ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

CASANDRA

Realizado por
Daniel Segura Ruiz
Tutorizado por
José Antonio Montenegro Montes
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, septiembre de 2016

Fecha defensa:
El Secretario del Tribunal

Resumen: El carácter arbitrario del comienzo, duración y dificultad de las actividades que acompañan al ejercicio discente afecta al rendimiento del alumno al tener que invertir un tiempo en la gestión y planificación de dichas tareas por tener que acomodarlas a su planificación diaria.

Con la intención de facilitar dicha tarea se desarrolló originalmente esta aplicación Android, que permitía a un usuario potencial planificar su tiempo con los siguientes elementos:

- Un horario semanal con las horas de docencia y los intervalos de tiempo disponible para trabajar.
- Una lista de tareas con un carácter periódico o esporádico (es decir con una carga de trabajo expresada en horas totales de dedicación necesaria y con una fecha límite determinada).
- Una agenda a lo largo de la cual desarrollar la planificación, con la fecha de inicio y fin del curnso en que se realizan las tareas.

Con esta información, se elaboraba una gestión del tiempo basada en algoritmos de planificación de procesos como RMS, DMS y EDF, que resultaron adecuados para la elaboración de planificaciones sencillas.

El objetivo de éste trabajo es dotar a la aplicación de unas planificaciones de mayor calidad teniendo en cuenta las preferencias del usuario y desarrollar las posibles mejoras que brindaba usando las preferencias horarias del usuario como heurístico para disponer de un juicio de valor de las planificaciones y así poder ajustar el resultado a sus gustos mediante técnicas de I.A. como CSP (Problemas de satisfacción de restricciones), hacer la interfaz de usuario más cómoda e intuitiva y añadir componente lúdico al seguimiento de las tareas realizadas, premiando el trabajo constante mediante una valoración periódica de la progresión y cumplimiento de las tareas

Palabras claves: Android, Planificación, Tareas

Abstract: The arbitrary nature of onset, duration and difficulty of the activities that learning implies, affects student performance as the student has to invest their time in planning and management of these tasks in order to accommodate them to their daily planning.

This Android application was originally developed with the intention of facilitating this task by allowing a potential user to plan their time with the following elements:

- A weekly schedule with teaching hours and time intervals available for work.
- A list of tasks with a newspaper or sporadic (i.e. with a workload expressed as a number of hours of dedication required and with a certain deadline).
- A calendar along which planning happens, with start and end date those of the course whithin which tasks are performed.

With this information, a time management was made based on process scheduling algorithms such as RMS, DMS and EDF, which were found suitable for making simple schedules.

The aim of this work is to provide higher quality schedules taking user preferences into acount and develop and using them as a heuristic to provide a way to evaluate schedules and adjust the results to their preferences using AI techniques like CSP (Constraint Satisfaction Problems), make the interface more intuitive and user-friendly and add a gaming component to the tracking of completed tasks, rewarding the ongoing work by a periodic assessment of the progress and fulfillment of the tasks.

Keywords: Android , Scheduling, Tasks .

Índice

1. Introducción

1.1. Motivación

En el marco de las enseñanzas regladas, se hace patente la necesidad de una metodología a la hora de afrontar con mesura y buena salud el conjunto de obligaciones que se van a ir sucediendo a lo largo del período de aprendizaje.

Por lo general no es suficiente una simple enumeración de las actividades que se desarrollarán en horarios fijos, sino que es necesaria una planificación del tiempo disponible tras éstas para realizar las tareas y afianzar conocimientos, es decir, para el correcto aprovechamiento de los contenidos.

El carácter heterogéneo del comienzo, duración y dificultad de las actividades que suelen acompañar al ejercicio discente, se puede convertir en un escollo que poco o nada tiene que ver con el objetivo de la enseñanza en sí, pero que repercute en el alumno, al tener que invertir el preciado tiempo disponible, en la gestión y planificación de dichas tareas, al tener necesariamente que acomodarlas a su vida diaria.

Este proyecto surge de la intención de facilitar dicha tarea y ayudar en la medida de lo posible a un uso más eficaz del tiempo del alumno, es decir, pretendemos automatizar el proceso de planificación del trabajo del alumno, que así pierda el menor tiempo posible en la gestión del tiempo, ya que haciendo uso de la planificación resultante de la ejecución del programa (que en esta mejora pretende ser la más cercana a sus preferencias en cuanto a horarios de estudio) resulte un plan satisfactorio para llevar a cabo todas sus tareas a tiempo.

Se ha elegido seguir desarrollando la aplicación que originalmente se elaboró como proyecto final de carrera, ya que brindaba muchas posibilidades interesantes de mejora. La plataforma de desarrollo seguirá siendo *Android*, por resultar adecuado un dispositivo portátil para planificar tus tareas tanto en el centro de estudio como en el hogar, además dicha plataforma cuenta con una gran variedad de herramientas de desarrollo gratuitas y es ampliamente utilizada por la comunidad a la que se dirige la aplicación, lo que nos permite llegar a un gran número de potenciales usuarios.

1.2. Objetivos

La aplicación original permitió que un usuario potencial (un alumno de enseñanza reglada) planificara su tiempo de estudio usando los siguientes elementos

- Un **horario escolar** con las horas de docencia impartidas al alumno y los intervalos

de tiempo disponible para que éste trabaje a lo largo de una semana (cuya mejor planificación posible es nuestro objetivo).

- ◇ Una serie de **tareas** que podían tener un carácter o bien *periódico* (aquella que implique dedicarle un tiempo concertado a una tarea con una cadencia establecida durante todo el período de validez del calendario), o *esporádico* (una carga de trabajo expresada en horas a ser realizado en el tiempo libre expresado en el horario escolar del usuario antes de una determinada fecha).

- ◇ Un **calendario escolar** o **Agenda**, que marcaba los *deadlines* de las *tareas esporádicas*, y que también impone una fecha de inicio y fin del curso, a lo largo de cual desarrollarán las *tareas periódicas*.

Con la información suministrada previamente, se elaboraba una gestión del tiempo basada en algoritmos de planificación de procesos como *RMS, DMS y EDF*, que finalmente resultaron útiles para la elaboración de planificaciones sencillas.

El objetivo principal de éste proyecto es dotar a la aplicación de unas planificaciones de mayor calidad, ya que si bien era posible obtenerlas éstas no tenían en cuenta las preferencias del usuario.

- ◇ Se dota a los intervalos horarios disponibles del usuario de información *cualitativa* y de este modo poder usar las preferencias del usuario en cuanto a la elección de determinados intervalos horarios frente a otros, para poder hacer valoración de las planificaciones, y usar dicha información como *heurístico* para suministrar una planificación resultante ajustada a sus gustos, abordando el problema con técnicas como las empleadas para la resolución de *Problemas de satisfacción de restricciones* (*CSP*). De este modo, ahora cuando se introduzca un intervalo de tiempo disponible para trabajar, también se deberá suministrar información acerca de cuán agradable resulta ese intervalo, generando finalmente (aneja a el horario escolar) una prelación de intervalos a ser utilizados, útil a nuestro enfoque de dar una planificación de calidad.

- ◇ Se mejora la interfaz de usuario, haciéndola más cómoda e intuitiva, buscando reducir el número de pulsaciones y procurando una presentación escueta y precisa de contenidos, para ello se ha incluido un *navigation drawer* (menú contextual que permite navegar cómodamente atajando a las distintas secciones principales de la aplicación). Se ha mejorado el contraste de los gráficos para procurar una lectura más agradable.

◇ Se añade un *componente lúdico*; la aplicación realiza un seguimiento de las tareas tras la planificación inicial de las tareas, así que cada vez que se acceda a la aplicación, ésta evalúa el estado de realización de las mismas y, si es necesario, replanificará o exigirá cambios en caso de que no sea planificable con las condiciones actuales; la parte de ludificación premia el trabajo constante, mediante una valoración periódica de la progresión de las diferentes tareas y el cumplimiento de sus *deadlines* (fechas límite para su realización).

El objetivo es motivar el compromiso mediante la captura del interés del estudiante e incentivar la constancia (haciendo uso del instinto intrínseco del ser humano por el gusto al juego), la ludificación, definida en sentido amplio, es *"el proceso de definir los elementos comprendidos en los juegos que los hace divertidos y motivan a los jugadores a seguir jugando, y usar esos mismos elementos en un contexto ajeno al juego para influenciar el comportamiento"*

◇ Otra mejora es la posibilidad de compartir los **horarios**, la aplicación permite importar y exportar un horario determinado, de modo que se puedan difundir de un modo sencillo un esquema horario que posteriormente los destinatarios pueden adaptar a sus gustos o necesidades. De este modo sería trivial enviar a un conjunto de alumnos los horarios de las asignaturas y que cada uno de los alumnos adaptara dicho horario incorporando únicamente las horas que tienen disponibles a lo largo de la semana que les vinieran bien para trabajar en las materias. Se pueden transmitir mediante cualquier vía, ya sea correo electrónico paso de mensajes con aplicaciones de mensajería, gestores de ficheros de *Android*, descarga desde alguna *url*, etc . . .

◇ En cuanto a los avisos y eventos de las tareas planificadas, cabe señalar que el resultado de las planificaciones la aplicación las integra en la plataforma utilizando la *API* de *Android Calendar Provider*, generando eventos de calendario para los trabajos a realizar, de éste modo toda la funcionalidad cualquier aplicación que trabaje contra el calendario de la plataforma es aplicable a los resultados de la planificación, no obstante, la aplicación realiza una serie de funciones básicas sobre estos resultados, es decir permite eliminar todos los eventos en caso de que se quiera deshacer la planificación actual, y también se encarga de actualizar los mismos cuando se hacen *replanificaciones* en el seguimiento de las tareas de manera automática.

1.3. Vocabulario

A continuación se precisará el significado de algunos términos utilizados a lo largo del texto.

◇ Tareas (el carácter de la tarea podrá ser periódico o esporádico):

- Periódicas: Una tarea periódica será aquella que implique dedicarle un tiempo concertado a una tarea con una cadencia establecida (e.g. practicar 3 horas diarias con un instrumento), se pedirán su frecuencia (qué días de la semana se llevará a cabo) y su duración.
- Esporádicas: En una tarea esporádica se indicará el tiempo total necesario para completar la tarea, que la aplicación posteriormente se encargará de distribuir adecuadamente para la consecución del objetivo antes de que acabe su plazo.(e.g. 150 horas para superar la asignatura de cálculo ó 3 horas para trabajo de Asignatura de Tecnología, etc . . .)

◇ Plazo: Hay en realidad dos intervalos de fechas que nos interesan.

- Agenda: El intervalo de fechas sobre las que tienen sentido las planificaciones (e.g. un semestre, un curso)
- Tareas: Si la tarea es esporádica será el día de comienzo y el día de finalización de la tarea. Si es periódica heredará los límites de la agenda y se planificará a lo largo de toda la extensión de ésta.

◇ Duración: para las tareas de *carácter esporádico* se les indicará tanto el tiempo estimado para la finalización de la tarea en horas (e.g. 400 horas para arrostrar la asignatura de cálculo) como , opcionalmente, el intervalo de tiempo razonable diario, ya que a determinadas tareas, sólo tiene sentido dedicarle un tiempo adecuado (en un día, un período demasiado pequeño sería insuficiente para haber entrado en materia y uno excesivo resultaría insostenible).

Caso de no indicar un tiempo razonable diario, se le da libertad a la aplicación para planificar la tarea como mejor convenga en las horas disponibles para el trabajo individual de ese día de la agenda. Si se indica un tiempo límite esto implica que en un mismo día no se planificarán más horas de esa tarea aunque fuese posible

◇ Preferencia: A los intervalos disponibles para realizar tareas, se les puede asignar una preferencia que es el indicador del intervalo (de los disponibles a lo largo de la semana que mejor conviene al usuario), hay cuatro grados posibles que se marcan en la creación de un intervalo, desde una estrella (*no quiero trabajar en este horario si es posible*) hasta cuatro estrellas (*es el horario idóneo*)

(a) Any.do [1] (b) Astrid [2]

Figura 1: Aplicaciones existentes

1.4. Aplicaciones existentes

De entre el nutrido grupo de aplicaciones de agenda y tareas disponibles para la plataforma *Android*, es bastante difícil encontrar aquellas que entren en el aspecto de hacer un reparto de trabajo en intervalos de tiempo disponible.

En general las aplicaciones suelen están enmarcadas en el contexto de las *"To-Do apps"* que generan eventos de agenda, de entre las cuales quizá cabe destacar *Astrid* o *Any.do*, pero ninguna realiza labores de planificación de tareas a lo largo de intervalos de tiempo de modo que una carga de trabajo se reparta en el tiempo. Generan recordatorios puntuales o periódicos y agendan notificaciones con una bella interfaz y multiplataforma.

Esta carencia para un campo que parece propicio como la enseñanza reglada, donde existen unos límites bien demarcados tanto en cargas de trabajo como en horarios, daría razón de ser a nuestra aplicación dentro del ecosistema, y quizás sirviera de buen baremo de carga de trabajo fuera del horario discente.

1.5. Equipamiento utilizado para el trabajo

Para el desarrollo del proyecto se han utilizado los siguientes elementos de software:

◇ Sistema operativo empleado: *Debian GNU-Linux 8 64bits*

◇ Herramientas de desarrollo: *Android Development Tools v.25.1.7*

◇ Bibliotecas de desarrollo: *Java 1.8.0*, *Android SDK rev.23*

◇ Hardware utilizado:

- Ordenador personal i7
- Móvil con S.O. *Android v. 4.2.1 "Jelly Bean"*

2. Descripción general de la aplicación

2.1. Requisitos de aplicación

Vamos a comenzar exponiendo los elementos que han sido utilizados para la construcción del modelo de la aplicación, describiendo someramente su función.

2.1.1. Agenda

Para poder programar la realización de tareas, en primer lugar nos planteamos un lapso de tiempo en el cual serán válidas las planificaciones. A éste período de tiempo lo llamaremos agenda, la cual constará en principio de una fecha de inicio y una fecha de fin. Dichas fechas servirán para aplicar un horario o plantilla semanal.

2.1.2. Horario o Plantilla Semanal

En una plantilla semanal se describirán los intervalos de tiempo (ocupado o disponible) para cada día de una *semana tipo*, es decir la plantilla que será aplicada a todas las semanas contenidas en el intervalo de tiempo de la agenda.

Una agenda podrá disponer de varias plantillas semanales dado que en el período marcado por la agenda un usuario puede y de hecho suele cambiar su gestión del tiempo.

> e.g. Es plausible pensar que los distintos semestres variarán la distribución de intervalos del horario de un alumno, así en una agenda con la duración de un año lectivo, pueden coexistir (no de modo simultáneo) distintos horarios ligados a ella.

2.1.3. Intervalos de una plantilla semanal

Los intervalos horarios dentro de una plantilla semanal pueden ser de dos tipos, en función de la clase de tarea que vayan a albergar.

- *Intervalo ocupado*: si el intervalo se va a utilizar para una tarea prefijada y constante como puede ser el horario de clase de un alumno, será un *intervalo ocupado* y sólo será usado con una finalidad informativa a la hora de mostrar los resultados de las planificaciones de un modo más completo, en el que no solo se muestren las tareas generadas por el planificador sino que también aparezcan aquellas a las que haya que atender obligatoriamente, así, de un vistazo a la agenda del dispositivo, se obtiene toda la información necesaria para una jornada concreta.
- *Intervalo disponible*: por otro lado tenemos aquellos intervalos horarios dentro de la plantilla semanal que usaremos como tiempo disponible para resolver tareas en cada

uno de los días de la plantilla semanal, estos intervalos indican tiempo disponible para la realización de tareas por parte del usuario.

2.1.4. Intervalos planificados

Hay otros tipos de intervalos, aquellos resultantes de aplicar una planificación, es decir, aquellos intervalos de tiempo que tienen asociada una tarea (mejor dicho una porción de ésta) que ha de ser llevada a cabo para el cumplimiento de una planificación.

- ◇ *Intervalo planificado tarea esporádica:* Es un lapso de tiempo proveniente del resultado de una planificación exitosa en la que la realización de una *tarea* es prorrateada en los intervalos disponibles de modo que logra acabarse antes de que expire su fecha de finalización.

- ◇ *Intervalo planificado tarea periódica:* Tiene un matiz que lo distingue del anterior y es que este tipo de períodos son asignados a una serie de días concretos, determinados en la plantilla semanal, éstos días son introducidos en la declaración de una nueva tarea periódica junto con la duración de la tarea para un día y se aplicará a lo largo de toda la "vida" de la agenda.

  Un ejemplo sería practicar con la harmónica media hora lunes y miércoles en el tiempo disponible. Este tipo de tarea provoca que el planificador genere una serie de intervalos hasta la fecha del final de la agenda (y no de la finalización de la tarea, que aquí no tiene sentido) en los que todos los lunes y miércoles se buscaría un hueco de media hora dentro del tiempo disponible del usuario para poder practicar.

2.1.5. Tareas

Las tareas son la descripción del trabajo a ser realizado en los intervalos disponibles de modo que el planificador tenga suficiente información como para disponerlos de modo que cumplan sus *deadlines* y también satisfaga las preferencias del usuario.

Toda tarea consta de un nombre que será usado en cualquier representación posterior en la agenda del dispositivo con que se visualice. Hay definidos dos tipos de tareas:

- ◇ *Tarea periódica:* éste tipo de tareas son aquellas que el usuario va a repetir unos días concretos cada semana, a lo largo de la duración del calendario e.g. "correr una hora sábados y miércoles", a parte del nombre tienen asociados los días de la semana en que habrán de ser planificadas y también la duración que tendrá. Es decir constan de:

- Nombre
- Duración de la actividad
- Días de la semana en que será ejecutada

◇ *Tarea esporádica:* Las tareas esporádicas son en definitiva una carga de trabajo que ha de ser resuelta antes de un plazo, con respecto a la carga de trabajo nos basamos en que las enseñanzas regladas disponen de una estimación del tiempo que habrá de ser dedicado a las tareas, necesitaran una fecha de comienzo y de finalización de la tarea (contenidas dentro de los límites de la agenda), y finalmente, para que las planificaciones sean adecuadas para ser llevadas a cabo por personas, éstas tienen un límite de tiempo para la realización de la misma de forma continuada.

> Esto es; por mucho tiempo disponible que tenga el usuario un día determinado, el planificador no permitirá que la tarea consuma de ese intervalo más que dicho límite, así, si se dispone de varias horas una tarde, una tarea extensa no ocupará más de lo que se haya estipulado para ella como máximo tiempo dedicado de forma continua.

En resumen constan de:

- Nombre
- Fecha de inicio
- Fecha fin
- Tiempo total estimado para la realización de la tarea
- Tiempo máximo dedicado de forma continua

2.1.6. Motor de planificación

El motor de planificación es, en principio, un algoritmo de scheduling, encargado de realizar una planificación.

Esto es que, dada una agenda y una serie de tareas, se encargará de devolver una lista de intervalos planificados dentro de los límites de la agenda de modo que todas las tareas se ejecuten si es posible dentro de sus tiempos límites y respetando las preferencias de los usuarios como pueda ser el máximo tiempo que se dedicará de modo continuado a una tarea, o bien de devolver un informe de que la planificación no es factible con las condiciones dadas.

Decimos que en principio será un algoritmo de scheduling, porque en realidad no se fuerza a que sea de este modo, los motivos para que los algoritmos sean de scheduling y

no, por ejemplo, de planning o CSP's serán discutidos más adelante, pero se ha tenido especial cuidado a la hora implementar esta parte del código aplicando patrones de diseño (en este caso strategy también llamado policy) de modo que no haya problema a la hora de añadir otros algoritmos que difieran de los elegidos.

La aplicación consta inicialmente con tres algoritmos de planificación de scheduling, basados en los algoritmos *DMS*, *RMS* (prioridades estáticas) y *EDF* (prioridades dinámicas).

2.1.7. Contexto de planificación

Una vez nuestro algoritmo de planificación haya hecho su labor, nos devolverá o bien un mensaje indicando que para el algoritmo elegido, con la agenda y las tareas seleccionadas, no ha sido posible encontrar una planificación válida, o bien un conjunto de intervalos planificados a lo largo del período de duración de la agenda, que es una planificación válida.

Que una planificación sea válida no implica que sea cierto todo lo que pone en ella, es decir, con el paso del tiempo (desde que la planificación fuera inicialmente realizada), es posible que un usuario haya cumplido con las tareas planificadas o que no, entonces necesitamos de un mecanismo que nos permita llevar el control de qué parte de la planificación es realmente trabajo realizado en un momento dado.

En función de la información suministrada por el usuario acerca del trabajo hecho podremos estimar de nuevo el tiempo restante de cada una de las tareas y volver a lanzar la planificación en caso necesario (siempre que no se haya cumplido alguna de las planificaciones, porque si se han ido cumpliendo las tareas en fecha, la planificación inicial continuaría siendo válida) retocando debidamente las condiciones iniciales.

Para poder llevar adecuadamente el control de las tareas que ya han sido planificadas se ha pensado en un *contexto de planificación* que está compuesto de una referencia a la *agenda*, una referencia (no una copia) a las *tareas* de las cuales además se conserva el tiempo que verdaderamente se ha dedicado a ellas, el *motor de planificación* que fue usado y el resultado de la misma, y por último una fecha que actuará como hito.

Es decir un contexto de planificación, grosso modo, consta o al menos contiene referencias de :

- ◇ Agenda
- ◇ Tareas que el usuario quería planificar
- ◇ Tiempos reales dedicados a cada una de las tareas

- ◇ Motor de planificación
- ◇ Resultado de los intervalos planificados
- ◇ Fecha en que fue realizada la ultima planificación

La fecha del contexto de planificación es fundamental a la hora de chequear si las tareas se van llevando al día o no. Cuando se planifican una serie de tareas en una agenda (con su plantilla semanal ya seleccionada) por primera vez, se creará un contexto de planificación con la fecha de creación, cada vez que el usuario acceda a la aplicación se contrastará la fecha actual del dispositivo con la del contexto de planificación y si hay tareas que debieran ser hechas entre ambas fechas se le preguntará al usuario si las ha realizado o no, y así se procederá a actualizar el contenido del contexto de planificación convenientemente para futuras comprobaciones (replanificando si fuese necesario y actualizando la fecha del contexto de planificación a la actual).

Nótese que la comprobación de la cantidad de trabajo real realizado de las tareas estará en función exclusivamente del uso que se haga de la aplicación, cuanto más tiempo se dilate el regreso a ésta, mayor cantidad de trabajo habrá de ser confirmado o procrastinado para cada tarea, pero se ha considerado que es mejor relegar el control al usuario, que andar incordiando con avisos a cada intervalo planificado de cada tarea, o cualquier otra forma de intrusismo en el quehacer de éste.

2.1.8. Uniendo las piezas

La idea es que un usuario defina un marco de tiempo (que hemos llamado *agenda*) y pueda conjugar tantos horarios (*plantillas semanales*) como necesite, así tendremos una agenda con un conjunto de plantillas semanales seleccionables, de las que sólo una será la activa.

Con esos dos elementos una agenda será capaz en el futuro de suministrar al planificador de tareas una lista de intervalos de tiempo disponibles, así éste solo tendrá que preocuparse, en principio, de ir consumiendo dicho tiempo a su elección.

Por otro lado tenemos las tareas que son la otra parte necesaria al planificador para realizar su función de distribuir el tiempo libre extraído de la agenda entre las tareas teniendo en cuenta las preferencias del usuario en éstas como máximo tiempo seguido, deadlines, etc.

En este punto hemos alimentado un planificador con *tareas* y *agenda* y: o bien obtenemos una planificación válida, o nos informaría de que no es posible la misma, pero aún nos queda contemplar el hecho de que ocurran imprevistos y las tareas no se vayan llevando a cabo, entonces utilizamos el mecanismo del contexto de planificación para evaluar lo hecho realmente hasta la fecha y en caso necesario volver a lanzar planificaciones con los datos de las tareas actualizados.

2.2. Flujo de datos

Procuraremos mostrar con muy pocos elementos la información requerida por la aplicación y el tratamiento que se da a los datos en el sistema desde un alto nivel de abstracción Usaremos para ello un diagrama *de flujo de datos de alto nivel* Figura [2], en el que ya se plasman los procesos que describen el proceso principal.

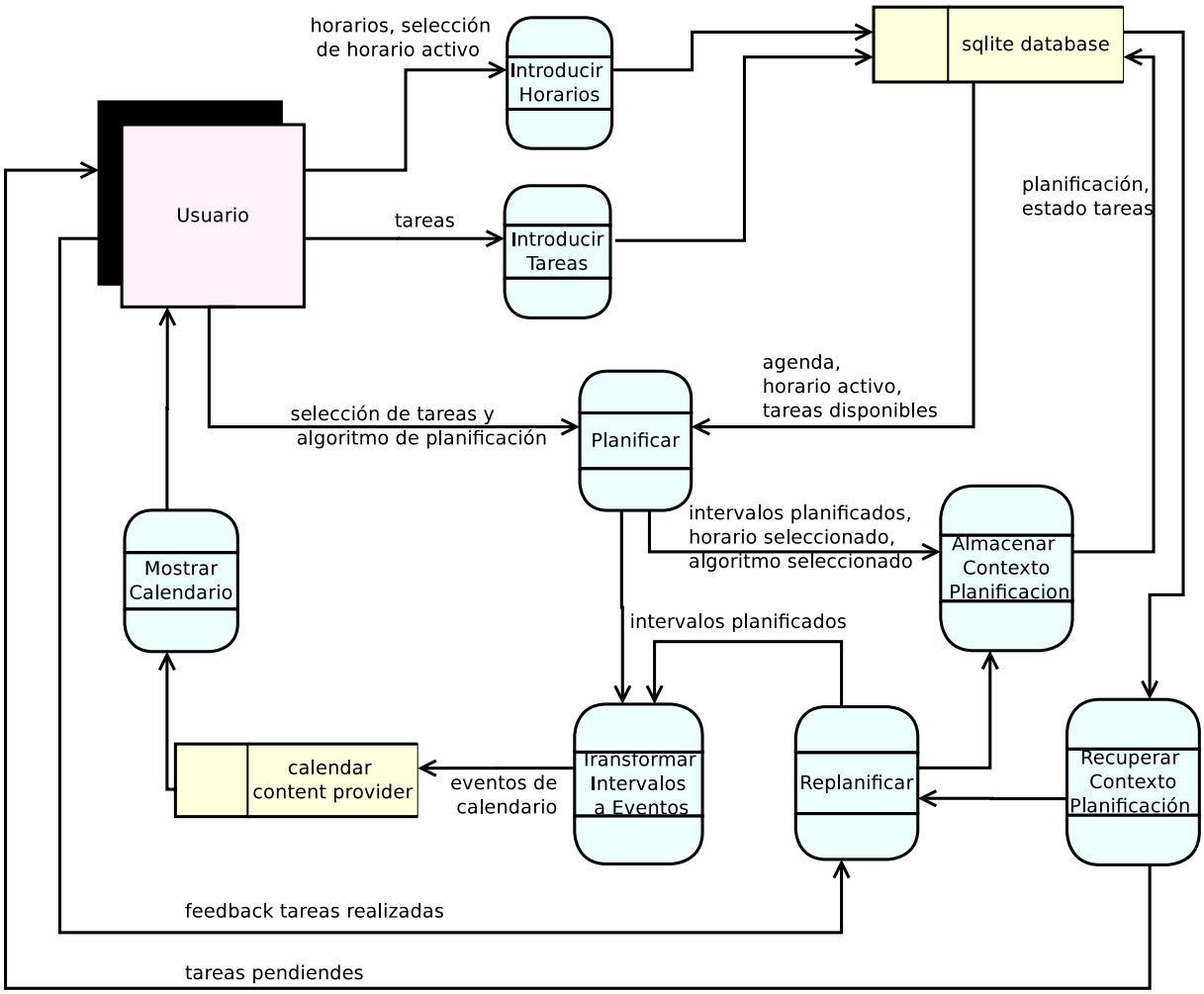


Figura 2: Flujo de datos de la aplicación

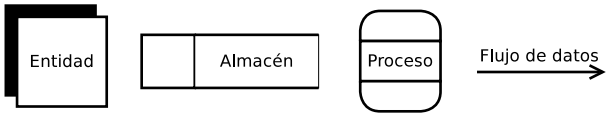


Figura 3: Elementos del grafo Figura [2]

Este tipo de acercamiento al análisis del problema nos ha servido para en primer lugar tener una idea más cercana a la realidad de la implementación de nuestra aplicación, y además también facilita una posterior reflexión acerca de las distintas posibilidades de desarrollo [3].

3. Algoritmos de planificación

El planificador (*scheduler*) es un componente funcional muy importante de los sistemas operativos, y esencial en los sistemas operativos de tiempo real. Su misión consiste en repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución.

Todo sistema operativo gestiona los programas mediante el concepto de proceso. En un instante dado, en el ordenador pueden existir diversos procesos listos para ser ejecutados, sin embargo, solamente uno de ellos puede ser ejecutado (en cada microprocesador). De ahí la necesidad de que una parte del sistema operativo gestione, de una manera equitativa, qué proceso debe ejecutarse en cada momento para hacer un uso eficiente del procesador

En los sistemas operativos en tiempo real, su comportamiento se caracteriza por garantizar que todo programa se ejecutará dentro de un límite máximo de tiempo. El planificador debe comportarse de manera que esto sea cierto para cualquier proceso.

Expresado en términos de la Teoría de la Planificación de Tareas de Tiempo Real (o *Real-Time Scheduling Theory*) [4] define

Propósito Satisfacer las restricciones temporales de las múltiples tareas de un sistema informático en tiempo real.

Procedimiento Planificar los recursos del sistema (*CPUs*) de acuerdo a ciertos algoritmos, de tal manera que la temporización del sistema sea predecible, comprensible y mantenible.

En estos casos, la finalidad del planificador es balancear o equilibrar la carga del procesador, impidiendo que un proceso monopolice el procesador o que sea privado de los recursos de la máquina.

En entornos de tiempo real, como los dispositivos para el control automático en la industria (por ejemplo, robots), el planificador también impide que los procesos se paren o interrumpan a otros que esperan que se realicen ciertas acciones. Su labor resulta imprescindible para mantener el sistema estable y funcionando.

Existen distintos niveles de planificación (basados en la frecuencia con la que se realiza cada uno), en los sistemas operativos de propósito general, existen tres tipos de planificadores.

- ◇ *Planificador a corto plazo*: también denominados *dispatcher* o *short term scheduler*, es el que se ha descrito previamente, siendo también el más importante.
- ◇ *Planificador a medio plazo* (*mid term scheduler*) está relacionado con aquellos procesos que no se encuentran en memoria principal (memoria virtual). Su misión es mover procesos entre memoria principal y disco (lo que se conoce como *swapping*)
- ◇ *Planificador a largo plazo* (*long term scheduler*) es el encargado de introducir nuevos procesos en el sistema y de finalizarlos.

Hay numerosas y variadas políticas de planificación, a continuación se enumeran algunas aunque lo habitual es utilizar políticas mixtas.

- ◇ Round-robin
- ◇ Round-robin con pesos
- ◇ Prioridades monótonas en frecuencia (RMS (Rate-monotonic scheduling))
- ◇ *EDF* (Earliest deadline first scheduling) o Menor tiempo de respuesta primero
- ◇ *FIFO* También conocido como FCFS "First Come, First Served"
- ◇ *SJF* Shortest Job First
- ◇ *CFS* Completely Fair Scheduler (ó Planificador Completamente Justo)
- ◇ *SRT* Shortest Remaining Time
- ◇ *SPT* Shortest Process Time
- ◇ Planificación mediante colas multinivel

Es habitual que se mezclen, es decir, podría ser que el planificador a corto plazo utilice *round-robin*, mientras que el planificador a largo plazo use varias colas *FIFO* y cada una de esas colas corresponda a una prioridad diferente.

Desde una perspectiva diferente existen dos tipos de algoritmos de planificación

Expropiativos : Éstos generan planificaciones en las que se puede cambiar la CPU de una tarea a otra en cualquier momento si así lo desea el planificador

No expropiativos : Los no expropiativos permiten que se ejecute el proceso hasta que acabe su trabajo. Es decir, una vez les llega el turno de ejecutarse, no dejarán libre la CPU hasta que terminen o se bloqueen.

3.1. Tabla comparativa algoritmos

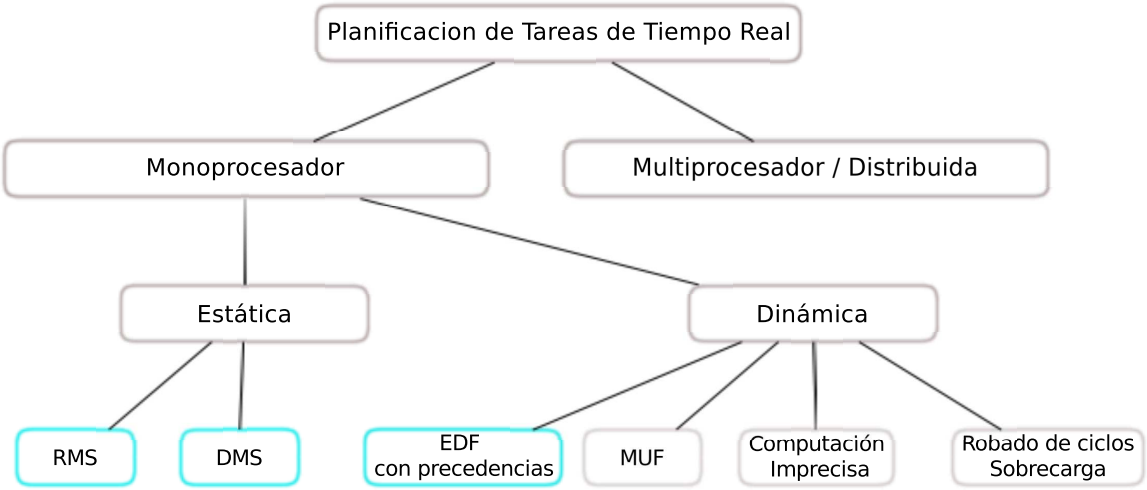


Figura 4: Pequeño esquema de los algoritmos de *Scheduling*

De entre todos los posibles algoritmos de *Scheduling* disponibles, hemos elegido los algoritmos *RMS* , *DMS* y *EDF* que procederemos a analizar posteriormente.

El criterio de elección de dichos algoritmos es:

- ◇ En primer lugar el hecho de que los algoritmos debían ser para sistemas monoprocesador, debido a que un ser humano no es capaz de realizar varias tareas de aprendizaje simultánemente
- ◇ El segundo es que entre los algoritmos elegidos debían estar tanto los que usen prioridades dinámicas (por ejemplo *EDF*) como aquellos que utilicen prioridades estáticas (como *RMS* o *DMS*)
- ◇ Todos los algoritmos deben permitir la planificación expulsiva (es decir que se puede cambiar la ejecución de una tarea a otra en cualquier momento si así lo desea el planificador), debido a que las tareas a realizar por el alumno tendrán unos tiempos límite de trabajo contínuo y necesitarmos cambiar de tarea para continuar el tiempo de estudio con otra.

Los tres algoritmos elegidos cumplen estas propiedades.

3.1.1. El algoritmo *RMS*

Es un algoritmo que utiliza prioridades estáticas (las prioridades de todos los procesos permanecen constantes a lo largo del tiempo).

Habría que puntualizar el hecho de que usualmente se realiza (erróneamente) una asignación de prioridades estáticas de acuerdo con la importancia que se atribuye al proceso,

y éste método de asignación de prioridades no es óptimo (el algoritmo se dice óptimo si en el caso de existir una solución, la encuentra) y puede conducir a fallos en el tiempo de respuesta de algún proceso, incluso en el caso en que el procesador esté poco utilizado.

Así que la importancia subjetiva no es uno de los criterios válidos para la asignación de las prioridades a los procesos que forman el sistema en tiempo real. Se puede comprobar que bajo determinadas condiciones, la asignación de prioridades de forma que el *proceso con menor período tenga mayor prioridad*, es óptima.

Este tipo de planificación se denomina Asignación Monotónica en Frecuencia (*RMS*), y es óptima para prioridades estáticas, es decir, si un sistema con prioridades estáticas tiene alguna planificación admisible (una planificación admisible es aquella que respeta las restricciones temporales (*deadlines*) de todas las tareas, ya sea con sus tiempos de cómputo máximos (tiempo real duro) o con sus tiempos medios (tiempo real suave), la *RMS* encontrará una planificación admisible.

Se define el factor de utilización del procesador como la fracción de tiempo que éste está ocupado ejecutando procesos. Este valor viene dado por la siguiente expresión:

$$U = \sum \frac{c_i}{p_i} \tag{1}$$

Siendo c_i el tiempo de cómputo del proceso y p_i su periodo. Se dice que un sistema utiliza completamente el procesador si la planificación *RMS* es admisible, y cualquier incremento en el tiempo de cómputo de un proceso hace que la planificación sea inadmisible. Como *Rms* es óptimo, si no es posible ninguna planificación admisible con *Rms*, no hay ninguna otra planificación con prioridad estática que sea admisible.

Se define *Factor de Utilización Garantizado* para n procesos (UG_n) , como el valor mínimo del factor de utilización entre los correspondientes a todos los sistemas en tiempo real con n procesos que utilizan completamente el procesador. Como se trata de un valor mínimo, puede haber casos de sistemas de tiempo real en los que sea posible alcanzar factores de utilización mayores que el garantizado. El sentido de este valor es que se garantiza que exista una planificación admisible siempre que:

$$U \leq UG_n \tag{2}$$

Se puede demostrar que para el caso de la planificación *Rms*

$$UG_n = n(2^{\frac{1}{n}} - 1) \tag{3}$$

Y para valores grandes de n, esto tiende al logaritmo neperiano de dos

$$\lim_{n\to\infty} UG_n \simeq \ln 2 \simeq 0{,}69 \tag{4}$$

De ahí se deduce que *Rms* es admisible cuando se cumple que $U < 0{,}7$, es decir, *Rms* es admisible cuando el factor de utilización del procesador es menor que el 70 %, para un número de procesos alto.

Se puede demostrar que si no existieran restricciones sobre el tiempo de respuesta, se podría obtener una planificación admisible para cualquier conjunto de procesos que verificaran la condición: $U \leq 1$

Sin embargo, la exigencia de respetar los tiempos de respuesta especificados limita el factor de utilización máximo a un valor menor que la unidad.

Por tanto, sólo se puede asegurar que existe una planificación admisible con prioridad estática si el factor de utilización es inferior a 0.7 . No obstante, en algunos casos, se pueden alcanzar valores más altos del factor de utilización. [4]

3.1.2. El algoritmo *DMS*

En este caso, no hay diferencias sustanciales con respecto al algoritmo *RMS*, en este caso el algoritmo *DMS* (*Deadline Monotonic Scheduling*) de asignación de prioridades estáticas, se basa en el criterio de asignar dichas prioridades a cada una de las tareas de acuerdo a su *deadline*.

A la tarea con el *deadline* más cercano le será asignada la mayor prioridad. Esta política de asignación de prioridades es *óptima* para un conjunto de tareas esporádicas o periódicas que cumplan con el siguiente modelo.

- ◇ Todas las tareas han de tener *deadlines* menores o iguales que su mínimo período
- ◇ Todas las tareas tienen tiempos de ejecución en el peor caso, que serán menores o iguales a sus *deadlines*
- ◇ Todas las tareas son independientes y ninguna bloqueará la ejecución de otra (por ejemplo accediendo a recursos compartidos mutuamente excluyentes *mutually exclusive shared resources*)
- ◇ Ninguna tarea puede suspenderse de modo autónomo
- ◇ Consideraremos que no consume recursos ni el lanzar una tarea (*release time*), ni tampoco el intercambiar la ejecución de una tarea por otra

3.1.3. El algoritmo *EDF*

Es un algoritmo de planificación dinámica, esto conlleva varios factores

- ◇ la planificación dinámica es más flexible.
- ◇ es necesaria para ciertos tipos de sistemas.
- ◇ permite contemplar diversos escenarios.
- ◇ no es estable.
- ◇ es un tema abierto de investigación.

En concreto, nuestro algoritmo utiliza el *deadline* de las tareas como la base para tomar decisiones de planificación.

El método de planificación por prioridad al *deadline* más cercano (*Edf* Earlier Deadline First) consiste en asignar en cada instante la prioridad más alta al proceso cuyo tiempo límite está más próximo. El tiempo límite de un proceso P_i en un instante t es el valor:

$$L_i(t) = a_i(t) + r_i \tag{5}$$

Donde $a_i(t)$ es el instante en que se ha producido la última activación del proceso P_i aún no satisfecha en t y r_i es el tiempo máximo de respuesta admisible para el proceso.

Si P_i no tiene ninguna activación pendiente en t, se supone que $L_i(t) = \infty$, por tanto, el proceso que se ejecuta en el instante t es aquel cuyo valor de $L_i(t)$ es mínimo.

Se asume que el plazo absoluto de una tarea es fijo y constante durante el tiempo de vida de la tarea. Cuando se utiliza el algoritmo *Edf* se exige que todas las tareas cumplan con sus plazos.

Esta condición puede verificarse previamente si conocemos todos los parámetros de las tareas que habrá en el sistema. La carga de trabajo del sistema es estática si los parámetros no cambian, o puede ser dinámica si nuevas tareas van llegando al sistema, de modo que cada nueva incorporación que se realice conlleva un nuevo test de planificabilidad, en este caso $U < 1$ [4].

Se pueden garantizar el cumplimiento de los *deadlines* si:

$$U = \sum_{n=1}^{n} \frac{c_i}{p_i} \tag{6}$$

Un ejemplo del efecto de la utilización del algoritmo podría ser, dadas unas tareas de ejemplo con las siguientes características

Tarea	Periodo (p_i)	Duración (c_i)	Factor de Uso (U)
Tarea 1	30	10	0,33
Tarea 2	40	10	0,25
Tarea 3	50	12	0,24
U_{total}=0,823	luego el sistema cumple el test de planificabilidad		

Figura 5: Datos del algoritmo *EDF*

Ofrecería esta planificación

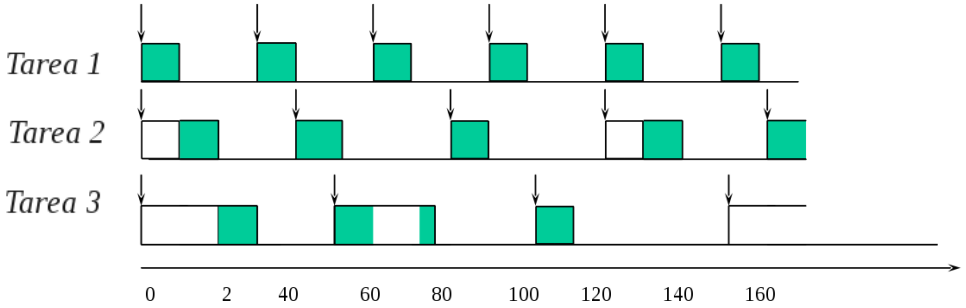


Figura 6: Ejemplo de uso del algoritmo *Edf*

3.2. Adaptación de algoritmos de planificación

El hecho de que existan algoritmos de uso tan común como los planificadores en el mundo de los sistemas operativos, (de vital importancia sobre todo los empleados en sistemas operativos de tiempo real, en los cuales se garantiza que las tareas han de ser ejecutadas antes de un determinado tiempo límite), hace pensar que quizás podrían ser herramientas útiles para otro tipo de reparto de recursos, en vez de tiempo de *CPU* tiempo libre de un usuario.

En general, todo sistema operativo gestiona sus programas mediante el concepto de proceso, en un instante dado, en el ordenador pueden existir diversos procesos listos para ser ejecutados, al igual que en un instante dado un estudiante puede disponer de varias tareas pendientes, y del mismo modo, si determinados procesos de un sistema en tiempo real requieren ser realizados antes de un tiempo límite, así ocurre también con las tareas del estudiante.

Si en el caso de los planificadores para sistemas monoprocesador sólo un proceso puede ser ejecutado a la vez, en el caso del estudiante sólo una tarea puede ser llevada a cabo simultáneamente, el modo en el que se repartan las tareas pendientes es precisamente

determinado por el algoritmo de *scheduling* utilizado.

Estos algoritmos cuidan mucho la gestión de los recursos disponibles, han de ser extremadamente eficientes ya que su uso es tan intensivo por parte del núcleo del sistema operativo que podrían degradar el rendimiento de todo el sistema.

Esta es una cualidad muy interesante, porque las plataformas móviles disponen de muchas limitaciones, usar este tipo de algoritmos redundará en una mayor vida útil de la batería y unos buenos tiempos de respuesta.

El tiempo de un alumno que esté siendo víctima de una enseñanza reglada suele estar marcado por un conjunto de horas designadas por la institución en las que se tendrán que realizar una serie de actividades y otros tiempos que habrá de reservarse el alumno para el estudio y el trabajo.

El tiempo reservado por el propio alumno para el trabajo en casa será el equivalente al tiempo de *CPU* disponible, es decir su mas preciado recurso como estudiante y el tiempo que deberá dedicar a tareas y estudio serán los procesos que habrán de ser finalizados antes de que lleguen determinados *deadlines* como son fechas de exámenes y de entregas de trabajos.

Con estos mimbres la aplicación necesitará

- Un horario compuesto de las horas de trabajo de la institución (esto es opcional pero la representación posterior en la agenda del dispositivo queda más coherente y completa) y *además* horas disponibles para estudio designadas por el alumno
- Un conjunto de tareas (esporádicas) que habrán de ser finalizadas antes de un plazo (e.g. dedicar un tiempo determinado a un tema de una asignatura antes de su evaluación), o periódicas (aquellas que serán repetidas ciertos días de la semana a lo largo de toda la extensión del calendario)

Como (por fortuna), las personas no somos máquinas ni nuestros cerebros *CPU's*, hay ciertas condiciones (humanitarias) que se pueden exigir a las tareas como el hecho de que no se esté trabajando de manera continuada cierta materia más de un determinado tiempo, o la preferencia de unas horas o días frente a otros a la hora de acometer tareas.

De este modo los distintos algoritmos de planificación podrán hacer su trabajo, si bien es cierto que algunas condiciones como la optimalidad de los algoritmos se podrían ver mermadas por las opciones adicionales añadidas por los usuarios a la realización de las tareas, como el tiempo máximo continuado para una tarea en concreto, o el hecho de que

por razones peregrinas éstas tareas puedan procrastinarse (ya que si entre dos accesos a la aplicación hubiera un trabajo que desarrollar ésta preguntará por la realización del mismo) y ello forzaría a la aplicación a volver a planificar las tareas teniendo en cuenta que hasta el momento algunas habrían sido parcialmente realizadas, otras habrán terminado y algunas estarían como al principio, y sería posible que ya no haya una planificación válida.

3.3. Problemas de satisfacción de restricciones (*CSP*)

La programación por restricciones es una metodología software utilizada para la descripción y posterior resolución efectiva de problemas típicamente combinatorios y de optimización, que son de aparición frecuente en áreas como *IA*, *Investigación operativa de sistemas*, *bases de datos*, etc...

Una vez se modelan esos problemas adecuadamente como problemas de satisfacción de restricciones se podrán utilizar las técnicas que éste campo del conocimiento nos ofrece.

Es decir, las etapas básicas de resolución de este tipo de problemas son

- ◇ Modelización
- ◇ Resolución mediante técnicas *CSP* específicas
  - Procesos de búsqueda apoyados por heurísticos
  - Procesos inferenciales

La idea en la que se fundamenta la programación de restricciones es resolver problemas mediante la declaración de restricciones sobre el *dominio* del problema, y en consecuencia encontrar soluciones a instancias de los problemas de dicho dominio que *satisfagan* todas las restricciones, y, en su caso, optimicen unos criterios determinados

3.3.1. Terminología y conceptos elementales

La programación de restricciones se divide en dos áreas claramente distintas que compartan terminología que son

- ◇ Satisfacción de restricciones (dominios finitos)
- ◇ Resolución de restricciones (dominitos infinitos, o lo suficientemente complejos para tratarlos así)

En nuestro caso, trataremos con problemas de dominio finito, (es decir *satisfacción de restricciones*) la adaptación a nuestro objetivo será expuesta con detalle en el siguiente punto.

La terminología será la siguiente

Modelización del problema Representación de un problema mediante un conjunto finito de variables y un dominio finito de valores para cada variable y un conjunto de restricciones que acotan las combinaciones válidas de valores para dichas variables. En este punto es crucial una correcta formalización del problema para no

olvidar ningún aspecto significativo para la resolución adecuada. Aquí hay que tener presente dos conceptos básicos

- ◇ La *potencia expresiva* de las restricciones, es la correcta interpretación de las restricciones y su paso al modelo
- ◇ La *eficiencia* de la representación, ya que dependiendo de la modelización el problema éste se resolverá de un más o menos eficaz, una mala representación redunda entre otros efectos en ciclos de trabajo innecesarios

Técnicas inferenciales Son las que deducen nueva información a partir de la información presentada de modo explícito. En este punto es interesante resaltar que dichas técnicas pueden (y deben) acotar y hacer más eficiente el proceso de búsqueda de soluciones.

Técnicas de búsqueda de solución Normalmente guiadas por heurísticos (que pueden ser dependientes o independientes del dominio de las variables), son técnicas que buscan finalmente un *valor* para cada variable que satisfaga todas las restricciones impuestas.

3.3.2. Definición de un problema *CSP*

Un problema de satisfacción de restricciones puede ser representado mediante una terna (X, D, C) donde

- ◇ X Es un conjunto de n variables $x_1, \ldots, x_n$
- ◇ $D =< D_1, \ldots, D_n >$ es una tupla de dominios finitos donde se interpretan las variables X, de tal modo que la i-ésima componente D es el dominio que alberga los posibles valores de x_i
- ◇ Y por último el conjunto de restricciones $C = c_1, \ldots, c_n$. Cada restricción está definida sobre un conjunto de k variables $var(c_i) \subseteq X$ que se denomina su ámbito, y restringe los valores que pueden de un modo simultáneo tomar las variables. Huelga decir que toda solución del problema ha de satisfacer *todas* las restricciones.

Así expuestos los términos una *instanciación* de variables es una dupla (x, a) que representa la asignación del valor a a la variable x . La instanciación de un *conjunto de variables* es una tupla de pares ordenados, donde a cada par se le asigna un valor. Bien pues se dice que una tupla es *localmente consistente* si satisface todas las restricciones formadas por variables $x_1, \ldots, x_i$ de la tupla.

Igualmente un valor $a_i \in D$ es *consistente* para x_i si existe al menos una solución del problema en la cual a la variable se le asigne dicho valor.

Finalmente una *solución* a un CSP, no es más que una asignación $(a_1, a_2, \ldots, a_n)$ de valores a todas sus variables de tal manera que se satisfagan *todas* las restricciones C, luego una solución es una tupla consistente que contenga a todas la variables del problema, y una solución parcial sería una tupla consistente con un subconjunto de X.

Un CSP es consistente si tiene al menos *una solución* es decir, una tupla consistente.

3.3.3. Restricciones

Éstan caracterizadas por su aridad, es decir el número de variables involucradas en la restricción, se clasifican en tres grupos, *unaria*, *binaria*, $n - aria$ *o* $no - binaria$ que son las que involucran a un número arbitrario de al menos 3 variables

Se pueden definir por extensión o por comprensión (en alguna bibliografía *intensionalmente*) mediante alguna función aritmética.

- ◇ La representación **por extensión** está formada por un conjunto determinado de tuplas cada una con k elementos y expresa el conjunto de valores que pueden tomar las k variables simultáneamente. Si el CSP estuviera sobre un dominio de valores contínuo sería imposible representar las restricciones por extensión.
- ◇ La representación **por comprensión** aquí podemos tener muchos tipos *disyuntivas* o *no-disyuntivas* en función de si expresan una o más de una relación disyuntiva entre las variables **e.g.**
  - *disyuntiva* $x_1 < x_2$
  - *no-disyuntiva* $x_1 < x_2 \vee x_2 < x_1$, es decir $x_1 \neq x_2$
- ◇ Las restricciones también pueden ser *cualitativas* cuando expresan una relación de orden entre variables
- ◇ O *métricas* si lo que expresan es una distancia métrica en el dominio de interpretación de las variables

3.3.4. Técnicas

Las técnicas que suelen llevarse a cabo para manejar un *CSP* son de tres tipos: *búsqueda sistemática*, técnicas inferenciales y técnicas híbridas.

Métodos de búsqueda Éstos se centran en explorar el espacio de estados del problema, estos métodos pueden ser *completos* si exploran todo el espacio de estados en

busca de una solución o bien *incompletos* si lo hacen parcialmente. Los completos garantizan encontrar una solución si existe, o bien demuestran que un determinado problema no es resoluble (de un modo exhaustivo), computacionalmente son muy costosos . Los dos métodos completos más importantes son

- ◇ *Generar y Testear (Generate and test)* Se generan las posibles tuplas de instanciación de todas las variables de forma sistemática y después se testea sucesivante sobre cada instanciación y la primera combinación que satisfaga todas las restricciones del problema, será la solución. Mediante este procedimiento el número de combinaciones generado es el producto cartesiano de la cardinalidad de los dominios de las variables.

$$\prod_{i=1,n} d_i$$

  Lo que implica necesariamente (en un problema real) muchas instanciaciones erróneas que serán rechazadas en la fase de testeo.

- ◇ *Backtracking Cronológico* Este método realiza una exploración en profundidad del espacio de búsqueda, instanciando sucesivamente las variables y comprobando ante cada nueva instanciación si las instanciaciones llevadas hasta ahora son localmente consistentes, si es así sigue con la instanciación si no, hace backtracking

Técnicas de inferencia Tienen como objetivo deducir nuevas restricciones que serán derivadas de las ya conocidas. De hecho estas técnicas van reduciendo del dominio de las variables los valores inconsistentes, o inducen restricciones implícitas entre las variables obteniendo un nuevo *CSP* equivalente con unas restricciones más fuertes, de este modo se logran dos objetivos.

- ◇ Obtener respuestas a preguntas relacionadas con las restricciones implícitas del problema original o sobre sus dominios
- ◇ Reducir el espacio de soluciones, al haber reducido el dominio (lo que también hace más eficientes los procesos de búsqueda)

Para lograr esto se comprueban las consistencias de *nodo*, arco ($2 - consistencia$), camino ($3 - consistencia$), global ($k - consistente$) que quizás queden fuera del objeto de este trabajo.

Técnicas híbridas Las *técnicas inferenciales* se utilizan como etapas de preproceso donde se detectan y se eliminan inconsistencias locales antes de que comience la etapa de *búsqueda*, con el fin de reducir este árbol, a mayor nivel de consistencia exigido

en el preproceso mayor tiempo de cómputo exigirá aunque más simplificado quedará el árbol de búsqueda, finalmente habrá que elegir un compromiso entre ambos.[5]

3.4. Inclusión de técnicas *CSP* en las planificaciones

En nuestra apliación disponemos de una serie de algorimos que son capaces de dar una planificación temporal de un conjunto de tareas a ser distribuidas sobre una determinada agenda.

Estos algoritmos bien conocidos que utilizamos fueron creados originalmente para el reparto de T_{CPU} (tiempo de *CPU*) entre tareas del sistema operativo y de los usuarios del sistema, son algoritmos que han de ocupar la menor cantidad posible de T_{CPU}, ya que los sistemas operativos **no deben** ocupar tiempo de proceso, el tiempo de proceso debe usarse en desempeñar las tareas del usuario; luego *a priori* habrían de resultar planificaciones (no muy "*finas*") pero con un coste temporal bien acotado y pequeño.

Como se vio que era viable adaptar estos algoritmos para nuestro objetivo (esto es realizar un reparto de tiempo disponible del alumno trasponiendo de algún modo procesos a trabajos de alumno y T_{CPU} a tiempo disponible de dicho alumno para estudiar) ahora el objetivo es dotar de cierta "*inteligencia*" a dichas planificaciones aprovechándonos de la velocidad con que se calculan.

Inicialmente se resolvían plafinicaciones sin tener en cuenta ningún otro criterio que no fuese el de los propios algoritmos anteriormente aquí descritos. Pero utilizando técnicas como *CSP* podemos tratar de mejorar el resultado teniendo en cuenta las preferencias del usuario.

Hasta ahora las limitaciones eran las siguientes.

- ◇ Las tareas se terminaran antes de que vencieran sus *deadlines* o antes de que finalizara el intervalo de tiempo en que esa *agenda* era válida.
- ◇ Las tareas no excediesen por cada día el máximo tiempo razonable para su realización, esto es en su definición se acotaba el máximo tiempo en un día que se podía dedicar a una determinada tarea, para "humanizar" el desempeño de trabajo.
- ◇ Las tareas periódicas tuvieran cabida en todos y cada uno de los días de la semana en que estaban programadas a lo largo de la vigencia del calendario

Si bien esto se resolvía, el conjunto de resultados posibles con *planificaciones admisibles* (es decir, que respetaran los *deadlines* de todas las tareas) podía ser muy amplio

y simplemente se devolvía la primera solución que resolviera el algoritmo de *scheduling* elegido (algoritmo modificado para satisfacer las restricciones anteriormente expuestas).

Figura 7: Seleccion optimalidad del intervalo

Ahora podemos tratar de enfrentarnos al problema de tener en cuenta la **calidad** de dicha solución de la siguiente manera: cuando el usuario está elaborando un horario, a la hora de introducir un intervalo de tiempo disponible para realizar tareas se le solicita que informe además de su grado de conveniencia a la hora de emplear ese tiempo para trabajar, así podemos disponer de una *prelación* de intervalos de tiempo de trabajo a la hora de planificar tareas de acuerdo las preferencias de trabajar en dichas franjas horarias por parte del usuario.

Bien, una vez captadas las preferencias por determinadas franjas temporales del alumno, se ha tratado de aprovechar dicha información utilizando las técnicas de *CSP*

En primer lugar hay que tener en cuenta que se puede establecer un sencillo *test de planificabilidad* (algoritmo que permite saber si puede existir una planificación del tipo deseado para un conjunto de tareas) si consideramos el sumatorio de tiempo disponible en la agenda y la suma de los tiempos consumidos por las *tareas* para una determinada *agenda*.

La representación de nuestro *CSP* podría ser la siguiente, nuestro conjunto de variables

- X es el conjunto de todos los intervalos disponibles para trabajar en una semana de una agenda ordenados con el uso horario habitual
- D el dominio de dichas variables es 0 o 1 (*CSP binario*)
- C el conjunto de restricciones es bastante complejo, pero deviene finalmente del resultado del motor de planificación

En primer lugar nuestra variables son $X = \{x_1, x_2, \ldots, x_n\}$ siendo n el número de intervalos disponibles a lo largo de la semana, tendrán un valor pertenecientes al dominio

$D = \{0, 1\}$ cuyo significado es si dicho intervalo será tenido en cuenta para la planificación sí se usará ($= 1$) o no se usará ($= 0$).

El orden en el cual las variables son asignadas durante la búsqueda puede tener un impacto significativo en el tamaño del espacio de búsqueda, en nuestro caso utilizaremos la técnica *CSP* de *generar y testear* de la siguiente manera:

1. En el estado inicial del problema que representaremos como el conjunto de intervalos $S = \{x_1, \ldots, x_n\}$ con aridad igual a los intervalos disponibles para trabajar ordenados de lunes a viernes de la mañana a la tarde, donde se ubicará el valor asignado a cada variable que será el valor del dominio 1 únicamente en aquellas posiciones en las que el intervalo tenga la mayor preferencia por parte del usuario, es decir el que más le guste para trabajar.

   Ejemplo Si un horario tuviera cuatro intervalos a la semana disponibles para el trabajo del alumno y sólo el primero tuviera la máxima calificación por parte del usuario el estado inicial se representaría como

   $$S_0 = \{1, 0, 0, 0\}$$

2. La fase de *generación* consiste en ir construyendo las distintas tuplas asignándole **en orden de preferencia** del usuario el valor del dominio 1 a aquellos intervalos disponibles que coincidan con el grado de preferencia evaluado. *i.e*: primero se irán poniendo a 1 los intervalos horarios de *cuatro estrellas* luego los de *cuatro estrellas y los de tres* y así hasta que en el último caso tengamos activados *todos* los intervalos.

3. La fase de *testeo* consiste en la llamada al motor de planificación con el conjunto de intervalos dado, con un *test de planificabilidad* se puede descartar iniciar la planificación simplemente con el sumatorio del tiempo disponible y el tiempo que se necesita para las tareas sin tener en cuenta las restricciones; si es factible se inicia la planificación con un horario que realmente es un subconjunto del suministrado por el usuario, pero el subconjunto más favorable posible, conforme vayan fallando las planificaciones debido a que no cumplan con restricciones "*más finas*"que la del *test de planificabilidad*, se irá generando otra tupla que insertará los intervalos del siguiente nivel (menos agradables para el usuario) hasta que finalmente se de con la planificación que haga el menor uso posible de los intervalos más desagradables para el usuario y cumpla los *deadlines* o devuelva el mensaje de error informando de en qué fecha y con qué tarea ha fallado la planificación solicitando una relajación de las condiciones iniciales del problema, bien disminuyendo la duración, el número

de tareas, el tiempo máximo permitido para que se trabaje de forma continuada en ellas o aumentando el tiempo disponible de estudio

De este modo, se aumenta el tiempo necesario para la planificación pero se hace en una serie de pasos discretos, como mucho tantos como intervalos disponibles de trabajo tenga un alumno a lo largo de la semana, que siempre será un número discreto y pequeño, a mayor cantidad de intervalos menor duración de estos, luego antes se descartará la planificación en caso de no fuese admisible y se pasará a evaluar otra con mayor número de intervalos.

Con este enfoque, dada la velocidad de ejecución del planificador base no hay un aumento sensible del tiempo de ejecución de las planificaciones en los casos evaluados y hay gran diferencia en el resultado de la planificación, mucho más cercana a las preferencias del usuario.

Hay que objetar que el planificador no es óptimo (un planificador es óptimo si, existiendo una planificación del tipo deseado, encuentra dicha planificación siempre) ya que tanto la expulsión de tareas por exceder el tiempo máximo de dedicación elegido por el usuario, como el orden en el que se insertan las tareas, que puede producir (como efecto lateral del *orden fijado* de asignación de tareas en los intervalos disponibles) una suerte de fragmentación interna que pudiera ser menor con alguna permutación del orden de asignación de tareas para cada intervalo (ya que éstas también tienen una duración mínima, no tendrían sentido asignaciones de 10' al final de un día) rompen esa propiedad.

Haber tenido en cuenta esas características lastraría los tiempos de cómputo debido al hecho de que habríamos de tener en cuenta las permutaciones a la entrada de los intervalos disponibles de las tareas, elevando la complejidad algorítmica y obteniendo un beneficio dudoso ya que sólo con planificaciones muy ajustadas se podría obtener algún beneficio.

3.5. Librería de planificación *Optaplanner*

Tomando como punto de partida los problemas planteados en el apartado anterior cabe pensar si convendría la utilización de *Frameworks* disponibles de acceso público y gratuito como *Optaplanner* (desarrollada bajo el auspicio de la compañia *RedHat*).

OptaPlanner (Figura 8) es un *Constraint Satisfaction Solver*, es decir posee un motor de planificación que permite optimizar la planificación de recursos. Resuelve los problemas clásicos de los *CSP* asignar un conjunto limitado de recursos limitados (empleados, activos, tiempo y dinero) para proporcionar productos o servicios a los clientes, rutas para vehículos, turnos de los empleados de turnos, planificación de trabajo, etc...

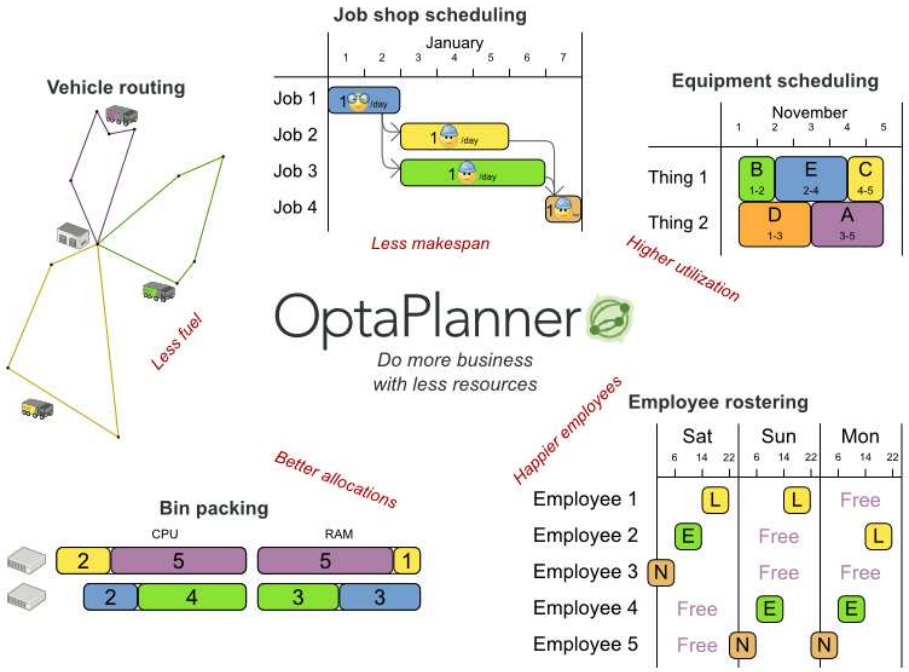


Figura 8: Funcionalidad general Optaplanner [6]

Es un motor de planificación ligero, escrito íntegramente en *Java*, y que, en principio, permite a los programadores resolver problemas de optimización de un modo eficaz. OptaPlanner según reza en su propaganda oculta al programador sofisticados algoritmos de optimización y heurística. Es software Open Source, y está publicado bajo la licencia *Apache Software*.

Ante las ventajas que presenta el uso de un *API*, no reinventar la rueda, el hecho de ser un código mucho más estable y testado y que en principio tiene el soporte de un equipo

de desarrollo que lo mejora con el tiempo, merece la pena intentar integrar esta *API* en nuestra aplicación.

Para ello se hizo uso de este *Framework* en una nueva rama *Git* para tratar de establecer una comparativa con nuestra aplicación y también ver cuan apropiada resultaba para integrarlo en una plataforma como *Android*.

El proceso fue el siguiente, la resolución de un problema con Optaplanner consta de cinco pasos

1. Modelar el problema, en este caso como una clase que implemente la interfaz *Solution*
2. Configurar un *Solver* que es el encargado de modelar una estrategia de resolución del problema
3. Inicializar los parámetros del problema
4. Aplicar el método *Solver.solve(problema)*
5. Obtener la mejor solucion del método anterior con *Solver.getBestSolution();*

3.5.1. Modelar el problema

Para representar el problema crearemos una clase *OptaplannerSolution* esto es, que contenga la lista de intervalos de plantilla semanal con sus preferencias así como la agenda y la lista de tareas

```
package es.uma.casandra.optaplanner;
import org.optaplanner.core.api.domain.solution.PlanningSolution;
import org.optaplanner.core.api.domain.solution.Solution;
import org ... .core.api.score.buildin.hardsoftlong.HardSoftLongScore;
import java.util.Collection;

@PlanningSolution
public class OptaplannerSolution implements Solution<HardSoftScore> {
    ...
}
```

Optaplanner provee múltiples implementaciones de *Score* pues es con puntuaciones con lo que estima la calidad de las soluciones ofrecidas, de entre todas ellas elegimos la HardSoftScore para de este modo poder penalizar con *hard scores* el incumplimiento de

las restricciones duras y descartar esa solución y con *soft scores* la utilización de los intervalos peor valorados por el usuario, pero permitir que el resultado siga siendo válido.

```
public class CasandraScoreCalculator implements
    EasyScoreCalculator<OptaplannerSolution> {
    @Override
    public HardSoftScore calculateScore(OptaplannerSolution sol) {
    ...
        //Calculamos las penalizaciones por violar restricciones duras:
        for(Tarea t:listaTmpTareas){
            //La duracion que no ha entrado en la planificacion
            if(t.getDuracion()>0){
                hardScore-= t.getDuracion();
            }
        }
        //Calculamos las penalizaciones por violar restricciones blandas;
        for(IntervaloPlantillaSemanal i: listaIntervalos){
            if(i.getSeleccionado()){
                softScore+=i.getValoracion()-5;
            }
        }
        return HardSoftScore.valueOf(hardScore,softScore);
    }
}
```

3.5.2. Configurar un Solver

Configurar un *Solver* por permite definir la estrategia o algoritmo de resolución. El *API* de *Optaplaner* nos permite configurarlo mediante un fichero de configuración *xml* que será indicado a la clase factoría *SolverFactory*

```
public static final String SOLVER_CONFIG =
    "es/uma/casandra/optaplanner/CasandraSolverConfig.xml"; ...
```

Y éste sería parte del fichero de configuración

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
  <solutionClass>es.uma.casandra.optaplanner.OptaplannerSolution</solutionClass>
  <entityClass>es.uma.casandra.core.agenda.IntervaloPlantillaSemanal</entityClass>
```

```
 <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
    <easyScoreCalculatorClass>es.uma.casandra.optaplanner.CasandraScoreCalculator
    </easyScoreCalculatorClass>
  </scoreDirectorFactory>
 ...
</solver>
```

La clase *Solver* puede utilizar varios algoritmos de optimización encadenados. A cada uno de los algoritmos que aparecen en el xml los denomina *phase*, para la prueba vamos a usar dos tipos de búsqueda

De las implementaciones que nos ofrece optaplanner vamos a utilizar un algoritmo de busqueda exhaustiva, y otro de búsqueda local

```
<!--Fases de busqueda Exhaustiva-->
<exhaustiveSearch>
    <exhaustiveSearchType>BRUTE_FORCE</exhaustiveSearchType>
</exhaustiveSearch>

....

<!--Fases de busqueda Local-->
<localSearch>
  <localSearchType>HILL_CLIMBING</localSearchType>
  <termination>
    <millisecondsSpentLimit>100</millisecondsSpentLimit>
  </termination>
</localSearch>
```

3.5.3. Inicializar el problema

Vamos a usar un *horario* con un número creciente de *intervalos* a lo largo de una semana. Los atributos de predilección de un intervalo horario en concreto será seleccionado pseudoaleatoriamente mediante Math.random() de la librería de *Java*.

Este horario será aplicado sobre la agenda de una semana en la que se planificarán tareas por valor de la mitad del tiempo disponible.

```
for(int numIntervalos = 7; numIntervalos< 70; numIntervalos +=1){
```

```
    CSPHoraio datos = generarDatos(numIntervalos, new
        GeneradorPreferencias() {
        @Override
        public int generarPreferenciaUsuario() {
            return (int) Math.round(Math.random() * 5);
        }
        });
        Agenda agenda = datos.agenda;
        List<IntervaloPlantillaSemanal> horario = datos.horario;
        List<TareaEsporadica> tareas = datos.tareas;

    resolverOptaplanner(agenda, tareas, horario,
        SOLVER_CONFIG_BUSQUEDA_EXHAUSTIVA);
    resolverOptaplanner(agenda, tareas, horario,
        SOLVER_CONFIG_BUSQUEDA_LOCAL);
    resolverCasandra(agenda, tareas, horario);


}

public static void resolverOptaplanner(Agenda
    agenda,List<TareaEsporadica> tareas, List<IntervaloPlantillaSemanal>
    horario, String solverXml){
    OptaplannerSolution csp = new OptaplannerSolution();
    csp.setAgenda(agenda);
    csp.setListaTareas(tareas);
    csp.setListaIntervalos(horario);
    Solver solver =
        SolverFactory.createFromXmlResource(solverXml).buildSolver();
    long ini = System.nanoTime();
    solver.solve(csp);
    long fin = System.nanoTime();
    OptaplannerSolution sol = (OptaplannerSolution)
        solver.getBestSolution();
    loguear(solverXml,horario,ini,fin,sol.getScore());
}
```

4. Diseño

4.1. Patrones de diseño

Los patrones de diseño buscan:

- ◇ Proporcionar catálogos de elementos reutilizables en el diseño de sistemas software.
- ◇ Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- ◇ Formalizar un vocabulario común entre diseñadores.
- ◇ Estandarizar el modo en que se realiza el diseño.
- ◇ Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

En Casandra hemos optado por hacer uso de patrones de diseño en varios apartados de la aplicación que exponemos a continuación [7], [8].

4.1.1. El patrón *Data Access Object*

Un *Data Access Object* (*DAO*, Objeto de Acceso a Datos) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una base de datos o un archivo.

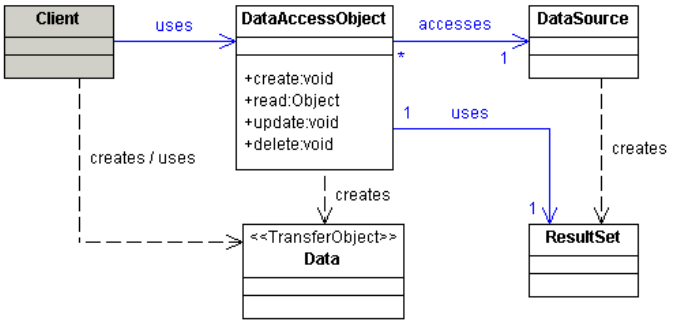


Figura 9: Representación del patrón *Data Access Object*

fuente: "http://www.corej2eepatterns.com/Patterns2ndEd/DataAccessObject.htm"

No todo son ventajas a la hora de utilizar este patrón, mostramos los pros y contras a continuación

- ◇ *ventajas* : La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Los Objetos de Acceso a Datos *DAO* pueden usarse en *Java* para aislar a una aplicación de la tecnología de persistencia Java subyacente (*API* de persistencia *Java*).

Utilizar Objetos de Acceso de Datos redunda en que la tecnología subyacente puede ser actualizada o cambiada sin necesidad de cambiar otras partes de la aplicación.

◇ *desventajas* : La flexibilidad tiene un coste. Cuando se añaden *DAO's* a una aplicación, la complejidad adicional de usar otra capa de persistencia incrementa la cantidad de código ejecutado durante tiempo de ejecución. La configuración de las capas de persistencia requiere en la mayoría de los casos mucho trabajo.

Las aplicaciones críticas con el rendimiento no deberían usar este patrón.

4.1.2. Utilización del patrón *Data Access Object* en nuestra aplicación

El uso de este patrón nos ha ofrecido varios beneficios para la persistencia de datos :

◇ Ha separado el acceso a los datos de la lógica de la aplicación

◇ Oculta la *API* con la que se accede a los datos

◇ Centraliza todos los accesos a los datos en un capa independiente

Cuando trabajamos con *DAO*, trabajamos en un mundo desconectado, donde nuestros datos deben persistir en objetos.

Por lo tanto, cuando se realiza una operación, abrimos la conexión a la base de datos, se ejecuta el comando, y si es una operación de lectura, se vuelca el contenido hacia una estructura de datos y se cierra la conexión.

Los *DTO (Data Transfer Object)* o también denominados *VO (Value Object)* son utilizados por *DAO* para transportar los datos desde la base de datos hacia la capa del modelo de la aplicación y viceversa.

En una aplicación, hay tantos *DAO's* como modelos. Es decir, en nuestra base de datos relacional, por cada tabla, tenemos un *DAO*.

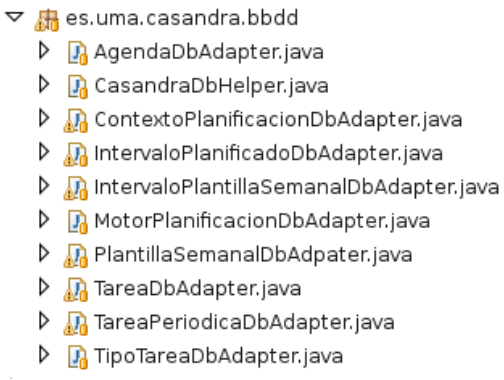


Figura 10: Patrón *DAO* - tablas

Sirva como muestra del uso del patrón, un fragmento simplificado del código para una de las múltiples tablas de la base de datos de nuestra aplicación

```
package es.uma.casandra.bbdd;

import ...

public class TareaDbAdapter {

    private SQLiteDatabase mDb;

    // Nombre de los campos de la tabla
    public static final String ID = BaseColumns._ID;
    public static final String NOMBRE = "NOMBRE";
    public static final String CARACTER = "CARACTER";
    public static final String FINICIO = "FINICIO";
    public static final String FFIN = "FFIN";
    public static final String DURACION = "DURACION";
    public static final String MAXMINSEGUIDOS = "MAXMINSEGUIDOS";
    public static final String FRECUENCIA = "FRECUENCIA";

    // Consulta creacion de la tabla tarea
    private static final String DATABASE_CREATE_TABLE_TAREA =
    ...
    public TareaDbAdapter(SQLiteDatabase casandra){
    ...
    public void crearTabla() {
        mDb.execSQL(DATABASE_CREATE_TABLE_TAREA);
    }
```

```
    public long insertarTarea(Tarea nueva){
    ...
    public List<String> getAllNombresTareas(){
    ...
    public Cursor getCursorTareas(){
    ...
    public List<Tarea> getAllTareas(){
    ...
    public Tarea getTarea(Long idTarea){
    ...
    private Tarea leerTareaDeCursor(Cursor cursor){
    ...
}
```

4.1.3. El patrón *Policy/strategy*

La definición formal sería la siguiente:

> "El patrón *Strategy* define una familia de algoritmos, encapsula cada uno, y los hace intercambiables. Este patrón permite que el algoritmo varíe independientemente del cliente que lo use."[7]

Este patrón es adecuado cuando

- ◇ Muchas clases relacionadas difieren solo en su comportamiento. Éste patrón provee un modo de configurar una clase con un comportamiento seleccionado de entre múltiples candidatos.
- ◇ Cuando se necesitan diferentes variantes de un algoritmo. Esas variaciones podrían ser implementadas como una jerarquía de clases de algoritmos.
- ◇ Cuando un algoritmo utiliza datos que los clientes no deberían conocer, se puede utilizar el patrón *Strategy* para evitar la exposición de complejas estructuras de datos específicas de determinados algoritmos.

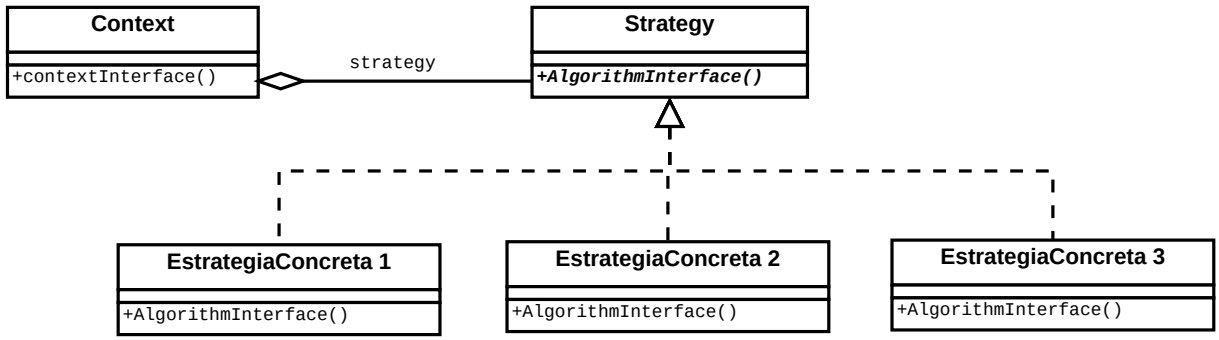


Figura 11: Patrón *Strategy*

Strategy Declara una interfaz común a todos los algoritmos soportados. *Context* usará esta interfaz para llamar al algoritmo definido por una estrategia concreta (*ConcreteStrategy*).

EstrategiaConcreta En nuestro ejemplo (*EstrategiaConcreta1*, *EstrategiaConcreta2*, *EstrategiaConcreta3*) . Implementan el algoritmo utilizando la interfaz de *Strategy*

Context El objeto *Context* posee varias características

- ◇ Está configurado con un objeto *EstrategiaConcreta*
- ◇ Mantiene una referencia a dicho objeto
- ◇ Podría definir una interfaz que permitiese a *Strategy* acceder a sus datos

4.1.4. Utilización de *Strategy* en nuestra aplicación

En nuestra aplicación hemos considerado apropiado utilizarlo en la elección del motor de planificación.

De entre los distintos algoritmos de *Scheduling* disponibles se han elegido tres diferentes *RMS, DMS* y *EDF*, al igual que se ha desarrollado una prueba de utilización de una *API* de terceros como *Optaplanner* (en otra rama de desarrollo de *Git*) para resolver problemas de planificación que podrían integrarse a nuestra aplicación utilizando este patrón, caso que se comentará más adelante.

Dado a que los únicos parámetros necesarios para realizar la planificación son una *Agenda* y una lista de *tareas* (como se puede ver en el fragmento de código anexo) cualquier enfoque que sea capaz de con esa información devolver una lista de *intervalos* planificados, sería un posible candidato a una implementación de la interfaz de planificador de tareas, lo que hace mucho más versátil la capacidad planificadora de la aplicación.

```
import java.util.List;
```

```
import es.uma.casandra.core.agenda.Agenda;
import es.uma.casandra.core.agenda.Intervalo;
import es.uma.casandra.core.tarea.Tarea;


public interface IPlanificador {

    /**
     * Dada la lista de intervalos de tiempo disponibles que es devuelta
        por la clase Agenda y una lista de tareas,
     * se generara una lista de intervalos planificados
     *
     * @param agenda
     * @param tareas
     * @return La lista de intervalos planificados
     */


    List<Intervalo> planificar (Agenda agenda, List<Tarea> tareas);

}
```

Con nuestra implementación, la clase *Strategy* (del patrón de diseño anteriormente expuesto) quedaría así

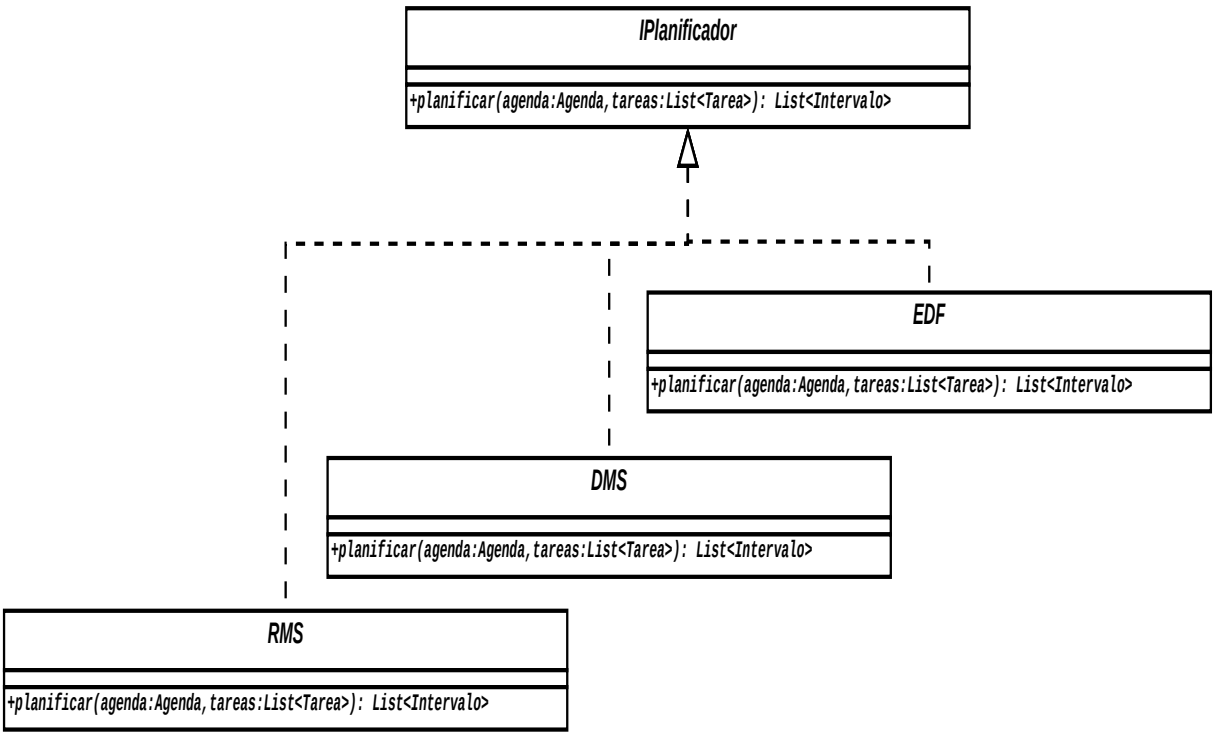


Figura 12: Aplicación del Patrón *Strategy* al motor de planificación

De este modo es sencillo tanto ampliar el número de motores de planificación en un futuro, como su uso a la hora de trabajar con ellos.

4.2. Diseño de objetos

Hemos procurado que las clases sean fieles representantes de entidades, y a su vez hemos agrupado éstas en *paquetes* con una funcionalidad común que facilite la comprensión y el análisis del código.

Seguidamente describimos someramente los conjuntos de clases (*paquetes*) empleados en el desarrollo y sus objetivos

Todo el código de la aplicación está dividido en tres paquetes principales:

- *es.uma.casandra.android*: que contiene los *Fragments* y *Activities* que se le presentarán a los usuarios
- *es.uma.casandra.bbdd*: que contiene las implementaciones del patrón *DAO* del que se hablará un poco más adelante, así como la clase de utilidades para nuestro uso de la base de datos *SQLite*, que hemos denominado *CasandraDbHelper*.
- *es.uma.casandra.core*: donde se encuentran las clases con que modelamos los elementos básicos de nuestra aplicación.

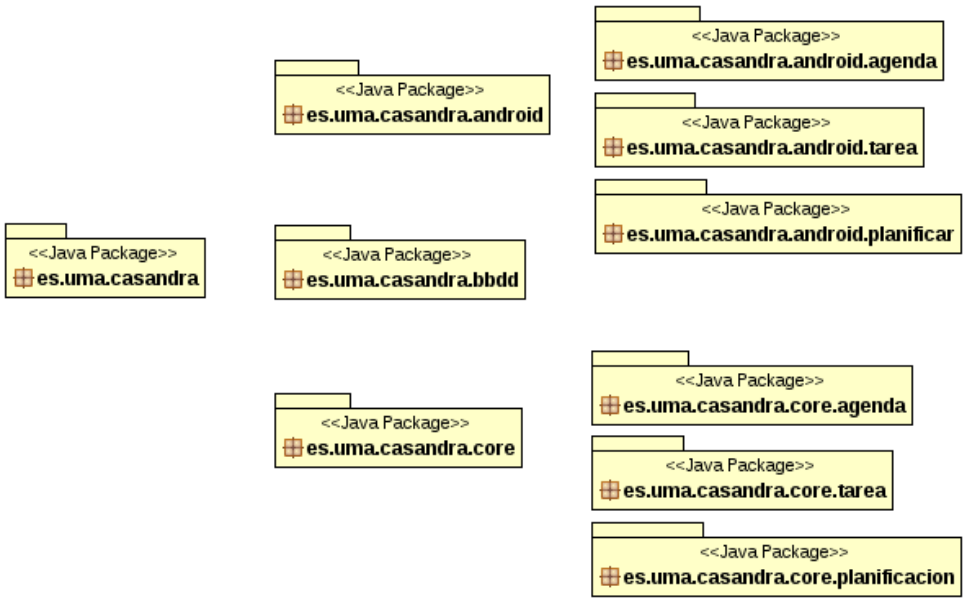


Figura 13: Organización de los paquetes *Java*

4.2.1. El paquete casandra.android

Este paquete contiene las *Activities* y los *Fragments* con los que interactúa el usuario, comenzando por la actividad especificada en el fichero *Android.manifest* como punto de entrada principal a la aplicación.

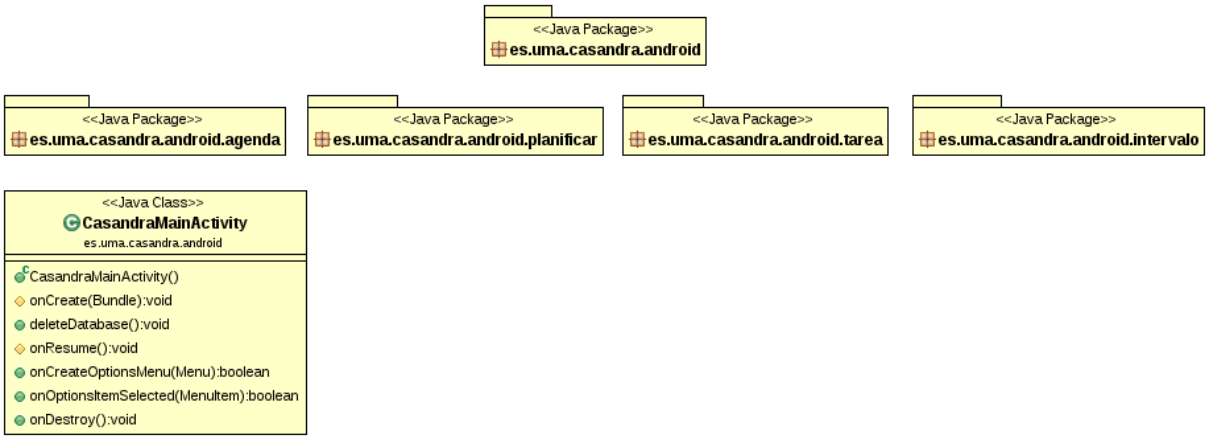


Figura 14: El paquete *casandra.android*

El resto de componentes se encuentran dividido en paquetes en función de los objetos para los que intentan recabar o mostrar información

- Para las tareas del usuario (*es.uma.casandra.tarea*)
- La agenda (*es.uma.casandra.agenda*)
- Y el estado actual de la planificación con *es.uma.casandra.planificar, y es.uma.casandra.replanifica*

4.2.2. El paquete es.uma.casandra.core

Como ya dijimos, es donde se encuentran las clases con que se modelan los elementos básicos de nuestra aplicación, se compone de tres subpaquetes, tarea, planificacion y agenda.

En el paquete *es.uma.casandra.core.tarea* contiene la clase *Tarea* y dos especificaciones de la misma que utilizamos en nuestros algoritmos de planificación.

El paquete *es.uma.casandra.core.planificacion* contiene las implementaciones de los algoritmos, así como una clase *ContextoPlanificacion*, que intenta encapsular toda la información necesaria para recrear una planificación y además implementa los métodos necesarios para realizar un mecanismo de confirmación del seguimiento de una planificación existente.

El paquete *es.uma.casandra.core.agenda* contiene las clases necesarias para presentar la agenda de un estudiante de un modo adecuado a nuestros algoritmos de planificación, así como varios métodos de utilidades para tratar el tiempo.

4.2.3. El paquete es.uma.casandra.bbdd

Este paquete contiene por un lado la clase *CasandraDbHelper* que es una extensión de la clase *SQLiteOpenHelper* suministrada por *Android* y es el punto principal de comunicación del resto de la aplicación con *SQLite*, pues es la responsable de obtener un objeto *SqliteDatabase*.

Por otro lado tenemos una clase *Adapter* para cada tabla del modelo de datos, que contiene por una parte las sentencias *DDL* de definición de su tabla asociada para que sean utilizadas en el momento de la creación de la base de datos por *CasandraDbHelper*, y por otra parte contiene una interfaz para permitir el acceso a los datos utilizando el patrón *Data Access Object* tal y como viene comentado en el apartado de *Patrones de diseño* de la memoria a tal fin.

4.3. Metodología de trabajo

Para el desarrollo de nuestro proyecto, se ha optado por seguir las pautas de un modelo de desarrollo clásico de software.

Seguir una metodología de desarrollo, ayuda a organizar las actividades y conducir al proyecto de manera paulatina y lógica, hacia su finalización.

Hemos usado un paradigma de ingeniería del software basado en el *desarrollo en cascada*, estructurando el desarrollo de la aplicación en un conjunto de fases de trabajo:

- ◇ *fase de análisis:* Se analizan y detectan las necesidades del sistema.
- ◇ *fase de diseño:* en la que nos planteamos cómo vamos a resolver el problema.
- ◇ *fase de codificación:* mediante la cual construimos la solución.
- ◇ *fase de pruebas:* donde comprobamos que el producto es el deseado y que está bien construido.
- ◇ *fase de corrección:* donde se corrigen los errores.

Vamos a pasar a comentar cada una de las fases de forma más detallada a continuación

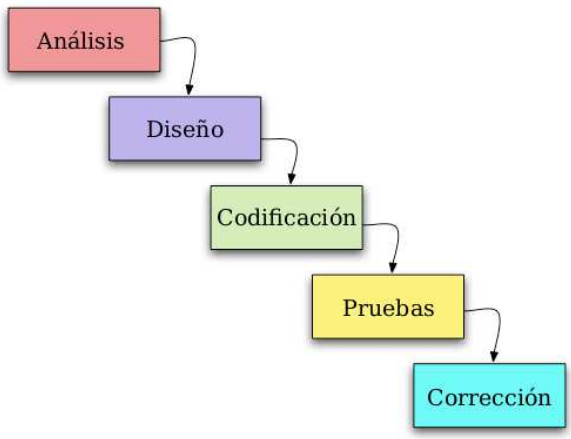


Figura 15: Modelo en cascada

4.3.1. Fase de análisis

En esta fase se analizaron las necesidades a cubrir para los usuarios finales de la aplicación, para determinar qué objetivos debe cubrir.

De esta fase se obtuvo una identificación completa de los requisitos a cubrir sin entrar en detalles de implementación de las mismas.

4.3.2. Fase de diseño

Se obtiene la descripción de la estructura global del sistema y la especificación de lo que debe hacer cada una de sus partes, así como la manera en que se combinan unas con otras.

Es la fase en donde se realizan los algoritmos necesarios para el cumplimiento de los requerimientos del usuario así como también los análisis necesarios para saber qué herramientas usar en la etapa de Codificación.

4.3.3. Fase de codificación

Es la fase en donde se implementa el código fuente, desarrollando e integrando las distintas actividades que formarán parte de la aplicación, así como la elaboración del contenido gráfico, la base de datos, etc ... haciendo en el proceso multitud de pruebas y ensayos para corregir errores.

4.3.4. Fase de pruebas

En la fase de validación, se comprobará que hemos construido lo que realmente se pretendía en la fase de análisis y que cumple su función.

4.3.5. Fase de corrección

En ésta penúltima fase, se corregirán todos los errores detectados en la fase de pruebas, y se añadirán mejoras en requisitos no funcionales en la medida de lo posible.

5. Herramientas utilizadas

5.1. Latex

Para la redacción de la memoria se ha optado por la herramienta LaTeX ,los motivos que han llevado a esta elección son los siguientes

- ◇ Es estable y multiplataforma.
- ◇ *Latex* permite redactar fácilmente documentos estructurados, lo que da uniformidad y ayuda a una correcta ilación de contenidos
- ◇ Controla en todo momento la numeración y las referencias cruzadas.
- ◇ Construye índices de contenidos, tablas o figuras.
- ◇ Ajusta los tamaños y tipos de letras según la parte del documento en que se hallen.

En cuanto a los inconvenientes, debido al uso en el anterior proyecto, buena parte de los inconvenientes del uso de LaTeX se ven atenuados, la curva de aprendizaje es menor al contar con un esquema válido de uso desde el primer momento de la redacción y se está familiarizado con los errores de compilación y el entorno de "desarrollo", que a su vez automatiza el proceso de compilación del texto.

5.2. Control versiones del código fuente

El código de partida lo desarrollamos bajo el paraguas de un control de versiones distribuido, en concreto *Git*. Los motivos de uso de un *SVC* (*System Version Control*) es la flexibilidad a la hora del desarrollo ya que se gestionan fácilmente los diversos cambios que se realizan sobre los elementos del software desarrollado y la configuración del mismo, almacenando una versión, revisión o edición de un producto, en el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación.

A lo largo de esta ampliación y mejora del software original se ha hecho un uso intensivo de esta herramienta, testando en distintas ramas de desarrollo diseños nuevos de interfaz, soluciones alternativas en partes críticas del programa y la integración de diferentes *Apis*.

Para la realización del proyecto se ha elegido *Git*, tanto para el código fuente de la aplicación *Android*, como para la elaboración de la memoria.

Si bien cualquiera de los diferentes sistemas de control de código fuente habría satisfecho nuestras necesidades para este proyecto, hemos elegido Git debido a que:

- ◇ Es código abierto y gratuito.
- ◇ Es muy popular, y hay multitud de documentación de calidad disponible
- ◇ Trabajar con ramas (*branchs*) es sencillo:

  Si se te ocurre una nueva característica, creas una nueva rama y comienzas a trabajar inmediatamente, saltar entre distintas ramas o fusionar ramas de nuevo con la principal no entraña mucha dificultad.

- ◇ Te permite almacenar (*Stash*) los cambios en tu rama actual, hacer trabajo en otra rama, comprobar los cambios y volver a la rama donde hicistes el *stash*.
- ◇ Git tiene integridad

  Todo en Git es verificado mediante una suma de comprobación (*checksum* mediante un hash *SHA-1*) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

- ◇ Git generalmente sólo añade información:

  Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier SVC, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

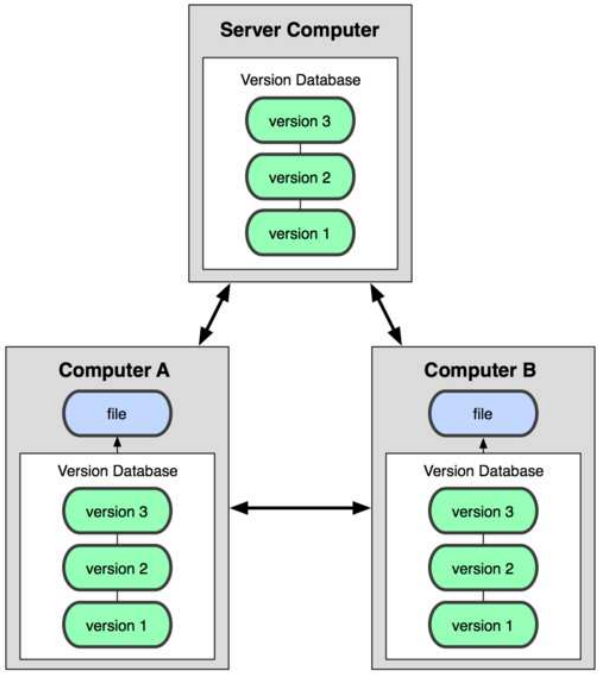


Figura 16: Sistema control versiones distribuido

fuente: Imagen tomada del libro "Pro Git, el libro oficial de Git"

El diseño de Git se basó en BitKeeper y en Monotone, y resulta de la experiencia del diseñador de Linux, Linus Torvalds, manteniendo una enorme cantidad de código distribuida y gestionada por mucha gente, que incide en numerosos detalles de rendimiento, y de la necesidad de rapidez en una primera implementación.

Entre las características más relevantes se encuentran:

- ◇ *Fuerte apoyo al desarrollo no lineal:* por ende rapidez en la gestión de ramas y mezclado de diferentes versiones. Git incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal. Una presunción fundamental en Git es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

- ◇ *Gestión distribuida:* al igual que *Darcs*, *BitKeeper*, *Mercurial*, *SVK*, *Bazaar* y *Monotone*, Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.

- ◇ Los almacenes de información pueden publicarse por HTTP, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión TCP/IP simple o a través de cifrado SSH.

- ◇ *Gestión eficiente de proyectos grandes:* dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución. Todas

las versiones previas a un cambio determinado, implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio (denominado autenticación criptográfica de historial).

◇ *Los renombrados se trabajan basándose en similitudes entre ficheros:* aparte de nombres de ficheros, pero no se hacen marcas explícitas de cambios de nombre con base en supuestos nombres únicos de nodos de sistema de ficheros, lo que evita posibles, y posiblemente desastrosas, coincidencias de ficheros diferentes en un único nombre.[9]

En la elección original de un *CVS* se realizó una comparativa entre los distintos productos accesibles para desempeñar la tarea y realmente cualquiera de los analizados hubiera *a priori* satisfecho nuestras necesidades.

(a) Subversion (b) Bazaar (c) Darcs (d) Git

Figura 17: CVS's valorados

5.3. Git y Android Studio

Android Studio, es el nuevo *IDE* para desarrollo de aplicaciones en *Android*, es un entorno maduro que está en desarrollo por parte de *Google* en colaboración con los propietarios del entorno *IntelliJ* sobre el que se basa, y ha desplazado a *Eclipse* que era de facto la antigua plataforma de desarrollo.

Android Studio viene preparado para trabajar con múltiples *SVC*, entre ellos *Git*, así como asistentes para configurar una cuenta en algún repositorio como *www.github.com* o *www.bitbucket.org* (el que ha sido nuestra elección, debido a que éste último permitía los repositorios privados). Así, con los parámetros de nuestra cuenta en bitbucket.org, configuramos *Android Studio* para que pudiera conectarse con el servidor y mantener sincronizado el avance del proyecto.

De este modo dispondremos de toda la funcionalidad de Git, pero de forma integrada con nuestro entorno de desarrollo, y además teniendo un repositorio Git siempre sincronizado con todas las ventajas que conlleva como copia de respaldo del trabajo realizado sobre la que puedes *"navegar"* a lo largo de toda la evolución del mismo.

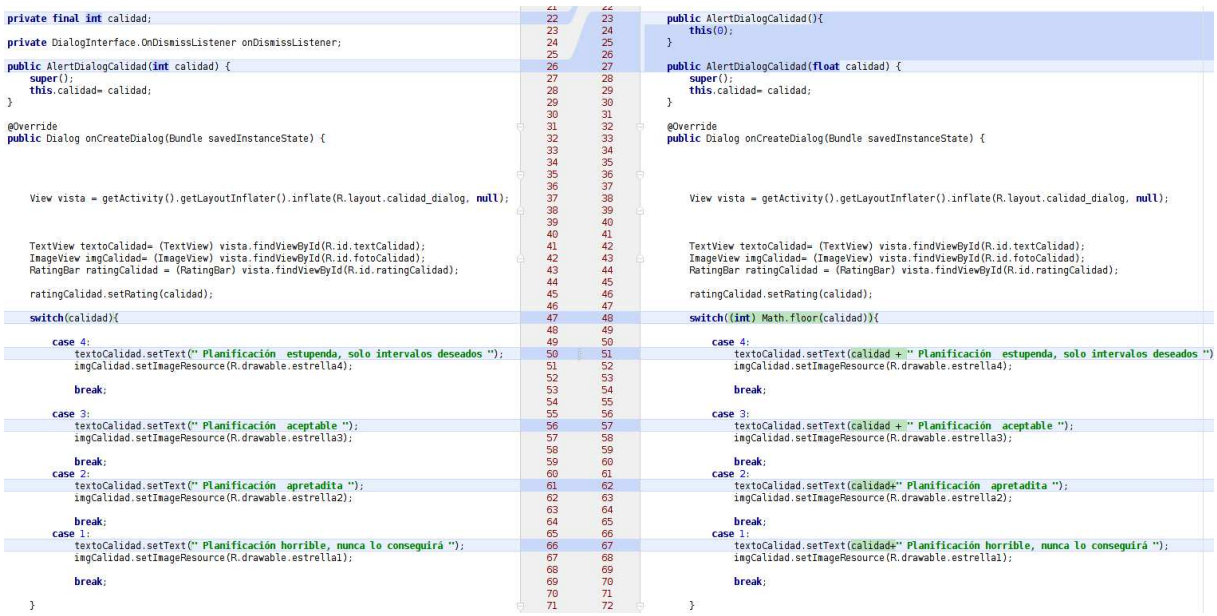


Figura 18: Ejemplo real de uso *Git* en entorno *Android Studio*

5.3.1. Gimp

Gimp ha sido una herramienta muy útil para la captura y tratamiento de los bocetos elaborados para el diseño de la aplicación, ya que fueron realizados expresamente para la aplicación y se tuvo que tratar las imágenes desde el boceto a lápiz hasta el gráfico final que forma parte del programa. *Gimp* (*GNU Image Manipulation Program*) es un programa de edición de imágenes digitales en forma de mapa de bits, tanto dibujos como fotografías. Es un programa libre y gratuito. Forma parte del proyecto *GNU* y está disponible bajo la Licencia pública general (*GPL*) de *GNU*.

Es un programa de manipulación de imágenes que ha ido evolucionando a lo largo del tiempo, ha ido soportando nuevos formatos, sus herramientas son más potentes, además funciona con extensiones o *plugins* y *scripts*

Gimp permite el tratado de imágenes en capas, para poder modificar cada objeto de la imagen en forma totalmente independiente a las demás capas en la imagen, también pueden subirse o bajarse de nivel las capas para facilitar el trabajo en la imagen, la imagen final puede guardarse en el formato *xcf* de *Gimp*, que soporta capas, o en un formato plano sin capas, como *png, bmp, gif, jpg, ...*

Posee también muchas herramientas y filtros para la manipulación de los colores y el aspecto de las imágenes, como enfoque y desenfoque, eliminación o adición de manchas, sombras, mapeado de colores así como un menú con un catálogo de efectos y tratamientos de las imágenes.

5.4. Inkscape

Es un editor de gráficos en formato vectoriales *SVG* (que son imagenes digitales formadas por objetos geométricos independientes como segmentos, polígonos, arcos, . . . , cada uno de ellos definido por distintos atributos matemáticos de forma, de posición, de color, . . .) , gratuito, libre y multiplataforma.

Las características de *SVG* soportadas incluyen formas básicas, trayectorias, texto, canal alfa, transformaciones, gradientes, edición de nodos, exportación de *svg* a *png*, agrupación de elementos, etc.

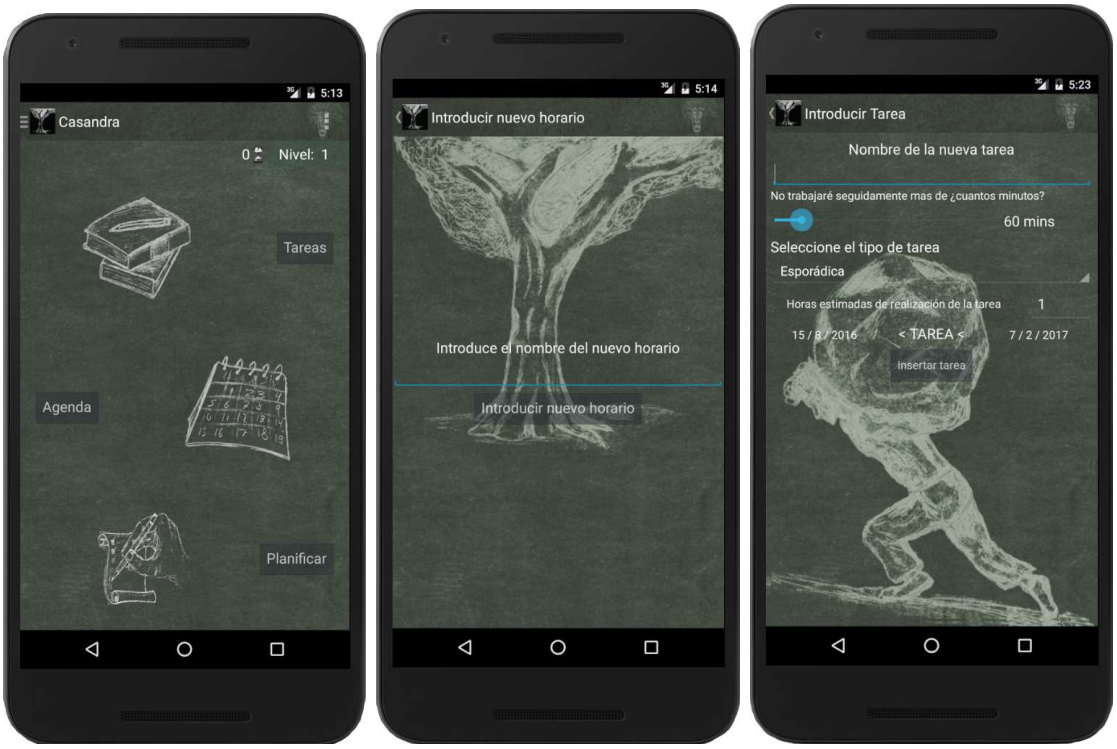
Tiene como objetivo proporcionar a los usuarios una herramienta libre de código abierto de elaboración de gráficos en formato vectorial escalable (SVG) que cumpla completamente con los estándares *XML, SVG* y *CSS2*.

En combinación con las herramientas anteriormente expuestas se puede obtener una correcta integración de las imágenes (elaboradas por nosotros para este proyecto) de una manera homogénea, desde un conjunto de esbozos a lápiz como estos

Figura 19: Imágenes elaboradas para el proyecto aún sin tratar

fuente: Fotografía dibujos originales para proyecto

Y quedan integrados así



(a) Principal (b) Nombre de horario (c) Inserción de tarea

Figura 20: Capturas de imágenes de la aplicación tras ser tratadas

5.5. SQlite

SQLite es un proyecto de dominio público creado por D. Richad Hipp que implementa una pequeña librería de aproximadamente 500Kb programada en lenguaje *C*, que funciona como un sistema de gestión de base de datos relacionales.
Android incorpora la librería *SQLite* que nos permite utilizar la base de datos mediante el lenguaje *SQL*, de una forma sencilla y utilizando muy pocos recursos del sistema y es la base de datos utilizada para almacenar toda la información que compone la aplicación, tanto a introducida por el usuario como la generada como el contexto de planificación, tareas , etc...

Para manipular la base de datos hemos usado la clase *SQLiteOpenHelper* que facilita tanto la creación, como la gestión de distintas versiones de dicha mase de datos.
La gran ventaja de utilizar esta clase es que ella se preocupará de abrir la base de datos si existe o de crearla si no existe. Incluso de actualizar la versión si decidimos crear una nueva estructura de la base de datos.

Algunas características de *SQLite* son

◇ Posee una única biblioteca es necesaria par acceder a bases de datos, y consume muy poca memoria

◇ En cuanto a rendimiento éste sistema de gestión de bases de datos relacional realiza operaciones de manera eficiente (más rápido en ocasiones que *MySQL* y *PostgreSQL*)

◇ *SQLite* se ejecuta en múltiples plataformas y sus bases de datos pueden ser fácilmente exportadas

◇ Es estable *SQLite* es compatible con *ACID*, en bases de datos se denomina *ACID* a un conjunto de características necesarias para que una serie de instrucciones puedan ser consideradas como una transacción, en concreto *ACID* es un acrónimo de *Atomicity, Consistency, Isolation and Durability* (Atomicidad, Consistencia, Aislamiento y Durabilidad) e implican:

- Atomicidad: es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias. Se dice que una operación es atómica cuando es imposible para otra parte de un sistema encontrar pasos intermedios. Si esta operación consiste en una serie de pasos, todos ellos ocurren o ninguno.
- Consistencia (o *Integridad*) : es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper las reglas y directrices de integridad de la base de datos. La propiedad de consistencia sostiene que cualquier transacción llevará a la base de datos desde un estado válido a otro también válido. "La Integridad de la Base de Datos nos permite asegurar que los datos son exactos y consistentes, es decir que estén siempre intactos, sean siempre los esperados y que de ninguna manera cambien ni se deformen. De esta manera podemos garantizar que la información que se presenta al usuario será siempre la misma."
- Aislamiento: es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error. Esta propiedad define cómo y cuándo los cambios producidos por una operación se hacen visibles para las demás operaciones concurrentes.
- Durabilidad (o *Persistencia*) : es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema y que de esta forma los datos sobrevivan de alguna manera.

◇ Implementa un gran subconjunto del estándar *ANSI SQL* del 92, incluyendo subconsultas, generación de usuarios, vistas y *triggers*

◇ Es de dominio público, y por tanto, se puede usar y redistribuir libremente

5.6. Android

Originalmente se eligió la plataforma *Android* por los siguientes motivos,

- Es una de las plataformas más populares en el desarrollo de aplicaciones para móviles, con una nutrida comunidad de desarrolladores, lo cual facilita el acceso a una adecuada documentación y solución de posibles incidencias.
- *Android* se desarrolla de forma abierta y se puede acceder tanto al código fuente como a la lista de incidencias donde se pueden ver problemas aún no resueltos y reportar problemas nuevos (*Google* liberó la mayoría del código de *Android* bajo la licencia *Apache*).
- Trae un *IDE* maduro, como *Android Studio* ampliamente usado por la comunidad de desarrolladores e integrado con la colección de herramientas de *ADT*.
- Android brinda una multitud de herramientas de desarrollo de manera gratuita a disposición de los usuarios

El ritmo de desarrollo para esta plataforma es abrumador, a pesar de su corta edad Desde el comienzo *Google* ha ido bautizando a las diferentes versiones de Android con el número de versión, y un alias alusivo a una comida dulce o postre cuyas primeras letras vienen dadas en orden alfabético.

Las distintas versiones de Android a lo largo de su historia son

- Android 1.0 *(Septiembre 2008)*
- Android 1.5 (Cupcake) *(Abril 2009)*
- Android 1.6 (Donut) *(Septiembre 2009)*
- Android 2.0 (Eclair) *(Octubre 2009)*
- Android 2.2 (Froyo) *(Mayo 2010)*
- Android 2.3 (Gingerbread) *(Diciembre 2010)*
- Android 3.0 (HoneyComb) *(Febrero 2011)*
- Android 4.0 (Ice Cream Sandwich) *(Octubre 2011)*
- Android 4.1 (Jelly Bean) *(Julio 2012)*
- Android 4.4 (KitKat) *(Octubre 2013)*

- ◇ Android 5.0 (Lollipop) *(Noviembre 2014)*
- ◇ Android 6.0 (MarshMallow) *(Octubre 2015)*
- ◇ Android 7.0 (Nougat) *(Agosto 2016)*

(a) v1.0

(b) v1.5

(c) v1.6

(d) v2.0

(e) v2.2

(f) v2.3

(g) v3.0

(h) v4.0

(i) v4.1

(j) v4.4

(k) v5.0 Lollipop

(l) v6.0 Marshmallow

(m) v7.0 Nougat

Figura 21: Versiones de *Android*

Lo que es un ritmo frenético en el que no han cesado de aparecer mejoras en el rendimiento, cambios sustanciales en el *API*, la máquina virtual sobre la que se ejecutan las aplicaciones, soporte *HW*, así como continuas modificaciones del entorno de desarrollo y políticas de diseño en cuanto al aspecto visual final de las aplicaciones.

En el proyecto anterior se hizo un comentario pormenorizado de las distintas mejoras en cada una de las versiones, pero entendemos queda un tanto fuera del objeto de este trabajo.

Cabe mencionar que Google tiene sus detractores, los hay de muchos tipos, desde los que establecen comparativas de estética y rendimiento con dispositivos de otros fabricantes, en las que en general no hay elementos críticos suficientes para posicionarse (entre otros motivos por el amplísimo parque de *Android* que abarca dispositivos de gama muy variada), hasta los que se adentran en cuestiones éticas como la *Free Software Foundation* que ha sido crítica con *Android* y han recomendado el uso de alternativas como *Replicant* (actualmente en la versión 4.2), debido a que los *drivers* y el *firmware* vital para

el adecuado funcionamiento de los dispositivos *Android* son normalmente patentados o propietarios, y a que *Google Play* invita al uso de software no libre sin notificarlo con claridad, esto puede ser una puerta abierta al uso torticero del dispositivo por parte de compañías de software y terceros, exponiendo al usuario a un uso indeseado de su dispositivo.

De igual modo se hizo un estudio comparativo con otros sistemas, que quedan ahora relegadas a título anecdótico ya que hubiera sido un enorme derroche de recursos portar todo el código. Se consideraron inicialmente como sistemas alternativos

(a) Replicant (b) Sailfish (c) Tizen (d) Mer (e) Ubuntu (f) FirefoxOS (g) Windows (h) iOS

Figura 22: Versiones de *Sistemas alternativos a Android*

Los principales componentes *Android* son

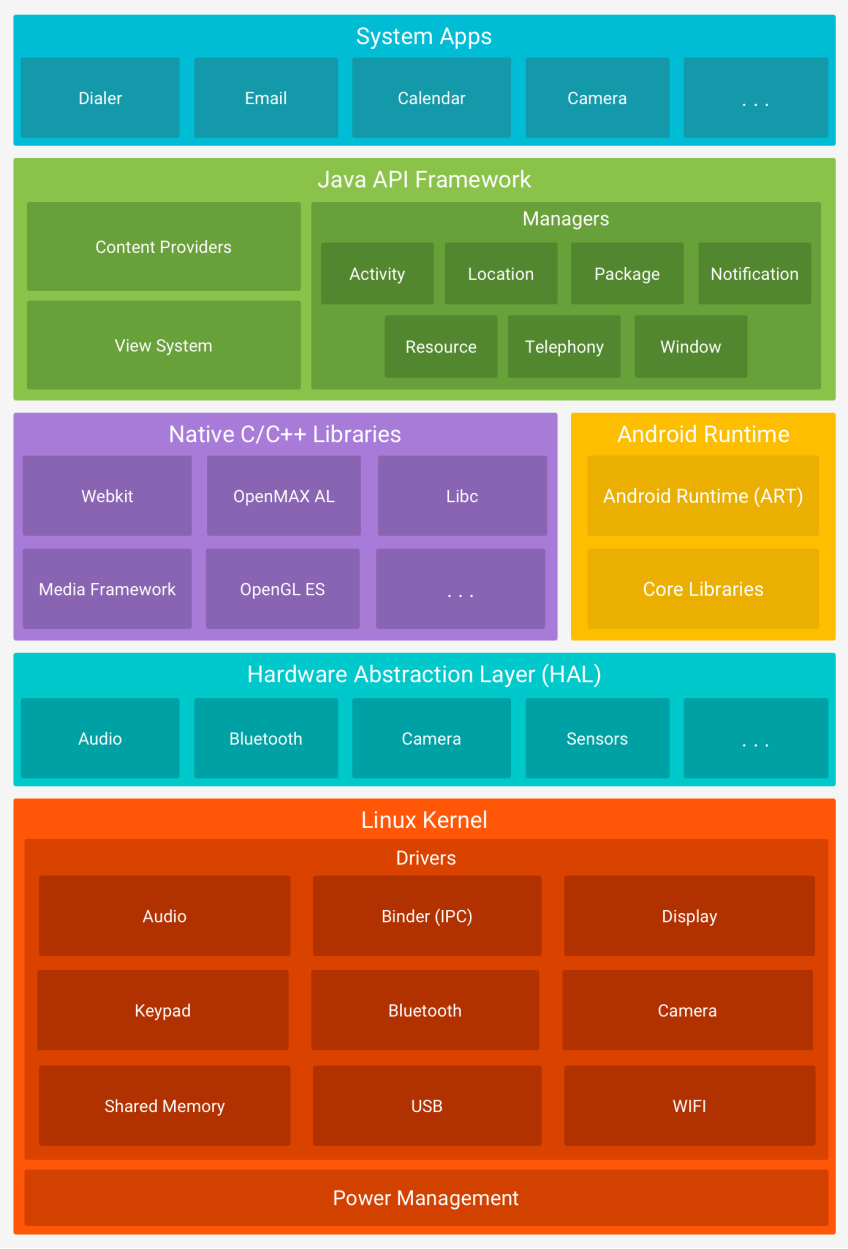


Figura 23: Versiones de *Arquitectura Android*

5.6.1. Núcleo *Linux Kernel*

Android depende de *Linux* para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, pila de red y modelo de controladores.

El núcleo del sistema operativo *Android* está basado en el kernel de *Linux* versión 2.6, similar al que puede incluir cualquier distribución de *Linux*, solo que adaptado a las características del hardware en el que se ejecutará *Android*, es decir, para dispositivos móviles.

El núcleo actúa como una capa de abstracción entre el hardware y el resto de las capas de la arquitectura.

El desarrollador no accede directamente a esta capa, sino que debe utilizar las librerías disponibles en capas superiores. De esta forma también nos evitamos el hecho de quebrarnos la cabeza para conocer las características precisas de cada teléfono. Si necesitamos hacer uso de la cámara, el sistema operativo se encarga de utilizar la que incluya el teléfono, sea cual sea.

Para cada elemento de hardware del teléfono existe un controlador (o driver) dentro del *kernel* que permite utilizarlo desde el software.

El *kernel* también se encarga de gestionar los diferentes recursos del teléfono (energía, memoria, etc.) y del sistema operativo en sí: procesos, elementos de comunicación (networking), etc.

5.6.2. Bibliotecas

Android incluye un conjunto de bibliotecas de *C/C++* usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del marco de trabajo de aplicaciones (*framework* de *Android*; algunas son:

- ◇ *System C library* (implementación biblioteca C estándar)
- ◇ *Webkit* (navegador)
- ◇ Bibliotecas multimedia (formatos de audio, imagen y vídeo)
- ◇ *OpenGl* (motor gráfico)
- ◇ *SSL* (cifrado de comunicaciones)
- ◇ *FreeType* (fuentes de texto)
- ◇ *SQLite* (Base de datos)

Se sitúa justo sobre el kernel la componen las bibliotecas nativas de *Android*. Están escritas en *C* o *C++* y compiladas para la arquitectura hardware específica del teléfono. Estas normalmente están hechas por el fabricante, quien también se encarga de instalarlas en el dispositivo antes de ponerlo a la venta. El objetivo de las librerías es proporcionar funcionalidad a las aplicaciones para tareas que se repiten con frecuencia, evitando tener que codificarlas cada vez y garantizando que se llevan a cabo de la forma "más eficiente".

5.6.3. *Runtime*

Desde la versión 5.0 de *Android 5.0* la máquina virtual *Dalvik* ha sido sustituida por *ART*, los beneficios en cuanto a rendimiento de esta nueva máquina virtual logran reducir el tiempo de ejecución hasta en un 33 %.

Dalvik era la antigua máquina virtual de Android, en la que cada aplicación corre su propio proceso, con su propia instancia de la máquina virtual *Dalvik*, *Dalvik* fue escrito de forma que un dispositivo pudiera arrancar múltiples máquinas virtuales de forma eficiente y ejecutar archivos en el formato *Dalvik Executable* (.dex), el cual está optimizado para un uso muy eficiente de la memoria.*Dalvik* era una variación de la máquina virtual de Java, por lo que *no es compatible con el bytecode Java*. *Java* se usa únicamente como lenguaje de programación, y los ejecutables que se generan con el *SDK* de *Android* tienen la extensión *.dex* que era específica para *Dalvik*, y por ello no podemos correr aplicaciones *Java* en *Android* ni viceversa

La principal diferencia entre *Dalvik* y *ART*, reside en que *Dalvik* ejecuta una maquina virtual interpretando el código al tiempo que se inicia la aplicación. En cambio, *ART* básicamente compila las aplicaciones antes de que sean ejecutadas.

Lo que significa que una primera instalación de una determinada aplicación llevará mucho más tiempo, y que las aplicaciones ocuparán más espacio del *internal storage*, pero al mismo tiempo, ya que las aplicaciones estarán completamente compiladas; tan pronto como estén instaladas en el sistema el tiempo que llevará abrir la aplicación será muy inferior al de hacerlo con la máquina virtual *Dalvik*, y además habría otra ventaja derivada de esto, como la parte de compilación ha sido llevada a cabo solo una vez (en la instalación de la aplicación) la carga del procesador es menor lo que redunda en una mejor vida de la batería y un mejor rendimiento global del sistema.

Una pequeña comparativa entre *Dalvik* y *Art* quizás arroje un poco de luz [10].

Dalvik	*Art*
Usa el enfoque *Just-In-Time (o JIT)* (es *grosso modo* una compilación de partes del código según se necesita y suelen formar parte de un intérprete). Lo cual redunda en un menor espacio de almacenamiento requerido, pero en un mayor tiempo de carga de las aplicaciones	Usa un enfoque (*Ahead-Of-Time (AOT)*), el cual compila la aplicación en el momento en que son instaladas, obteniendo como resultado unos mejores tiempos de carga y un menor uso del procesador (ya que sólo se compila una vez)
La *Caché* se va construyendo a lo largo del tiempo, luego los tiempos de arranque son más rápidos	La *Cache* es construida en el arranque, luego reiniciar el dispositivo requiere de un tiempo significativamente mayor
Trabaja mejor con dispositivos que dispongan de poca memoria interna, ya que el espacio ocupado es menor	Consume mucha más memoria interna, ya que almacena las aplicaciones compiladas además de los *APKs*

Cuadro 1: Comparativa máquinas virtuales *Dalvik* y *Art*

5.6.4. *Framework*

Los desarrolladores tienen acceso completo a los mismos *API's* del *framework* usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del *framework*). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.

La siguiente capa está formada por todas las clases y servicios que utilizan directamente las aplicaciones para realizar sus funciones. La mayoría de los componentes de esta capa son bibliotecas Java que acceden a los recursos de las capas anteriores a través de la máquina virtual *Dalvik*. Siguiendo el diagrama encontramos:

- *Activity Manager*: Se encarga de administrar la pila de actividades de nuestra aplicación así como su ciclo de vida.
- *Windows Manager*: Se encarga de organizar lo que se mostrará en pantalla. Básicamente crea las superficies en la pantalla que posteriormente pasarán a ser ocupadas por las actividades.
- *Content Provider*: Esta librería es muy interesante porque crea una capa que encapsula los datos que se compartirán entre aplicaciones para tener control sobre cómo se accede a la información.
- *Views*: En Android, las vistas son los elementos que nos ayudarán a construir las interfaces de usuario: botones, cuadros de texto, listas y hasta elementos más avanzados como un navegador web o un visor de Google Maps.

- ◇ *Notification Manager*: Engloba los servicios para notificar al usuario cuando algo requiera su atención mostrando alertas en la barra de estado. Un dato importante es que esta biblioteca también permite jugar con sonidos, activar el vibrador o utilizar los led's del teléfono (si los tuviese).

- ◇ *Package Manager*: Esta biblioteca permite obtener información sobre los paquetes instalados en el dispositivo Android, además de gestionar la instalación de nuevos paquetes. Con paquete nos referimos a la forma en que se distribuyen las aplicaciones *Android*, estos contienen el archivo .apk, que a su vez incluyen los archivos *1*.dex con todos los recursos y archivos adicionales que necesite la aplicación, para facilitar su descarga e instalación.

- ◇ *Telephony Manager*: Con esta librería podremos realizar llamadas o enviar y recibir SMS/MMS, aunque no permite reemplazar o eliminar la actividad que se muestra cuando una llamada está en curso.

- ◇ *Resource Manager*: Con esta librería podremos gestionar todos los elementos que forman parte de la aplicación y que están fuera del código, es decir, cadenas de texto traducidas a diferentes idiomas, imágenes, sonidos o layouts.

- ◇ *Location Manager*: Permite determinar la posición geográfica del dispositivo Android mediante GPS o redes disponibles y trabajar con mapas.

- ◇ *Sensor Manager*: Nos permite manipular los elementos de hardware del teléfono como el acelerómetro, giroscopio, sensor de luminosidad, sensor de campo magnético, brújula, sensor de presión, sensor de proximidad, sensor de temperatura, etc.

- ◇ *Cámara*: Con esta librería podemos hacer uso de la(s) cámara(s) del dispositivo para tomar fotografías o para grabar vídeo.

- ◇ *Multimedia*: Permiten reproducir y visualizar audio, vídeo e imágenes en el dispositivo.

5.6.5. Aplicacion

La capa superior de la de pila software la forman las aplicaciones. En esta capa conviven todas las aplicaciones del dispositivo, tanto las que tienen interfaz de usuario como las que no, tanto las nativas (programadas en C o C++) como las programadas en *Java* y tanto las que vienen de serie con el dispositivo como las instaladas por el usuario.

Aquí está también la aplicación principal del sistema: Inicio (Home), también llamada a veces lanzador (*launcher*), porque es la que permite ejecutar otras aplicaciones proporcionando la lista de aplicaciones instaladas y mostrando diferentes escritorios donde se

pueden colocar accesos directos a aplicaciones o incluso pequeñas aplicaciones incrustadas (o *widgets*), que son también aplicaciones de esta capa.

Lo principal a tener en cuenta de esta arquitectura es que todas las aplicaciones, ya sean las nativas de *Android* (proporcionadas por *Google*), las que incluye de serie el fabricante del teléfono o las que instala después el usuario utilizan el mismo marco de aplicación para acceder a los servicios que proporciona el sistema operativo .

Esto implica dos cosas:

- Podemos crear aplicaciones que usen los mismos recursos que usan las aplicaciones nativas (nada está reservado o inaccesible)
- Podemos reemplazar cualquiera de las aplicaciones del teléfono por otra de nuestra elección.

Este es el verdadero potencial de Android y lo que lo diferencia de su competencia: un gran control por parte del usuario del software que se va a ejecutar en su dispositivo móvil [11],[12].

5.6.6. Modelo de datos

Android aporta sus propios conceptos para almacenar y compartir datos entre aplicaciones, aunque en última instancia dichos conceptos se terminen implementando mediante enfoques tradicionales (en la mayoría de los casos).

Disponemos de :

Sistema de Archivos Tanto internos de la aplicación como externos por medio de la compatibilidad con tarjetas *SD*

SQLite Una base de datos relacional (*SQLite*), que no tiene todas las funciones de los productos de base de datos *cliente/servidor* comerciales, pero ofrece todo lo necesario para almacenamiento local de datos, a la vez que resulta rápida y sencilla de utilizar.

SharedPreferences Es el objeto que permite almacenar el estado global de la aplicación, se pueden crear privadas de la aplicación o hacerlas accesibles para otras aplicaciones. Se accede a ella a través del contexto (*Context*) desde el que se trabaje, muchas clases de *Android* tienen una referencia a *Context* (e.g. *Activity* o *Service*)

Uri Un enfoque basado en *URI* para compartir datos entre aplicaciones denominado *Content Provider*, como cada aplicación se ejecuta en su propio proceso (normalmente), y los archivos y datos que almacena no son accesibles para otras aplicaciones, ésta es una buena manera de compartir, consultar, añadir, actualizar o eliminar información entre distintas aplicaciones.

En nuestra aplicación se ha hecho uso de los *Content Provider*, sobre todo a la hora de utilizar la gestión del calendario con una parte de la *API* proporcionada por *Google Calendar Provider*, perteneciente a la *Android Open Source Project (AOSP)*, luego no es de código cerrado y no estamos encadenados a ninguna aplicación en concreto.

Esto implica que prácticamente cualquier aplicación que gestione calendarios en *Android* podrá visualizar los resultados de las planificaciones).

Además, esto nos permite almacenar los intervalos resultantes de las planificaciones como eventos de un calendario que se integra de manera natural con los demás calendarios que hallan en el sistema (si bien con elementos distintivos como el color en el que aparece el evento, nombre, etc...), luego las aplicaciones de agenda de *Android* pueden mostrar los eventos de todos los calendarios integrados de manera homogénea.

> e.g. en un mismo día aparecerán tanto el resultado de las planificaciones de nuestra aplicación como otros eventos introducidos por el usuario y otras aplicaciones

La herramienta de almacenamiento de datos que utilizamos con mayor profusión es la base de datos **SQLite**, por la naturaleza de la aplicación hemos considerado que es la mejor solución para tratar adecuadamente la no despreciable cantidad de información que puede llegar a generar.

El modelo relacional es el de la figura 24

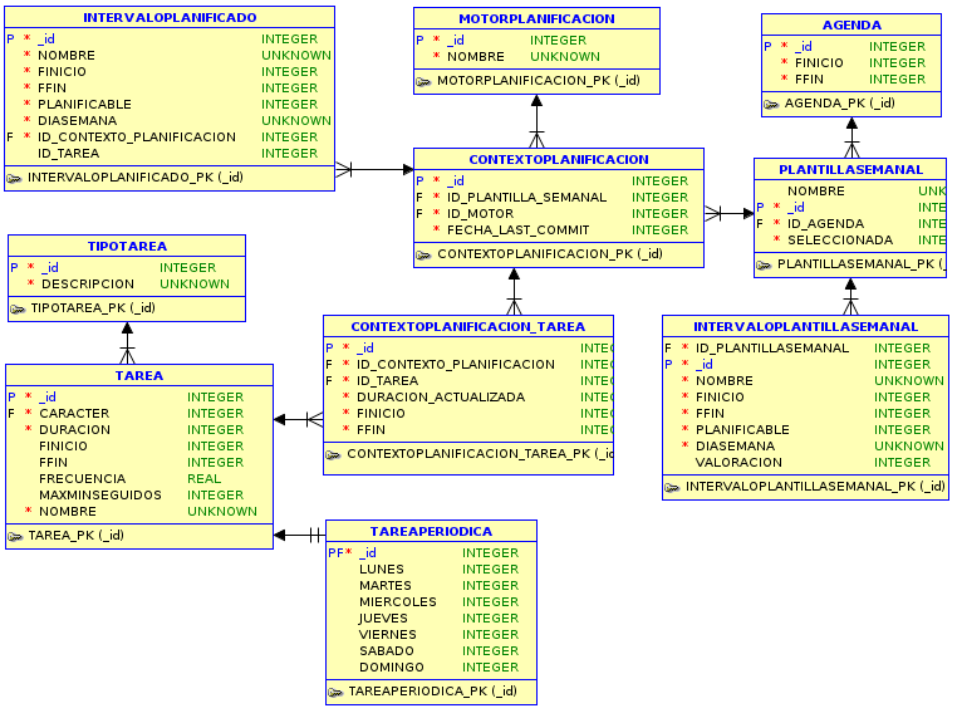


Figura 24: Modelo de datos de la aplicación *BBDD SQLite*

6. Conclusión

6.1. Tests de rendimiento

Para evaluar el rendimiento de la aplicación, se ha procedido a ejecutar una serie de *tests* que emulan peticiones que serían de uso común a la aplicación, variando el número de tareas, intervalos de tiempo disponible, y duración de las mismas para los distintos motores de planificación disponibles.

Se creó un código *Java* que obtenía los promedios de *50* ejecuciones consecutivas de los tres motores de planificación (*Rms*, *Dms*, *Edf*) aplicados sobre tareas esporádicas de 5 horas de duración intentando planificar de manera sucesiva *5, 10, 20 y 50* tareas en una plantilla semanal cuyos intervalos disponibles para trabajar estaban distribuidos en fragmentos de 60 minutos, a lo largo de toda la semana.

Grosso modo parte del fragmento de código utilizado para testar los tiempos de respuesta sería éste

```
private static long medir(IPlanificador algoritmo, Integer nTareas,
    Integer dTareas, Integer nIntervalos, Integer dIntervalos) {

        Calendar cal = new GregorianCalendar();

        Agenda agenda = new Agenda(...);

        List<IntervaloPlantillaSemanal> plantilla =
        new ArrayList<IntervaloPlantillaSemanal>();

        for(DIASSEMANA d : diasDeLaSemana){
            ...
            plantilla.add(new IntervaloPlantillaSemanal(d, ini,
                cal.getTime(), "free" , true, 5));
            ...
        }

        agenda.asignaIntervalosPlantillaSemanal(plantilla);

        List<Tarea> tareas = new ArrayList<Tarea>();

        for(long i=0; i< nTareas; i++){
```

```
            tareas.add(new TareaEsporadica(i, "tarea"+i, dTareas,
                dIntervalos, iniAgenda, finAgenda));
        }

        long ini = System.nanoTime();
        try {
            algoritmo.planificar(agenda, tareas);
        }
        long fin = System.nanoTime();

        return fin-ini;
    }
```

Para obtener la duración se hacía una llamada a *System.nanoTime()* antes y después de la ejecución de cada algoritmo en cada una de las 50 iteraciones acumulando el resultado para hacer el promedio.

Para las pruebas se utilizó una máquina virtual de *Java java-7-openjdk-armhf* ejecutando la aplicación desde la linea de comandos en un equipo con un procesador *ARM Cortex-A8 800MHz* y sistema operativo *Debian GNU/linux jessie/sid* desde consola.

Con los siguientes resultados mostrados en la figura 25

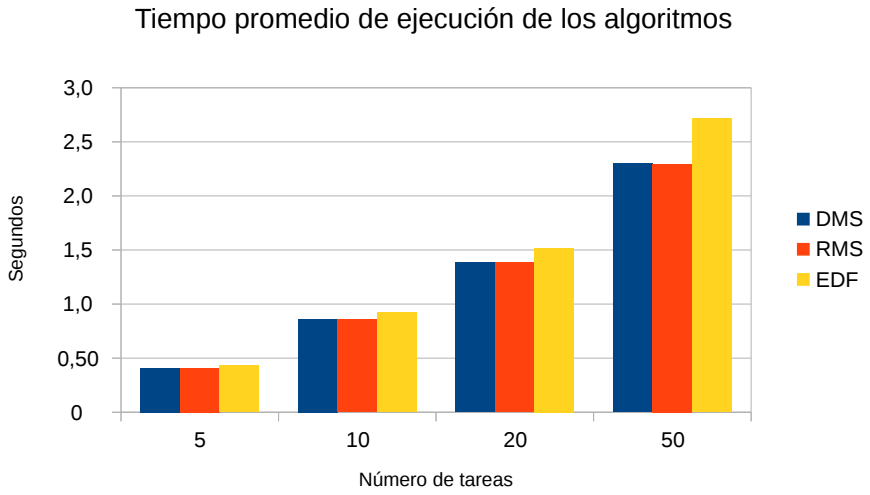


Figura 25: Gráfica de tiempos promediados para los distintos motores de planificación

Número de tareas	Promedio DMS	Promedio RMS	Promedio EDF
5 Tareas	0,40707”	0,40666”	0,43150”
10 Tareas	0,86023”	0,85885”	0,92299”
20 Tareas	1,38599”	1,38345”	1,51861”
50 Tareas	2,30011”	2,29361”	2,71523”

Con estos datos se puede apreciar que el algoritmo con asignación de prioridades dinámicas (*EDF*) requiere un mayor tiempo de *CPU* debido a que además de hacer un reparto (en función de las prioridades) de las tareas por cada intervalo disponible, debe ordenar tras cada intervalo ya planificado la lista de tareas con sus nuevas prioridades (en función de cuan cercana esté cada una de su *deadline* correspondiente, como exige dicho algoritmo), en la gráfica se aprecia ese coste adicional, tanto mayor cuanto más tareas haya.

6.2. Uso de librería Optaplaner

De los siguientes resultados de rendimiento sobre la elección de los intervalos adecuados para las planificaciones teniendo en cuenta las preferencias del usuario efectuadas en [8] se puede deducir que:

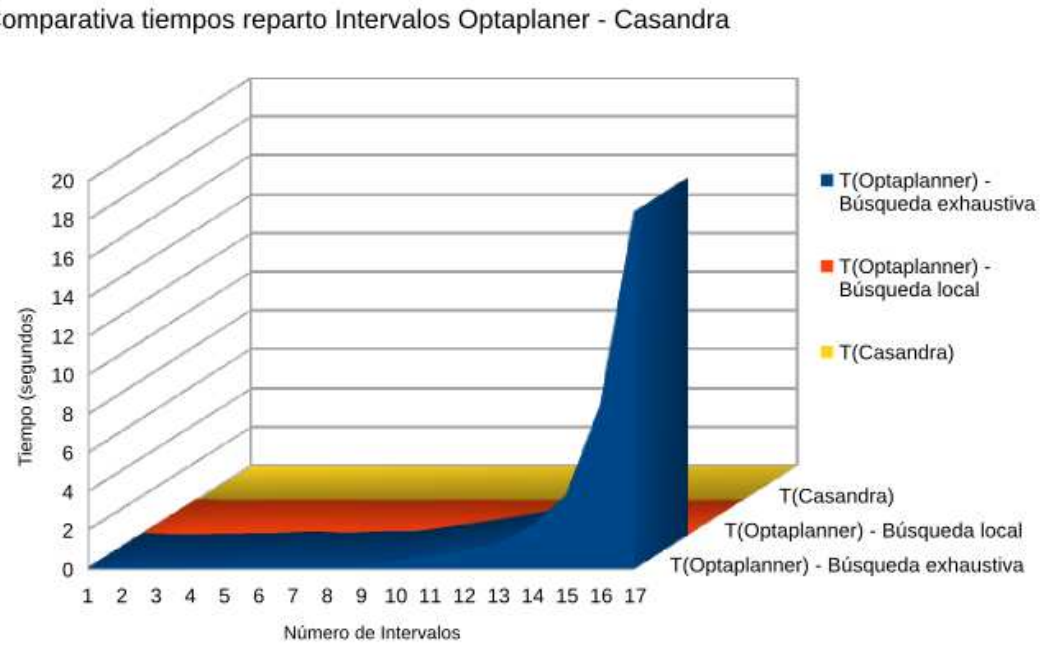


Figura 26: Comparativa asignacion intervalos

En igualdad de condiciones, es decir con búsqueda local los tiempos obtenidos por nuestro desarrollo y por *Optaplanner* son equivalentes, el reparto es casi inmediato, *Casandra* usa el heurístico de conocer qué intervalos son los deseados por el usuario y genera intervalos en función de dicha información haciendo prácticamente un reparto en un orden de tiempo lineal y *Optaplanner* lo resuelve usando la misma información pero con

el método indirecto de las puntuaciones negativas de los *hard y soft Scores*, el método exhaustivo se dispara en complejidad, y no es operativo.

En definitiva, hay que retorcer el diseño original de la aplicación para poder adaptarlo a las exigencias del modelo de *Optaplanner* que a su vez es pesado y consume más recursos del dispositivo móvil ya que provee mucha funcionalidad, pero que no será aprovechada en nuestra aplicación.

Si la aplicación tuviera que dar soporte a problemas de optimización de alguna otra índole sería una opción más acertada haber incluído soporte a esta librería desde la fase de diseño de la aplicación, pero incluirla en una fase tardía del desarrollo es un verdadero quebradero de cabeza.

6.3. Conclusiones académicas

Este trabajo, según mi entender, obligó a afrontar el desarrollo completo de una aplicación de modo autónomo (aunque tutelado), con libertad para explorar soluciones alternativas, esto fuerza a dotar de sentido un proyecto, es decir tomar perspectiva y pensar de manera global, lo cual no es habitual a lo largo de la formación como estudiante, se entiende que para montar juguetes hace falta armarse de las piezas suficientes.

La adquisición de nuevos conocimientos y divagar entre libros de mi interés han sido quizás los puntos más dulces, compaginar toda esta tarea con la vida personal y el trabajo (que resta tiempo sin cesiones), hacen que la experiencia sea exigente.

6.4. Posibles mejoras

Hay multitud de mejoras que se le podrían aplicar a esta aplicación, por enumerar algunas

- Podría mejorarse el aspecto monolítico con el que se tratan los horarios en el proyecto, habilitándose opciones para prolongar tareas a lo largo de una ilación de horarios de períodos de tiempo disjuntos, o mezclar distintos horarios coincidentes en el tiempo agregando sus intervalos y resolviendo los conflictos que pudieran hallarse.
- Generar módulos o facilitar la integración con plataformas de aprendizaje tipo *Moodle*
- Resolver excepciones de la planificación de manera automática; se podrían capturar *indicaciones* por parte del usuario para resolver excepciones de la planificación por

falta de tiempo, como pudieran ser el cese de determinadas tareas periódicas o la eliminación de deadlines de algunas tareas, etc...

- Se podría agregar la posibilidad de gestionar dependencias entre tareas, de modo que algunas tareas forzosamente han de estar satisfechas antes de poder incoar las subsiguientes.

A. Manual de usuario

Casandra es una aplicación orientada al mundo estudiantil, que se encarga de gestionar una agenda digital de modo que además de ver cuales son los horarios programados, se pueda introducir los momentos disponibles a lo largo de la semana para el estudio, y pedirle que planifique las tareas y las horas de trabajo de modo que queden entregadas (o aprendidos los contenidos) antes de su fecha límite (o pruebas escritas), sin necesidad de ir haciendo excesivas cábalas.

Es decir, que cuando se vayan acumulando tareas por hacer, basta con que se le comunique el tiempo que se necesita dedicar a la tarea y cuando es la entrega, y la agenda generará eventos en el calendario de *Android* de modo que si se cumplen, se lograría llegar a tiempo a la entrega de todas las tareas (o con los exámenes preparados, etc. . .)

Se le puede pedir a la aplicación que cuando se trabaje en una materia, no lo haga más de un determinado tiempo, de modo que el motor de planificación cuando vea que se va a exceder ese límite, tome cartas en el asunto y emplace la tarea a la que se ha dedicado el máximo tiempo permitido al siguiente día disponible y para aprovechar el tiempo que aun quede, seleccionará otra tarea diferente de las aún pendientes (si hubiera).

¿Qué ocurre si un día no trabajo? Cuando pase el tiempo y se vuelva a entrar a la aplicación ésta le preguntará si ha realizado o no las tareas (desde que se planificaron hasta ese instante), si algunas no se pudieron realizar y otras sí, se planificará de nuevo teniendo en cuenta el trabajo que se hizo y el que no, así que de un modo automático tenga la agenda actualizada con datos reales.

Hay veces que es imposible realizar las tareas a tiempo, en este caso, se le propondrá que, o bien estudie desestimar algún trabajo, o que amplíe el tiempo disponible para trabajar en el horario, cosa que logramos editando las tareas o el horario en la aplicación y pulsando el botón planificar de nuevo.

A.1. Interfaz de usuario

Al arrancar casandra encontramos una pantalla principal que consta de los siguientes elementos: Un cajón de navegación que se abre y cierra pulsando en el botón de inicio (4), un botón de menú (5), tres botones de acceso directo a las pantallas principales (1-3) y para la parte lúdica de la aplicación un marcador (6) con nuestra puntuación.

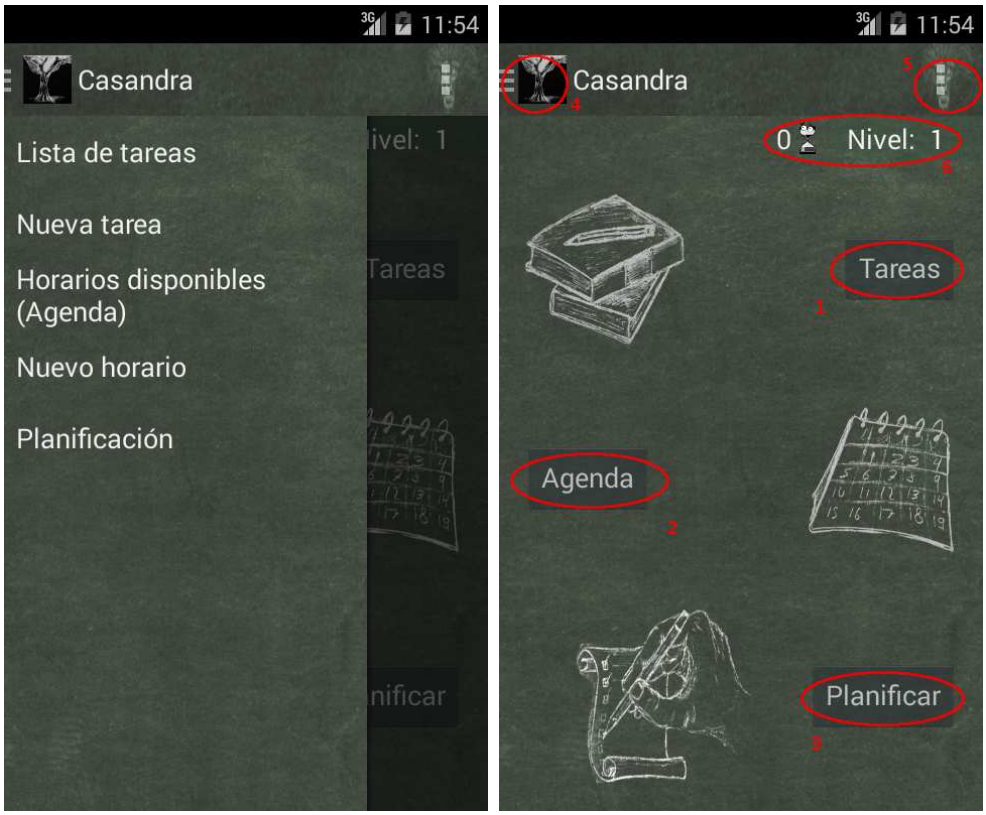


Figura 27: Manual - Pantalla principal

Botón Tareas (1) Es un acceso directo a la parte de la aplicación encargada de la introducción, edición y eliminación del trabajo con que se va a alimentar una agenda. Se corresponde con la opción 'Lista de tareas' del cajón de navegación.

Botón Agenda (2) Pulsando accedemos, del mismo modo que pulsando en 'Horarios disponibles' del cajón de navagación, a una pantalla que nos permitirá definir horarios y .

Botón Planificar / Consultar planificación (3) Una vez introducidas algunas tareas y un horario, se pueden lanzar planificaciones haciendo uso de esta tercera opción.

Botón Inicio (4) Pulsando en la esquina superior izquierda, que hemos llamado botón de inicio, se abrirá o cerrará el cajón de navegación. En otras pantallas aparecerá en esta esquina un signo <en lugar de las tres barras horizontales, lo que indicará que al pulsar volvemos a un nivel superior.

Botón de opciones (5) Con el mismo efecto que el botón de opciones del teléfono, nos permite acceder a la opciones descritas en el apartado A.6

Marcadores de nivel y puntuación (6) Aquí aparecen los puntos que hemos ido ganando con nuestro uso de la aplicación.

Figura 28: Manual - Ganar puntos

Los puntos se van ganando conforme se hace seguimiento de las tareas, cada vez que se regresa a la aplicación se premia el interés por saber como van tus planificaciones con saturninos y cuantos más objetivos se logren (tareas terminadas) más se incrementarán tus saturninos.

A.2. Definir tareas

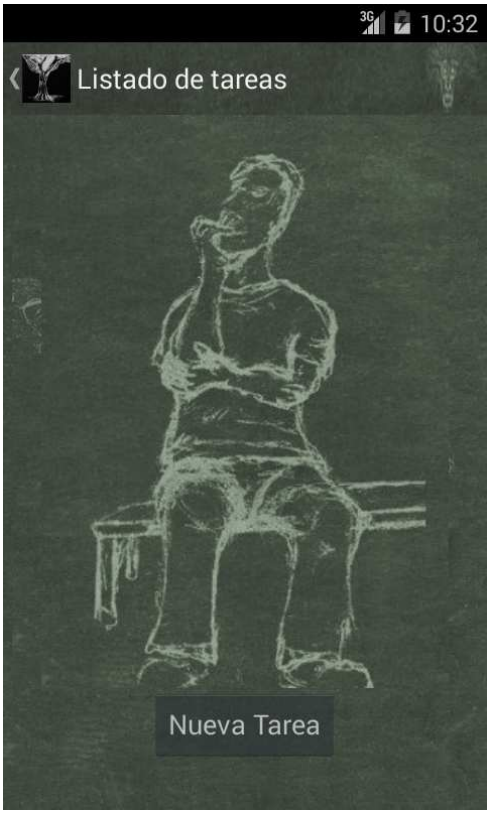


Figura 29: Manual - Listado tareas vacío

Cabe reseñar antes de empezar, que las tareas no están ligadas a ningún horario en concreto, se puede cambiar de horario y hacer otras planificaciones con las mismas tareas, son conceptos independientes, por un lado está el tiempo del que se disponga y por otro la cantidad de trabajo por hacer, es en *planificar* donde se unen ambos.

La primera vez que se accede al apartado de *Tareas*, al no haber ninguna insertada, aparecerá la lista vacía.

Para **insertar una tarea** podemos utilizar el botón 'Nueva Tarea' que aparece en la parte inferior.

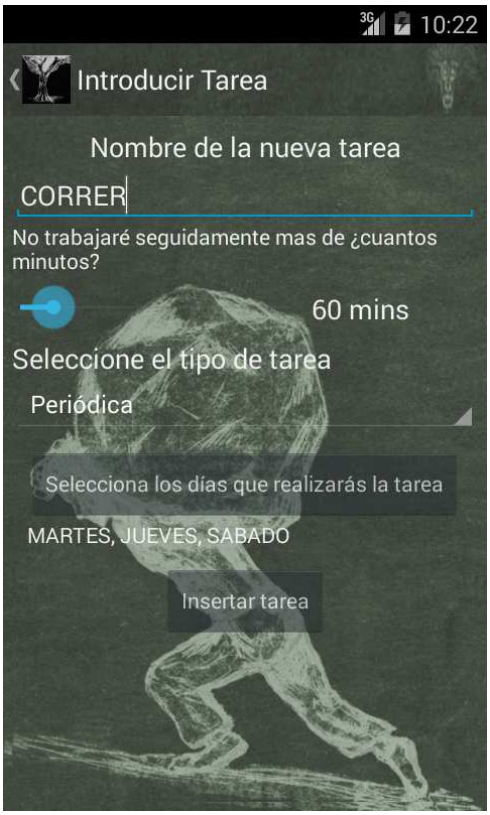


Figura 30: Manual - Introducir tarea periódica

Ésta puede ser de dos tipos periódica o esporádica, la captura anterior se refiere a la periódica ya que así se ha hecho la elección en el apartado *seleccione el tipo de tarea*. Se le pone un nombre, y se le indica también el tiempo que durará.

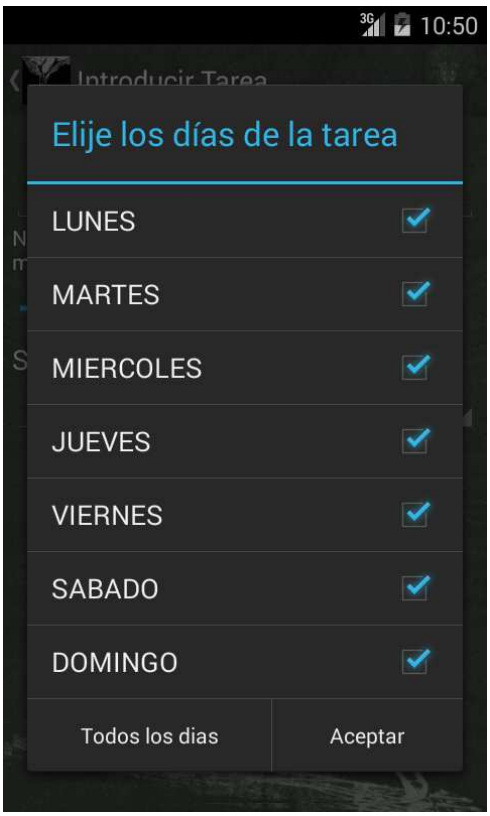


Figura 31: Manual - Elegir días de la semana para tarea periódica

Esto significa que esta tarea se planificará a lo largo de los límites de la agenda, los días elegidos (aparece una ventana de diálogo que nos permite elegir qué dias de la semana la vamos a realizar)

Es la opción adecuada para definir aquellas tareas como hacer ejercicio, o practicar con algún instrumento, que no tienen una cantidad de horas para ser terminadas. Una vez hecha la elección se pulsará el botón *insertar tarea*. Y ya formará parte del conjunto de tareas de tu aplicación.

El otro tipo son las tareas esporádicas, estas requieren algo más de información como una fecha de inicio, de finalización (ambas se introducen pulsando las fechas que aparecen a ambos lados de "<TAREA>") y la cantidad de tiempo estimado para la realización total de la tarea, por ello varía un poco la interfaz de entrada de datos, resultando

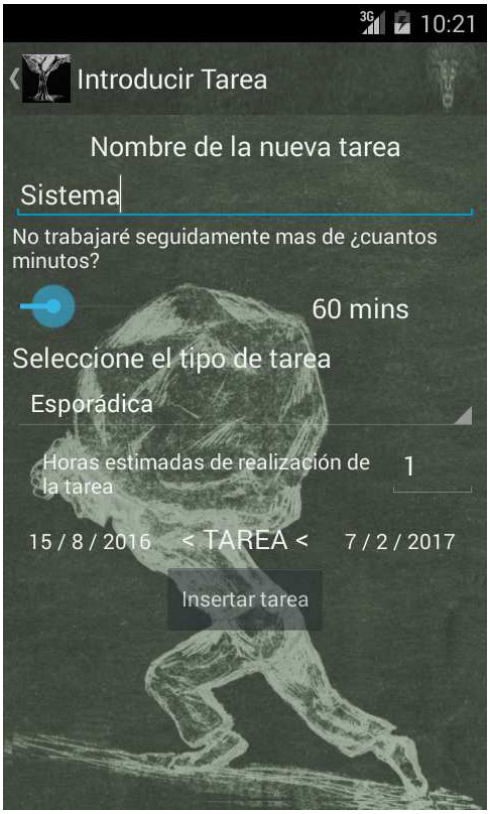


Figura 32: Manual - Introducir tarea esporádica

El tiempo introducido en este tipo de tareas (al igual que en las periódicas) es muy importante, si no se lograra reunir el total del tiempo estimado para la realización de la tarea antes de la fecha de finalización de la misma, la aplicación reaccionará diciendo que la planificación no es factible con los datos introducidos, pero eso se verá en el apartado A.4 *planificar*.

Una vez se hayan introducido una serie de tareas en nuestra aplicación, el acceso al apartado de tareas, muestra un listado de las mismas, y el mismo botón de inserción de nueva tarea en la parte inferior de la pantalla del dispositivo. A continuación se puede ver un ejemplo con datos de prueba.

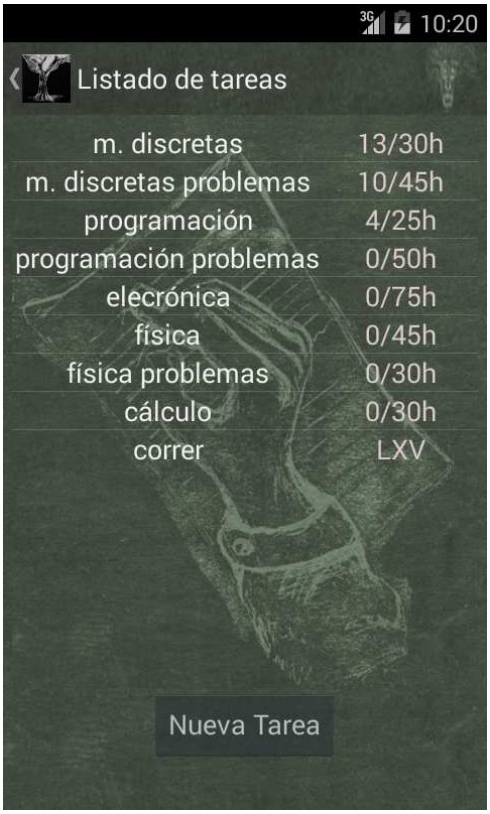


Figura 33: Manual - Listado tareas con inserciones

Lo primero a destacar es que ahora, aparte del nombre de la tarea, en la columna de la derecha, aparecen **dos tipos de datos**, y estos tipos de datos vienen en función del tipo de tarea de que se trate

- ◇ Si es una tarea *esporádica* aparece una fracción que indica la cantidad de tiempo que efectivamente se lleva realizado de la tarea en total, es decir $\frac{tiempo\ trabajado}{tiempo\ total\ de\ la\ tarea}$

  e.g. $\frac{0}{tiempo\ total\ de\ la\ tarea}$ en una tarea que está por comenzar

- ◇ Si se trata de una tarea *periódica* aparecen una serie de letras que no son más que las iniciales de los días de la semana en que se espera que tal tarea se lleve a cabo

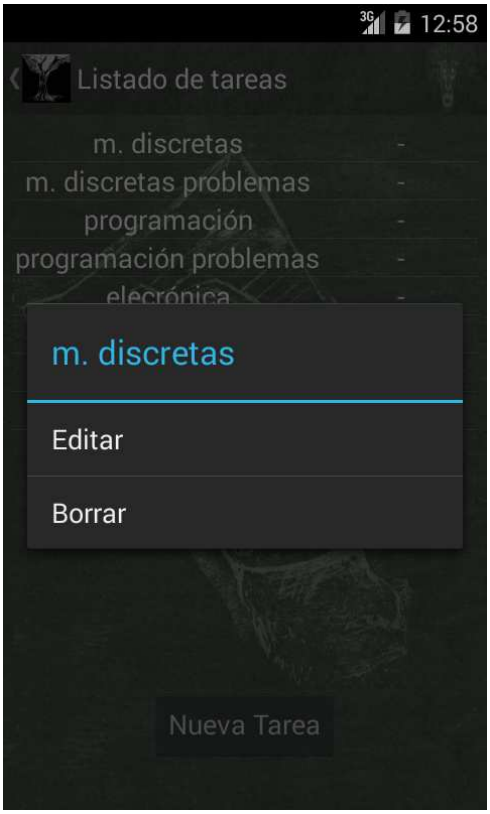


Figura 34: Manual - Edición y borrado en el listado de tareas

En esta ventana, se pueden editar y borrar las tareas a nuestro antojo sin más que mantener una pulsación larga sobre el elemento de la lista deseado, y aparecerá el siguiente menú

Cuando existe una planificación en curso, nos aparece la opción adicional de incluir la tarea seleccionada en la planificación, como se muestra en la figura.

A.3. Definir horario y tiempos libres en la Agenda

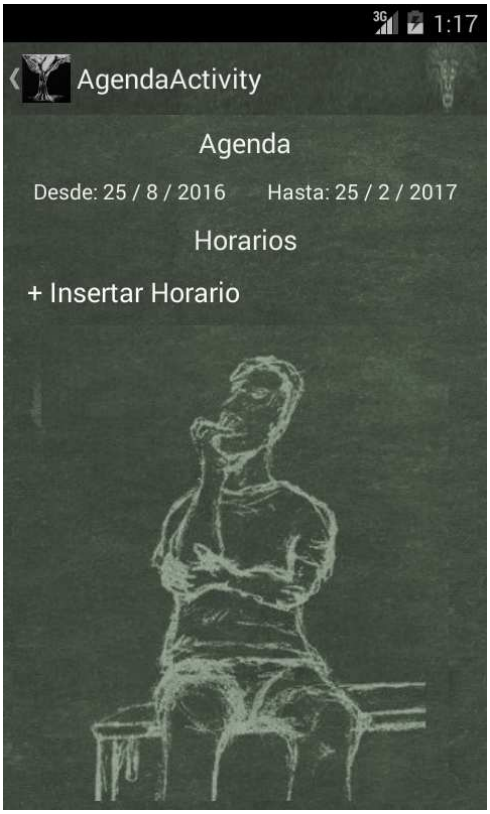


Figura 35: Manual - Agenda vacía de horarios

La primera vez que se entra en la *Agenda*, en la parte inferior de la pantalla ocurre lo mismo que cuando se accede a *Tareas*, que no existe ningún horario aun y muestra la lista vacía, pero la parte superior, se pueden definir los límites temporales de la *Agenda*.

Para cambiar los límites de la agenda, sólo hay que efectuar una pulsación larga sobre la fecha que viene por defecto, si se hace una pulsación normal aparece un bocadillo en pantalla explicando qué hacer para cambiar la fecha.

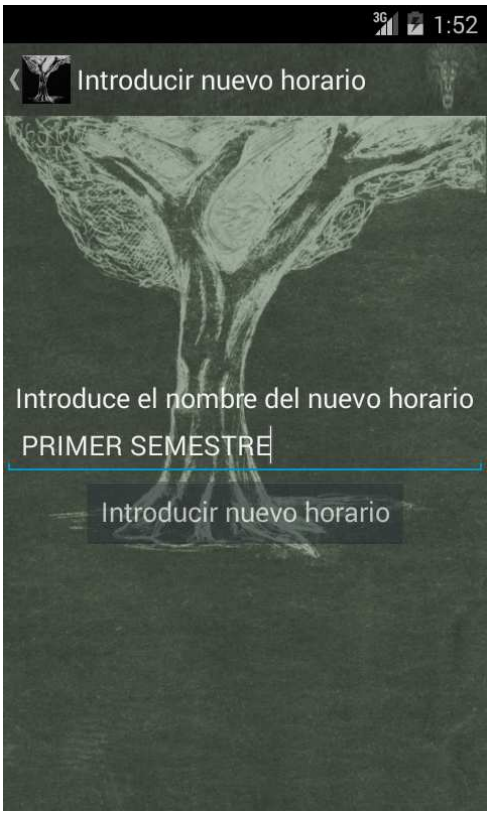


Figura 36: Manual - Agenda creación de un nuevo horario

Si se quiere añadir un nuevo horario, hay que pulsar el texto + *Insertar horario* y se accederá a otra actividad que nos va a pedir el nombre del horario

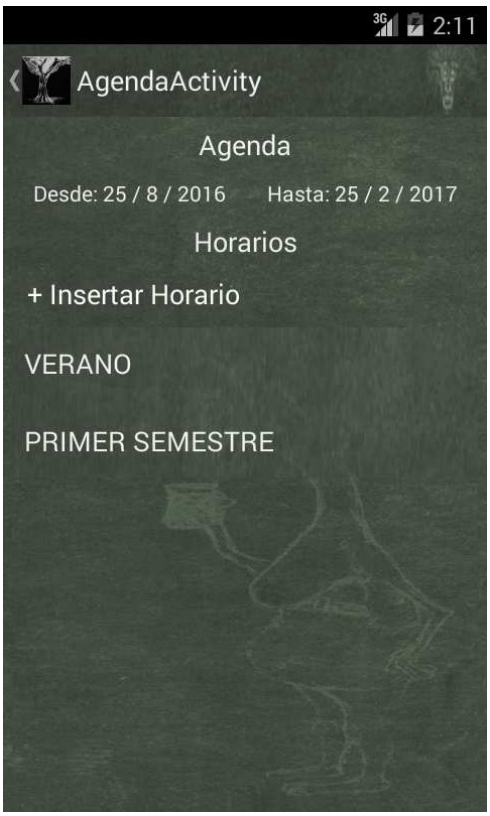


Figura 37: Manual - Agenda con horarios insertados

Una vez introducido e insertado (pulsando el botón de *Introducir nuevo horario*, se volverá al menú principal de *Agenda*, donde ahora en el listado aparecerán todos los horarios que se hayan introducido.

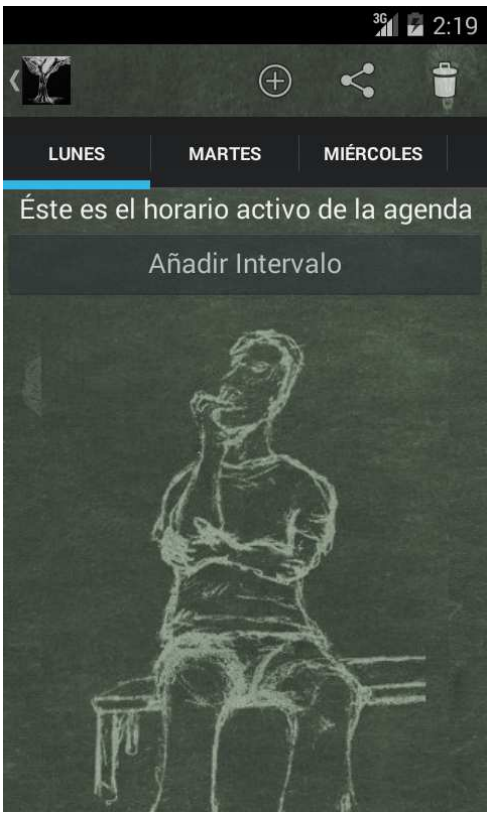


Figura 38: Manual - Insertar intervalo horario con agenda activa

El siguiente paso, una vez insertado un nombre para un horario, es precisamente *hacer* el horario, esto es introducir una serie de intervalos de tiempo, de dos tipos,

- ◇ Aquellos intervalos que están fijados por obligaciones (e.g. clases)
- ◇ Aquellos intervalos de tiempo en los que se pueden realizar trabajos, estudiar, etc. . .

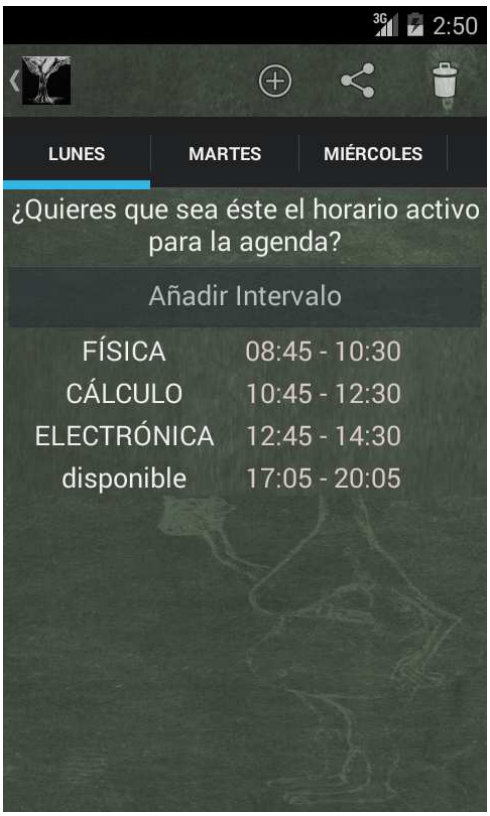


Figura 39: Manual - Insertar intervalo horario con agenda no activa

Para introducir cualquiera de estos intervalos, primero hay que pulsar de la lista de horarios aquel en el que se quiera trabajar. Y entraremos en la siguiente actividad, que es la inserción de intervalos horarios por cada día de la semana. Cuando se accede a un horario recién creado, éste aun no contiene intervalos.

En esta ventana se pueden observar varias cosas, la primera son los días de la semana que aparecen en la parte superior de la ventana, al deslizarnos sobre este menú nos permitirá movernos a lo largo de los dias de una semana e introducir los intervalos de tiempo para un día en concreto.

Justo bajo los días de la semana aparece un texto que dice que éste es un *horario activo*, como la aplicación permite introducir tantos horarios como se considere oportuno (pero no tiene sentido (en principio) planificar tareas sobre más de un horario a la vez), en todo momento sólo habrá un horario activo a la vez.

Cuando se genera un horario nuevo, se convierte por defecto en el horario activo y si hubiera algún otro activo en ese momento dejaría de estarlo. Si se accede a un horario que no es activo este mensaje lo indicaría diciendo *¿Quieres que sea éste el horario activo para la agenda?* , si se pulsa sobre el texto, lo haremos el horario activo de nuestra aplicación.

Es muy sencillo cambiar la propiedad de que un horario sea el activo, pero es importante, ya que es sobre el que se lanzarán las nuevas planificaciones.

Bien, ahora nos queda añadir los intervalos de tiempo, los hay como ya dijimos de dos tipos el fijo y el que se dedicará para trabajar (disponible). Para ambos se pulsará el botón añadir intervalo una vez por la barra superior nos hayamos desplazado al día deseado. Y aparecerá la siguiente actividad

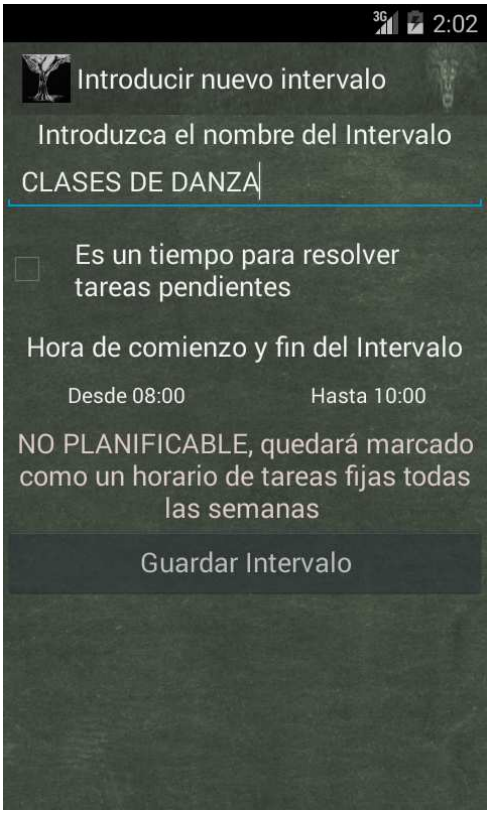


Figura 40: Manual - Insertar intervalo horario fijo

Mientras no se chequee la opción de *Es un tiempo para resolver tareas pendientes*, se tomará este intervalo como uno **fijo**, el intervalo de tiempo insertado no se usará para planificar, solo para que aparezca en el horario con un fin informativo (e.g. el horario de clase). Aquí se insertará la hora de comienzo y final del intervalo y un nombre (e.g. el nombre de la asignatura). Luego, se pulsa en guardar intervalo, y la aplicación ya tendrá en ese *Horario* el intervalo añadido, esta operación se puede repetir cuantas veces desee a lo largo de los días de la semana del *Horario* en cuestión hasta completar el horario del curso, a falta de los intervalos de tiempo que vaya a dedicar a trabajar, los intervalos disponibles.

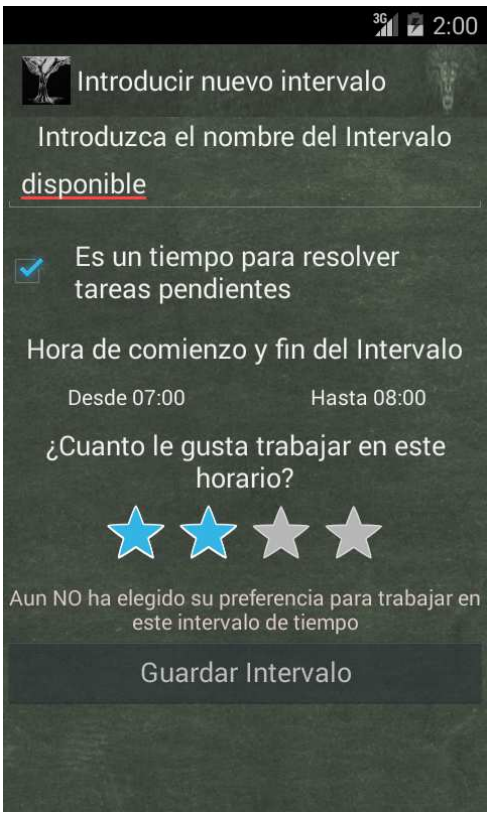


Figura 41: Manual - Insertar intervalo disponible en horario

Para introducir los intervalos **disponibles**, se hace exactamente igual que los fijos (pero no es necesario ponerle un nombre, en el calendario siempre aparecerán como *disponible*), marcando la opción de *Es un tiempo para resolver tareas pendientes*, en tal caso también se solicitará el **nivel de satisfacción** a la hora de tener que trabajar en ese intervalo concreto . Esta información será utilizada por la aplicación para tratar de encontrar un horario que se adapte a las preferencias del usuario.

El resultado final es un horario completo, tanto con las tareas fijadas en el horario de la semana (como las clases lectivas) como los intervalos de tiempo disponibles para trabajar, el cual podremos exportar y compartir, sustituir por otro importado o eliminar de la aplicación.

Para estas acciones disponemos de los botones del *actionbar*:

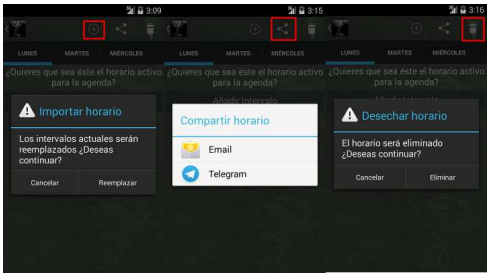


Figura 42: Manual - Importar, Exportar y Desechar horarios

A.4. Planificar

Bien, ya que disponemos de una serie de tareas a realizar, y un horario con intervalos de tiempo disponible para realizarlas, como hemos visto anteriormente, aquí es donde entra en juego la actividad de *Planificar*.

En resumidas cuentas coge un horario activo, un conjunto de tareas y las reparte de modo que todas las tareas se completen antes de que expire su fecha de finalización usando los intervalos disponibles de la agenda.

Cuando entramos a la actividad planificar desde la pantalla principal lo primero que nos aparece es el listado de las tareas insertadas en la aplicación (tanto periódicas como esporádicas) para que, o bien se seleccionen todas mediante la opción del diálogo o bien se marquen aquellas que hemos decidido planificar:

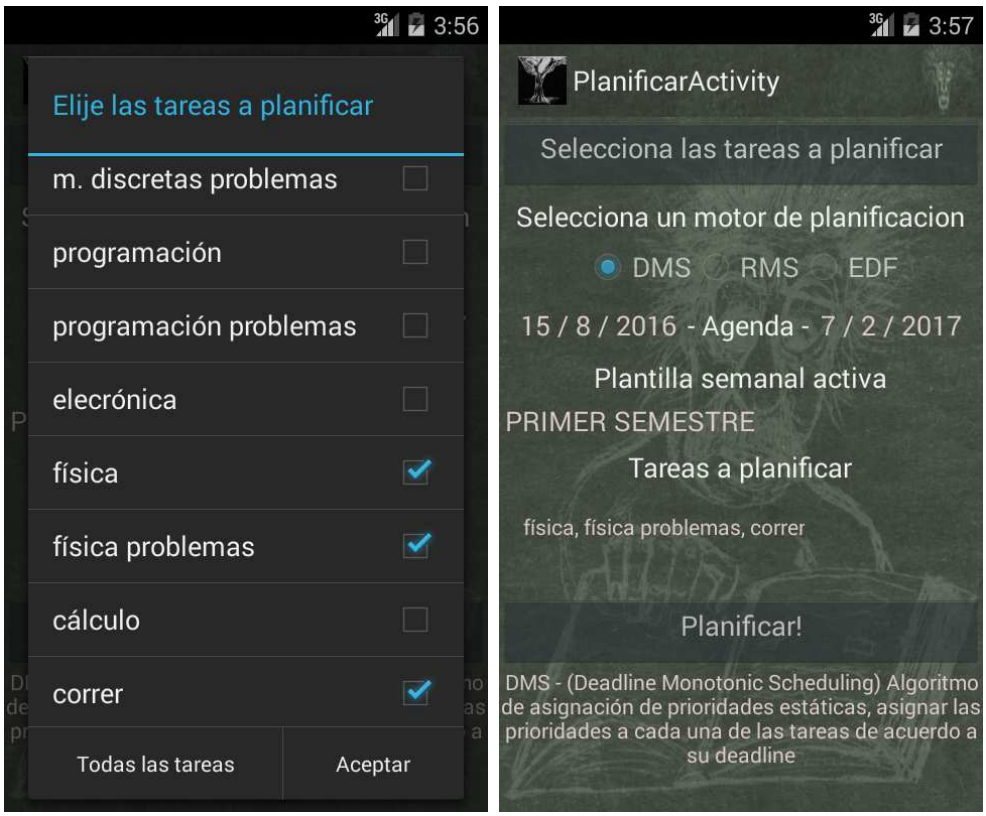


Figura 43: Manual - Selección de tareas a planificar

Siempre se puede volver a hacer aparecer el diálogo pulsando el botón *Selecciona las tareas a planificar* en la parte superior de la pantalla.

Vemos como automáticamente selecciona la *Agenda* activa, los límites de fecha del calendario, y el horario (plantilla semanal activa).

Figura 44: Manual - Planificando

Ya solo queda elegir uno de los motores de planificación disponibles (cada uno probablemente, dará una planificación distinta) y pulsar el botón *Planificar*. Aparecerá una ventana de espera.

Con este acto ocurren dos cosas, la primera es que se resuelve el problema de planificación intentando utilizar el menor número de intervalos de los introducidos como disponibles en el horario prevaleciendo según las preferencias del usuario, resultando una serie de asiganciones de tareas a intervalos. La segunda cosa que sucede es que estos intervalos se insertan como un calendario propio llamado *Planificación Casandra* integrado en la plataforma *Android* con el resto de calendarios.

Figura 45: Manual - Calidad de una planificación

Además se informa al usuario de cuánto se ha ajustado el resultado a sus preferencias mediante un diálogo con la media de estrellas de los intervalos elegidos y una ilustración, que indica el uso.

A menor número de estrellas indica un uso intensivo de intervalos de tiempo disponibles que no agradan al usuario.

Figura 46: Manual - Resultado de una planificación

El resultado a la hora de visualizarlo con el calendario de *Android* (probablemente variará según la aplicación que se utilice y la versión de la misma) y se tendrá acceso a la funcionalidad que aporte dicha aplicación.

Si la planificación *ha tenido éxito*, entonces nuestra aplicación saltará a la aplicación que tenga en su dispositivo para visualizar calendarios a la fecha en la que esté el primer intervalo planificado introducido. Siempre podrá acceder a su calendario y ver esta planificación, (aunque éstas planificaciones pueden ir cambiando con el tiempo ver apartado siguiente), y se podrán hacer las operaciones habituales con los eventos de un calendario normal de *Android*.

Figura 47: Manual - No se puede planificar

Si la planificación *no ha tenido éxito*, es decir hemos tenido o muy poco tiempo disponible o demasiada carga de trabajo, en tal caso saltará un diálogo de aviso con una serie de sugerencias y una ventana que indica qué tarea ha vencido su fecha de expiración.

A.5. Confirmación de Tareas

Figura 48: Manual - Procrastinar tareas

Una vez hecha una planificación, se hará un seguimiento de las tareas a ver si se están llevando a cabo o no a lo largo del tiempo, es decir:

Cada vez que se accede a la aplicación se comprueba la fecha actual del sistema y la fecha en la que se accedió por última vez a la aplicación, con el intervalo de tiempo calculado entre la última vez que se accedió a la aplicación y el día de hoy se mira qué tareas deben estar hechas (acumula el tiempo de cada intervalo planificado para cada una de las tareas en este tiempo), y se pregunta al usuario al entrar si las ha hecho o no.

Esto se hace mostrando al usuario la lista de tareas de modo que seleccione aquellas que realmente ha llevado a cabo. Con esta nueva información el sistema **recalcula** la planificación actualizando los eventos del calendario de manera automática.

Otra cosa a tener en cuenta es que en el menú inicial el botón de *Planificar*, una vez hecha una planificación cambia su estado a *Ver planificación* de modo que cuando se acceda a la aplicación y ya haya una planificación en marcha se pueda simplemente consultar.

A.6. Opciones del menú

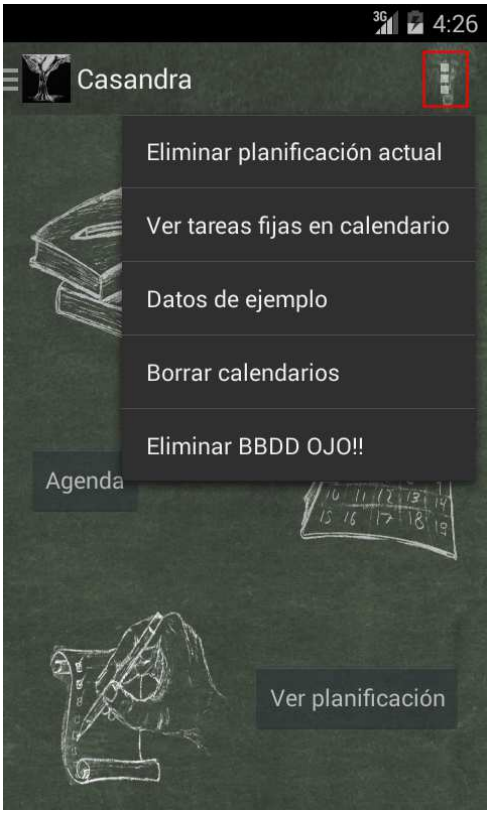


Figura 49: Manual - Menú de opciones

Estas son todas las opciones que aparecen al pulsar el botón de menú del dispositivo electrónico. A continuación describiremos brevemente el funcionamiento de cada una

Eliminar planificación actual Si se quiere lanzar una nueva planificación y descartar la existente, elija esta opción

Ver tareas fijas en Calendario Esta opción hace que en el calendario de la aplicación no solo se muestre las tareas planificadas, sino que también aparezcan las fijas

Datos de ejemplo Introduce unos cuantos datos de ejemplo, para explorar la aplicación sin necesidad de ir insertando información

Borrar Calendarios Elimina los eventos generados en el calendario de la aplicación. Usar con precaución, para que vuelvan a aparecer habría que lanzar de nuevo una planificación

Eliminar BBDD sirve para resetear el estado de la aplicación, eliminará absolutamente toda la información de tareas horarios y planificaciones, debe usarse con mucha precaución

Referencias

[1] Inc. Any.do. `http://www.any.do/`, 2016. App Android Any.do.

[2] Alex Baker. `https://play.google.com/store/apps/details?id=org.tasks`, 2016. App Android Astrid.

[3] Trish Sarson Chris Gane. *Structured systems analysis : tools and techniques*. Prentice-Hall, first edition, 1979.

[4] Juan Antonio Fernández Madrigal. Planificación de tareas en sistemas en tiempo real. *Departamento de Ingeniería de Sistemas y Automática, Universidad de Málaga*, 2010.

[5] Roque Marín Morales José T Palma Méndez. *Inteligencia Artificial*. McGraw Hill, first edition, 2008.

[6] Red Hat. `http://www.optaplanner.org/`, 2016. Optaplanner.

[7] Ralph Johnson y John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, first edition, 1994.

[8] Elisabeth Freeman Erich Freeman. *Head First Design Patterns*. O'Reilly, first edition, 2004.

[9] Scott Chacon. *Pro Git*. Apress, first edition, 2009.

[10] Aatif Khan. What is art? *http://www.addictivetips.com/android/art-vs-dalvik-android-runtime-environments-explained-compared/*, 2014.

[11] Aurora Rodríguez. `http://androideity.com`, 2016. Aurora Rodríguez.

[12] Google. `http://developer.android.com`, 2016. Android Developers.