Universidad de Valladolid

Escuela de Ingeniería Informática de Valladolid

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Cliente ligero de Plastic SCM para tabletas

Autor:

D. Sergio Luis Para

Tutor:

D. Benjamín Sahelices Fernández

Resumen

El Software de Control de Versiones se ha convertido, desde su aparición, en una herramienta fundamental para la gran mayoría de desarrolladores, en cuanto a que permite no sólo mantener versiones de un fichero, sino ser capaz de revisar cómo ha cambiado este a lo largo del tiempo.

Entre la gran cantidad de software de control de versiones que existe actualmente (algunos software libre y gratuito, como *Git* o *Subversion*, y otros propietarios, como *Perforce*), se encuentra, desde finales del año 2006, Plastic SCM.

Aunque inicialmente estuviese disponible únicamente para Windows -con posibilidad de ejecutarse en GNU/Linux y macOS gracias a diversas librerías de compatibilidad-, el equipo de Códice Software ha invertido, a lo largo de los últimos dos años, un esfuerzo considerable en proporcionar a los usuarios de estos dos últimos sistemas operativos una interfaz de usuario y experiencias nativas. El siguiente salto exploratorio en pos de satisfacer las necesidades de sus clientes era, por tanto, una vez alcanzados los tres sistemas operativos de escritorio mayoritarios, las plataformas móviles, permitiendo la revisión de código no únicamente en un ordenador, sino allí donde el usuario se encuentre. El presente documento trata sobre el desarrollo para Android y parcialmente para iOS de dicha aplicación móvil.

Abstract

Version Control Software has become, since it first appeared, into a fundamental tool for the vast majority of developers, as it allows them not only to keep different versions of a file, but in empowering them to review its changes across those revisions.

Since the end of the year 2006, Plastic SCM belongs to the different existing Version Control Software (some of them free –as in speech- like Git or Subversion, some of them proprietary like Perforce).

Although at its birth Plastic SCM was only available for Windows -with the possibility of running it in top of GNU/Linux and macOS through different compatibility libraries-, the team at Codice Software has invested a considerable effort in providing GNU/Linux and macOS's users a native user interface. The next frontier in order to satisfy their client's needs -after being available for the three most-used desktop operative systems- was mobile platforms, allowing code reviews without the need of a computer. The present document is about the development of the named mobile application for Android and partially for iOS.

Contenido

Resumen 2

Abstract 3

Capítulo I – Introducción 8

Capítulo II – Contexto tecnológico 9

2.1 - Software de control de versiones 9

2.2 – Plastic SCM 11

2.2.1 - Branching y merging 11

2.2.2 - Software de Control de Versiones Distribuido 12

2.2.3 – Software de Control de Versiones multiplataforma 14

2.2.4 – Especializado en diferencias *side by side* 15

Capítulo III – Análisis de requisitos 17

3.1 - User Stories 18

3.2 - Casos de uso identificados 21

3.2.1 - Añadir repositorios del primer servidor a la aplicación 21

3.2.2 - Añadir repositorios de un servidor a la aplicación 22

3.2.3 - Eliminar repositorios de la aplicación 23

3.2.4 - Actualizar credenciales de acceso a un servidor 23

3.2.5 - Cambiar el repositorio de visualización 24

3.2.6 - Visualizar el listado de changesets / ramas / etiquetas del repositorio 24

3.2.7 - Visualizar el Branch Explorer de un repositorio 25

3.2.8 - Cambiar las opciones de visualización del Branch Explorer 25

3.2.9 - Cambiar el filtro de fechas del Branch Explorer 25

3.2.10 - Obtener el listado de diferencias de un punto 26

3.2.11 - Visualizar las diferencias entre dos revisiones de un fichero 26

3.2.12 - Buscar objetos en el Branch Explorer 27
3.2.13 - Ejecutar una búsqueda avanzada 27
3.2.14 - Ordenar resultados de búsqueda según criterio 28
3.2.15 - Explorar el contenido del repositorio 28
3.2.16 - Cambiar el punto en el que se explora el contenido del repositorio 28
3.2.17 - Obtener un listado del historial de un fichero 29
3.2.18 - Descargar un fichero del repositorio 29
Capítulo IV – Diseño 31
4.1 - Diseño de la Interfaz de usuario 31
4.1.1 - Actividad de *login* 32
4.1.2 - Actividad principal 35
4.1.3 - Listados de ítems 38
4.1.4 - Explorador de ramas 40
4.1.5 - Vista de búsquedas (listado de ramas, *changesets* y etiquetas) 42
4.1.6 - Listado de diferencias 44
4.1.7 - Diferencias *side by side* 45
4.2 – Diagramas del diseño 47
4.2.1 - Diagrama general de *namespaces* 47
4.2.2 - Diagrama de clases del *namespace* BranchExplorer 48
4.2.3 - Diagrama de clases de alto nivel del *namespace* "Differences" 49
4.2.4 - Diagrama de clases de alto nivel del *namespace* "Data" 49
4.2.5 - Diagrama de clases de alto nivel del *namespace* "Views" 50
4.2.6 - Diagrama de secuencia "Añadir repositorios del primer servidor" 51
4.2.7 - Diagrama de secuencia "Cambiar el repositorio de visualización" 52
4.2.8 – Diagrama de secuencia "Visualizar el listado de ramas del repositorio" 53

4.2.9 - Diagrama de secuencia "Visualizar el Branch Explorer del repositorio" 54

4.2.10 - Diagrama de secuencia "Obtener el listado de diferencias en un punto" 55

Capítulo V – Implementación 56

5.1 - El Branch Explorer 56

5.1.1 - Código legado independiente de la plataforma 56

5.1.2 - El Branch Explorer en Android 57

5.1.3 - El Branch Explorer en iOS 85

5.2 - Diferencias de texto *side by side* 93

5.2.1 - Código legado independiente de la plataforma 93

5.2.2 - Dibujado de diferencias *side by side* en Android 94

Capítulo VI – Conclusiones 101

6.1 - Líneas de actuación futuras 102

Bibliografía 103

Anexo A: Decisión tecnológica de Xamarin respecto a un desarrollo nativo 107

Anexo B: *Plasticprotocol* 109

Capítulo I – Introducción

Al mercado del software de control de versiones existente en el año 2006, en el que se encontraban opciones ya veteranas por aquel entonces como Git, Subversion, o Perforce, se añade Plastic SCM. Inicialmente disponía únicamente de un cliente nativo para Windows, pero posteriormente se sumaron a su ecosistema otros productos como Gluon (un cliente de Plastic SCM "para artistas" debido a la facilidad con la que permite manejar formatos tales como imágenes o documentos), SemanticMerge (un software de apoyo a la tarea de *merge*, complementario tanto para Plastic SCM como para otros VCS), extensiones para los entornos de desarrollo más populares, e interfaces nativos para el resto de sistemas operativos mayoritarios, así como una interfaz Web para poder obtener métricas que permitiesen seguir fácilmente la evolución del repositorio.

Como tema central de este Trabajo de Fin de Grado se decidió explorar el mercado móvil. Gracias a *Xamarin.Android*, a *Xamarin.iOS*, y a la experiencia previa del equipo de Códice Software con Mono *Runtime*, se sabía que era posible adaptar gran parte de la base de código del cliente de Plastic SCM para que este pudiese ser ejecutado en dispositivos móviles.

De esta forma, se facilita así el desarrollo de partes tales como el Explorador de Ramas (*Branch Explorer*) y las diferencias lado a lado (*side by side*), siendo únicamente necesario desarrollar el código controlador y de interfaz de usuario, y proporcionando ya desarrolladas y su correcto funcionamiento asegurado con diversos *tests* partes fundamentales para la viabilidad del proyecto, tales como la comunicación con el servidor.

El objetivo principal es el de dotar a los clientes con un cliente móvil que facilite el seguimiento de la evolución de uno o más repositorios de código de Plastic SCM, haciendo para ello del Branch Explorer y de las diferencias *side by side* sus dos puntos principales, permitiendo también acceder a sus usuarios a la potencia del sistema de *queries* implementado tanto en los clientes de escritorio como en la interfaz de consola. Se busca cubrir aquellos huecos a los que un cliente de escritorio tradicional no puede llegar. Es decir, aquellas ocasiones en las que un desarrollador o un jefe de equipo necesita ver el contenido del repositorio cuando no dispone de un ordenador -por ejemplo, reuniones con el equipo, viajando, etc.- Se marca como requisito que dicho cliente sea lo más similar posible a las aplicaciones de escritorio, para que así la curva de aprendizaje del usuario sea mínima, adaptándose no obstante a las limitaciones de interacción propias que imponen la carencia de ratón y teclados físicos, así como una pantalla pequeña.

Se entiende que la aplicación móvil de Plastic SCM no va a ser, para un cliente, un punto determinante en la compra de una licencia del mismo, que es simplemente un complemento al software "de escritorio", no un sustituto. Se decide encarar este desarrollo como prueba de concepto, para valorar, al final del mismo, si merece la pena continuar su desarrollo.

En resumen, los objetivos son los siguientes:

- Desarrollar una aplicación móvil…
- …que permita desde una tableta revisar la evolución del repositorio…
- …haciendo del Branch Explorer y de las diferencias *side by side* sus dos puntos centrales…
- …con una interfaz de usuario familiar para los usuarios de los clientes de escritorio…
- …reutilizando la mayor cantidad de código posible de los clientes de escritorio ya existentes.

Capítulo II – Contexto tecnológico

En el contexto del desarrollo de software, el *software de control de versiones* (VCS por sus siglas en inglés) es la herramienta que permite a los desarrolladores mantener múltiples revisiones de un fichero, siendo una revisión el estado de dicho fichero en un punto concreto de tiempo.

2.1 - Software de control de versiones

Mantener múltiples revisiones de cada fichero que conforma un proyecto software ayuda no sólo a poder revertir el estado del mismo a un punto anterior -porque se haya introducido de manera involuntaria un fallo, porque un cliente necesite una versión anterior del proyecto desarrollado, etc.- sino que aporta gran flexibilidad a cómo dicho proyecto puede ser desarrollado, y permite revisar cómo ha cambiado el proyecto a lo largo del tiempo.

El ciclo de trabajo con un VCS es, a grandes rasgos, descargar el contenido del repositorio de código en un punto concreto del mismo (clonado, creando una *working copy*, o copia de trabajo del contenido en dicho punto), modificar una serie de ficheros según sea necesario, y guardar (*escribir*, *commit*, *checkin*...) dichos cambios de vuelta al repositorio. La flexibilidad que aporta el software de control de versiones al desarrollo es que estas operaciones pueden ser llevadas a cabo de forma simultánea por distintos desarrolladores sin que los cambios de ninguno afecten a los de los demás, y sin que se comprometa en ningún momento una versión del proyecto considerada como estable.

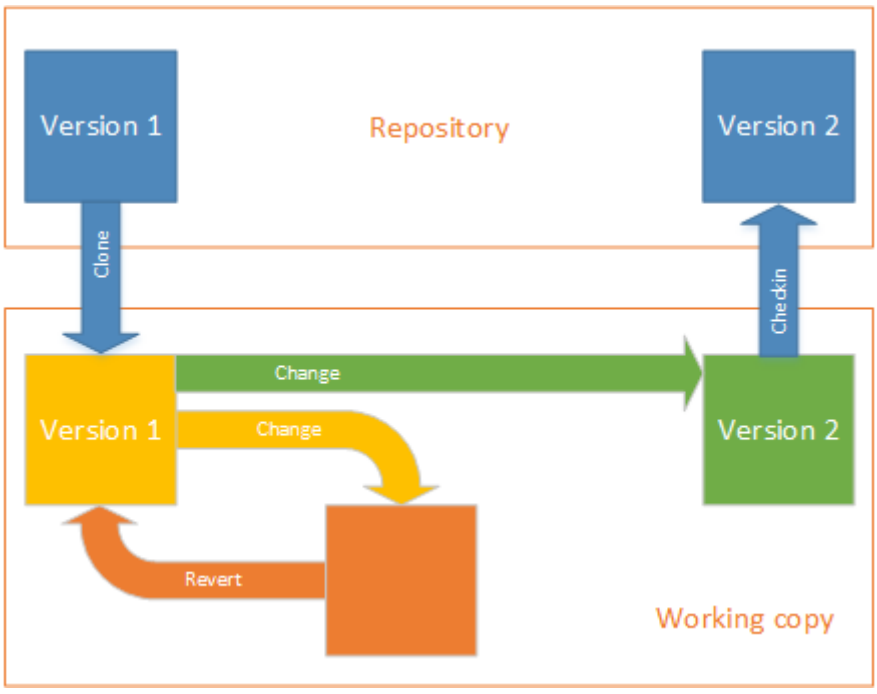

FIGURA 1. CICLO DE TRABAJO BÁSICO CON UN VCS

Esto es posible gracias a los distintos modelos de *branching* que se pueden aplicar a la hora de realizar el desarrollo apoyado por el SCM. Llegados a este punto, es necesario explicar qué elementos conforman la

historia de un repositorio. La terminología puede variar dependiendo de qué software se utilice, siendo la más común la que se usa a continuación:

- *Branches* o ramas (en Plastic SCM, *branches*): una rama es el duplicado del contenido del repositorio en un punto concreto del mismo, permitiendo que los cambios sucedan en esa rama en paralelo a los cambios de otras ramas sin que estos entren en conflicto. Salvo las ramas de primer nivel, toda rama tiene una rama madre.
- *Changesets, commits,* o cambios (en Plastic SCM, *changesets*): un *changeset* es la representación de un cambio incremental respecto al anterior en cuanto a contenido del repositorio se refiere. Los *changesets* se organizan en ramas, y son creados cada vez que un desarrollador graba en el repositorio los cambios hechos sobre su copia de trabajo.
- *Labels*, *tags,* o etiquetas (en Plastic SCM, *labels*): son marcas que se pueden realizar sobre un *changeset*, de forma que este queda identificado y diferenciado del resto a deseo del desarrollador. Normalmente identifican puntos estables y testados del proyecto que se esté desarrollando, como *milestones*[1] o *releases*[2] del mismo.
- *Workspace*, *working copy* o copia de trabajo (en Plastic SCM, *workspace*): el *workspace* es una copia del contenido del repositorio en un *changeset* concreto. Los desarrolladores hacen cambios sobre los ficheros de su *workspace*, pudiéndolos grabar de vuelta al repositorio si los consideran buenos -creando un *changeset*-, o pudiendo deshacerlos sin afectar a la historia del repositorio si no consideran que reúnen los requisitos de calidad necesarios. Mientras que el repositorio normalmente tendrá un tiempo de vida largo -abarcando toda la historia del proyecto, desde su primer cambio hasta la última *release*-, el *workspace* puede crearse y borrarse en cualquier momento según sea necesario.
- *Merge* o *integration* (en Plastic SCM, *merge*): es una de las operaciones más importantes que se pueden llevar a cabo en el software de control de versiones, sin la cual el uso del mismo carecería de sentido en la mayoría de las ocasiones. Consiste en reconciliar en un punto concreto de la historia del repositorio dos o más conjuntos de cambios: ya que las ramas permiten que varios desarrolladores trabajen en paralelo sobre el mismo proyecto, a no ser que los cambios en dichas ramas vayan a ser descartados -cosa poco común-, lo normal es querer re-integrarlos para crear una nueva versión del proyecto. El *merge* es la mezcla de dichos cambios, que puede ser realizada de forma automática si no existen conflictos entre las distintas versiones a ser mezcladas, o manual, si dichos conflictos existen y han de ser solucionados por una persona. El *merge* es una operación compleja en torno a la cual hay estudios verdaderamente detallados, e incluso patentes de metodologías y algoritmos de *merge* concretos.

[1] Una *milestone* marca, en un proyecto software, metas del mismo que se han de alcanzar para poder afirmar que el proyecto ha sido exitoso.

[2] Una *release* marca, en un proyecto software, la madurez del mismo. Ayudan a diferenciar cosas tales como la compatiblidad entre versiones, la funcionalidad añadida, o la estabilidad del mismo (*alpha, beta, release candidate…*)

2.2 – Plastic SCM

La primera versión de Plastic SCM es liberada a finales del año 2006 con el objetivo de competir con otros (D)VCS ya asentados en el mercado como Git, Mercurial, Perforce, o ClearCase, algunos de ellos con más de quince años de existencia en aquel momento.

2.2.1 - Branching y merging

Uno de los principales pilares de Plastic SCM a la hora de ser promocionado es la simplicidad de realizar operaciones de *branching* y de *merging*, es decir, la rapidez a la hora de crear ramas (*branching*) que apoyen metodologías de trabajo tales como *rama por tarea* -permitiendo trabajar de forma paralela a más de un desarrollador-, y la sencillez a la hora de reunificar los cambios de una rama en otra (*merging*). Para facilitar todavía más estas labores, en la versión 2.0 de Plastic SCM (publicada en el año 2008) hace su aparición el Explorador de Ramas o *Branch Explorer*, un diagrama en el que se muestra de un vistazo la evolución del repositorio a través de la evolución de sus elementos. La siguiente ilustración corresponde a la última revisión gráfica del Branch Explorer.

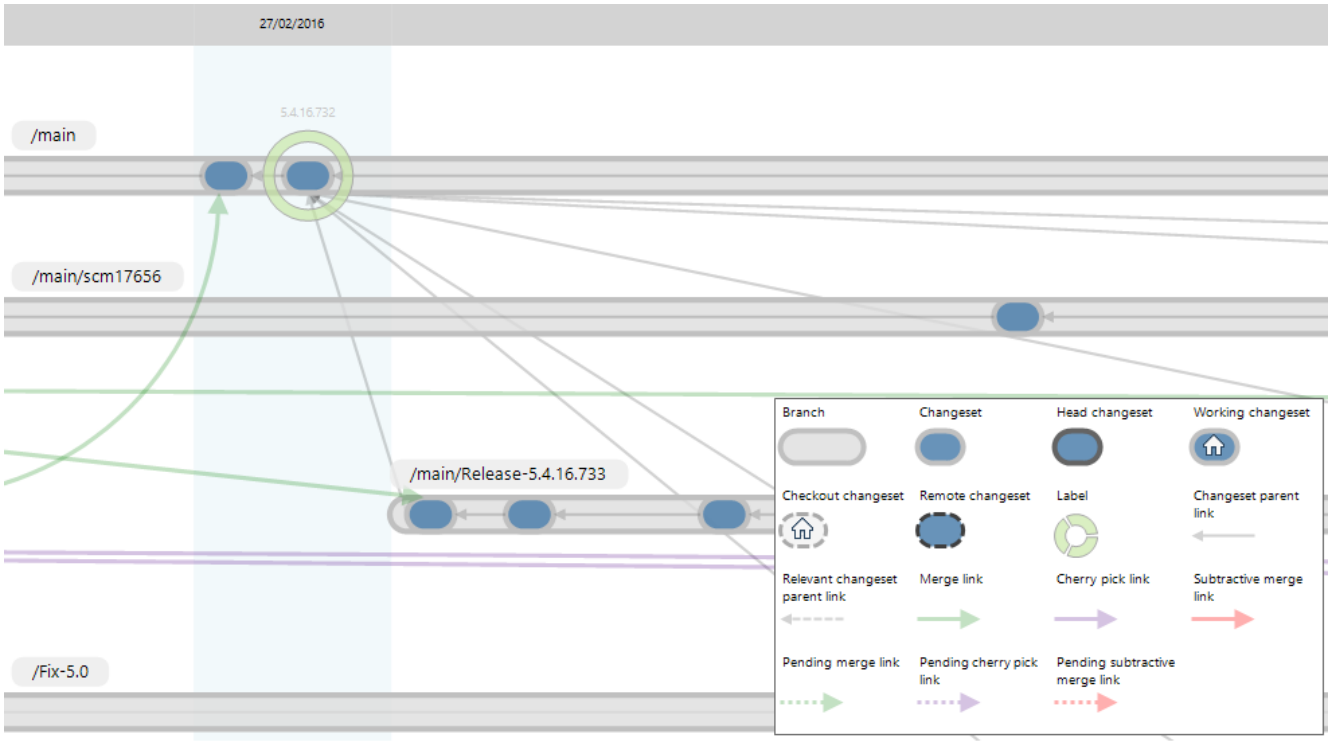

FIGURA 2. EJEMPLO DEL BRANCH EXPLORER

En la leyenda adjuntada en la *figura 2*, se puede observar cómo se representan los elementos mencionados al principio de esta sección. Aparecen algunas figuras cuyo significado no se han mencionado anteriormente: los *"mergelinks substractivos"* y los *"cherrypicks"*. Son formas de *merge* avanzadas, consistente el primero en un *"merge inverso"* (es decir, deshacer los cambios hechos en el *changeset* del que se esté haciendo *merge* substractivo) y el segundo en hacer *merge* únicamente de los cambios de un *changeset* concreto, sin tener en cuenta todos los anteriores al mismo hasta el ancestro común de los *changesets* implicados en la operación de *merge*. Para más información acerca de los mismos, se puede consultar la *Advanced Version Control – Pocket Guide* que Códice Software tiene a disposición *online* [1].

2.2.2 - Software de Control de Versiones Distribuido

Otra de las características de Plastic SCM que es necesario señalar es que se trata de un software de control de versiones *completamente distribuido,* significando esto que los desarrolladores pueden trabajar de forma autónoma sin necesidad de encontrarse en la misma red local, aunque Plastic SCM puede configurarse también para trabajar en modo *completamente centralizado*, o incluso en configuraciones intermedias que se adapten a las necesidades de cada equipo. Esto es posible gracias a la anatomía cliente-servidor de Plastic SCM.

El servidor es el lugar donde se almacenan los repositorios de código, y, por lo tanto, toda la información relativa a los cambios del mismo, tal como las ramas, los *changesets*, las etiquetas, los *mergelinks*... junto a la información de quién ha creado cada elemento, y en qué momento. Plastic SCM almacena dichos cambios en una base de datos, estando disponibles varios *backends* de BBDD para adaptarse a las necesidades de cada cliente (entre ellos, SQLite para servidores que solo vayan a ser usados por una persona o equipos pequeños, Microsoft SQL Server y MySQL para entornos donde el rendimiento es crítico, etc). De esta forma, junto con otras configuraciones adicionales avanzadas -como el tamaño de los *buffers* que se usan internamente-, el servidor de Plastic SCM puede escalar desde ser un proceso con una huella mínima de memoria en el ordenador portátil de un desarrollador independiente hasta ocupar varios gigabytes de RAM en el servidor dedicado de una empresa con varios cientos de desarrolladores.

El cliente es la herramienta que cada desarrollador usará para crear *workspaces* para un repositorio, haciendo *checkin* de sus cambios contra el servidor, deshaciendo los cambios de su *workspace*, visualizando diferencias entre cambios, etc. Se comunica con el servidor a través de una conexión TCP/IP (independientemente de que el servidor se encuentre en la misma máquina del cliente), permitiendo realizar diferentes combinaciones entre clientes y servidores según sean las necesidades de cada equipo.

Así, por ejemplo, si se quiere trabajar en modo completamente centralizado, cada desarrollador instalará en su máquina únicamente el cliente de Plastic SCM, encontrándose el servidor seguramente en una máquina dedicada dentro de la red local de la oficina.

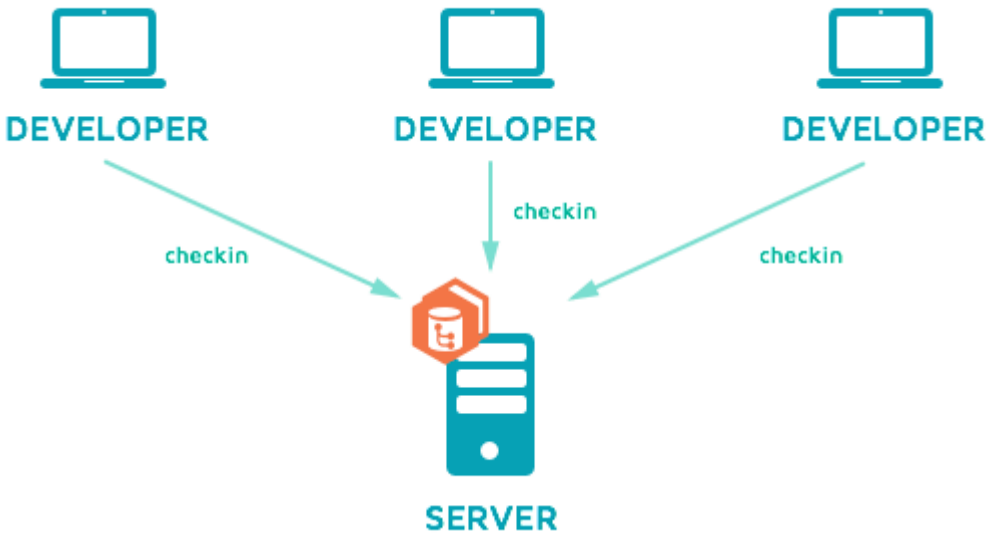

FIGURA 3. MODELO DE TRABAJO COMPLETAMENTE CENTRALIZADO

Si por el contrario se quiere trabajar en modo completamente distribuido, cada desarrollador dispondrá de un servidor de Plastic SCM en su máquina, sincronizando entre servidores los cambios en un modelo de red *peer-to-peer*, donde ningún servidor de repositorios será *autoritativo* sobre el resto de ninguna manera -por ejemplo, conteniendo lo que se considere la copia *canónica* de un repositorio-.

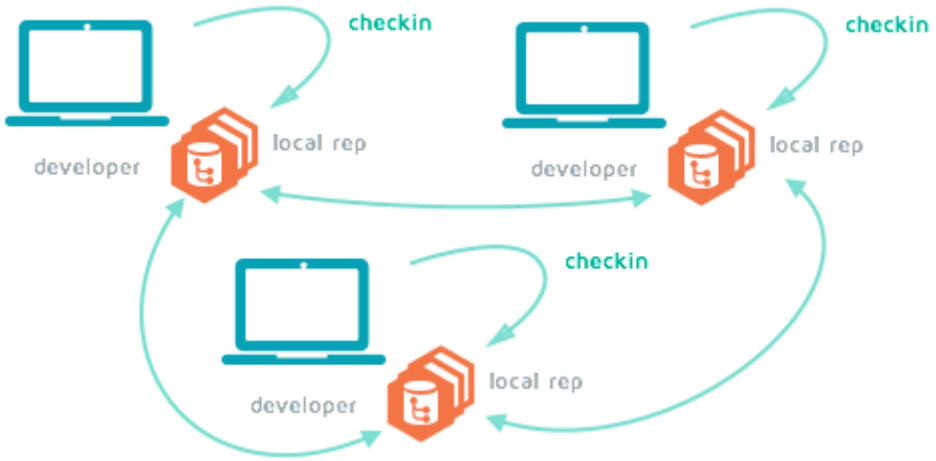

FIGURA 4. MODELO DE TRABAJO COMPLETAMENTE DISTRIBUIDO

Se puede configurar también un esquema intermedio, en el que existe un servidor central con absolutamente toda la historia de todos los repositorios con los que se trabaje en el equipo -las copias *canónicas* de los mismos-, y en el que cada desarrollador dispone de un servidor local en su máquina en el que únicamente existe una copia parcial (por ejemplo, solo una copia de la rama principal y aquellas ramas creadas por el desarrollador, pero no las creadas por sus compañeros) de los repositorios relevantes para cada individuo, con el fin de que estos puedan trabajar *offline* sólo con aquellos ficheros imprescindibles para sus tareas, ahorrando así tamaño en disco.

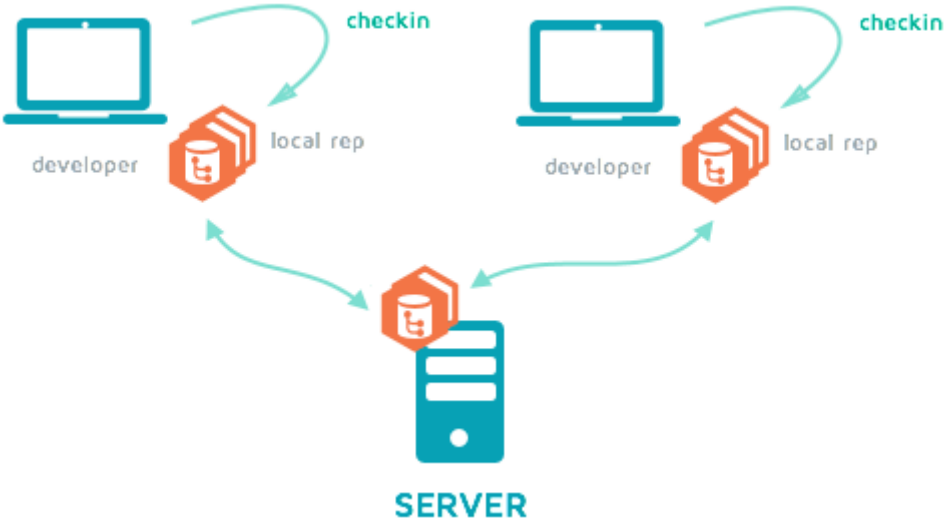

FIGURA 5. MODELO DE TRABAJO MIXTO

2.2.3 – Software de Control de Versiones multiplataforma

Al comienzo de su desarrollo, el equipo de Códice Software se centró en hacer de Plastic SCM una aplicación para el sistema operativo Windows. Así, se usó para su desarrollo C#, con WPF para la interfaz de usuario (Windows Presentation Foundation), y Visual Studio como IDE. Posteriormente, gracias a diversas herramientas como *ports* de WPF, y el *Mono Runtime*, se pudo ejecutar Plastic SCM en sistemas operativos GNU/Linux y Mac OS X (posteriormente OS X, actualmente macOS).

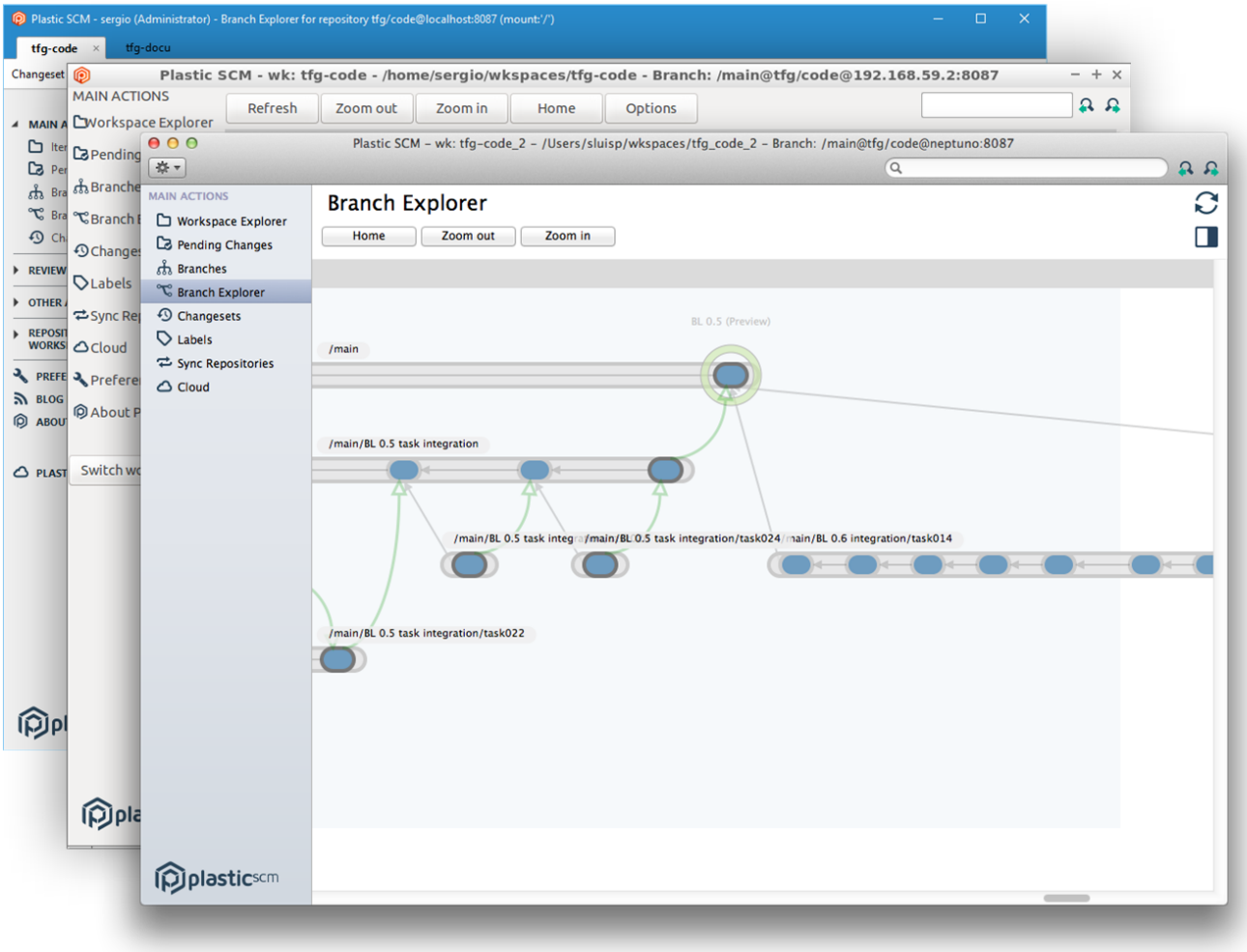

FIGURA 6. PLASTIC SCM EN WINDOWS, GNU/LINUX, Y MACOS

Sin embargo, una de las principales quejas de los clientes en estas plataformas era la interfaz de usuario. Escuchando dichas críticas, el equipo de Códice Software ha estado invirtiendo durante los últimos años un esfuerzo considerable en mantener tres interfaces de usuario nativas, en WPF/Winforms para Windows, GTK para GNU/Linux, y Cocoa para macOS. El esfuerzo es asumible gracias a los *bindings* de estos *frameworks* para C#, así como al *Mono Runtime* en las plataformas que no son Windows. Estas nuevas interfaces de usuario sirven además de base para el sistema interno de *testing* de interfaz de usuario, sustituyendo a la solución comercial *TestComplete* que se usaba para testear la interfaz de Windows, y simplificando la UI -tanto a nivel de código como de opciones de cara al usuario-, de forma que las interfaces para GNU/Linux y macOS avanzan en funcionalidad y apariencia a la par.

2.2.4 – Especializado en diferencias *side by side*

Tal y como se señaló al principio del documento, una de las funcionalidades que otorga cualquier software de control de versiones es la de poder visualizar cómo ha cambiado el contenido del repositorio de código a lo largo del tiempo. Si bien estas revisiones se pueden llevar a cabo por mera curiosidad, es de esperar, en un equipo de desarrolladores profesional, que todo el código que vaya a ser integrado en las versiones finales del producto cumplan ciertos estándares de calidad mínima y guías de estilo marcadas por el equipo.

Plastic SCM busca hacer de dichas revisiones de código una tarea sencilla, permitiendo obtener las diferencias entre dos puntos cualesquiera de la historia (*changesets*, ramas, *labels*) desde múltiples puntos de la aplicación de escritorio (siendo desde el Branch Explorer uno de los más usuales y sencillos).

A la hora de explorar las diferencias entre dos revisiones del mismo fichero podemos clasificar la forma de mostrar visualmente dichos cambios en dos categorías: el *diff unificado* y el *diff side by side*. El primero ha sido habitual, por su sencillez a la hora de ser representado en pantalla, en herramientas de terminal (como la utilidad `diff` incluida normalmente por defecto en sistemas `*NIX`).

```
Index: System.Data.SQLite/SQLite3.cs
===================================================================
--- System.Data.SQLite/SQLite3.cs
+++ System.Data.SQLite/SQLite3.cs
@@ -1779,11 +1779,11 @@
                     break;

                 }
             }
     }

-    internal override void Bind_Blob(SQLiteStatement stmt,
SQLiteConnectionFlags flags, int index, byte[] blobData)

+    internal override void Bind_Blob(SQLiteStatement stmt,
SQLiteConnectionFlags flags, int index, byte[] blobData, int bLength)

     {

         SQLiteStatementHandle handle = stmt._sqlite_stmt;
```

FIGURA 7. EJEMPLO PARCIAL DE DIFF UNIFICADO

Sin embargo, el segundo permite entender más fácilmente cómo ha cambiado el fichero, mostrando lado a lado ambas revisiones, y señalando en ellas mediante colores los puntos de cambio entre ambas.

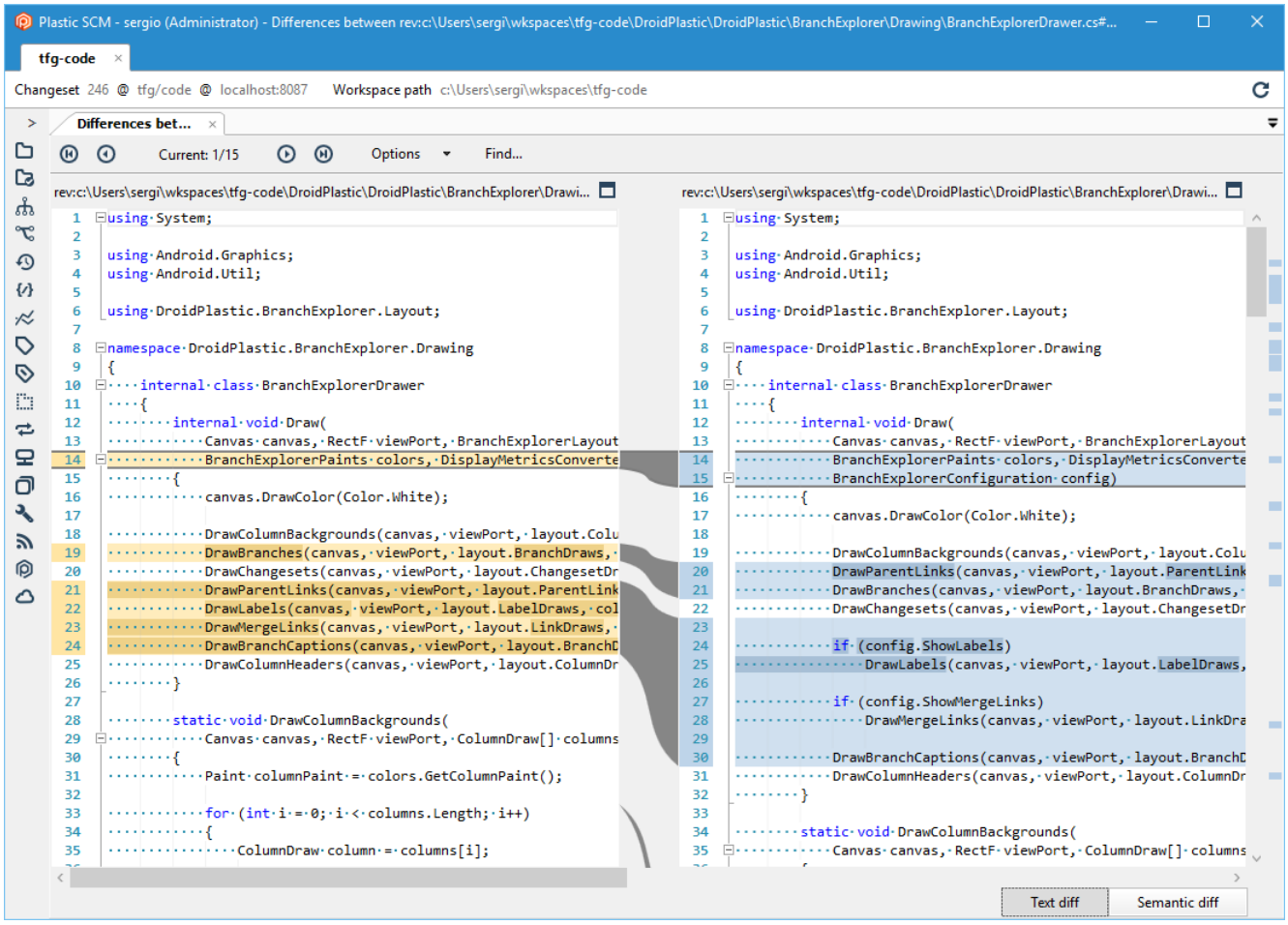

FIGURA 8. DIFERENCIAS DE TEXTO LADO A LADO

Esta segunda forma es mediante la cual Plastic SCM muestra las diferencias de texto, siendo el cálculo de las mismas otro de los puntos más importantes de Plastic SCM a la hora de ser promocionado como producto, poseyendo incluso dos de los miembros más veteranos del equipo una patente al respecto [2].

Capítulo III – Análisis de requisitos

Según *Agile Product Management with Scrum* [3], tener una visión general del producto a desarrollar es fundamental para conseguir desarrollarlo. La visión o borrador -*sketch*- del mismo ayuda a marcar objetivos, permitiendo seleccionar aquellos que son críticos. La visión del producto puede ir evolucionando, pero se considera lo suficientemente madura cuando permita responder, de forma concisa, una serie de preguntas sobre el mismo, tales como:

- **¿Quién va a comprar el producto?** Los usuarios de un cliente móvil de Plastic SCM serán aquellos que ya hayan comprado una licencia de Plastic SCM, o aquellos a los que se les haya concedido una licencia por cualquier otra circunstancia -se usa el producto en su lugar de trabajo, desarrollan software para una organización sin ánimo de lucro…-.
- **¿Quién es el usuario objetivo?** El usuario objetivo será un desarrollador de software o un *manager* de equipo que tenga bajo su dirección desarrolladores de software.
- **¿Qué necesidades va a satisfacer el producto?** Si bien la aplicación, al menos en sus primeras versiones, no va a permitir desarrollar software desde una tableta o teléfono -tampoco existen entornos de desarrollo en estas plataformas-, sí va a permitir revisar cambios sin la necesidad de disponer del equipo de trabajo principal.
- **¿Qué valor añadido proporciona el producto?** Permite revisar cambios -tanto de estructura como contenido del repositorio- en un dispositivo portátil como es una tableta, sin la necesidad de usar un ordenador -que es más difícil de transportar- ni de instalar un cliente completo.
- **¿Qué funcionalidades son críticas para satisfacer las necesidades seleccionadas, y, por lo tanto, para el éxito del producto?** El producto deberá permitir revisar cambios tanto de estructura del repositorio como de ficheros concretos, permitiendo a sus usuarios responder a preguntas tales como *qué cambios se han hecho entre un determinado changeset y el anterior*, o *qué cambios se han integrado en una determinada rama*.
- **¿Cómo se compara el producto respecto a otros existentes, tanto propios como de la competencia?** El producto propio equivalente es el cliente de Plastic SCM en sus versiones para Windows, GNU/Linux y macOS. Hay muy pocas diferencias entre los tres, con distribución similar de secciones y opciones, para que un usuario que utilice Plastic SCM en una o más plataformas no deba enfrentarse a una curva de aprendizaje pronunciada para manejar el software con soltura. El objetivo es que el cliente móvil sea lo más parecido a los clientes de escritorio, con las adaptaciones necesarias a una interfaz táctil, para que la curva de aprendizaje sea, igualmente, lo más pequeña posible. No se busca rivalizar con otros clientes móviles de software de control de versiones, pues no existe ninguno compatible con Plastic SCM, y se entiende que la aplicación móvil no va a ser un punto decisivo de venta de licencias, únicamente un añadido.
- **¿Es el producto viable? ¿Puede ser desarrollado?** La experiencia del equipo en el desarrollo de Plastic SCM para tres sistemas operativos distintos usando de base la tecnología de Xamarin, así como la extensa base de código desarrollada independiente del framework de interfaz de usuario garantizan que sí. Dicho desarrollo es posible, y la potencial reutilización de código reduce además la complejidad del mismo.

Así pues, una vez se ha establecido una visión general del producto, se puede comenzar con el análisis de requisitos del mismo.

3.1 - User Stories

El equipo de desarrollo de Plastic SCM sigue la metodología ágil *Scrum*. En dicha metodología es habitual elaborar *User Stories,* o historias de usuario. Mike Cohn [4] las define como:

> Descripciones cortas y simples de una funcionalidad, narradas desde el punto de vista de aquel que desea dicha funcionalidad -habitualmente un usuario-, y que siguen el esquema *"Como <tipo de usuario>, quiero <una funcionalidad> para <alcanzar una meta>".*

Dichas historias de usuario permiten dividir la funcionalidad a desarrollar en tareas, así como priorizar unas funcionalidades sobre otras, y elaborar estimaciones del tiempo de desarrollo que cada funcionalidad puede consumir. Así pues, durante la fase de análisis del cliente móvil de Plastic SCM se elaboraron *User Stories*, permitiendo despejar de manera temprana dudas sobre los límites de la funcionalidad del mismo.

Se identificaron, en una primera revisión, las siguientes:

- *Add repositories from a server* – Añadir repositorios de un servidor.
    - *As a **user**, I want to add multiple repositories to the app regardless of the server they're in, so that I can have available all the repositories I need.*
    - Como **usuario**, quiero añadir a la aplicación múltiples repositorios independientemente del servidor en el que se encuentren, para tener a mi disposición todos los repositorios que necesite.
- *Select specific repositories from a server* – Seleccionar repositorios específicos de un servidor.
    - *As a **user**, I want to choose which repositories from a server will be available in the app, so that I can focus on the ones that matter to me.*
    - Como **usuario**, quiero escoger qué repositorios de un servidor estarán disponibles en la aplicación, para poder centrarme en aquellos que sean relevantes para mí.
- *Add new repositories* – Añadir nuevos repositorios.
    - *As a **user**, I want to add new repositories to the app at any time, so that I can have available all the repositories I need.*
    - Como **usuario**, quiero añadir nuevos repositorios a la aplicación en cualquier momento, para tener a mi disposición todos los repositorios que necesite.
- *Update access credentials* – Actualizar las credenciales de acceso.
    - *As a **user**, I want to update my credentials for a given repository at any time, so that I can still connect to it even if my security configuration changes.*
    - Como **usuario**, quiero poder actualizar mis credenciales para un repositorio concreto en cualquier momento, para poder conectarme a los repositorios que contenga, aunque mi configuración de seguridad en el servidor cambie.
- *Delete repositories from the app* – Eliminar repositorios de la aplicación.
    - *As a **user**, I want to delete from the app a given repository at any time, so that I can focus only on the relevant ones to me.*
    - Como **usuario**, quiero eliminar de la aplicación un repositorio en cualquier momento, para poder centrarme únicamente en los que sean relevantes para mí.
- *Switch between repositories* – Cambiar entre repositorios.

    - *As a **user**, I want to switch between repositories easily, so that I can quickly access the information that is relevant to me.*
    - Como **usuario,** quiero cambiar fácilmente entre repositorios**,** para poder acceder rápidamente a la información que sea relevante para mí.
- *Browse repository history* – Explorar la historia del repositorio.
    - *As a **user**, I want to browse the repository history -branches, labels, changesets, mergelinks- on the tablet, so that I can understand how it evolved.*
    - Como **usuario,** quiero explorar la historia -ramas**,** *changesets***,** etiquetas**,** *mergelinks*- del repositorio desde la tableta**,** para poder entender cómo éste ha evolucionado.
- *Browse repository content* – Explorar el contenido del repositorio.
    - *As a **user**, I want to list the repository content on a given point, so that I can understand its structure.*
    - Como **usuario,** quiero listar el contenido del repositorio en cualquier punto**,** para poder entender su estructura.
- *Diff repository points* – Listar diferencias entre puntos.
    - *As a **user**, I want to diff two objects -changesets, branches-, so that I can easily understand the files changed, moved, added, and deleted, between two points.*
    - Como **usuario,** quiero poder listar diferencias entre dos objetos -*changesets*, ramas-, para poder entender qué ficheros cambiaron**,** fueron movidos**,** fueron añadidos**,** o fueron borrados**,** entre dos puntos.
- *Find objects in the Branch Explorer* – Encontrar objetos en el Branch Explorer.
    - *As a **user**, I want to quickly find objects in the Branch Explorer, so that I can easily access the information that is relevant to me.*
    - Como **usuario,** quiero encontrar rápidamente objetos en el *Branch Explorer***,** para poder acceder rápidamente a la información relevante para mí.
- *Filter Branch Explorer's displayed objects* – Filtrar los objetos mostrados en el *Branch Explorer*.
    - *As a **user**, I want to filter the information displayed in the Branch Explorer, so that I can focus on the information that is relevant to me.*
    - Como **usuario,** quiero filtrar la información mostrada en el *Branch Explorer***,** para poder enfocarme en la información relevante para mí.
- *Browse file history* – Explorar el historial de un fichero.
    - *As a **user**, I want to see the history of a file, so that I can understand when was it changed and by who.*
    - Como **usuario,** quiero ver el historial de un determinado fichero**,** para poder entender cuándo fue éste cambiado**,** y por quién.
- *Diff two revisions of a file* – Ver las diferencias entre dos revisiones de un fichero.
    - *As a **user**, I want to diff two revisions of a file, so that I can understand the changes on a given file between two points.*
    - Como **usuario,** quiero ver las diferencias entre dos revisiones de un fichero**,** para poder entender las modificaciones sobre ese fichero entre dos puntos.
- *Execute advanced queries* – Ejecutar búsquedas avanzadas.
    - *As a **user**, I want to search for objects -changesets, branches, labels- using advanced queries, so that I can easily find the information that is relevant to me.*

    - Como **usuario**, quiero buscar objetos -*changesets*, ramas, etiquetas- utilizando términos de búsqueda avanzados, para poder encontrar fácilmente la información relevante para mí.
- *Order query results by some given criteria* – Ordenar resultados de búsqueda según criterio.
    - *As a **user**, I want to order the information displayed following different criteria, so that I can easily find the information that is relevant to me.*
    - Como **usuario**, quiero ordenar la información visible siguiendo diferentes criterios, para poder encontrar fácilmente la información relevante para mí.
- *Download files from a repository* – Descargar ficheros de un repositorio.
    - *As a **user**, I want to download files from a repository, so that I can open, visualize, and modify them in the application of my choice.*
    - Como **usuario**, quiero descargar ficheros de un repositorio, para poder abrirlos, visualizarlos, y modificarlos en otras aplicaciones.

Estas *user stories* fueron filtradas en aquellas que se consideró que debían formar parte del *mínimo producto viable,* aquel con las funcionalidades esenciales para satisfacer los puntos marcados en la visión global del producto, pero sin funcionalidades avanzadas o secundarias. Por ejemplo, a pesar de que nunca se llegó a contemplar poder hacer desde el cliente móvil ciclos de *checkout-checkin* (descargar del servidor el contenido completo de un repositorio en un punto concreto, modificar uno o más ficheros, y subir de nuevo los cambios al servidor, creando, por lo tanto, un nuevo *changeset*), sí se consideró la posibilidad de descargar ficheros individuales de revisiones concretas, para que el usuario pudiese guardar dicho fichero en un directorio de su dispositivo y, posteriormente, abrirlo con una aplicación compatible -por ejemplo, ficheros de PDF, documentos de Microsoft Office, etc-. Las *user stories* seleccionadas para ser desarrolladas fueron las siguientes:

- Añadir repositorios de un servidor.
- Seleccionar repositorios específicos de un servidor.
- Cambiar entre repositorios.
- Explorar la historia del repositorio.
- Listar diferencias entre puntos.
- Filtrar los objetos mostrados en el *Branch Explorer*.
- Ver las diferencias entre dos revisiones de un fichero.
- Ejecutar búsquedas avanzadas.

Elaborar estas historias de usuario fue posible sin consultar a clientes externos de Plastic SCM porque el propio producto es una herramienta más en el desarrollo del mismo. Todos los trabajadores de Códice Software, incluidos perfiles menos técnicos como el equipo de diseño gráfico, utilizan Plastic SCM como parte de sus herramientas cotidianas.

3.2 - Casos de uso identificados

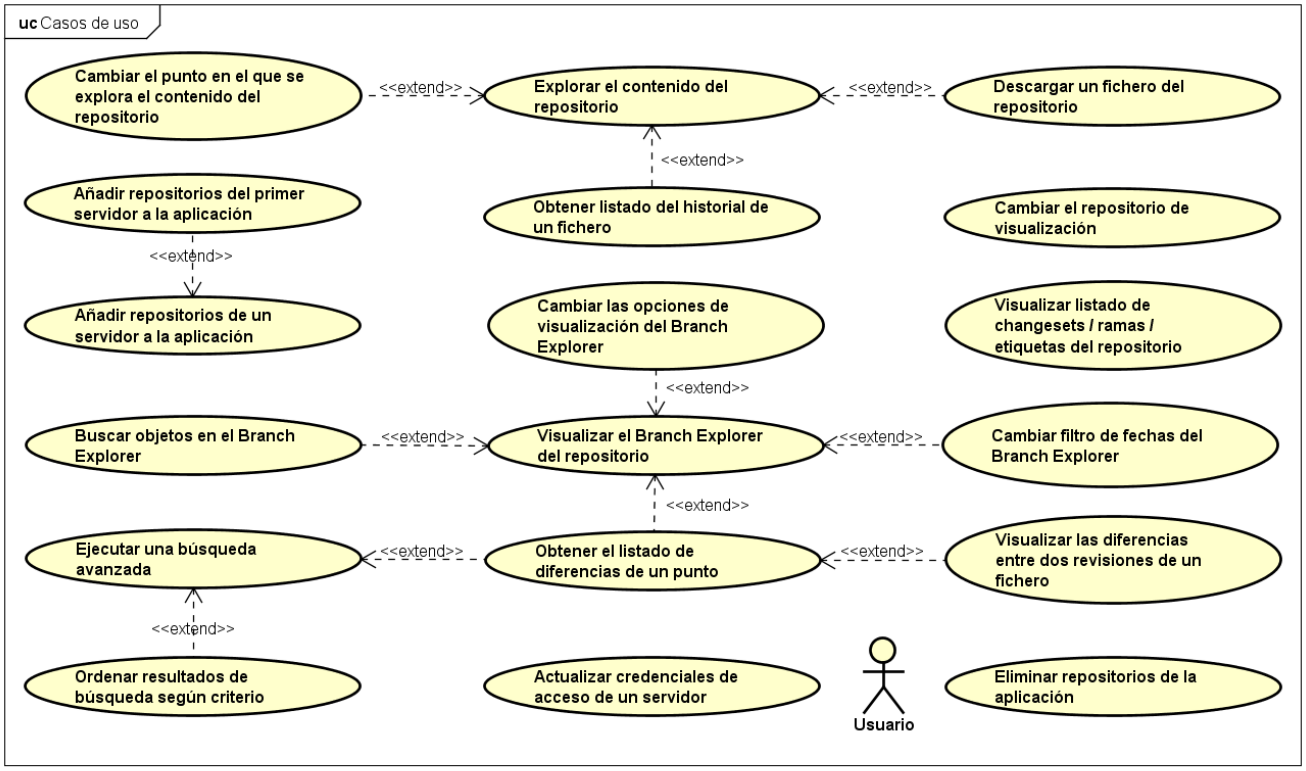

FIGURA 9. DIAGRAMA DE CASOS DE USO IDENTIFICADOS

3.2.1 - Añadir repositorios del primer servidor a la aplicación

Actor principal	Usuario
Precondiciones	El usuario dispone de un servidor de Plastic SCM con una versión compatible con la aplicación móvil.
Postcondiciones	El usuario habrá añadido a la aplicación uno o más repositorios de un servidor de Plastic SCM.
Escenario principal	1. El usuario abre por primera vez la aplicación móvil de Plastic SCM. 2. La aplicación muestra un formulario con los campos necesarios para que el usuario introduzca datos de conexión a un servidor de Plastic SCM. 3. El usuario introduce una dirección de host en formato *[dirección]:[puerto]*, un nombre de usuario, y, opcionalmente, una contraseña. 4. La aplicación habilita el botón de conexión. 5. El usuario pulsa el botón de conexión. 6. La aplicación muestra un diálogo, bloqueando la interfaz, con el acuerdo de licencia. 7. El usuario escoge la opción correspondiente a aceptar el acuerdo de licencia. 8. La aplicación obtiene del servidor el modo de autenticación, comprueba que las credenciales son correctas, y recupera de él la lista de repositorios, mostrándosela al usuario. 9. El usuario escoge uno o más repositorios de la lista mostrada. Escoge, a continuación, la opción correspondiente a "Añadir". 10. La aplicación guarda referencias a los repositorios seleccionados, junto a los datos de autenticación del usuario. A continuación, la aplicación muestra la

	pantalla principal, con el contenido del primer repositorio seleccionado, una vez ordenados por orden alfabético.
Flujos alternativos	5. Si el servidor especificado no se encuentra en un formato válido de *[dirección]:[puerto]*, la aplicación muestra un mensaje de error. A continuación, este caso de uso continúa en el paso 3. 7. Si el usuario escoge la opción correspondiente a rechazar el acuerdo de licencia, la aplicación no realiza ninguna acción. A continuación, este caso de uso queda sin efecto. 8. Si el servidor está configurado de forma que se necesita una contraseña de usuario, y este no la ha proporcionado, la aplicación muestra un mensaje de error. A continuación, este caso de uso continua en el paso 3. 8. Si los datos de autenticación proporcionados por el usuario no son correctos, la aplicación muestra un mensaje de error. A continuación, este caso de uso continua en el paso 3. 8. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso continua en el paso 5. 9. Si el usuario escoge la opción correspondiente a "cancelar", la aplicación esconde la lista de repositorios. A continuación, este caso de uso queda sin efecto. 9. Si el usuario no escoge ningún repositorio antes de escoger la opción correspondiente a "añadir", la aplicación no realiza ninguna acción. A continuación, este caso de uso continúa en el paso 9.

3.2.2 - Añadir repositorios de un servidor a la aplicación

Actor principal	Usuario
Precondiciones	El usuario dispone de un servidor de Plastic SCM con una versión compatible con la aplicación móvil.
Postcondiciones	El usuario habrá añadido a la aplicación uno o más repositorios de un servidor de Plastic SCM.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a añadir nuevos repositorios de un servidor. 2. La aplicación muestra un formulario con los campos necesarios para que el usuario introduzca datos de conexión a un servidor de Plastic SCM. 3. El usuario introduce una dirección de host en formato [dirección]:[puerto], un nombre de usuario, y, opcionalmente, una contraseña. 4. La aplicación habilita el botón de conexión. 5. El usuario pulsa el botón de conexión. 6. La aplicación obtiene del servidor el modo de autenticación, comprueba que las credenciales proporcionadas son correctas, y recupera de él la lista de repositorios, mostrándosela al usuario. 7. El usuario escoge uno o más repositorios de la lista mostrada. Escoge, a continuación, la opción correspondiente a "Añadir". 8. La aplicación guarda referencias a los repositorios seleccionados, junto a los datos de autenticación del usuario. A continuación, la aplicación regresa a la vista en la que se encontraba antes de iniciar el caso de uso.
Flujos alternativos	5. Si el servidor especificado no se encuentra en un formato válido de *[dirección]:[puerto]*, la aplicación muestra un mensaje de error. A continuación, este caso de uso continúa en el paso 3.

	6. Si el servidor está configurado de forma que se necesita una contraseña de usuario, y este no la ha proporcionado, la aplicación muestra un mensaje de error. A continuación, este caso de uso continua en el paso 3. 6. Si los datos de autenticación proporcionados por el usuario no son correctos, la aplicación muestra un mensaje de error. A continuación, este caso de so continua en el paso 3. 6. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso continúa en el paso 5. 7. Si el usuario escoge la opción correspondiente a "cancelar", la aplicación esconde la lista de repositorios. A continuación, este caso de uso queda sin efecto. 7. Si el usuario no escoge ningún repositorio antes de pulsar en la opción correspondiente a "añadir", la aplicación no realiza ninguna acción. A continuación, este caso de uso continua en el paso 7.

3.2.3 - Eliminar repositorios de la aplicación

Actor principal	Usuario
Precondiciones	El usuario dispone de dos o más repositorios añadidos a la aplicación.
Postcondiciones	El usuario habrá eliminado de la aplicación uno o más repositorios.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a visualizar el listado de repositorios añadidos a la aplicación. 2. La aplicación muestra el listado de repositorios añadidos a la aplicación, agrupados por el servidor al que pertenecen. 3. El usuario escoge, de entre sus opciones de menú, la correspondiente a eliminar de la aplicación el repositorio que desee. 4. La aplicación elimina, de entre las referencias a repositorios guardadas, la correspondiente al repositorio escogido por el usuario. A continuación, actualiza el listado de repositorios que se está mostrando al usuario.
Flujos alternativos	4. Si la aplicación únicamente tiene guardadas referencias al repositorio que ha escogido eliminar el usuario, la aplicación muestra un mensaje de error indicando que no se pueden eliminar todos los repositorios. A continuación, este caso de uso queda sin efecto.

3.2.4 - Actualizar credenciales de acceso a un servidor

Actor principal	Usuario
Precondiciones	El usuario dispone de uno o más servidores añadidos a la aplicación.
Postcondiciones	El usuario habrá actualizado sus credenciales para que la aplicación móvil pueda conectarse a un servidor de Plastic SCM determinado.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a visualizar el listado de repositorios añadidos a la aplicación. 2. La aplicación muestra el listado de repositorios añadidos a la aplicación, agrupados por el servidor al que pertenecen. 3. El usuario escoge, de entre sus opciones de menú, la correspondiente actualizar las credenciales de un servidor. 4. La aplicación muestra al usuario un formulario con los campos correspondientes a nombre de usuario (rellenado) y contraseña (en blanco) para dicho servidor.

	5. El usuario actualiza el contenido del campo correspondiente al nombre de usuario y/o del campo correspondiente a la contraseña. A continuación, pulsa en la opción correspondiente a "Aceptar". 6. La aplicación obtiene del servidor el modo de autenticación y comprueba que las credenciales son correctas. Actualiza, entre sus referencias a servidores, los datos de autenticación proporcionados.
Flujos alternativos	5. Si el usuario pulsa la opción correspondiente a "Cancelar", la aplicación no realiza ninguna acción. A continuación, este caso de uso queda sin efecto. 6. Si los datos de autenticación no son correctos, la aplicación muestra un mensaje de error. A continuación, este caso de uso continua en el paso 5. 6. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.5 - Cambiar el repositorio de visualización

Actor principal	Usuario
Precondiciones	El usuario dispone de dos o más repositorios añadidos a la aplicación.
Postcondiciones	La aplicación habrá actualizado el repositorio actual de navegación, resolviendo contra el repositorio escogido cualquier operación posterior en la que intervenga comunicación con un servidor.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a visualizar el listado de repositorios añadidos a la aplicación. 2. La aplicación muestra el listado de repositorios añadidos a la aplicación, agrupados por el servidor al que pertenecen. 3. El usuario escoge, de entre sus opciones de menú, la correspondiente a cambiar el repositorio de navegación actual por aquel que desee. 4. La aplicación actualiza la dirección de repositorio y las credenciales de usuario que utilizará para obtener datos del servidor. A continuación, ocultará el listado de repositorios y actualizará el contenido de la vista actual con los datos relevantes del repositorio escogido.
Flujos alternativos	3. Si el usuario escoge, de entre sus opciones de menú, la correspondiente a cancelar, la aplicación esconde el listado de repositorios. A continuación, este caso de uso queda sin efecto. 4. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.6 - Visualizar el listado de changesets / ramas / etiquetas del repositorio

Actor principal	Usuario
Precondiciones	El usuario dispone de uno o más repositorios añadidos a la aplicación.
Postcondiciones	El usuario habrá navegado a aquella vista de la aplicación que contenga el listado relevante según la opción.
Escenario principal	1. El usuario escoge, de entre sus opciones de navegación, la correspondiente a navegar a la vista de listados que desee. 2. La aplicación abre la vista escogida por el usuario. La aplicación recupera, de entre la configuración de usuario, la consulta específica para la vista seleccionada, y la ejecuta contra el repositorio actual.
Flujos alternativos	2. Si no existe una *consulta* guardada para la vista y repositorio seleccionados, genera una consulta por defecto. A continuación, este caso de uso continua en el paso 2.

	2. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.7 - Visualizar el Branch Explorer de un repositorio

Actor principal	Usuario
Precondiciones	El usuario dispone de uno o más repositorios añadidos a la aplicación.
Postcondiciones	El usuario podrá visualizar el Branch Explorer del repositorio.
Escenario principal	1. El usuario escoge, de entre sus opciones de navegación, la correspondiente al Branch Explorer. 2. La aplicación abre la vista del Branch Explorer. Si no existen datos previos para mostrar, recupera de entre la configuración guardada el filtro de fechas correspondiente, y se comunica con el servidor para descargar dichos datos. Si existen datos previos para mostrar, los restaura. Recupera de entre la configuración guardada las opciones de visualización, y dibuja el Branch Explorer aplicando dichas opciones.
Flujos alternativos	2. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.8 - Cambiar las opciones de visualización del Branch Explorer

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar el Branch Explorer de un repositorio"
Postcondiciones	El usuario habrá cambiado los objetos que se muestran en el Branch Explorer.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a cambiar los objetos que se muestran en el Branch Explorer. 2. La aplicación muestra un menú con los objetos que se muestran en el Branch Explorer, acompañados de una marca que indique si se están mostrando o no actualmente. 3. El usuario cambia las opciones de visualización marcando y desmarcando elementos. A continuación, pulsa en la opción correspondiente a "Aceptar". 4. La aplicación guarda las nuevas opciones de visualización del Branch Explorer para el repositorio actual. A continuación, redibuja el Branch Explorer, aplicando la nueva configuración.
Flujos alternativos	3. Si el usuario escoge la opción correspondiente a "Cancelar", la aplicación esconde el menú de configuración. A continuación, este caso de uso queda sin efecto.

3.2.9 - Cambiar el filtro de fechas del Branch Explorer

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar el Branch Explorer de un repositorio"
Postcondiciones	El Branch Explorer que visualizará el usuario contiene únicamente los cambios realizados entre las fechas especificadas por el usuario.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a cambiar el filtro de fechas del Branch Explorer.

	2. La aplicación muestra un menú con la fecha de inicio y la fecha de fin de los datos cargados en el Branch Explorer, con la fecha de fin desactivada si se deben descargar datos creados hasta el momento de la consulta. 3. El usuario modifica la fecha de inicio del filtro y/o la fecha de fin. A continuación, escoge la acción correspondiente a "Aceptar". 4. La aplicación guarda el nuevo filtro de fechas. A continuación, descarga del servidor los datos del Branch Explorer con el filtro de fechas seleccionado, y redibuja el diagrama.
Flujos alternativos	3. Si el usuario escoge la acción correspondiente a "Cancelar", la aplicación esconde el menú de configuración de fechas. A continuación, este caso de uso queda sin efecto. 4. Si la fecha de inicio introducida por el usuario es posterior a la fecha de fin, la aplicación intercambia las fechas de inicio y de fin. A continuación, este caso de uso continua en el paso 4. 4. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.10 - Obtener el listado de diferencias de un punto

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar el Branch Explorer de un repositorio", o bien ha completado el caso de uso "Visualizar listado de changesets / ramas / etiquetas del repositorio".
Postcondiciones	El usuario visualizará un listado con las diferencias ocurridas en un punto de la historia del repositorio respecto al punto inmediatamente anterior.
Escenario principal	1. El usuario realiza una pulsación larga sobre un *changeset* o una rama para obtener su menú contextual. 2. La aplicación muestra el menú contextual del changeset o la rama. 3. El usuario escoge, de entre sus opciones de menú, la correspondiente a calcular las diferencias del punto. 4. La aplicación descargar del servidor el listado de diferencias del punto, y se lo muestra al usuario, agrupando diferencias por modificados, añadidos, movidos, y borrados.
Flujos alternativos	3. Si el usuario escoge la opción correspondiente a "Cancelar", la aplicación oculta el menú contextual del objeto. A continuación, este caso de uso queda sin efecto. 4. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.11 - Visualizar las diferencias entre dos revisiones de un fichero

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Obtener el listado de diferencias de un punto".
Postcondiciones	El usuario visualizará las diferencias de un fichero entre una revisión y la anterior, o bien el fichero si no existe revisión previa con la que comparar.
Escenario principal	1. El usuario escoge, de entre el listado de diferencias mostrado por la aplicación, un fichero.

	2. La aplicación descarga del servidor las revisiones relevantes para realizar la comparación. A continuación, muestra las diferencias del fichero *side by side*, o únicamente el contenido del fichero si no existe una revisión previa.
Flujos alternativos	1. Si el usuario escoge un fichero que no es de texto, y que, por lo tanto, no se puede visualizar en la aplicación, la aplicación no realiza ninguna acción. A continuación, este caso de uso continua en el paso 1. 2. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.12 - Buscar objetos en el Branch Explorer

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar el Branch Explorer de un repositorio".
Postcondiciones	El Branch Explorer se encuentra centrado en el resultado de la búsqueda del usuario.
Escenario principal	1. El usuario rellena el campo de búsqueda correspondiente a buscar un objeto en el Branch Explorer. A continuación, escoge la acción correspondiente a "ejecutar búsqueda". 2. La aplicación busca un objeto cuyo nombre, GUID, o propietario, coincidan o contengan el término de búsqueda. A continuación, marca dichos objetos como "iluminados", redibuja el diagrama, y lo centra en el primer objeto encontrado.
Flujos alternativos	1. Si el usuario cambia el foco del campo de búsqueda sin escoger la acción correspondiente a "ejecutar búsqueda", la aplicación no realiza ninguna acción. A continuación, este caso de uso queda sin efecto. 2. Si la aplicación no encuentra ningún objeto que pueda ser filtrado por el término de búsqueda, muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.13 - Ejecutar una búsqueda avanzada

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar listado de changesets / ramas / etiquetas del repositorio".
Postcondiciones	El usuario visualizará aquellos objetos del repositorio que se ajusten a sus términos de búsqueda.
Escenario principal	1. El usuario modifica la parte personalizable de la *query* de la vista de listado de changesets / ramas / etiquetas de repositorio. A continuación, escoge la acción correspondiente a "ejecutar búsqueda". 2. La aplicación resuelve los campos de búsqueda de la query, ejecuta la query contra el servidor, y muestra en el listado los objetos recuperados. Finalmente, guarda la query en las preferencias del usuario para el listado y repositorios correspondiente.
Flujos alternativos	1. Si el usuario cambia el foco del campo de búsqueda sin escoger la acción correspondiente a "ejecutar búsqueda", la aplicación no realiza ninguna acción. A continuación, este caso de uso queda sin efecto. 2. Si la aplicación no puede resolver algún campo de la consulta porque la semántica de la misma sea incorrecta, muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

	2. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.14 - Ordenar resultados de búsqueda según criterio

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Visualizar listado de changesets / ramas / etiquetas del repositorio", o bien ha completado el caso de uso "Ejecutar búsqueda avanzada".
Postcondiciones	El usuario visualizará aquellos objetos del repositorio que se ajusten a sus términos de búsqueda, ordenados según el criterio escogido.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a ordenar el listado de resultados de búsqueda. 2. La aplicación muestra un menú con las posibles claves mediante las que ordenar el listado, estando seleccionada la opción actual, y permitiendo escoger una única opción simultáneamente. 3. El usuario escoge, de entre las claves disponibles, aquella que desee. Escoge, a continuación, la opción correspondiente a "Aceptar". 4. La aplicación reordena la lista de resultados de búsqueda en función de la clave escogida por el usuario.
Flujos alternativos	3. Si el usuario escoge la opción correspondiente a "cancelar", la aplicación no realiza ninguna opción. A continuación, el caso de uso queda sin efecto.

3.2.15 - Explorar el contenido del repositorio

Actor principal	Usuario
Precondiciones	El usuario dispone de uno o más repositorios añadidos a la aplicación.
Postcondiciones	El usuario podrá visualizar el contenido del repositorio en una posición determinada.
Escenario principal	1. El usuario escoge, de entre sus opciones de navegación, la correspondiente a navegar a la vista de ítems del repositorio. 2. La aplicación recupera, de entre la configuración de usuario, la posición en la que la vista de ítems debe cargarse. Recupera del servidor el listado de ítems en esa posición y la muestra al usuario.
Flujos alternativos	2. Si no existe una posición guardada en la que la vista de ítems deba cargarse, la aplicación la asigna por defecto el valor de la rama main. A continuación, este caso de uso continua en el paso 2. 2. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.16 - Cambiar el punto en el que se explora el contenido del repositorio

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Explorar el contenido de un repositorio".
Postcondiciones	El usuario podrá visualizar el contenido del repositorio en la posición especificada.
Escenario principal	1. El usuario escoge, de entre sus opciones de menú, la correspondiente a cambiar el punto en el que se explora el contenido del repositorio. 2. La aplicación muestra un menú con la posibilidad de cambiar el punto en el que se explora el contenido del repositorio a una rama o un changeset.

	3. El usuario escoge el tipo de objeto en el que quiere explorar el contenido del repositorio. 4. La aplicación muestra un listado de los objetos más recientes del tipo seleccionado, con la posibilidad de realizar consultas avanzadas. 5. El usuario selecciona, de entre las opciones del listado, el objeto en el que quiere explorar el contenido del repositorio. 6. La aplicación recarga el listado de ítems con el contenido del repositorio en el punto seleccionado. Guarda, entre la configuración del usuario, el punto en el que se está explorando el repositorio.
Flujos alternativos	3. Si el usuario escoge la opción correspondiente a "cancelar", la aplicación esconde el menú de selección de tipo de objeto. A continuación, este caso de uso queda sin efecto. 5. Si el usuario escoge la opción correspondiente a "cancelar", la aplicación esconde el menú de selección de objeto. A continuación, este caso de uso queda sin efecto. 6. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.17 - Obtener un listado del historial de un fichero

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Explorar el contenido del repositorio", o bien ha completado el caso de uso "Cambiar el punto en el que se explora el contenido de un repositorio".
Postcondiciones	El usuario podrá visualizar el historial de revisiones del fichero escogido.
Escenario principal	1. El usuario realiza una pulsación larga sobre un ítem, en la vista de ítems, para acceder a su menú contextual. 2. El sistema muestra el menú contextual para el ítem especificado. 3. El usuario escoge, de entre las opciones de menú disponibles, aquella correspondiente a la visualización del historial del fichero. 4. La aplicación muestra al usuario un listado con el historial de revisiones del fichero.
Flujos alternativos	3. Si el usuario escoge la opción correspondiente a "Cancelar", la aplicación oculta el menú contextual del ítem. A continuación, este caso de uso queda sin efecto. 4. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

3.2.18 - Descargar un fichero del repositorio

Actor principal	Usuario
Precondiciones	El usuario ha completado el caso de uso "Explorar el contenido del repositorio", o bien ha completado el caso de uso "Cambiar el punto en el que se explora el contenido de un repositorio".
Postcondiciones	El usuario habrá descargado a su almacenamiento local una revisión de un fichero del repositorio.
Escenario principal	1. El usuario realiza una pulsación larga sobre un ítem, en la vista de ítems, para acceder a su menú contextual. 2. El sistema muestra el menú contextual para el ítem especificado.

	3. El usuario escoge, de entre las opciones de menú disponibles, aquella correspondiente a la descarga a almacenamiento local del ítem. 4. La aplicación descarga, desde el repositorio a un directorio predeterminado, el ítem seleccionado.
Flujos alternativos	3. Si el ítem escogido no es un fichero (es un directorio), la opción correspondiente a la descarga a almacenamiento local del ítem no se encuentra disponible. A continuación, este caso de uso continua en el paso 2. 3. Si el usuario escoge la opción correspondiente a "Cancelar", la aplicación oculta el menú contextual del ítem. A continuación, este caso de uso queda sin efecto. 4. Si la conexión contra el servidor falla, la aplicación muestra un mensaje de error. A continuación, este caso de uso queda sin efecto.

Capítulo IV – Diseño

4.1 - Diseño de la Interfaz de usuario

Con los casos de uso identificados y las *User Stories* presentes, se inició el diseño de la interfaz de usuario mediante una especificación detallada de la misma, con las opciones más importantes disponibles, y un flujo de trabajo y de interacción definido sobre el que poder empezar a programar funcionalidad.

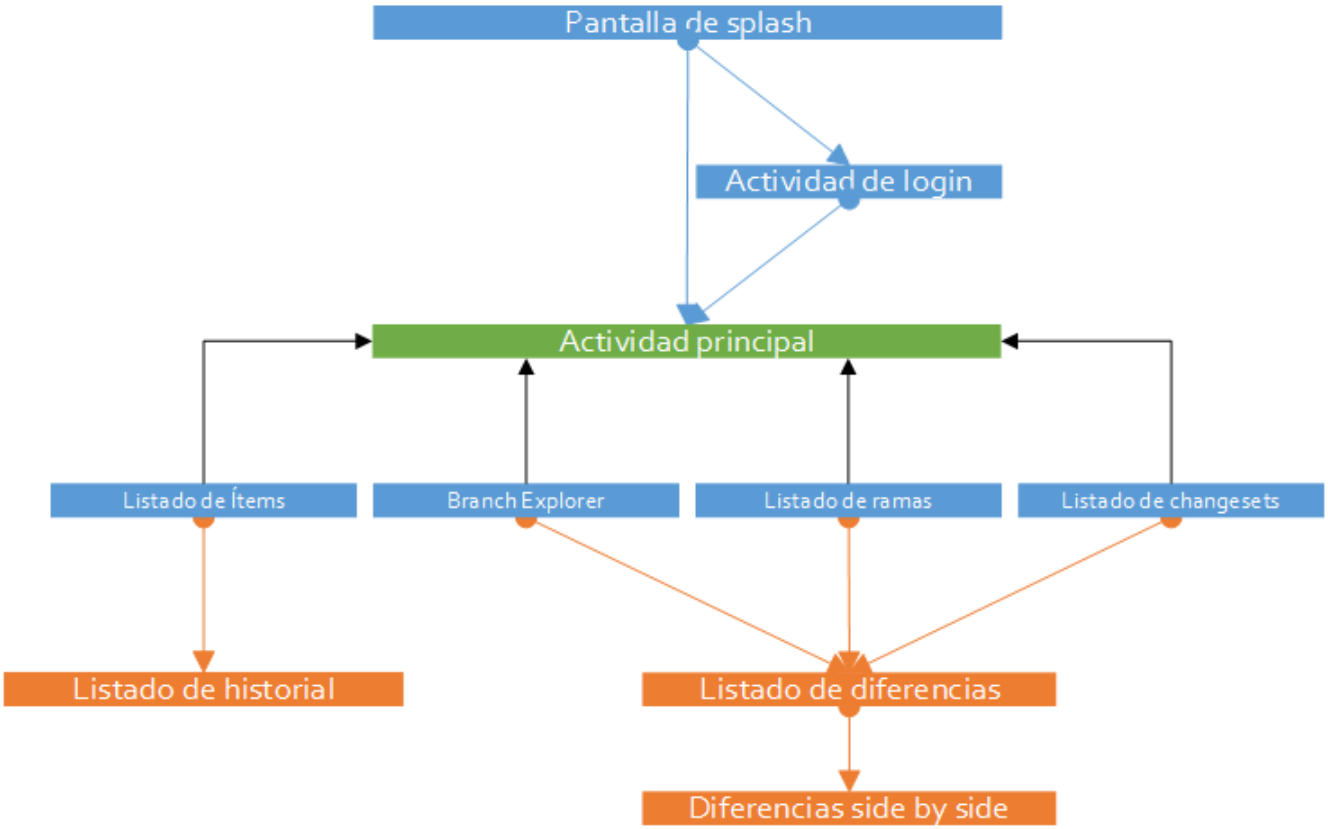

FIGURA 10. ESQUEMA DE NAVEGACIÓN EN EL PROTOTIPO

La navegación en la aplicación comenzaría, de esta forma, por la pantalla de *splash* -o *pantalla de inicio-*, que consiste únicamente en una ventana con el logo de la aplicación. Es una práctica común en las aplicaciones móviles, y sirve, además de para reforzar la imagen de marca, para decidir si se debe lanzar la actividad principal o la pantalla de *login*, dependiendo de si existen datos guardados sobre el usuario y sus repositorios.

En la *figura 10*, las actividades de primer nivel están marcadas de color azul, mientras que las consideradas secundarias están marcadas en naranja. Las flechas negras indican cuál es el contexto de la actividad, mientras que las azules y naranjas indican el orden de navegación que debe seguir el usuario para acceder a ellas.

Las herramientas que se utilizaron para el prototipado de la interfaz de usuario fueron Microsoft Visio y el propio editor de *layouts* de Microsoft Visual Studio. En Android, las vistas se pueden maquetar utilizando XML, lo que permite un prototipado rápido de las mismas una vez se conocen los elementos y anotaciones que se pueden utilizar, con resultados que se pueden utilizar en una versión final a medida que se progresa en las

iteraciones. Microsoft Visio permitió añadir anotaciones sobre el propósito de cada elemento, su comportamiento, la navegación entre pantallas, etc.

4.1.1 - Actividad de *login*

La actividad de *login* sirve para añadir a la aplicación móvil de Plastic SCM los primeros repositorios. Tiene una apariencia diferente a la que tendría aquella desde la que se añaden sucesivos repositorios porque sirve al doble propósito de, por un lado, presentación de la aplicación, y por otro, para que el usuario acepte el acuerdo de licencia del software[3].

FIGURA 11. PROTOTIPO DE LA PANTALLA DE *LOGIN*

En la actividad de *login* se encuentra un formulario con todos los campos necesarios para conectarse a un servidor de Plastic SCM: la dirección del mismo, un nombre de usuario, y una contraseña. Al configurar la utilidad de consola de Plastic SCM en un ordenador, actualmente es necesario especificar la configuración de seguridad de Plastic SCM. Destacan las siguientes:

[3] Con el fin de que la aplicación sea lo más realista posible, se han añadido acuerdos de licencia que el usuario debería aceptar antes de continuar con el uso de la aplicación, al estilo del software comercial.

1. Nombre de usuario: el nombre que un usuario tenga en el sistema operativo sobre el que se esté ejecutando el servidor de Plastic SCM.
2. Nombre de usuario + contraseña: Plastic SCM maneja un fichero de configuración de usuarios, donde cada uno tiene una contraseña que puede escoger.
3. *Active Directory*: Plastic SCM delega la autenticación de usuarios en Active Directory, un producto de Microsoft para manejar usuarios en entornos corporativos.
4. LDAP: Plastic SCM delega la autenticación de usuarios en un servidor LDAP configurado por el administrador.

Sin embargo, los clientes de escritorio y el cliente móvil detectan automáticamente la configuración de seguridad del servidor, para, en función de ello, comprobar que los datos de acceso son correctos de una forma o de otra.

En la pantalla de *login*, el botón de conexión se encontrará desactivado si los campos de dirección o de usuario -indispensables- están vacíos. Una vez han sido rellenados, este se activa.

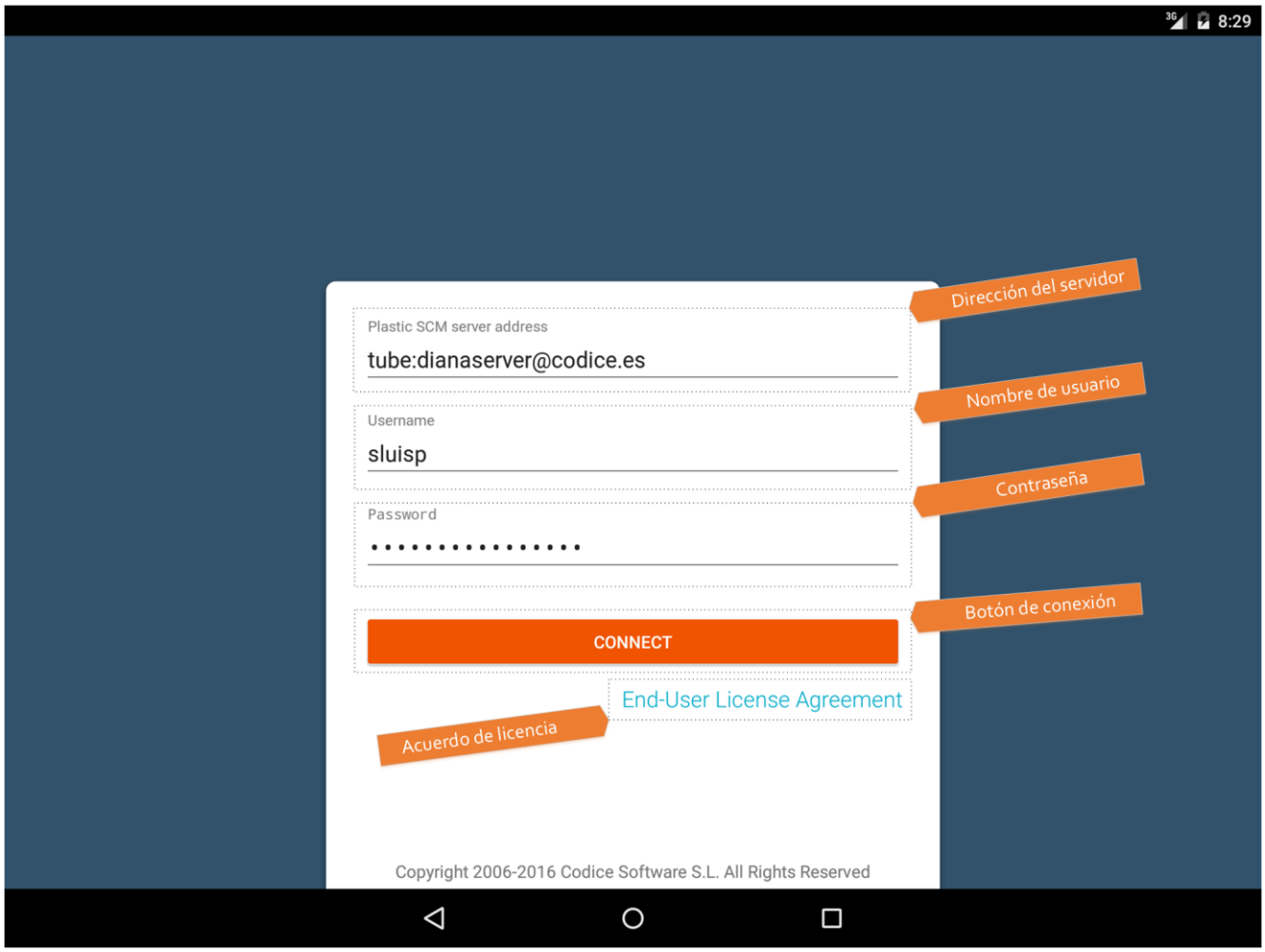

FIGURA 12. PROTOTIPO DE LA PANTALLA DE *LOGIN* CON EL BOTÓN DE CONEXIÓN ACTIVADO

Al pulsar sobre el botón de conexión, y antes de conectarse al servidor, la aplicación bloquea la interfaz con un diálogo en el que se le advierte al usuario que continuar con la operación supone aceptar el acuerdo de licencia del producto.

Según las guías de diseño de Android [5], un diálogo de alerta puede tener hasta tres botones. Los dos agrupados al final del mismo (que, en aquellos dispositivos configurados en idiomas que se leen de izquierda a derecha, es el lado derecho) deben realizar acciones *negativa* y *positiva*. Se define como acción *positiva* aquella que vaya a cambiar el estado de la aplicación de alguna manera, y *negativa* aquella que sencillamente elimine el diálogo del foco sin realizar ninguna acción. Así, por ejemplo, en un diálogo que pida confirmación sobre borrar algún elemento del sistema de ficheros, la acción positiva va a ser "Borrar", y la acción negativa va a ser "Cancelar".

Adicionalmente se puede añadir al diálogo una tercera acción, al principio del mismo, que sea neutral, y que habitualmente será utilizada para acceder a información adicional sobre la acción positiva del diálogo.

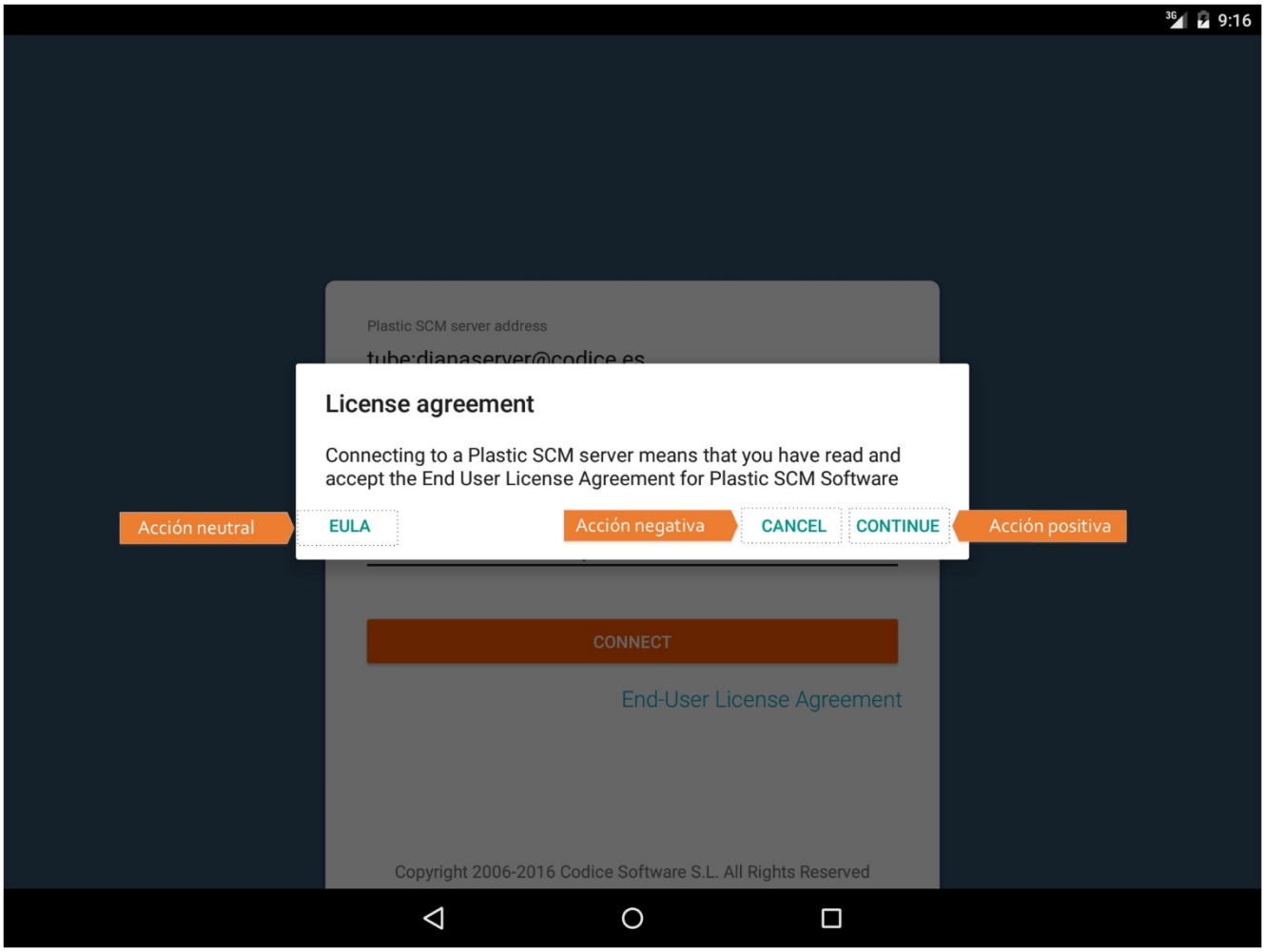

FIGURA 13. PROTOTIPO DEL DIÁLOGO DE *ACUERDO DE LICENCIA*

Una vez se han recuperado del servidor los repositorios disponibles, estos son mostrados al usuario de nuevo en un diálogo que bloquea la interfaz. En este caso no habrá opción neutral, únicamente una negativa (que, según el caso de uso "Añadir repositorios del primer servidor a la aplicación", únicamente ocultaría el diálogo, y una acción positiva, que añade referencias a los repositorios a la aplicación y lleva al usuario a la actividad principal.

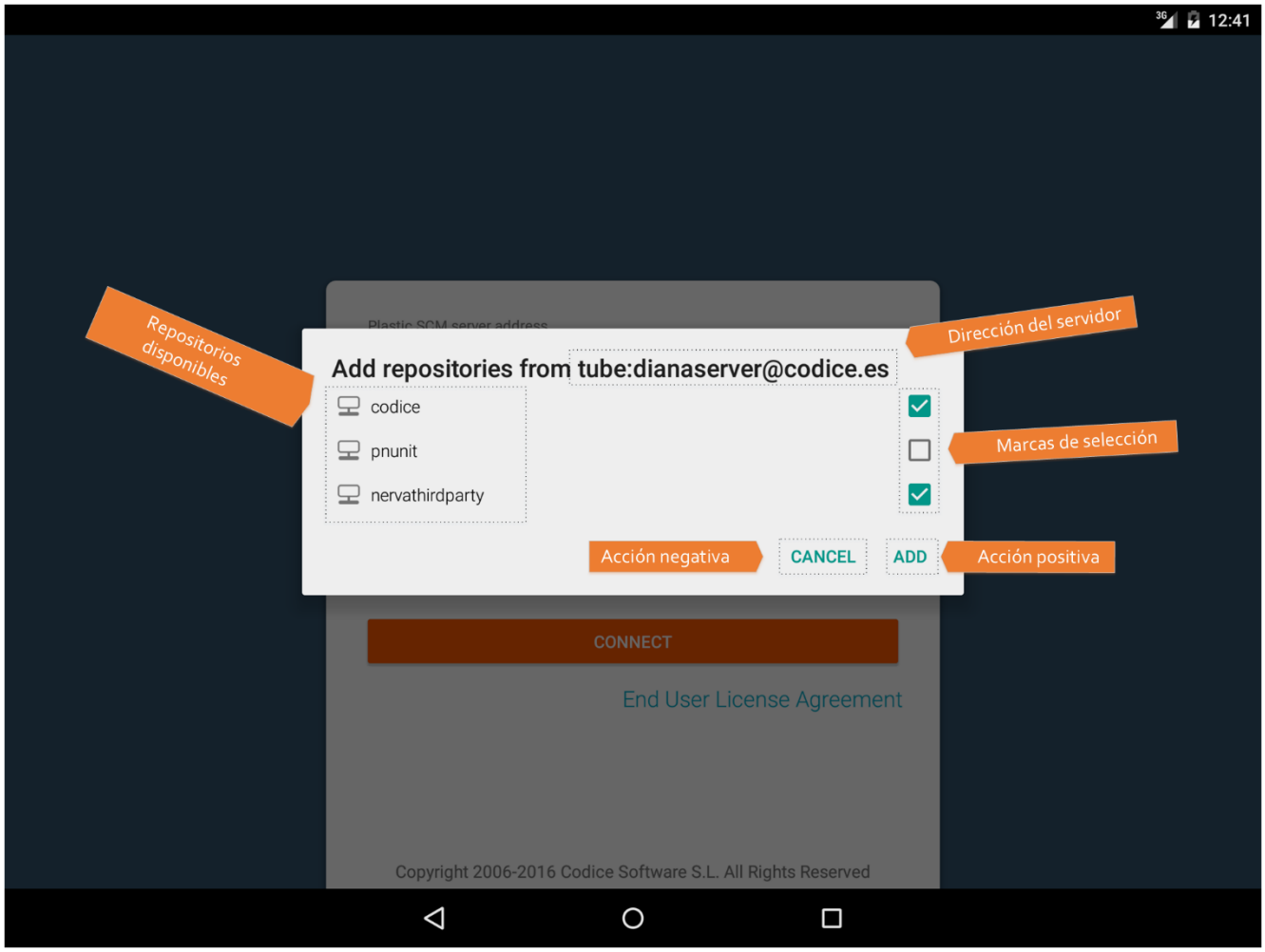

FIGURA 14. PROTOTIPO DEL DIÁLOGO DE SELECCIÓN DE REPOSITORIOS

4.1.2 - Actividad principal

La llamada "Actividad principal" (castellanizando la terminología de Android, en la que la *Main Activity* es aquella `Activity` donde se van a realizar las operaciones más importantes de la aplicación, y donde el usuario va a pasar la mayor parte del tiempo de interacción) va a servir de *contenedor* para todas las actividades consideradas de primer nivel (o de mayor importancia) que un usuario va a poder realizar en el cliente móvil de Plastic SCM. Por lo tanto, proveerá la funcionalidad de navegación entre dichas actividades, así como de navegación entre los distintos repositorios añadidos a la aplicación.

A la hora de implementar la navegación entre las distintas actividades de primer nivel que un usuario puede realizar en la aplicación, se optó por la navegación lateral. Dicho paradigma fue incluido recientemente en la guía de estilo oficial de Android [6] -donde se especifican detalles tales como medidas, número máximo de iconos, etc, con el fin de mantener coherencia visual y de comportamiento entre las aplicaciones que lo implementen-, y además es el que se utiliza en los clientes de escritorio de Windows, GNU/Linux, y macOS, por lo que resultará familiar desde el primer momento a todos los usuarios de la aplicación.

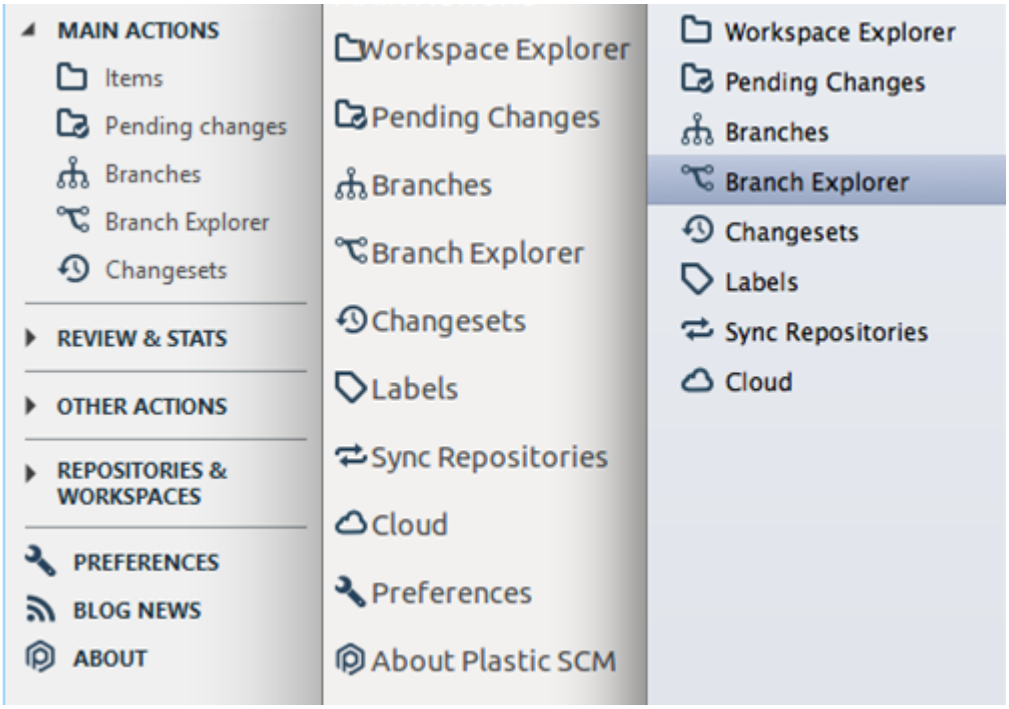

FIGURA 15. NAVEGACIÓN EN WINDOWS, GNU/LINUX Y MACOS

En la *figura 15* se puede observar cómo es la barra de navegación lateral en los clientes de escritorio actuales. Para la aplicación móvil, se adaptó el esquema de colores de las distintas tonalidades de azul del cliente completo a gris/naranja para hacer más llamativo el elemento seleccionado (ya que el naranja también forma parte de la paleta de colores de la imagen de marca de Plastic SCM). Para que las secciones fuesen fácilmente identificables, se utilizaron los mismos iconos que en las versiones de escritorio, y además en el mismo orden de aparición.

El panel de información que se puede ver en la parte inferior de la *figura 16* es algo de lo que no dispone la versión de Windows de Plastic SCM, pero que introdujeron las versiones de GNU/Linux y macOS. En dicho panel se muestra información tal como el progreso de una operación, o el error de la misma en caso de haberlo, cambiando de color según cuál sea el significado del mensaje en pantalla.

En la barra superior, llamada en terminología de UI de Android "*ActionBar*", se encontrará el título de la vista que se esté mostrando actualmente, así como un botón para desplegar el listado de repositorios (que no se encuentra en la figura 16 por haber sido añadido posteriormente).

En sucesivas revisiones de la aplicación se añadió a la *ActionBar,* además del título, un subtítulo indicando el repositorio actual por el que se está navegando, y se substituyó el listado de ítems por el de *labels*, tal y como se puede ver en la *figura 17*.

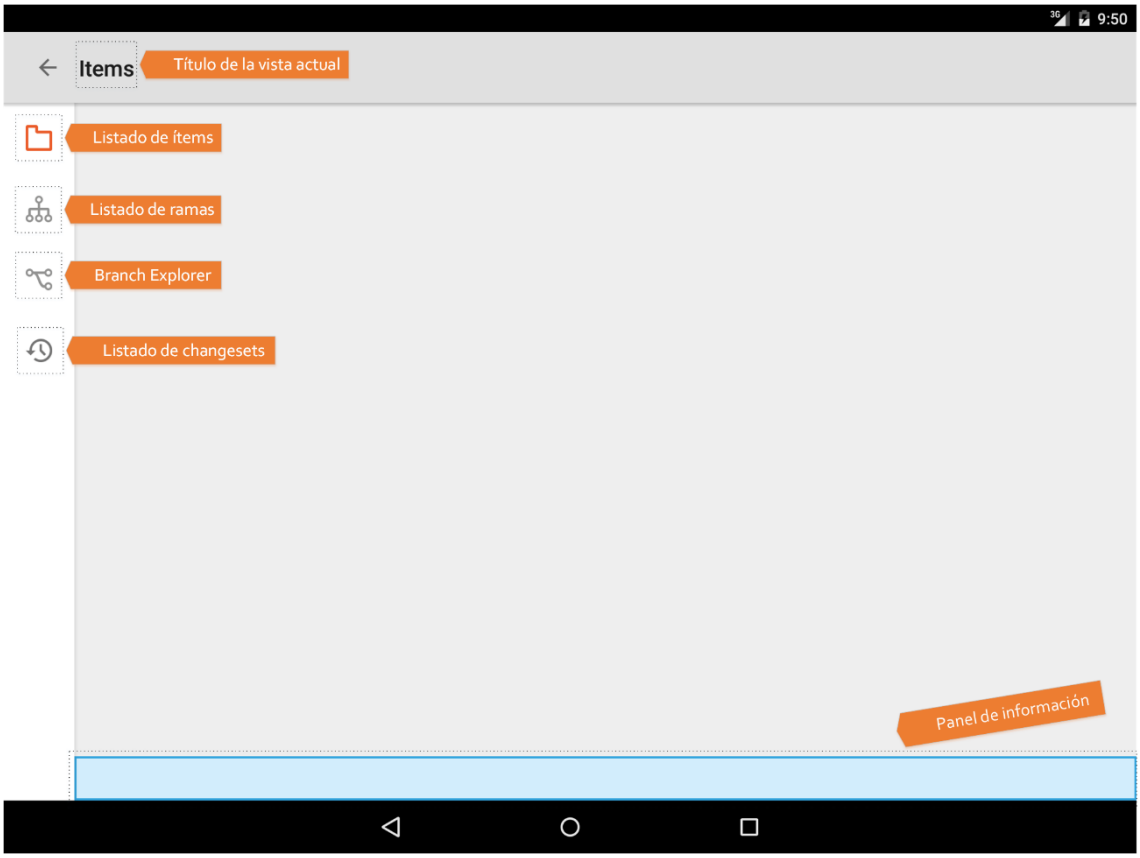

FIGURA 16. PROTOTIPO DE LA *ACTIVIDAD* PRINCIPAL

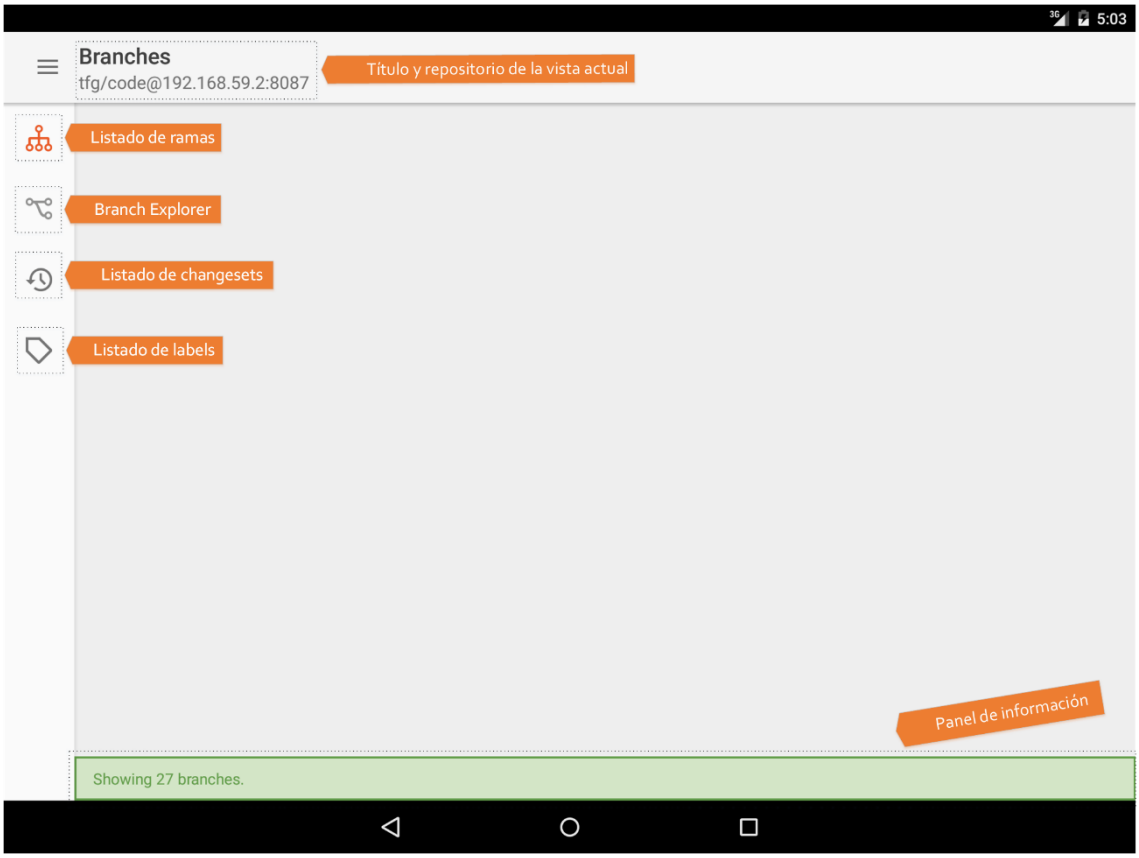

FIGURA 17. VERSIÓN FINAL DE LA *ACTIVIDAD* PRINCIPAL

4.1.3 - Listados de ítems

La aplicación de escritorio de Plastic SCM conoce en qué punto del repositorio se encuentra un *workspace* gracias a un fichero llamado *selector*, que se encuentra en el directorio oculto `.plastic` (dentro del propio *workspace*) en el que almacenan datos adicionales para el correcto funcionamiento del software.

El *selector* no es más que un fichero de texto plano con información descriptiva sobre el repositorio y la rama, etiqueta, o *changeset*, cuyo contenido se encuentra cargado en el *workspace*, y que se actualiza cada vez que se realiza alguna operación que así lo requiera, como la de *switch*. Una primera versión del prototipo de la aplicación móvil de Plastic SCM utilizaba el mismo concepto de *selector* para que el usuario supiese desde qué punto del repositorio se había cargado la lista de ítems, con el fin de reducir el tiempo de desarrollo. En futuras versiones el *selector* desaparecería, dando paso a un menú con listados de etiquetas, *changesets*, y ramas, para que el usuario pudiese cambiar el punto desde el que se carga la lista de ítems sin editar explícitamente dicho *selector*.

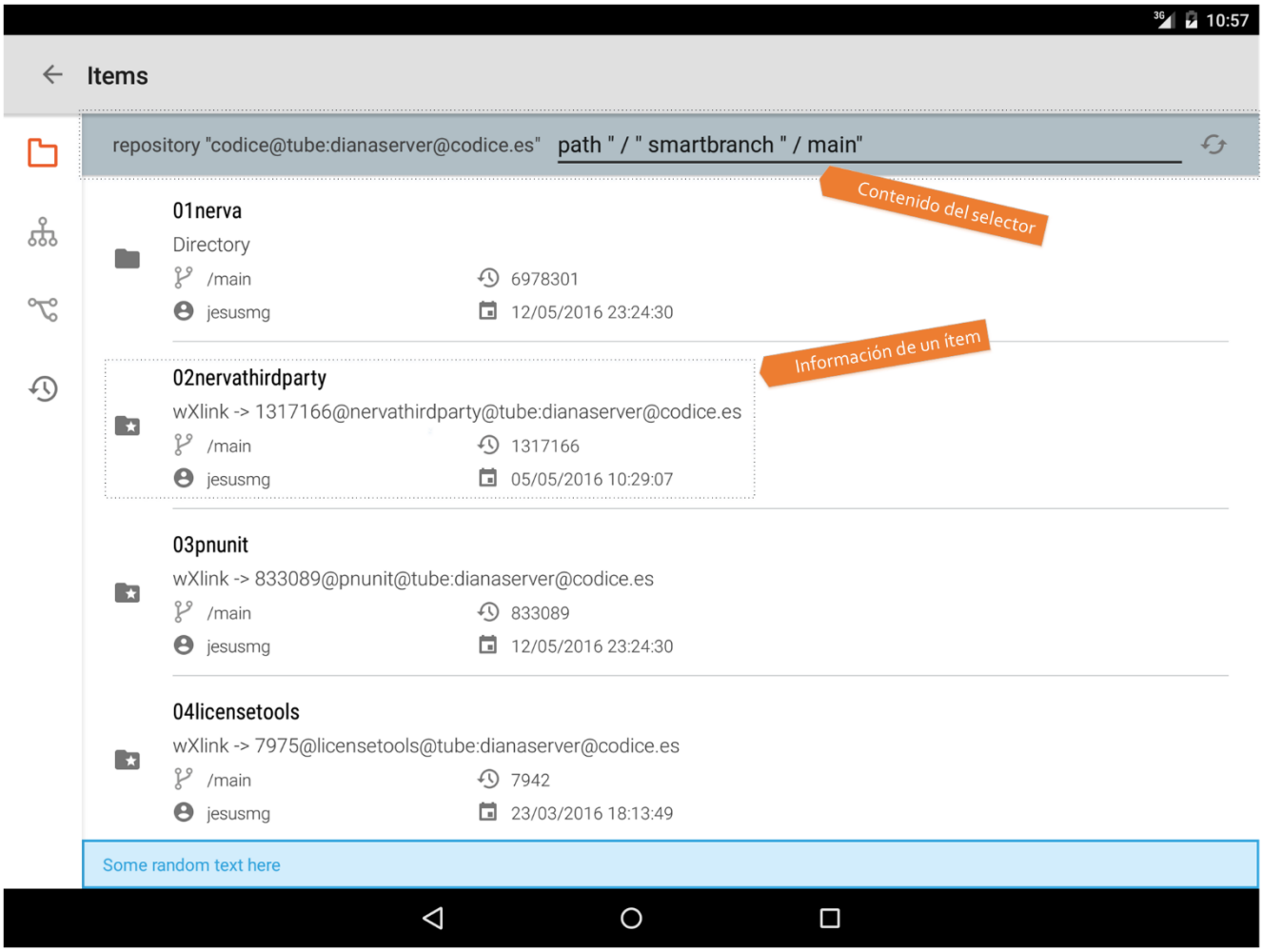

FIGURA 18. PROTOTIPO DE LA VISTA DE ÍTEMS

Para asegurar que en todo momento se carga contenido del repositorio en el que el usuario se encuentra en la aplicación, el *selector* se puede editar parcialmente, siendo la parte que indica el repositorio fija.

La información de cada ítem constaría, al igual que en el cliente de escritorio, de su nombre, un icono identificativo, una descripción del mismo (si es un directorio, un fichero de texto, un binario…), así como en qué rama fue editado por última vez, por quién, en qué *changeset* y en qué fecha. No se muestra la información del estado del ítem (controlado, ignorado, oculto, *checked-out*…) porque esta únicamente existe para *worskpaces*. Explorando el contenido del repositorio en cualquier otro punto sólo se muestran los elementos controlados.

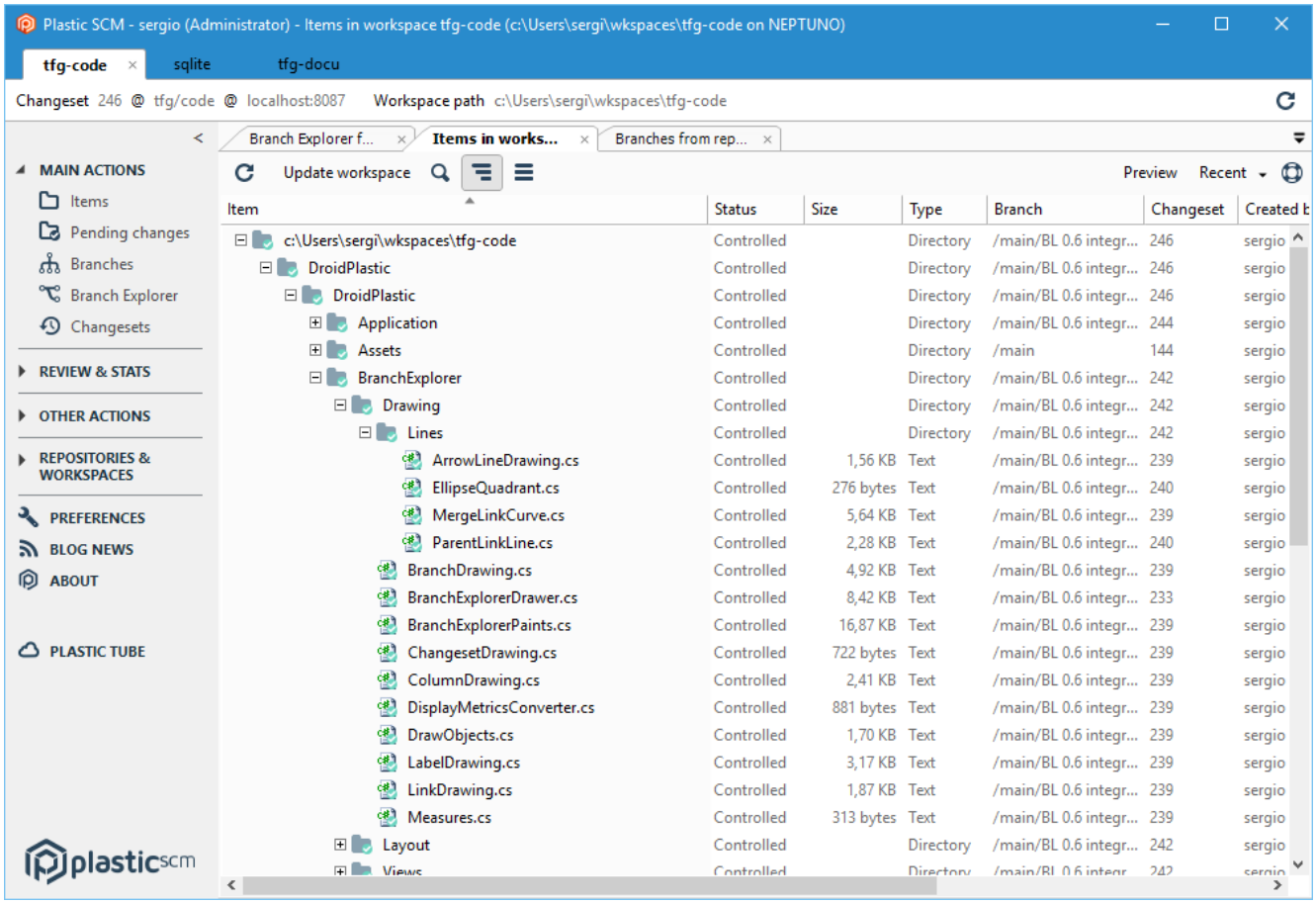

FIGURA 19. VISTA DE ÍTEMS EN LA APLICACIÓN DE WINDOWS

Así mismo, en el prototipo tampoco existiría la opción de explorar el contenido como un árbol (tal y como se ve la *figura 19*) sino que, una vez se ha navegado al interior de un directorio, el contenido del mismo pasaría a ocupar el listado completo, utilizando, para la navegación hacia arriba en el árbol de directorios, o bien el botón del sistema "*atrás*", o bien un elemento situado en la posición `0` del listado, con los dos puntos propios de las interfaces de texto que referencian al directorio padre (`..`).

Finalmente, la vista de ítems desapareció de la definición del *producto mínimo viable* por el tiempo de desarrollo que conllevaría en comparación con el valor que le puede aportar a un usuario que no vaya a poder crear *workspaces* en su tableta (pues explorar el contenido del repositorio en un punto distinto al *workspace* actual es una funcionalidad residual en comparación con la de poder listar las diferencias en dicho punto).

4.1.4 - Explorador de ramas

El *Branch Explorer* es el punto central de Plastic SCM para la mayoría de sus usuarios debido a la gran cantidad de información que en él se muestra, y a la cantidad de operaciones que desde él se puede realizar.

Como ya se ha ahondado (y se ahondará más posteriormente) en el Branch Explorer, en esta sección simplemente se compararán el prototipo del mismo en la aplicación móvil frente a cómo es este en la aplicación de Windows.

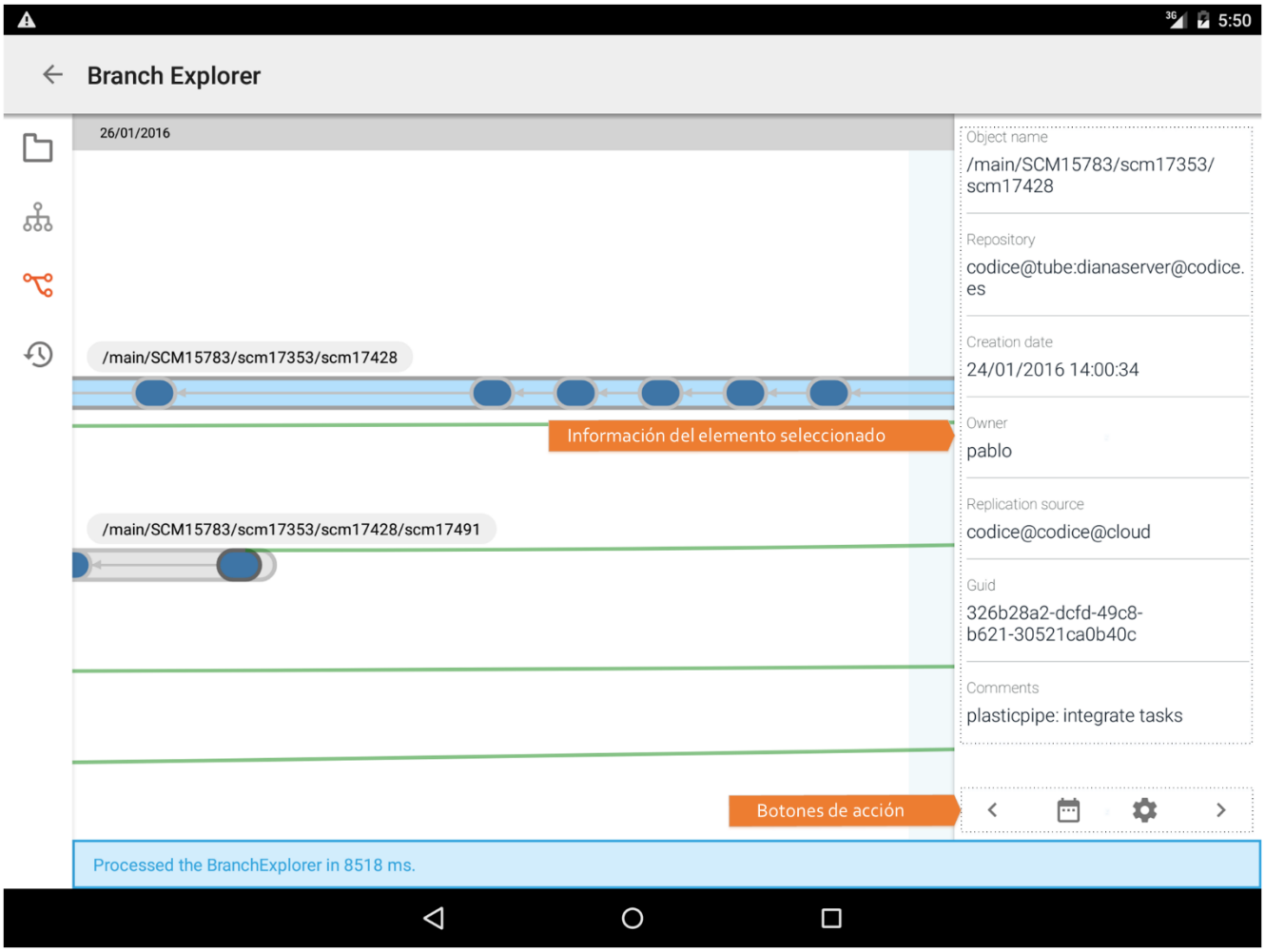

FIGURA 20. PROTOTIPO DEL BRANCH EXPLORER

Al igual que en los clientes de Windows, GNU/Linux, y macOS, el Branch Explorer dispone de un panel lateral en el que se muestra la información del objeto seleccionado (que es destacado sobre los demás cambiando su color de fondo). En la *figura 20*, se muestra, por orden, el nombre del objeto (que en el caso de un *changeset*, como estos no tienen nombre asignado por el usuario, será el número identificativo del mismo), el repositorio al que pertenecen (que será, en la aplicación móvil, aquel que se esté visualizando), la fecha de creación, quién ha creado dicho objeto (*owner*), cuál es el origen de replicación (en caso de existir), el `Guid` del objeto, y los comentarios del mismo.

Los botones de acción que se ven en la ilustración son, de izquierda a derecha:

- **Desplazar el Branch Explorer a la izquierda**: la versión de escritorio tiene un panel de navegación para desplazarse rápidamente por el Branch Explorer. Este botón supliría dicha funcionalidad, permitiendo navegar en un solo golpe dos veces el ancho de la Viewport (más sobre esto en el *Capítulo V – Implementación*) hacia la izquierda.
- **Cambiar el filtro de fechas**: permitiendo editar cuál es la fecha de inicio y de fin de las que se muestran datos en el explorador de ramas.
- **Opciones de visualización**: permitiendo ocultar ramas, *changesets*, etiquetas, y *mergelinks*, y permitiendo mostrar el nombre completo o parcial de cada rama.
- **Desplazar el Branch Explorer a la derecha,** de igual forma que el primer botón.

En la ***¡Error! No se encuentra el origen de la referencia.***, se pueden observar los equivalentes señalados.

FIGURA 21. BRANCH EXPLORER EN WINDOWS

4.1.5 - Vista de búsquedas (listado de ramas, *changesets* y etiquetas)

En el prototipo de la aplicación móvil de Plastic SCM inicialmente había dos vistas de búsquedas: la de *changesets* y la de ramas. Finalmente, al desechar el listado de ítems, se añadió también la vista de etiquetas.

Dichas vistas consisten en un cuadro de búsquedas donde el usuario pueda introducir consultas avanzadas que se ajusten a sus necesidades, y el listado de resultados, donde se mostrará la información recuperada del servidor.

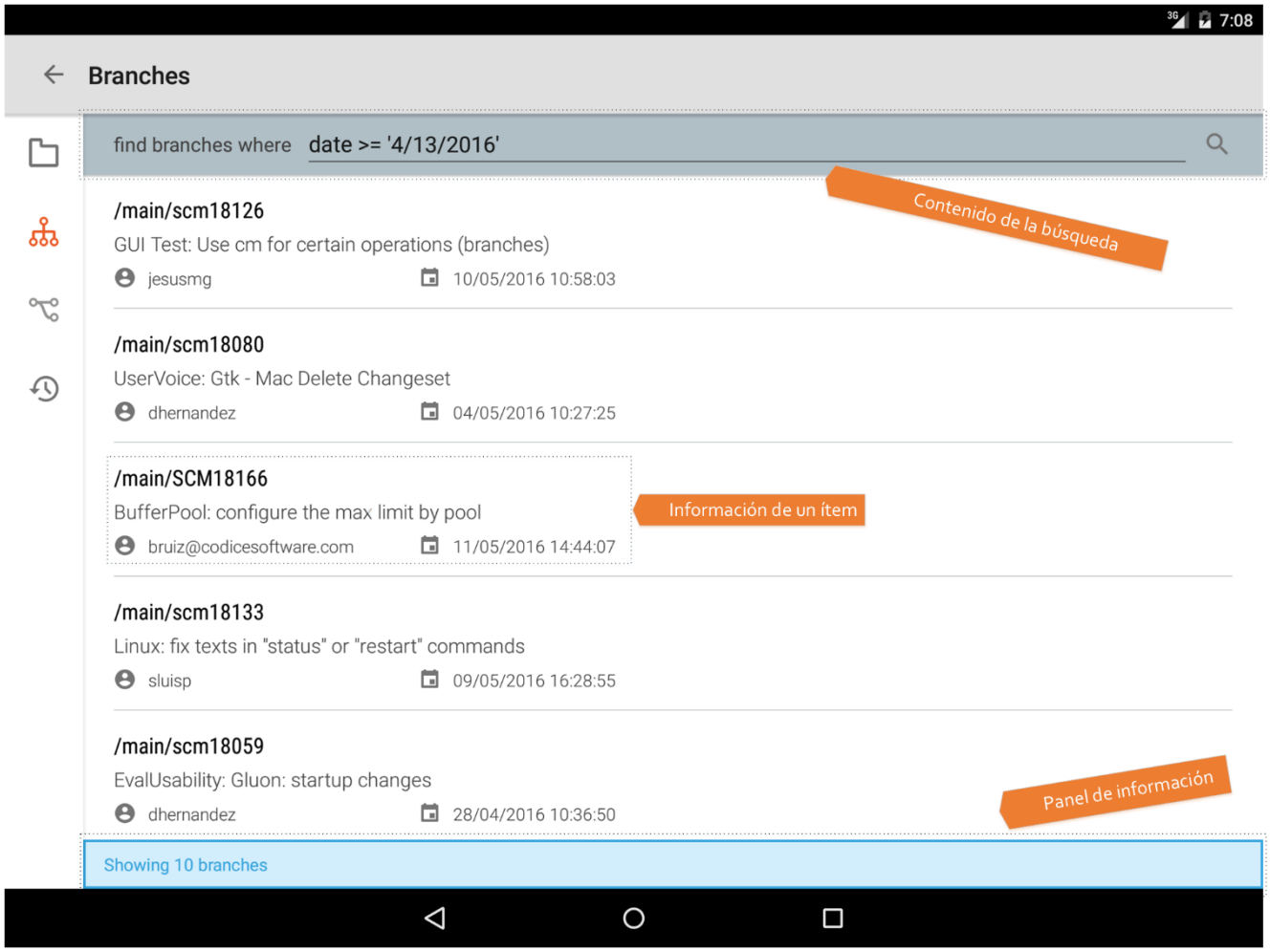

FIGURA 22. PROTOTIPO DE LA VISTA DE LISTADO DE RAMAS

Para evitar que el usuario busque tipos de elementos que no se esperan (como ramas en la vista de *changesets*), la primera parte de la búsqueda, donde se especifica el objeto, no es editable. Plastic SCM no soporta además *queries* con más de un tipo de objetos en los resultados (`find branches where […] and changesets where […]`), por lo que el usuario no pierde funcionalidad en ningún momento.

Cuando el usuario introduce una búsqueda vacía (eliminando la cláusula del *where*), tal y como se especificó en los casos de uso, la aplicación utiliza como cláusula *where* por defecto la fecha actual menos un mes.

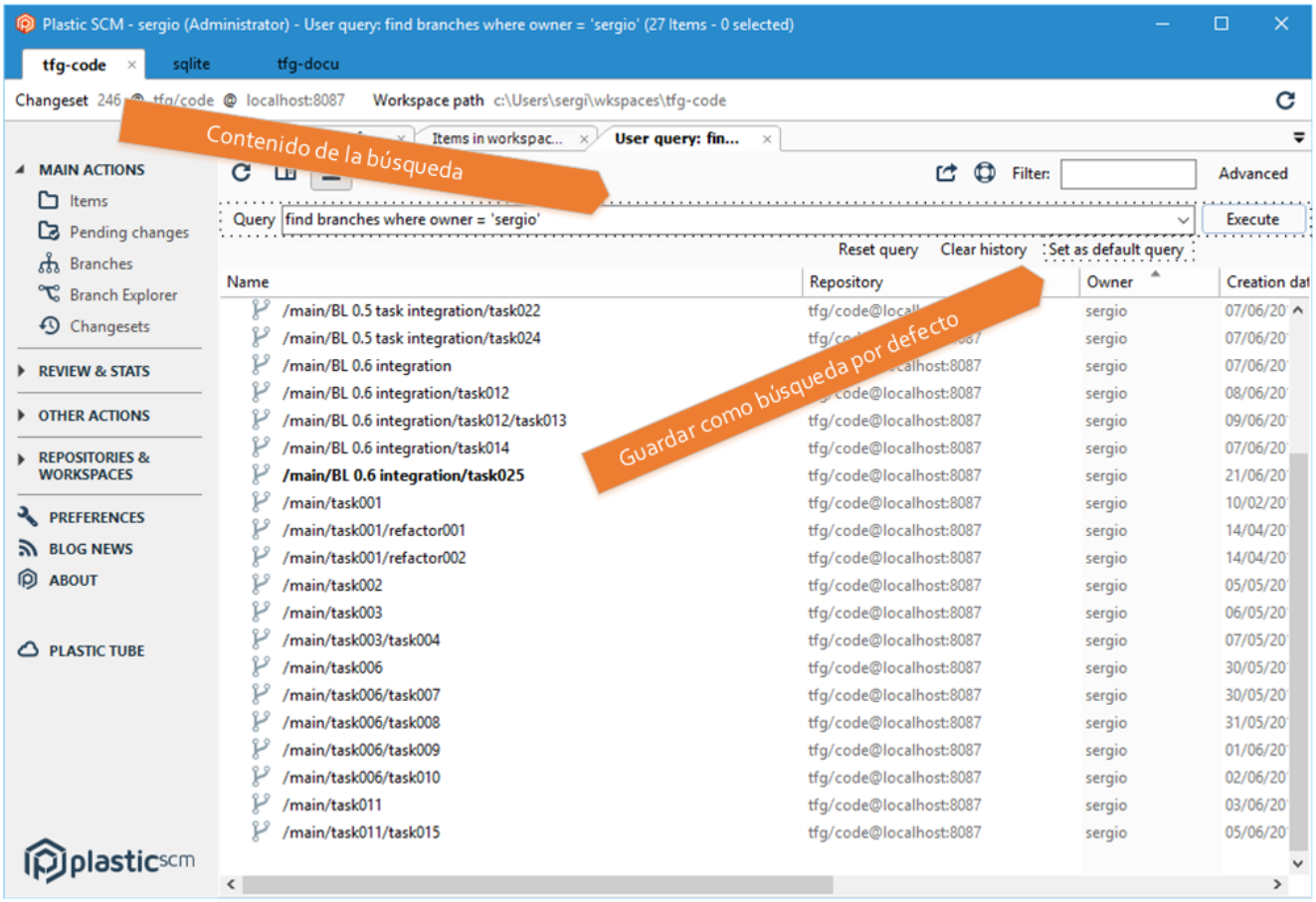

FIGURA 23. VISTA DEL LISTADO DE RAMAS EN WINDOWS

Mientras que la aplicación de escritorio de Plastic SCM da la posibilidad de establecer la consulta actual como consulta por defecto, la aplicación móvil guardará siempre la consulta introducida por el usuario si esta difiere de dicha consulta por defecto (borrándola si el usuario introduce una cláusula *where* vacía, como ya se ha señalado).

4.1.6 - Listado de diferencias

La aplicación de escritorio de Plastic SCM permite obtener listado de diferencias entre dos puntos cualesquiera del repositorio. En concreto: un *changeset* con su anterior o con cualquier otro *changeset*, una rama con su rama padre, con cualquier otra rama, o con cualquier etiqueta, y una etiqueta con otra etiqueta. Para la primera versión de la aplicación móvil de Plastic SCM se decidió, por simplicidad, permitir únicamente obtener listados de diferencias de un *changeset* con su anterior, y de una rama con su padre.

Los cambios aparecen agrupados, en la versión de escritorio, en cuatro grandes categorías: cambiados, añadidos, eliminados y movidos. Adicionalmente, si ha habido operaciones de *merge* en el elemento del que se estén obteniendo las diferencias, los ítems aparecerán duplicados en las categorías de *merge* correspondiente. En la aplicación móvil, al menos en la primera versión, no se distinguirá entre elementos modificados en una operación de *merge* o no.

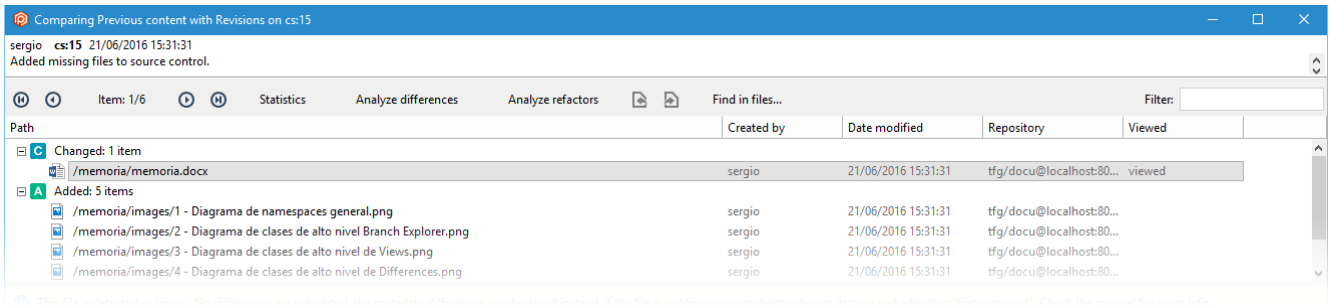

FIGURA 24. LISTADO DE DIFERENCIAS EN WINDOWS

Mientras que en Windows el listado de diferencias forma parte de la vista de una pantalla más grande, en la que también se pueden ver las diferencias *side by side* del elemento seleccionado, por lo limitado del tamaño de una tableta, en la aplicación móvil el listado de diferencias y las diferencias *side by side* forman parte de *actividades* diferentes.

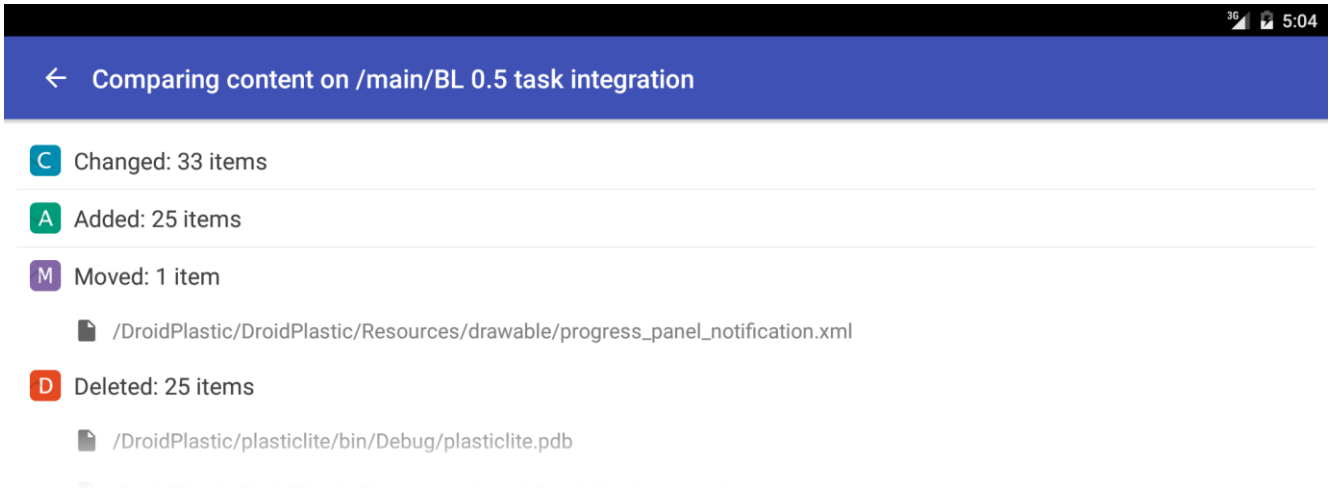

FIGURA 25. PROTOTIPO DEL LISTADO DE DIFERENCIAS

Los listados se pueden igualmente colapsar por categorías para facilitar la navegación.

4.1.7 - Diferencias *side by side*

Las diferencias *side by side* es otro de los puntos más importantes en la aplicación de escritorio de Plastic SCM, y por eso se consideró imprescindible en la visión del *mínimo producto viable*. Se ahondará más en las diferencias *lado a lado* en la sección *5.2 - Diferencias de texto side by side*, por lo que aquí únicamente se compararán el prototipo de la aplicación móvil y su equivalente en el escritorio.

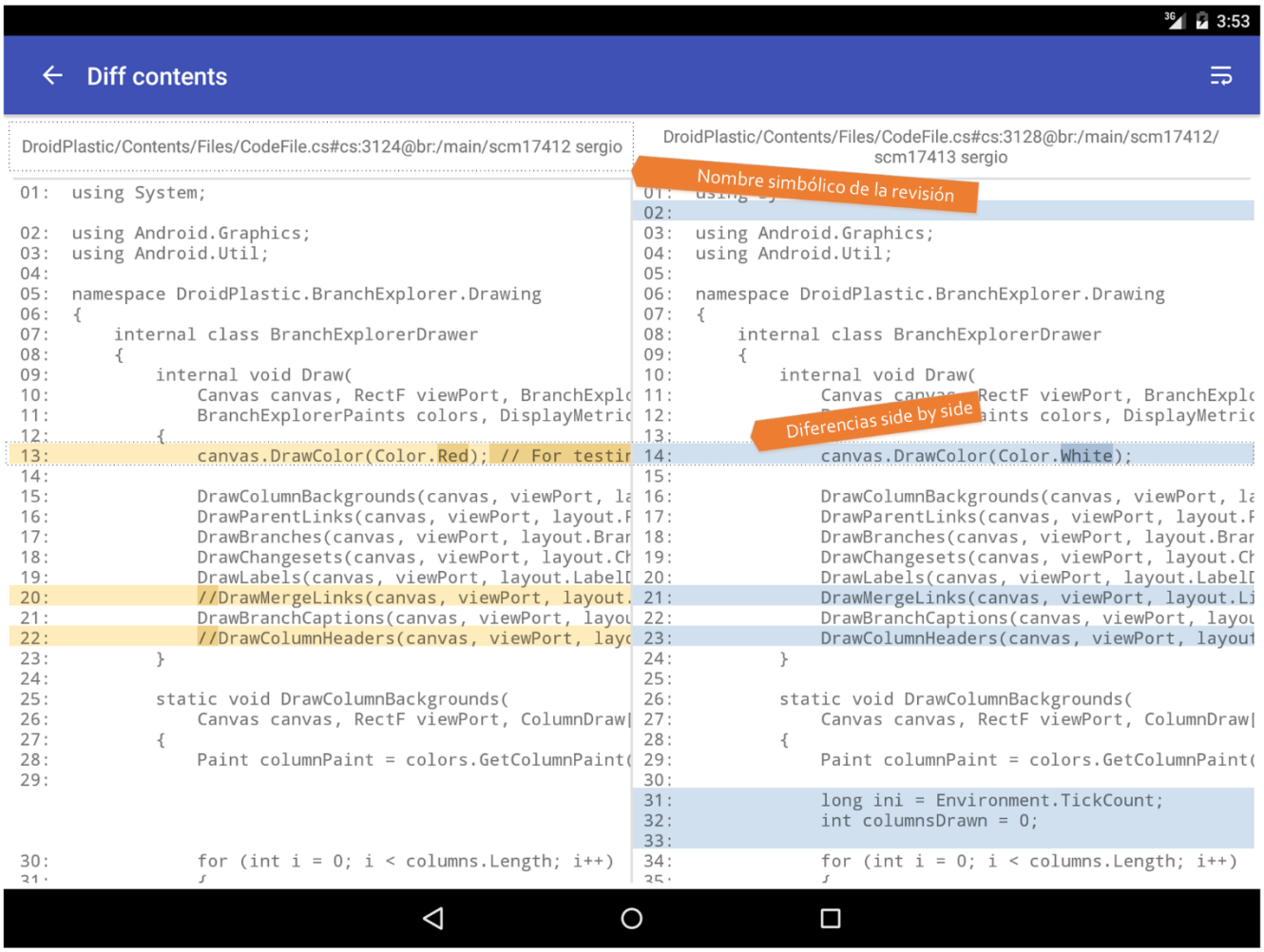

FIGURA 26. PROTOTIPO DE LAS DIFERENCIAS *SIDE BY SIDE*

En la vista de diferencias se comparan simultáneamente dos revisiones del mismo fichero, siendo la revisión izquierda aquella cuya fecha de creación quede más alejada en el tiempo. Para ayudar al usuario a identificarlas, cada revisión va acompañada de un nombre "simbólico". Es decir, como el fichero no es accesible por el usuario dentro del sistema de ficheros de la tableta (aunque en las versiones de escritorio este sea guardado en un directorio temporal para poder realizar el cálculo de diferencias), el nombre simbólico identifica inequívocamente una revisión del fichero en un punto concreto del repositorio.

Mientras que las cajas de texto en las versiones de escritorio de Plastic SCM están sincronizadas para mantener la correspondencia entre líneas en los movimientos de *scroll* para que las operaciones de añadido/borrado y movido de texto no afecten a cómo el código es visualizado, en el cliente móvil dicha sincronización se consigue intercalando líneas vacías allí donde sea necesario.

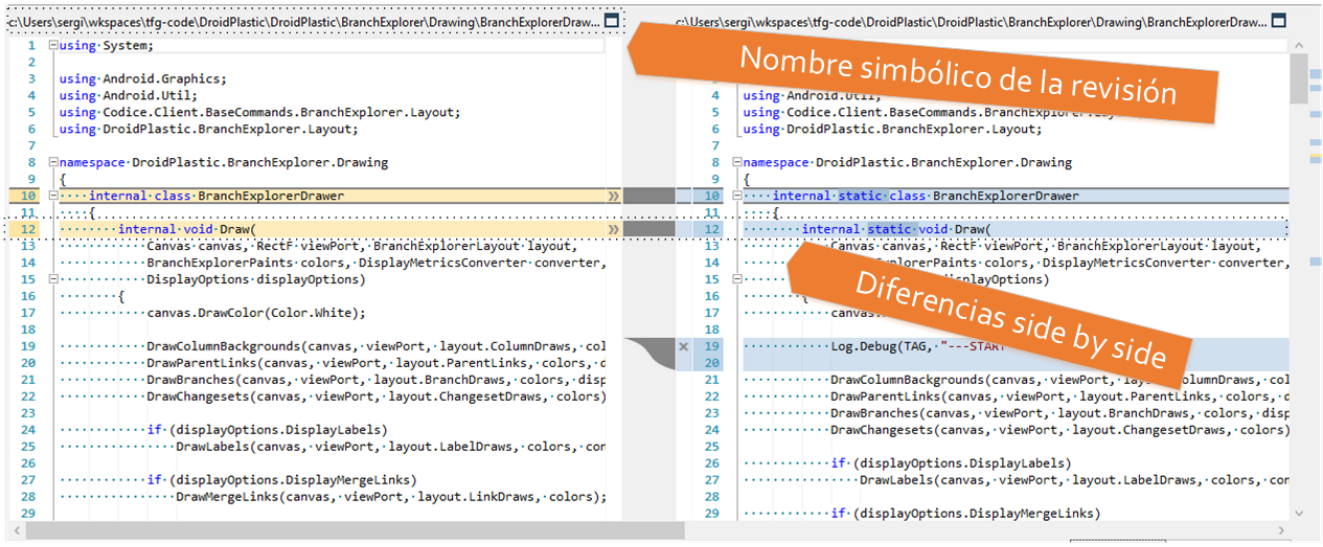

FIGURA 27. DIFERENCIAS *SIDE BY SIDE* EN WINDOWS

Se puede observar, así mismo, cómo el cliente de escritorio cuenta (menos en macOS en el momento de escribir esta memoria) de resaltado de sintaxis para el lenguaje de programación identificado mediante la extensión del fichero. En la *figura 27* se ven únicamente diferencias de texto. En versiones recientes de Plastic SCM en Windows se introdujo, para ciertos lenguajes (como C# o Java) las *diferencias semánticas*, en las que se puede identificar si se han modificado signaturas de métodos, se han añadido o modificado miembros de clase, `imports`, `usings`, etc, comparando no solo texto plano sino el árbol sintáctico que se puede elaborar a partir del código.

4.2 – Diagramas del diseño

Se detallan a continuación los diagramas de clases y de secuencia más relevantes a la hora de poder desenvolverse en torno al código del cliente de Android de Plastic SCM. Por su extensión, los diagramas de clases son *de alto nivel* (significando esto que no se detallan ni miembros de clase ni métodos, pero sí relaciones). Los diagramas de secuencia son detallados, omitiendo aquellas partes que no resulten relevantes para comprender la relación entre los distintos elementos.

4.2.1 - Diagrama general de *namespaces*

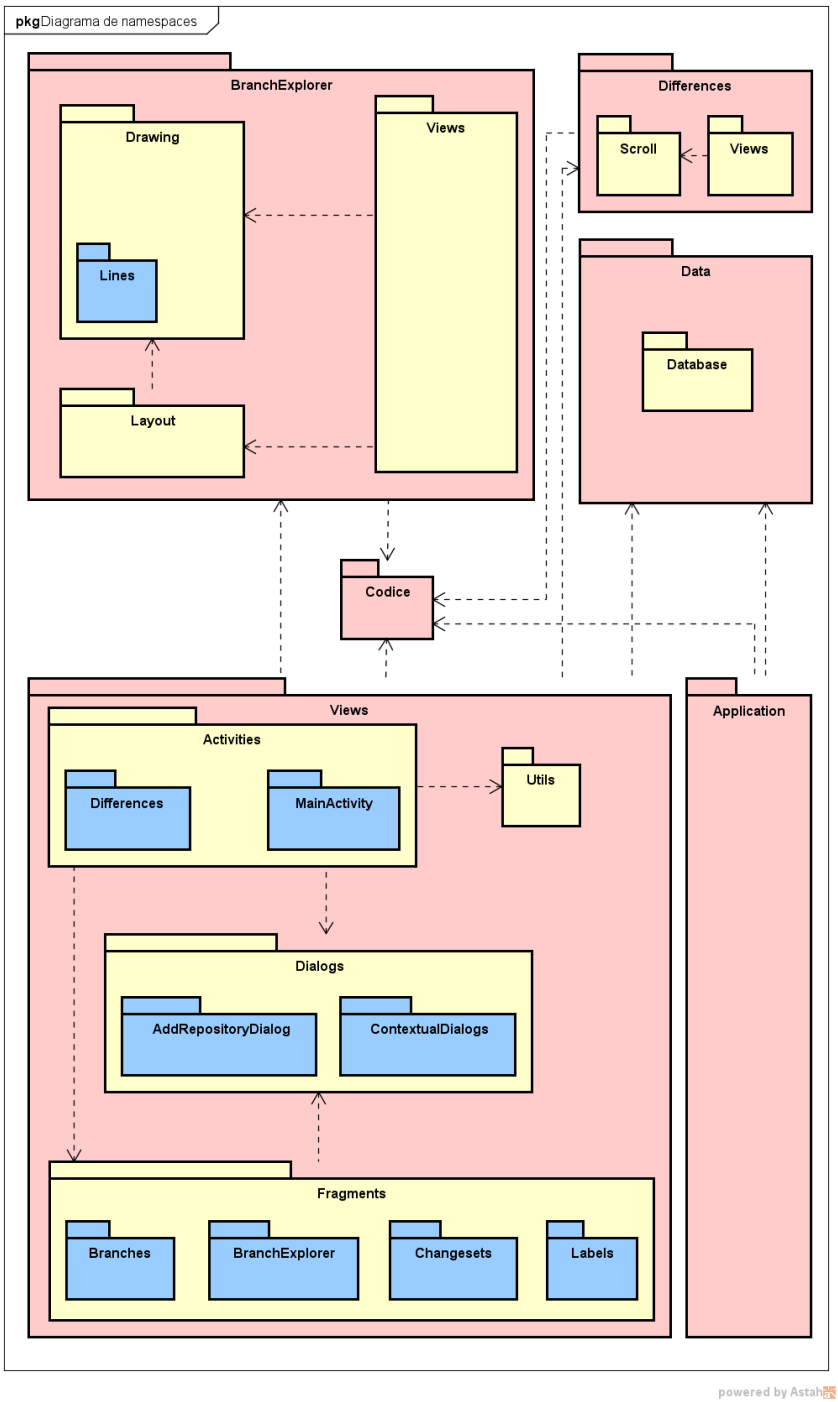

FIGURA 28. DIAGRAMA GENERAL DE *NAMESPACES*

4.2.2 - Diagrama de clases del *namespace* BranchExplorer

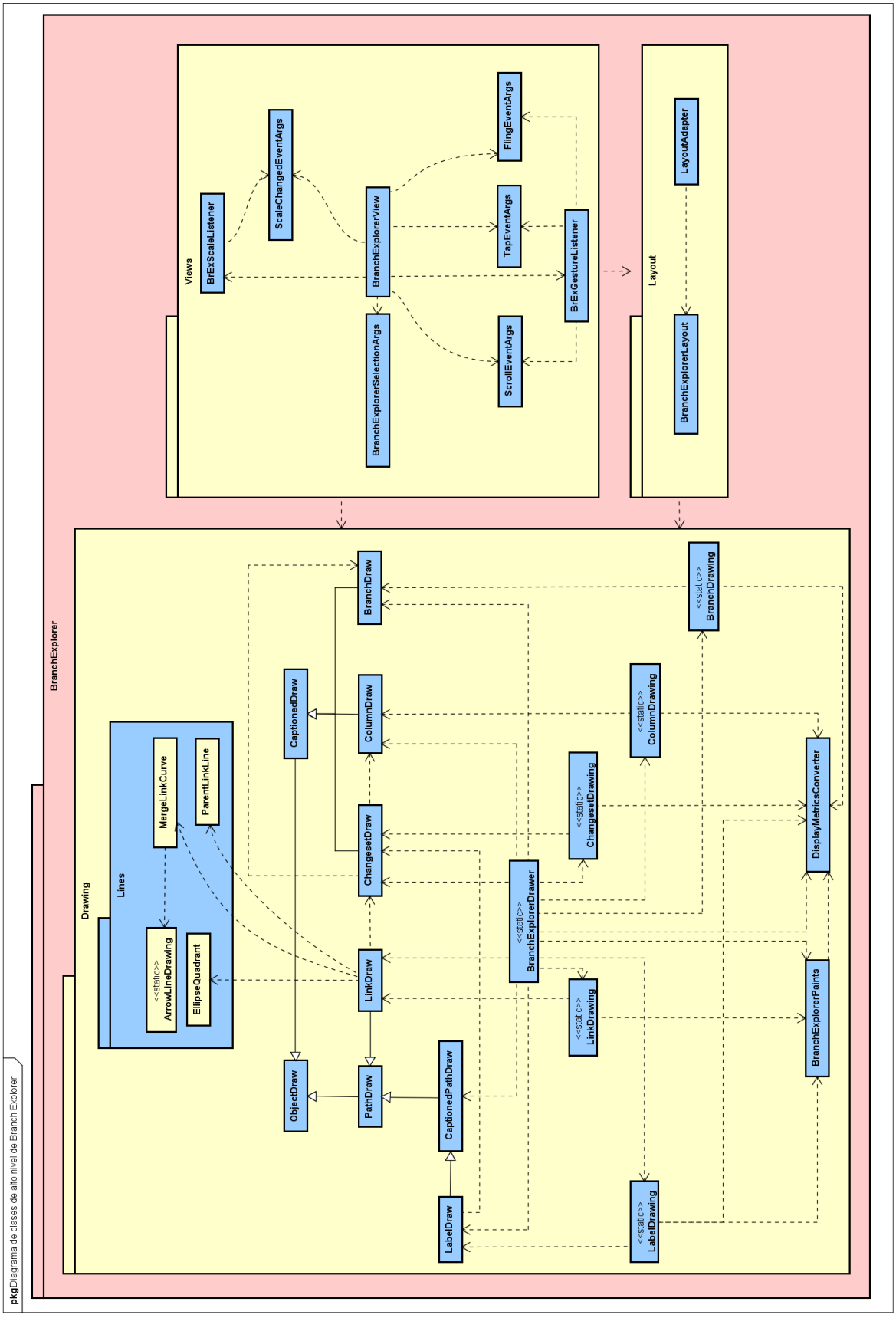

FIGURA 29. DIAGRAMA DE CLASES DEL *NAMESPACE* DROIDPLASTIC.BRANCHEXPLORER

4.2.3 - Diagrama de clases de alto nivel del *namespace* "Differences"

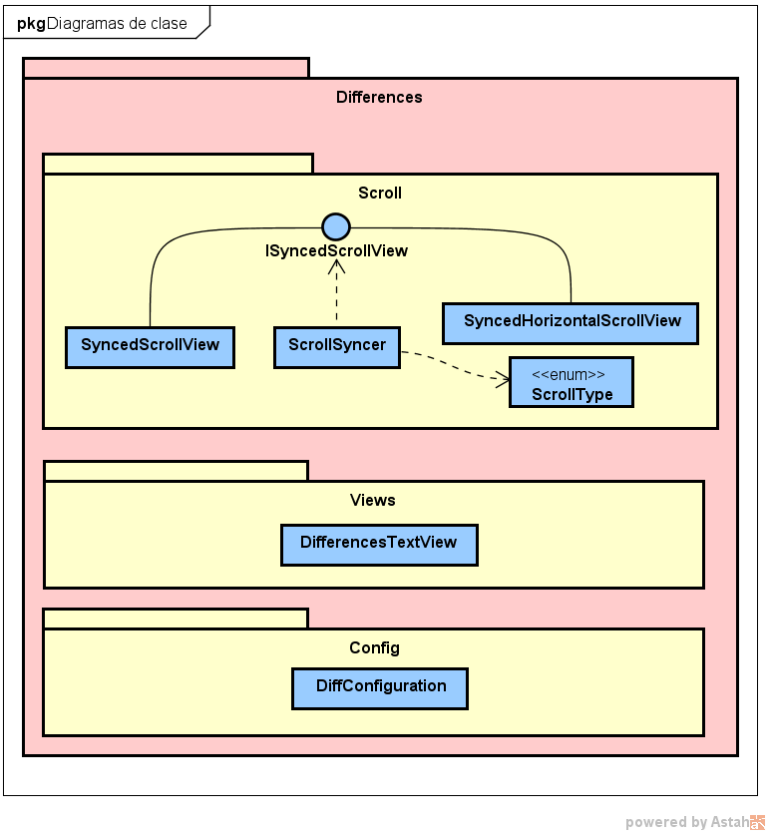

FIGURA 30. DIAGRAMA DE CLASES DEL *NAMESPACE* DROIDPLASTIC.DIFFERENCES

4.2.4 - Diagrama de clases de alto nivel del *namespace* "Data"

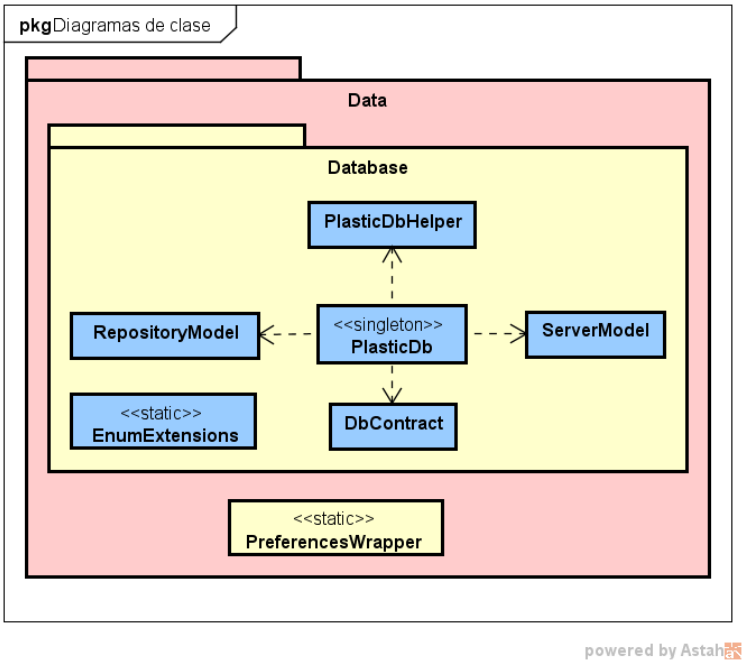

FIGURA 31. DIAGRAMA DE CLASES DEL *NAMESPACE* DROIDPLASTIC.DATA

4.2.5 - Diagrama de clases de alto nivel del *namespace* "Views"

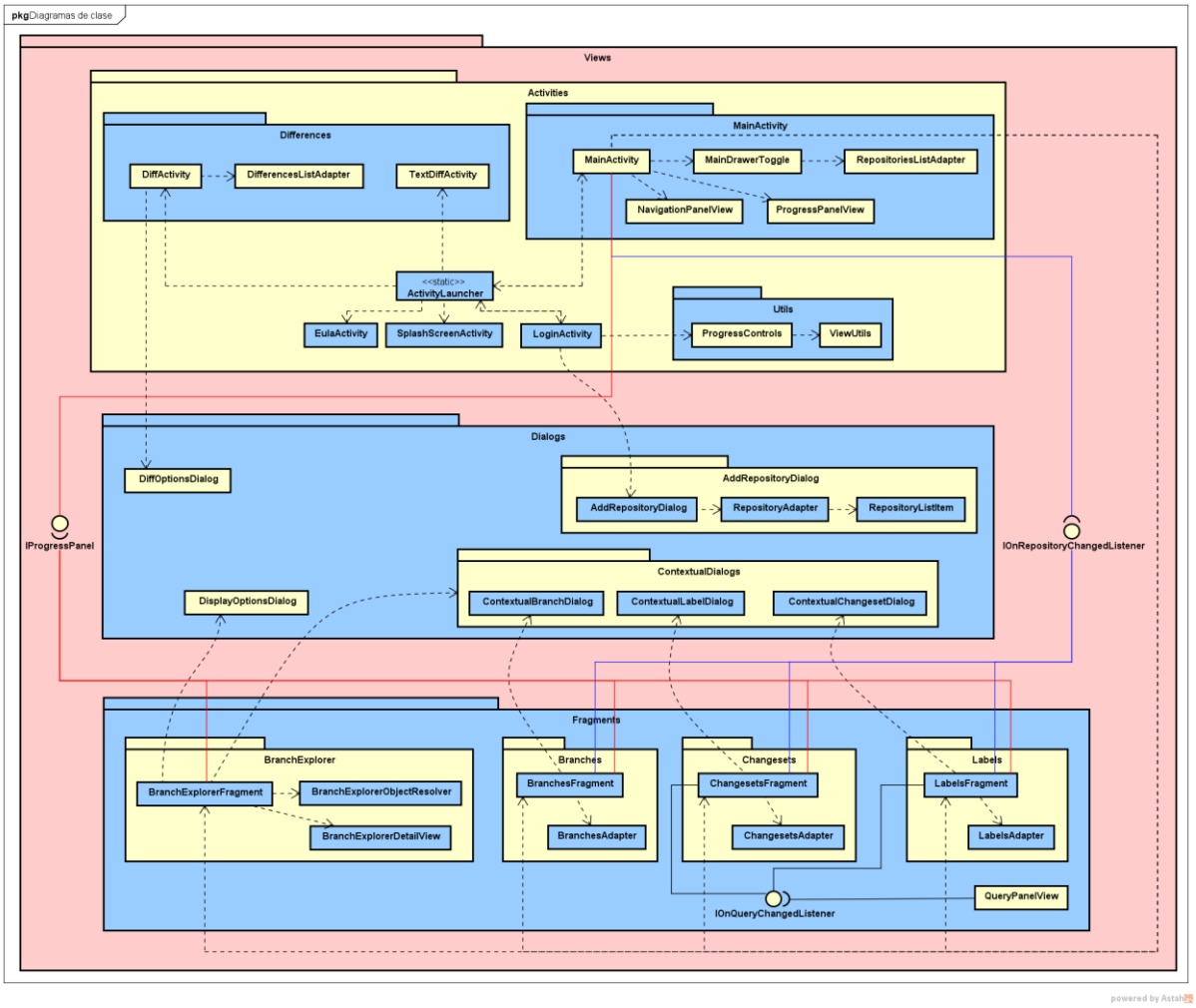

FIGURA 32. DIAGRAMA DE CLASES DEL *NAMESPACE* DROIDPLASTIC.VIEWS

4.2.6 - Diagrama de secuencia “Añadir repositorios del primer servidor”

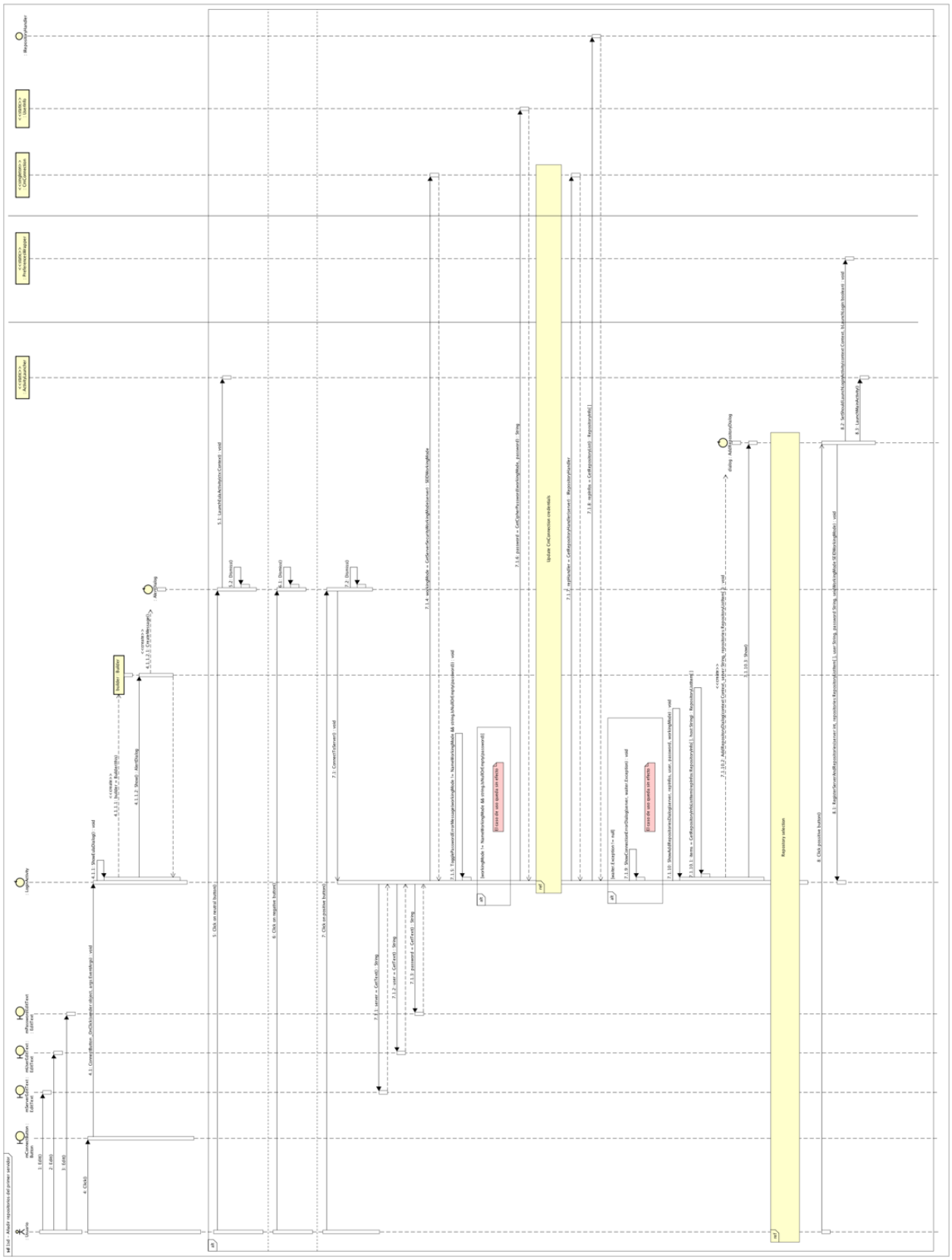

FIGURA 33. DIAGRAMA DE SECUENCIA "AÑADIR REPOSITORIOS DEL PRIMER SERVIDOR"

4.2.7 - Diagrama de secuencia “Cambiar el repositorio de visualización”

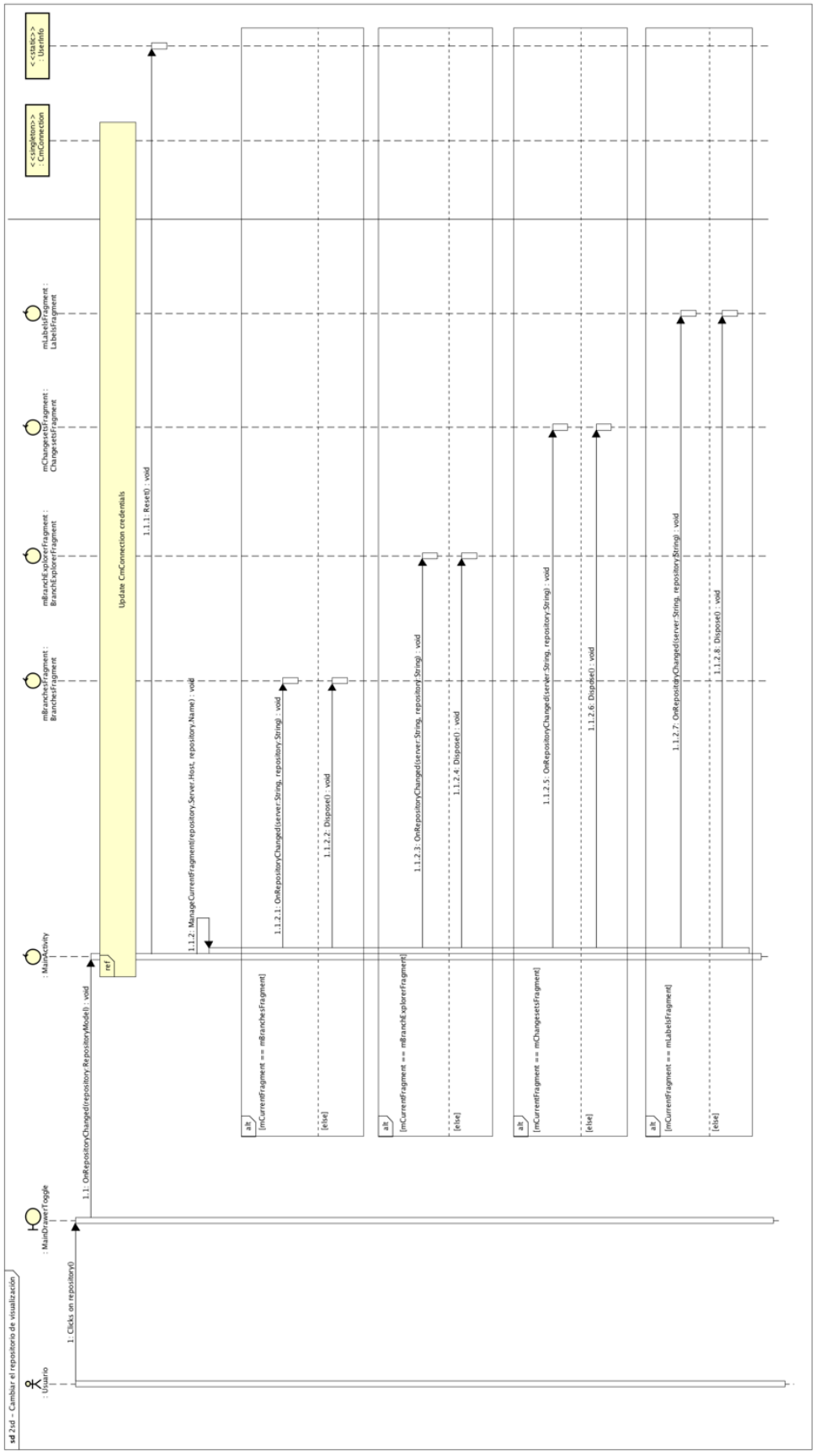

FIGURA 34. DIAGRAMA DE SECUENCIA "CAMBIAR EL REPOSITORIO DE VISUALIZACIÓN"

4.2.8 – Diagrama de secuencia “Visualizar el listado de ramas del repositorio”

El siguiente diagrama de secuencia describe de igual forma “*Visualizar el listado de changesets del repositorio*” y “*Visualizar el listado de labels del repositorio*”.

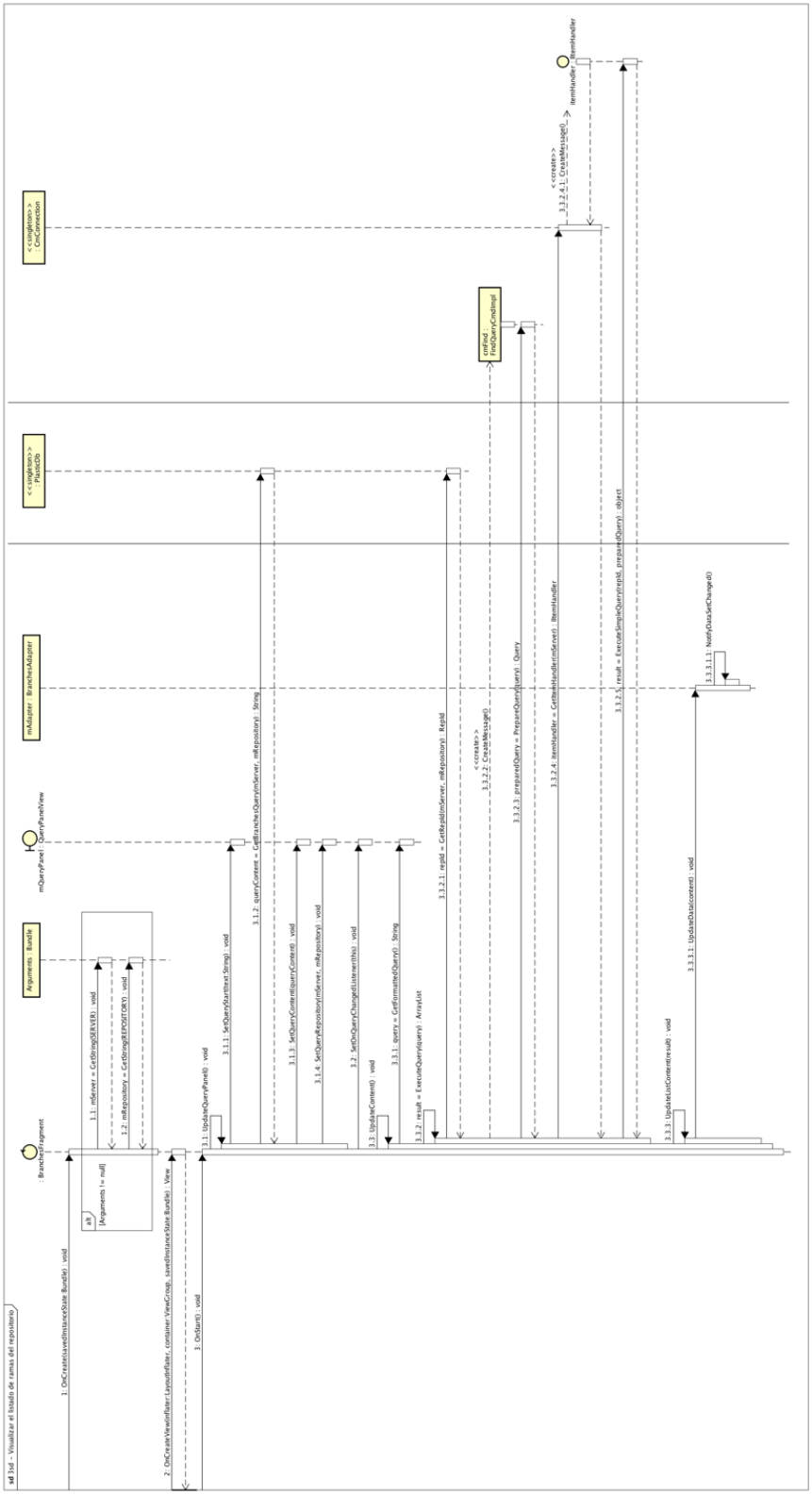

FIGURA 35. DIAGRAMA DE SECUENCIA "VISUALIZAR EL LISTADO DE RAMAS DEL REPOSITORIO"

4.2.9 - Diagrama de secuencia “Visualizar el Branch Explorer del repositorio”

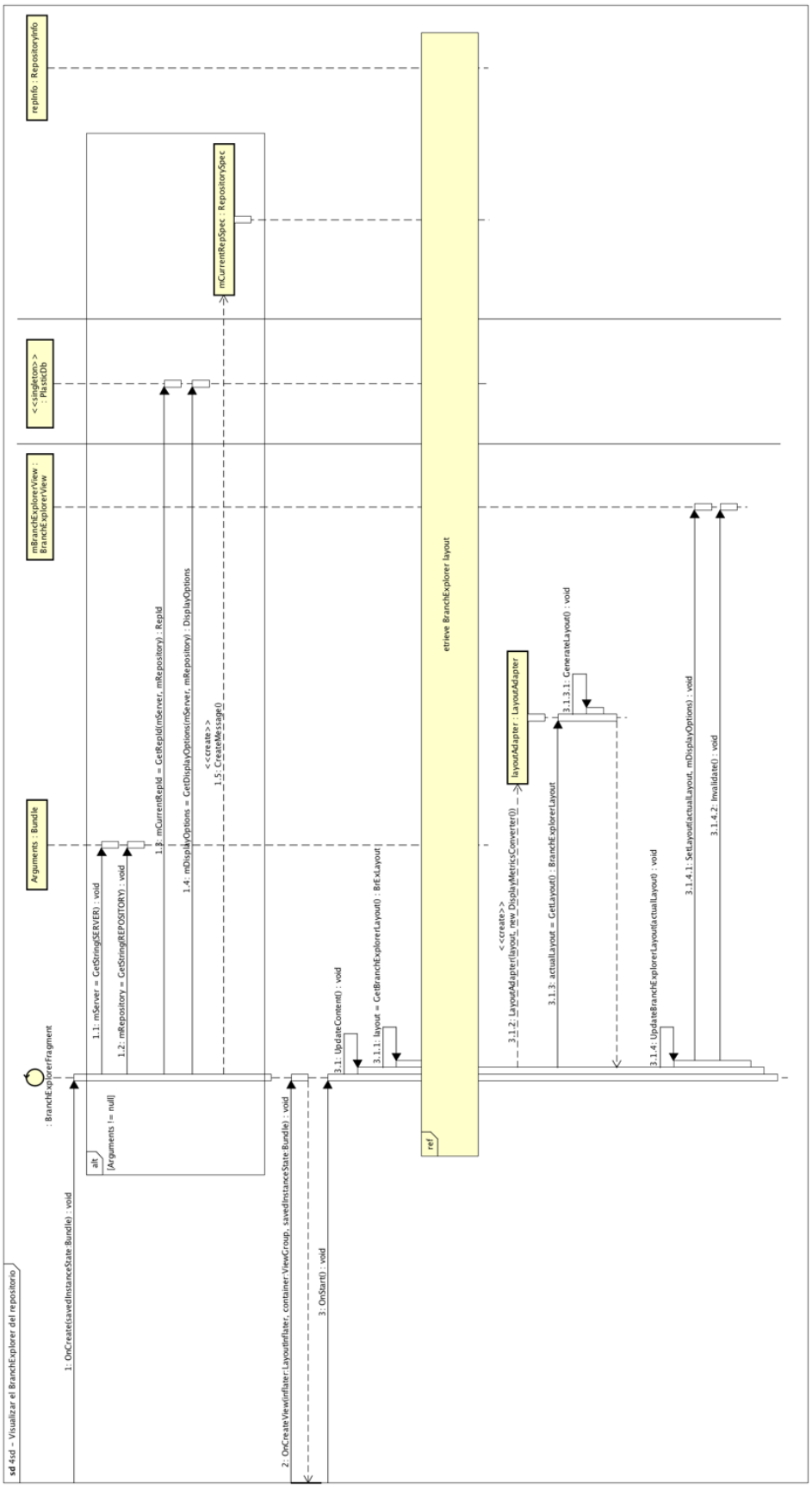

FIGURA 36. DIAGRAMA DE SECUENCIA "VISUALIZAR EL BRANCH EXPLORER DEL REPOSITORIO"

4.2.10 - Diagrama de secuencia “Obtener el listado de diferencias en un punto”

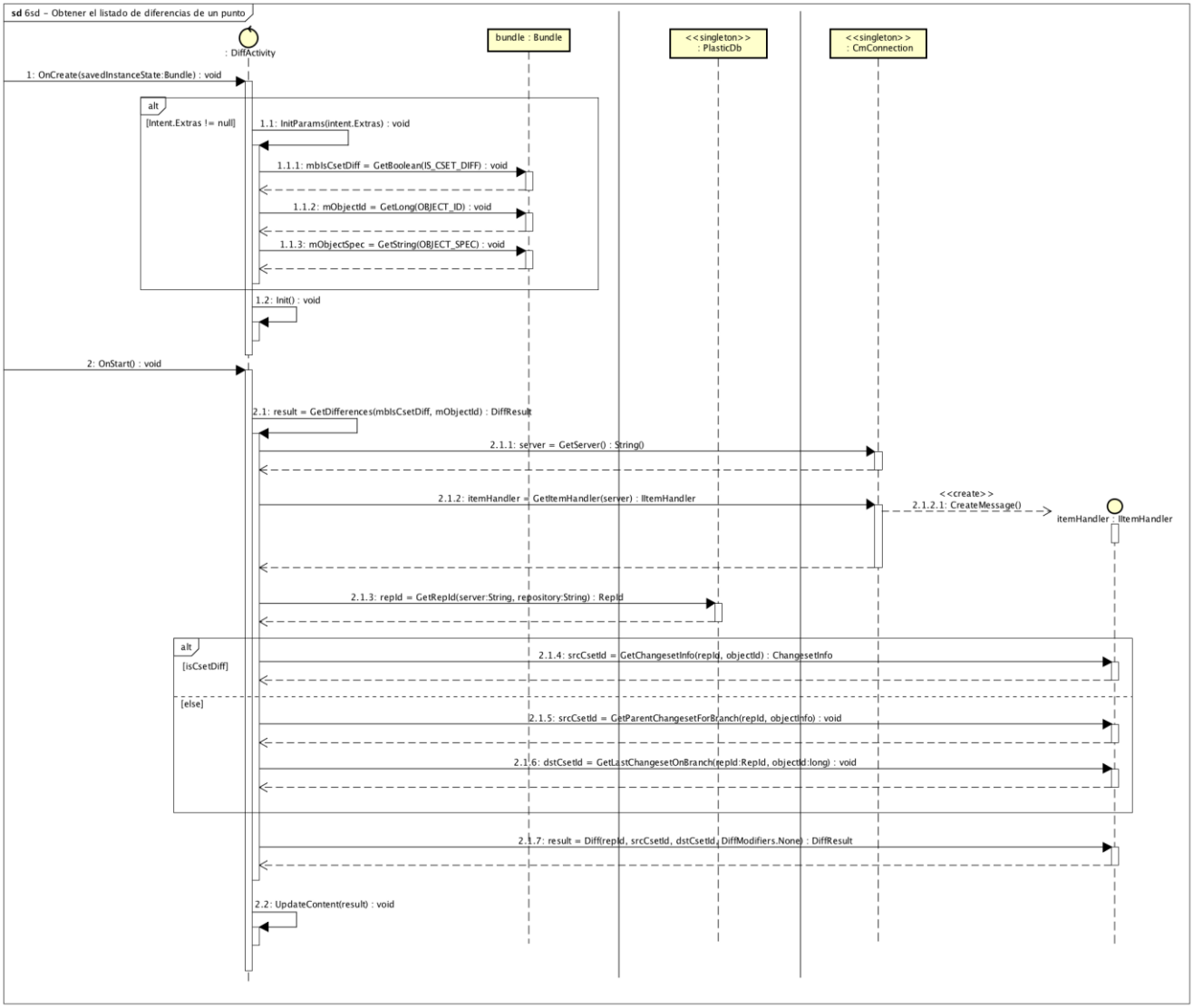

FIGURA 37. DIAGRAMA DE SECUENCIA "OBTENER EL LISTADO DE DIFERENCIAS EN UN PUNTO"

Capítulo V – Implementación

5.1 - El Branch Explorer

El Explorador de Ramas (*Branch Explorer* en adelante) es una parte fundamental de Plastic SCM en los clientes de escritorio. Una de las grandes fortalezas de Plastic SCM es la gestión de ramas (*branching y merging*), favoreciendo metodologías de trabajo tales como *rama por tarea* [7] (o *Task Driven Development* en inglés). En dicha metodología cada desarrollador crea una nueva rama (*branching*) para trabajar de forma independiente y paralela al resto de sus compañeros. Cuando la tarea se puede dar por concluida, esta es integrada de nuevo (*merging*) en otra rama destino, que puede ser o bien la rama principal, o una rama de integración -ya dependerá del modelo de *branching* que siga cada equipo-.

El Branch Explorer es, por lo tanto, el lugar en el que los desarrolladores pueden hacer cosas tales como ver de forma gráfica su progreso en una determinada tarea, desplazarse a puntos anteriores de la misma, crear ramas desde etiquetas marcadas como puntos estables para el desarrollo de sus tareas, y pueden mezclar sus cambios con los de sus compañeros. No es, desde luego, el único punto de la aplicación en el que se pueden llevar a cabo estas acciones. Los distintos listados de *branches*, *changesets* y *labels* permiten hacer esto mismo, pero la cantidad de información y la naturalidad a la hora de ver cómo se relacionan estos elementos entre sí que aporta un diagrama gráfico es uno de los principales atractivos de Plastic SCM.

Es por esto que el Branch Explorer, permitiendo al usuario desplazarse sobre el diagrama con la facilidad que aporta una interfaz táctil, es también uno de los puntos centrales en la aplicación móvil de Plastic SCM.

5.1.1 - Código legado independiente de la plataforma

El Branch Explorer hace su aparición por primera vez en Plastic SCM en la versión 2.0, publicada en el año 2008. El diagrama era visualmente mucho más sencillo de lo que es en la actualidad, pero los elementos más importantes ya estaban representados. *Changesets*, ramas, *labels* y *mergelinks*.

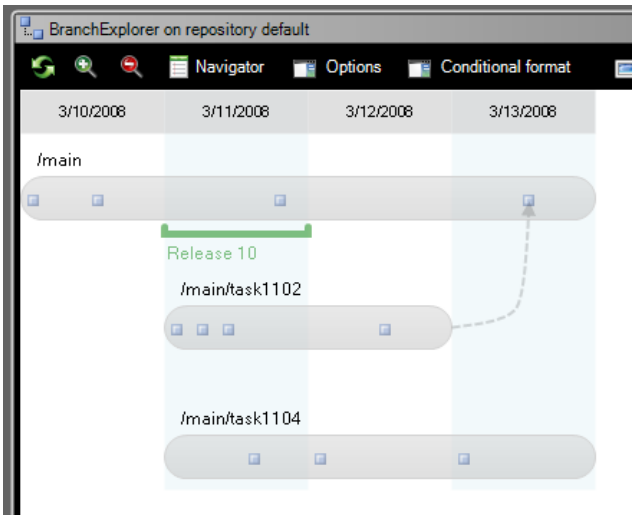

FIGURA 38. PRIMERA VERSIÓN DEL BRANCH EXPLORER

A medida que se fueron sumando al ecosistema de Plastic SCM no solo el cliente de Windows, sino también las extensiones para poder integrarse en distintos entornos de desarrollo (tales como Eclipse o Visual Studio), se pudo llevar a estos el Branch Explorer, ya parte fundamental de la aplicación, gracias a la separación del código de cálculo de la geometría del mismo (en adelante *layout*) del código de dibujado de sus elementos. Un ejemplo de esto es el Branch Explorer en la extensión de Eclipse. El cliente de Plastic SCM, ejecutándose sobre .NET Framework, calcula el *layout* y lo serializa a disco como texto (para evitar problemas con cosas tales como el *endianness* en caso de serializarlo en formato binario). La extensión de Eclipse, ejecutándose sobre la *Java Virtual Machine*, deserializa dicho *layout* y lo dibuja en pantalla.

Dicha serialización a texto se utilizó en las primeras versiones de este Trabajo de Fin de Grado para comprobar que, efectivamente, era posible dibujar el Branch Explorer sin grandes problemas (memoria, velocidad...) en una plataforma móvil. Posteriormente se pudo añadir todo el código de cálculo de *layout* del cliente, dejando de depender de ficheros estáticos para el dibujado del Branch Explorer.

5.1.2 - El Branch Explorer en Android

Se repasan a continuación los puntos más relevantes de la implementación del Branch Explorer en la plataforma Android. Se comenzará con un repaso breve a cómo se construyen interfaces de usuario en dicho sistema operativo, para continuar con un ejemplo sencillo de implementación de una vista personalizada, y terminando con la información específica de cómo se dibuja el Branch Explorer en Android, poniendo especial atención a cómo se han solventado las limitaciones que impone un dispositivo portátil como puede ser una tableta.

El árbol de vistas

Los bloques de alto nivel de interfaz de usuario a la hora de crear una aplicación en Android son las `Activity`. La documentación de Google [8] las define como *una única cosa en la que el usuario se puede centrar al mismo tiempo*, y muestran, por lo tanto, una interfaz con la que el usuario puede interactuar y llevar a cabo una determinada tarea. Como el usuario puede entrar y salir de una `Activity` en cualquier momento -o rotar el dispositivo-, el sistema operativo alertará mediante *callbacks* -que definen el ciclo de vida de una Acitvity- a la instancia que corresponda de estas acciones. Es responsabilidad [9], por tanto, de dicha `Activity`, guardar el estado de la interfaz en el momento en el que el usuario la abandona, y de restaurarlo -si es necesario- cuando el usuario regrese. Un ejemplo típico de esto es una aplicación de email cualquiera. Si el usuario está escribiendo un mensaje, abandona la aplicación para contestar una llamada, y regresa posteriormente a la aplicación, el comportamiento razonable es que la ventana de composición del mensaje siga tal y como se dejó cuando el usuario la abandonó (o, como mínimo, que el mensaje se haya guardado en una carpeta de borradores).

Sin embargo, no es responsabilidad de una `Activity` dibujar su interfaz de usuario, aunque sí lo sea proporcionar el nodo raíz de su jerarquía de *layout*. En última instancia, lo que cualquier `Activity` hace es proporcionar un `Layout`, bien mediante la llamada `setContentView(int layoutResId)`, que hace que el sistema operativo "infle" -*inflate* en inglés- el *layout* que se encuentre descrito en el fichero XML con identificador `layoutResId`, o bien creando el *layout* y cada uno de sus elementos internos mediante código. Así pues, la interfaz de usuario se compondrá de distintos tipos de `Layout`, que se encargarán de posicionar sus elementos en un determinado orden (por ejemplo, una `LinearLayout` con orientación vertical ordena sus

vistas anidadas desde la parte superior de la pantalla hasta la inferior), y de dichas vistas (comúnmente denominados `widgets`), que serán con los que el usuario interaccione.

Podemos ver en el siguiente ejemplo cómo la actividad de *login* de la aplicación se compone, entre otros elementos, de un `LinearLayout` con orientación vertical dentro del cual hay varios `widgets`:

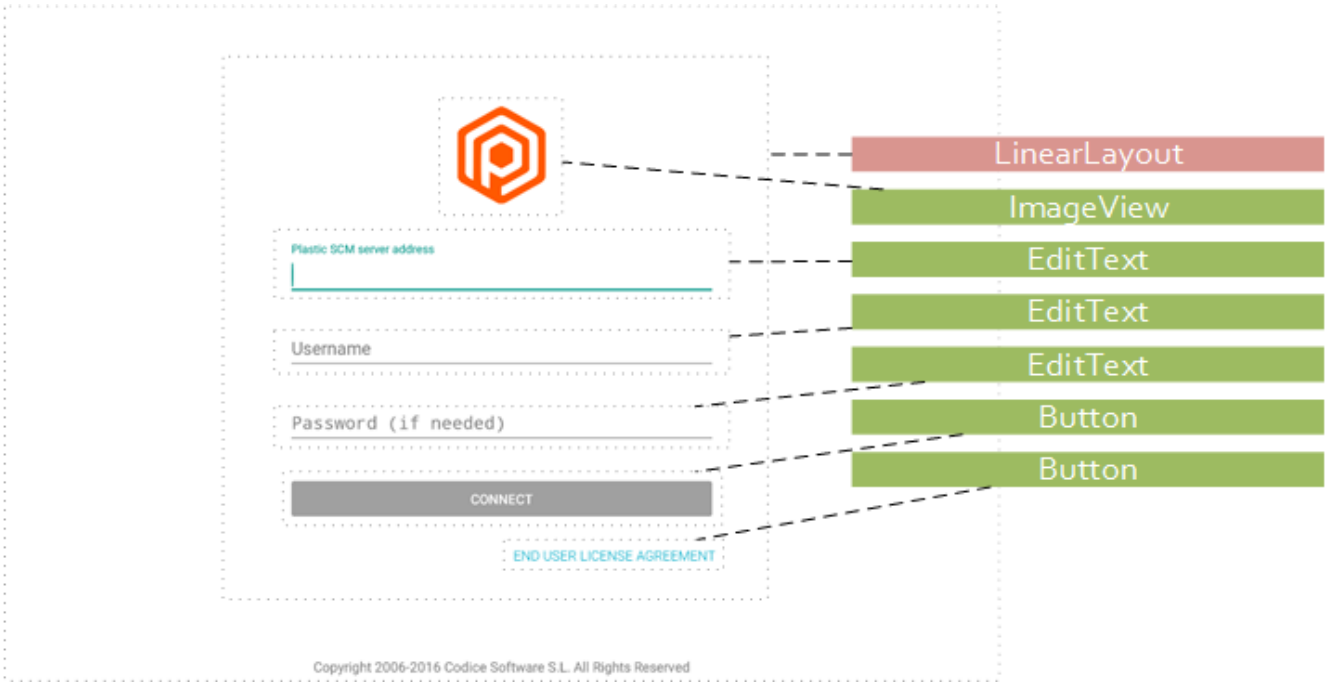

FIGURA 39. EJEMPLO DE ÁRBOL DE VISTAS EN ANDROID

El proceso de dibujado de la interfaz se realiza en dos pasos [10]: uno de medida y otro de *layout*. En el paso de medida, el árbol de vistas se recorre en orden descendente desde la raíz, midiendo el tamaño de cada vista. Una vez finaliza este paso, todas las vistas tienen almacenadas sus dimensiones. En el segundo paso, el de *layout*, cada `ViewGroup` es encargada de posicionar sus hijos en base a las medidas calculadas anteriormente, y cada `View` es responsable de dibujar su contenido en el Canvas que se le pasa como argumento en su método `onDraw(Canvas canvas)`. Este es el método que se deberá sobrescribir para implementar vistas que realicen un dibujado personalizado.

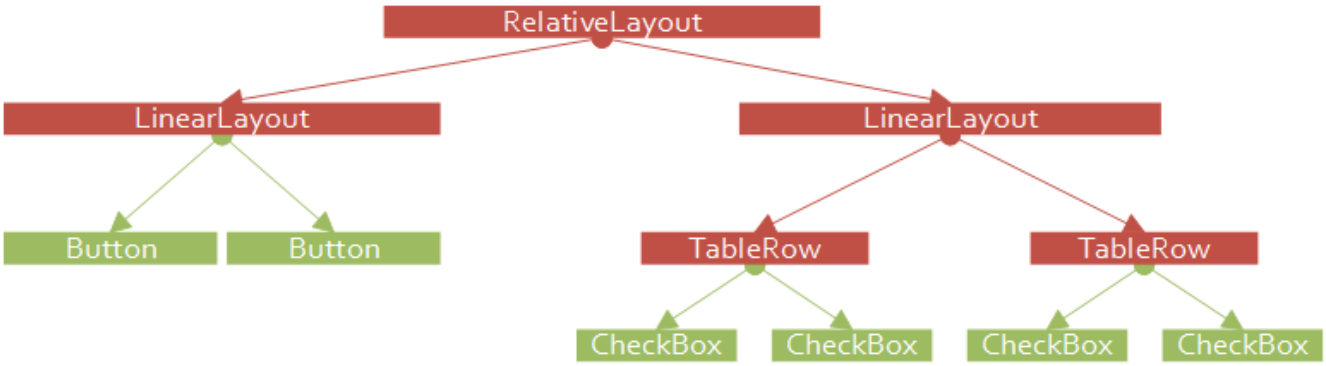

FIGURA 40. EJEMPLO DETALLADO DE ÁRBOL DE VISTAS

A la hora de implementar una vista personalizada, hay dos caminos a seguir. El primero es que dicha `View` no sea más que una agrupación de `Views` ya existentes. Por ejemplo, una `SearchView` no es más que la agrupación de un `EditText` -donde el usuario introduce el término a buscar-, una `ImageView` en la posición de *start* (que varía entre izquierda y derecha dependiendo de la dirección de lectura del idioma en el que esté configurado el dispositivo) con una imagen de lupa, indicando la finalidad, y una segunda `ImageView`, en la posición de *end*, con un símbolo de aspa, para que el usuario pueda borrar de una sola vez todo el contenido del `EditText`. Adicionalmente se puede añadir una `ListView` en la que se muestren sugerencias de búsqueda o un historial.

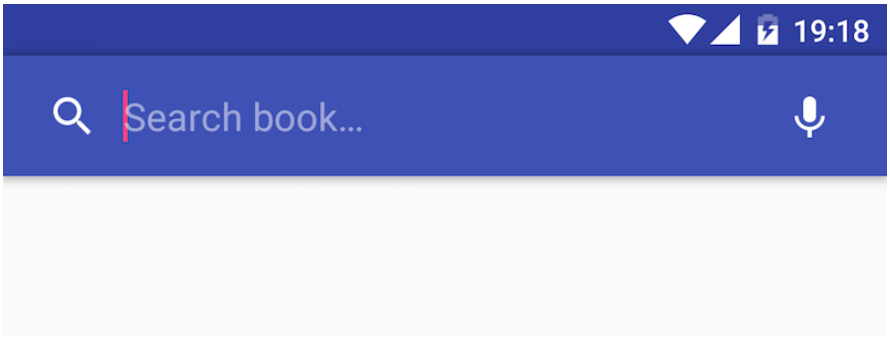

FIGURA 41. EJEMPLO DE UNA VISTA PERSONALIZADA COMPUESTA DE OTRAS VISTAS EXISTENTES

De esta forma, no será necesario en ningún caso sobrescribir el método `onDraw` de dicha vista. Sencillamente se conectarán los eventos de los distintos elementos entre sí para implementar el comportamiento deseado.

El segundo camino pasa por implementar al completo todo el comportamiento de la vista. Manejar las pulsaciones sobre la misma, la dirección que deban llevar los textos, los movimientos de *scroll* y *zoom*, etc. Esta segunda forma es mediante la que se ha implementado el Branch Explorer en Android, y es la que requiere sobrescribir el ya mencionado método `onDraw(Canvas canvas)`.

Los cambios en la interfaz de usuario se ven reflejados en la información del estado que cada vista almacena de sí misma, y este a su vez se ve reflejado cuando la vista dibuja su contenido sobre un Canvas. Buscando un ejemplo concreto, pasemos a describir cómo se realizan estos pasos con un `Button`. El botón tendrá, habitualmente, uno o más colores dependiendo de su estado (si está habilitado o deshabilitado, enfocado, pulsado, etc). También tendrá un texto indicativo de qué hace ese botón, y, en las últimas versiones de Android (posteriores a 5.0 *Lollipop*), una animación de *ripple*, que no es más que una animación de un círculo con centro el lugar de una pulsación, que crece hasta ocupar la totalidad del fondo para posteriormente desvanecerse.

A la hora de dibujarse, dicho botón tendrá que hacer, en líneas generales, las siguientes operaciones:

1. Escoger un color de fondo según su estado.
2. Dibujar sobre el canvas un rectángulo que indique cuáles son los bordes en los que el botón va a responder a eventos.
3. Dibujar sobre el canvas, centrado en los bordes en los que va a responder a eventos, el texto que el programador le ha dado.

Y cuando un usuario pulsa dentro de los límites de la vista del botón, este tendrá que hacer lo siguiente:

1. Calcular si el punto `(x, y)` en el que el usuario ha tocado se encuentra dentro de los bordes en los que el botón responde a eventos (que puede no ser el caso si el botón tiene margen interno). En caso negativo, no sería necesario redibujar la vista. En caso afirmativo, la vista tiene que ser invalidada para que sea dibujada de nuevo -es el *framework* quien llama al método `onDraw`, no la propia vista-
2. Dibujar sobre el canvas el rectángulo que indica los bordes del botón.
3. Dibujar sobre el canvas, con centro el lugar de pulsación del usuario, el comienzo de la animación de *ripple*.
4. Dibujar sobre el canvas, centrado en los bordes en los que el botón responde a eventos, el texto que el programador le ha asignado.
5. Si tiene algún *listener* de evento de pulsación, avisarle de que el botón ha sido pulsado.
6. Mientras la animación de *ripple* no haya terminado, invalidarse de nuevo para que el código de dibujado se ejecute, avanzando en cada paso la animación de *ripple*, hasta que esta termine.

Así, el método `onDraw` de los widgets incluidos en el *framework* de Android termina siendo ejemplo de implementaciones largas, llenas de comprobaciones de casos *esquina,* como por ejemplo si el *layout* se ha de dibujar de izquierda a derecha o viceversa (dependiendo del país), si el texto que se va a dibujar entra completo o es necesario truncarlo, en caso de truncarlo en qué posición se deben colocar las elipsis, etc.

Conceptos básicos y vocabulario común

Un `Canvas` es, de acuerdo a la documentación de Google [11], un objeto que *almacena comandos de dibujado*. Define métodos para dibujar sobre él primitivas tales como texto, figuras geométricas, o mapas de bits. Es decir, en un canvas se indica *qué* debe ser dibujado. Tiene su origen de coordenadas en la esquina superior izquierda, con el eje de abscisas creciendo hacia el lateral derecho hasta el ancho de la vista -*width*- y el eje de ordenadas creciendo hacia la parte inferior hasta el alto de la vista -*height*-.

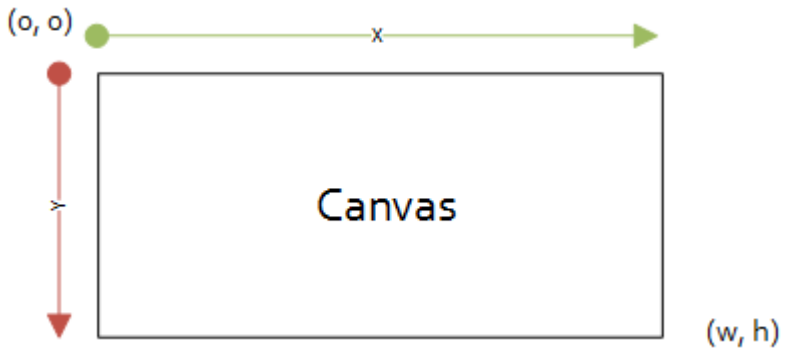

FIGURA 42. SISTEMA DE COORDENADAS EN ANDROID.GRAPHICS.CANVAS

De forma complementaria, un objeto `Paint` almacena información sobre *cómo* debe ser dibujada una primitiva. Entre otras cosas, `Paint` proporciona información sobre el color de relleno (*fill*) y de borde (*stroke*) con el que se va a dibujar una primitiva, si se deben utilizar técnicas de *antialiasing*[4], o cuál debe ser el espaciado entre letras.

[4] Técnica software para disminuir efectos de bordes cortantes allí donde deberían ser suaves

```
public override void OnDraw(Canvas canvas)
{
    Paint fillPaint = new Paint();
    fillPaint.SetStyle(Paint.Style.Fill);
    fillPaint.Color = Color.Blue;

    Paint strokePaint = new Paint();
    strokePaint.SetStyle(Paint.Style.Stroke);
    strokePaint.Color = Color.Black;

    RectF rectangle = new RectF(30, 30, 90, 90);
    canvas.DrawRect(rectangle, fillPaint);
    canvas.DrawRect(rectangle, strokePaint);
}
```

FIGURA 43. EJEMPLO BÁSICO DE DIBUJADO SOBRE UN CANVAS

Por ejemplo, para dibujar sobre un canvas un cuadrado, se necesitará, además del propio canvas, un `RectF` (rectángulo con coordenadas especificadas en punto flotante) que defina la posición del objeto, y dos objetos `Paint`, uno para el relleno de la forma y otro para el borde (o reutilizar el mismo cambiando su `Style` y su `Color`)

La pieza de código que se puede leer en la *figura 43*, una vez ejecutada, dibujará sobre el canvas un cuadrado cuya esquina superior izquierda se encontrará en las coordenadas `(30, 30)` del canvas, cuyo lado derecho terminará en el `X=90`, y cuyo lado inferior termine en el `Y=90` (teniendo, por lo tanto, un ancho y alto de 60).

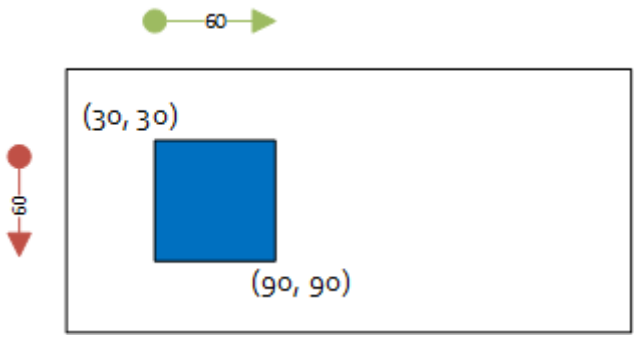

FIGURA 44. EJEMPLO DE UN CANVAS CON UN DIBUJO

Hasta ahora no se han mencionado las unidades en las que trabaja el canvas. En los últimos años, distintos fabricantes han ido mejorando la tecnología con las que fabrican las pantallas de teléfonos móviles y tabletas, aumentando la densidad de *píxeles por pulgada* (*dpi*, *dots per inch*) hasta el punto en el que el ojo humano es incapaz de distinguirlos de forma individual a la distancia habitual de uso de este tipo de dispositivos. Así, Android adoptó una nueva unidad de medida [12] para facilitar a los programadores el maquetado de interfaces, el *dp*, o *density-independent pixel*. Esta medida se define como el equivalente a un píxel físico en una pantalla de 160 píxeles por pulgada, de forma que, si los *dpi* aumentan, el *dp* escalará proporcionalmente ocupando más píxeles reales según sea necesario para ocupar el mismo tamaño físico.

$$real\ pixels = density\ independat\ pixels \times {}^{dots\ per\ inch}/_{160}$$

Todas las medidas que se definan en ficheros de layout XML estarán en *dp*. Sin embargo, no es el canvas quien hace la conversión de *dp* a píxeles reales. Este trabaja en píxeles, y, por lo tanto, al dibujar la vista personalizada habremos de hacernos cargo de dicha transformación para garantizar que la `View` se ve igual de bien en cualquier dispositivo, sea cual sea la densidad de píxeles por pulgada de su pantalla.

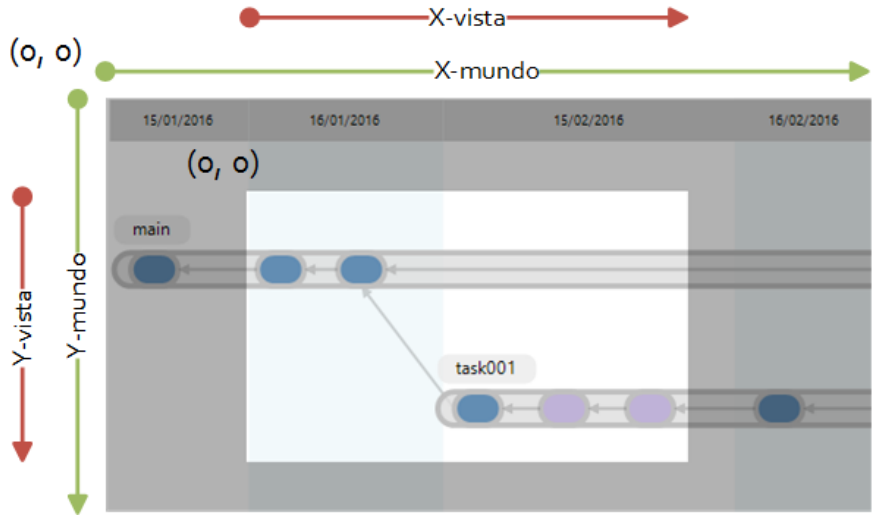

FIGURA 45. VIEWPORT

Otro concepto que es necesario definir antes de continuar es el de *viewport* o área de clip. En Plastic SCM no hay limitación del tamaño del Branch Explorer. Su punto `(0, 0)` se situará -aproximadamente- en el primer *changeset* del repositorio, pudiendo crecer a lo ancho y a lo alto de manera infinita. Sin embargo, en Android, la vista sobre la que lo dibujemos sí tendrá un ancho y un alto finito, por lo que habrá que distinguir entre las coordenadas *de mundo* (que son las del Branch Explorer) y coordenadas *de vista*, que son las de la vista sobre la que el Branch Explorer se dibuja. Este concepto de Viewport tendrá una importancia capital a la hora de optimizar el dibujado del Branch Explorer -como más adelante se señala en la sección La importancia de la velocidad de dibujado-, así como a la hora de hacer *scroll* y *fling*.

Una de las cosas que se pierden a la hora de hacer una implementación personalizada de una vista es la facilidad que nos otorgan herramientas ya existentes en el *framework* como `ScrollView` u `HorizontalScrollView` para manejar el *scroll* y el *fling* del contenido.

El *scroll* se define como la acción de mover el *viewport* a través del contenido mediante un gesto de arrastre [13]. El *scroll* cambiará la distancia entre el origen de coordenadas del *viewport* y el origen de coordenadas del mundo a la misma velocidad y en la misma medida que el usuario arrastre su dedo sobre la pantalla -y de manera proporcional al *zoom*-. Al movimiento de *scrolling* simultáneo sobre los ejes X e Y se lo denomina *panning*.

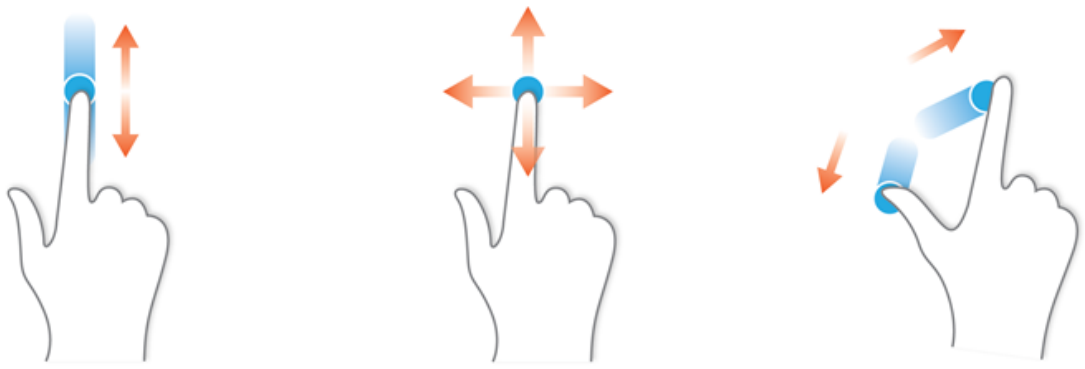

FIGURA 46. GESTOS DE *SCROLL*, *PAN*, Y *UNPICH TO ZOOM*

El *fling* -o *flick* dependiendo de la fuente que se consulte- es un tipo especial de *scroll* que sucede cuando el usuario arrastra rápidamente su dedo en la pantalla para, posteriormente, levantarlo. El *fling* causará que el movimiento de *scroll* siga sucediendo durante cierto tiempo a pesar de que no se esté realizando ningún gesto en la pantalla, decelerando hasta que el *viewport* se deja de mover sobre el contenido.

La deceleración del gesto de *fling* se calcula mediante las llamadas *easing equations*, que proporcionan al movimiento una inercia acorde a la velocidad con la que el usuario estaba desplazando el *viewport* cuando cesó el gesto. Para asegurar que cada desarrollador no ha de implementar sus propias *easing equations*, proporcionando inercias incoherentes entre aplicaciones para el gesto de *fling*, el *framework* de Android proporciona una clase llamada `Scroller`. Un `Scroller` se inicializa con datos relativos a la posición actual del *viewport*, la velocidad del gesto de *fling*, y las posiciones `X` e `Y` máximas a las que puede llegar el *viewport* relativas al mundo. El `Scroller` no se encarga de realizar el *scroll* automáticamente a la vista, pero sí proporciona nuevos valores de `X` y de `Y` cada vez que se le pida que los calcule, en función del tiempo que haya pasado desde la última vez que los actualizó. De esta forma, en el método `onDraw` se puede calcular el desplazamiento producido por un gesto de *fling* de forma coherente con el resto de las aplicaciones que un usuario pueda tener instaladas.

Implementación de una vista sencilla

Una vez se han definido los conceptos necesarios, se puede proceder a la implementación de una vista sencilla que ayude a seguir la implementación real del Branch Explorer. Nos detendremos en los puntos clave para proporcionar a nuestro ejemplo de movimientos de scroll, fling, y zoom. La funcionalidad será limitada, simplemente se dibujará en pantalla un círculo que se podrá desplazar con el dedo.

Una vista recibe eventos de toque por parte del usuario en el método `OnTouchEvent(MotionEvent e)`. El objeto `MotionEvent` recibido como parámetro proporciona, entre otros datos, las coordenadas `X` e `Y` relativas a la vista donde ha sucedido el toque, o la cantidad de dedos -*punteros*- involucrados en el gesto. Es necesario, pues, sobrescribir este método para poder implementar gestos en la vista. Sin embargo, no será necesario implementar los cálculos pertinentes para identificar y manejar ciertos gestos (cálculos tales como el cambio de distancia entre dos *punteros* para obtener el cambio en el factor de escalado que habrá que aplicar a la vista para tener zoom). El *framework* de Android proporciona dos clases *helper* para esta tarea [14], que, a partir de los `MotionEvent` que les sean suministrados, hacen los cálculos pertinentes y notifican de los eventos

identificados a los *listeners* que tengan registrados. Para los gestos de *scroll* y *fling*, será la clase `GestureDetector` [15] quien se encargue de hacer la detección, siendo necesario extender de la clase `SimpleOnGestureListener` [16] para ser notificados de los gestos detectados. De manera complementaria, `ScaleGestureDetector` [17] detectará gestos de escalado con dos o más dedos, y notificará de ello a su *listener*, que extenderá `SimpleOnScaleGestureListener` [18].

```
internal class ExampleGestureListener : GestureDetector.SimpleOnGestureListener
{
    public override bool OnDown(MotionEvent e)
    {
        return true;
    }

    public override bool OnScroll(MotionEvent e1, MotionEvent e2, float distanceX,
        float distanceY)
    {
        mScrollDetectedEvent?.Invoke(distanceX, distanceY);
        return true;
    }

    public override bool OnFling(MotionEvent e1, MotionEvent e2, float velocityX,
        float velocityY)
    {
        mFlingDetectedEvent?.Invoke(velocityX, velocityY);
        return true;
    }

    public override bool OnSingleTapConfirmed(MotionEvent e)
    {
        mShortPressDetectedEvent?.Invoke(e.GetX(), e.GetY());
        return true;
    }

    public override void OnLongPress(MotionEvent e)
    {
        mLongPressDetectedEvent?.Invoke(e.GetX(), e.GetY());
    }
}
```

FIGURA 47. EJEMPLO DE IMPLEMENTACIÓN DE SIMPLEONGESTURELISTENER

El motivo de devolver `true` en el método `OnDown` del *listener* es para indicar que será esa instancia específica de `SimpleOnGestureListener` quien se podrá hacer cargo del gesto. Si existiese para la vista más de un *listener* -por ejemplo, dependiendo del área donde se haya realizado el toque-, en ese método se haría la comprobación de las coordenadas para saber si puede encargarse de manejar el gesto. Si se devuelve `false`, ningún método más del *listener* será llamado.

El resto de métodos de `SimpleOnGestureListener` serán llamados por el `GestureDetector` a medida que el gesto que representan sea detectado. La implementación de este listener es sencilla, simplemente se invocan[5] los eventos registrados con los parámetros relevantes. Cabe señalar que `OnSingleTapConfirmed` será llamado cuando el `Detector` pueda asegurar que se ha realizado un gesto de *tap* único (es decir, un gesto

[5] El operador "?" es llamado de "propagación de nulos" (*null propagation operator*), y está incluido en la definición de C# 6. El evento se invoca únicamente si la referencia al mismo no es nula. Es el equivalente de "`if (mEvent != null) mEvent(arg1, arg2)`"

de tocar la pantalla y levantar rápidamente el dedo). Introduce cierto retraso al esperar un segundo *tap*, dependiendo de la velocidad de doble *tap* configurada en los ajustes de accesibilidad del sistema, para permitir -por ejemplo, a personas con movilidad reducida- ejecutar el gesto a una velocidad más lenta de lo normal.

```
internal class ExampleScaleListener : ScaleGestureDetector.SimpleOnScaleGestureListener
{
    internal float ScaleFactor { get; set; } = 1F;

    internal float FocusX { get; set; }

    internal float FocusY { get; set; }

    internal ExampleScaleListener(float minScaleFactor, float maxScaleFactor)
    {
        mMinScaleFactor = minScaleFactor;
        mMaxScaleFactor = maxScaleFactor;
    }

    public override bool OnScale(ScaleGestureDetector detector)
    {
        ScaleFactor *= detector.ScaleFactor;
        FocusX = detector.FocusX;
        FocusY = detector.FocusY;

        ScaleFactor =
            Math.Max(mMinScaleFactor, Math.Min(mMaxScaleFactor, ScaleFactor));

        mScaleChangedEvent?.Invoke();

        return true;
    }
}
```

FIGURA 48. EJEMPLO DE IMPLEMENTACIÓN DE SIMPLEONSCALEGESTURELISTENER

Aunque podría ser la propia vista quien manejase el factor de zoom, puede ser `SimpleOnScaleGestureListener` quien asegure que el factor de escalado se encuentra entre los límites deseados, (siendo `1F` sin escalado, `0.5F` un escalado a la mitad del tamaño real, y `2F` un escalado al doble del tamaño real), simplificando posteriormente el código de la vista. Las propiedades `FocusX` y `FocusY` permitirán que el zoom ocurra alrededor del centro de los dedos involucrados en el gesto, y no en otro punto como el centro de la vista o el origen de coordenadas.

Para que una vista personalizada pueda ser dispuesta en un Layout, se habrá de implementar, como mínimo, el constructor que recibe únicamente como parámetro `Context`[6] [19]. Sin embargo, si la vista no se va a crear mediante código, sino que va a estar definida en un fichero XML, también será necesario sobrescribir el constructor que, además del `Context`, recibe un `IAttributeSet` como parámetro (que será un *Set* con las propiedades de la vista definidas en el XML). Como cualquiera de estos dos constructores puede ser llamado -dependiendo de cómo se vaya a disponer la vista-, lo ideal es crear un método que inicialice los miembros

[6] `Context` es una clase abstracta, implementada por el sistema, que provee información sobre el estado actual de la aplicación, y que es necesaria para acceder a recursos tales como aquellos definidos en ficheros XML (*strings*, dimensiones, colores…), o para lanzar Activities, entre otras. Toda Activity es una subclase de Context y puede actuar como tal.

necesarios, y que sea llamado desde todos los constructores que se implementen. En este ejemplo únicamente se implementará el segundo constructor mentado.

```
public class ExampleView : View
{
    public ExampleView(Context context, IAttributeSet attrs) : base(context, attrs)
    {
        Init(context);
    }

    void Init(Context context)
    {
        mScaleListener = new ExampleScaleListener(MIN_SCALE_FACTOR, MAX_SCALE_FACTOR);
        mScaleDetector = new ScaleGestureDetector(context, mScaleListener);

        mScaleListener.ScaleChanged += OnScaleChanged;

        mScroller = new Scroller(context);

        mGestureListener = new ExampleGestureListener();
        mGestureDetector = new GestureDetector(context, mGestureListener);

        mGestureListener.FlingDetected += OnFling;
        mGestureListener.ScrollDetected += OnScroll;
        mGestureListener.ShortPressDetected += OnShortPress;
        mGestureListener.LongPressDetected += OnLongPress;

        mCanvasClipBounds = new Rect();
        mCurrentViewPort = new RectF();
    }
}
```

FIGURA 49. EJEMPLO DE IMPLEMENTACIÓN DE UNA SUBCLASE DE ANDROID.VIEWS.VIEW

El miembro `mCurrentViewPort` será el *viewport* de la vista. `mCanvasClipBounds` será un miembro auxiliar que permitirá recuperar del Canvas su traslación respecto a su origen de coordenadas -más sobre esto en breve-.

Para mantener la posición X e Y actual del mundo respecto a la vista, se usarán dos variables auxiliares, `mPosX` y `mPosY`, que serán actualizadas donde sea necesario -esto es, en los métodos de *scroll* y a la hora de actualizar el avance del movimiento de *fling* actual-.

```
void OnScroll(float distanceX, float distanceY)
{
    mPosX -= distanceX/mScaleListener.ScaleFactor;
    mPosY -= distanceY/mScaleListener.ScaleFactor;

    Invalidate();
}
```

FIGURA 50. EJEMPLO DE IMPLEMENTACIÓN DE *SCROLL*

La clase `GestureListener` notifica del *scroll* en términos de diferencias de X y de Y respecto a la última notificación (`distanceX` y `distanceY`). El motivo de restar las deltas a las variables auxiliares es porque lo

deseado es mover el *viewport* en dirección contraria al gesto de *scroll*. Es decir, si el usuario arrastra el dedo de derecha a izquierda, el movimiento deseado del *viewport* sobre el mundo es de izquierda a derecha.

Para que el desplazamiento sea acorde al nivel de zoom, es necesario corregir las distancias con el factor de escalado. En caso contrario, cuando el factor de escalado fuese menor que 1, el *viewport* se desplazaría sobre el contenido a una velocidad menor que el gesto de *scroll*, y viceversa para factores mayores que 1.

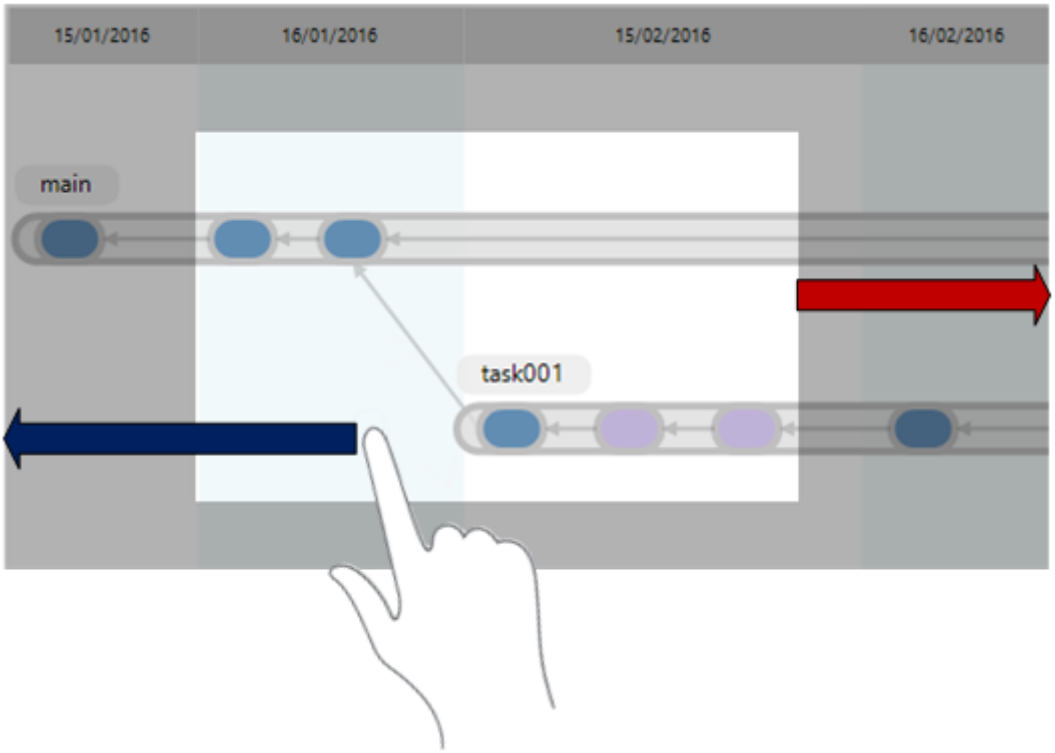

FIGURA 51. MOVIMIENTO DE *SCROLL* DEL *VIEWPORT* SOBRE EL CONTENIDO

```
void OnScaleChanged()
{
    Invalidate();
}
```

FIGURA 52. EJEMPLO DE IMPLEMENTACIÓN DE ONSCALECHANGED

Cuando el zoom de la vista cambie, lo único necesario será llamar a `Invalidate`, que *invalidará* la vista -es decir, la marcará como *sucia* o desactualizada-, forzando el redibujado de la misma. El escalado del Canvas se hará directamente en el método `OnDraw`. Cuando se detecte un gesto de *fling*, se inicializará el `Scroller` con los parámetros necesarios para que, cada vez que se llame sobre `Scroller.ComputeScrollOffset`, este proporcione unos puntos X e Y válidos. Si se quisiese limitar el gesto de *fling* (por ejemplo, si el *mundo* es una fotografía de tamaño finito con un ancho y alto de 500 píxeles), se haría de la siguiente forma:

```
void OnFling(float velocityX, float velocityY)
{
    mScroller.Fling((int) mPosX, (int) mPosY, (int) velocityX, (int) velocityY, 0, 500,
0, 500);

    Invalidate();
}
```

FIGURA 53. EJEMPLO DE IMPLEMENTACIÓN DE *FLING*

Así, el *fling* queda limitado por la izquierda en `X=0`, por la derecha en `X=500`, por arriba en `Y=0`, y por abajo en `Y=500`.

```
public override bool OnTouchEvent(MotionEvent e)
{
    mScaleDetector.OnTouchEvent(e);

    if (mScaleDetector.IsInProgress)
        return true;

    bool result = mGestureDetector.OnTouchEvent(e);

    switch (e.Action)
    {
        case MotionEventActions.Down:
            if (!mScroller.IsFinished)
            {
                mScroller.ForceFinished(true);
                result = true;
            }
            break;
    }

    return result;
}
```

FIGURA 54. EJEMPLO DE IMPLEMENTACIÓN DE ONTOUCHEVENT

Manejar los gestos sobre la vista es algo menos intuitivo, y es necesario hacerlo en el orden correcto para que la vista reaccione como espera el usuario.

El método `OnTouchEvent` ha de retornar `true` si la vista ha podido hacerse cargo del gesto realizado por el usuario [20]. El tipo de gesto se puede identificar por el atributo `Action` del `MotionEvent`, destacando entre los más comunes `Down` -se ha iniciado un gesto de pulsación-, `Up` -ha finalizado un gesto de pulsación- o `Move` -un gesto de movimiento ha ocurrido durante un gesto de pulsación-. Si en `OnTouchEvent` se devuelve `true`, el sistema dejará de buscar la vista que pueda responsabilizarse del gesto. Si no, seguirá llamando a `OnTouchEvent` sobre las vistas cuyos límites contengan el punto (o los puntos) de toque, en el orden en el que estén situadas en el árbol de vistas, hasta terminar en las últimas hojas del mismo. Poniendo como ejemplo una `ScrollView` que tenga una `TextView` como vista hija, hay varios gestos que un usuario puede realizar. Si el usuario arrastra el dedo arriba y abajo, la `ScrollView` desplazará el contenido. Sin embargo, ante una pulsación larga sobre el texto, no es la `ScrollView` quien ha de responder, sino la `TextView`, marcando como seleccionado el texto situado en el lugar de la pulsación.

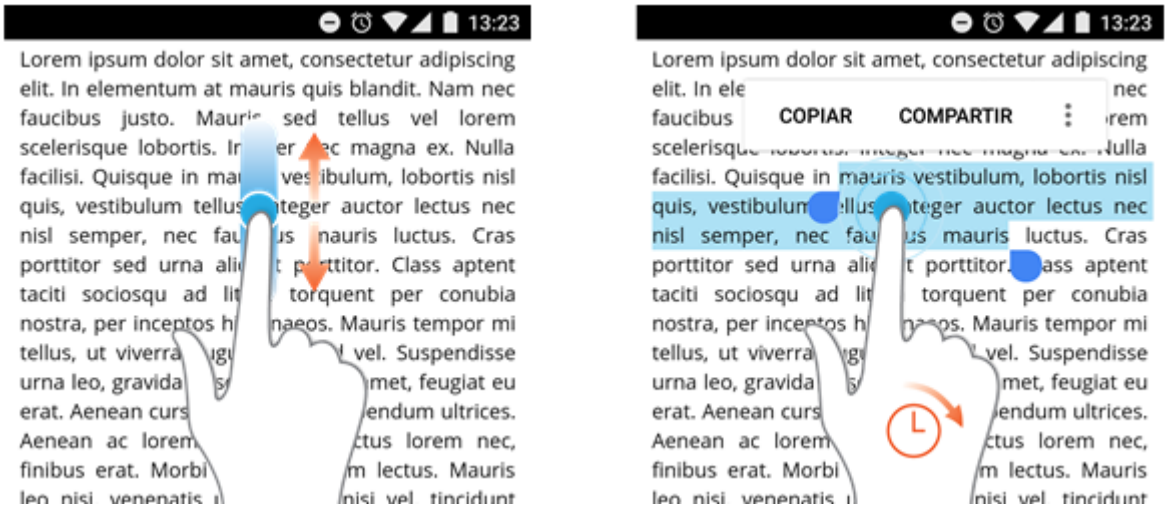

FIGURA 55. EJEMPLO DE "DISPATCHING" DE DISTINTOS EVENTOS DE TOQUE

Desgranando la implementación del método `OnTouchEvent` anteriormente expuesta, es al `ScaleGestureDetector` a quien primero se le pide que procese el `MotionEvent` recibido porque, en caso contrario, siempre quedaría enmascarado por el `GestureDetector` como un gesto de *scroll*. Si el gesto es de inicio de pulsación (`MotionEventActions.Down`), y está sucediendo un movimiento de *fling*, este habrá de ser parado de forma inmediata.

Queda únicamente por revisar la implementación del método `OnDraw` de la vista, donde el *scroll* guardado se convertirá en *scroll* real, y se manejarán el *fling* y el zoom.

```
protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);

    canvas.Save();

    canvas.Scale(
        mScaleListener.ScaleFactor,
        mScaleListener.ScaleFactor,
        mScaleListener.FocusX,
        mScaleListener.FocusY);

    canvas.Translate(mPosX, mPosY);

    UpdateViewPort(canvas);

    if (!mScroller.IsFinished)
    {
        mScroller.ComputeScrollOffset();
        mPosX = mScroller.CurrX;
        mPosY = mScroller.CurrY;
    }

    UpdateViewPort(canvas);

    DrawCircle(canvas);

    canvas.Restore();

    if (!mScroller.IsFinished)
        Invalidate();
}
```

FIGURA 56. EJEMPLO DE IMPLEMENTACIÓN DEL MÉTODO ONDRAW

Es común rodear el código de dibujado con las llamadas `canvas.Save()` y `canvas.Restore()`, que sirven para guardar en una pila FIFO el estado actual del canvas [21] en cuanto a rotación, traslación y escalado [22] se refiere. Si, por ejemplo, se quiere dibujar un cuadrado rotado 25º, como es imposible rotar un `RectF` en el momento de la llamada `canvas.DrawRect(RectF rect, Paint p)`, lo que se hace es rotar el canvas 25º en un sentido, realizar el dibujo, y rotarlo de nuevo 25º en el sentido opuesto, o, sencillamente, guardar el estado del canvas, rotarlo, realizar el dibujo, y restaurar el estado. Proteger el código de dibujo con las llamadas `Save` y `Restore` garantiza que, mientras que el código de dibujado mantenga también la simetría del `Restore` si se realiza algún `Save` adicional, al terminar de dibujar sobre el canvas este tendrá la misma rotación, traslación y escala que al comenzar.

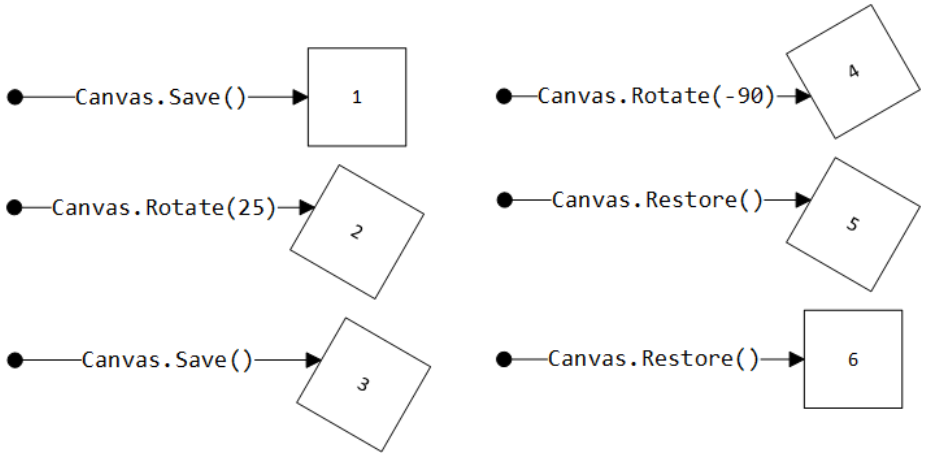

FIGURA 57. EJEMPLO DE CANVAS.SAVE() Y CANVAS.RESTORE()

Hay que tener en cuenta que, en realidad, el canvas ni rota, ni escala, ni se traslada literalmente. Tal y como se señaló en la sección *Conceptos básicos y vocabulario común*, un canvas tiene el mismo tamaño que la vista que se dibuja sobre él. Las operaciones de trasladado y rotación únicamente modifican la matriz de transformación [23] del canvas, que es una matriz 3x3 cuyo valor por defecto es la matriz identidad, y cuyos coeficientes significan lo siguiente [24] [25]:

$$\begin{bmatrix} scaleX & skewX & translateX \\ skewY & scaleY & translateY \\ 0 & 0 & 1 \end{bmatrix}$$

- `scaleX`: escalado en el eje X. Un cuadrado de `(30 x 30)` píxeles con un escalado en el eje X de 2, y un escalado de 1 en el eje Y, medirá `(60 x 30)` píxeles. Igual para el eje Y.
- `translateX`: traslado en el eje X. Igual para el eje Y.
- `skewX`: sesgo en el eje X. En dibujo, el sesgo se utiliza para conseguir efectos de rotación de las primitivas dibujadas. Igual para el eje Y.

FIGURA 58. EJEMPLO DE *SKEW EN LOS EJES X E Y*

Con esta matriz se consigue mapear las coordenadas del contenido en coordenadas de la vista en un solo punto del código, sin necesidad de hacer las transformaciones de dichas coordenadas antes de pasárselas al canvas. Por ejemplo, si las coordenadas mundo son X_m=20, Y_m=35, con una traslación en el eje X de 10, y una traslación en el eje Y de 25, las correspondientes coordenadas en la vista son, tal y como cabría esperar, X_v=30, Y_v=60.

$$\begin{pmatrix} 1 & 0 & 10 \\ 0 & 1 & 25 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 20 \\ 35 \\ 1 \end{pmatrix} = \begin{pmatrix} 30 \\ 60 \\ 1 \end{pmatrix}$$

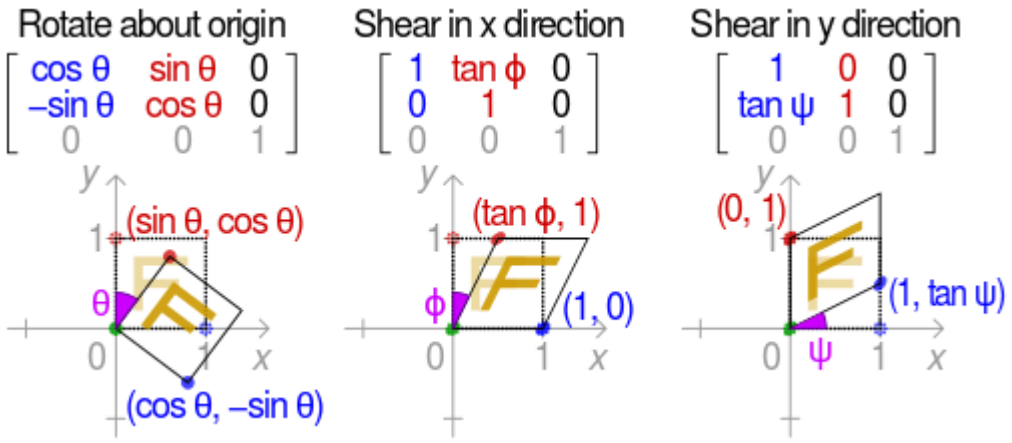

FIGURA 59. EJEMPLOS DE MATRICES DE TRANSFORMACIÓN

Las coordenadas del *viewport* se calculan, en el caso del ejemplo, en el método `UpdateViewport`, y es la última llamada antes del código de dibujado, una vez que la escala y la traslación del canvas no van a variar más hasta la siguiente llamada de `OnDraw`. El cálculo de dichas coordenadas es sencillo:

```
void UpdateViewPort(Canvas canvas)
{
    canvas.GetClipBounds(mCanvasClipBounds);

    mCurrentViewPort.Left = mCanvasClipBounds.Left;
    mCurrentViewPort.Top = mCanvasClipBounds.Top;
    mCurrentViewPort.Right = mCanvasClipBounds.Left + Width/mScaleListener.ScaleFactor;
    mCurrentViewPort.Bottom = mCanvasClipBounds.Top + Height/mScaleListener.ScaleFactor;
}
```

FIGURA 60. ACTUALIZACIÓN DE LA VIEWPORT

`Canvas.GetClipBounds(Rect bounds) : bool` devuelve, en el `Rect` pasado como argumento, la región de clip del canvas (y, adicionalmente, `true` si el clip no es un rectángulo vacío). Es decir, aquel rectángulo tal que, si se dibuja fuera de sus coordenadas, dicho dibujo no aparecerá en pantalla porque quedará fuera de los límites del canvas. Los lados derecho e inferior se calculan en función del origen del rectángulo de clip (`Left`, `Top`) y el factor de escalado actual, además del ancho y alto de la vista (`Width` y `Height`).

Finalmente, el círculo se dibuja, en el ejemplo, en el método `DrawCircle(Canvas canvas)`. El canvas puede pasarse a otros métodos e incluso a otros objetos para que lo manipulen y dibujen sobre él. La única restricción es que todas las llamadas sobre el canvas han de realizarse en el mismo hilo (que, a la hora de dibujar una vista, es el denominado *hilo de interfaz de usuario*, o *UI Thread*). En caso contrario, la ejecución de la aplicación finalizará por un *acceso ilegal*, ya que en Android (al igual que en otras plataformas como WPF), únicamente el hilo que ha creado un elemento de interfaz de usuario puede modificarlo.

La importancia de la velocidad de dibujado

Una pantalla no muestra movimiento real, crea la ilusión del mismo mediante una rápida sucesión de imágenes con leves diferencias entre sí. Se denomina *cuadros por segundo* o *frames per second* (FPS en adelante) a la velocidad que una pantalla, o, en general, cualquier dispositivo que muestre una imagen, es capaz de

cambiar el contenido de la misma. A partir de los 12 FPS el ojo humano percibe movimiento, mientras que, por debajo de ese límite, es capaz de distinguir individualmente las imágenes que tratan de crear la ilusión. Por diferentes circunstancias, en la industria cinematográfica se marcaron como estándar los 24 FPS.

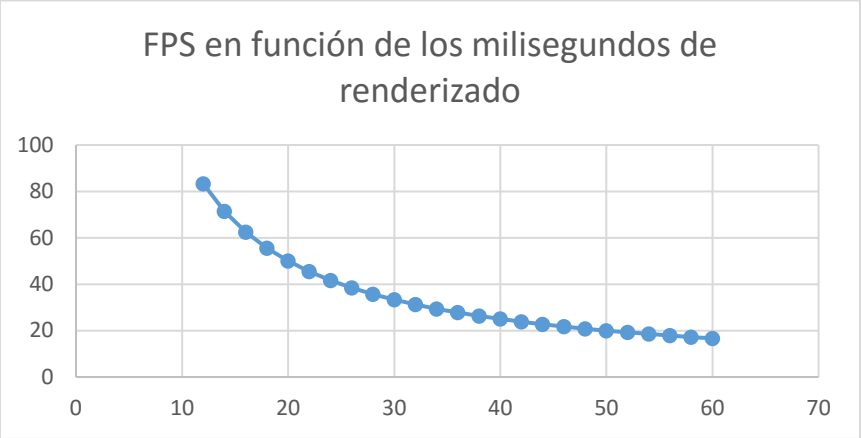

FIGURA 61. RELACIÓN DENTRE LOS *FRAMES PER SECOND* Y EL TIEMPO DE RENDERIZADO DE CADA UNO

Sin embargo, desde Google [26], a la hora de diseñar interfaces de usuario, recomiendan encarecidamente que la UI pueda moverse a 60 FPS. El motivo de este límite teórico es que, de acuerdo a según qué fuentes, la mayoría de humanos no pueden percibir las mejoras que pueden otorgar, por ejemplo, 120 FPS. En aras de facilitar la verificación de la fluidez de la interfaz, en la versión de Android 4.3 Jellybean se introdujo, entre las opciones de desarrollador, un *profiler* del rendimiento de la UI. Mediante barras verticales el dispositivo muestra el tiempo que se tarda en dibujar cada marco. Si bien no se dan medidas exactas para cada marco, un tiempo que quede por encima de la línea verde (16 milisegundos) señala que, en teoría, el usuario ha podido percibir la caída de FPS.

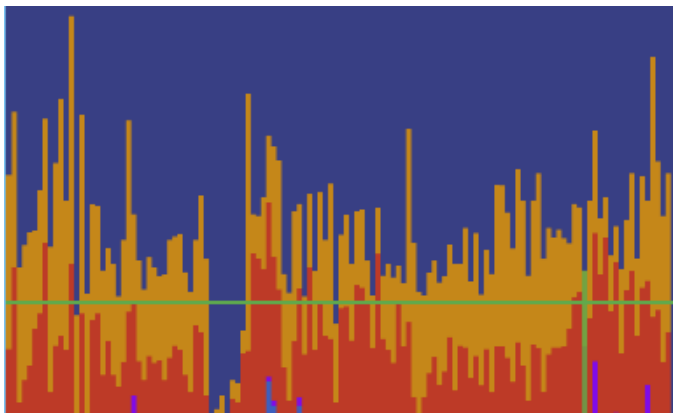

FIGURA 62. EJEMPLO DE *PROFILING* EN LA APLICACIÓN DE RELOJ

Aquí es donde resulta clave el concepto de *viewport*. Dibujar sobre el canvas es una operación costosa en cuanto a tiempo se refiere, por lo que el objetivo debería ser disminuir la cantidad de objetos que terminan dibujados al mínimo. Sabiendo que el *viewport* es la región, en coordenadas de *mundo*, de la parte que terminará siendo visible en coordenadas de *vista*, en el código de dibujado se habrá de desechar lo más rápido posible cualquier dibujo que no entre dentro del área de clip.

En el Branch Explorer de Plastic SCM, cada objeto del mismo (cada *changeset*, cada *label*, cada *rama* y cada *link*) está rodeado por un rectángulo que, dependiendo del objeto, servirá para dibujarlo -más sobre esto en *Dibujado del Branch Explorer*-, pero siempre servirá para saber si se encuentra dentro del área de clip mediante la operación `Intersects`, que incluyen la gran mayoría de implementaciones de representaciones de rectángulos (`Android.Graphics.RectF`, `System.Drawing.Rectangle`, `CoreGraphics.CGRect`, etc).

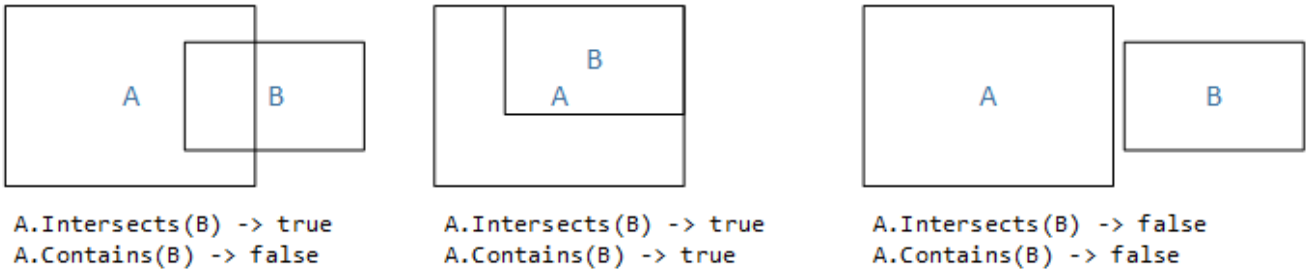

FIGURA 63. EJEMPLO DE *INTERSECTS* Y *CONTAINS*

Hay que señalar que, tal y como se aprecia en la figura, la operación válida para el dibujado es `Intersects`, ya que se desea dibujar un objeto en cuanto intersecte mínimamente con el clip, no cuando esté totalmente contenido en él.

Completando el ejemplo de la sección anterior, el método `DrawCircle(Canvas canvas)` es el siguiente.

```
void DrawCircle(Canvas canvas)
{
    if (!CircleIntersectsRect(mCenterX, mCenterY, mRadius, mCurrentViewPort))
        return;

    canvas.DrawCircle(mCenterX, mCenterY, mRadius, mCirclePaint);
}

static bool CircleIntersectsRect(float centerX, float centerY, float radius, RectF rect)
{
    if (rect.Contains(centerX, centerY))
        return true;

    float distanceX = Math.Abs(centerX - rect.CenterX());
    float distanceY = Math.Abs(centerY - rect.CenterY());

    if ((distanceX <= rect.Width()/2) || (distanceY <= rect.Height()/2))
        return true;

    if ((distanceX > rect.Width() / 2 + radius)
        || (distanceY > rect.Height() / 2 + radius))
        return false;

    double cornerDistance = Math.Pow((distanceX - rect.Width()/2), 2)
         + Math.Pow((distanceY - rect.Height()/2), 2);

    return cornerDistance <= Math.Pow(radius, 2);
}
```

FIGURA 64. DIBUJADO DENTRO DEL *ÁREA DE CLIP*

Se intenta asegurar tan pronto como sea posible que el círculo intersecta con el *viewport*, siendo la comprobación más sencilla que el centro del círculo está contenido por el *viewport*, finalizando con el *corner case* de que el círculo intersecte con la esquina. Una comprobación más sencilla habría sido crear un cuadrado con origen en `(mCenterX – mRadius, mCenterY – mRadius)` y ancho y alto igual a `2 x mRadius`, y comprobar una intersección entre rectángulos, pero esto provocaría que el círculo fuese dibujado también cuando se encontrase fuera del *viewport*, cercano a las esquinas.

Las restricciones que imponen Skia y el objeto Path

El Branch Explorer en su versión de escritorio está compuesto, entre otros, de dos objetos gráficos de una complejidad geométrica media que son pre-calculados al adaptar el *layout* del Branch Explorer al *framework* específico que se vaya a utilizar para dibujarlo, y que no se vuelven a calcular hasta que dicho Branch Explorer no es recargado por completo. Estos son las ramas y los *mergelinks.*

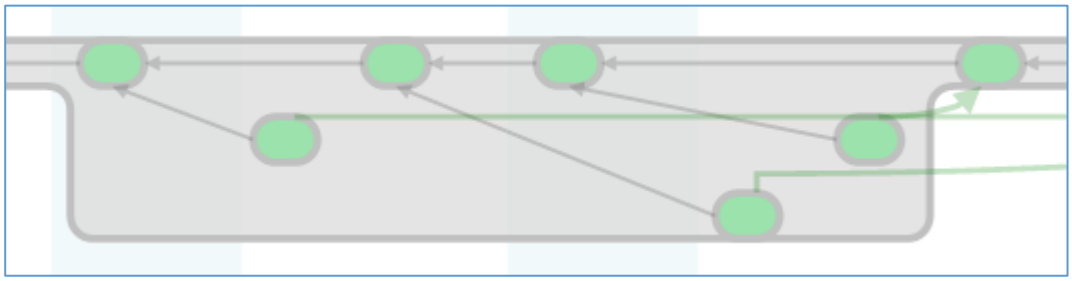

FIGURA 65. EJEMPLO DE DIBUJO DE UNA SUB-RAMA

La complejidad de las ramas radica en que pueden contener sub-ramas. Una sub-rama es una anomalía que surge cuando dos desarrolladores hacen cambios en paralelo sobre la misma rama y con la misma base. La sub-rama existe para almacenar los cambios de ambos desarrolladores, y no desaparece hasta que no se hace *merge* de las dos -o más- cabezas de la rama, reunificándolas en una sola. El cálculo del dibujo de las sub-ramas implica continuos cálculos de ángulos para que las fusiones sean suaves, así como correcciones de anchura y de altura a cada momento para garantizar que se recubren todos los *changesets* que se encuentren en la sub-rama.

La complejidad de los *mergelinks* es que estos pueden no iniciarse en el propio *changeset*, sino que pueden añadir una pequeña línea vertical de margen entre el *changeset* y la curva del *mergelink*. Esta línea depende del ratio de crecimiento de la curva, y sirve para separarla visualmente de la rama en caso de que el crecimiento vertical de la curva con respecto al horizontal sea pequeño. Se puede observar también en la imagen de ejemplo de sub-rama: mientras que el primer *mergelink*, que parte del segundo *changeset* del dibujo comenzando por la izquierda, no tiene línea de separación vertical, la del quinto *changeset* sí la tiene.

En los clientes de escritorio, para representar estas formas se utiliza un objeto gráfico llamado `Path`, que tiene equivalente en todas las plataformas (siendo `android.graphics.Path` el equivalente en Android). Un `Path` es un objeto que almacena múltiples figuras geométricas, que pueden ser segmentos de línea, curvas cuadráticas, y curvas cúbicas[7], entre otros. Es un objeto muy útil a la hora de dibujar tanto sub-ramas como *mergelinks*, pues su API permite crear dichas figuras desplazándose a lo largo del contorno de las mismas. Sin embargo, en Android, el tamaño gráfico de `Path` que puede dibujarse sobre un canvas está limitado en anchura

[7] Forma de curva definida por funciones cuadráticas y cúbicas, respectivamente.

y en altura. Si el `Path` excede dicho límite, este no será dibujado, y Android añadirá al log el siguiente mensaje de error:

```
Path too large to be rendered into a texture
```

Dichos `Paths` forman parte de la librería gráfica que se utilice para el dibujado en la plataforma específica. Se denomina formalmente *librería gráfica* al software que es capaz de generar y/o dibujar imágenes en base a unos modelos matemáticos y unos patrones de iluminación, texturas, etc. Así pues, detrás de toda imagen generada por ordenador -y, por lo tanto, a la hora de dibujar el Branch Explorer-, hay una librería más o menos compleja, dependiendo de su propósito y de sus capacidades.

La librería gráfica detrás del dibujado del canvas en Android se llama Skia [27]. Desarrollada originalmente por Skia Inc., fue comprada por Google en 2005 y se usa actualmente, además de en Android, en Google Chrome, Chrome OS[8], Mozilla Firefox y Sublime Text 3[9], entre otros.

Gracias a que la base de la versión comercial de Android distribuida por Google y otros fabricantes es de código abierto -AOSP, Android Open Source Project [28]-, es posible examinar el código fuente de la clase `android.graphics.Canvas`, y seguir las llamadas de código desde que se trabaja sobre un objeto Java hasta que dicha llamada termina en la librería gráfica. A pesar de que en este Trabajo de Fin de Grado se esté utilizando Xamarin.Android, no hay que olvidar que este es, en su mayoría, *bindings* a las clases Java que forman Android, no una re-implementación del *framework*.

Canvas delega la funcionalidad de `DrawPath(Path path, Paint paint)` en una llamada a código nativo, definida en `core/jni/android/graphics/Canvas.cpp` [27]. Este, a su vez, delega en `SkCanvas.cpp`, ya dentro del *framework* de Skia (`src/core/SkCanvas.cpp` [28]), donde se termina llamando a la función `SkCanvas::onDrawPath(const SkPath& path, const SkPaint& Paint)`.

Finalmente, el límite del tamaño de un `Path` que puede ser renderizado en un canvas, tal y como señala el ingeniero de Google Romain Guy en la comunidad de Android Developers [29], viene determinado por la GPU del dispositivo. El tamaño máximo *"mínimo"* garantizado es de `2048 x 2048` píxeles, un tamaño que puede ser superado por cualquier rama y por cualquier *mergelink*, por lo que pre-calcular la geometría de estos objetos es inútil, y calcularla en detalle a la hora del dibujado, costoso.

Dibujado del Branch Explorer

El Branch Explorer se dibuja por etapas, en cada una añadiendo todos los elementos del mismo tipo, de forma que se pueden distinguir sobre el mismo una serie de capas. Se enumera a continuación el orden de dibujado de los elementos, así como detalles de los mismos, desde las capas inferiores a las superiores.

Fondo: para mantener la coherencia con el resto de clientes, este es blanco y uniforme. Cabe señalar que, por defecto, el color de fondo de un canvas en Android no es blanco. Sin embargo, la llamada `canvas.DrawColor(Color color)` cubre toda la región actual de clip del color especificado.

[8] Sistema operativo ligero basado en Google Chrome.
[9] Editor de texto desarrollado por Jon Skinner.

Columnas: las columnas en las que se divide el Branch Explorer permiten seguir, en orden cronológico, los cambios realizados en el repositorio. Una columna agrupa todos los *changesets* que fueron realizados el mismo día. Hay que señalar que una *label* -o etiqueta- puede ser creada en cualquier momento sobre cualquier *changeset*, pudiendo estar distanciadas en más de un día la fecha de creación del *changeset* y la fecha de creación de la *label*, por lo que las columnas no señalan la fecha de creación de estas últimas. Hay que aclarar también que, como al crear un *mergelink* se crea también el *changeset* de destino, la fecha de creación de un *mergelink* coincide con la fecha de creación de dicho *changeset*.

Las columnas abarcan el ancho necesario para recubrir sus *changesets*, y la altura necesaria para alcanzar la parte superior e inferior de la vista. Esta información se encuentra entre la que proporciona el código legado de cálculo de *layout*, por lo que, finalmente, el dibujado de columnas se reduce al dibujado de un rectángulo con lados izquierdo y derecho aquellos proporcionados, y superior e inferior los lados superior e inferior de la viewport.

Parentlinks: cada *changeset* queda relacionado con su padre, en el diagrama, por un segmento de línea finalizado en una punta de flecha que apunta al padre. No forman parte de la información de *layout* que proporciona el código legado, pero esta sí permite calcularlos de una forma sencilla. En los clientes de escritorio los *parentlinks* están definidos por dos puntos, uno de inicio y uno de fin, y por una cabeza de flecha. El código de dibujado en dichos clientes de escritorio calcula si el *parentlink* se encuentra parcial o totalmente dentro de la viewport actual de la siguiente forma:

1. Si el punto de origen o el punto de fin se encuentran en el área de clip, dibujar el *parentlink*. Si no:
2. Si la línea que va desde el punto de origen hasta el punto de fin corta alguno de los lados del área de clip, dibujar el *parentlink*. Si no:
3. No dibujar el *parentlink*.

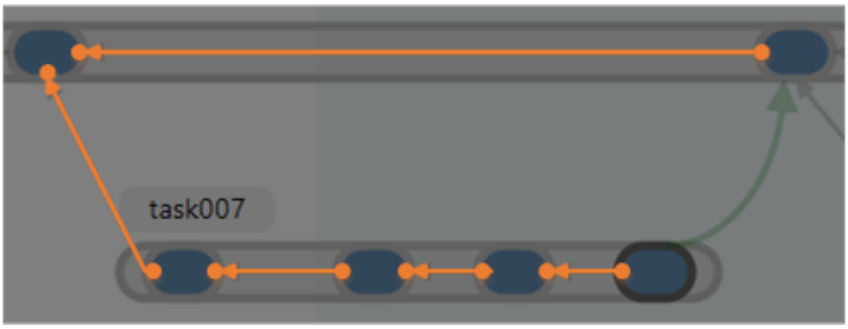

FIGURA 66. *PARENTLINKS* EN LOS CLIENTES DE ESCRITORIO

Para un Branch Explorer de pruebas consistente en la rama `main` completa del repositorio principal de código de Plastic SCM, -mas 79 ramas adicionales, 6707 *changesets* en total-, la cantidad total de *parentlinks* se situaba en torno a los 6500. Calcular si cada uno de esos 6500 *parentlinks* se encontraba dentro del área de clip provocaba una pérdida de tiempo que se traducía en un número muy bajo de *frames* por segundo, por lo que en la versión móvil de Plastic SCM se ha optado por dibujar de manera explícita únicamente los *parentlinks* que cumplan que:

a) Los *changesets* que involucra se encuentran en niveles diferentes -por lo que el *parentlink* no es una línea horizontal, sino oblicua-.

b) Los *changesets* que involucra no se encuentren dentro de la línea principal de una rama.

Esto permitió reducir, en el *layout* de pruebas, la cantidad total de *parentlinks* de ~6500 a apenas ~200 (ya que la mayoría de dichos elementos está entre *changesets* situados dentro de la línea principal de una rama), con el consiguiente aumento en el rendimiento. Adicionalmente, el cálculo que decide si el *parentlink* ha de ser dibujado no se realiza como una intersección de su recta con cualquiera de los lados del área de clip. Cada *parentlink* tiene ahora definido un rectángulo que marca sus límites, y del cual dicho *parentlink* es la diagonal en caso de que sea entre *changesets* de distintos niveles, o lo corta por el centro en horizontal si los *changesets* están al mismo nivel. De esta forma puede que algún *parentlink* se dibuje a pesar de no encontrarse dentro del área de clip porque sus límites sí lo intersectan, pero la cantidad de operaciones que se deben realizar para el cálculo de intersecciones son mucho menores.

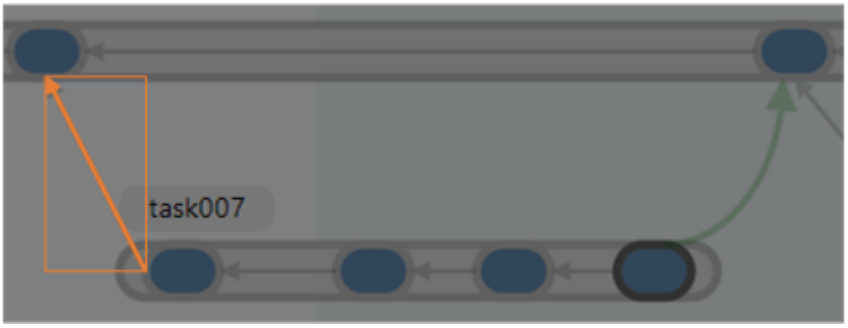

FIGURA 67. *PARENLINKS* EN EL CLIENTE MÓVIL

Branches: estas se dibujan con cierto grado de transparencia para poder distinguir, a través de ellas, los *parentlinks* sobre los que se sitúen. Como ya se mencionó en la sección *Las restricciones que imponen Skia y el objeto Path*, debido a su tamaño arbitrario, la geometría de la rama no se puede pre-calcular. Sí se dispone, sin embargo, y de nuevo proporcionado por el código legado, el límite izquierdo, derecho, superior e inferior de la rama. Por tanto, a la hora de dibujar una rama, es necesario calcular si dichos límites intersectan en el área de clip. Una vez se ha comprobado que sí, una rama es un rectángulo con los bordes redondeados, por lo que el dibujado se hace en los siguientes pasos:

1. Cálculo de los límites izquierdo y derecho del dibujo de la rama:
    i. El lado izquierdo será igual al máximo entre el lado izquierdo de la rama y el lado izquierdo de la región de clip menos un límite de seguridad -para evitar fallos estéticos-.
    ii. El lado derecho será igual al mínimo entre el lado derecho de la rama y el lado derecho de la región de clip más un límite de seguridad -para evitar fallos estéticos-.
2. Dibujado de un rectángulo con los bordes redondeados con lados izquierdo y derecho los calculados en el paso anterior, y superior e inferior los proporcionados.
3. Dibujado de una línea horizontal que cruce el rectángulo por la mitad, desde el lado izquierdo más el límite de seguridad hasta el lado derecho menos el límite de seguridad.

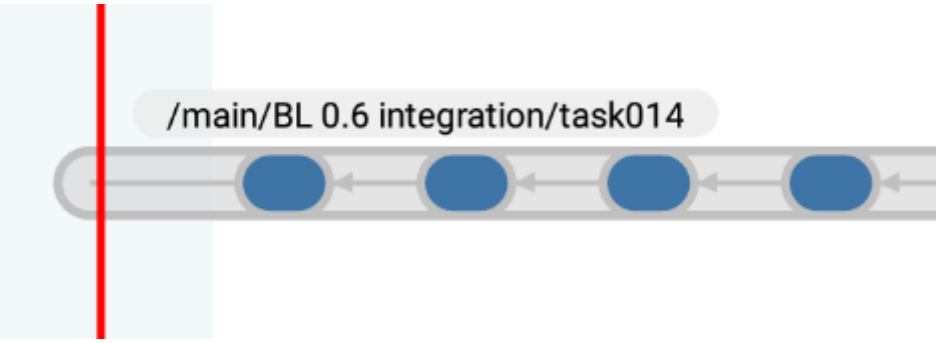

FIGURA 68. DIBUJADO DE UNA RAMA EN EL LÍMITE DE LA VIEWPORT

El paso 3) sirve para simular los *parentlinks* que se omitieron en el dibujado de los mismos. En la figura superior, la línea roja marca el límite del *viewport* -el cálculo del mismo ha sido modificado para poder ver cómo es el dibujo fuera de sus límites-. Mientras que la rama real `/main/BL 0.6 integration/task014` todavía tiene más *changesets* a la izquierda, el dibujo termina inmediatamente después del límite del *viewport*. Se puede apreciar de igual forma en el dibujo que el límite de seguridad no es más que el radio de curvatura del borde de la rama.

Changesets: el *changeset* se representa en el Branch Explorer por un rectángulo con las esquinas redondeadas, con un borde ligeramente más oscuro al habitual si es la cabeza de su rama. En los clientes de escritorio, los *changesets* son coloreados de forma distinta dependiendo del origen de replicación de los mismos, siendo estos colores personalizables por el usuario. Adicionalmente, el *changeset* en el que se encuentre el *workspace* actual está marcado con el icono de una casa. En el cliente móvil, debido a que no existen *workspaces*, no hay ningún *changeset* marcado como tal. Sin embargo, a pesar de que en los datos de dibujado del Branch Explorer sí se disponga del origen de replicación, en esta primera versión el color de los *changesets* no varía en base al mismo.

Mientras que el dibujado del *changeset* no entraña ninguna dificultad, hay que señalar que, asociado al mismo, sí está la punta de flecha del *parentlink* hijo que esté a su mismo nivel. Mientras que los *parentlinks* entre *changesets* de distintas ramas y *changesets* que se encuentren dentro de una sub-rama sí se dibujan explícitamente, los que relacionan *changesets* de la misma rama se simulan, como ya se ha mencionado anteriormente. La línea del *parentlink* es dibujada junto a la rama, mientras que la flecha es dibujada junto al *changeset*.

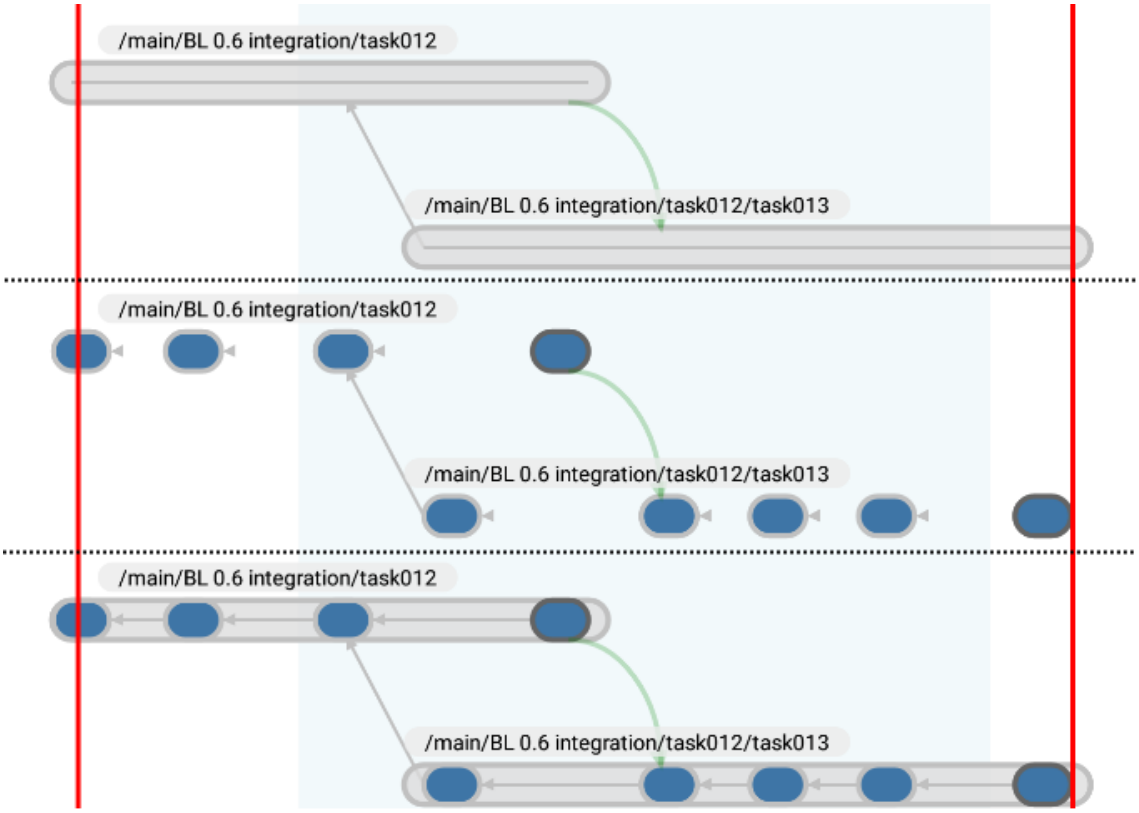

FIGURA 69. DIBUJADO DE RAMAS Y *CHANGESETS* POR SEPARADO

Desactivando en el código por separado el dibujado de ramas y el de *changesets*, se puede distinguir en conjunto cómo son realmente dichos *parentlinks*.

Labels: las *labels* o etiquetas están representadas por una forma de corona que rodea el *changeset* etiquetado, acompañada del nombre de dicha etiqueta en la parte superior. Como son dibujos pequeños -en cuanto a lado se refiere-, estos sí son pre calculados sobre un objeto Path. El texto de la etiqueta, sin embargo, no se dibuja directamente sobre el canvas. Para ello se utiliza un `StaticLayout`, una clase usada para el dibujo de texto que, una vez dispuesto, no vaya a cambiar.

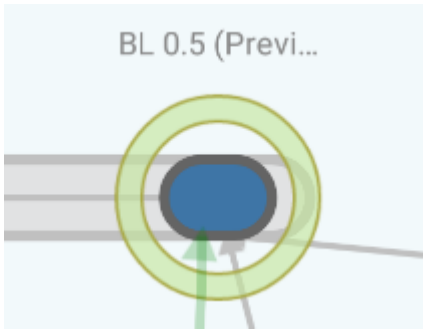

FIGURA 70. EJEMPLO DE UNA *LABEL*

Como el texto de la etiqueta (de nuevo, para mantener la coherencia con el resto de aplicaciones) no supera en ancho al ancho de la etiqueta, es necesario truncarlo al final. Para poder truncar el texto, es necesario saber cuál va a ser su ancho en función de la fuente. La información de dibujado de texto, como ya se señaló en la

sección Conceptos básicos y vocabulario común, la tiene el objeto `Paint` que se vaya a usar para dicho dibujado. Sin embargo, en Android se dispone de una clase estática llamada `TextUtils` que, entre la funcionalidad que ofrece, se encuentra la de truncado de texto a un ancho determinado en función del `TextPaint` que se vaya a utilizar.

```
static void CreateLabelCaptions(LabelDraw label, BranchExplorerPaints colors)
{
    TextPaint textPaint = colors.GetLabelCaptionPaint();

    string ellipsizedText = TextUtils.Ellipsize(
        label.Captions[0],
        textPaint,
        label.Bounds.Width(),
        TextUtils.TruncateAt.End);

        label.TextLayout = new StaticLayout(
            new Java.Lang.String(ellipsizedText),
            textPaint,
            (int)label.Bounds.Width(),
            Android.Text.Layout.Alignment.AlignCenter,
            0.0F,
            1.0F,
            true);
}
```

FIGURA 71. CREACIÓN DEL STATICLAYOUT QUE CONTENDRÁ EL NOMBRE DE LA *LABEL*

Una vez creado el `StaticLayout`, no se puede cambiar el `TextPaint` que utiliza, debido a que esto podría cambiar las medidas del texto y, tal y como se dijo, `StaticLayout` está diseñado para ser utilizado con texto que no vaya a cambiar. De hecho, la documentación de `StaticLayout` advierte al respecto, en el método `getPaint()` de la siguiente manera [32]:

`Return the base Paint properties for this layout. Do NOT change the paint, which may result in funny drawing for this layout.`

Sin embargo, cambiar el color del `TextPaint` sí está permitido -ya que no afecta a las medidas del texto-, y será algo necesario a la hora de indicar que la etiqueta está seleccionada.

```
label.TextLayout.Paint.Color = label.IsSelected
    ? Color.DarkGray
    : Color.Gray;
```

FIGURA 72. CAMBIO DE COLOR DEL TEXTPAINT ASOCIADO A UN STATICLAYOUT

La clase Canvas no ofrece ningún método para dibujar un `StaticLayout` en las coordenadas necesarias. Es, de hecho, `StaticLayout`, quien dispone de un método `Draw(Canvas c)`, donde se le proporciona el canvas necesario para que se dibuje, pero sin ningún control sobre la posición donde terminará el texto. Es necesario, pues, trasladar el canvas antes del dibujado, y restaurarlo después.

```
float xTranslate = label.Bounds.Left;
float yTranslate = label.Bounds.Top - _textHeight - CAPTION_LABEL_MARGIN;

canvas.Save();
canvas.Translate(xTranslate, yTranslate);
label.TextLayout.Draw(canvas);
canvas.Restore();
```

FIGURA 73. TRASLADADO DEL CANVAS PARA EL DIBUJADO DEL TEXTO

Mergelinks: los *mergelinks* indican, mediante una curva, una operación de *merge* entre dos *changesets*. En función del tipo de la operación representada -*merge* tradicional, *merge* substractivo, *cherry pick*, *cherry pick* substractivo…- el color de la curva cambiará. Dicha curva es el recorrido de un cuarto de elipse, terminada en una cabeza de flecha que apunta al *changeset* de destino, y cuyos *changesets* implicados se encuentran en el centro vertical y horizontal del rectángulo que envuelve la elipse completa -al extremo que sea necesario-.

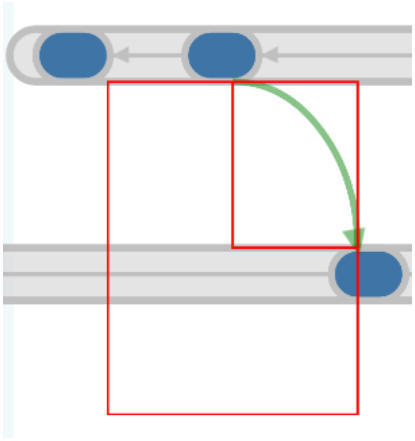

FIGURA 74. RECTÁNGULOS IMPLICADOS EN EL CÁLCULO DEL *MERGELINK*

En la figura superior se pueden apreciar los rectángulos que se utilizan para el dibujado del *mergelink.* El propósito del cuadrado interior es calcular si la curva ha de ser dibujada, mientras que el exterior sirve para dibujar la sección de la elipsis necesaria. El canvas, a la hora de dibujar un arco, necesita los siguientes parámetros: un rectángulo que contenga dicho arco, un ángulo de inicio, y un ángulo de *sweep* o recorrido (en sentido horario), ambos en grados. Dependiendo del cuadrante -de los cuatro en los que se puede dividir- donde se deba situar el *mergelink*, el ángulo inicial y el de recorrido variarán tal y como se indica:

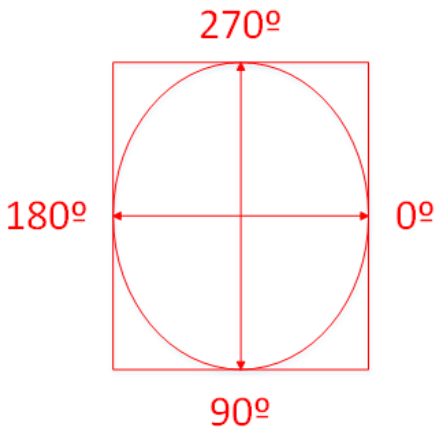

FIGURA 75. ÁNGULOS EN UN CANVAS DE ANDROID

Así, por ejemplo, en el mergelink de la figura, el ángulo inicial será 270º, con un *sweep* de 90º (para recorrer la parte de la elipsis que va de los 270º a los 0º).

Etiquetas de nombres de rama: las ramas van acompañadas de una etiqueta que ha de dibujarse, si el principio de la rama queda dentro del *viewport*, al inicio de la rama, y si queda fuera, en el margen izquierdo del *viewport*. De esta forma, la etiqueta se va desplazando junto a la rama, siempre visible.

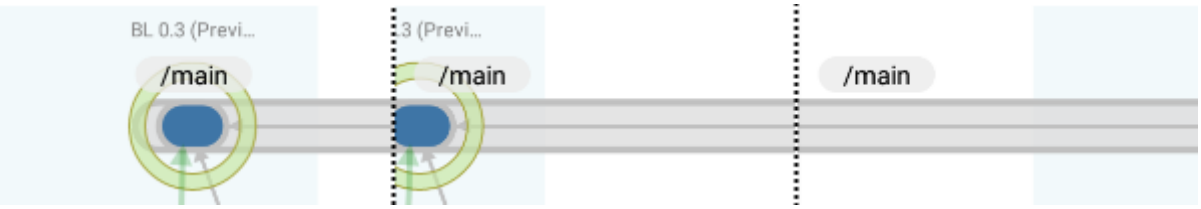

ILUSTRACIÓN 1, DESPLAZAMIENTO DEL NOMBRE DE RAMA

La forma de dibujar dicho texto es exactamente igual a la que se utiliza para dibujar el nombre de las etiquetas, mediante un `StaticLayout` y desplazando el canvas donde sea necesario para que el texto quede por encima de la rama, con sus respectivos márgenes inferior y lateral.

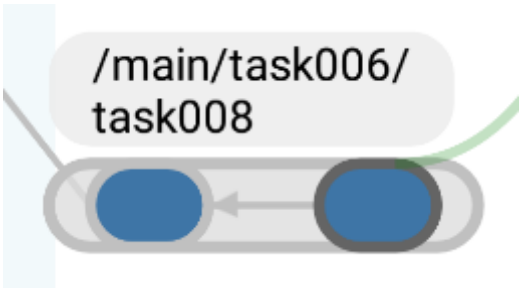

ILUSTRACIÓN 2, NOMBRE DE UNA RAMA ACOMODADO EN VARIAS LÍNEAS

Puede suceder que el ancho de la rama no sea el suficiente para acomodar en una única línea la totalidad del nombre. En ese caso, este habrá de separarse en varias. Es algo de lo que se encarga, así mismo, `StaticLayout`.

Cabeceras de columna: las cabeceras de las columnas tienen, como texto indicativo, la fecha en la que se crearon los *changesets* que abarca dicha columna. Se dibuja en último lugar porque ha de quedar encima de todo el contenido. Al igual que con el nombre de etiquetas y ramas, las fechas se dibujan sobre el canvas mediante un `StaticLayout`.

5.1.3 - El Branch Explorer en iOS

Una vez se han establecido conceptos básicos como el de *viewport* o área de clip, o los movimientos de *scroll*, *zoom* y *fling*, trasladarlos de Android a cualquier otra plataforma es trivial. Además, a la hora de dibujar el Branch Explorer en iOS, se cuenta con la ventaja de que ya existe un cliente de escritorio para macOS, sistema operativo de escritorio también de Apple, por lo que las diferencias entre el dibujado en la tableta y en el escritorio serán mínimas debido al objetivo que se marcó la nombrada compañía en que el salto entre plataformas para los desarrolladores fuese lo más simple posible.

Diferencias y similitudes respecto a Android

A la hora de trasladar los conceptos de dibujado de Android a iOS, habrá que entender unas diferencias fundamentales pero simples.

- El dibujado en iOS no se realiza sobre un canvas pasado por el sistema a la vista, se realiza sobre un *contexto gráfico*, que será volcado posteriormente sobre una *Layer* (más sobre esto a continuación).
- El dibujado en iOS es independiente de la densidad de pantalla. Mientras que en Android era necesario hacer la conversión de *densitiy-independant pixels* a *pixels reales*, iOS hace estas transformaciones automáticamente, incluso librándose del concepto de píxeles para usar sencillamente *puntos*.
- Los sistemas de coordenadas de ciertas operaciones cambian. Por ejemplo, a la hora de medir ángulos para dibujar arcos, estos se miden en radianes, en vez de en grados.

Sin embargo, muchos de los conceptos se mantienen (no ya solo por similitud con Android, sino con cualquier otro *framework* de interfaz de usuario), siendo dos de ellos el sistema de coordenadas de las vistas (con origen en la esquina superior izquierda), y los árboles de vistas.

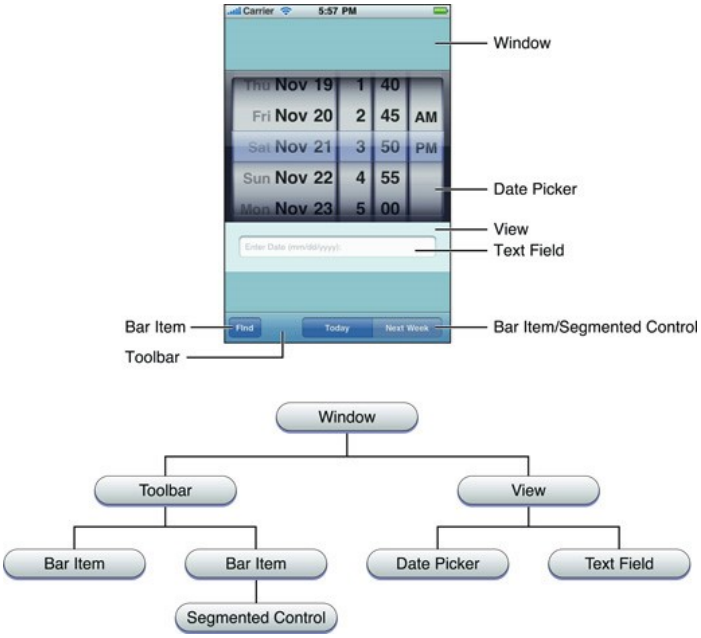

FIGURA 76. ÁRBOL DE VISTAS EN IOS

De igual forma, para implementar una *vista personalizada,* mientras que en Android había que heredar de la clase `View`, en iOS el equivalente será `UIView`.

UIView y CALayer: cómo se relacionan y en qué se diferencian

Debido a la política de nombrado de Apple en la que cada clase comienza por las iniciales del *framework* al que pertenece, las sigas CA hacen referencia a que las clases `CALayer` y `CALayer` pertenecen al *framework* de Core Animation, la infraestructura que Apple ofrece en su plataforma iOS para facilitar al desarrollador el dibujado en pantalla (bien de forma explícita a través de sus clases, bien de manera implícita a través de `UIKit`).

Un objeto de tipo `Layer` (es decir, una `CALayer` o cualquiera de sus subclases) es una representación de una superficie bidimensional, organizada en un espacio tridimensional, cuyo cometido es manejar contenido basado en imágenes. La principal función de una Layer es servir de respaldo para el contenido que una `UIView` muestra en pantalla, capturándolo en un mapa de bits que pueda ser fácilmente manipulado por el hardware gráfico del dispositivo iOS correspondiente, así como las transformaciones realizadas sobre este contenido. Es la unidad fundamental de dibujado en iOS, pues toda `UIView` depende de una o más Layers para mostrarse. De hecho, una `UIView` simplemente delega en su Layer subyacente muchas de las opciones que se pueden manipular de la primera (como, por ejemplo, la opacidad). Esta función de *almacenaje* (comúnmente referida a lo largo de la documentación oficial como *backing store*) de lo que se muestra en pantalla será importante posteriormente a la hora de comprender las diferencias entre distintos tipos de *layers*.

Las `Layer` no son, bajo ninguna circunstancia, un reemplazo de las `UIView` como elementos de una interfaz de usuario. Son una manera de dar *infraestructura* a dichas vistas, hasta el punto de que en iOS (al contrario que en macOS) no existe ninguna `UIView` sin una `CALayer` asociada como *backing store* de la misma. Las `Layer` son una forma de dibujar y transformar eficientemente el contenido de una `UIView`, son la forma que la `UIView` tiene de reflejar gráficamente su estado interno. Sin embargo, las `Layer` no aceptan interacción por parte del usuario.

Adicionalmente, una `Layer` se encarga de manejar la geometría de su contenido, en cuanto a que almacena información sobre el tamaño del mismo, su posición en pantalla, su ángulo de rotación, su factor de escalado, o cualquier otra transformación que se haya hecho sobre el mismo. Además, una `Layer` tiene propiedades de las que una `UIView` carece. Por ejemplo, el *anchor point*, que define el punto alrededor del cual se ha realizado una manipulación de la imagen (como el centro de un movimiento de rotación).

Como ya se ha señalado antes, los sistemas de coordenadas en iOS tienen su origen, al igual que en Android, en la esquina superior izquierda, creciendo el eje de abscisas hacia el lateral derecho, y el de ordenadas hacia el borde inferior. Las unidades que se utilizan para medir distancias son representadas en punto flotante, y son independientes de la densidad de píxeles por pulgada de la pantalla sobre la que se esté dibujando la `UIVIew`. La plataforma promete que, a la hora de dibujar una `CALayer`, la misma medida en el sistema de coordenadas de una vista tendrá el mismo tamaño físico de cara al usuario independientemente del dispositivo iOS utilizado, algo muy conveniente cuando la densidad de píxeles por pulgada ha variado tanto a lo largo de la historia de los dispositivos de Apple. Mientras que el iPhone original tenía una densidad de 163 dpi, la primera pantalla Retina (nombre comercial) del iPhone 4 contaba con 326 dpi, una densidad que ha subido en los recientes modelos iPhone 6 y *6 plus* hasta los 401 dpi.

Dibujado y sistema de coordenadas

Para poder dibujar sobre la `CALayer` subyacente (o cualquiera de sus subtipos) a una `UIView`, es necesario [33] sobrescribir el método `Draw` (si se va a redibujar la porción visible de la vista al completo) o `DrawRect` (si se va a redibujar únicamente una parte de la porción visible de la vista). Mientras que en Android el sistema proporcionaba de forma explícita un objeto de tipo Canvas sobre el que realizar el dibujado, en iOS habrá que *obtener* un *contexto gráfico*. Dicho contexto gráfico es un objeto configurado por el sistema, que describe dónde y cómo se ha de dibujar una primitiva sobre la `Layer` (junto a otros atributos como el color que se debe utilizar, el área de clip disponible, fuente de letra, etc), en contraposición a la dualidad en Android que existía entre las clases `Canvas` y `Paint`.

```
public override void DrawRect(CGRect rect, UIViewPrintFormatter formatter)
{
    base.DrawRect(rect, formatter);

    using (CGContext gc = UIGraphics.GetCurrentContext())
    {
        SetupGraphics(gc);

        mDrawer.Draw(gc, mLayout, GetRectF(rect));
    }
}

public override void Draw(CGRect rect)
{
    base.Draw(rect);

    using (CGContext gc = UIGraphics.GetCurrentContext())
    {
        SetupGraphics(gc);

        RectangleF clip = GetRectF(rect);

        mDrawer.Draw(gc, mLayout, clip);
    }
}
```

FIGURA 77. EJEMPLO DE LOS MÉTODOS DRAW Y DRAWRECT

La documentación oficial recomienda no intentar obtener un *contexto gráfico* fuera de estos dos métodos, pues al igual que en Android era el sistema quien pasaba el canvas al código de dibujado de la vista, fuera de la ejecución de estos dos métodos -que son llamados por el sistema- es muy probable que no exista un contexto gráfico configurado por el sistema.

El método `SetupGraphics` simplemente añade configuración adicional al `CGContext` obtenido para que, entre otras cosas, utilice técnicas de *antialiasing* en el dibujado. Por último, se envía al código de dibujado tanto el `Graphic Context` como el *layout* del Branch Explorer y la región de clip -obtenida a su vez del `CGContext`- con el fin de poder descartar rápidamente objetos que no se vayan a poder visualizar en el Branch Explorer porque queden fuera de la región.

En la *figura 78* se puede observar cómo, al igual que en Android, es `CGContext` quien tiene la responsabilidad del *qué* se dibuja (`AddPath`) y como al contrario que en Android, tiene también la responsabilidad del *cómo* se dibuja (`FillPath`, `StrokePath`, `SetFillColor`, `SetStrokeColor`).

```
internal void DrawChangeset(CGContext gc, ChangesetDraw changeset)
{
    gc.SetFillColor(BranchExplorerColors.GetChangesetColor(changeset));
    gc.AddPath(changeset.Path);
    gc.FillPath();

    gc.SetStrokeColor(BranchExplorerColors.GetChangesetBorderColor(changeset));
    gc.SetLineWidth(Measures.ChangesetBorder(changeset));
    gc.AddPath(changeset.Path);
    gc.StrokePath();
}
```

FIGURA 78. EJEMPLO DE DIBUJADO SOBRE UN CGCONTEXT

La región de clip enviada al código de dibujado corresponde al área de la vista visible representada en el sistema de coordenadas de la propia vista. Sin embargo, al contrario que en Android, toda `UIView` dispone de dos sistemas de coordenadas. Uno de ellos representa la localización del contenido de la vista respecto a la propia vista: es la propiedad `Bounds`. Todo objeto que sea dibujado por la vista se encontrará dentro de dichos *bounds*. La segunda de estas propiedades, `Frame`, representa las coordenadas de la vista respecto a su *supervista* (o vista padre) dentro del árbol de vistas.

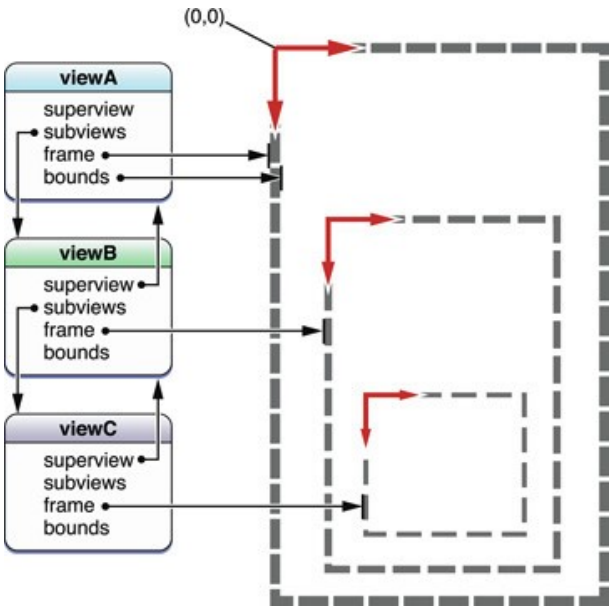

FIGURA 79. PROPIEDADES *FRAME* Y *BOUNDS* RESPECTO A LAS VISTAS Y A SUS *SUPERVISTAS*

Distintos tipos de Layer

Dependiendo de la finalidad de la `UIView`, esta puede cambiar el tipo de `Layer` que actúa como *backing store* de la misma. Esto se hace, en Xamarin.iOS, de la siguiente forma:

```
[Export("layerClass")]
public static Class GetLayerClass()
{
    return new Class(typeof(CATiledLayer));
}
```

FIGURA 80. CAMBIO DEL TIPO DE LAYER EN UNA UIVIEW

Así, por ejemplo, `CATextLayer` es un tipo de Layer especializada en el renderizado de texto (no la única que puede renderizar texto, pero sí la que se espera que lo haga de la forma más eficiente), `CAMetalLayer` está especializada en el renderizado de texturas de *Metal* [34], el API de iOS para el dibujado acelerado por GPU, y la `CATransformLayer` está especializada en crear *auténticas jerarquías 3D* (de acuerdo a su documentación), en vez de la jerarquía aplanada que usan el resto de Layers.

El tipo de Layer utilizada en la prueba de concepto de dibujado del Branch Explorer en iOS fue la `CATiledLayer`. Este último tipo está específicamente pensado para el dibujado de imágenes que exceden, por mucho, el tamaño de pantalla (pudiendo ser virtualmente infinitas en cada dirección), y que además cambien poco o nada (como es el caso del Branch Explorer, que cambia, una vez dibujado, únicamente cuando la selección de objetos cambia).

La `CATiledLayer` divide el contenido a mostrar en pantalla en pequeños cuadrados o *tiles* [35]. A medida que el contenido se desplaza por la vista, la `CATiledLayer` va ordenando de forma asíncrona el dibujado de dichos *tiles*, encargándose del cacheo de los mismos y su posterior liberación cuando ya no vayan a ser necesarios. Dicho dibujado sucede en el ya mencionado método `Draw` de la correspondiente `UIView`. Para que la aparición de dichos *tiles* no sea visualmente desagradable, `CATiledLayer` los hace aparecer mediante una animación de *fade in*. La duración de dicha animación, así como el tamaño del *tile*, no son propiedades que se puedan modificar directamente. Sin embargo, sí se pueden crear subtipos de `CATiledLayer`, ya que esta no es una clase sellada, en los que se sobrescriban las propiedades correspondientes.

Reconocimiento de gestos y UIScrollView

Al igual que en Android, donde se necesitaban como mínimo dos objetos para detectar toques (un `GestureDetector` y una subclase de `SimpleOnGestureListener`) además de tener que sobrescribir el método `OnTouchEvent` para interceptar el `MotionEvent` correspondiente, en iOS se pueden añadir a las vistas *reconocedores de gestos*, o *gesture recognizers*, que hacen los cálculos y el seguimiento necesarios para distinguir distintos tipos de gesto sobre la pantalla. De esta forma, en la creación de la vista se crean tantos objetos especializados en reconocer gestos concretos como gestos se quieran manejar, y se registran en la `UIView` como *reconocedores*, con un método *handler* para manejar el evento -o bien una expresión *lambda*-.

Los *reconocedores* son clases muy especializadas, reconociendo gestos muy concretos. Por ejemplo, `UIScreenEdgePanGestureRecognizer` identifica únicamente gestos de arrastre que comienzan cerca de

los bordes de la pantalla (que pueden servir, por ejemplo, para que el usuario navegue a la pantalla anterior). La creación del *recognizer* y su registro se harían de la siguiente manera:

```
void CreateGestureRecognizers()
{
    UIScreenEdgePanGestureRecognizer recognizer =
        new UIScreenEdgePanGestureRecognizer();
    recognizer.AddTarget(() => HandleScreenEdgePanGesture(recognizer));
    AddGestureRecognizer(recognizer);
}

void HandleScreenEdgePanGesture(UIScreenEdgePanGestureRecognizer recognizer)
{
    // TODO
}
```

FIGURA 81. EJEMPLO DE GESTURERECOGNIZER

Sin embargo, una `CATiledLayer` es, por diseño, perfecta para usarla en conjunto con una `UIScrollView`. Mientras que, en Android, a través de los `GestureDetector`, era necesario manejar la traslación y el escalado del Canvas (ya que `ScrollView` y `HorizontalScrollView` funcionan únicamente con scroll vertical y horizontal, respectivamente, y nunca simultaneo, aunque se aniden), en iOS una `UIScrollView` puede hacer *scroll* vertical y horizontal simultáneo (o gesto de *panning*) a sus vistas anidadas. De esta forma se elimina la necesidad de manejar en la vista personalizada todos los gestos que puede hacer el usuario, quedando únicamente por reconocer los gestos de toque o *tap* que determinan la selección de un objeto en el Branch Explorer.

Así, la jerarquía de vistas del Branch Explorer quedaría de la siguiente manera. Una `UIScrollView`, encargada de manejar tanto los gestos de scroll como de zoom (proporcionando, además, scroll con inercia), una `UIView` personalizada donde se dibujará el Branch Explorer (a la que llamaremos `UIBranchExplorer`), y la `CATiledLayer` subyacente a la `UIBranchExplorer` (aunque en sentido estricto no forma parte de la jerarquía por formar parte indivisible de `UIBranchExplorer`).

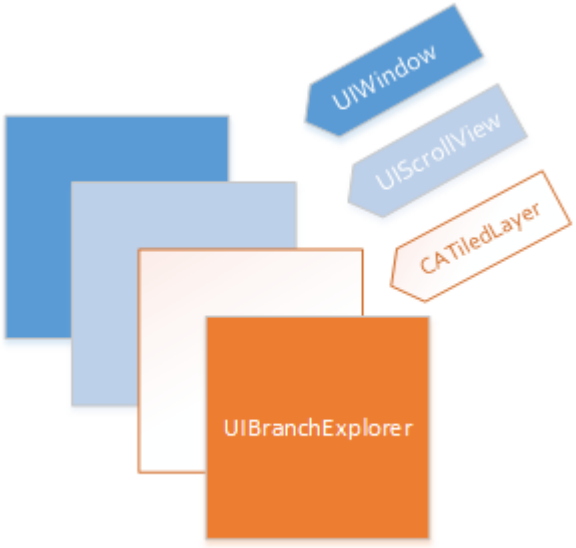

FIGURA 82. JERARQUÍA DE VISTAS DEL BRANCH EXPLORER EN IOS

La única responsabilidad -en cuanto a gestos se refiere- que queda pendiente por manejar en `UIBranchExplorer` es la selección de objeto del Branch Explorer. Al igual que en Android toda `View` disponía de un método llamado `OnTouchEvent`, donde se recibía una instancia de `MotionEvent` que, entre otra información, contenía la relativa al tipo de gesto ejecutado, la cantidad de punteros involucrados, y las posiciones X e Y de todos ellos, en iOS una `UIView`, por el hecho de ser subtipo de `UIResponder`, dispone de muchos más métodos para responder a toques. Destacan los siguientes:

- `TouchesBegan`: es llamado cuando uno o más dedos tocan una UIView.
- `TouchesMoved`: es llamado cuando uno o más dedos se desplazan a través de una UIView.
- `TouchesEnded`: es llamado cuando uno o más dedos se levantan del toque sobre una UIView.
- `TouchesCancelled`: es llamado cuando un gesto es cancelado por un evento tal como un aviso de memoria baja (no cuando el usuario *cancela* el toque, cosa que realmente no puede hacer).
- `MotionBegan`: es llamado cuando se inicia un gesto de movimiento (tal como agitar el dispositivo).
- `MotionEnded`: es llamado cuando finaliza un gesto de movimiento.
- `MotionCancelled`: es llamado cuando un gesto de movimiento es cancelado por un evento tal como un aviso de memoria baja.
- `PressesBegan`: es llamado cuando un botón físico asociado a la UIView actual es pulsado. En modelos más recientes de iPhone, el gesto *force touch* (pulsar con fuerza sobre la pantalla del dispositivo) puede ser manejado a través de este evento.
- `PressesCancelled`: es llamado cuando se cancela el gesto de presión sobre un botón físico por eventos tales como un aviso de memoria baja.
- `PressedChanged`: es llamado cuando la presión sobre el botón físico varía. Para botones físicos la presión varía entre 0 y 1.
- `PressesEnded`: es llamado cuando la presión sobre el botón físico finaliza.

De esta forma, la distinción del tipo de gesto que se hacía en Android mediante la propiedad `Action` de `MotionEvent` en iOS ya no es necesaria. Las acciones se distinguen según el método que se haga cargo de ellas. Para detectar toques sencillos sobre el Branch Explorer, únicamente será necesario implementar el método `TouchesBegan`.

```
public override void TouchesBegan(NSSet touches, UIEvent evt)
{
    base.TouchesBegan(touches, evt);

    UITouch touch = touches.AnyObject as UITouch;

    if (touch == null)
        return;

    CGPoint touchLocation = touch.LocationInView(this);

    float x = (float)touchLocation.X;
    float y = (float)touchLocation.Y;

    ObjectDraw touchedObject = BrExObjectSelector.TrySelectObject(x, y, mLayout);

    if (touchedObject == mSelectedObject)
        return;

    if (touchedObject != null)
    {
        mPreviousSelectedObject = mSelectedObject;
        mSelectedObject = touchedObject;
        mSelectedObject.IsSelected = true;
        SetNeedsDisplayInRect(mSelectedObject.Bounds);
    }

    if (mPreviousSelectedObject != null)
    {
        mPreviousSelectedObject.IsSelected = false;
        SetNeedsDisplayInRect(mPreviousSelectedObject.Bounds);
    }
}
```

FIGURA 83. EJEMPLO DE IMPLEMENTACIÓN DE TOUCHESBEGAN

5.2 - Diferencias de texto *side by side*

Tradicionalmente, existen dos formas de mostrar diferencias de texto. El llamado *diff unificado* se vale únicamente de texto para señalar las porciones de un documento que han cambiado, utilizando caracteres tales como la cruz para indicar que se han añadido líneas, el guion para indicar que se han eliminado, o números para marcar los caracteres concretos que han cambiado dentro de una línea. Aunque es una forma muy extendida por su sencillez de representación en la pantalla, y se usa, por ejemplo, a la hora de colaborar en proyectos software para enviar *parches* (ya que existen herramientas especializadas como `patch`, capaces de interpretar y aplicar las diferencias señaladas en un *diff unificado*, permitiendo facilitar el proceso de integrarlas), no es la forma más sencilla de visualización.

La segunda forma de mostrar diferencias de texto, las diferencias *lado a lado* o *side by side*, muestran -como su nombre indica- las dos revisiones de un fichero lado a lado (comúnmente la más antigua, o *source*, a la izquierda), señalando con colores las regiones del documento que hayan cambiado, y, en versiones avanzadas, sincronizando el desplazamiento de ambas revisiones para compensar cosas tales como regiones añadidas o borradas, para que en todo momento las regiones relevantes se encuentren a la misma altura. Como ya se ha señalado en secciones anteriores de este mismo documento, uno de los pilares de venta de Plastic SCM, promocionado en la portada de su página Web, son dichas diferencias *lado a lado*. De esta manera, apoyándose en el software de control de versiones, un desarrollador puede entender cómo ha evolucionado un fichero, por qué se han tomado ciertas decisiones de desarrollo, e incluso localizar en qué puntos se pudieron introducir fallos, con el fin de poder atajarlos más fácilmente.

5.2.1 - Código legado independiente de la plataforma

El equipo de Códice Software cuenta entre sus integrantes con auténticos expertos en el cálculo de diferencias de texto, hasta el punto de haber podido registrar al respecto patentes [2] en Estados Unidos y en Europa. Al igual que sucedió con el Branch Explorer, a medida que nuevos productos se iban incorporando al ecosistema de Plastic SCM (tales como las extensiones de entornos de desarrollo, o más recientemente, los clientes de GNU/Linux y macOS), era imprescindible que el cálculo de diferencias e incluso el código de apoyo al dibujado de las mismas fuese independiente de la plataforma, con el fin de evitar re-implementar los mismos algoritmos una y otra vez.

El cálculo de diferencias más avanzado se puede encontrar actualmente en el cliente de escritorio de Windows. Otro de los productos de Códice Software, bautizado como SemanticMerge, se creó como apoyo al proceso de *merge* prometiendo facilitarlo *entendiendo el código*, es decir, haciendo un análisis semántico del mismo, mucho más profundo que un mero análisis sintáctico que permitiese el cálculo de diferencias de texto tradicionales. Dicho cálculo de diferencias semántico se integró en el año 2015, a partir de la versión 5.4.16.648 [36], en el cliente de Plastic SCM de Windows, permitiendo identificar diferencias con un nivel de detalle mucho mayor que el que permite un análisis de texto.

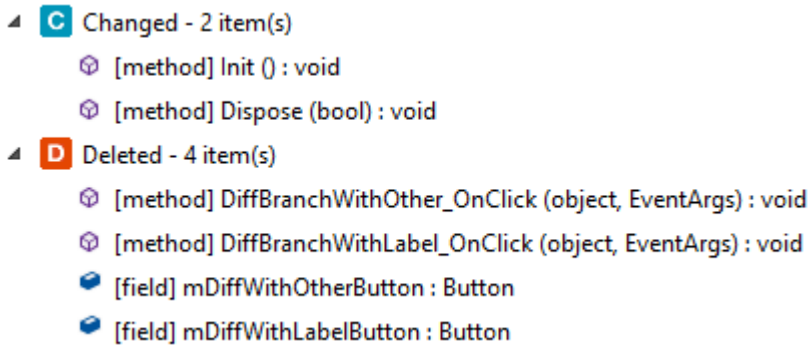

FIGURA 84. EJEMPLO DEL *SEMANTIC OUTLINE* DE WINDOWS

Gracias al panel resumen (*semantic outline*) que muestra dicho cliente, el desarrollador puede agrupar rápidamente las diferencias según pertenezcan al nombre de un método, a la modificación en la signatura de métodos, a la modificación en la sección de *imports/usings* etc. Sin embargo, para que dicho análisis *semántico* funcione, es necesario tener instalado software de apoyo en la máquina (para crear el árbol de sintaxis del fichero), dependiendo del lenguaje a calcular las diferencias semánticas: por ejemplo, el compilador Roslyn para las diferencias semánticas en C#, o la Java Virtual Machine para las diferencias semánticas en Java.

No es, sin embargo, lo único que hacen al visionado de diferencias en el cliente de Windows el más avanzado de los tres. La capacidad para mover código de una revisión anterior a una nueva con un único botón (en caso de ser editable la revisión más reciente), o la sincronización entre porciones de código movido (para poder comparar lado a lado secciones que hayan cambiado de posición en el documento, o incluso que hayan sido movidas a otros documentos), hacen del cliente de Windows aquel con el visor de diferencias más completo y complejo.

El resaltado de sintaxis para una gran variedad de lenguajes era otra de las características avanzadas del cliente de Windows, aunque posteriormente fue adoptado por el de GNU/Linux, siendo actualmente (junio de 2016) el cliente de macOS el único cliente de escritorio que no cuenta todavía con resaltado de sintaxis.

5.2.2 - Dibujado de diferencias *side by side* en Android

A la hora de mostrar en Android las diferencias de texto *side by side*, una vez que estas ya han sido calculadas, se presentaron varios retos. El primero de ellos, poder sincronizar dos `TextView` que muestren ambas revisiones del fichero de manera correcta, fundamental para que en casos tales como la aparición de código añadido, movido, o borrado, no se degradase la sencillez de visualizar dichas diferencias. El segundo de los retos, consistente en el resaltado de regiones de texto de dichos `TextView` a dos niveles: el primero, de línea completa, y el segundo, de regiones dentro de la propia línea.

Sincronización de TextViews

En la sección de implementación del Branch Explorer se señaló la necesidad de manejar el *scroll* a través de un `GestureDetector` por un motivo muy simple. Las clases por defecto que proporciona Android para manejar el *scroll* de contenido, `ScrollView` y `HoritonzalScrollView`, únicamente manejan *scroll* vertical u horizontal respectivamente, mientras que el Branch Explorer debía poder moverse en cualquier dirección. Aunque una `HorizontalScrollView` se anide dentro de una `ScrollView` (o viceversa), el orden de

dispatching de eventos de toque hace que en cada momento sea una de las dos únicamente la que responda al gesto de scroll, pudiendo ser únicamente vertical u horizontal, pero no ambos simultáneamente.

Sin embargo, a la hora de mostrar texto en pantalla, un único tipo de movimiento de scroll de manera simultánea puede ser considerado suficientemente bueno, evitando tener que manejarlo mediante un `GestureDetector` haciendo traslaciones del `canvas` antes de que la `ScrollView` personalizada se lo pase a sus vistas hijas para que dibujen su contenido. Lo único necesario sería, en todo caso, ser capaces de sincronizar los movimientos de dos o más `ScrollView`.

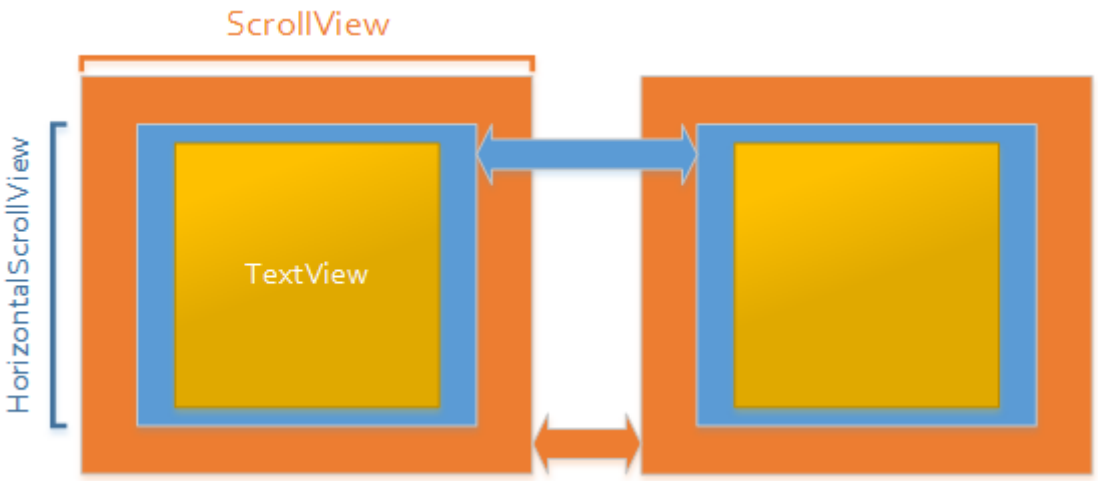

FIGURA 85. JERARQUÍA DE VISTAS DE LA PANTALLA DE DIFERENCIAS

Así pues, la jerarquía de vistas quedaría tal y como se ve en la anterior figura. El contenido de ambas revisiones, así como las diferencias entre ambas, se muestran dentro de un `TextView`. Este se encontrará anidado dentro de una `HorizontalScrollView`, quien, a su vez, se encontrará anidada en una `ScrollView`. La sincronización de cada `ScrollView` deberá realizarse con su homónima, para que tanto el desplazamiento horizontal como el vertical se mantengan coherentes entre ambas revisiones.

Hacer esto es relativamente sencillo. Se implementó patrón observador de la siguiente manera:

- Una `SyncedScrollView` (subtipo de `ScrollView`) y una `HorizontalSyncedScrollView` (subtipo de `HorizontalScrollView`), que son capaces de notificar de los cambios en el *scroll* a un observador.
- Una clase para la sincronización, el observador, encargada de ser notificada de cambios por cualquier `ScrollView`, y notificar de estos cambios al resto de `ScrollViews`.

Cuando el *scroll* de una vista cambia, esta notifica al observador. Este a su vez recorre todos los clientes que tenga registrados, notificándoles del cambio de *scroll* únicamente cuando sea necesario (cuando el cliente no sea quien ha lanzado la notificación, y cuando el cliente realice el mismo tipo de *scroll* que el notificador).

Que la `TextView` se encuentre dentro de una `HorizontalScrollView` ayuda, además, a que ésta ajuste su ancho al contenido, y no al ancho de su vista padre, por lo que se evita que se coloquen automáticamente saltos de línea para ajustar el texto a la pantalla, manteniendo la configuración original del fichero.

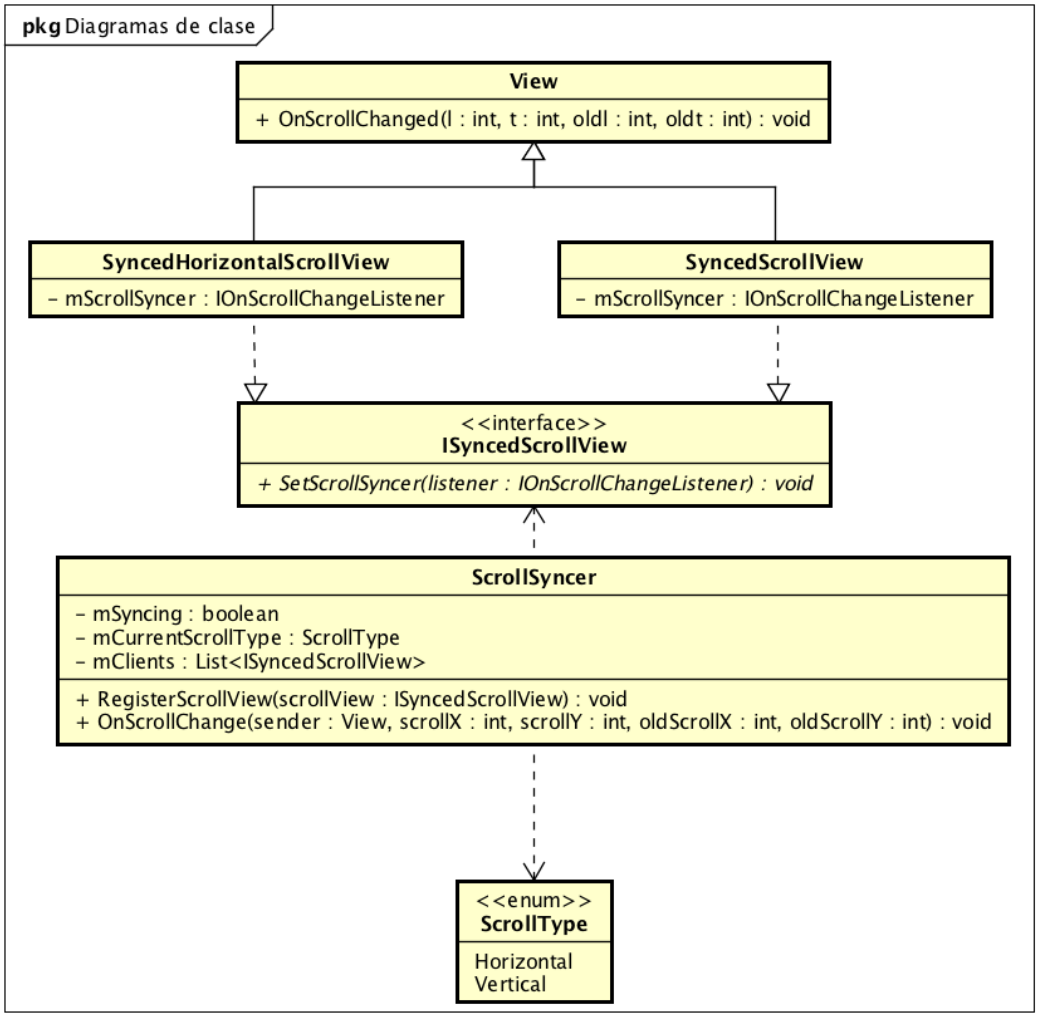

FIGURA 86. DIAGRAMA DE CLASES DETALLADO DE DROIDPLASTIC.DIFFERENCES.SCROLL

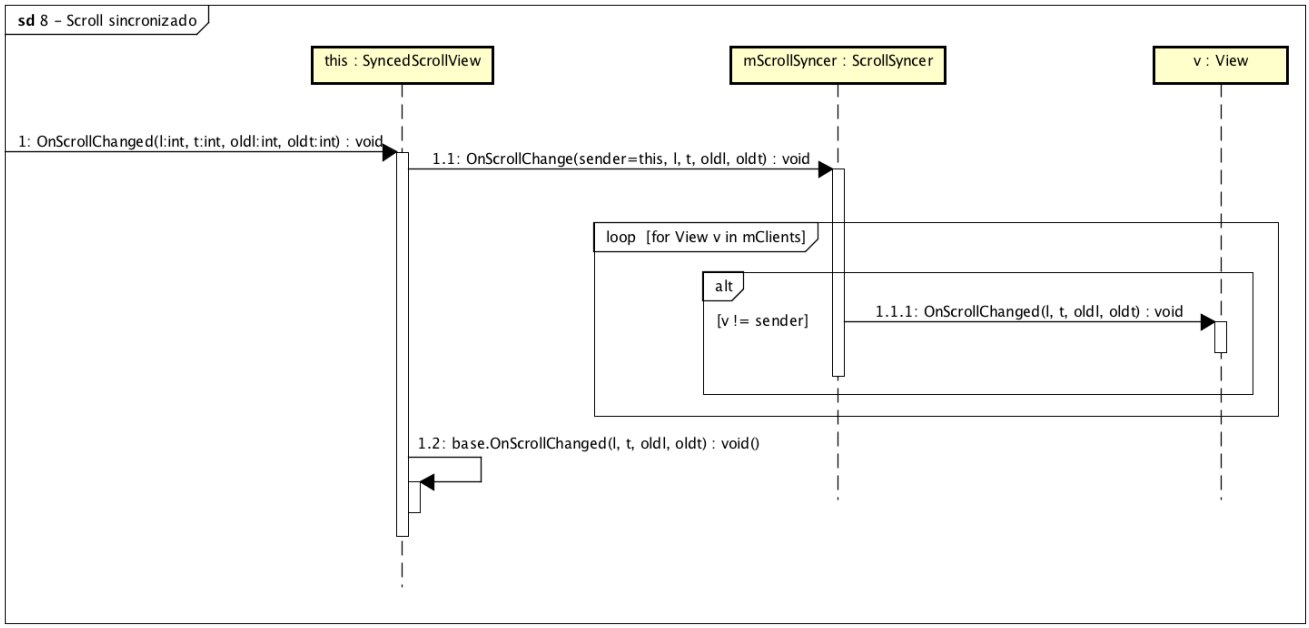

FIGURA 87. DIAGRAMA DE SECUENCIA DE SINCRONIZACIÓN DE SCROLL

Resaltado de regiones en una TextView

Tal y como se señaló en las secciones correspondientes al dibujado de vistas personalizadas en Android, toda `View` tiene un método `OnDraw` en el que el sistema le pasa a la vista un Canvas sobre el que dibujarse. El caso de los `TextView`, por complejo que pueda ser el correcto renderizado de texto, no es distinto.

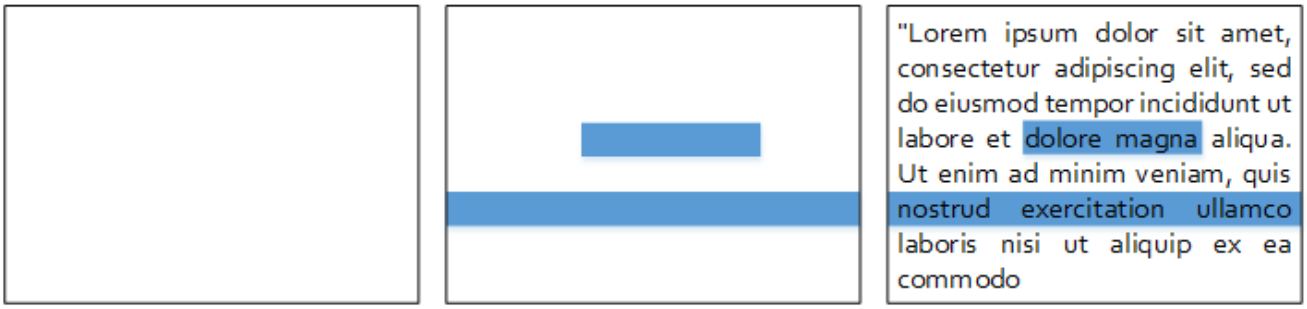

FIGURA 88. RESALTADO DE TEXTO EN TRES FASES

Entre la información de las diferencias que retorna el código legado también se encuentra, como se señaló anteriormente, información sobre el dibujado de las mismas (independiente así mismo de cualquier *framework* como pueden ser Windows Presentation Foundation o GTK). Dicha información consiste en rangos de línea que deben ser pintados junto a su color (`ColorTextRegion`), y rangos de caracteres dentro de una línea que deben ser pintados, de nuevo junto a su color (`ColorInsideLineTextRegion`). Con la información contenida por estos (`InitialLine` y `EndLine` para las regiones de línea, `InitialColumn` y `EndColumn` para las regiones de caracteres) se pueden calcular regiones en términos del sistema de coordenadas de la vista, ya dependientes del *framework* concreto, así como el color.

Dibujar las regiones que abarcan líneas completas es algo trivial. `TextView` dispone del método `getLineHeight()` (que en Xamarin.Android se convierte en la propiedad computada `LineHeight`), que, en último término, delega en la `TextPaint` correspondiente el cálculo de dicha altura. Así, la región que cubre de las líneas *n* a *m* es el rectángulo que abarca de `X=0` a `X=Width`, y de `Y=(n-1)*LineHeight` hasta `Y=m*LineHeight`.

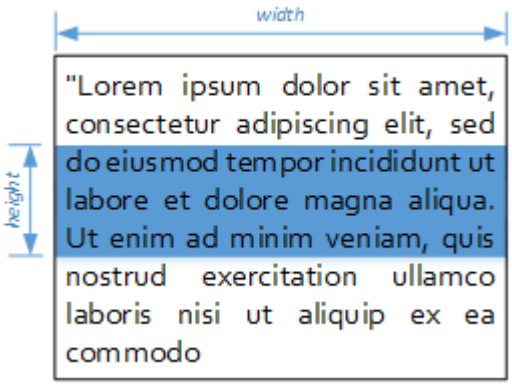

FIGURA 89. PINTADO DE REGIONES DE LÍNEA COMPLETA

Sin embargo, el dibujo de regiones que abarcan caracteres dentro de una línea no es inmediato. A la hora de representar textos en pantalla se usa una *tipofaz* (o *familia de fuentes*), es decir, un conjunto de glifos (que

representarán letras, números, y otros símbolos como "@" o "$") con unas características de diseño comunes. Una de esas características de diseño es el espaciado entre letras, ente los que se pueden distinguir dos vertientes: el espaciado proporcional y el *monoespaciado.*

FIGURA 90. MONOESPACIO Y ESPACIADO PROPORCIONAL

Por convención y por las ventajas que ofrecen las fuentes monoespaciadas en interfaces compuestas únicamente por texto (cuadrado de columnas, por ejemplo), es el tipo que se suele utilizar a la hora de representar código, y, por lo tanto (y para mantener la coherencia con el resto de clientes), para representar diferencias de texto en Plastic SCM.

Al igual que un objeto `TextPaint` puede medir la altura de un texto, también puede medir la anchura del mismo, independientemente del tipo de espaciado que use su fuente. Como la información de las diferencias proporcionada por el código legado incluye carácter inicial y final de las regiones de columna, el rectángulo que cubre la región se calcula de la siguiente forma:

```
RectF GetRectangleForCharacterRange(string line, int startChar, int endChar)
{
    int startIndex = startChar + mLineIndicatorLength;
    int endIndex = endChar + mLineIndicatorLength + 1;

    float leftOffset = Paint.MeasureText(line, 0, startIndex);
    float rangeWidth = Paint.MeasureText(line, startIndex, endIndex);

    return new RectF(leftOffset, 0, leftOffset + rangeWidth, 0);
}
```

FIGURA 91. CÁLCULO DE REGIONES DE COLUMNAS

En el extracto de código anterior hay que señalar el miembro de clase `mLineIndicatorLength`. Es la medida, en caracteres, del indicador de número de línea. `TextView` no proporciona ninguna forma de añadir numeración a sus líneas, por lo que el método más sencillo es, a la hora de modificar el texto, separarlo en líneas, añadiendo al principio de las mismas el número. Como lo que se espera es que los finales de los indicadores

estén alineado a la derecha (para no descuadrar el texto), es necesario calcular, antes de añadir ninguno, la longitud que dicho indicador deberá tener (y que variará en función de la cantidad total de líneas).

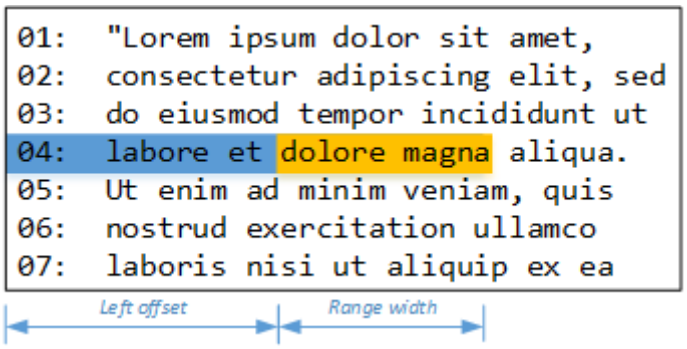

FIGURA 92. MEDIDAS DE UNA REGIÓN DE CARACTERES

Esta forma de añadir los números de línea permite intercalar líneas en blanco (para sincronizar regiones añadidas, movidas, y borradas) sin afectar a la numeración. El código de cálculo de diferencias proporciona un mapeo entre las líneas de la revisión izquierda y la revisión derecha, que no es más que una lista de enteros, en las que la posición de la lista se corresponde a la línea de la revisión, y el entero almacenado en esa posición, a la línea correspondiente de la revisión complementaria. Si en algún punto de este mapeo se encuentra un cero, significa que no hay correspondencia directa con la otra revisión, por lo que será necesario intercalar dichos espacios en blanco.

```
public void SetTextContent(string text, List<int> linesMapping)
{
    mLinesMapping = linesMapping;

    string[] realLines = text.Split(
        new[] {Environment.NewLine},
        StringSplitOptions.None);

    mVirtualLines = new string[linesMapping.Count];

    mMaxLineNumberStringLength = (int) Math.Log10(linesMapping.Count) + 1;
    mLineIndicatorLength = mMaxLineNumberStringLength + LINE_NUMBER_SEPARATOR.Length;
    string formatString = "{0," + mMaxLineNumberStringLength + ":D2}" +
LINE_NUMBER_SEPARATOR + "{1}";

    for (int i = 0; i < linesMapping.Count; i++)
    {
        int index = linesMapping[i] - 1;

        mVirtualLines[i] = index < 0
            ? string.Empty
            : string.Format(formatString, linesMapping[i], realLines[index]);
    }

    Text = string.Join(Environment.NewLine, mVirtualLines);
}
```

FIGURA 93. INSERCIÓN DE NÚMEROS DE LÍNEA

Una vez que se tiene listo el código de dibujado de regiones, únicamente queda crear una vista personalizada subtipo de `TextView`, y sobrescribir su método `OnDraw`, llamando en orden al código de dibujado de regiones de líneas, seguido del código de dibujado de regiones de columna, y finalizando con la llamada al método `OnDraw` de la superclase, que será donde se dibuje el texto:

```
protected override void OnDraw(Canvas canvas)
{
    canvas.DrawColor(Color.White);

    if (DrawingInfo == null || LineRanges == null)
    {
        base.OnDraw(canvas);
        return;
    }

    InitLineHeight();

    PaintLineRegions(canvas, DrawingInfo.GetTextRegions());
    PaintInsideLineRegions(canvas, DrawingInfo.GetInsideLineRegions());

    base.OnDraw(canvas);
}
```

FIGURA 94. DIBUJADO DE REGIONES EN DIFFERENCESTEXTVIEW

Cabe destacar que si se asigna a la `TextView` en el fichero de *layout* correspondiente un color de fondo, este será dibujado después de las regiones (`base.OnDraw(canvas)`), por lo que estas no se verían reflejadas en el dibujo final.

Capítulo VI – Conclusiones

Al principio del presente documento se planteó el desarrollo del cliente móvil de Plastic SCM como una tarea exploratoria para comprobar hasta qué punto era viable, y de qué formas podía este resultar útil a la cartera de clientes existente. Tras haber creado un cliente para tabletas que cuenta con las partes fundamentales de sus contrapartes de Windows, GNU/Linux, y macOS, tales como el Branch Explorer, las diferencias *side by side*, y el sistema de consultas, puede decirse que dicho desarrollo se ha completado de manera exitosa, sentando la base para actuaciones futuras que completen la visión del producto planeada inicialmente –más sobre esto en "líneas de actuación futuras"). Se ha creado un nuevo cliente que se integra, tanto en apariencia como en comportamiento, en el ecosistema de aplicaciones ya existentes en torno a Plastic SCM, funcionando, sin ninguna adaptación en el otro extremo, con cualquier servidor de Plastic SCM en versiones posteriores a la 5.4.16.752.

Mediante la metodología ágil *scrum*, la elaboración de historias de usuario para identificar la funcionalidad crítica, y dividiendo dicha funcionalidad en tareas estimadas temporalmente, desarrollándolas en momentos puntuales incluso en paralelo gracias a un correcto uso del software de control de versiones que ocupa este documento (el desarrollo del cliente móvil de Plastic SCM se ha realizado utilizando, como no podía ser de otra forma, Plastic SCM), he podido reforzar aquellos conocimientos, adquiridos tanto en la Universidad como en el lugar de trabajo, relativos a la gestión de proyectos y a la gestión de configuraciones.

Debiendo crear una interfaz de usuario para tabletas que resultase familiar a aquellas personas acostumbradas al resto de clientes de Plastic SCM en el escritorio, con los primeros bocetos en papel y las rápidas iteraciones de los mismos, siempre realizadas en contacto con el resto de integrantes del equipo de desarrollo de Códice Software, apliqué aquellas metodologías aprendidas en asignaturas tales como *Interacción Persona Computador*.

A la hora de integrar código ya existente en una nueva base pude observar y poner en práctica patrones de diseño aprendidos en asignaturas como *Desarrollo Basado en Componentes y Servicios* o *Diseño de Software*. Si bien en análisis, diseño y desarrollo iterativos -en lugar de en cascada- no se favorece la exhaustividad en cada una de esas etapas de forma temprana, no generando gran cantidad de artefactos tales como diagramas o documentación, los conocimientos sobre estas fases resultaron, igualmente, imprescindibles para el éxito del proyecto.

En definitiva, este Trabajo de Fin de Grado ha servido a dos propósitos. El primero de ellos, el más importante, es el de haber podido afianzar y practicar conocimientos y habilidades adquiridas en las aulas durante los últimos años, estando al cargo de un proyecto a lo largo de todas sus fases -como contrapunto a los proyectos en aulas, que tienden a centrarse únicamente en ciertos aspectos de los mismos-. El segundo es haber podido proporcionar a Códice un nuevo cliente con el que podrán continuar su exploración del mercado móvil si así lo desean, trabajando junto a un equipo profesional de desarrollo que en todo momento ha actuado como mentor, poniendo pacientemente sus conocimientos y su experiencia a disposición del autor de estas líneas.

6.1 - Líneas de actuación futuras

Una vez se ha completado el desarrollo del *producto mínimo viable*, si entra en los intereses de futuro de Códice Software, una de las primeras líneas de actuación sería iniciar una distribución de la aplicación como *beta cerrada*, a un conjunto limitado de clientes, para esclarecer si realmente el cliente móvil ligero puede llegar a tener demanda, y, en caso afirmativo, cuál es la funcionalidad restante para ser realmente útil para dichos clientes. La distribución de software en estado *beta* es una práctica habitual en Códice Software, con contacto personal y directo a través de correo electrónico con los posibles interesados en participar en encontrar fallos en los productos y dar su opinión respecto a los mismos a cambio de poder utilizarlos durante una temporada gratis o a precio reducido. Un ejemplo de ello es la *beta* de SemanticMerge 2.0, distribuida a lo largo del mes de junio de 2016.

Entre los casos de uso iniciales contemplados para el cliente ligero de Plastic SCM, si bien no se contemplaba la posibilidad de crear *workspaces* completos, sí se había previsto la funcionalidad de descargar revisiones concretas de ficheros. Finalmente, dicha funcionalidad quedó fuera del llamado *mínimo producto viable*, pero si el *feedback* de los clientes que reciban la aplicación resulta positivo, podría incluirse en siguientes revisiones. Sin embargo, en actualizaciones futuras de Chrome OS [35], el sistema operativo de Google para ordenadores portátiles, se va a poder ejecutar aplicaciones de Android, tal y como anunció la compañía en su conferencia Google I/O 2016. A pesar de que no existan entornos de desarrollo completos para Android, sí existen editores de texto, editores de imágenes, y *suites* ofimáticas como Microsoft Office, por lo que podría resultar de interés que la aplicación móvil de Plastic SCM recibiese una funcionalidad parecida a la de otro de los clientes de la empresa, *Gluon*, que permite mantener un *workspace* parcial en vez de completo, con la posibilidad de subir de nuevo los cambios al repositorio.

En este cliente ligero no se ha abarcado el testeo del mismo. Todo el código que se está reutilizando de los clientes de escritorio ya está siendo testeado regularmente (de hecho, como mínimo cada vez que un miembro del equipo finaliza una tarea), por lo que duplicar dichos tests en este proyecto habría carecido de sentido. Sí faltan, sin embargo, tests de interfaz de usuario. Los clientes de GNU/Linux y macOS, así como Gluon, son probados mediante un sistema de testing desarrollado en la propia empresa, prescindiendo así de soluciones comerciales como *TestComplete*[10]. Sería de interés integrar también el testing de las aplicaciones móviles en dicho sistema. De no resultar posible a nivel técnico, existen igualmente *frameworks* para elaborar tests de interfaz de usuario compatibles con Xamarin.Android y Xamarin.iOS, como TestComplete Mobile, o más cercano a las tecnologías de Xamarin utilizadas para el desarrollo del proyecto, Xamarin.UITest, que además se integra con Visual Studio (el IDE de preferencia en Códice Software).

[10] Solcuión de testing automático desarrollada por la empresa estadounidense SmartBear. Proporcionan también software para la automatización de tests de usuario en aplicaciones Android e iOS.

Bibliografía

[1] B. Ruiz Arroyo and P. Santos Luaces, "Method for determining similarity of text portions". Europa Patent EP2390793 A1, 30 Noviembre 2011.

[2] R. Pichler, Agile Product Management with Scrum - Creating products that customers love, Addison-Wesley, 2010.

[3] M. Cohn, User Stories Applied, Addison-Wesley, 2004.

[4] Google Inc., "Dialogs - Components - Google design guidelines," [Online]. Available: https://material.google.com/components/dialogs.html. [Accessed 28 Mayo 2016].

[5] Google Inc., "Bottom navigation - Components - Google design guidelines," [Online]. Available: https://material.google.com/components/bottom-navigation.html. [Accessed 15 Junio 2016].

[6] Códice Software S.L., "Branch per Task guide in Plastic SCM," [Online]. Available: https://www.plasticscm.com/branch-per-task-guide/index.html. [Accessed 10 Junio 2016].

[7] Google Inc., "Activity," [Online]. Available: https://developer.android.com/reference/android/app/Activity.html. [Accessed 10 Junio 2016].

[8] Google Inc., "Activities," [Online]. Available: https://developer.android.com/guide/components/activities.html#Lifecycle. [Accessed 10 Junio 2016].

[9] Google Inc., "How Android draws views," [Online]. Available: https://developer.android.com/guide/topics/ui/how-android-draws.html. [Accessed 10 Junio 2016].

[10] Google Inc., "Canvas," [Online]. Available: https://developer.android.com/reference/android/graphics/Canvas.html. [Accessed 10 Junio 2016].

[11] Google Inc., «Como admitir varias densidades de pantalla,» [En línea]. Available: https://developer.android.com/training/multiscreen/screendensities.html. [Último acceso: 10 Junio 2016].

[12] Wikipedia, the free Encyclopedia, «Pointing device gesture,» [En línea]. Available: https://en.wikipedia.org/wiki/Pointing_device_gesture. [Último acceso: 10 Junio 2016].

[13] Google Inc., "Animating a Scroll Gesture," [Online]. Available: https://developer.android.com/training/gestures/scroll.html. [Accessed 2016 Junio 13].

[14] Google Inc., «GestureDetector,» [En línea]. Available: https://developer.android.com/reference/android/view/GestureDetector.html. [Último acceso: 13 Junio 2016].

[15] Google Inc., «GestureDetector.SimpleOnGestureListener,» [En línea]. Available: https://developer.android.com/reference/android/view/GestureDetector.SimpleOnGestureListener.html. [Último acceso: 13 Junio 2016].

[16] Google Inc., «ScaleGestureDetector,» [En línea]. Available: https://developer.android.com/reference/android/view/ScaleGestureDetector.html. [Último acceso: 13 Junio 2016].

[17] Google Inc., «ScaleGestureDetector.SimpleOnScaleGestureListener,» [En línea]. Available: https://developer.android.com/reference/android/view/ScaleGestureDetector.SimpleOnScaleGestureListener.html . [Último acceso: 13 Junio 2016].

[18] Google Inc., «Context,» [En línea]. Available: https://developer.android.com/reference/android/content/Context.html. [Último acceso: 13 Junio 2016].

[19] J. Steele and N. To, The Android Developer's cookbook, Addison-Wesley, 2011.

[20] R. Rogers, J. Lombardo, z. Medieks and B. Meike, Android Application Development, O'Reilly, 2009.

[21] S. H. Hashimi and S. Komatineni, Pro Android, Apress, 2009.

[22] Apple Inc., «Matrix Transforms,» [En línea]. Available: https://developer.apple.com/library/safari/documentation/AudioVideo/Conceptual/HTML-canvas-guide/MatrixTransforms/MatrixTransforms.html . [Último acceso: 13 Junio 2016].

[23] J. Hughes, A. Van Dam, M. McGuire, D. F. Sklar, J. K. F. S. D. Foley y K. Akeley, Computer Graphics - Principles and Practice (third edtion), Addison-Wesley, 2014.

[24] J. E. Howland, «Representing 2D Transformations as Matrices,» 28 Junio 2004. [En línea]. Available: http://www.cs.trinity.edu/~jhowland/cs2322/2d/2d/. [Último acceso: 4 Julio 2016].

[25] M. Larabel, "The Skia 2D Graphics Library from Google - Phoronix," 18 Abril 2011. [Online]. Available: http://www.phoronix.com/scan.php?page=news_item&px=OTMoMw . [Accessed 13 Junio 2016].

[26] Google Inc., «Welcome to the Android Open Source Project,» [En línea]. Available: https://source.android.com/. [Último acceso: 13 Junio 2016].

[27] Google Inc., «Canvas.cpp source code,» [En línea]. Available: https://android.googlesource.com/platform/frameworks/base.git/+/android-4.2.2_r1/core/jni/android/graphics/Canvas.cpp. [Último acceso: 14 Junio 2016].

[28] Google Inc., «SkCanvas.cpp source code,» [En línea]. Available: https://github.com/google/skia/blob/master/src/core/SkCanvas.cpp . [Último acceso: 14 Junio 2016].

[29] R. Guy, «2048 pixels limit of hardware acceleration,» 25 Octubre 2012. [En línea]. Available: https://groups.google.com/forum/#!topic/android-developers/HGkFV4RAAAM. [Último acceso: 14 Junio 2016].

[30] Google Inc., "Layout," [Online]. Available: https://developer.android.com/reference/android/text/Layout.html#getPaint(). [Accessed 15 Junio 2016].

[31] Apple Inc., «Defining a Custom View,» [En línea]. Available: https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/CreatingViews/CreatingViews.html#//apple_ref/doc/uid/TP40009503-CH5-SW23. [Último acceso: 2016 Junio 27].

[32] Apple Inc., «Metal for developers,» [En línea]. Available: https://developer.apple.com/metal/. [Último acceso: 28 Junio 2016].

[33] Apple Inc., «CATiledLayer Class Reference,» [En línea]. Available: https://developer.apple.com/library/ios/documentation/GraphicsImaging/Reference/CATiledLayer_class/index.html#//apple_ref/occ/cl/CATiledLayer. [Último acceso: 27 Junio 2016].

[34] P. Santos Luaces, «Track refactored code across files with Plastic SCM,» 11 Agosto 2015. [En línea]. Available: http://blog.plasticscm.com/2015/08/track-refactored-code-across-files-with.html. [Último acceso: 29 Junio 2016].

[35] D. Redi y E. Taylor, «The Google Play store, coming to a Chromebook near you,» 19 Mayo 2016. [En línea]. Available: https://chrome.googleblog.com/2016/05/the-google-play-store-coming-to.html . [Último acceso: 4 Julio 2016].

[36] Microsoft Corporation, «.NET Framework Remoting Overview,» 7 Febrero 2011. [En línea]. Available: https://msdn.microsoft.com/en-us/library/kwdt6w2k(v=vs.100).aspx . [Último acceso: 4 Julio 2016].

Anexo A: Decisión tecnológica de Xamarin respecto a un desarrollo nativo

A la hora de escoger un *framework* para el desarrollo de la aplicación móvil había que tener en cuenta una serie de factores en base a los cuales dicho desarrollo podría complicarse hasta el punto de resultar inviable, bien por cuestiones de tiempo, bien por dificultad técnica.

El primero de estos factores, crítico para el éxito del proyecto, es que la aplicación fuese compatible con la nueva capa de red de Plastic SCM, *plasticpipe* (*plasticprotocol*). Esto comprendería sencillamente la capacidad de leer y escribir en un socket la información necesaria siguiendo el protocolo, especificado y documentado recientemente. Ya existen pruebas internas de interoperabilidad con un cliente ligero escrito en Go capaz de ejecutar unos pocos métodos sencillos (como el listado de repositorios), por lo que el lenguaje escogido, de no ser C#, no haría imposible dicha comunicación, pero sí impondría la necesidad de reescribir la parte de *plasticpipe* correspondiente a *plasticprotocol*, aumentando el tiempo de desarrollo y siendo susceptible a la introducción de *bugs* ya eliminados y asegurados mediante el sistema de *testing* de los clientes de escritorio y del servidor de Plastic SCM. Adicionalmente, reescribir el código de red en otro lenguaje forzaría a mantener dos bases de código paralelas cada vez que se quiera introducir un cambio en dicha capa, cosa poco habitual pero plausible (añadir nuevos métodos, solucionar fallos en -o aumentar la eficiencia de- la serialización de ciertos tipos, etc).

El segundo de estos factores, crítico también, es la necesidad de poder realizar en el cliente el cálculo de diferencias de ficheros de texto. Dicho cálculo no es trivial, y, aunque reescribirlo tampoco resultaría imposible (al fin y al cabo, no se apoya en ningún API exclusivo de Windows o de .NET Framework, o que no tenga equivalente en, por ejemplo, Java), sí sería una tarea que consumiría una gran cantidad de horas, de nuevo aumentando la posibilidad de introducir fallos que en la base de código de C# no existen, duplicando en el camino código (aunque este ya sea lo suficientemente estable como para apenas recibir cambios).

El tercer factor tiene que ver con la calidad del producto percibida por el cliente. Aunque la aplicación cumpla su función sin fallos y de manera fluida, un mal aspecto visual puede arruinar la experiencia de usuario. Anteriormente, Plastic SCM tenía una única interfaz, la del sistema operativo Windows, y esta llegaba a los clientes que trabajaban con GNU/Linux y con macOS a través de distintas librerías de compatibilidad que permitían el funcionamiento de *Windows Presentation Foundation* en estos entornos. Sin embargo, esto hacía de la GUI de Plastic SCM algo percibido por los clientes como *"estridente"*: su aspecto visual no era acorde a ninguna guía de estilo de dichos sistemas operativos, y además sufría de frecuentes fallos debido a problemas en las librerías de compatibilidad. Tras los esfuerzos del equipo en llevar interfaces nativas a GNU/Linux y macOS, era deseable ofrecer también la misma experiencia en las plataformas móviles.

Poder trabajar con C#, de forma que se reutilizasen partes de la base de código ya existente, así como proporcionar interfaces de usuario nativas en Android e iOS, es algo que proporcionaba Xamarin en sus versiones Xamarin.Android y Xamarin.iOS. Con esto no se quiere decir que el código fuese directamente compatible: por ejemplo, a pesar de que, en Windows, GNU/Linux, y macOS, a través de las clases del *namespace* `Path` se pueda acceder fácilmente a los ficheros, en Android no existe un equivalente directo, por lo que fue necesario adaptar determinadas partes.

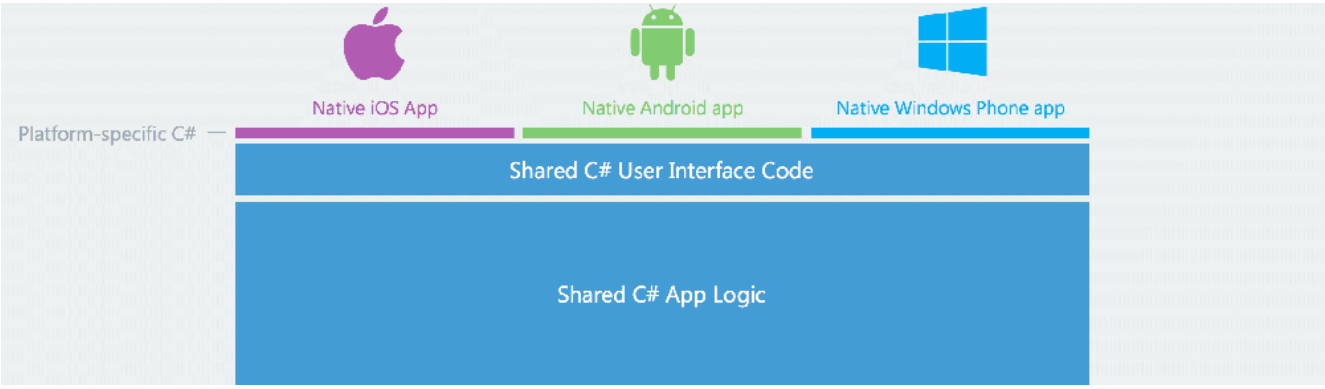

FIGURA 95. ARQUITECTURA TÍPICA DE UNA APLICACIÓN CON XAMARIN

Sin embargo, poder disponer de la capa de red y del cálculo de diferencias sin necesidad más que de seleccionar aquellas clases necesarias del cliente, añadiendo compilación condicional donde fuese preciso para esquivar funcionalidad que, o bien no se requiere en el cliente, o bien no se puede proporcionar, aceleró el desarrollo hasta el punto de que la mayoría de código es aquel dependiente de la plataforma (dibujado de las diferencias, dibujado del Branch Explorer, navegación entre actividades, guardado y recuperación de preferencias de usuario, etc).

Anexo B: *Plasticprotocol*

Como se ha detallado en el *Capítulo II – Contexto tecnológico*, Plastic SCM es un programa configurado como un cliente y un servidor que se comunican a través de la red. Para ello, ambos se apoyan en una base común de código, denominada internamente *plasticpipe*, y unos tipos comunes definidos en el *assembly*[11] `commontypes`.

Cuando se comenzó a desarrollar Plastic SCM, el protocolo de comunicación se apoyó sobre .NET Remoting [38], una tecnología de Microsoft para invocación de métodos remotos (RPC) que ya está considerada obsoleta (reemplazada por *Windows Comunication Foundation*). Como cliente y servidor dependen de `commontypes`, cualquier cambio en dicho *assembly* provocaba cambios en la serialización que hace .NET Remoting de los tipos (ya que incluye el número de versión del *assembly* en el que se definen, entre otra información), y, por lo tanto, se rompía la compatibilidad entre cliente y servidor que no utilizasen la misma versión de `commontypes`.

Además, otro de los problemas que plantea .NET Remoting es la saturación en la comunicación. Tests internos situaron la pérdida por metadatos en la descarga en un 30% del total de la comunicación, mientras que en la carga ascendía a un 70%. Esto es debido a que, para la invocación de un método, Remoting envía, en Unicode, a través de la red, el nombre completo de la interfaz donde dicho método esté definido, además de la información completa de la versión de `commontypes`. La carga extra de datos realmente innecesarios para el funcionamiento de Plastic SCM, más allá de Remoting, empeoraba en operaciones que necesariamente se han de hacer encadenadas, como por ejemplo resolver un nombre de usuario bajo ciertas configuraciones de seguridad.

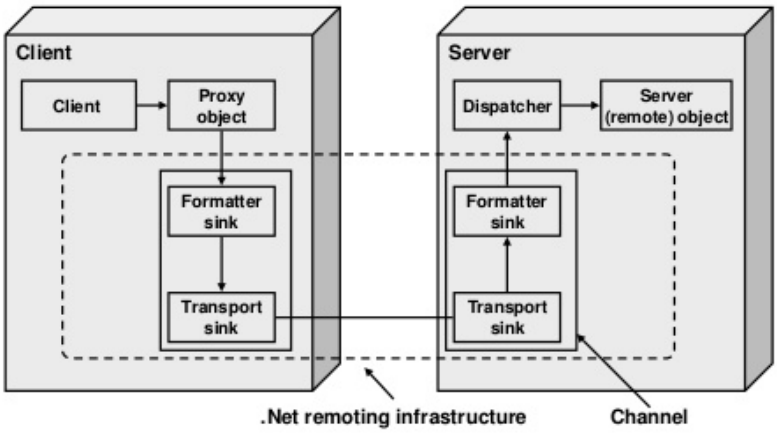

FIGURA 96. ARQUITECTURA DE .NET REMOTING

Para tipos o bien complejos o bien con una gran carga de información, el equipo de Códice definió una serialización personalizada, incluyendo compresión a nivel del *transport sink*, y escribiendo en el *stream* de red únicamente tipos primitivos en el mismo orden en el que se fuesen a leer al otro lado, sin incluir ninguna información adicional como definiciones de los mismos. Esto eliminaba una de las ventajas de Remoting, la

[11] En el contexto de .NET, un *assembly* es un conjunto de código pre-compilado que puede ser ejecutado por el .NET Runtime. En el caso concreto de `commontypes`, el término assembly se refiere a DLL, *dynamic-link library*, una librería que es cargada por el *runtime* en el momento de ser necesitada.

serialización transparente, en aras de mejorar el rendimiento general de las operaciones críticas como el *checkin*.

El estudio para reemplazar .NET Remoting por un protocolo de comunicación propio -bautizado como *plasticprotocol*- empezó en 2014, pero no fue hasta finales de 2015 cuando se empezó a trabajar en ello, y no es hasta el momento de escribir estas líneas, junio de 2016, que está prácticamente finalizado.

Aunque la nueva capa de red de Plastic SCM no forma parte del alcance de este trabajo, participé activamente en su desarrollo entre los meses de noviembre de 2015 y febrero de 2016. Debido a la importancia de *plasticprotocol* para que el cliente ligero de tabletas sea posible, se indica a continuación en líneas generales su funcionamiento.

La funcionalidad *core* (o principal) de Plastic SCM se divide en *handlers*. Así, por ejemplo, el `ItemHandler` se encarga -entre otras- de operaciones tales como las *queries* y el *checkin*, el `RepositoryHandler` de la creación y eliminación de repositorios, el `BranchHandler` de la creación de ramas, etc.

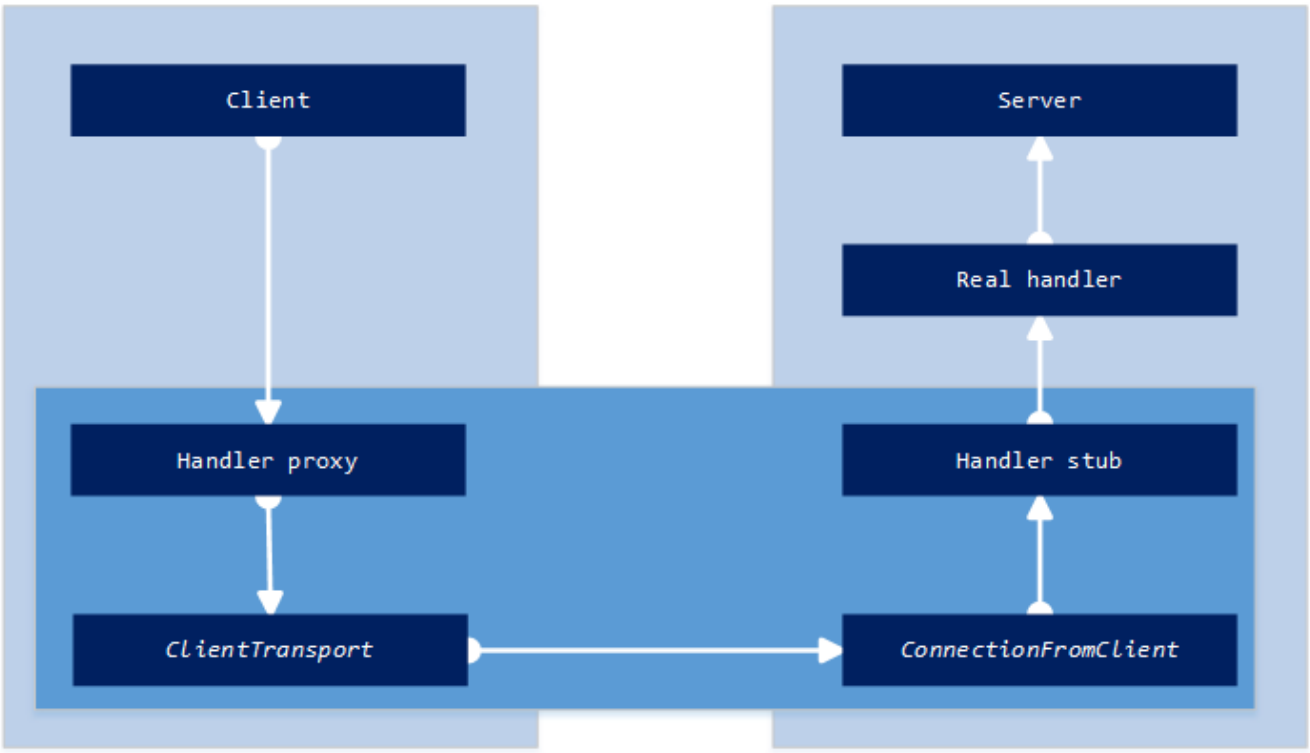

FIGURA 97. ARQUITECTURA DE PLASTICPROTOCOL

En *plasticprotocol*, por cada *handler* el cliente crea un *proxy*, y el servidor crea un *stub*. A la hora de invocar un método, el cliente envía, a través de `ClientTransport`, entre otra información, un byte que indica el número de método que desea invocar, seguido del mensaje, que contiene, a su vez, un byte indicando número de versión del mismo -para garantizar compatibilidad hacia atrás- y los argumentos que reciba el método que se está invocando.

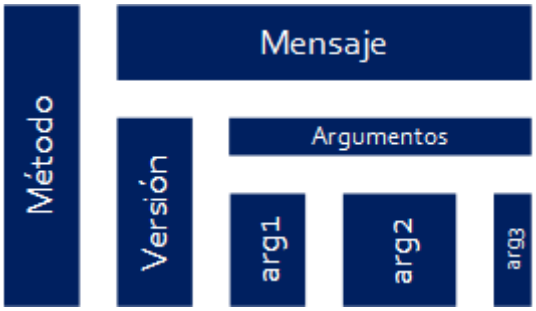

FIGURA 98. PARTE DE LA TRAMA DE INVOCACIÓN DE UN MÉTODO EN PLASTICPROTOCOL

El servidor, en `ConnectionFromClient`, determina, de entre los stubs registrados, aquel que declare que puede ejecutar el método especificado en el byte de método. Una vez identificado, le pasa el resto del *stream*, con el que el *stub* lee primero la versión del mensaje, para después deserializar el mensaje correcto y ejecutar el método con los argumentos que contenga el mensaje deserializado.

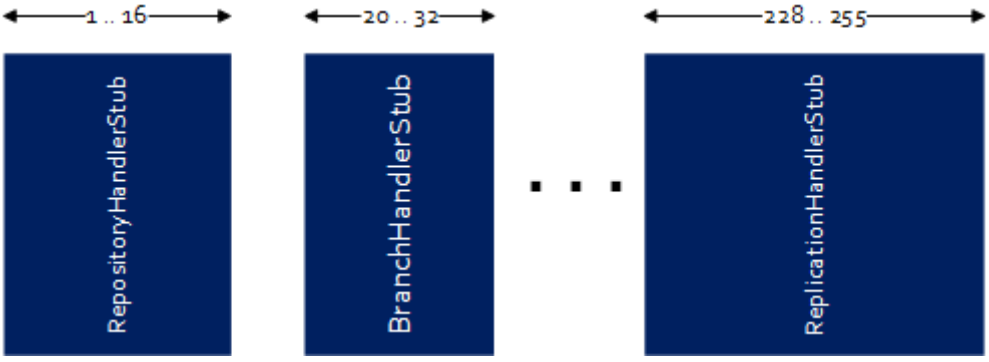

FIGURA 99. DECLARACIÓN DE MÉTODOS POR *STUB* EN PLASTICPROTOCOL

Una vez el servidor ha ejecutado el método con el *handler* real (y no con el *stub*), si no ha habido ningún error, se envía un primer byte de estado indicando el éxito de la operación, seguido del mensaje de respuesta, que, igualmente, incluye un byte de versión. Si ha sucedido alguna excepción en la ejecución del método, al cliente se le envía un byte de estado indicando el error, seguido de la excepción serializada.

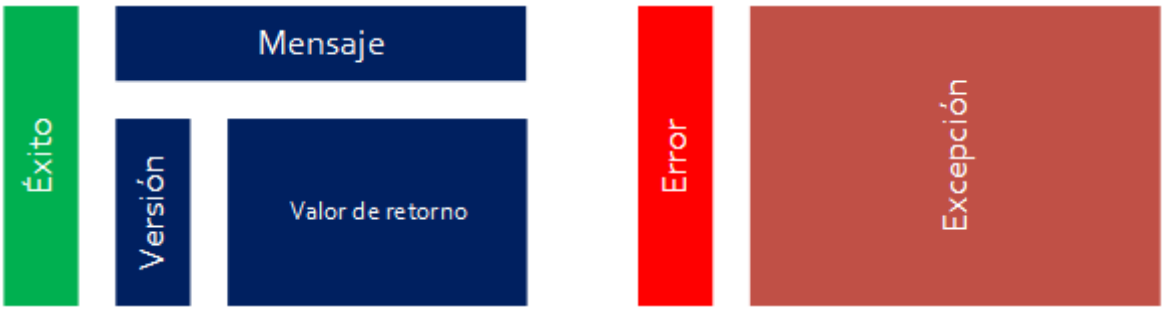

FIGURA 100. MENSAJES DE ÉXITO Y DE ERROR

Gracias al nuevo *plasticprotocol*, a pesar de utilizar versiones distintas del *assembly* `commontypes`, se ha conseguido que las versiones 5.0 y 5.4 de Plastic SCM sean compatibles entre sí, y es posible que el cliente de tabletas del que este Trabajo de Fin de Grado se conecte a servidores con versión igual o posterior a la 5.4.16.752 a pesar de no tener ninguna versión de `commontypes` en absoluto.