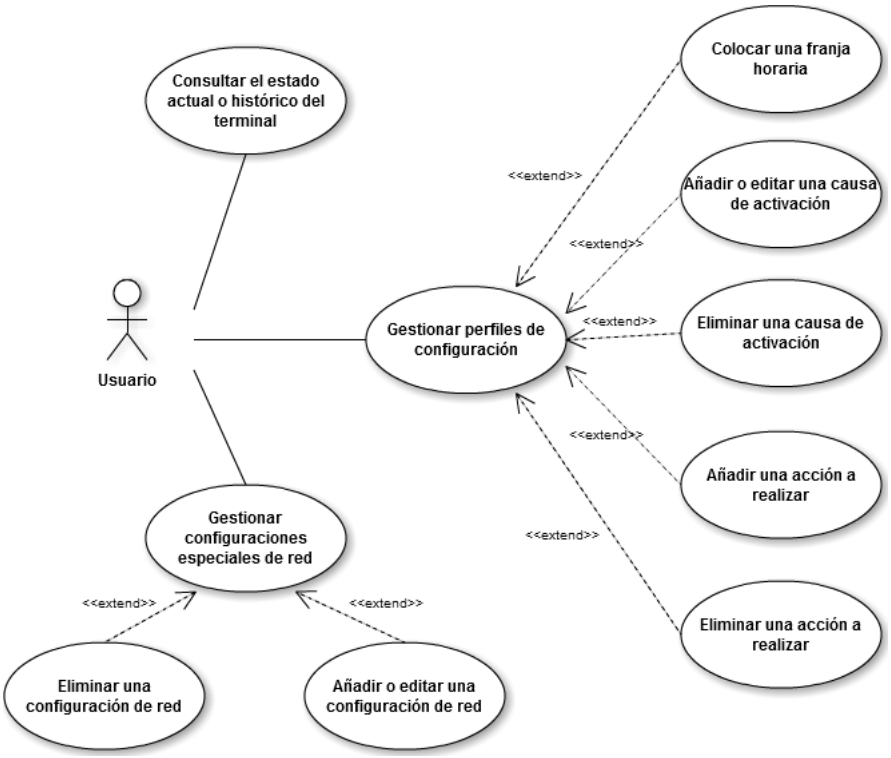

Figura 6.1: Commandroid: Diagrama de caso de uso

Caso de uso: Consultar el estado actual o histórico del terminal

El usuario, al pulsar sobre la sección de *Datos*, pulsará el botón que le permite elegir el tipo de información. La aplicación mostrará una lista de las variables de aquellos datos que tiene almacenados. El usuario escoge entonces una, y otras opciones de visualización y el sistema lo muestra en forma de gráfica.

Caso de uso: Gestionar perfiles de configuración

El usuario accede a la sección de perfiles cuando desea modificar algun perfil de configuración. El sistema le muestra una lista de los disponibles, de haberlos, y el usuario decide crear uno nuevo o editar uno de los disponibles.

Este caso de uso está compuesto por dos casos definidos a continuación.

Caso de uso: Eliminar un perfil de configuración

El usuario accede a la sección de *Perfiles*, en la que el sistema le muestra los perfiles existentes. El usuario procede entonces a pulsar el botón de eliminar perfil y el sistema pregunta al usuario si está de acuerdo. El usuario confirma la acción y el perfil se elimina.

Caso de uso: Crear o editar un perfil de configuración

El usuario accede a la sección de *Perfiles*, en la que el sistema le muestra los perfiles existentes. El usuario pulsa sobre uno de los perfiles o sobre el botón de crear uno nuevo. El sistema muestra varios controles para añadir acciones, causas, franjas horarias, un botón de activación y una caja de texto para nombrar al perfil. El usuario introduce esos datos, pulsa el botón aceptar y el perfil se crea.

Este caso de uso está compuesto por los cinco siguientes.

Caso de uso: Colocar una franja horaria

El usuario, tras decidir crear o editar un perfil, pulsa sobre el botón de la hora de inicio. El sistema le muestra una ventana donde puede elegir las horas de aplicación. El usuario introduce la información de comienzo y vuelve a repetirlo con la hora de finalización. El sistema comprueba si el orden cronológico entre ambas es correcto y modifica el perfil.

Caso de uso: Añadir o editar una causa de activación

El usuario, tras decidir crear o editar un perfil, escoge la sección de activadores. El sistema muestra una lista de los activadores actuales. El usuario pulsa sobre el botón de añadir y el sistema le muestra una serie de opciones. Tras elegir una, el usuario escoge un valor que debe cumplir esa activación, un operador si el activador lo requiere, y el sistema almacena la nueva condición de activación del perfil.

Caso de uso: Eliminar una causa de activación

El usuario, tras decidir crear o editar un perfil, escoge la sección de activadores. El sistema muestra una lista de los activadores actuales. El usuario pulsa sobre el botón de eliminar y el sistema le pide confirmación. Tras la respuesta del usuario, elimina o no el activador escogido.

Caso de uso: Añadir una acción a realizar

El usuario, tras decidir crear o editar un perfil, escoge la sección de acciones. El sistema muestra una lista de las acciones actuales. El usuario pulsa sobre el botón de añadir y el sistema le muestra una serie de opciones. Tras elegir una acción, el usuario escoge un valor que deberá modificar cuando se active y el sistema almacena la nueva información.

Caso de uso: Eliminar una acción a realizar

El usuario, tras decidir crear o editar un perfil, escoge la sección de acciones. El sistema muestra una lista de las acciones actuales. El usuario pulsa sobre el botón de eliminar y el sistema le pide confirmación. Tras la respuesta del usuario, elimina o no el activador seleccionado.

Caso de uso: Gestionar configuraciones especiales de red

El usuario accede a la sección de redes cuando desea modificar o crear alguna configuracion especial para redes inalámbricas. El sistema le muestra una lista de los disponibles, de haberlos, y el usuario decide crear alguno o editar uno de los existentes.

Este caso de uso está compuesto por dos casos definidos a continuación.

Caso de uso: Añadir o editar una configuración de red

El usuario accede a la sección de redes. El sistema muestra una lista de las configuraciones especiales de redes creadas. El usuario pulsa sobre el botón de añadir configuración de red. El sistema le muestra un formulario para que introduzca el nombre de la configuración especial, un

identificador de la red y otras opciones como preferencia o variables para fijar una IP estática. El usuario introduce los datos y activa las opciones que le interesen, y pulsa el botón de añadir y el sistema guarda la nueva configuración.

Caso de uso: Eliminar una configuración de red

El usuario accede a la sección de redes. El sistema muestra una lista de las configuraciones especiales de redes creadas. El usuario pulsa el botón de eliminar una de las configuraciones y el sistema le pide confirmación. Tras la respuesta del usuario se elimina o no el elemento de la lista.

6.3.3. Requisitos no funcionales

Requisito de interfaz: Usabilidad

El sistema debe poseer una interfaz limpia, y concisa, que no favorezca ningún tipo de ambigüedades ni resulte incómoda de usar. Además, los textos, gráficos e información deben representarse de forma clara para todo tipo de resoluciones.

Requisito de interfaz: Completitud en el control

La aplicación debe proporcionar el número adecuado de herramientas que permitan al usuario cumplir los objetivos del desarrollo en cuanto a ahorro de batería y conectividad del terminal, sin esconder opciones ni favorecer unas sobre otras con más o menos espacio gráfico.

Consumo de batería

El proyecto debe responder correctamente en su ejecución a unos patrones de gasto de batería estables. Si consumiese más energía de la que se pretende ahorrar, el sistema carecería de sentido y sus ventajas no serían tales. Las comprobaciones de los perfiles y el guardado de datos deben ser gestionados con cuidado.

Rendimiento

En toda aplicación de Android se debe cumplir con tiempos de respuesta a acciones del usuario menores a 5 segundos, excepto en casos de tiempos de carga por acceso a internet o derivados. Si la aplicación no respondiese adecuadamente en ese lapso de tiempo, Android cancelaría su ejecución produciendo resultados no deseados.

Soporte para idiomas

En cuanto sea posible, el sistema debe estar preparado para mostrar toda cadena de texto en el lenguaje que el usuario tenga especificado en su terminal. Cuantos más idiomas sea posible incluir, más variedad de usuarios pueden verse favorecidos, fortaleciendo la utilidad de la aplicación.

6.4– Restricciones adicionales

Para completar los requisitos no funcionales del apartado anterior, es necesaria una explicación más detallada de los mismos y de varias dificultades que encontraremos en el desarrollo pero que por su naturaleza, no se pueden modelar correctamente como un requisito no funcional.

La primera eventualidad, y una de las más importantes, es el consumo de batería. En todo el documento se repite constantemente este término y objetivo. Es evidente que si una aplicación destinada a ahorrar batería no lo consigue con ninguno de sus modos de configuración, necesita más tiempo de desarrollo y optimización en su proceder.

En el caso que nos ocupa, el sistema posee otros dos objetivos a cumplir, con lo que su utilidad puede verse favorecida si al menos consiguiese proporcionar más conectividad y control al usuario aunque por ello gaste algo más de batería. En resumen, en este proyecto se desea dar completo control al usuario sobre el gasto que se produce sobre posibles errores o limitaciones del sistema operativo, dándole a elegir más opciones de las que disponía anteriormente para poder aumentar o la conectividad o el ahorro, o intentar conseguir un término medio.

Otro de los problemas que nos podemos encontrar, que aunque no exista como tal en los requisitos no funcionales va a ser clave en el desarrollo, son los relativos a la configuración de ahorro de energía que posea el dis-

positivo, ya sea en el sistema operativo, o en algún controlador o *driver*. Este percance fue presentado en la introducción de esta fase de análisis de requisitos, y nos obligará a investigar sobre este tipo de políticas y su impacto en el mercado de terminales, así como las posibles soluciones que se hayan efectuado desde la comunidad de desarrolladores.

Existe también una restricción muy relacionada con el punto anterior que limitará el desarrollo de la aplicación. Se trata del alcance y las limitaciones que posee la *API* pública de Google. Una API es aproximadamente un compendio de los recursos que un componente software nos proporciona para poder usarlo correctamente. Es posible que para preservar un funcionamiento adecuado del dispositivo, o para evitar problemas de seguridad, se oculten deliberadamente aspectos de control de la API que nos limiten como desarrollador. En otros casos, también puede ocurrir que por la *juventud* del sistema aún haya funcionalidades no disponibles que se añadirían en un futuro.

Entonces, tanto por aquellas funciones que aún no estén disponibles como las protegidas por el creador, nos encontraremos algunas limitaciones en nuestro área de acción. Aún así, existen opciones más complejas y no exentas de riesgo que nos permitirían realizar acciones más allá de nuestro alcance. Normalmente aparecen como consecuencia de ingeniería inversa u otros métodos similares, y con ellos se podrían alcanzar nuevos objetivos.

Sin embargo, confiar el código de una aplicación a tales métodos pueden exponerla a condiciones cambiantes del hardware y del software, dejándola vulnerable y en ciertos casos, inestable. Por ello, en éste proyecto se apostará completamente por la API pública de Google que nos proporciona información detallada para un buen aprendizaje del desarrollador y nos garantiza cierta seguridad ante posibles cambios futuros.

Por último, no sólo habría que tener cuidado con los cambios que pudieran llegar de factores externos sino también de los internos. Cualquier nueva función que se desee añadir puede conllevar nuevos errores o problemas estructurales, con lo que es necesaria una correcta fase de diseño que realice una aplicación robusta y con un código fácil de modificar y de ampliar.

6.5– Prototipado de la interfaz de usuario

El prototipado de la interfaz, aunque se especifica como la última sección de análisis de requisitos en este documento, es un proceso que ocurre

durante toda la fase al completo. Desde el primer momento, con las entrevistas a los usuarios, se les presenta un prototipo en papel de la interfaz para captar los primeros y evidentes problemas que pudieran surgir.

Todo este procedimiento no sólo nos ayudará a crear interfaces más intuitivas y fáciles de usar, ya que también nos proporciona nuevas ideas, soluciones o incluso nos expone nuevas necesidades de los usuarios que podríamos solventar.

Para conseguir todos estos beneficios, se les presenta a los usuarios un primer boceto de la interfaz, para que opinen de forma general qué les parece. Esta primera versión hereda muchos de los conceptos del proyecto formativo previo *ToDoBars*, como la separación de la aplicación en cuatro secciones, así como de la distribución de los elementos de cada una. De hecho, ciertas partes del código serán reutilizadas y mejoradas tanto para este nuevo proyecto como para reutilizarlo también en el futuro.

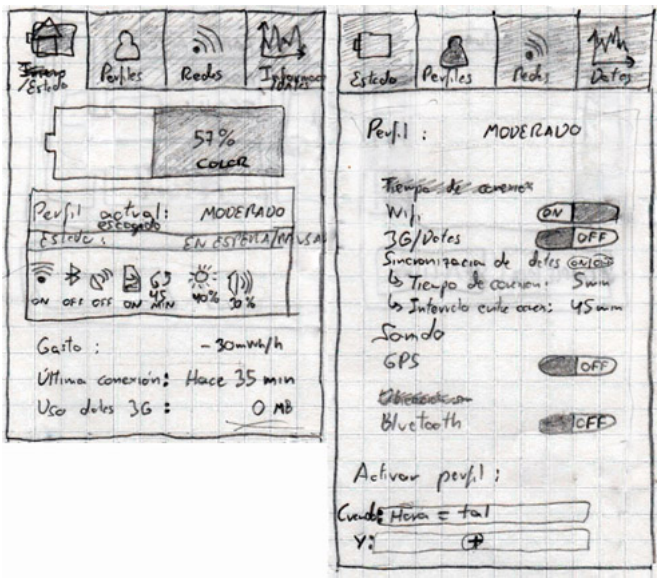

Figura 6.2: Primer prototipo de Commandroid. Pestañas 1 y 2

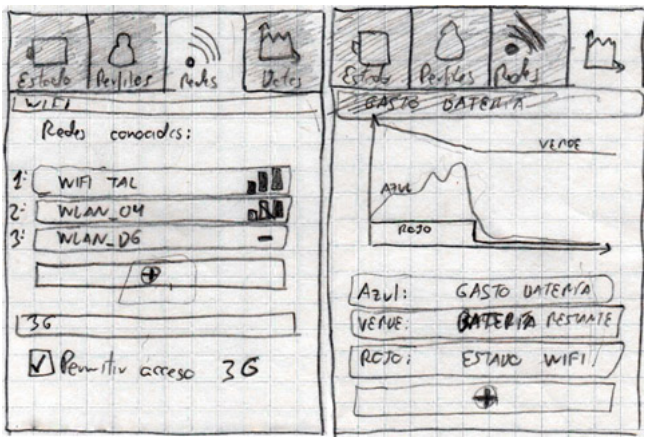

Figura 6.3: Primer prototipo de Commandroid. Pestañas 3 y 4

Tras la exposición de la interfaz a los usuarios, estos responden con buena acogida al diseño. Algunos de ellos destacan la gran cantidad de controles de la pestaña *perfiles*, y preguntan en general las funciones de ciertos elementos cuya explicación dentro de la interfaz es leve, como mucho.

Parece necesario renombrar ciertas características e intentar usar términos más coloquiales o intuitivos para ciertos elementos. En cualquier caso, la interfaz se irá mejorando poco a poco, pero como su impacto es menor en las prestaciones de una aplicación como ésta, aprovecharemos la ocasión para consultar con los usuarios si las funcionalidades (más tangibles ahora que durante las encuestas) les parecen correctas.

Tras la nueva consulta, gustan las ideas sobre mejorar las conexiones inalámbricas Wi-Fi desde la tercera pestaña, ya que Android parece conectarse a la última red conectada, a pesar de existir una conocida con mayor potencia y/o mayor cercanía. Los sistemas de consulta de información pasan ligeramente desapercibidos excepto por la posibilidad de contrastarlos a la vez con el nivel de batería, para poder consultar cómo baja o sube el gasto según las conexiones activas.

Sin embargo, es en la pestaña perfiles donde existe al mismo tiempo mayor confusión, y mayor interés. El sistema, hasta este punto, pretende darle control al usuario sobre esas variables, pero los usuarios aún lo ven complejo de utilizar. De hecho, uno de ellos pregunta si es necesario establecer todas las variables cuando sólo se quiere activar una de ellas.

Se hace prioritario entonces volver a diseñar de nuevo la pestaña y

orientar el sistema de forma distinta. Este proceso se adelanta ligeramente a la fase de diseño, donde deberemos prever este tipo de confusiones o problemas y encaminar la aplicación a una forma u otra. Gracias a esta revisión, se ha observado que tener que usar todos los controles, o mostrárselos todos de golpe al usuario puede provocar confusión, y aparece una solución: mostrarlos como una lista, en la que el usuario los vaya añadiendo uno a uno según les interese.

Entonces, con las distintas evaluaciones de los usuarios, se modifican los requisitos funcionales hasta llegar al nivel que se aprecia en las secciones anteriores, en las que los perfiles poseen dos listas: activadores y acciones. La aplicación consultará si los activadores se cumplen, y entonces ejecutaría las acciones del perfil. De esta forma, dejamos al usuario la posibilidad de mezclar los perfiles como más le interese, para que pueda controlar con flexibilidad las características de su teléfono o terminal. En la sección de diseño se entrará en detalle sobre las ventajas que esta decisión acarrea y como se afrontarán los posibles problemas a surgir.

Tras este proceso, la interfaz queda aproximadamente de la siguiente forma:

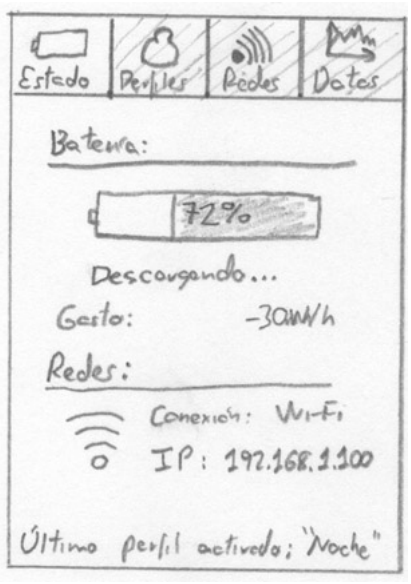

Figura 6.4: Segundo prototipo de Commandroid. Pestaña 1.

Figura 6.5: Segundo prototipo de Commandroid. Pestaña 2 y detalle.

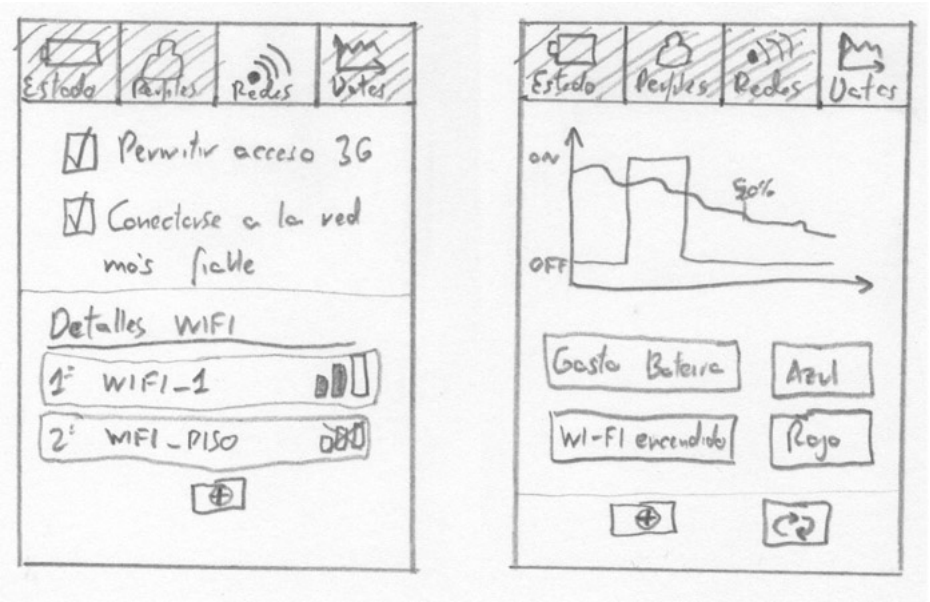

Figura 6.6: Segundo prototipo de Commandroid. Pestañas 3 y 4.

Una vez conseguido un diseño de la interfaz aproximadamente satisfactorio, procedemos a la fase de diseño, donde se creará la arquitectura y estructura de la aplicación, respetando siempre los objetivos, requisitos y prototipos de interfaz ya definidos.

CAPÍTULO 7

Commandroid: Diseño

Tras haber completado la fase de análisis de requisitos, y establido pues todas las pautas que debe cumplir la aplicación, es hora de comenzar el diseño de la estructura de todo el sistema.

Para cubrir todos estos frentes abiertos, serán necesarios tres subsistemas distintos. En cierto modo, todos estarán relacionados para conseguir los objetivos marcados en secciones anteriores, pero son lo suficientemente característicos para considerarse independientes.

El primero de todos ellos se centra en la obligación de proporcionar la información que necesite el usuario. Para ello, este subsistema de datos debe capturar toda información que el sistema operativo nos vaya proporcionando con sus avisos en forma de Intents.

De esta forma, cada vez que se modifique el estado del terminal, Android nos remitirá esa información, y nos encargaremos de almacenarla para que el usuario pueda consultar en todo momento el histórico de sus valores.

Además, será de especial utilidad para saber en todo momento el estado de esas variables, y conocer con exactitud cómo está el terminal en esos momentos. Estos datos serán utilizados para el segundo subsistema, que depende del primero para su funcionamiento.

Esta segunda parte se encargará de analizar si los planes de ahorro

definidos por el usuario se cumplen o no, es decir, que todas sus condiciones o activadores den un resultado cierto al compararlos con el estado actual del dispositivo. Entonces, al cumplirse este perfil, se ejecutaría su lista de acciones, pudiendo entonces modificar los niveles de sonido, el estado de las redes inalámbricas, etcétera.

Dicho subsistema dependerá de los datos del primero, ya que necesita comparar las precondiciones de los perfiles con los últimos datos recibidos de aquellas variables que puedan causar su activación. Por ello, durante el desarrollo e implementación, el subsistema de datos será el primero en ser creado, pues de su correcto funcionamiento depende el subsistema de perfiles. Esto podría afectar ligeramente al ciclo de vida del proyecto, ya que para poder efectuar un diseño correcto del segundo subsistema, sería conveniente, aunque no obligatorio, tener implementado el primero.

El tercero de todos ellos tratará de gestionar las conexiones inalámbricas para encontrar la mejor red posible, aunque difiera en criterio con el sistema operativo. Este suele elegir una red y no modificar su acceso incluso aunque no se disponga de conexión a internet. El subsistema deberá proporcionar al usuario herramientas sobre estos controles y la posibilidad de alternar ciertas configuraciones con menos limitaciones que Android. Se trata de un subsistema más independiente de los anteriores, aunque en cierto modo también hará uso de las nuevas informaciones relativas a la conexión que reciba el subsistema de datos, y por supuesto, estará regido por el comportamiento del subsistema de perfiles, ya que posee control sobre el sistema de encendido de dichas redes.

Durante esta fase de diseño, se definirán los procedimientos a seguir con mayor detalle que el ofrecido en el proyecto formativo ToDoBars. En algunos casos, se simplificarán imágenes y explicaciones relativas a la interfaz gráfica, la cual suele requerir muchas clases y estructuras que no están estrictamente relacionadas con el funcionamiento de la aplicación. Serán descritas de forma genérica o en conjunto, pero no se entrará en detalle de cada una de ellas, ya que sus comportamientos son análogos.

Dicho esto, pasamos al diseño del primer subsistema.

7.1– Subsistema de datos

Para recibir todos los datos que necesitamos, primero hay que investigar todo lo relativo a Android y sus sistemas de avisos frente a cambios y nuevos eventos. Existe buena bibliografía al respecto, especialmente la

oficial de Android en *Android developers* [9], pero también ciertos libros [10] [11], que nos explican con todo lujo de detalles cómo funcionan estos paquetes de información.

En concreto, existen algunos Intents que encajan a la perfección con las necesidades de la aplicación, como *ACTION BATTERY CHANGED* con datos de batería o *ACTION SCREEN OFF* cuando la pantalla se apaga. Necesitaremos pues una forma de capturar estos mensajes, procesarlos, realizar las acciones pertinentes y almacenar los datos que contengan.

La forma adecuada de capturar esos mensajes pasa por crear una clase propia que herede de BroadcastReceiver, proporcionada por Android como base para construir nuevos receptores. Esta clase, además de sobreescribir el método que trata el Intent que va a recibir, debe especificarle de alguna forma al sistema operativo qué datos necesita.

Para conseguirlo, tenemos dos opciones. La primera es la clásica y debe usarse siempre que sea posible, ya que es una forma más limpia y apropiada de realizarlo. Se trata de añadir la clase como un Intent Filter dentro del *Manifest* de la aplicación. El segundo método consiste en crear una instancia del receptor desde un Service o Activity, pero quedaría sujeto a su existencia y su ciclo de vida. En ocasiones, será obligatorio el uso del segundo método, ya que algunos Intent poseen protecciones especiales.

Supongamos que tuvieramos ya construidos dichos receptores. Necesitamos algún punto donde volcar y tratar esta información.

En el caso del proyecto previo ToDoBars, fue posible construir la aplicación alrededor de una clase Activity que cargaba o guardaba sus datos cuando se abría o cerraba respectivamente. Como no necesitaba ningún tipo de información extra, ni monitorización, ni realizaba código en segundo plano (excepto las alarmas, pero de eso se encarga AlarmManager de Android), no era necesaria la existencia de un Service.

Sin embargo, en el proyecto que nos ocupa, va a ser una pieza fundamental de todo el sistema. Usaremos esta nueva clase denominada *DataService* como centro neurálgico desde el cual se enviará y recibirá toda esta información. DataService controlará pues el subsistema de datos, desempaquetándolos y almacenándolos debidamente, pero también tendrá un papel clave en el resto de subsistemas, aunque no se tratará en esta sección.

Con respecto a recibir y guardar los datos, para crear código fácil de modificar y mantener, no es buena idea dejarle toda la tarea al DataService. Sería necesario pues crear un camino de envíos de información en los

que el Service haga de intermediario pero no reuna en una misma clase todo el código de tratamiento de los paquetes. Este procesamiento del final de la cadena se debería realizar en un tipo de clases que cumplan ciertos requisitos que DataService pueda utilizar y crear según necesite, pero con el mismo tipo de accesos y recursos, sin que conozca sus funcionamientos internos.

La mejor forma de diseñar tal eventualidad, es creando una clase madre de la que heredarán todas aquellas que deban encargarse de procesar y guardar la información. A ésta clase le pediremos que sea capaz de desmontar un paquete de información, que será un Bundle derivado de los datos que traiga el Intent, y almacenarlos.

Al mismo tiempo, la naturaleza de estos datos puede ser muy diversa. Por ejemplo, en el Intent descrito anteriormente con el nombre de *Action Battery Changed* nos llegan multitud de datos, ya sea el nivel de la batería, su temperatura, su estado, etcétera. Si volvemos a caer en el mismo error de ejecutar todo este código en un mismo lugar, podría volverse ciertamente engorroso cualquier intento de modificación futura, con lo que repetiremos el mismo esquema, y cada una de las clases que debe procesar los paquetes, tendrá otras inferiores que serán el receptáculo final de cada uno de los valores que trae. La mejor forma de visualizar tal diseño, es con un diagrama de clases que lo muestre adecuadamente.

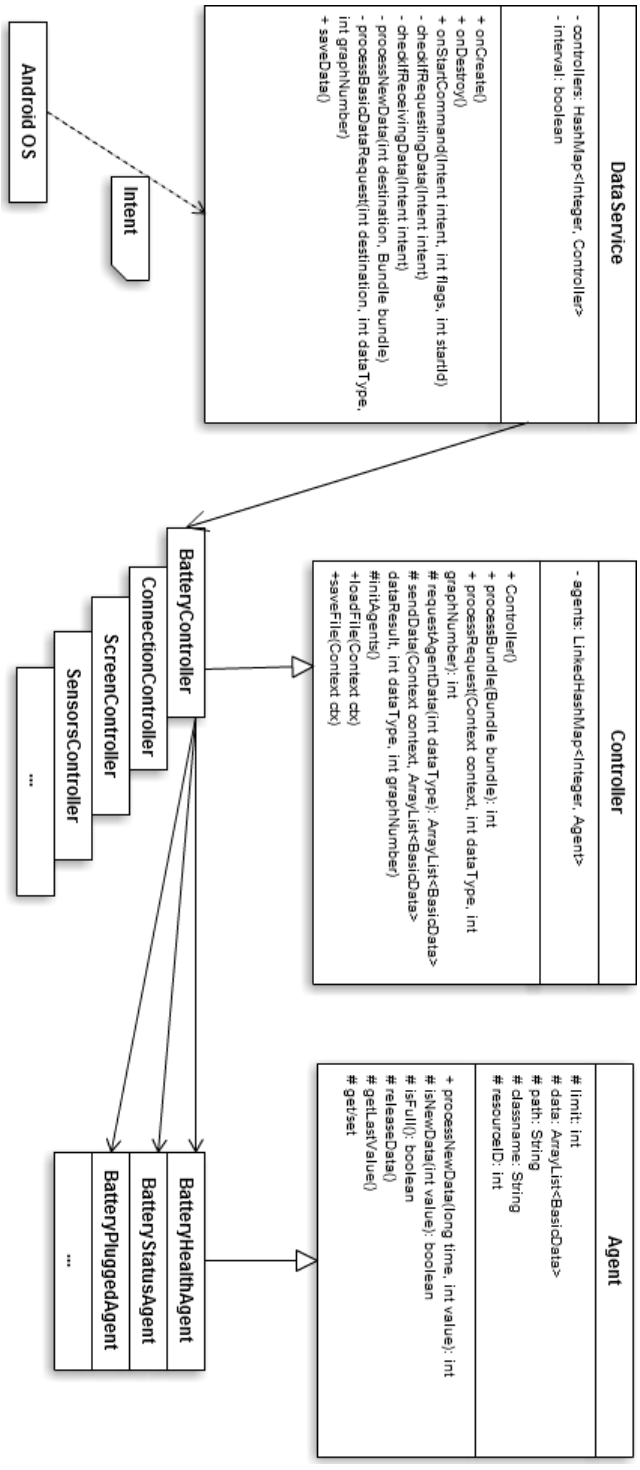

Figura 7.1: Diagrama de clases del subsistema de datos

El sistema funcionará de la siguiente forma, explicada con más detalles y con nombres específicos para las clases:

- Primero, llega un Intent de alguno de los tipos concretos que hemos avisado a Android que necesitamos, ya sea comunicándoselo mediante el *Manifest*, o creando en tiempo de ejecución un IntentFilter. Estos BroadcastReceiver personalizados obtendrán la información, y si ésta no está empaquetada en forma de Bundle, se transforma en uno y se envía con un Intent al DataService. Este Intent posee un identificador del Controller al que va dirigido y que debe procesarlo.
- Entonces llega al DataService. Este comprueba el identificador, y consulta en su lista de Controllers (que han debido crearse junto al Service) si posee alguno con ese ID de destino. Si existe, llama a su método *processBundle* que ha debido heredar e implementar de su clase madre Controller. Si no existe un controlador, ignora el dato recibido.
- Una vez dentro del método *processBundle*, se obtienen y separan todos los valores de aquellas variables que nos interesan, aunque ya han sido filtradas por el Receiver correspondiente.
- Cada uno de los Controllers, posee uno o varios *Agent* o *agente* que se encargarán de controlar cada uno una variable y de almacenarla en memoria si le llegan señales de que el Service se debe cerrar. Cada uno de ellos tiene un identificador al que acceder cuando se necesita la variable correspondiente.

Con este sistema, todo cambio que se produzca en el terminal quedará almacenado en las estructuras correspondientes definidas en el diagrama de clases. Los últimos valores de las variables conformarán a su vez un *estado actual* que podría ser de utilidad para el siguiente subsistema.

7.2– Subsistema de perfiles

Pasamos ahora a realizar el diseño del subsistema de perfiles. Esta sección del proyecto será la más importante de todas, ya que de ella depende el cumplimiento de los objetivos del mismo. Como se ha ido describiendo durante la fase de análisis de requisitos, el concepto de ejecución de los perfiles ha pasado por varias etapas. En un primer momento, se pretendía conseguir que cada perfil coloque al terminal en un determinado estado que definiera el usuario, con gran cantidad de variables a modificar.

Por ejemplo, un perfil que se activase a una hora, cuando se bajase de cierto nivel de batería, definiría el estado al que debe cambiar el terminal, con valores para cada una de las redes inalámbricas y otras configuraciones.

Esta idea, aunque a nivel de interfaz pareciese razonable, suponía grandes problemas en su ejecución. La solución fue entonces limitar estos estados a simples acciones sin repetición que cada perfil podría ejecutar si se activa. De esta forma, no es necesario definir cómo va a quedar el terminal en su completitud, sino simplemente aquellas variables que nos interesen.

Aún así, este sistema da mucho que pensar antes de poder ponerse en práctica. Se debe tener especial cuidado en el sistema de condiciones, o podrían darse casos de condiciones reiterativas. Un ejemplo: el perfil 1 se activa cuando el receptor Wi-Fi está activo y lo apaga. Luego el perfil 2, que se activa cuando el receptor Wi-Fi esté inactivo, se ejecuta y lo enciende. Así una y otra vez.

Un procedimiento de ese tipo sería inviable, con lo que es conveniente antes incluso de la fase de diseño real de este subsistema tener claro qué necesitamos y qué queremos conseguir. Entonces, para implementar dichos perfiles se ha decidido usar dos tipos de datos distintos.

El primer tipo son aquellas variables del subsistema de datos que puedan ofrecer algún dato de interés para activar perfiles, como el nivel de batería, o si el terminal está enchufado. Estas estructuras las denominaremos *activadores* o a veces con el término inglés *triggers*. Cada uno de ellos, de cara a la interfaz de usuario, tendrá un número identificador, un operador en los casos de variables contínuas, y un valor con el que comparar el estado actual. Con estos campos, podemos definir por ejemplo un activador con identificador de nivel de bateria, operador *menor que*, y valor 50 si queremos que se active cuando baje el indicador del cincuenta por ciento.

Una vez se cumpliesen todas las condiciones de activación, estuviese el perfil activo, y cumpliese los requisitos de hora si los hubiere, entonces pasamos a ejecutar las acciones o *actions*. Estas acciones tienen una estructura similar a los activadores, pero con ciertas modificaciones. Las acciones también poseen una variable identificadora con utilidad para casos especiales, y un valor, que será el nuevo estado que colocarle a la variable. Cada acción, también estará obligada a implementar un método que la ejecute, si es que puede hacerse desde su código.

En caso contrario, el DataService reconocerá mediante el ID de la

acción que no puede ejecutarse directamente y que es su deber notificárselo al sistema adecuado. Estas limitaciones serán descritas con mayor detalle en la fase de implementación, pero en resumen son debidas a que ciertos datos sólo son accesibles desde un Service, y a que ciertas acciones o modos especiales necesitan mantenerse durante un tiempo, y el carácter volátil de una acción no lo permitirían.

Todo esto se ve mucho mejor ilustrado con un diagrama de clases del subsistema. Se representará de forma ligeramente simplificada, evitando constructores redundantes, o importaciones y herencia de clases básicas de Java y Android.

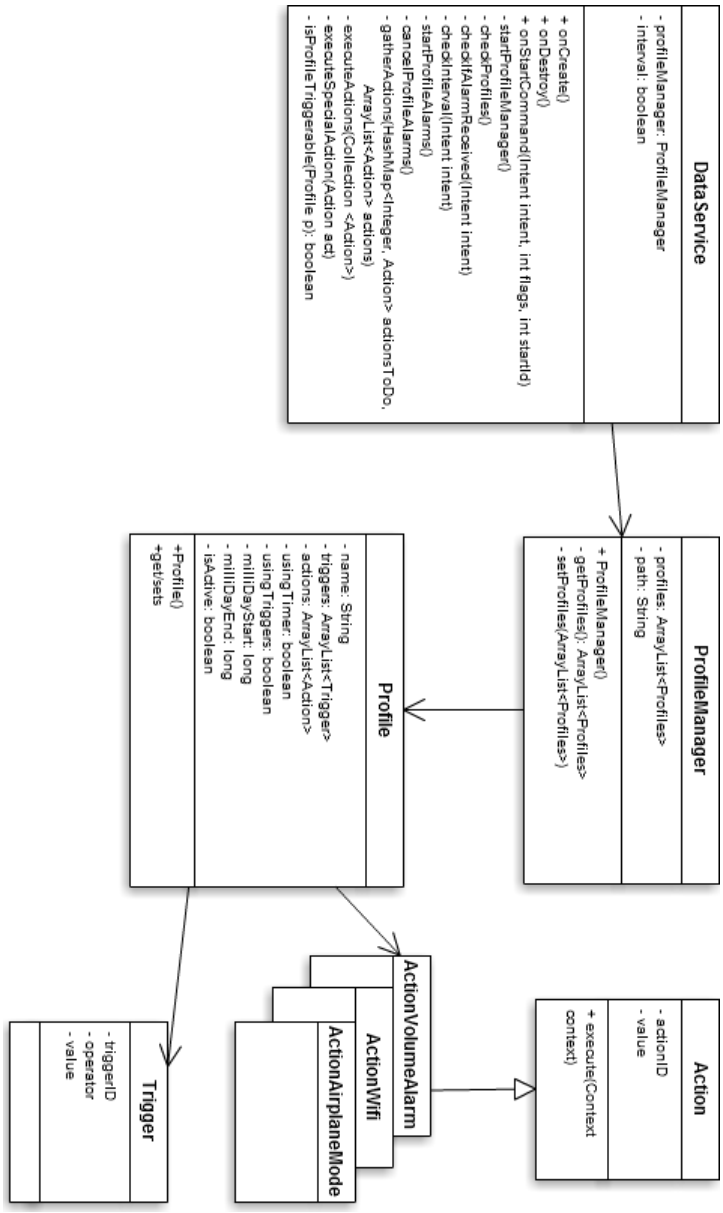

Figura 7.2: Diagrama de clases del subsistema de perfiles

Según el diseño, los actions deben implementar el método execute() de Action, y estos y los triggers deben tener ciertos atributos que nos permitan ejecutarlos y evaluarlos respectivamente. Un perfil o *profile* tendrá entonces una lista para cada tipo, además de ciertas variables que permitan conocer si el usuario permite su activación, o si se debe ejecutar únicamente a ciertas horas.

Todos los perfiles serán gestionados por una clase denominada ProfileManager que se encargará de reunir todos los métodos necesarios para su uso. Tendrá la lista de perfiles y se encargará de almacenarlos en disco cuando sea necesario.

Por otra parte, vuelve a aparecer DataService como núcleo de la aplicación. En éste caso tendrá una instancia de ProfileManager para gestionar todos estos perfiles, pero será el Service el que resuelva los conflictos que pudieran aparecer.

Aunque en esta fase de diseño no se entrará en mucho detalle al respecto, existirán entonces en DataService diversos algoritmos encargados de evitar conflictos entre varios perfiles que se activen a la vez, y escoger aquellos comandos o acciones con más prioridad, permitiendo una mezcla de las que no coincidan.

Además, es necesario preparar al DataService para posibles consultas y modificaciones de los perfiles que pudieran hacerse desde *MainActivity*. La interfaz de usuario, que en su segunda pestaña permitirá editarlos y comprobarlos, abrirá una nueva Activity llamada *EditProfileActivity* y se mantendrá en espera de posibles respuestas, que se producirían si hay nueva información al respecto. Si el usuario no ha cambiado nada, esta nueva interfaz no devolvería nada. Por supuesto, todo este intercambio de información se realizará mediante Intents, ya que es el mejor sistema que nos proporciona Android para comunicar dos elementos aparentemente independientes (aunque forman parte del mismo proceso, no comparten la mayoría de los datos), y por ello se enviarán los perfiles por este medio.

Con esto completamos el diseño del subsistema de perfiles, que será ampliamente detallado en la sección de implementación, especialmente aquella información relativa a la resolución de conflictos entre perfiles, y a la ejecución de modos especiales por mediación de DataService.

7.3– Subsistema de redes

El subsistema de redes se trata de una adición de menor envergadura que las anteriores. Aunque su funcionamiento puede ser muy útil de cara a aumentar la conectividad y opciones de control del usuario con respecto a las conexiones inalámbricas, es posible que un buen número de usuarios no las necesite a priori.

Probablemente la mayoría de los usuarios de Android no posee formación detallada sobre los sistemas de conexiones a internet. Entonces, aunque pudieran conocer términos como *dirección IP*, o *servidor DNS* debido a ciertas ocasiones en las que pierden conexión a la red, es difícil que a la mayoría de los mismos les interese modificar la configuración de Android al respecto.

Todo este sistema tiene la meta de cubrir dos necesidades o problemas que se dan al menos, en versiones cercanas a Froyo (2.2) y GingerBread (2.3). En ellas, no se pueden definir por ejemplo dos configuraciones IP estáticas distintas. Normalmente los usuarios dejarían esta opcion deshabilitada y en la mayoría de los casos dispondrían de internet gracias a los automatismos permitidos por el protocolo *DHCP*. Sin embargo, en algunos sitios esta opción no está disponible, con lo que se hace necesario establecer una IP fija. En esos casos, se vuelve irritante modificar la configuración una y otra vez si se desean opciones distintas para cada punto de acceso, o si algunos puntos la necesitan y otros no.

Además, aunque la mayoría de los usuarios se conectan normalmente a una sóla red Wi-Fi en su hogar, trabajo u otro sitio, cierto porcentaje podría necesitar unas mejores conexiones entre dos o más enrutadores situados en diversos puntos de dichos lugares. Algunos de ellos podrían tener errores en la conexión, ya sea de forma física o lógica y carecer de acceso a internet, a pesar de tener mayor potencia inalámbrica. O poseer menor potencia, no tener conexión, pero ser la última red a la que se conectó el terminal.

Es en esos casos cuando Android decide conectarse la última red enlazada (que podría ser de otro piso, u otra habitación, u otra ala del edificio) independientemente de su potencia o capacidad de acceso a internet. Realizar entonces una búsqueda de otras redes más apropiadas o de las que tengamos conocimiento de su mayor efectividad sería muy útil.

Para implementar estas mejoras diseñaremos unos parámetros parecidos al sistema de perfiles. En este caso poseeremos una lista de configura-

ciones extras que dependerán de las redes inalámbricas ya conocidas. Para añadir una configuración extra nueva, escogeremos un SSID o identificador de una de esas redes, y estableceremos los parámetros de conexión. Entre ellos, la opción de habilitar o no direccionamiento IP estático, los campos que necesitaría para tal conexión (dirección IP, puerta de enlace, máscara de subred, dns, etcétera) así como otras opciones de preferencia o fiabilidad que nos puedan ayudar a cambiar entre redes con un criterio definido por el usuario.

Se trata entonces de un subsistema más simple en su diseño que en su ejecución, ya que sólo necesitaremos alguna clase que gestione estas configuraciones, y el DataService que las pondrá en práctica cuando sea necesario. Por ello, el diagrama de clases y de funcionamiento es más sencillo de visualizar, y no necesita más detalles de los proporcionados. Será en la fase de implementación cuando aumente su dificultad debido a las conexiones con las interfaces de usuario, que ampliarán la cantidad de clases necesarias y las líneas de código del proyecto al completo.

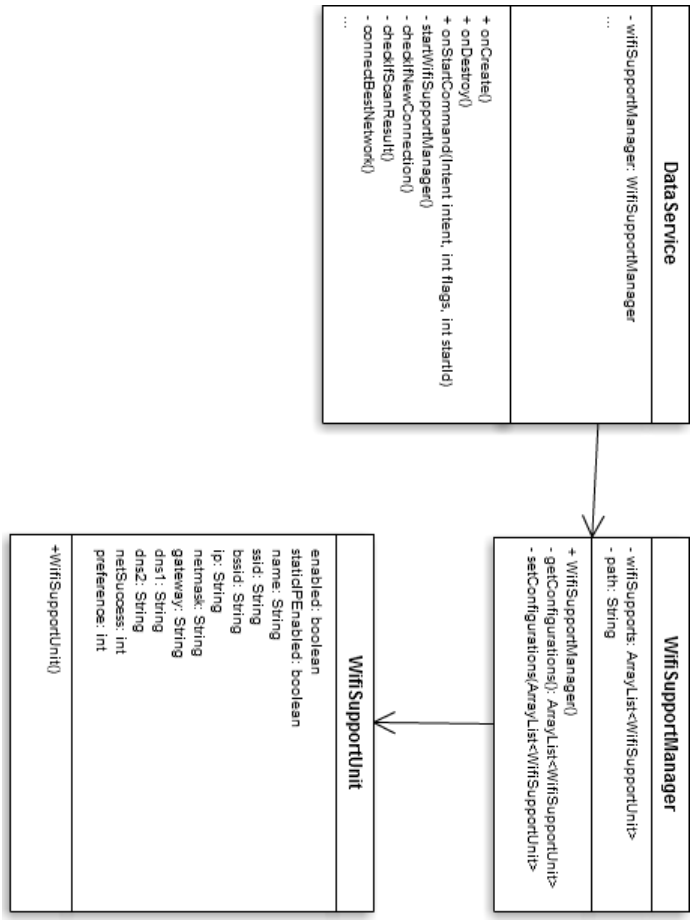

Figura 7.3: Diagrama de clases del subsistema de redes

CAPÍTULO 8

Commandroid: Implementación

Con los prototipos de interfaz realizados y el diseño de los distintos aspectos de la aplicación, se procede pues a la fase de implementación.

Aunque en el capítulo anterior hemos descrito el funcionamiento general que debe tener el sistema, y sus principales características, en esta sección el detalle y cantidad de información será mayor, sobretodo debido a la incorporación de la interfaz en todos ellos.

Además de las secciones dedicadas a cada subsistema, de implementaciones ligeramente más complejas que las esperadas, este capítulo dispone de dos secciones nuevas relativas a la interfaz. La primera de ellas se sitúa a continuación y tratará sobre los primeros aspectos relativos a la interfaz y a su apariencia. Luego se situarán las secciones de los subsistemas y los problemas encontrados, con especiales aclaraciones sobre la unión entre código e interfaz.

8.1– Estructura inicial de la interfaz de usuario

Cada botón, pestaña o menú desplegable que se añada a la aplicación necesitará varias cosas. Primero, debemos crearlo de alguna de las formas que nos proporciona Android. El esquema clásico es producir una instancia del objeto desde el código en tiempo de ejecución y manejarlo como un objeto normal del lenguaje, sin embargo, el sistema operativo nos proporciona una estructura más fácil de modificar y sencilla de mantener en

forma de XML.

Cada uno de los archivos XML que destinemos a la interfaz se situarán en la carpeta *layout* del proyecto, y contendrán los elementos que queremos incluir. Mediante las múltiples herramientas y estructuras que se proporcionan, escogemos una que englobe a todo el archivo, y vamos insertando poco a poco nodos, respetando siempre la estructura de árbol. Con los controles de Layouts podemos definir relaciones entre ellos, colocandolos al borde de la pantalla, unos al lado de otros, o incluso agrupándolos con propiedades especiales (ViewGroups) que incluyan controles táctiles de todo tipo.

Para observar el formato de estos archivos, usaremos como ejemplo uno de los archivos del proyecto, pero eliminando todas aquellas opciones que pudieran confundir al lector o dificultar la lectura del documento.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
       android"
3      android:layout_width="fill_parent"
4      android:layout_height="fill_parent" >
5
6        <TextView
7            android:id="@+id/actionname"
8              android:layout_width="wrap_content"
9              android:layout_height="wrap_content"
10             android:layout_alignParentLeft="true"
11             android:layout_alignParentTop="true">
12
13     </TextView>
14
15       <TextView
16         android:id="@+id/actionvalue"
17             android:layout_width="wrap_content"
18             android:layout_height="wrap_content">
19         </TextView>
20         <ImageView
21           android:id="@+id/deleteactionicon"
22             android:layout_width="wrap_content"
23             android:layout_height="wrap_content"
24             android:layout_marginLeft="5dp">
25     </ImageView>
26
27
28 </RelativeLayout>
```

Código 8.1: Esquema de un archivo XML de interfaz

Si el lector desea en cualquier momento de la documentación consultar el código de la aplicación, puede dirigirse a la forja RedIris donde figura este proyecto. Estos archivos de interfaz se sitúan en el directorio */res/-layouts/*, y otros similares pero enfocados a estilo gráfico se colocan en el directorio /res/drawables/.

Al estilo de los lenguajes de marcas que pueblan la red desde sus inicios, se deben respetar las reglas de apertura y cierre de las mismas. Una marca se puede abrir y cerrar en el mismo momento, pero si se dejase abierta, debe existir una posición de cierre. Tampoco se puede cerrar alguna marca cuyos elementos hijos o sucesores no hayan sido cerrados previamente.

```
1 <A>           Apertura.
2 </A>          Cierre.
3 <A />         Apertura y cierre en una sola marca.
4 <A></A> <B></B> Correcto.
5 <A><B></A></B> Incorrecto.
```

Código 8.2: Reglas básicas de un archivo XML o HTML

Además de todas estas reglas, el sistema xml nos permite asignarles atributos a cada una de las marcas. En el fragmento de código real anterior se muestran tres variables indispensables para cada elemento de la interfaz. Dos de ellas son esenciales y casi obligatorias, y determinan el tamaño del elemento. Son *android layout width* y *android layout height* y usándolas podemos colocarles un tamaño numérico y una unidad de medida, heredar el tamaño del elemento padre, o asignarles el mínimo posible para que sean visibles.

La tercera propiedad es el identificador, y nos permitirá recuperarlos desde la sección de código, para añadirles nuevas propiedades o enlazarlos con otros elementos para darles carácter táctil, por ejemplo.

Con estas estructuras nos ahorramos una tremenda cantidad de código redundante en los archivos fuente del proyecto, y los externalizamos y separamos del esqueleto del mismo. De esta forma, si deseamos cambiar el aspecto de la aplicación, las modificaciones a realizar son muchísimo más sencillas.

Un ejemplo de acceso a estos archivos sería el siguiente:

```
1     this.setContentView(R.layout.main);
2     TextView titulo = (TextView) this.findViewById(R.id.titulo);
3     titulo.setText("Probando!");
```

Código 8.3: Acceso desde el código a los ficheros de interfaz

En este acceso, primero le especificamos al sistema que nuestro archivo de interfaz general para esta Activity es *main.xml*. De ahí, Android obtiene la estructura de los elementos, su apariencia y sus identificadores. Si queremos recuperar o acceder a una simple caja de texto con identificador *titulo*, sólo tenemos que llamar al método *findViewById* que nos lo devolverá si no nos hemos equivocado de nombre. Una vez conseguido, podemos modificarle los mismos valores que posee en el XML, otros nuevos, o simplemente enlazar su comportamiento con otras clases de nuestra creación.

Para visualizar en todo momento estas estructuras e interfaces, el entorno de desarrollo Eclipse y el SDK de Android nos proporciona una ventana en la que comprobar los cambios de forma instantánea. Se trata de una aproximación, con lo que es recomendable siempre ponerlos a prueba en un terminal para mayor seguridad.

Gracias a estas herramientas, durante el desarrollo de los siguientes tres subsistemas se complementarán las estructuras de datos, los algoritmos y las interfaces creadas, que poco a poco ganarán en utilidad y apariencia.

8.2– Subsistema de datos

Tal y como se describió en la fase de diseño, el subsistema de datos, aunque parece un mero añadido para informar al usuario, también es vital para poder crear el subsistema de perfiles, con lo que se convierte en primera prioridad para ser implementado.

Durante aquella etapa se realizó una investigación exhaustiva sobre los Intents que Android envía periódicamente y que nos podrían resultar de interés. Tras una selección de todos los posibles, los mejores para capturar en primera instancia parecen ser *Action Battery Changed* para los cambios de la batería, *Action Screen On* y *Action Screen Off* para el bloqueo de pantalla, y *Action Connectivity Change*.

Para capturarlos, algunos simplemente necesitan figurar en el *Manifest*, pero otros, debido a protecciones especiales deben ser capturados por Receivers creados en tiempo de ejecución. En cualquiera de los casos, sean ejecutados de una forma o de otra, estos receptores deben poseer su respectiva clase.

Técnicamente podríamos reunirlos todos en una clase, y que esta sea el punto de entrada de todos los Intent que recibamos de Android, pero

incluso si separáramos los Receivers especiales de los que se colocan en el *Manifest*, la recepción y tratamiento sería más engorroso de lo necesario, y las consecuencias en DataService serían las mismas. También sería posible recibirlo todo en DataService y que figure como entrada de estos datos, pero sin embargo aumentaría aún más sus funciones y su código, y perderíamos la cómoda posibilidad de que cada Receiver empaquete los datos de una forma concreta (sólo con lo necesario) para que DataService los distribuya sin apenas *consultarlos*.

Por ello, pasamos a crear las clases que necesitamos para tal recepción. Por ahora serán tres y heredarán de BroadcastReceiver, pero podrían aparecer más en un futuro de forma sencilla, si necesitamos más información que mostrarle al usuario. El esquema de su acción sería algo como:

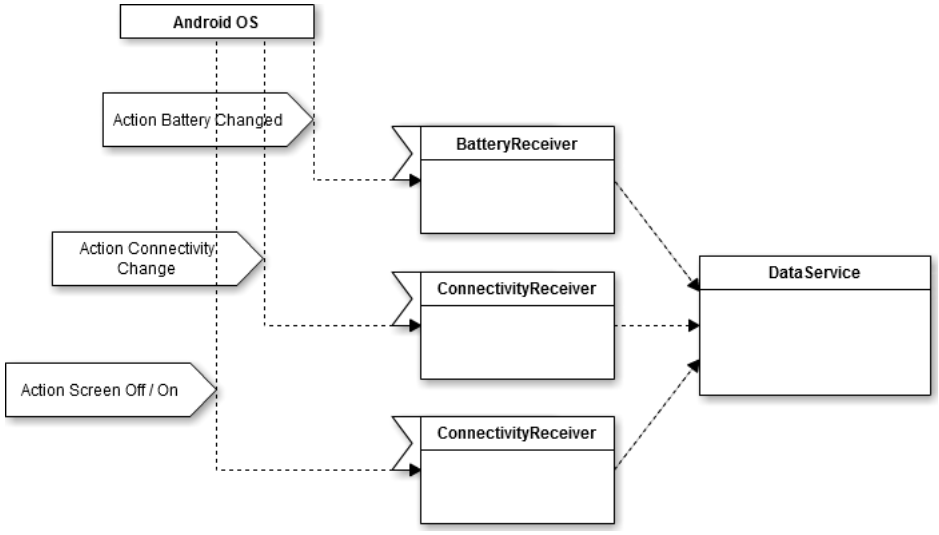

Figura 8.1: Esquema de los receptores de información.

Cada uno de ellos procesa los datos, ignora aquellos que no interesen, y los envía a DataService en un nuevo Intent junto con un número que identifica al nuevo receptor. Por ejemplo, *ConnectivityReceiver* desecharía algunos datos de conexión sobrantes y enviaría a DataService un Intent con un identificador concreto para que la información se procese por la clase correcta.

Con este sistema, DataService no necesita saber qué contiene cada uno de los Intents que recibe, si no que aquellos que ya tengan destinatario son redirigidos automáticamente a los responsables de almacenarlo. Además, si es de algún tipo importante como los eventos de pantalla, modificaría sus variables para comenzar a trabajar en el sistema de perfiles o no.

Como ya vimos en la fase de diseño, los responsables para el procesado de datos son los denominados Controllers. Cada uno de ellos está identificado por un número y posee los métodos necesarios (algunos heredados de la clase madre) para dividir un paquete concreto de información en las variables adecuadas. Esta clase contendrá también una lista de *Agents* que se encargarán de almacenar esos valores. Cada Agent se encarga de una variable distinta y pueden llamar a una clase estática externa FileManager cuando DataService les ordene almacenarse en un archivo.

Con este sistema implementado, ya tendríamos una forma de capturar toda esa información, pero no disponemos de estructuras ni interfaces que nos permitan mostrarla al usuario. Para ello, se hace necesario crear la primera y cuarta pestaña del prototipo de la interfaz.

Como primer paso, es evidente la necesidad de una clase Activity. Aunque se ha descrito anteriormente, es ahora cuando la denominada clase MainActivity entra en acción. Tendrá, por ahora, métodos típicos para cumplir con los requisitos de ciclo de vida que Android impone. Al contrario que en el proyecto formativo ToDoBars, esta vez no se ejecutará ningún código en esta clase.

En aquel proyecto, no era estrictamente necesario un Service o *Servicio* ya que el código de la aplicación, excepto las alarmas de las que se encargaba Android, nunca necesitaba ejecutarse en segundo plano. Por ello, la clase Activity principal que se usó estaba más cargada de código de lo habitual, aún más si tenemos en cuenta la inexperiencia sobre el sistema.

Como resultado, aquella clase principal englobaba tantas operaciones que sus líneas de código aumentaban sin cesar, convirtiéndola en un elemento ligeramente inestable y difícil de mantener. Este caso guarda muchas similaridades con el antipatrón de desarrollo *The Blob*[12], que inten-

taremos evitar en este nuevo proyecto.

Esta vez se evitará colocar en la Activity todo el código que se pueda delegar en clases relacionadas. Sólo se utilizarán algunas variables de estado, elementos gráficos y sus controles táctiles.

Entre estos elementos visuales se encuentra el sistema de cuatro pestañas que se utilizó en el proyecto anterior. Siguiendo con las pautas descritas, en esta ocasión se creará una clase externa denominada *MainBarClickListener* que al implementar onClickListener podrá controlar las pulsaciones producidas en las pestañas y cambiar de vista gracias a un nuevo ViewFlipper.

Para que todos estos elementos sean de fácil acceso y modificación, se creará un sistema de archivos que no almacenen muchas marcas cada uno. El primero de todos ellos tendrá toda la interfaz de las pestañas superiores, con las líneas de color justo debajo. Este archivo tendrá un enlace a otros cuatro, que serán las cuatro secciones que alternará el mencionado ViewFlipper.

Además, como algunas de las pestañas poseen listas de elementos, y estas listas poseerán ArrayAdapters que las vincularán con sus gráficos correspondientes, se creará un archivo por cada lista que defina como se visualiza cada elemento. En los dos subsistemas involucrados se entrará en mayor detalle.

Entonces, con este sistema de pestañas, de organización de archivos, y con cuidado de no ejecutar apenas código en la Activity, pasamos a crear las dos pestañas involucradas en el subsistema de datos, la primera y la última.

Tras crear los archivos de interfaz y los elementos visuales que los definen, se hace patente la necesidad de un método de consulta de estos datos. Si hacemos memoria, es el DataService el que posee los datos, y ahora nos situamos desarrollando botones y controles desde Activity. Puede parecer que estamos en un mismo rango de acción, pero no es así. Cada Activity y cada Service no pueden comunicarse fácilmente entre sí excepto por elementos estáticos, compartición de datos mediante permanencia de información (bases de datos o archivos) o alguno de los sistemas que proporciona android (Content Providers e Intents). Como ya se están utilizando en gran medida los paquetes Intent y existe una cierta familiaridad con dicha clase, será la opción más directa.

Si queremos entonces que los controles de interfaz se actualicen con la

información necesaria, tenemos varias opciones. O realizamos el pedido de información a DataService mediante Intents, o directamente los capturamos desde la Activity con algún Receiver.

En cierto modo, usaremos ambos. El primero se utilizará para la cuarta pestaña, donde el usuario debe especificar qué datos debe consultar, y no nos bastará con el último dato que nos proporcione Android, sino que debemos mostrar un histórico de valores.

Para obtener esa información, le enviamos un Intent con ciertos parámetros identificadores para que sepa que la petición va dirigida hacia él, que se trata de una petición de datos y no de un envío, y además qué tipo de datos queremos. Estos tipos de datos son los que se han ido almacenando durante todo el subsistema de datos, y es ahora cuando la petición y el identificador se ve más claro. No sólo se utilizaron para almacenarlos, si no para consultarlos por mediación de DataService.

En la primera versión de la aplicación, se ha decidido usar el sistema de permanencia de datos Serializable de Java, pero en un futuro cercano es extremadamente recomendable el aprendizaje específico de SQLLite y su uso desde Android. Se ha pospuesto dicha opción para no comprometer el desarrollo temporalmente ni en cantidad y calidad de código, y concentrar todo el aprendizaje en el SDK de Android, pero es una tecnología muy interesante, especialmente para el subsistema en el que nos encontramos.

Dicho esto, suponemos que llega la respuesta por parte de DataService. Es hora de mostrar la información al usuario. Implementar una librería para realizar este pintado en pantalla no parece la mejor idea, sobretodo porque estos pasos ya habrán sido recorridos por muchas personas. Es el momento de investigar y encontrar código de terceros que realice este servicio que aunque necesario para este desarrollo, no es parte directa de los objetivos a conseguir.

Tras cierta búsqueda se encuentran dos librerías que parecen cumplir con las necesidades de la aplicación. Una de ellas es *GraphView* [13], pero tras probarla y encontrar ciertos errores, se decide instalar *AChartEngine* [14], más compleja de utilizar pero más estable y con mejores resultados.

Tras crear un tipo de datos al que es necesario convertir el valor de las variables para mostrarlos por pantalla, se prepara a este sistema de gráficas para recibir los datos de un BroadCastReceiver que preparamos para la ocasión. Cuando los datos llegan casi al instante de haberlos pedido, el usuario pulsa el botón de generar gráfico y finalmente aparece. Las opciones de personalización en cuanto a colores y ejes nos permiten colo-

car botones de selección de estilo al usuario, y con los identificadores de las variables, obtener una descripción textual de aquellos valores discretos que posean.

Con esto ya tendríamos la cuarta pestaña, en la que hemos usado la petición mediante Intents.

El segundo sistema de petición de información a Android, en cambio, nos permite flexibilizar la interfaz de usuario de la primera pestaña y mostrar aquellos datos que, aunque no se almacenen, sean de interés para el usuario. Además, sólo mostrarán el estado actual del terminal, con lo que podemos evitarnos más consultas al DataService (que podrían tener retraso) y apuntamos un receptor en el Manifest o en el código de la aplicación que se encargue de recibirlos.

Una vez se reciben estos datos en la clase *ActivityBatteryReceiver*, se los mostramos al usuario de forma cómoda y gráfica, al estilo de ToDoBars con una ProgressBar en la pestaña principal, para que de un vistazo rápido pueda consultar el estado del terminal.

Con todos estos elementos, las pestañas primera y última deberían estar implementadas y listas para funcionar. En el periodo de pruebas se comentarán los posibles problemas encontrados, pero las explicaciones de funcionamiento a nivel de desarrollador terminan aquí. Si el lector desea consultar más información o el código de la aplicación (tanto para subsistemas como interfaces, etcetera), es bienvenido a acceder al mismo mediante los enlaces a la forja de RedIris que figurarán en la bibliografía.

8.3– Subsistema de perfiles

En la sección de perfiles seguiremos paso a paso las pautas marcadas en la fase de diseño, pero añadiendo el punto de vista de la creación de la interfaz y del papel que juega MainActivity.

La segunda pestaña, según el prototipado realizado, necesita desde el primer momento una lista de perfiles. Estos perfiles ya fueron definidos en etapas anteriores y en definitiva deben poseer atributos para reconocer su nombre, las horas de inicio y fin y una lista de causas y consecuencias.

Dicha clase *Profile* se implementará no sólo con los atributos ya definidos, sino también con métodos específicos para poder enviarlos dentro de Intents gracias a la clase *Parcelable*. Y en el caso del guardado de archivos,

también será necesario heredar de la clase Serializable de Java.

El mismo dilema se repite para los *Triggers* o activadores y *Actions* o acciones. Deben implementar la clase Serializable si han de ser guardados en memoria para que el usuario no pierda sus configuraciones, y también Parcelable para que MainActivity pueda pedir la lista de perfiles o retornarla una vez sea editada.

Como se puede observar, será necesario un sistema de petición y de recepción de los perfiles. Repetimos entonces las pautas desarrolladas con anterioridad, con métodos para recibir y enviar información desde DataService consultando al gestor de perfiles ProfileManager (que es el que se guarda en memoria en última instancia) y añadir un Receiver para MainActivity. Cada Intent tendrá unos códigos o identificadores para no recibir información errónea o distinta a lo requerido.

Volviendo al problema de la lista, y tras haber implementado las estructuras de datos, es necesario crear una clase que herede de ArrayAdapter. Con esta clase podremos pintar en pantalla cada uno de los perfiles con un aspecto concreto, mostrando por ejemplo su nombre, y una casilla indicando si están activos o no, con posibilidad de modificación directa por el usuario.

En el código, dicho adapter recibe el nombre de ProfileAdapter y nos mostrará también dos iconos para gestionar estos perfiles y cumplir con los requisitos funcionales de la aplicación. Uno nos permitirá editar el seleccionado y el otro borrarlo. Además, abajo dispondremos de un botón para añadir uno nuevo.

Con esto ya hemos cumplido con la gestión general de los perfiles, pero no con su creación y sus pormenores internos. Necesitamos una Activity que nos permita editar cada uno de ellos. Su nombre será *EditProfileActivity*.

Esta nueva entidad tendrá que estar especificada en el Manifest, para que pueda ser llamada desde MainActivity con el botón de añadir o con el de editar perfil. En el caso de modificar uno existente, en el Intent que se envía para ejecutar la segunda Activity se incluirá el perfil que se quiere editar. Si esta segunda interfaz acaba su ciclo de vida y no devuelve nada, es porque el usuario habría pulsado el botón BACK e ignorado los cambios. Si, en cambio, se adjunta un perfil (habiendo comprobado que posee nombre y al menos una acción), se deberá a que el usuario ha aceptado los cambios con el botón de añadir el perfil. Entonces se incluirá en la lista como uno nuevo o se modificará si ya existía.

EditProfileActivity deberá tener varias opciones para rellenar todos estos atributos. El control de selección de nombre es evidente y fácil de desarrollar, pero el resto de elementos requiere cierto procesado. Por ejemplo, los botones de selección de hora llamarán a Android para obtener esos datos (variando su aspecto de una versión a otra), pero una vez introducidos debemos comprobar que la fecha de finalización no sea inferior a la de inicio, por razones obvias.

También las listas de acciones y de activadores necesitarán sus respectivos ArrayAdapter para mostrarse de una forma gráfica u otra. De hecho, para estos casos se trata de una clase mucho más importante que para los perfiles, ya que poseerán diversos botones según el tipo de la variable.

Así, una variable numérica como el nivel de la batería requeriría de tres botones. Uno para cambiarla por otra variable, otro para el operador matemático a utilizar, y otro para el valor. Con esto damos lugar a condiciones del tipo *nivel de bateria, mayor o igual, 40 por ciento*. En el caso de variables discretas (encender, apagar, etcétera) sólo es posible usar el operador de asignación, y no aparecerá la selección de operador. En el caso de las acciones, no es necesario ningún operador, ya que no es una condición a comprobar.

Con esto completamos la zona de acción del usuario. Ya puede modificar perfiles, añadirles activadores o acciones y modificar otros parámetros como la hora o el nombre. Es el momento de implementar y detallar el código que se ejecuta en segundo plano, es decir, todo lo relativo a DataService y el subsistema de perfiles.

Cuando MainActivity pasa a suspensión o se cierra, envía los perfiles modificados a DataService. Este, en cuanto el usuario se ausente y se active la protección de la pantalla, recibirá un Intent del tipo Screen Off. Es entonces cuando comienza a funcionar.

Con el usuario ausente, según las horas definidas por los perfiles, o a cada cierto tiempo si existe nueva información del subsistema de datos, se efectuará un *choque* de perfiles.

Esta *competición* entre estos obtendrá como resultado una lista de acciones a realizar, mezclando las de aquellos perfiles que cumplen sus condiciones de activación.

La mencionada mezcla se realizará dando prioridad a las acciones de los perfiles activados por una hora concreta. Entre ellos, los de menor rango serán los primeros. Estos depositarán sus acciones en la lista. Luego

los de mayor rango, y así hasta que hayamos recorrido todos los perfiles horarios. Si existiera empate, se activarían primero aquellos con mayores causas cumplidas.

Más tarde se recorren los perfiles sin hora, de mayor a menor número de condiciones cumplidas. Todos deben cumplir todos sus requisitos de activación, pero tendrán prioridad aquellos que más requisitos tienen. Uno a uno siguen colocando acciones en el conjunto.

Durante todo el proceso, si queremos insertar alguna acción que ya esté en la lista, se ignorará su modificación, ya que proviene de un perfil de menor categoría y se continuará con los siguientes.

De esta forma, se obtiene una lista de acciones que ejecutar y el sistema se encarga de realizarlas. Por ejemplo, si tuviéramos activos un perfil con horario que cubre el momento actual, sin condiciones extra, y que activase la red Wi-Fi, y otro sin horario que la desactivase sin ninguna condición previa, la red se activaría. Más tarde, si vuelve a producirse un *choque de perfiles* con nuevos datos y el periodo de tiempo del primer perfil ha pasado, las acciones del segundo no competirán ya con las del primero, y en este caso, apagaría finalmente la red.

Algunas de estas acciones podrían requerir un tratamiento especial, y en vez de ejecutarse mediante su método execute(), serán redirigidas según su identificador a una clase que las gestione adecuadamente. Por ejemplo, las acciones sobre el Wi-Fi requieren un tratamiento especial para mantener las conexiones activas en entornos con ahorro de energía activado. Para ello, se ha creado una clase WifiActionManager que mantenga esas conexiones en activo o desactivadas según la ultima acción Wi-Fi que se ejecutó.

En cuanto a la implementación de la competición de perfiles, será necesario ordenar los perfiles según los criterios de prioridad por horario y por número de condiciones. Necesitaremos crear pues varios Comparator según nos hagan falta y usarlos para ordenar por dicha preferencia estas configuraciones.

En resumen, cada competición se produce cada cinco minutos si hay nuevos datos o si se recibe una alerta por hora de un perfil. Pero se deben cumplir dos requisitos. Primero, que el usuario no haya desbloqueado el móvil puesto que estaría usándolo y no debemos interferir en sus quehaceres. Y segundo, que hayamos recibido nuevos datos de conexión o batería que justifiquen una nueva evaluación. También se realizaría un primer *choque* si el usuario acaba de dejar el móvil, para ver qué modificaciones nos

corresponde realizar.

Con todas estas técnicas ya implementadas, el subsistema de perfiles está completo y funcional. Como posibles mejoras, habría que colocar filtros para evitar que acciones que pudieran entorpecerse entre sí (como el modo avión y activaciones de wifi), y nuevos modos de conexiones inalámbricas, si fuera posible.

De hecho, al inicio de éste documento figuraba como objetivo secundario y de menor importancia la posibilidad de controlar las redes móviles o *3G*. Durante el desarrollo se ha invertido gran cantidad de tiempo en investigar cómo realizar tal cambio, sin embargo, el SDK oficial de Android no permite controlar dichas redes, quizás, para proteger a los usuarios de posibles facturas de teléfono indeseadas por aplicaciones dañinas. Existen sin embargo formas de evitar ese control y de activarlas, pero necesitan intrincados métodos más cercanos a ingeniería inversa o similares que a un desarrollo estándar.

Queda entonces por comprobar si en el futuro tal acción puede realizarse o si existe alguna opción que Google vea adecuada para implementar tal funcionalidad. Por ahora, y como primera versión, lidiar con los principales problemas de las redes Wi-Fi y darle poder al usuario para evitarlos es una gran recompensa al esfuerzo de este desarrollo.

En cualquier caso, no será hasta la sección de conclusiones donde se evaluarán los éxitos conseguidos por la aplicación y sus mejoras futuras.

8.4– Subsistema de redes

La implementación del subsistema de redes es más sencilla que los anteriores. Al poseer también una lista de objetos, necesitará también un sistema para guardarlos, un Adapter que los muestre en pantalla, unos protocolos de intercambio de información entre la MainActivity y DataService, y en definitiva, la mayoría de opciones que ya teníamos implementadas para los perfiles.

Mucho de este código se puede reutilizar sin apenas cambios importantes. La lista es muy similar, ya que en definitiva se trata de configuraciones especiales para las redes Wi-Fi que se pueden activar o desactivar. Existe una salvedad y es que mostraremos también el nombre del SSID o identificador de la red.

También será necesario en éste caso crear una clase Activity que se encargue de editar estas configuraciones. Para este tipo de objetos necesitaremos más atributos que el subsistema anterior. Necesitamos campos de texto para el nombre y las configuraciones IP estáticas, botones para seleccionar la red, y algún elemento que nos permitan seleccionar la preferencia de conexión.

Volveremos a colocar abajo de la interfaz un botón para añadir el perfil si ha sido editado, añadiéndolo si el formato de las direcciones IP es correcto y si el perfil tiene nombre.

En cambio, por el lado del DataService si existirán algunos cambios. Esta vez podrá actuar tanto si el usuario está usando el terminal como si no, siempre que se le dé permiso. Para poder actuar, también serán necesarios nuevos Receivers (o reutilizar alguno anterior) que controlen las conexiones inalámbricas en dos casos posibles.

El primer caso se daría cuando el móvil se conecta automáticamente a un punto de acceso. Es el deber del subsistema comprobar si ese SSID y BSSID existen en la lista de configuraciones, y si el usuario lo dictaminó necesario, activar las configuraciones de IP estáticas necesarias.

El segundo caso se da cuando el terminal está aún buscando redes Wi-Fi a las que asociarse. Es entonces cuando entran en juego valores como la preferencia, si es que la red está en las configuraciones especiales de la aplicación, o la potencia de la señal. Quizás en futuras versiones, incluso, se podría crear una variable que mida el éxito de las conexiones a internet, para evitar aquellas redes que suelen dar problemas.

Con este sistema, si el usuario lo desea, las conexiones inalámbricas podrán disponer virtualmente de varias configuraciones estáticas de direccionamiento IP, distintas para cada punto de acceso. Esta herramienta podría ser útil para usuarios que en el hogar disponen de varios puntos de acceso o para lugares de trabajo con redes amplias y distintas configuraciones topológicas. En ciertos foros de desarrollo de Android se cuestionan estos problemas y otros similares de desconexiones de red con la terminología *Wi-Fi roaming* [15], y parece existir cierta necesidad de conexiones automáticas a los puntos más accesibles o preferidos.

CAPÍTULO 9

Commandroid: Pruebas

9.1– Pruebas iniciales. Android Virtual Device.

La herramienta conocida como *Android Virtual Device* o AVD se trata del principal sistema de pruebas que se ha usado durante no sólo este proyecto sino también del anterior.

Se trata, en esencia, de un simulador de un terminal al que podemos instalar en cualquier momento nuestra aplicación y comprobar cómo se ve y cómo funciona. Tiene multitud de opciones de configuración para simular tarjetas de memoria SD, memoria interna, o cualquier tipo de capacidad que un móvil actual tendría.

Aunque sus ventajas se hacen notar, cabe destacar que la velocidad no es su fuerte. Según la versión del SDK que poseamos puede que existan mejoras, pero por cada versión que avanza el sistema operativo, más lentitud se nota en los terminales virtuales que lo reproducen.

De todas formas, sigue siendo una herramienta excelente. En cualquier momento podemos activarlos y encenderlos como si los tuviéramos en la palma de la mano, y el arranque y la instalación de nuestras aplicaciones es sumamente sencillo. Es un sistema muy útil para todas aquellas pruebas de interfaz que necesitemos, no sólo para comprobar que todo funciona y no existen errores, también para consultar si los iconos y sus tamaños se mantienen en buenas proporciones en distintas resoluciones.

Además este sistema supone una primera línea de batalla contra cualquier error o *bug* que pudiera comprometer a toda la aplicación. Si por algún casual nos hemos dejado algún elemento sin inicializar, estamos accediendo erróneamente a un recurso, o en definitiva el desarrollo no es estable, podemos comprobarlo rápidamente con la unidad virtual y todos los recursos que ofrece. Estos recursos de depuración también están disponibles si conectamos por USB un móvil Android con driver compatible. Podemos efectuar capturas de pantalla, ver los procesos activos, o consultar el sistema de logs de información de nuestra aplicación y de todo el sistema, entre otras herramientas.

El proyecto que nos ocupa ha sido duramente probado en cada una de sus versiones y actualizaciones en varios terminales virtuales antes de su instalación en un teléfono real. Siempre que se ha encontrado un error ha bastado con buscar la causa, arreglarlo, y probar suerte de nuevo con suma velocidad.

Todos estos errores suelen ser despistes o modificaciones del código en secciones importantes cuyos cambios producen a su vez más problemas. En estos casos las interfaces mal inicializadas o mal accedidas por XML suelen ser las que fallan nada más ejecutar la aplicación, comprometiendo la estabilidad de la aplicación, y por ello son los primeros a eliminar desde la unidad virtual.

Sin embargo, existen otros problemas que no se pueden observar desde esta herramienta. Por ejemplo, tiene ciertas limitaciones con la simulación de redes inalámbricas, con lo que para el segundo proyecto ha sido especialmente necesario realizar su fase de pruebas en móviles reales. También algunos errores sólo se observan durante un uso prolongado, y como la unidad virtual depende del uso del ordenador en el que está instalado, normalmente no se pueden efectuar pruebas de largo recorrido con la misma fiabilidad.

Por ello, necesitamos realizar pruebas más intensivas, tanto en terminales reales como en asegurar el correcto funcionamiento de todas nuestras estructuras de datos.

9.2– Pruebas continuas, de rendimiento y unitarias.

Antes incluso de instalar la aplicación en los dispositivos virtuales, tenemos a nuestra disposición una herramienta ampliamente conocida para

la realización de pruebas unitarias, *JUnit*.

En cualquier momento que lo necesitemos, podemos hacer uso de la herramienta JUnit estándar de Java incluyéndola en nuestro proyecto (o de forma externa) y probar detenidamente el correcto funcionamiento de nuestras estructuras de datos. Además, Android posee sus propias extensiones de tal librería, con lo que las pruebas unitarias que dependan del ciclo de vida del sistema o del SDK también son realizables.

Incluso para comprobar el rendimiento de las interfaces visuales, existe una extensión realizada en Python que mediante una línea de comandos envía pulsaciones y eventos aleatorios a la interfaz para observar si su comportamiento es correcto.

Tras realizar estas pruebas unitarias, o al menos las más estándares y relacionadas con las estructuras de datos y su permanencia, el sistema parece estable. La información se guarda y se recupera de forma correcta y la interfaz no tiene errores críticos, más allá de los típicos errores de estilo o apariencia.

Sin embargo, en la aplicación que nos ocupa son necesarias otro tipo de pruebas, que evalúen el verdadero potencial de la aplicación y comprueben si funciona correctamente.

Entre los objetivos se especificó claramente que se desea ahorrar batería sin comprometer la conectividad. Esta métrica variable de conexión no es fácil de medir, ni es una unidad muy objetiva y fija, sin embargo, el gasto de batería es una métrica muy importante que no debemos perder de vista.

Si intentando ahorrar energía hubiéramos gastado más aún de lo normal, la aplicación carecería de utilidad, y los usuarios renegarían de ella. Por otra parte, incluso aunque ahorrase batería, si sacrificase mucho tiempo de conexión, se trataría del mismo caso pero en el sentido opuesto.

Por ello se han realizado diversas pruebas con la aplicación de forma interna usando el subsistema de datos, así como mediante el uso de aplicaciones externas. Todos estos resultados serán detallados en la sección de conclusiones, donde se especificarán qué aplicaciones externas de ayuda han sido usadas así como el ahorro conseguido de batería.

9.3– Pruebas de consumo

Siguiendo con las pautas indicadas en la sección de objetivos, el proyecto debe poseer unos costes de ejecución mínimos, y sus consecuencias sobre las redes inalámbricas y su gasto debe cumplir ciertos requisitos.

Como se ha mencionado anteriormente, debido al carácter ahorrativo de la aplicación, si el uso de la aplicación perjudicase al conjunto general de consumo del usuario sin aportarle ventajas, esta perdería su utilidad.

El objetivo de esta aplicación es entonces reducir el gasto producido por las redes inalámbricas o sensores en general, que en ocasiones al dejarlos encendidos sin necesidad real de usarlos, producen un consumo innecesario de batería. Por ello, se quieren dar herramientas al usuario para controlar en todo momento qué realiza o no su terminal.

Para probar si se cumplen los objetivos, se han usado principalmente dos recursos. El primero es fundamentalmente el propio subsistema de datos del proyecto. Con él, seremos capaces de mostrar en pantalla un resumen del gasto producido con el paso del tiempo en forma de gráfica, e incluso también contrastarlo con otros valores como los estados de conexión o de batería, para relacionar ese gasto con sus fuentes.

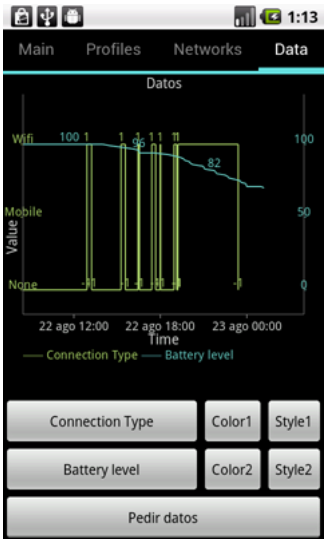

Figura 9.1: Gráficas del sistema.

Por ejemplo, en la imagen 9.1 podemos observar cómo se muestran algunos tipos de datos. En esta captura, se muestra el nivel de energía,

así como el estado de las conexiones del dispositivo. Se han asignado ciertos valores numéricos (elegidos por el SDK de Android, en realidad) para mostrarlos, y gracias a *AChartEngine* somos capaces de asignarles cadenas de texto en la gráfica.

De esta forma somos capaces de observar en qué momento se agrava el gasto y a causa de qué red se produce. Aunque pudieran estar activas dos fuentes de datos como el Wi-Fi y la red de datos móviles, esta gráfica se basará en los datos de Android, que nos indican cuál es la red preferida por la que se enviará la información, y por tanto, la que realiza el consumo de energía.

La segunda herramienta a utilizar para controlar el gasto, será una fuente externa, que pondrá el punto de vista objetivo para juzgar el gasto y la corrección de los datos del sistema. La aplicación se denomina *BatteryDrain* [16], y se trata de un sistema sencillo para almacenar y consultar en qué momento se pierden puntos porcentuales de energía.

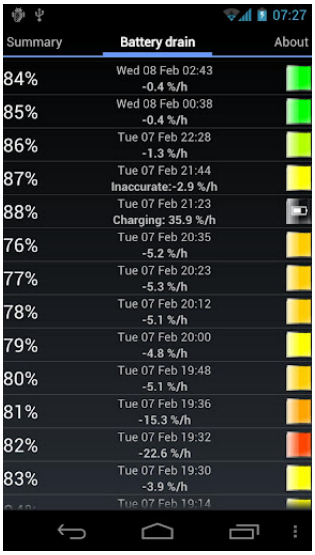

Figura 9.2: Interfaz de aplicación BatteryDrain

Su interfaz se puede observar en la figura 9.2, y su sencillez es su punto fuerte. Nos muestra en forma de lista cada uno de los momentos de gasto porcentual acompañados de una estimación del gasto de esa franja de tiempo al comparar cada punto con el anterior, asignándole también un color.

Una de sus limitaciones es que no nos menciona qué tipo de redes o

sensores están produciendo ese gasto, con lo que habrá que complementar el uso de esta aplicación con el subsistema de datos de *Commandroid* y comparar los resultados.

Con estos dos sistemas se prueban durante esta fase, y por cada una de las mejoras introducidas el gasto, y se comprueban varias cosas.

Primero, que el gasto producido por el modo especial de activación para móviles con ahorro de batería gasta algo más de lo normal, ya que tiene que forzar ciertos procesos de Android para conseguirlo. Esto no supone un problema, ya que partíamos de un terminal al que no se le permite al usuario mantener activa su conexión Wi-Fi, con lo que no existe comparación, y hemos conseguido proporcionar al usuario una nueva opción.

Segundo, que el modo de conexión cíclico añadido cumple con los requisitos del gasto. Mediante el apagado y encendido de las redes durante periodos concretos de cinco minutos y lo elegido por el usuario respectivamente, se consigue cierto ahorro en comparación a un encendido completo durante todo ese proceso.

Y tercero, que los modos proporcionados para desactivar cualquier fuente de gasto durante zonas horarias, como por ejemplo por la noche, brindan al usuario otra opción para ahorrar energía cuando no esté usando el terminal pero lo necesite encendido ante posibles llamadas. Con esto se le proporciona más control, cumpliendo con parte de los objetivos iniciales.

Todos estos datos y resultados se comprobarán y comentarán en la sección de conclusiones, haciendo especial incapié en el cumplimiento de las metas especificadas al inicio del proyecto.

CAPÍTULO 10

Comparación con otras alternativas

Gracias a las facilidades que Google proporciona a los desarrolladores, el número de aplicaciones disponibles para Android es enorme. Antes de desarrollar el proyecto, realicé una búsqueda generalizada para comprobar si existía alguna aplicación que arreglara el problema de las desconexiones Wi-Fi para aquellos terminales que las suspendieran por políticas ocultas de ahorro.

En mi caso, encontré pocas que trataran este problema, y muchas de ellas parecían ejecutar ciertos planes de consumo de batería que les parecía adecuado sin prácticamente informar al usuario. Por ello, me dispuse a crear este proyecto y conseguir evadir la presencia del mencionado bug.

Durante el desarrollo, sin embargo, mi rango de búsqueda y capacidad de información sobre tales problemas y sobre las gestiones de redes inalámbricas aumentó en gran medida. Es entonces cuando comienzo a encontrar un buen número de aplicaciones que si bien no solucionan todos los problemas descritos en este documento, por solitario sí eran capaces de modificar ciertos comportamientos. Otras, en cambio, son muy útiles para mostrar información de batería. Y entre todas ellas, algunas se hacen extremadamente difíciles o molestas de usar, y otras son brillantes en su ejecución.

En esta sección se van a describir todas aquellas aplicaciones que para bien o para mal han sido puntos de referencia, de competencia, y aquellas que se han publicado durante la creación de éste proyecto e incluso han

sido creadas con unas metas muy similares, haciendo evidente que estas necesidades están más extendidas de lo que inicialmente pensaba.

La primera de todas ellas ha sido al mismo tiempo una de las más útiles y de las más fáciles de utilizar. Se llama *BatteryDrain* o *My Battery Drain Analizer* [16]. La descubrí aproximadamente tras completar el subsistema de datos, pero a falta de una interfaz que pudiera mostrarlos gráficamente, y diversos problemas con una librería previa de gráficos, no podía consultar mis propias variables de consumo de forma cómoda.

Esta aplicación me ha permitido desde entonces consultar cuánto gasto de batería realiza mi móvil durante un uso normal, y compararlo con los diversos modos de conexión de mi proyecto. De hecho, en la próxima sección de *Conclusiones* será utilizada como evaluador externo, independiente y objetivo para comprobar la efectividad del desarrollo realizado.

La segunda llegó con cierto don de la oportunidad justo tras el desarrollo del subsistema de perfiles. Se trata de GO Power Master [17], publicada a mitad de la creación de este proyecto, y con un impresionante parecido. El objetivo de esa aplicación es muy similar, ya que dispone de diversos modos preconfigurados o personalizables que activar a ciertas horas del día para ahorrar batería. Además, también dispone de una sección donde consultar datos relativos a la batería en forma de gráfica. Incluso el parecido gráfico en los iconos con el prototipado inicial de la interfaz de este documento es innegable.

Aún con todos estos problemas de similaridad, la aplicación desarrollada dispone de varias opciones de visualización de otros tipos de datos, modos de conexión específicos para cubrir bugs de Android o que establecen periodos de conexión, así como configuraciones especiales para redes Wi-Fi, con lo que sigue poseyendo factores únicos que la distinguen. En cualquier caso, la aparición de una aplicación tan similar no tiene porqué ser algo negativo. Si el destino de esta aplicación fuera exhaustivamente comercial, sería sin duda un obstáculo, pero incluso en esa situación, los millones de descargas de la aplicación mencionada demuestran claramente que existe una necesidad para un buen número de usuarios que requiere mayor control sobre su gasto de energía. Esto supone un buen apoyo psicológico a la utilidad de ésta aplicación, cuya finalidad es la de aprender y obtener formación sobre Android y sus sistemas de creación de aplicaciones mientras se consigue un producto útil, como mínimo, para el desarrollador y los usuarios encuestados.

Además de éstas dos aplicaciones, que han sido importantes de una forma u otra durante la creación de este proyecto, existe una cantidad

impresionante de aplicaciones de todo tipo para ahorrar batería, ejecutar configuraciones personalizadas según el momento en que nos encontremos, y también para mostrar diversa información. Muchas de ellas han sido encontradas de forma tardía, y por ello no han sido influencias para el desarrollo, sin embargo, son buena fuente de conocimiento para observar hasta donde se puede llegar y cómo.

Una de ellas es *JuiceDefender* [18], casi un referente obligado sobre estas cuestiones. La aplicación, como GO Power Master posee diversos perfiles predefinidos sobre gasto de energía, con algunos más agresivos y otros más permisivos, pero dando también la posibilidad de crear uno personalizado. Además, tiene opciones de control para vigilar y controlar el gasto que pudieran estar realizando otras aplicaciones externas, y un modo similar al de este proyecto que permite conexiones cada cierto tiempo.

Otra muy conocida es *BatteryDoctor* [19], que permite encender o apagar los dispositivos inalámbricos según nos interese, aunque su interfaz y opciones son más limitadas que con respecto a las aplicaciones anteriormente descritas.

Y por último terminamos con *GreenPower* [20], que nos muestra una interfaz más colorida y similar a la interfaz Metro que usa Microsoft en sus nuevos dispositivos, con muchos tipos de opciones ordenadas en forma de rectángulos y cuadrados por doquier. Sus funciones son ya conocidas por el lector, y han sido mencionadas en aplicaciones anteriores. Posee modos de configuración diurnos y nocturnos, activación o no de las redes inalámbricas, e información del estado de la batería, todo accesible de forma directa.

Con todas estas aplicaciones podemos observar de forma general el estado de Google Play sobre estas cuestiones de ahorro de energía. Desde hace cierto tiempo la variedad de aplicaciones para controlar estos factores es extremadamente amplia, tanto que se hacen difíciles de buscar, no ya por su cantidad, sino por descubrir qué nombre y palabras clave poseen aquellas que busca y necesita el usuario.

No queda sino basarse en las ideas vertidas por la comunidad de desarrolladores para ofrecer un mejor producto y seguir mejorando este proyecto en la medida de lo posible. Aunque la aplicación muestre un estado de completitud correcto, el camino a recorrer es largo hasta depurarla al máximo, prestando siempre atención a los usuarios y sus necesidades en cuanto a interfaz, utilidades y consumo de energía.

CAPÍTULO 11

Conclusiones

11.1– Cumplimiento de objetivos

Para comprobar el cumplimiento de las metas estipuladas al inicio del documento, conviene recordar cuales eran. Durante la mayoría de los capítulos anteriores se les ha hecho referencia según ha sido necesario, pero es ahora cuando deben ser comprobados con claridad.

Los objetivos principales de la aplicación, son tres:

- Minimizar el gasto en batería del terminal.
- Maximizar la conectividad del dispositivo.
- Garantizar control e información al usuario sobre el gasto de batería y sus conexiones.

Todos están intrínsecamente relacionados y una inclinación extrema por uno de ellos podría poner en riesgo los otros dos. Por ejemplo, una aplicación que se encargase de desconectar sensores y redes inalámbricas con un criterio estricto podría perjudicar la conectividad y cumplir el segundo punto, ahorrar mucha batería cumpliendo el primero, y ser o más configurable dejando el tercero en un terreno intermedio.

En nuestro caso, la intención durante el desarrollo ha sido principalmente proporcionarle al usuario el control absoluto, reforzando nuestros

intereses en el tercer punto. Al crear el proyecto con esta meta en mente, eliminamos la posibilidad de imponer nuestro criterio para los dos primeros puntos, y le dejamos al usuario una variedad de herramientas para que en algunos momentos refuerce la conexión y en otros el ahorro, según le sea de interés.

Con las pruebas realizadas en la etapa correspondiente, se ha comprobado gracias al propio subsistema de datos y a la aplicación externa *BatteryDrain* que el consumo cumple con lo necesario.

La aplicación brinda al usuario la posibilidad de usar, aproximadamente tres criterios para las conexiones inalámbricas, que según la versión actual sólo se aplican a las redes Wi-Fi, por los problemas con el SDK descritos con anterioridad. Estas tres opciones son:

- Activar la red correspondiente. Se ha hecho incapié en garantizarle al usuario una conexión fiable y contínua durante todo el tiempo que necesite y haya estipulado en los perfiles de configuración. Incluso si el terminal que utiliza tiene perfiles de ahorro extremo de batería que desactive las redes en su ausencia (es decir, cuando se produce el bloqueo de pantalla), se han tomado las medidas necesarias para que en los modos denominados *Keep on* esto no ocurra.
- Mantener pequeños periodos de conexión alternados con mayores lapsos de desconexión. Mientras no se reciba otra acción que modifique este modo, se ejecutarán sucesivamente con el tiempo de desconexión definido por el usuario. Si el periodo de conexión se desea modificar, en próximas iteraciones se implementará con un menú de configuración, pero actualmente se muestran como una opción óptima para actualizar los servicios sin comprometer la batería. Con este método, el usuario podrá mantener un flujo de actualizaciones escalonado que, aunque no mantenga los datos actualizados en el más estricto tiempo real, sí proporciona un término medio en gasto y conexión.
- Desconexión controlada. Si el usuario ha terminado de usar las redes inalámbricas puede desactivarlas manualmente, pero si necesita algún tipo de automatismo lo encontrará en ésta aplicación. Podrá establecer horas a las que no se deben tener encendidos estos dispositivos y así ahorrar batería e incluso evitar interrupciones durante el sueño.

Con estas tres aproximaciones se espera que el usuario tenga las suficientes opciones como para distribuir la energía de su terminal como vea

necesario.

Además, los datos obtenidos por *BatteryDrain* y el subsistema de datos de este proyecto *Commandroid*, refuerzan la utilidad de la aplicación y su cumplimiento de estos objetivos. En la siguiente tabla 11.1 se pueden consultar un resumen de las mediciones de consumo realizadas en los distintos modos.

Cuadro 11.1: Resultados de consumo de energía según el modo de uso.

	Gasto medio por hora
Usuario activo. Uso intensivo.	60 % / h
Usuario activo. Uso normal.	40 % / h
Modo asegurar conexión	17,6 % / h
Modo cíclico: 5 min	10,5 % / h
Modo cíclico: 30 min	2,325 % / h
Modo cíclico: 60 min	1,2 % / h
Desconexión	0,3 %/h

Los dos primeros son estados normales del terminal, en los que el usuario está utilizando activamente las redes inalámbricas. En el caso de una red lejana o de poca calidad, la energía necesaria para el envío de los datos aumenta, y el móvil puede verse afectado. También se pueden alcanzar estas cotas de gasto si el uso de CPU es alto o si muchos procesos están enviando y recibiendo información a la vez.

El segundo es un caso más típico. En los terminales *ZTE* medidos el gasto suele ser similar y ronda el cuarenta por ciento por hora. Dichas mediciones se han tomado cuando el usuario estaba usando la red Wi-Fi para consultar alguna red social mediante su correspondiente aplicación, algún navegador instalado y en segundo plano algunos clientes de mensajería instantánea.

A continuación se encuentran los modos que se pueden activar mediante este proyecto. Todos ellos cuentan con la ausencia del usuario, y su permiso en forma de perfil o plan de acción para efectuar los cambios. Es aquí donde vemos la mayor variedad de consumo. Si el usuario no vuelve para consultar alguna notificación (ya que alteraría los datos obteniendo de nuevo picos de gasto sobre el cuarenta por ciento) el gasto va desde el 17 porcentual de una conexión permanente al casi nulo 1,2 porcentual por hora.

En las siguientes imágenes 11.1 y 11.2 se pueden observar un ejemplo reciente de obtención de estos datos, visto desde ambos puntos de vista según ambas aplicaciones.

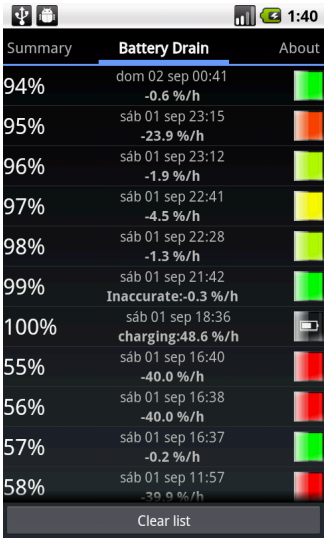

Figura 11.1: Ejemplo de obtención de los datos de consumo

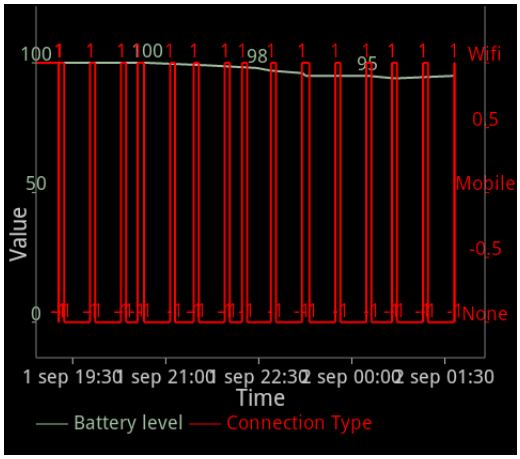

Figura 11.2: Ejemplo de obtención de los datos de consumo

En la primera imagen podemos observar los periodos amarillos o verdes que reflejan un gasto aproximadamente bajo. Los tiempos de esta muestra fueron cinco minutos de conexión separados entre sí por 25 minutos, es decir, conexiones cada media hora. Durante la muestra, el usuario accedió al terminal en dos ocasiones.

Estos accesos se reflejan en ambas imágenes. En la figura 11.1 el acceso del usuario ocasiona un gasto mayor de lo normal, que se refleja con el característico color rojo, y un gasto medio del veinte por cierto, haciendo media entre el gasto típico de 40 por ciento y el gasto casi nulo de los periodos de media hora.

En la segunda figura se aprecia de un modo más sutil, tanto que quizás el lector prefiera acceder a los archivos originales de las imágenes, que por economía de espacio han sido reducidas de tamaño. Cada vez que el usuario ha accedido a la aplicación, ha sido a causa de una notificación realizada durante un periodo de conexión. Por ello, en cuanto dejaba el móvil en estado de suspensión, el plan se reactivaba con una nueva conexión de cinco minutos. Si el usuario no vuelve bien porque no oiga nuevas notificaciones o porque estas no se produzcan, el plan continúa alternando desconexiones.

Entonces, las señales cuadradas de la imagen, en color rojo, aumentan de anchura cuando el usuario utiliza el terminal para consultar el aviso. Por ello, algunas son mayores que la media, y la mayoría tienen el mismo tamaño de cinco minutos. La imagen también nos proporciona una estimación del gasto, que apenas baja un cinco puntos porcentuales en todo el proceso, del máximo al 95 por ciento.

Gracias a estos datos, y a los anteriores, podemos decir que los objetivos se cumplen. Se ha proporcionado al usuario de herramientas para mejorar su control sobre estos factores, y en su aplicación no se ha apreciado errores de bulto en cuanto a consumo.

Aún así, existe un margen amplio de maniobra para optimizar los métodos de colisión de perfiles así como de modos de conexión, que será ampliado en la próxima sección, poniendo fin a los capítulos formales del proyecto junto con el manual de usuario.

11.2– Mejoras futuras

Llegados a este punto conviene recordar todos aquellos aspectos que, por falta de tiempo, problemas con las herramientas, o simplemente por

la llegada de nuevas sugerencias en secciones finales del desarrollo, no han sido implementados.

- Conexiones o desconexiones de la red de datos móviles: A pesar de investigar un buen tiempo sobre este problema, no se han encontrado soluciones óptimas y aceptadas por el SDK público de Android para controlar tales redes inalámbricas. Es necesario redoblar el esfuerzo de investigación y encontrar alguna solución que no afecte a la calidad o independencia de la aplicación, ya que según la experiencia del mercado de aplicaciones *Google Play*, esta acción puede llegar a realizarse.

- Realizar un Widget que active o desactive algún perfil en concreto de forma rápida para el usuario. Puede hacerse incómodo entrar a la aplicación para simplemente desactivar su funcionamiento completo o parcial.

- Crear un icono de notificación que se muestre en la barra de tareas si existe algún perfil a activar cuando el usuario se desconecte. De este modo, si el usuario detecta que se han producido cambios y no recuerda los permisos ofrecidos a la aplicación, poseerá arriba un indicador para recordar que estos perfiles están ejecutándose en su ausencia.

- Aumentar la cantidad de datos ofrecidos al usuario, tanto en la pestaña general como en la sección de datos. La estructura y el diseño de la aplicación permiten la inclusión futura de nuevas opciones con relativa sencillez, excepto por la creación de nuevas clases receptoras para nuevos datos.

- Entre estos nuevos datos podrían situarse los sensores del terminal, que por su amplitud y variedad finalmente no han entrado en el desarrollo. Con ellos podrían crearse nuevas causas que amplifiquen el control del usuario, como por ejemplo, detectando la cantidad de luz, movimiento o inclinación para ejecutar ciertas acciones cuando el terminal parezca estar en un bolsillo, o viajando de un lugar a otro.

- Añadir causas o activadores relacionados con la posición del usuario. Aunque con el sensor *GPS* y los sensores *Bluetooth* existen los mismos problemas de activación y desactivación no permitida por el SDK de Android, sí podríamos recibir sus datos y modificar los perfiles según la localización del usuario, por ejemplo.

Todas estas mejoras, junto a otras de usabilidad y estabilidad de la aplicación serán aplicadas en posteriores versiones e iteraciones del desarrollo, que no tienen ya lugar en esta documentación. Se convierte pues en obligación del desarrollador optimizar la aplicación para poder subirla a la red de aplicaciones de Google y escuchar las sugerencias de los usuarios para mejorarla lo máximo posible.

Como conclusión final, cabe dejar un poco de lado la aplicación y centrarse en la formación recibida, que es en definitiva el mejor producto conseguido durante este desarrollo. Aunque los inicios del proyecto han sido lentos y ampliamente supeditados a los procesos de investigación, se ha percibido una gran mejoría en los procesos de diseño e implementación, produciéndose cada vez más rápido.

Esto ha sido especialmente patente en el desarrollo del segundo proyecto, que se ha nutrido de muchos de los conocimientos adquiridos en el primero, así como de algunas ideas, interfaces y código que si bien no ha sido muy reutilizable, en esta nueva versión lo es. De hecho, durante el segundo proyecto hay secciones y subsistemas que han ido retroalimentándose entre sí, dejando patente cierta calidad de diseño que ha permitido adaptar con comodidad aspectos comunes.

Llegados ya a este punto final del proyecto, es buen momento para volver a agradecer a todos los desarrolladores y aplicaciones externas mencionados durante la documentación. Además de las aplicaciones utilizadas para la evaluación, como *BatteryDrain* y de aquellas competitivas hasta el extremo como *GO Power Manager* que han servido como referencia final, existen algunas que han sido esenciales desde el primer momento para poder crear este proyecto.

La primera de ellas está presente en todo el código LaTeX, y es la clase PClass [21] para la creación de documentación de proyectos de Fin de Carrera para el departamento MA1, que simplifica enormemente la redacción y organización de los mismos.

También han sido esenciales las librerías gráficas utilizadas para representar los datos en la cuarta pestaña. Sin ellas, gran parte del desarrollo se hubiera gastado en *reinventar la rueda*, creando desde cero unas estructuras de código que han sido ya implementadas multitud de veces en multitud de sistemas. Finalmente la escogida fue AChartEngine [14], cuyas aptitudes y libertad en su licencia nos permiten mejores condiciones de implementación.

Y finalmente toca reconocer al asequible y accesible sistema de Android para desarrollar aplicaciones y subirlas al mercado oficial de Google, así como su extensa documentación, e inestimable comunidad de desarrolladores [22] [23].

CAPÍTULO 12

Glosario

En esta sección se describirán todos aquellos términos técnicos mencionados durante el documento que necesiten una explicación detallada de su significado. Si algun elemento mencionado anteriormente no aparece en este capítulo, se recomienda al lector una búsqueda por la bibliografía del proyecto, situada en las últimas posiciones de la documentación.

- Activity: Es la fachada de toda aplicación, y el vínculo entre la misma y el usuario. Al igual que el elemento *Service*, están sujetos a los ciclos de vida que Android les imponga, debiendo cerrarse o *descansar* cuando se les exija.
- Service: Es el código de la aplicación que necesite ejecutarse en segundo plano. Tiene un ciclo de vida más extenso que Activity, ya que actúa sin mediación directa del usuario, pero siguen sujetos a un ciclo de vida controlado por el sistema. Tanto Activity como Service se comunican entre sí por Intents o datos compartidos (base de datos, etc).
- *Content Providers*: Sistema proporcionado por Android para compartir información entre distintas aplicaciones, como por ejemplo, para disponer de un sistema de *portapapeles*, para copiar y *pegar* información entre ellas.
- Ciclos de vida: Las aplicaciones y sus servicios en segundo plano tienen un tiempo de vida limitado. El sistema operativo puede cerrarlas o pausarlas cuando necesite más memoria disponible o si han

terminado de ejecutarse, y para ello, debemos prepararlas convenientemente.

- Intents: Son los mensajes que discurren por el sistema operativo continuamente. Algunos se pueden consultar y recibir mediante Receivers, pero otros se pueden crear de forma personalizada y exclusiva para nuestra aplicación. Transmiten tipos básicos como números y booleanos, pero también objetos completos si los preparamos para ello.

- Bundle: Es un tipo de objeto que empaqueta otros elementos dentro de un Intent. Se pueden utilizar para estructurarlos o para recibir datos ya empaquetados de algún Intent específico de Android.

- Parcelable: Se trata de la interfaz que un objeto ha de implementar si quiere poder ser enviado dentro de un Intent. Por ejemplo, si un Service envía un objeto de tipo *Tarea* a una Activity, la segunda debe tener un Receiver, y la clase *Tarea* debe tener métodos de tipo Parcelable que la preparen para su envío.

- Serializable: Es un sistema de permanencia de datos de Java que permite guardar objetos en un archivo en disco. Si un objeto ha de ser *serializado*, debe implementar ciertos métodos para su creación y reconversión. Parcelable es el sistema de serialización para Android, más rápido y optimizado para la plataforma, mientras que Serializable se consideraría su antecesor y solución general para el lenguaje Java.

- Vistas o *View*: Son los elementos gráficos de una aplicación. Los hay de todo tipo: botones, menús desplegables, selectores de opciones, etc. Para añadirles funcionalidades cuando se pulsen o modifiquen su estado, se usan las interfaces de tipo *Listener*

- Interfaces de tipo *Listener*: Cada clase que las implemente, tendrá un método que será llamado ante un cambio en una vista o View, como por ejemplo, pulsar un botón o escribir texto. En esa función se aplicarán los cambios necesarios causados por tal evento.

- *Widget*: El término se usa indistintamente para los elementos View que son interactuables con el usuario dentro de un Activity, y también para pequeños menús gráficos que se pueden colocar en el escritorio principal del sistema para acceder a su información de forma rápida y directa.

- Layout: Son los patrones que definen la posición de los elementos gráficos en una aplicación. Por ejemplo, *LinearLayout* nos permite colocar una lista vertical u horizontal de objetos en pantalla,

mientras que *RelativeLayout* nos da recursos para colocarlos donde queramos, definiendo incluso relaciones de posición entre ellos.

- ViewGroups: Se tratan de estructuras que agrupan varias vistas o Layouts añadiéndoles cierto comportamiento especial. Por ejemplo, *ScrollView* permite bajar o subir una lista de elementos mediante gestos táctiles, y *TabHost* y *ViewFlipper* permiten alternar entre varias pestañas de contenido.

- *ArrayAdapter*: Se trata de una clase que nos permite enlazar un elemento de una lista de objetos (como un *Array*), con su correspondiente apariencia gráfica (un *ListView* o *GridView*) y comportamiento en cuanto a pulsaciones (añadiéndoles algún *Listener*). A menos que necesitemos una personalización extrema, se trata de un sistema cómodo y útil que nos evita crear el aspecto de cada elemento manualmente. Además, dispone también de una vista automática para listas vacías.

- Permisos: Para cuidar la seguridad, los permisos de acceso de una aplicación están convenientemente categorizados y notificados, tanto al crearla como al instalarla. Así, el usuario sabe en todo momento qué secciones de su información se están usando o se van a usar. Se especifican en un archivo denominado *Manifest*.

- Manifest: Supone el *contrato* entre el desarrollador y el sistema operativo. En él se estipulan los permisos que la aplicación necesita para ejecutarse, cada Service o Activity que la componen, y las clases que reciban datos del sistema. Utiliza una estructura XML de árbol, que será interpretada por el sistema.

- XML: Lenguaje de marcas con estructura de árbol que permite estructurar propiedades de algún objeto de forma clara y legible para el desarrollador sin que suponga dificultad de interpretación para un programa. Se usa con bastante frecuencia en Android, tanto para delimitar las interfaces, como para crear archivos de estilo, de idiomas e incluso gráficos.

- Receivers: Las clases que heredan de BroadcastReceiver pueden recibir datos del sistema operativo periódicamente. Para que Android las reconozca y les envíe dicha información, deben figurar en el Manifest y tener los permisos adecuados.

- AlarmManager: Es una clase de Android que se encarga de gestionar el sistema de alarmas. Si nuestra aplicación necesita realizar una actividad a una hora concreta, antes que utilizar algún sistema clásico de reloj de Java, podemos usar éste sistema para que envíe un

Intent que podamos tratar como nos convenga, escapando también del problema del ciclo de vida limitado.

- Notificaciones: Android posee un sistema en la barra superior que nos va notificando de los eventos que ocurren en nuestro terminal, ya sean de batería, wifi, llamadas entrantes o mensajes entrantes. Como desarrollador, se pueden crear notificaciones al usuario cuando ocurra algún evento en especial, siempre que usemos la clase *NotificationManager* que las gestiona.

- *Log* y *Logcat*: Son los sistemas de impresión de texto para detección de errores. Nos permite enviar texto a un sistema de monitorización y consultar en todo momento el estado de nuestra aplicación y de los errores que pudieran encontrarse.

- Diagrama *Gantt*: Se trata de un tipo de diagrama que nos permite observar la separación en diversas tareas de un proyecto de forma gráfica. Cada una de ellas ocupa un rectángulo mayor o menor según su duración, y forman una cadena de eventos según las relaciones temporales que posean entre sí.

- Método *Pert*: Es una estructura que nos permite ver la subdivisión de tareas de un proyecto de forma gráfica. Está formado por círculos que representan estados, y por líneas de flujo que los conectan, que representan acciones. No pueden existir ciclos, y cualquier camino nos lleva al estado final, de una forma u otra. El camino cuyas actividades no puedan retrasarse porque retrasarían el proyecto entero se denomina *camino crítico* y debe ser prioritario en el desarrollo.

- Método *Roy*: Es un sistema similar al método PERT, pero esta vez las actividades son los nodos, y las aristas son sus relaciones. Cada actividad se representa con un cuadrado, y se colocan sus tiempos de inicio y finales, y también su holgura, que es el tiempo que podría retrasarse sin implicar a la finalización del proyecto. Por lógica y por definición, una actividad del camino crítico no tiene holguras.

- Modelo Cocomo: Sistema para calcular y estimar la dificultad de un proyecto software mediante datos como la experiencia del equipo desarrollador, la complejidad del lenguaje a utilizar, el número de trabajadores o las miles de líneas de código necesarias.

- *Eclipse*: Entorno de desarrollo usado durante el proyecto. Tiene multitud de versiones y de complementos, y aunque su uso más extendido es el desarrollo Java, existen variaciones de todo tipo, como *Aptana* para desarrollo web.

- Interfaz de usuario: Se trata del conjunto de elementos gráficos que permiten al usuario realizar tareas, y consulter y modificar información con el sistema. Debe ser lo más cómoda y fácil de usar posible para que no se convierta en una barrera insalvable para el usuario, y sea capaz de controlar sin apenas formación todas las herramientas ofrecidas.

- *Smartphone*: Son la nueva hornada de teléfonos móviles que no sólo incorporan la obligatoria función de realizar llamadas, sino también multitud de sensores y capacidades multimedia y de conexión inalámbricas, pareciéndose cada vez más a un ordenador que a un clásico teléfono fijo de escritorio.

- Android Market o *Google Play*: Mercado oficial de aplicaciones de Google para su sistema Android. Cualquier persona puede adquirir una licencia de desarrollador por un módico precio, y apenas existen trabas para colgar las aplicaciones creadas.

- *Froyo*: Android en su versión 2.2.

- *Gingerbread*: Android en su versión 2.3.

- *Ice Cream Sandwich*: Android en su versión 4.0.

- *3g* o datos móviles: Sistema inalámbrico que permite la conexión a internet en cualquier lugar, ya sea mediante antenas de telefonía y o por satélite.

- SDK: Conjunto de herramientas de desarrollo que proporciona el creador de un sistema para que desarrolladores de todo el mundo creen software para el mismo. Sus modificaciones deben ser tratadas con sumo cuidado para no perjudicar la retrocompatibilidad del código creado.

- API: Recursos que proporciona de forma pública un compontente software para que pueda ser usado por desarrolladores externos. Sus modificaciones deben ser tratadas con sumo cuidado para no perjudicar la retrocompatibilidad del código creado.

- BSSID y SSID: Identificadores para las redes inalámbricas Wi-Fi. SSID se trata simplemente del nombre que recibe la red, y se puede compartir entre varios enrutadores para simular una red mayor. BSSID sin embargo es un direccionamiento que nos permite identificar al dispositivo que emite la información. Un SSID entonces puede identificar a múltiples BSSID, pero cada BSSID es único.

CAPÍTULO 13

Anexo: Manual de usuario

Una vez completado el desarrollo, es necesario crear un correcto manual para explicar a los nuevos usuarios en qué consiste la aplicación y cómo funciona.

Tal y como se definió en los objetivos, y en prácticamente todo el documento, la aplicación trata de darle al usuario poder para controlar todo lo que ocurre en su teléfono cuando no esté, con la principal intención de ahorrar batería sin perder conectividad.

Para ello, la aplicación se divide en cuatro pestañas principales:

- Sección *Main* o principal.
- Sección *Profiles* o perfiles.
- Sección *Networks* o redes.
- Sección *Data* o datos.

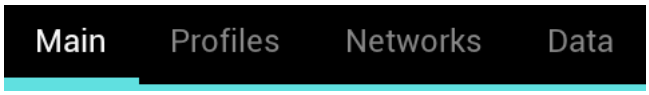

Figura 13.1: Manual de usuario: Selección de menú

Todas se acceden mediante la barra superior, que delimitan cuatro áreas de acción de la aplicación. Si el usuario está en alguna de ellas, y pulsa el botón BACK, la aplicación se minimizará, y volverá a otra que el usuario estuviera usando antes. Si pulsa el botón MENU en cualquier sección de la aplicación, no ocurrirá nada, puesto que se ha optado por mostrar todos los menús de forma gráfica. En próximas versiones, sin embargo, es posible que aparezcan nuevos submenús para configuraciones varias, e incluso podrían modificarse algunas secciones de estas interfaces para añadir nuevas opciones o simplificar la apariencia.

Ahora, pasemos pues a describir cada una de estas subdivisiones.

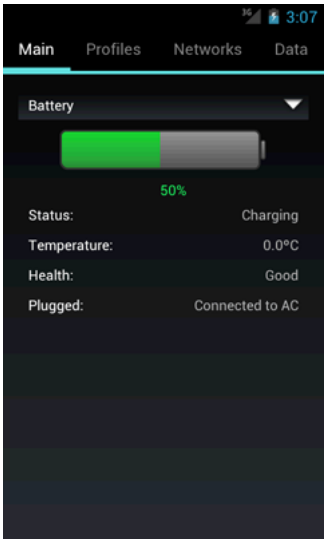

Figura 13.2: Manual de usuario: Menú principal

El cometido de la primera pestaña es mostrar al usuario una selección de los principales datos que debería conocer de su batería o conexiones inalámbricas. En esta primera versión, no se han incluido aún los datos de las conexiones, pero próximamente se situarán justo debajo y se mostrará el estado de otros sistemas. Por ahora, lo esencial es conocer el nivel de la batería y en qué situación está, y más adelante, mediante la inclusión de contadores de tiempo y cálculos de gasto fiables, mostrarle al usuario el tiempo restante estimado.

La segunda sección es la más importante de toda la aplicación, y contiene las herramientas necesarias para controlar el comportamiento del terminal cuando el usuario no esté usándolo. En cuanto el usuario llega a esta pestaña, se le muestra una lista de los perfiles, que pueden estar activados o desactivados, y los controles para añadirlos, editarlos o borrarlos.

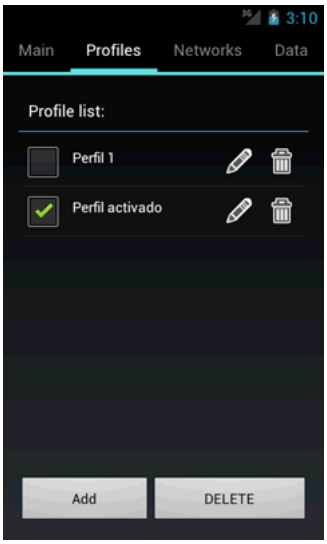

Figura 13.3: Manual de usuario: Menú perfiles

Se mostrará también el nombre de cada perfil para identificarlo.

Antes de entrar en detalle sobre las interfaces de edición de los perfiles, es el momento de describir este sistema.

Cada uno de los perfiles tendrá unas causas de activación, que bien pueden ser eventos horarios, o condiciones específicas como una caída del nivel de batería hasta cierto punto, o que el terminal esté conectado a una fuente de energía. No se trata de algo obligatorio, ya que podría interesarnos tener una configuración estándar para cuando no estemos usando el terminal.

También poseerán unas acciones a realizar cuando el perfil se cumpla, como por ejemplo modificar el volumen de los distintos sonidos (multimedia, alarma y tono de llamada), activar el modo avión, o activar modos especiales para la red Wi-Fi que la active, desactive o alterne periodos de acción e inacción para ahorrar batería.

Todos estos perfiles se activarán según unos criterios. Con el usuario ausente, a ciertas horas definidas por los perfiles, o a cada cierto tiempo si existe nueva información del subsistema de datos, se efectuará un *choque* de perfiles.

Esta *competición* entre estos obtendrá como resultado una lista de acciones a realizar, mezclando las de aquellos perfiles que cumplen sus

condiciones de activación.

La mencionada mezcla se realizará dando prioridad a las acciones de los perfiles activados por una hora concreta. Entre ellos, los de menor rango serán los primeros. Estos depositarán sus acciones en la lista. Luego los de mayor rango, y así hasta que hayamos recorrido todos los perfiles horarios. Si existiera empate, se activarían primero aquellos con mayores causas cumplidas.

Más tarde se recorren los perfiles sin hora, de mayor a menor número de condiciones cumplidas. Todos deben cumplir todos sus requisitos de activación, pero tendrán prioridad aquellos que más requisitos tienen. Uno a uno siguen colocando acciones en el conjunto.

Durante todo el proceso, si queremos insertar alguna acción que ya esté en la lista, se ignorará su modificación, ya que proviene de un perfil de menor categoría y se continuará con los siguientes.

De esta forma, se obtiene una lista de acciones que ejecutar y el sistema se encarga de realizarlas. Por ejemplo, si tuviéramos activos un perfil con horario que cubre el momento actual, sin condiciones extra, y que activase la red Wi-Fi, y otro sin horario que la desactivase sin ninguna condición previa, la red se activaría. Más tarde, si vuelve a producirse un *choque de perfiles* con nuevos datos y el periodo de tiempo del primer perfil ha pasado, las acciones del segundo no competirán ya con las del primero, y en este caso, apagaría finalmente la red.

Cada competicion se produce cada cinco minutos o si se recibe una alerta por hora de un perfil. Pero se deben cumplir dos requisitos. Primero, que el usuario no haya desbloqueado el móvil puesto que estaría usándolo y no debemos interferir en sus quehaceres. Y segundo, que hayamos recibido nuevos datos de conexión o batería que justifiquen una nueva evaluación. También se realizaría un primer *choque* si el usuario acaba de dejar el móvil, para ver qué modificaciones nos corresponde realizar.

Volviendo a la interfaz, le indicaremos al sistema todas estas cuestiones pulsando sobre el botón con forma de lápiz de cada perfil (para editarlo) o en el botón inferior de añadir, para crear uno nuevo.

Para ello, tendremos tres pestañas. En la primera colocaremos el periodo de activación si lo requiere, y un nombre obligatorio. En la segunda, especificaremos causas especiales que deba cumplir el perfil. Y en la tercera, aquellas acciones que queremos realizar.

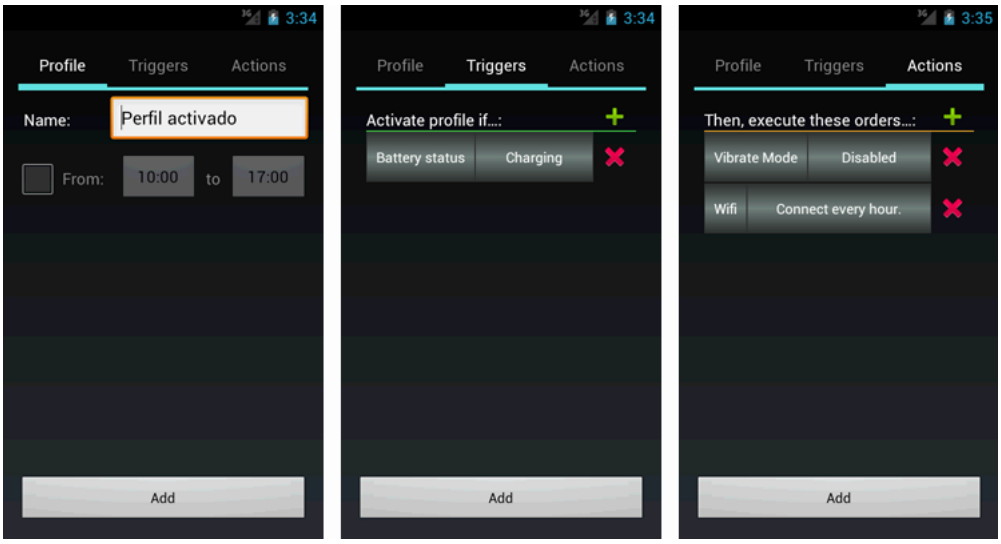

Figura 13.4: Manual de usuario: Creación de un perfil.

En estas dos ultimas pestañas, por cada elemento podemos cambiarlo si pulsamos en su nombre que lo identifica, cambiar el operador si se trata de cálculos numéricos, y finalmente el valor, que bien puede ser un número o una opción textual. Cada uno se añade con la cruz verde y se borra con la cruz roja. En todo momento, podremos añadir el perfil, si tiene nombre, y tiene al menos, una acción. Un perfil sin acción no tendría sentido, y un perfil sin nombre es difícil de identificar.

En la tercera sección se muestra al usuario los controles relativos a las redes inalámbricas. Debido a las limitaciones del SDK de Android, por ahora sólo se muestran las herramientas para las redes Wi-Fi, que de hecho, son las que poseen problemas de conexión que Android arrastra desde sus primeras versiones y queremos evitar.

Para ello mostraremos dos controles, que permitirán a la aplicación operar mientras el usuario está usando el terminal (screen on) y cuando no esté (screen off).

Se mostrará también una lista de configuraciones especiales, para que el usuario indique qué necesita. Cada uno de estos elementos se asociará con un BSSID y/o SSID que son identificadores de redes Wi-Fi. Una vez adscritos a una red, el usuario introducirá la preferencia con la que quiere que se conecte a la misma, y posibles configuraciones especiales como las relativas a las IP estáticas (ip, puerta de enlace, máscara de subred y dns).

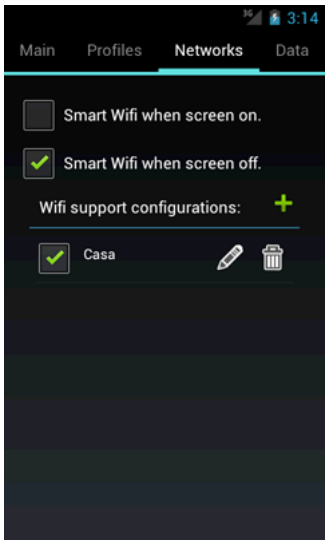

Figura 13.5: Manual de usuario: Menú redes.

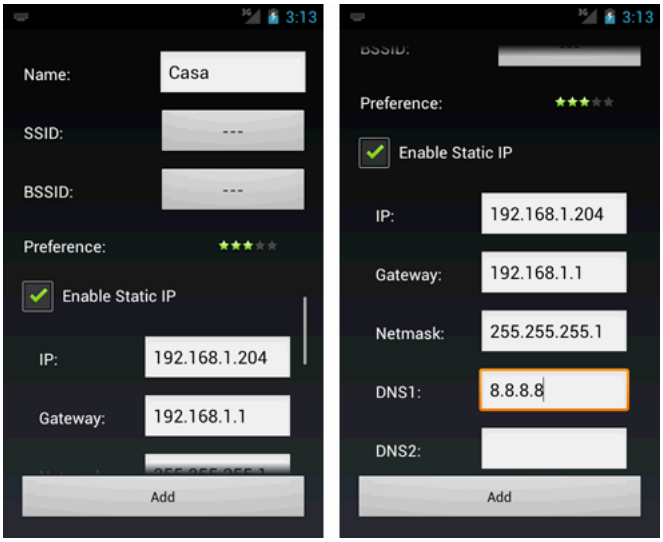

Figura 13.6: Manual de usuario: Creación de configuración especial de red.

Con esta información, si el usuario permite a la aplicación actuar, ya sea en su presencia o en su ausencia, el sistema intentará siempre conectarse a la red preferida en las existentes, y si existiera algún problema en la red, o simplemente el usuario lo requiere, modificará la configuración única de Android sobre IP estáticas para adaptarse a esta nueva conexión.

En todo momento el usuario tendrá abajo de la interfaz un botón para confirmar los cambios, o si pulsa el botón BACK del terminal, descartarlos.

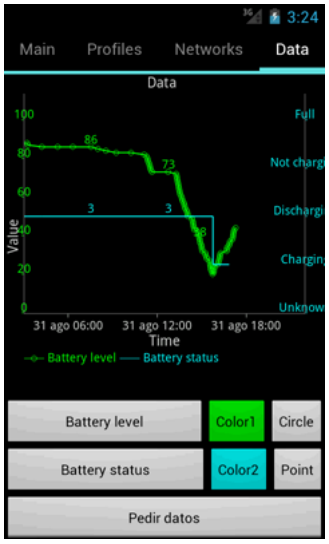

Figura 13.7: Manual de usuario: Menú datos.

Pasamos ahora a la cuarta y última sección, la relativa al subsistema de datos. Durante todo momento posterior a su instalación, la aplicación recopilará información de conexión, batería o sensores para mostrarla al usuario si lo necesita, en esta pestaña.

En esta interfaz, existirán tres botones para controlar cada uno de los dos ejes de la gráfica a producir. El primero seleccionará el tipo de datos, el segundo el color del trazo y el tercero el tipo de puntos. El primer elemento es obligatorio, pero el estilo del trazo si no se introduce nada se realizará con colores aleatorios y estilo de punto normal.

Con estos datos se utilizará un componente software de terceros denominado *AChartEngine* [14] que nos permitirá mostrar los datos al usuario con una gráfica típica y personalizar los valores de los ejes para identificar información con valores textuales, asignándoles un valor numérico.

Con este resumen se dispone ya de una correcta formación para la utilización de la interfaz de usuario y la aplicación. Si el lector posee alguna sugerencia o se encuentra con algún error imprevisto, puede comunicármelo mediante los datos expuestos en ésta documentación, accediendo a mi perfil de desarrollador en la forja RedIris donde se colgará este proyecto de fin de carrera, o en el futuro, o buscando esta aplicación en Google Play o Android Market.

Gracias por su atención y espero que mi aplicación le sea útil.

Bibliografía

[1] Android Developers. Platform Versions. [Consulta: Abril 2012] http://developer.android.com/resources/dashboard/platform-versions.html

[2] Issue 9781 on Android forums. WiFi entering PSP when screen off. [Consulta: Abril 2012] http://code.google.com/p/android/issues/detail?id=9781

[3] Gantt Project. [Consulta: Mayo 2012] http://www.ganttproject.biz/

[4] Método Lean (Startup) [Consulta: Agosto 2012] http://en.wikipedia.org/wiki/Lean_Startup

[5] Eric Ries. The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. [Consulta: Agosto 2012] http://en.wikipedia.org/wiki/Lean_Startup

[6] Android Developers. [Consulta: Abril 2012. http://developer.android.com/index.html

[7] Mark L. Murphy. The Busy Coders Guide to Android. [Consulta: Abril 2012]

[8] Mark L. Murphy. Beginning Android 2. [Consulta: Abril 2012]

[9] Android Developers. Intents. [Consulta: Marzo 2012] http://developer.android.com/reference/android/content/Intent.html

[10] Mark L. Murphy. The Busy Coders Guide to Android. Part IV: Intents. [Consulta: Marzo 2012]

[11] Mark L. Murphy. Beginning Android 2. Chapter 17: Creating Intent Filters. [Consulta: Marzo 2012]

[12] The Blob antipattern. Sourcemaking Antipatterns. http://sourcemaking.com/antipatterns/the-blob [Consulta: Mayo 2012]

[13] Librería GraphView para mostrar gráficas en Android. http://www.jjoe64.com/p/graphview-library.html [Consulta: Abril 2012]

[14] Librería AChartEngine para mostrar gráficas con ejes personalizables. www.achartengine.org [Consulta: Mayo 2012]

[15] Issue 12649 WIFI roaming not working. http://code.google.com/p/android/issues/detail?id=12649 [Consulta: Mayo 2012]

[16] Aplicación BatteryDrain en GooglePlay. [Consulta: Agosto 2012] https://play.google.com/store/apps/details?id=com.WazaBe.android.BatteryDrain

[17] Aplicación GO Power Master en GooglePlay. [Consulta: Agosto 2012] https://play.google.com/store/apps/details?id=com.gau.go.launcherex.gowidget.gopowermaster&hl=es

[18] Aplicación JuiceDefender en GooglePlay. [Consulta: Agosto 2012] https://play.google.com/store/apps/details?id=com.latedroid.juicedefender&hl=es

[19] Aplicación BatteryDoctor en GooglePlay. [Consulta: Agosto 2012] https://play.google.com/store/apps/details?id=net.lepeng.batterydoctor

[20] Aplicación Green Power en GooglePlay. [Consulta: Agosto 2012] https://play.google.com/store/apps/details?id=org.gpo.greenpower

[21] Librería de LaTeX PClass para redactar documentación. https://forja.rediris.es/projects/latexpfc/ [Consulta: Agosto 2012]

[22] Comunidad de desarrolladores XDA Developers. [Consulta: Junio 2012] http://forum.xda-developers.com/forumdisplay.php?f=564

[23] Comunidad de desarrolladores StackOverflow. [Consulta: Junio 2012] http://stackoverflow.com/