TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

"Diseño Dirigido por Modelos de Aplicaciones para Dispositivos Móviles"

Curso 2016/2017

Alumno/a:

Tarik Imlahi Rivas

Director/es:

Jesús Manuel Almendros Jiménez

Agradecimientos

En primer lugar, quiero dar las gracias a mi tutor *Jesús Manuel Almendros Martínez* por haberme ayudado en la elaboración de mi proyecto y el haberme dado libertad para desarrollarlo y llevarlo a cabo con mi enfoque y a mi ritmo.

Seguidamente, quiero agradecer a todos los profesores que me han impartido clase en la carrera, a los que me lo pusieron fácil y sobre todo a los que me lo pusieron difícil, ya que en el desarrollo de mi proyecto, he tenido que echar mano de casi todo lo que en un día quise olvidar una vez aprobada la asignatura dada. Menos mal que no lo hice. Cuánta razón tenía cuando nos dijo una vez *D. Antonio Corral Liria*: "Sabéis más de lo que os imagináis". Esta frase ha hecho eco en mi cabeza durante todo este tiempo

Como no, debo de agradecer a mis padres *Layachi* y *Matilde* que día a día y a lo largo de toda mi vida, hayan creído en mi persona y en mi capacidad más que yo mismo. A mis hermanos *Nadia* y *Youssef* por su apoyo, a mis amigos por aburrirles con mis discursillos informáticos, y muy especialmente a mi esposa *Patricia Pérez Oliván* por su paciencia y ayuda, y a mi hijo *Elías* por darle sentido a mi vida así como la motivación que me hace falta para luchar día a día.

También, doy un fuerte agradecimiento a mi querido amigo, que ya no está presente entre nosotros, *Emilio Muñoz Esteban*, informático y grandísima persona, por haberme guiado en todos nuestros años de amistad y haber sido el mejor amigo que alguien pudiese desear.

Por último, quiero agradecer de corazón la ayuda recibida por mis compañeros de clase en más de una ocasión y decir que sin ellos no hubiese sido ni la mitad de ameno

Tabla de contenido

1 Introducción.

En este proyecto se estudiará cómo adaptar la metodología utilizada en las asignaturas de MDS 1 y MDS 2 al desarrollo de aplicaciones móviles. La metodología impartida en estas asignaturas está basada en UML, y se basa en el uso de diagramas de casos de uso, diagramas de clases, diagramas de estados y diagramas de secuencia para el modelado de la aplicación.
Los **diagramas de casos de uso y de clases** sirven para modelar la estructura de la aplicación: interfaces de usuario y base de datos. Los **diagramas de estados** sirven para modelar la lógica de los interfaces de usuario. Finalmente, los **diagramas de secuencia** sirven para modelar la interacción entre la interfaz de usuario y la base de datos.
A partir de estos modelos se realiza la implementación generando código con las herramientas Eclipse y Visual Paradigm. Por un lado, los interfaces de usuario se diseñan con Visual Paradigm y su diseño (en forma de diagrama de clases) se exporta a clases Java. Por otro lado, la base de datos, diseñada con el diagrama de clases, se implementa gracias a la generación de código ORM de Visual Paradigm.

Este proyecto, por tanto, se enmarca, dentro del campo conocido como **"Desarrollo dirigido por Modelos"**, que ha tomado auge en los últimos años tanto a nivel académico como industrial. Los modelos forman parte de los artefactos que el ingeniero debe manejar, y son pieza fundamental en el desarrollo del software actual. Este auge ha venido provocado por la aparición de herramientas cada vez más potentes que manejan modelos como parte del proceso del desarrollo del software.

El objetivo primordial de este estudio es cómo la metodología basada en UML que se utiliza para el desarrollo de aplicaciones de escritorio puede ser adaptada a un contexto diferente. En particular, debemos estudiar por un lado qué herramientas son las más adecuadas. Por un lado, tenemos que disponer de una herramienta de modelado en UML. Por otro necesitamos una herramienta para la codificación de aplicaciones móviles. Pero no solo esto es necesario, sino que además deberíamos poder desarrollar la codificación apoyada en los modelos que hemos descrito, al igual que se hace en MDS 1 y MDS 2. Nos encontramos con los siguientes problemas. Por un lado, la herramienta utilizada en MDS 1 y MDS 2 (Visual Paradigm), está pensada principalmente para el diseño de aplicaciones Java, y genera código Java.

El estudio previo que realizamos para decantarnos por una plataforma en la que desarrollar este proyecto, nos reveló varias opciones para profundizar e intentar encajar dicha metodología, como, por ejemplo:

- Integración de Visual Paradigm con *Android Studio* [1]
- Integración de Visual Paradigm con Visual Studio y usando *Xamarin*[2]
- Integración de Visual Paradigm con Eclipse

Aun probando que las posibles integraciones de Visual Paradigm con las distintas plataformas eran factibles, nos surgieron ciertos reparos a la hora de comenzar a experimentar, ya que, aunque finalmente fuese posible encajar la metodología, existía el factor de la curva de aprendizaje de cómo implementar aplicaciones para dispositivos móviles no habiendo hecho nunca ninguna.

Sin embargo, para el desarrollo de aplicaciones móviles disponemos de otras herramientas (Visual Studio y Xamarin), que permiten diseñar e implementar aplicaciones móviles. En particular, estas herramientas permiten diseñar el interfaz de usuario e implementar los métodos del interfaz de usuario. Hemos elegido Visual Studio conjunto sus herramientas propias de modelado y el componente Xamarin para desarrollo en C#[3]de aplicaciones móviles multiplataforma. Nos decantamos por esta opción por tener más acercamiento al lenguaje de C# y disponer de experiencia gracias a las asignaturas en las que se ha cursado este lenguaje conjuntamente a los distintos Patrones de Diseño[4], por lo que a la hora de implementar facilitaría el trabajo y sería mucho más rápido la adaptación y menos pronunciada la curva de aprendizaje.

En cuanto a la herramienta de modelado, hemos mantenido Visual Paradigm para los diagramas de casos de uso y los diagramas de estados. Los diagramas de clases y de secuencia han sido modelados con las herramientas de modelado propias de Visual Studio. El código ha sido generado con la herramienta Xamarin para Visual Studio. Por último, hemos utilizado para la base de datos, servicios Webs implementados con la herramienta Ruby On Rails.
La aplicación que hemos desarrollado es muy sencilla en su funcionamiento, pero el objetivo principal del proyecto no era realizar una aplicación compleja sino mostrar cómo realizar el proyecto MDS1 y MDS 2 en el contexto de dispositivos móviles.

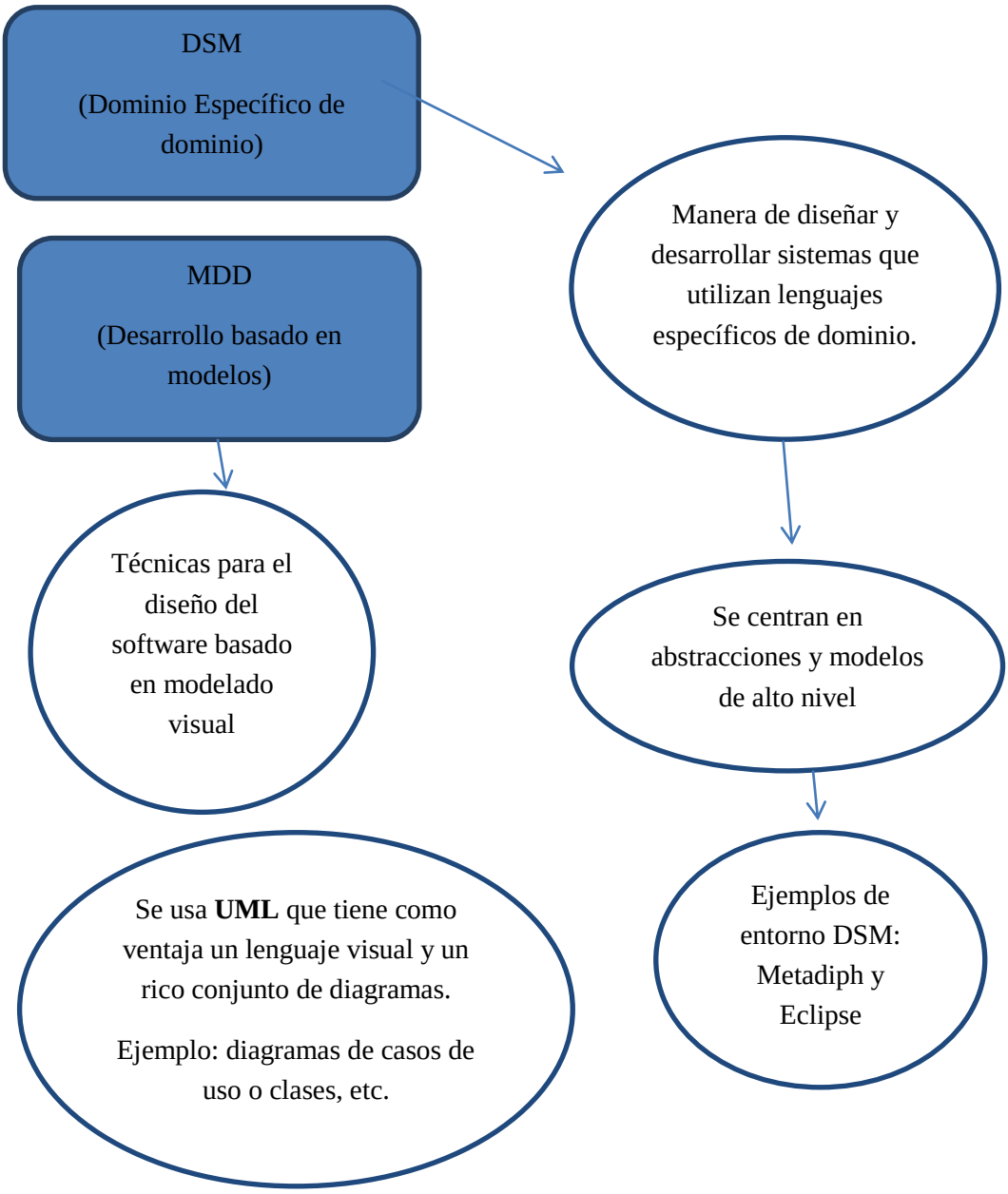
Esta memoria va a mostrar de principio a fin cómo aplicar la metodología en Xamarin empezando por los conceptos básicos de cómo desarrollar hasta cómo Modelar con Xamarin. Servirá como guía para futuros desarrolladores que quieran empezar a usar esta plataforma y mostrará el camino más rápido a la solución evitando los obstáculos a los que nos hemos visto enfrentados en todo el transcurso de este proyecto.

.

2 Metodología de Modelado en UML

Este apartado está basado en el artículo "UML Modeling of User and Database Interaction" publicado en la revista *Computer Journal* donde los autores *Jesús Manuel Almendros Jiménez y Luis Fernando Iribarne Martínez* presentan una técnica de modelado y diseño basado en UML para la interacción de usuarios con la aplicación y de la aplicación con la base de datos[5].

Este apartado tratará de plasmar de forma clara y sencilla cómo se organiza paso a paso esta técnica de modelado y diseño.



Un entorno DSM puede proveer arquitecturas que soportan aplicaciones de negocio cuyas principales tareas son:

- **Interacciones de usuario:** Se realizan a través de interfaces de usuario y pueden ser tanto de entrada (Cliente) como de salida (Servidor).
- **Interacciones de BBDD:** Consisten en consultas y actualizaciones desde un servidor de BBDD y son debidas a las interacciones de usuario. Incluyen los procesos Cliente y servidor.

En una aplicación típica C/S el usuario interactúa con el sistema a través de una interfaz estática o dinámica que interactúa con la BBDD.

Una arquitectura adecuada separaría aspectos como la lógica de presentación, lógica de negocio y transacción de BBDD.

Uno de los propósitos es el Modelo Vista Controlador (MVC):

- **Vista:** Diseña patrones para interfaces de usuario dándole soporte visual y mostrando el estado del modelo de componentes.
- **Modelo:** Es el que se encarga de interactuar con la BBDD.
- **Controlador:** Es la parte que relaciona las otras dos capas (lógica) relaciona las interacciones del usuario con las interacciones en la BBDD.

En este apartado se presentarán las técnicas de diseño para interacciones de usuario y base de datos, las mismas se modelarán en diagramas de clases, de estado y de secuencias.

En los diagramas de interacciones de usuario se representarán las interacciones de salida, las consultas de datos y las transacciones representarán las interacciones de entrada.

Pasos a Seguir para el modelado en UML de la interacción de usuarios y bases de Datos

PASO 1: Construcción de Diagramas de Casos de Uso

En este primer paso el objetivo es crear un diagrama de casos de uso en el que se identifican los actores u los casos de uso. Estos están conectados por asociaciones. Generalizaciones entre actores y generalizaciones e inclusiones entre casos de uso.

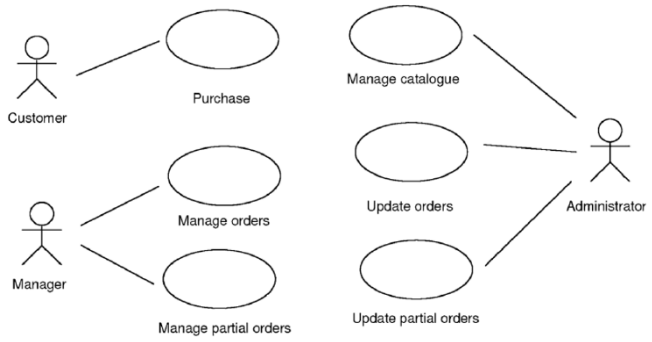
Generalización entre actores: Puede haber varias clases particulares de actores y que compartan roles con otro más general.

Generalización entre casos de uso: Denota un caso más particular de un caso de uso

Inclusión entre casos de uso: A partir de un caso de uso se puede acceder a otros casos de uso.

Ejemplo:

- **El Cliente:** Interacciona con algunas ventanas para consultar y comprar.
- **Manager de pedidos**: Gestiona los pedidos del cliente.
- **Administrador:** Gestiona el catálogo, actualiza los pedidos, inserta, elimina, etc.



Cada actor es representado y asociado con las tareas que puede realizar

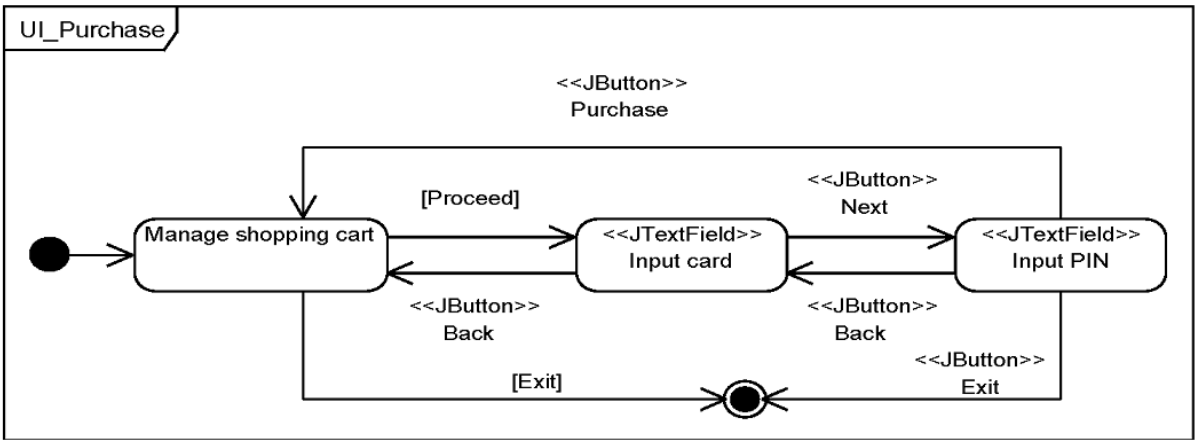
PASO 2: Construcción de Diagramas de Interacción de Usuario

Consiste en crear diagramas de estado para la descripción de la interfaz de usuario.

Cada caso de uso (por ejemplo: tarea) necesita una interfaz de usuario (por ejemplo: ventana). Cada interfaz se describe por medio de un diagrama de interacción de usuario en el que aparecen las interacciones de entrada/salida y las transiciones del usuario (acciones).

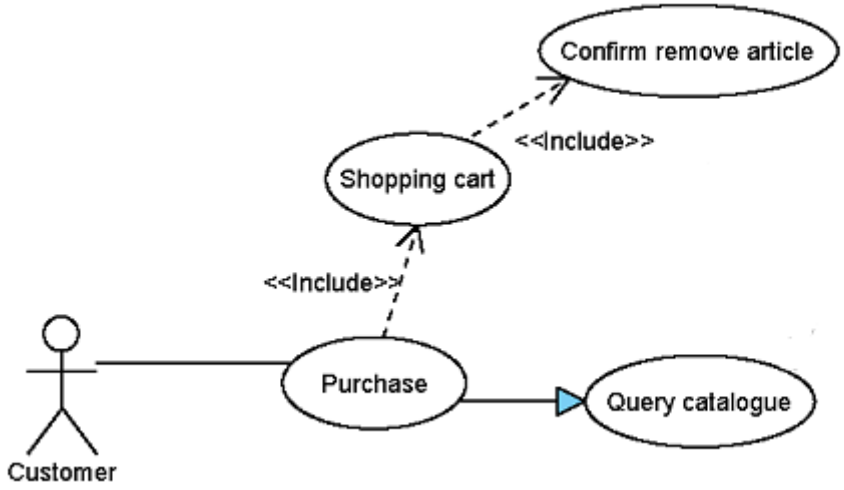
Los diagramas de interacción pueden compararse por medio de relaciones de inclusión y generalización.

Ejemplo: Se describe el proceso del cliente para adquirir un libro, es decir, como interactúa con los componentes de la interfaz de las futuras ventanas del sistema. Los estados y transiciones son estereotipados con el nombre del componente de la interfaz de usuario.



PASO 3: Construcción de Diagramas de Interfaces de Usuario

Aquí se construye una versión especializada del diagrama de casos de uso construido en el PASO 1, en el que cada caso de uso representa o una tarea o estados en los que los casos de uso han sido descompuestos en diagramas de interacción de usuario propios del PASO 2



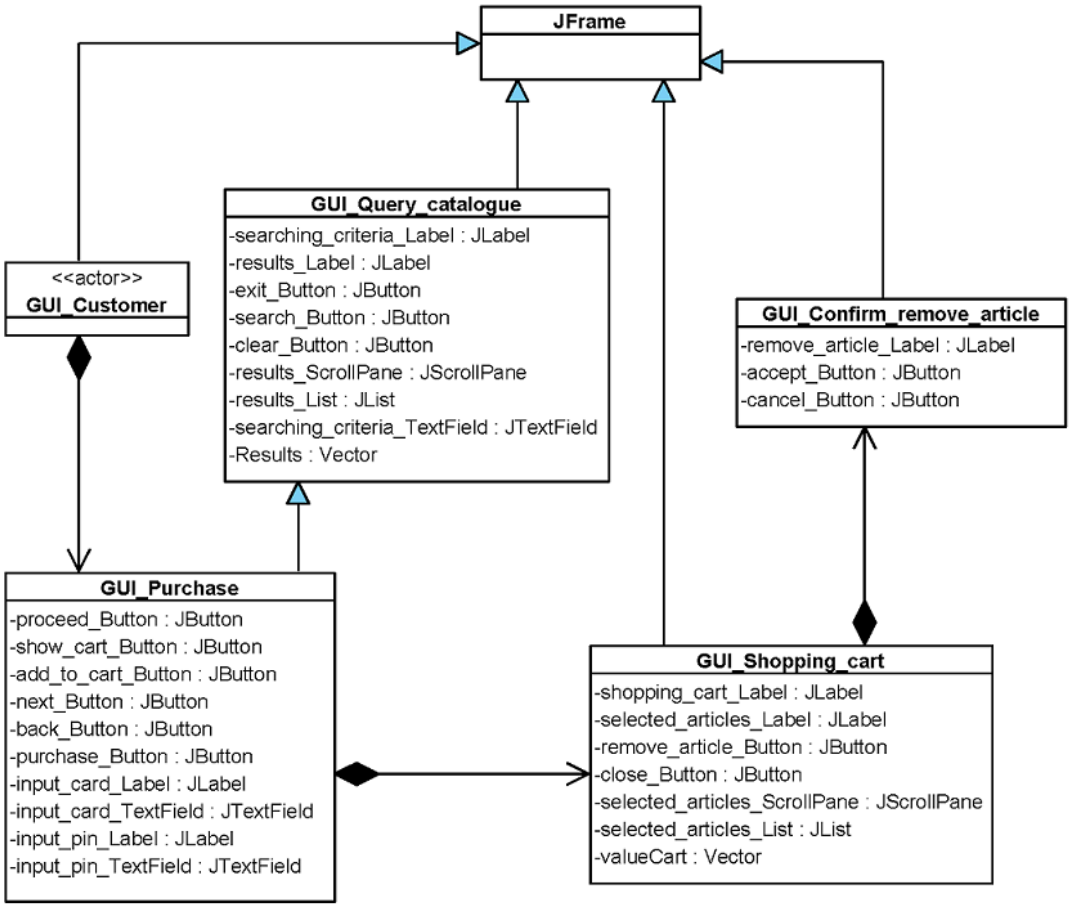
PASO 4: Construcción de Diagramas de Clases de Interfaces

Contienen las clases para la interfaz de usuario. Cada **caso de uso del diagrama de interfaz de usuario** es representado por medio de **ventanas.**

Al haberse descrito en el paso anterior cada caso de uso por medio de un diagrama de interacción de usuario, podemos identificar los **componentes de interfaz de usu**ario para cada ventana. => **El diagrama de interfaz** de usuario incluye clases para cada interfaz de usuario conectado por medio de asociaciones a la clase ventana.

Ejemplo:

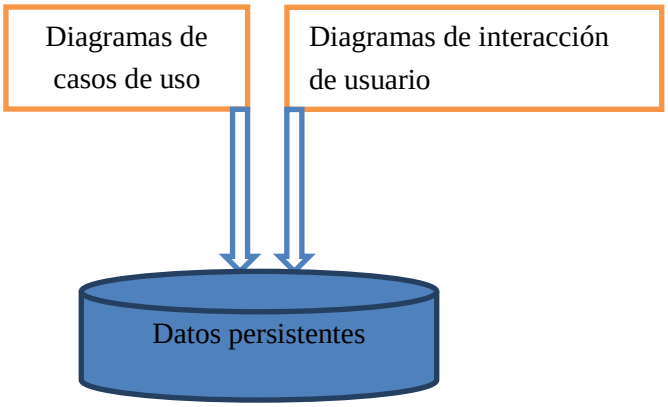
De cada caso de uso del diagrama anterior diseñaremos una clase para cada ventana perfectamente relacionada con sus asociaciones y todas ellas son una especialización (herencia) de JFrame. Incluye todos los componentes de la interfaz de usuario.



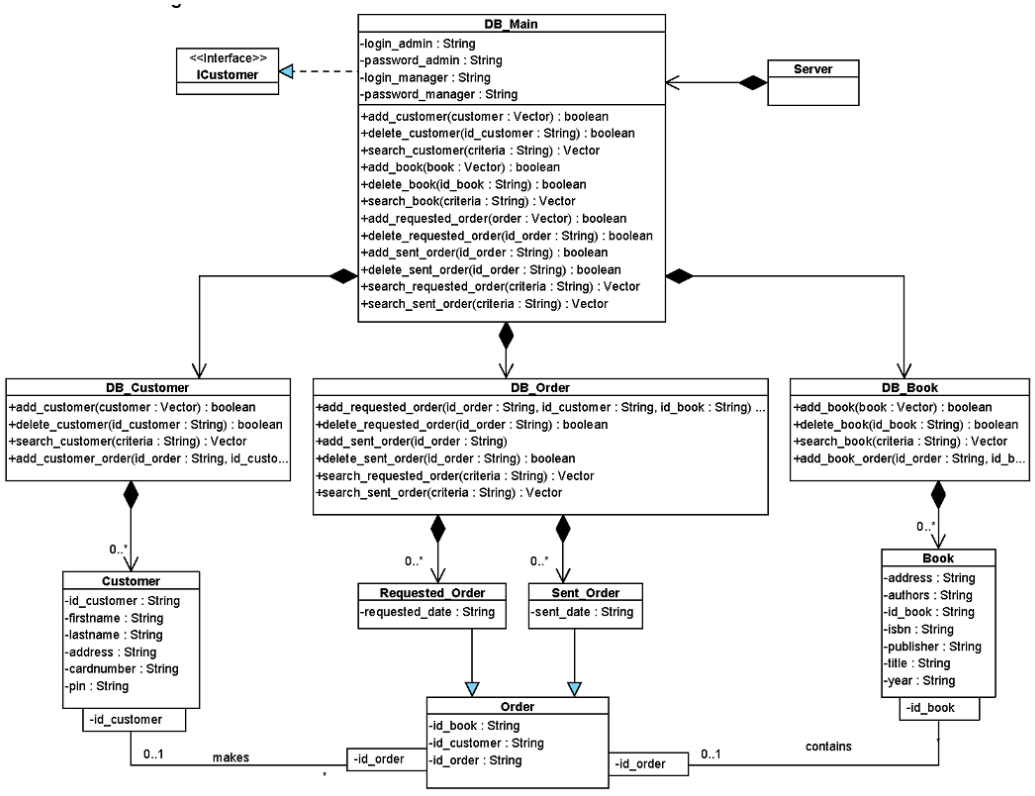
PASO 5: Construcción de Diagramas de Clases de Base de Datos

El quinto paso consiste en la creación de un diagrama de clases para los componentes de la base de datos. A partir de los diagramas de casos de uso y de interacción de usuario se pueden identificar qué tipos de *datos persistentes* [6]deben ser almacenados en la aplicación.

Una de las aportaciones de este estudio ha sido la descripción por medio de diagramas de secuencias de la base de datos no solo interacciones con la interfaz de usuario sino también con la base de datos con el fin de proveer y actualizar los datos enviados o solicitados.

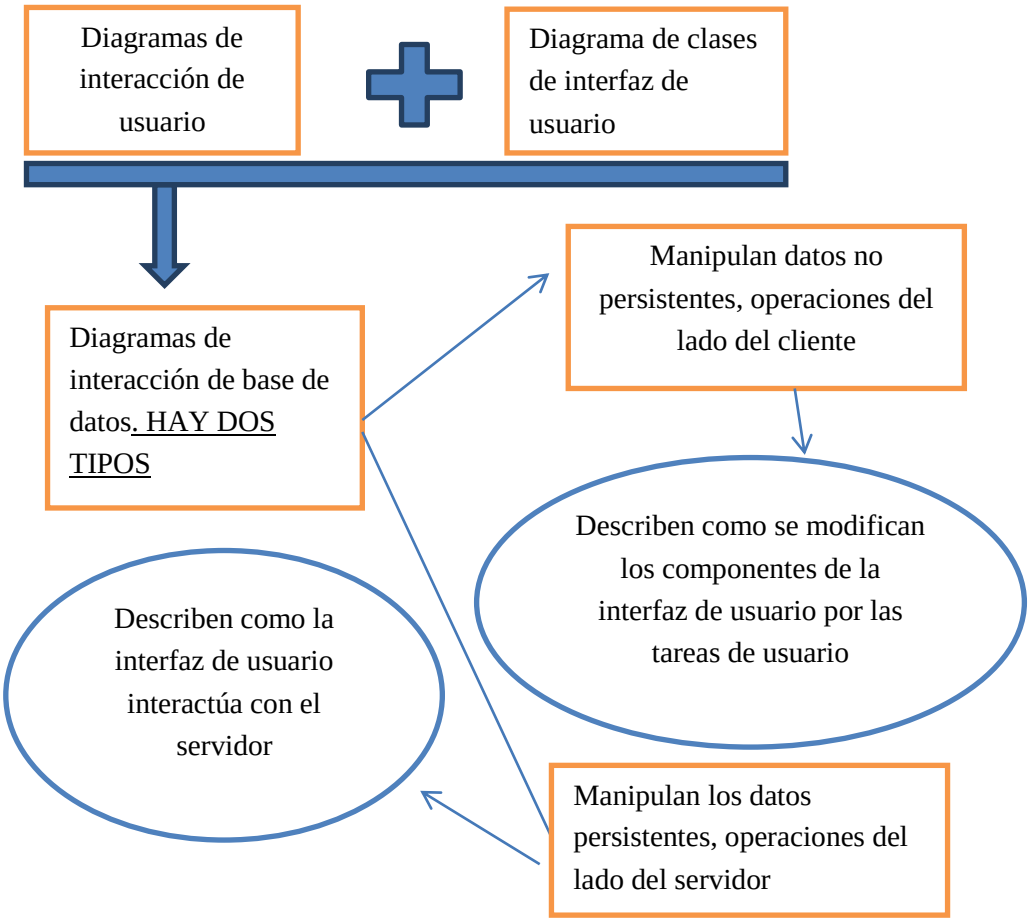


Ejemplo: Para nuestro ejemplo la clase principal está compuesta de tres tablas, clientes pedidos y libros cada una ofrece un conjunto de métodos que serán identificados más tarde cuando se modele el último diagrama.



PASO 6: Construcción de Diagramas de Interacción de Bases de Datos

Desde el diagrama de casos de uso y los diagramas de interacción del usuario se pueden identificar los datos persistentes que serán almacenados por la aplicación. Serán del lado del servidor y mantenidos por la consulta o actualización de la interfaz de usuario. Se puede adoptar un diseño en el que existe una clase principal en la base de datos que es un contenedor de clases tipo tabla. Se asegura la distribución y normalización, pero se fuerza a que las operaciones sean más complejas. La clase principal es la encargada de los accesos y de la creación y destrucción de otras tablas.



El último paso es la construcción de un conjunto de diagramas de secuencias especiales para la descripción de la interacción de las interfaces de usuario con la base de datos a partir de los diagramas anteriores.

Se distinguen dos clases:

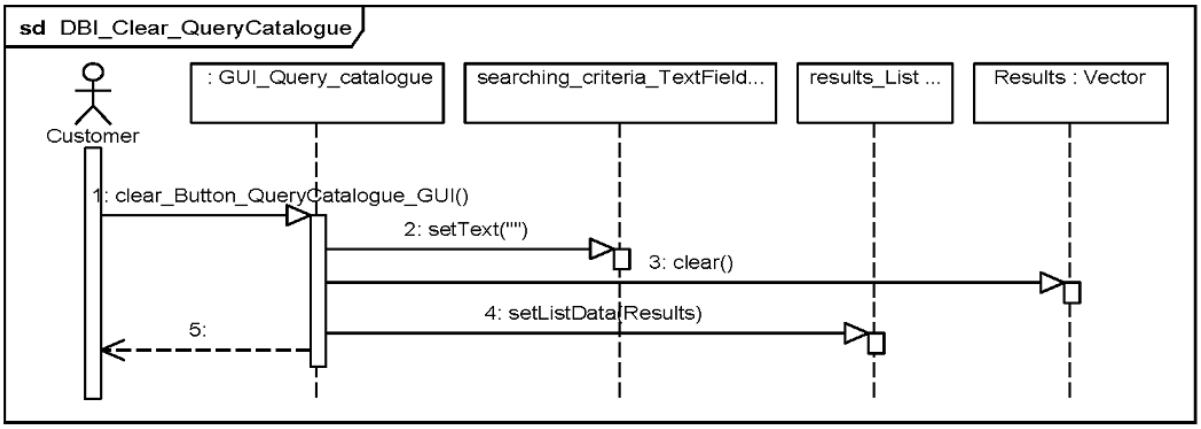
- Los que mantienen datos no persistentes en el lado cliente.
- Los que mantienen datos persistentes en el lado servidor.

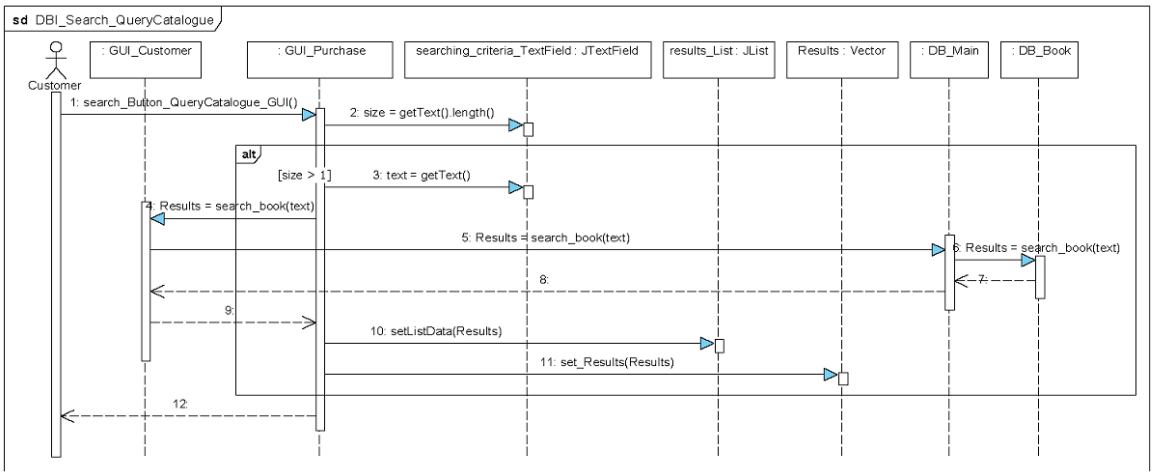
En la primera clase los diagramas describen los casos de uso en los cuales el usuario añade o borra elementos de los contenedores de la interfaz de usuario, esos diagramas fueron definidos en el primer paso, identificados en el segundo y descritos en el tercero.
La segunda clase describen como los casos de uso interactúan con el servidor y consulta o actualiza los datos persistentes en la BD.

En este caso las transiciones son asociadas a componentes del tipo JList, JTextField o JButton en la mayoría de los casos.
Para el ejemplo vamos a distinguir tres tipos de interacciones:

- Contenedores de datos no persistentes
- Contenedores de solo datos persistentes
- Contenedores de ambos tipos de datos, persistentes y no persistentes.

Como último punto, se puede resaltar la generación de código a partir de este método de modelado, en el que a partir del diagrama de clases de interfaces se generan plantillas de código para las ventanas, y a partir del diagrama de clases de la base de datos se genera código de la base de datos (gracias a ORM de Visual Paradigm).





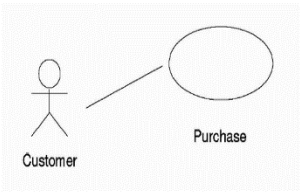
SOPORTE DE HERRAMIENTAS

Para llevar a cabo esta técnica de diseño en aplicaciones móviles, las herramientas que utilicemos deben cumplir ciertos requisitos:

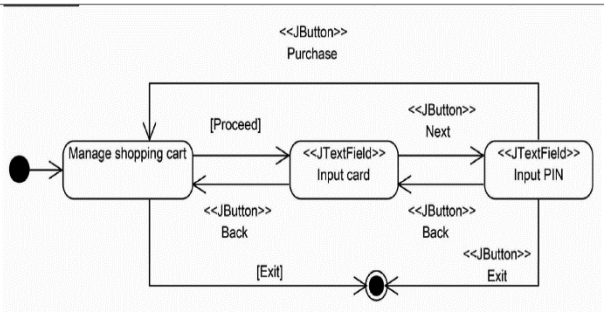
- La herramienta debe permitir modelar diagramas de casos de uso, estado y secuencia
- Debe proporcionar una biblioteca de interfaz de usuario, componentes y estereotipos UML para cada componente de la interfaz de usuario para ser utilizados en los diagramas de estado y de secuencia.

Tales componentes de interfaces de usuario deben de estar clasificados según los siguientes criterios:

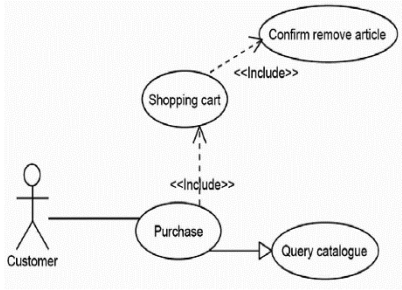
- **De Entrada, de salida o ambas**
- **Por similitud en comportamiento**
- **Especificación de componentes de interfaz de usuario**
- **Convenciones de nombrado:** Se deberá de generar un diccionario con los nombres de todos los elementos de los diagramas.
- **Sincronización de modelo:** Ayuda al analista a acciones como nombrar los diagramas de interacción de usuario, asociar ventanas a un diagrama de interacción de usuario, incluso podría generar automáticamente un diagrama de UI



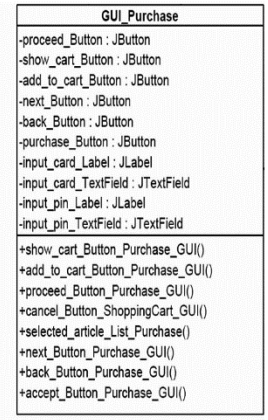
Step 1. Building a use case diagram



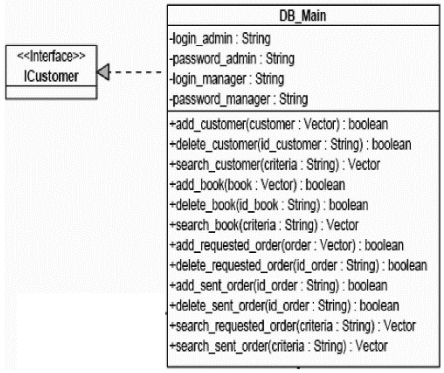
Step 2. Building the user interaction diagram



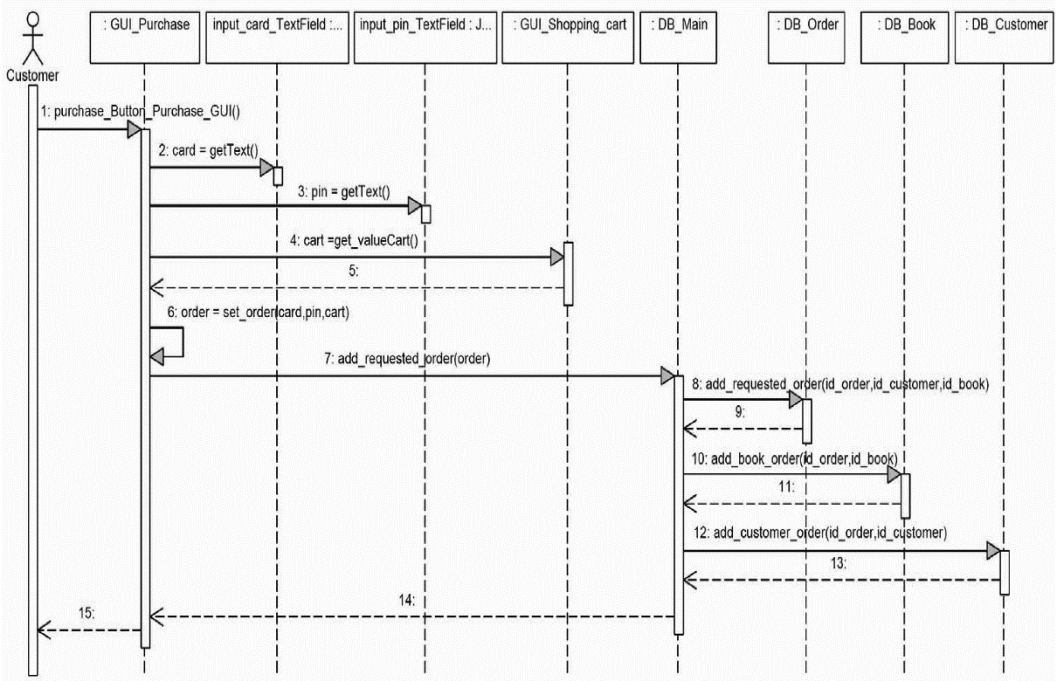
Step 3. Building the user interface diagram



Step 4. Building the user interface class

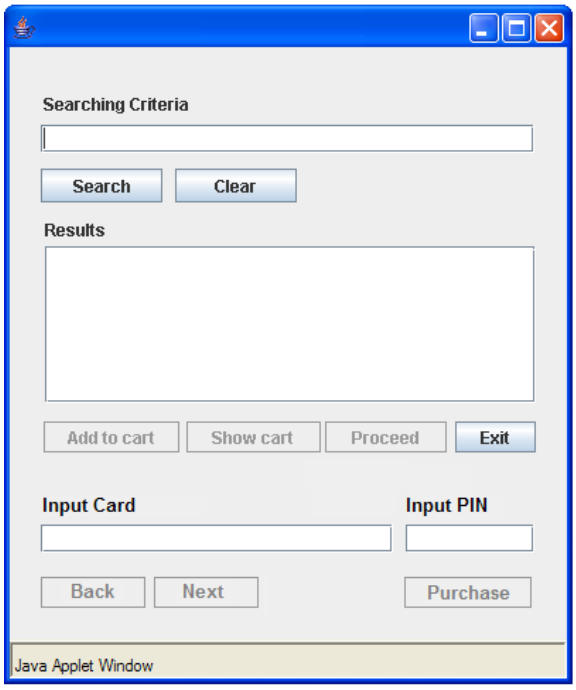


Step 5. Building the database class

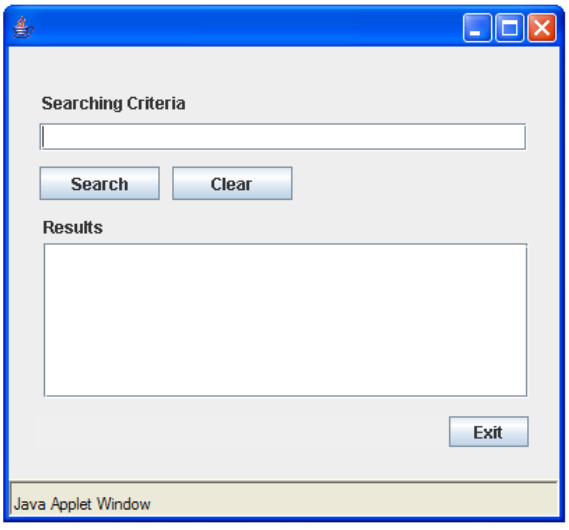


Step 6. Building the database-interaction diagram

The Purchase window



The Query Catalogue window



The Shopping Cart window

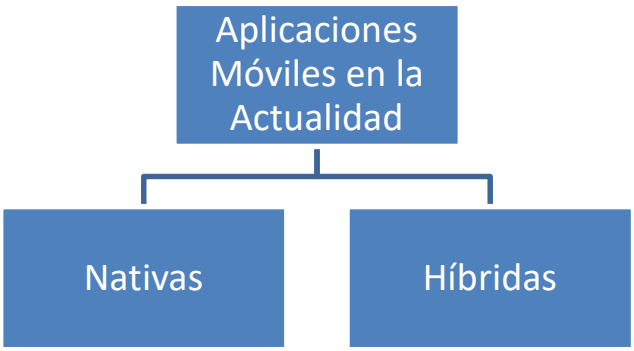
The Confirm Remove Article window

3 Apps Nativas VS Híbridas

Una de las encrucijadas que debe se debe afrontar en el desarrollo de aplicaciones móviles es si se debe enfocar hacia la utilización de plataformas híbridas o nativas. No es una pregunta fácil de contestar ya que depende de una gran cantidad de factores externos que hay que tener en cuenta.

Quizás el factor más importante a tener en cuenta es la audiencia a la que va destinada la aplicación, los usuarios puede que accedan únicamente a través de una red corporativa, mientras que en otros casos la aplicación está mucho más orientada al cualquier tipo de usuario. En muchas ocasiones el punto de partida determina qué camino se debe tomar a la hora de desarrollar una aplicación, siempre teniendo en cuenta sus ventajas e inconvenientes, que describiremos detalladamente en este apartado.

En la actualidad existen dos tipos de aplicaciones móviles, las nativas y las híbridas.



- Las aplicaciones nativas, específicas para una plataforma móvil concreta (*IOS*[7] o Android) y se desarrollan usando las herramientas creadas por los fabricantes Xcode y Android Studio respectivamente y generalmente tienen mejor rendimiento que las híbridas.
- Las aplicaciones híbridas usan tecnologías estándar como *HTML5*[8], *JavaScript*[9] y *CSS*[10] para crear aplicaciones multiplataforma que funcionan en múltiples dispositivos. Además, incorporan un tanto por ciento de código nativo para subsanar las limitaciones de la parte web, como pueden ser por un lado el acceso al hardware específico de los dispositivos, cámara, GPS, acelerómetros y por otro lado acceso a gestión de sesiones, almacenamiento offline de bases de datos, etc. En términos generales las apps híbridas se escriben más rápido que las nativas.

Seguidamente haremos un estudio comparativo comenzando por describir por separado cada uno de los tipos aplicaciones y posteriormente haremos una comparativa a groso modo entre ellas, valorando las ventajas y desventajas de cada una de las opciones de desarrollo.

3.1 Aplicaciones Nativas

Las aplicaciones nativas proporcionan la mejor usabilidad posible. Existen ciertas cosas que sólo se pueden realizar usando código nativo:

- Gestos multitáctiles
- Gráficos rápidos y eficientes
- Animaciones fluidas
- Acceso al hardware nativo

Se debe a que el código se compila directamente a código máquina, mientras que las híbridas añaden una transformación más por el camino. Sin embargo, cada plataforma con la que estemos trabajando debe desarrollarse por separado ya que, aunque la lógica de la aplicación es la misma para las diversas plataformas, el lenguaje, APIS[11] y proceso de desarrollo es totalmente diferente, aunque el paradigma sea el mismo.

Normalmente el proceso de desarrollo es largo y complejo para proyectos medianos o largos ya que requiere un mayor nivel por parte del desarrollador. Las aplicaciones nativas necesitan de herramientas como IDEs específicos para la plataforma, gestión de proyecto, control de versiones y herramientas de *profiling*[12] y *testing*[13]. Estas herramientas son necesarias porque el proceso de desarrollo es complejo. La ventaja que aporta es que el proceso está muy bien documentado.
Existen más de 2.500 libros de desarrollo para iOS y Android e innumerables recursos de programación online.

Además, los usuarios están acostumbrados a la interfaz de su plataforma, con lo que la aplicación les resulta inmediatamente familiar, mientras que en las híbridas hay que implementar una interfaz propia que a los usuarios les resultará desconocido de partida.

Otra ventaja muy evidente es que existe una gran comunidad alrededor de estas plataformas. Cualquier duda es rápidamente resuelta mediante la búsqueda en sitios especializados como stackoverflow.com o en los propios foros de discusión de los fabricantes, mientras que cuando nos enfrentamos al desarrollo de apps híbridas es más difícil encontrar ayuda, ya que la cantidad de programadores especializados en desarrollo híbrido es mucho menor.

3.2 Aplicaciones Híbridas

Las aplicaciones híbridas son aplicaciones web en un navegador nativo, como una UIWebView en iOS o una WebView en Android (no Safari o Chrome). Las apps híbridas se desarrollan usando HTML, CSS y Javascript, y después se embeben en una aplicación nativa, a veces usando únicamente código nativo, o a veces usandoplataformas como *Cordova*[14].
El desarrollo de apps híbridas es siempre más rápido y simple, y el acceso al hardware se realiza mediante código nativo o mediante plugins (si el desarrollo se ha realizado por ejemplo en Cordova) o incluso mediante el uso de extensiones o *frameworks*[15] creados por los propios desarrolladores de las librerías.

El principal problema de las apps híbridas es que dependen del navegador integrado para funcionar, por lo que su rendimiento siempre es menor que en las nativas. Por otro lado, el acceso a datos se realiza mediante inyección de código en el navegador y mediante la ejecución de Javascript, con lo que el mantenimiento se complica mucho con el paso del tiempo.

El problema de este tipo de depuración es que no se puede acceder directamente a las variables y objetos del código, sino que hay que extraer el código directamente de la webview y los compiladores no están preparados para ello. Más adelante se explica más en detalle esta problemática.

Sin embargo, es posible delegar partes importantes de la aplicación al código nativo, de forma que el desarrollador puede llegar a soluciones de compromiso, e introducir partes del código de forma nativa implementando, por ejemplo, módulos para el acceso a la cámara, almacenamiento de datos local etc, creando aplicaciones con un porcentaje nativo (30% nativo 70% web).

El desarrollo de una app híbrida es más barato inicialmente, aunque más caro a largo plazo. Por ello muchas startups desarrollan su app híbrida al principio y cuando consiguen financiación desarrollan una nueva versión nativa. Algunos ejemplos pueden ser apps como Preguntados o Wallapop
Algunos desarrolladores deciden emplear tecnologías más concretas o especializadas, optando por plataformas específicas que detallaremos más adelante, como pueden ser REACT, NATIVESCRIPT, Intel® XDK, XAMARIN, PHONEGAP (aka TITANIUM STUDIO), CORDOVA por sus características especiales, porque se adapta mejor al tipo de producto que van a desarrollar y existen extensiones creadas por la comunidad que les facilita el trabajo, o simplemente porque son más expertos en dichas plataformas.

Además, hay que destacar que es muy difícil depurar una app híbrida cuando se producen errores

- Muchas veces es complicado determinar si el error se encuentra en la parte nativa o en la parte web ya que comúnmente se mezclan ambos paradigmas por comodidad.
- No hay herramientas tan potentes para trazar requests(demandas) y responses(respuestas) de las peticiones que se realizan
- Es muy complicado depurar el código Javascript cuando se ejecuta en un WebView

La principal problemática es que cuando se trabaja con una app nativa el código fuente está bien distribuido en managers, singletons y clases, siendo fácil hacer un seguimiento de los problemas e incidencias que nos encontramos en un código simple y manejable.
Cuando trabajamos con una aplicación híbrida al final todo el código a ejecutar y/o analizar pasa por el mismo punto, por una única función que es llamada cuando la carga de la parte web finaliza. Eso crea una función enorme, ya que la misma contiene todos los casos de uso de la aplicación con lo que debes estar parando constantemente en esa función para ir depurando la funcionalidad.
Por supuesto, no existe acceso a los objetos, métodos o variables, por lo que cualquier error ha de ser localizado por intuición, analizando el html y/o el javascript.
Por otro lado, las plataformas nativas ofrecen herramientas muy avanzadas de depuración que permiten optimizar la aplicación en sus aspectos fundamentales, como uso de CPU, memoria, red y disco.

Existen 3 formas principales de comunicar una app con la parte web:

- **Local:** Todos los datos se almacenan localmente porque no se requiere actualizar

- **Servidor:** Los datos están almacenados en un servidor porque se actualizan frecuentemente o requieren de servicios adicionales para funcionar

- **Mixta:** Los datos se obtienen de un servidor, pero se almacenan localmente para cuando no exista conexión a Internet

3.3 Comparativas

Tabla rápida de comparación app nativa vs híbrida

	Nativa	Híbrida
Lenguaje de Desarrollo	Nativo	Nativo y Web
Acceso al dispositivo	Completo	Parcial
Características Exclusivas	Todas	Moderadas
Velocidad	Muy rápida	Media
Tiendas	Disponible	Disponible
Requieren Aprobación	Obligatoria	Obligatoria
Portabilidad	Ninguna	Alta
Gráficos Avanzados	Alto	Medio
UI/UX (Experiencia de Usuario)	Alto	Medio
Acceso a APIS Nativas	Alto	Medio
Costo de Desarrollo	Alto	Medio

Tabla detallada de comparación app nativa vs híbrida

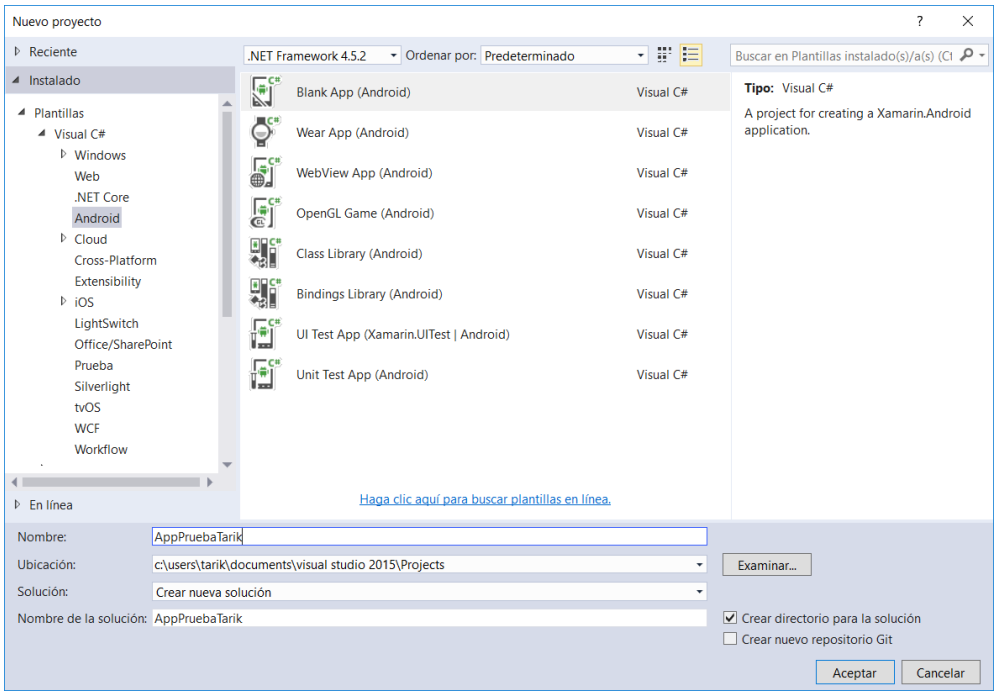
Criterio	Nativa	Híbrida	Comentarios
Soporte Multiplataforma	✵	✵✵✵✵✵	Las apps híbridas pueden ser portadas a múltiples plataformas. Sin embargo, para cada plataforma particular debe desarrollarse un contenedor nativo. El soporte de dispositivos antiguos es complicado porque los navegadores no soportan todas las funciones. La ventaja al desarrollar apps híbridas es muy evidente: un único código es válido para todas las plataformas mientras que en las aplicaciones nativas es necesario escribir el código cada vez
Tiempo de Respuesta	✵✵✵✵✵	✵✵✵✵	Las aplicaciones nativas son siempre más rápidas.
Experiencia de Usuario	✵✵✵✵✵	✵✵✵	Las apps nativas son mucho mejores en términos de usabilidad. Además es posible acceder al hardware nativo, GPS, cámara, lista de contactos, etc. Las apps híbridas deben hacer un sobreesfuerzo para simular el comportamiento de las apps nativas, a las que están acostumbrados los usuarios. Por ello deben implementar librerías como jQuery Mobile, Swipe, iScroll,Zynga, etc
Coste de Desarrollo	✵✵✵	✵✵	Es más barato desarrollar una app híbrida ya que los desarrolladores son menos especializados y los tiempos se acortan al usar tecnologías más conocidas y tener la posibilidad de usar capas de presentación de terceros.
Coste del Soporte	✵✵	✵✵✵✵	Mantener una app nativa es más sencillo ya que cuenta con herramientas de depuración avanzadas, mientras que en las híbridas debe analizarse el HTML y las funciones javaScript y realizar inyección de datos y código.
Reusabilidad del Código	✵✵	✵✵✵✵	El código escrito para las apps nativas vale exclusivamente para ellas, con lo que hay que reescribir partes comunes en diferentes lenguajes. Una app híbrida tiene un único código fuente para todas las plataformas
Tests Funcionales	✵✵✵	✵✵✵✵	Las apps nativas deben ser probadas y certificadas para cada plataforma, sin embargo las híbridas funcionan sin problemas en todos los dispositivos en la parte de UI/UX
Soporte Offline	✵✵✵✵	✵✵✵	Las apps nativas son mejores

			siempre que no se requiere conectividad a Internet. El cacheo in-browser en HTML5 ya está disponible aunque su funcionalidad es por ahora muy limitada.
Costo Time2Market	֍֍֍	֍֍֍֍֍	Desplegar una app nativa en múltiples plataformas es complicado por los tests y certificaciones necesarias. Las aplicaciones híbridas ganan en este terreno por goleada ya que se realiza un único despliegue inicial y luego el mantenimiento se facilita mucho ya que el contenido se actualiza vía servidor.
Leverage Web Side	֍֍	֍֍֍֍	Las apps híbridas se benefician de infinidad de recursos y componentes de terceros que facilitan mucho el trabajo de desarrollo.
Pool de Recursos	֍֍	֍֍֍֍	Hay muchos más desarrolladores de back que de front, con lo que existen muchos más recursos aplicables a aplicaciones híbridas que nativas.
Estadísticas de Uso	֍֍֍	֍֍֍֍֍	Mientras que en las aplicaciones nativas es complicado recolectar datos de usos y se necesita utilizar librerías y almacenamiento de terceros mientras que en híbridas la carga recae en back y es mucho más automático
Actualización Rápida	֍	֍֍֍֍֍	En las aplicaciones nativas es muy lento desplegar nuevas versiones o hacer rollout mientras que en las híbridas es inmediato ya que al abrir la app los usuarios tienen inmediatamente los cambios

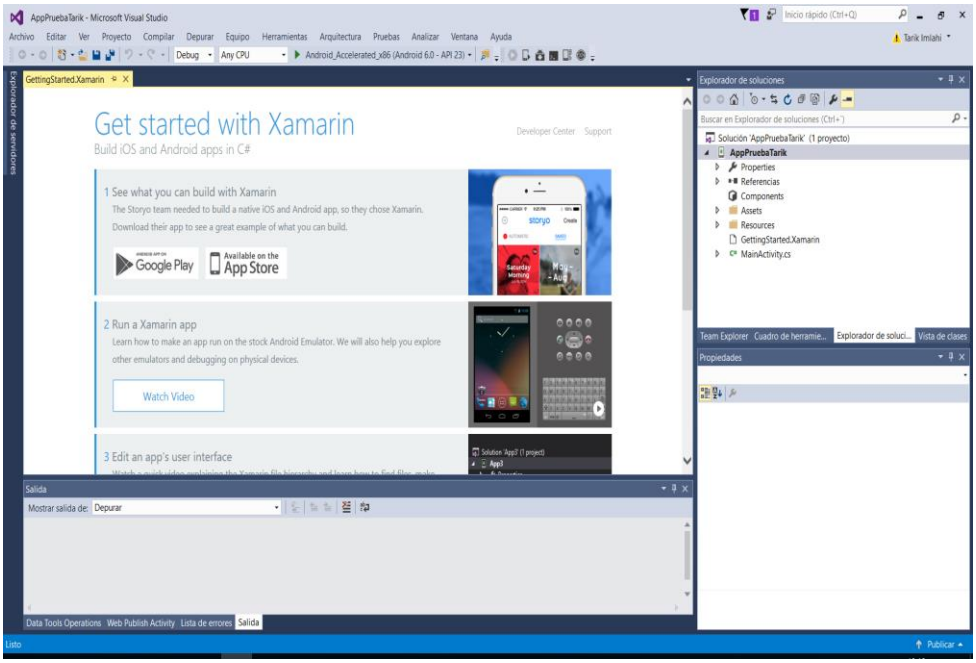
4 Visual Studio y Xamarin. Guía Rápida.

En este apartado explicaremos el funcionamiento de la herramienta de Xamarin para el desarrollo de aplicaciones en Android en Visual Studio. La razón de que hagamos este pequeño tutorial de cómo hacer una pequeña aplicación es para que posteriormente se pueda entender las ventajas y limitaciones a la hora de poder usar la metodología de trabajo a nivel de modelado.

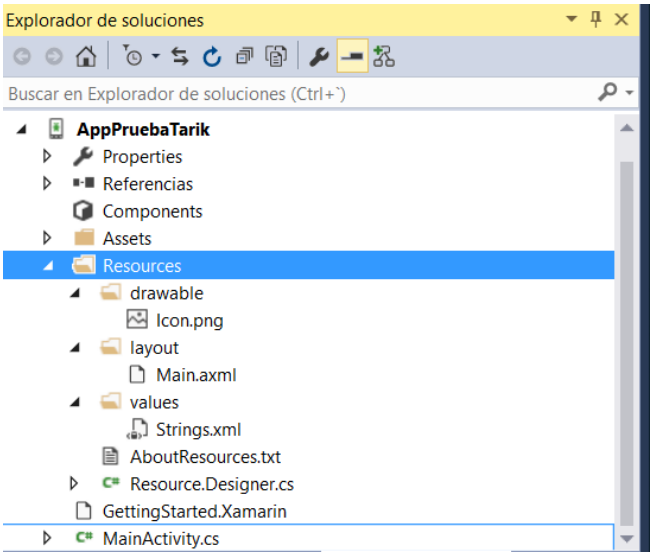
Para empezar, en Visual Studio debemos crearnos un Nuevo Proyecto Android. Como vemos en la siguiente imagen tenemos por defecto seleccionada la opción de "Crear Directorio Para la Solución". Si quisiéramos crear también un repositorio Git también podríamos hacerlo marcando la opción correspondiente. Si asignamos primero el nombre al proyecto, por defecto se rellena automáticamente el mismo a la solución, si lo hacemos a la inversa podemos llamar a nuestra solución de manera diferente. Pulsamos Aceptar.



Seguidamente ya estamos dentro de nuestro entorno que nos abre automáticamente el archivo "GettingStarted.Xamarin" que contiene enlaces a su página dónde muestran ejemplos con los que poder empezar a desarrollar en Xamarin.



Las ventanas que más vamos a usar son las del "**Explorador de Soluciones**", "**Cuadro de Herramientas**" y "**Propiedades**". Dentro del explorador de Soluciones visualizamos el contenido de Nuestra Solución, en este caso vemos nuestro proyecto de Android y dentro de él tenemos varias carpetas y desplegables. Los que nos deben interesar ahora mismo es la carpeta "**Resources**".



Dentro de Resources vemos tres carpetas:

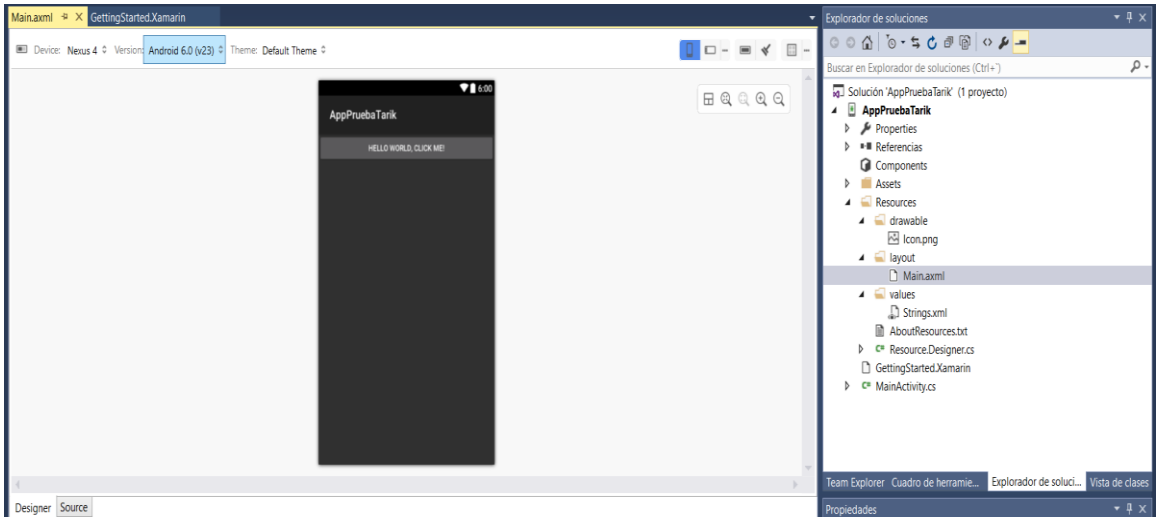
Drawable: Aquí estarán albergadas las imágenes que utilizaremos tanto para los fondos de pantalla como para el icono representativo de la aplicación. El icono que veríamos como acceso directo a la aplicación.

Layout: Esta carpeta es la más importante ya que alberga las Vistas que tendrá nuestra aplicación. Archivos con extensión ".axml "

Values: Dentro está el archivo String.xml que contiene etiquetas que contienen los nombres de ciertos componentes.

4.1 Vistas y Activities:

El nuevo proyecto viene por defecto con una **Vista o Layout** y una **Activity** de Prueba. La vista albergada en la carpeta Layout se llama **Main.axml** , y es la correspondiente vista en este caso de la Activity **MainActivity.axml.**

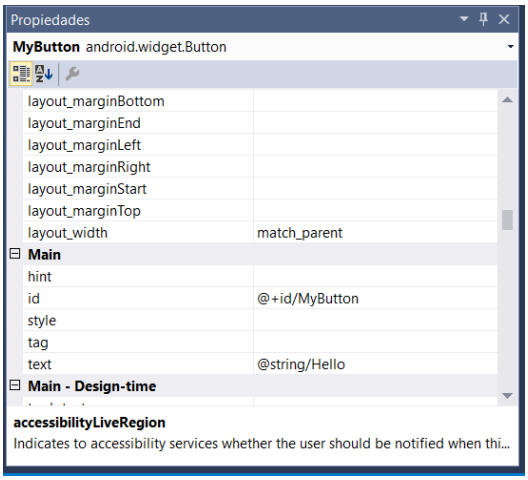


Digamos que al seguir el patrón de diseño Model View – View Model (MVVM) la vista no contiene más que los componentes descritos a través de lenguaje xml tal que así:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/MyButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/Hello" />
</LinearLayout>
```

Tenemos una vista liviana a nivel de código ya que a ninguno de los componentes se les asigna una funcionalidad, tan solo se describen, se les añade un identificador, un tamaño, color, márgenes respecto a otros componentes, posiciones…etc.

El contenido de la Vista se puede describir usando código (a veces muy necesario), o la herramienta de Propiedades en forma de Formulario, muy usado al principio, pero a veces más lioso cuando se quiere modelar la vista con más detalle. La vista la podemos observar desde esos dos puntos de vista, a nivel de Código (Source) o a nivel de Diseñador (Designer) como hemos podido observar en las anteriores imágenes.



En la vista de Diseñador, si por ejemplo cliqueamos sobre el botón que tenemos en el Layout podemos ver en la ventana de Propiedades todos los atributos de ese botón en particular. Ahora mismo nos vamos a fijar más en los parámetros que van a servirnos para definir el funcionamiento de nuestra aplicación y no en aquellos tipos posición, color, tipo de letra, estilo, etc.

En el aparatado Main de esta ventana, si lo desplegamos vemos dos atributos que debemos usar obligatoriamente. El primero es el "id" y el segundo es "text". Con el id asignamos un identificador que debe ser único a nuestro botón ya que puede causarnos problemas el asignar el mismo nombre a distintos componentes ya que digamos que todos se guardan en una misma localización indiferentemente del Layout en que estemos trabajando. El formato del atributo id es @+id/<Nombre del botón> y el de text es @string/<Nombre>.

En este caso el identificador del botón es @+id/MyButton y su texto @string/Hello. El identificador del botón sirve para poder ser capturado desde nuestra activity **MainActivity** y el identificador del texto sirve para asignar un texto base a ese botón. El nombre que va a recibir queda referenciado en el archivo **String.xml** de la carpeta **Values** . Veamos el contenido de este archivo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="Hello">Hello World, Click Me!</string>
    <string name="ApplicationName">AppPruebaTarik</string>
</resources>
```

Vemos que efectivamente, el nombre base con el que se asigna a ese identificador es el que nos aparece dentro del botón en la primera pantalla al ejecutar la aplicación. Al igual que el nombre de la aplicación que también viene referenciado aquí. No nos hemos parado a indagar el porqué de dicho proceso de asignación de una variable que referencie a un nombre para solo definir un texto, pero su razón tendrá puesto que es uno de los pasos que nos instan a seguir los propios desarrolladores de Xamarin. En definitiva, este paso solo lo hemos hecho para los botones de la aplicación por lo que nos ha quitado un tiempo significativo.

Lo realmente significativo para nosotros y base para empezar a entender cómo se trabaja con vistas y activities se explica en el siguiente apartado.

4.2 Captura de componentes de la Vista desde la Activity

Este es el código de nuestra **MainActivity:**

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;

namespace AppPruebaTarik
{
    [Activity(Label = "AppPruebaTarik", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        int count = 1;

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.Main);

            // Get our button from the layout resource,
            // and attach an event to it
            Button button = FindViewById<Button>(Resource.Id.MyButton);

            button.Click += delegate { button.Text = string.Format("{0} clicks!", count++); };
        }
    }
}
```

Aquí relacionamos la **MainActivity** con el **Layout Main** :

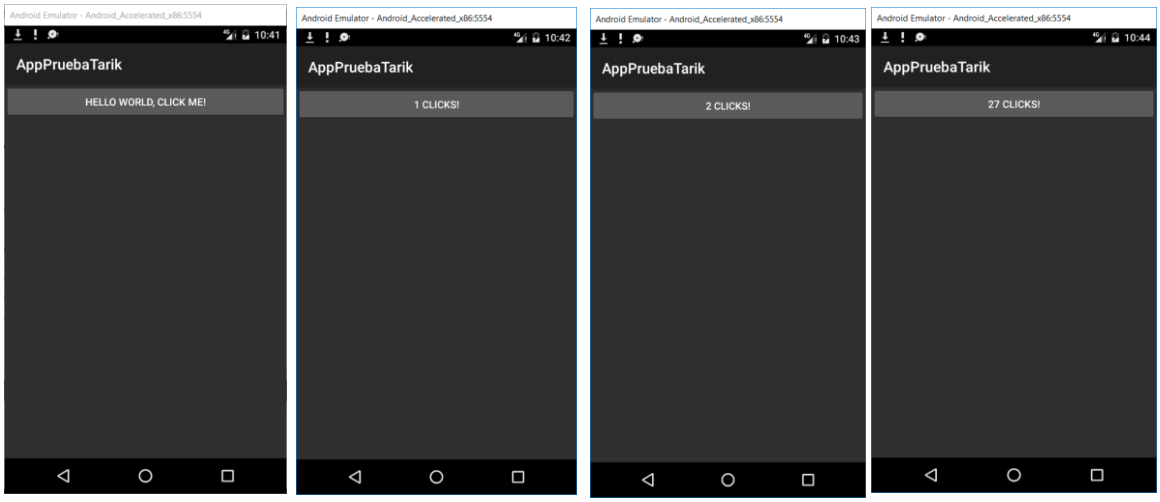
```
SetContentView(Resource.Layout.Main);
```

Y capturamos el botón de esta otra:

```
Button button = FindViewById<Button>(Resource.Id.MyButton);
```

Creamos una instancia de una variable botón y le asignamos su correspondiente componente Button buscándolo a través del id correspondiente. Una vez capturado nuestro botón, como vemos ya podemos asignarle funcionalidades a sus eventos. Como por ejemplo al click sobre él. Vemos que se define en código un método delegado que hace que se asigne un texto a modo de contador que incrementa la variable global en una unidad por cada Click. El resultado se muestra en la siguiente página y vemos que la aplicación funciona.

La funcionalidad de la aplicación es sencilla; la vista dispone de un botón cuyo texto inicial es "HELLO WORLD, CLICK ME!" que al pulsarlo simula un contador de Clicks que se incrementa una unidad en cada una de las sucesivas pulsaciones.



Hasta aquí hemos aprendido a :

- Asignar un identificador a un componente de la vista
- Asignar un Layout a una Activity
- Capturar un componente de la Vista desde una Activity

Ya conocemos unos conceptos muy básicos y técnicas muy necesarios que se van usar muchas veces a lo largo del desarrollo de cualquier aplicación para cualquier tipo de componente.

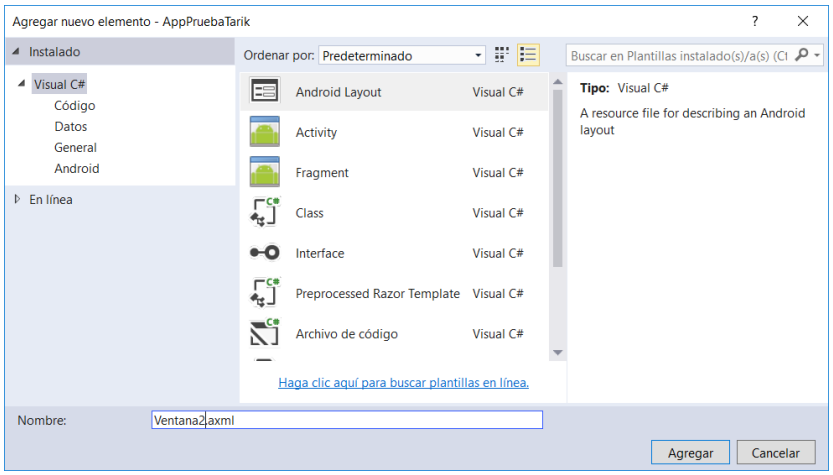
Lo puntos que se deben conocer también son los siguientes:

- Crear una nueva Vista
- Insertar nuevos componentes en la vista
- Crear una nueva Activity
- Comunicación entre Activities.

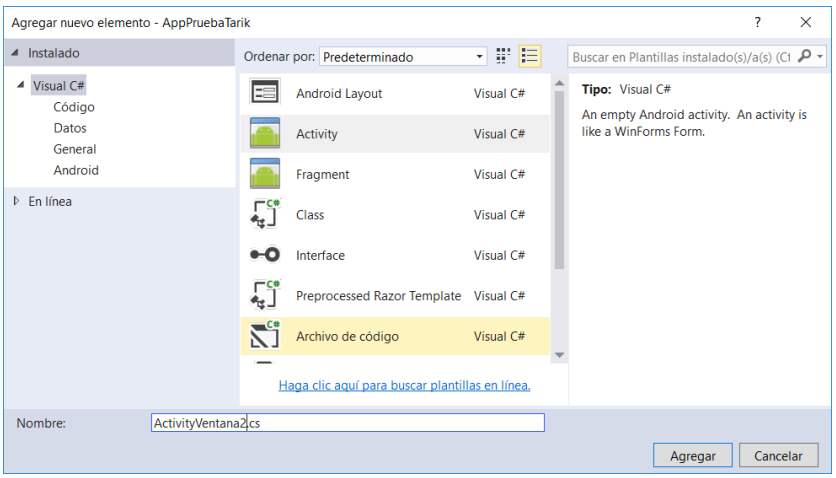
4.3 Comunicación entre Activities

Crearemos un nuevo Layout que vamos a llamar **Ventana2** que va a recibir la información que metamos en un campo de texto de la anterior **Main.** Como ya sabemos, la vista no es la que envía la información a la siguiente, ni la siguiente es la que la recibe. Los responsables de enviar y recibir son las Activities correspondientes. En este caso, la **MainActivity** enviará información a la **Ventana2Activity** y las vistas mostraran los resultados.

Paso1: Creamos un nuevo Layout con nombre **Ventana2** en la carpeta Resources/layouts



Paso2: Creamos una nueva clase Activity en el directorio raíz con nombre **ActivityVentana2**



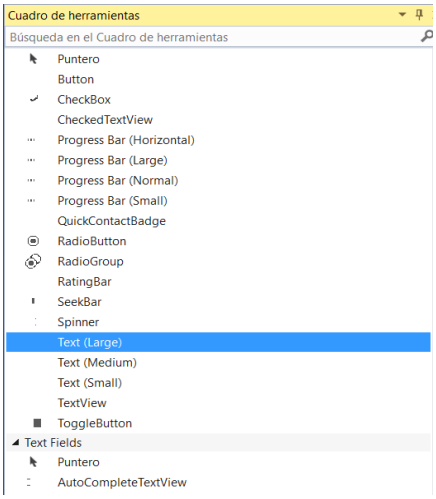
Paso3: Asignar a **ActivityVentana2** el Layout **Ventana2**

```
namespace AppPruebaTarik
{
    [Activity(Label = "ActivityVentana2")]
    public class ActivityVentana2 : Activity
    {
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);
            SetContentView(Resource.Layout.Ventana2);

        }
    }
}
```

Paso4: Crear un **TextView** para mostrar el mensaje que enviaremos.

Para ello en la vista de Diseñador del Layout podemos elegir del cuadro de Herramientas entre todos los componentes el que queramos y arrástralo a nuestra vista. En este caso elegiremos un texto grande a modo de Título cambiante. Asignaremos de identificador **TituloCambiante.**



Paso5: Ahora capturaremos ese TextView desde Ventana2Activity añadiendo el siguiente código dentro de nuestro Oncreated:

```
namespace AppPruebaTarik
{
    [Activity(Label = "ActivityVentana2")]
    public class ActivityVentana2 : Activity
    {
        TextView TituloCambiante;
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);
            SetContentView(Resource.Layout.Ventana2);

            TituloCambiante= FindViewById<TextView>(Resource.Id.TituloCambiante);
         }
    }}
```

Paso 6: Ahora que tenemos nuestra segunda ventana preparada vamos a modificar la MainActivity para enviar un mensaje a la otra Activity.

Lo que he hecho ha sido renombrar el texto del botón a través de String.xml. Y ahora vamos a cambiar su funcionalidad para que el mismo mensaje que se mostraba en el botón se muestre pero en el TextView de la siguiente Ventana2. Veremos así como enviar información de una Activity a otra.

```
   button.Click += delegate {

                string mensaje = string.Format("{0} clicks!", count++);

                var SiguienteActivity = new Intent(this,
typeof(ActivityVentana2));
                SiguienteActivity.PutExtra("Mensaje", mensaje);
                  StartActivity(SiguienteActivity);

            };
```

Como vemos en el código, instanciamos una Actividad a través de la clase **Intent** y posteriormente activamos esa actividad. Asignamos un extra en el que introducimos un nombre como primer parámetro a modo de variable a la que podrá acceder nuestra actividad de destino, y el contenido de esa variable como segundo parámetro.

Si quisiéramos pasar un objeto perteneciente a una clase, dicho objeto deberíamos serializarlo con **JSON** (Es muy importante instalar previamente el **paquete NuGet NewtonSoft.Json** en nuestro proyecto) y en el destino des-serializarlo de la siguiente manera, como por ejemplo:

El que envía:

```
var MySerializedObject = JsonConvert.SerializeObject(t);


            var inte = new Intent(this, typeof(MenuReceta));
            inte.PutExtra("MyDataReceta", MySerializedObject);
            StartActivity(inte);
```

El que recibe:

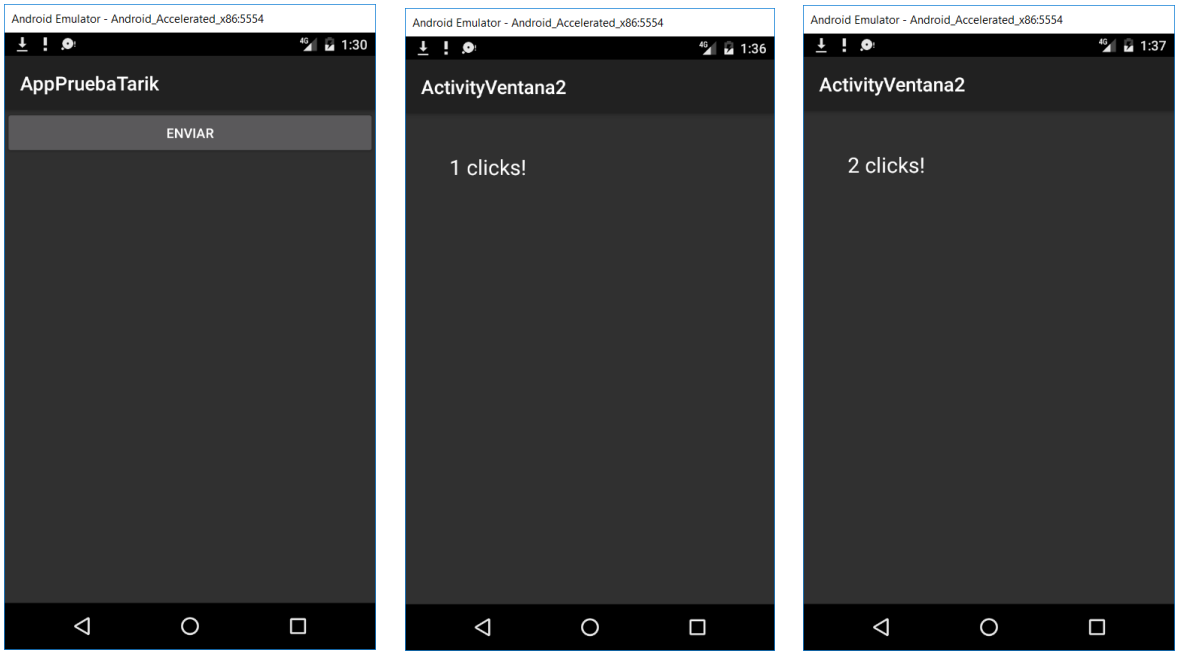
```
var MyJsonStringRecetaCoincidencia = Intent.GetStringExtra("MyDataReceta");
var MyObject = (Receta)JsonConvert.DeserializeObject(MyJsonStringRecetaCoincidencia,
typeof(Receta));
```

Paso7: Ahora debemos recibir el mensaje en el receptor y mostrarlo como título en nuestro TextView. Añadimos el siguiente código:

```
var mensaje = Intent.GetStringExtra("mensaje");

TituloCambiante.Text = mensaje;
```

Si probamos nuestra aplicación, cuando pulsamos el botón, nos visualiza el resultado en la otra ventana, si volvemos y lo hacemos repetidas veces vemos que nuestro mensaje se va incrementando ya que, aunque hayamos cambiado de vista, la anterior actividad sigue activa mientras nosotros no digamos lo contrario (Ciclos de Vida de las Actividades en Android, en este documento no se explicará al no ser fundamental para nuestro objetivo)



Hasta el momento, hemos explicado lo necesario para poder entender el enfoque que hemos usado para adaptar la metodología de Modelado del Software a esta plataforma. No nos detendremos a explicar más detalles acerca de Xamarin ya que toda la documentación se encuentra en la página oficial de Xamarin en forma de Guías y Tutoriales al igual que en foros y en FAQ elaborados por los desarrolladores.

4.4 Aspectos a mejorar en Xamarin

Desde nuestro punto de vista hay muchas particularidades de Xamarin que deberían mejorarse, sin embargo, a estas alturas, quizás los desarrolladores de la plataforma ya hayan solventado los problemas a los que nos hemos enfrentado

- La última versión de Xamarin tenía problemas a nivel de Diseñador, tardaba mucho en cargar las vistas, de hecho, a veces se quedaba indefinidamente cargando. Arreglamos este problema remontándonos dos versiones anteriores de Xamarin.
- El hecho de compartir una misma "bolsa" para albergar los identificadores de los componentes de las vistas es arriesgado. En nuestro caso, al ser relativamente pequeña mi aplicación no tuvimos ningún problema en asignar nombres diferentes a los identificadores, sin embargo, si la aplicación fuese más grande se podrían haber tenido problemas a la hora de evitar conflictos en ese nivel.
- Otro problema significativo es que a veces la aplicación no responde o aparecen errores de compilación donde no los hay. Repasando todo el código no se atisba ningún error y depurando saltan excepciones sin sentido. Ese es el momento de la regla de oro de Xamarin:
  La solución que suele funcionar es la siguiente: Copiar el contenido del proyecto en cuestión un proyecto nuevo; solamente las clases (Activities) y las carpetas que contienen recursos, renombrar el espacio de nombres de todas las clases, añadir los componentes y paquetes nuget extra al proyecto y compilar nuevamente.
- Respecto al anterior punto, cabe añadir que los problemas que se mencionan son realmente debidos a la depuración de la aplicación al usar los emuladores propios de Visual Studio. Sin meternos en profundidad y haciendo mención a comentarios de otros desarrolladores que han tenido contacto con otras herramientas nativas como Android Studio, ese problema y otros son comunes también en dichas plataformas. Emuladores muy lentos, salto de Excepciones sin sentido, etc. Finalmente, al usar nuestro propio dispositivo móvil, esos problemas desaparecieron haciendo que el desarrollo en el entorno mejorase significativamente

5 Especificación Preliminar. Aplicación Móvil Empty Fridge 1.0

Como se especificó en la documentación del anteproyecto, la aplicación Empty Fridge 1.0 (Nevera Vacía) que se pretende desarrollar principalmente ofrece un servicio útil a aquellas personas que por motivos de tiempo, de carencia de experiencia o de imaginación, o por cualquier otro motivo, no supieran qué cocinar en un momento dado con los ingredientes presentes en su nevera o despensa.

A modo de especificación preliminar del proyecto vamos a comentar todas las funcionalidades de la aplicación. Lo haremos de modo esquemático para tener una vista general más clara, aunque en la vida real, esta especificación preliminar podría dársenos en otro formato, tipo resumen de una entrevista, cuestionario, etc. Y nosotros posteriormente sacar nuestros esquemas o directamente con o sin herramientas extraer todos los casos de uso para comenzar a elaborar nuestro diagrama.

Nuestra aplicación deberá proveer de los siguientes servicios para los distintos roles:

USUARIO INVITADO:

- Entrar en la aplicación
- Registrarse
- Login (el usuario es invitado hasta que se identifica correctamente)
- Ver listado de Recetas
- Ver listado de Categorías
- Meter Ingredientes y Ver recetas acordes con estos ingredientes

USUARIO REGISTRADO:

- Acceso a las mismas opciones que el usuario Invitado
- Crear Recetas

USUARIO ADMINISTRADOR:

- Acceso a las mismas opciones que el usuario Registrado
- Panel de Administración: Gestionar Recetas, Gestionar Usuario, Gestionar Ingredientes.

A parte de estas funciones nuestra aplicación deberá mostrar información detallada a nivel nutricional de todas las recetas generada automáticamente en función de los ingredientes y su cantidad que contiene cada receta.

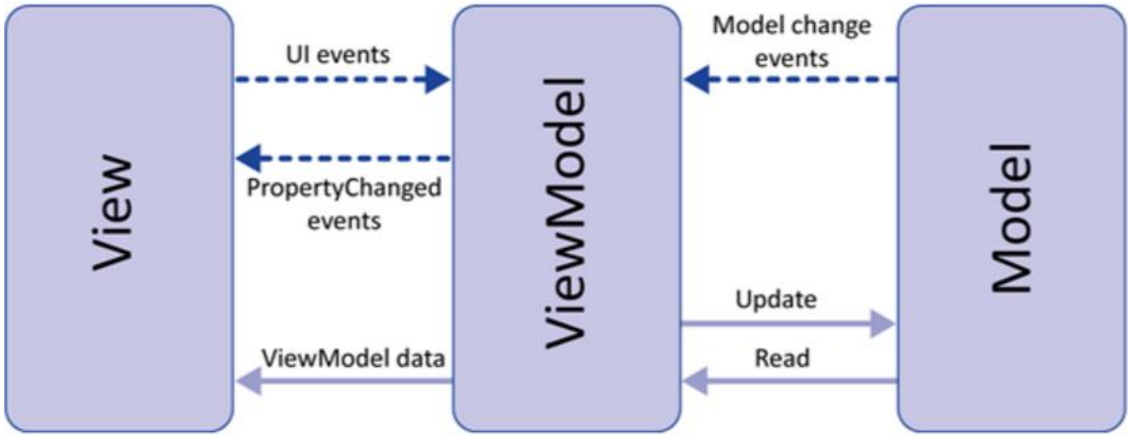
6 Patrones de Diseño: MVVM y Repository

La metodología de modelado que queremos aquí aplicar, se usaba en las prácticas de las asignaturas de MDS1 y MDS2 en aplicaciones que seguían un claro patrón de Diseño Modelo-Vista-Controlador (MVC) ya que usábamos vistas "conectadas" a una clase. Una misma vista era Vista y Controlador a la vez. En esta nueva plataforma vemos que el patrón hemos usado es un aproximado de Model-View-ViewModel (MVVM).

En cuanto el patrón que rige la conexión con datos externos, ya sea Base de Datos Locales o Servicios Web es el Repository ya que disponemos de una clase que engloba todos los métodos de acceso a nuestros datos, a los que podemos llamar si instanciamos nuestra clase repositorio desde cualquier sitio.

Explicaremos por encima cada uno de estos patrones que hemos usado obligatoriamente. Aunque haya otros patrones que hemos usado, incluso sin darnos cuenta varios de ellos (Adapter[16] por ejemplo), los más representativos son:

6.1 Patrón MVVM



Este patrón se basa resumidamente en que tenemos una vista, la cual solo dispone de componentes que la forman y no tiene funcionalidad alguna. Digamos que, si ponemos botones en esa vista, al cliquearlos no habrá ninguna acción ligada a ese evento porque no hay código de ese tipo ligado a ella, no hay ninguna clase controlador que la dirija. Para ello, hay que asignarle una clase que la controle.

El ViewModel es la clase intermedia entre la vista y el modelo y hace de capa de presentación. En nuestro caso en Xamarin, nuestras vistas son los layout .axml que no tienen ninguna funcionalidad. Nuestra clase Activity es la que hace de ViewModel pero de una manera especial. La comunicación

entre Activity y la Vista se hace capturando la Vista desde la Activity, es decir, que una misma vista puede ser capturada por muchas clases y funcionar de manera distinta según la clase que la controle.

En nuestro caso, no tenemos un modelo comparable con el del esquema. Nuestra Activity hace casi todo. Lo más parecido al Modelo sería nuestra clase repositorio y a su vez nuestro servicio web. Que explicaremos seguidamente.

6.2 Patrón Repository

En pocas palabras, un repositorio es un mediador entre el dominio de la aplicación y los datos que le dan persistencia. Con este planteamiento podemos pensar que el usuario de este repositorio no necesitaría conocer la tecnología utilizada para acceder a los datos, sino que le bastaría con saber las operaciones que nos facilita este “mediador”, el repositorio.

En la asignatura de MDS2 implementábamos este patrón a través de ORM, sin embargo, aquí al ser una aplicación móvil en la cual queremos que los datos se compartan para todo el mundo, nos hemos decantado por utilizar servicios web. Es decir, disponemos de una base de datos albergada en un servidor y un servicio web albergado en el mismo servidor. El servicio web por medio de json codifica la información contenida en dicha base de datos y la ofrece por medio de HTTP. Nuestra aplicación hace las funciones de CRUD (Create, Read, Update, Delete) por medio de sencillas llamadas por HTTP.

Sin embargo, es un poco ineficaz a nivel de código crear la llamada cada vez que necesitamos hacer algo con nuestro servicio web. Por lo que hemos creado una clase llamada Repositorio que alberga métodos que implementan dichas llamadas y esos métodos son accesibles desde las demás clases de nuestro proyecto. Así pues, con una sencilla instanciación de nuestra clase repositorio podemos usar esos métodos, parametrizarlos y disfrutar de las ventajas de la información en la nube.

6.3 Servicios Web con Ruby y Clase Repositorio

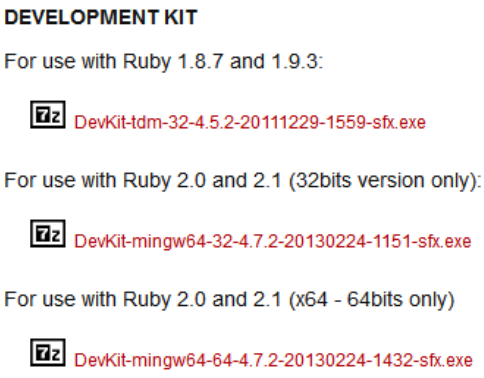
Ruby es un lenguaje de programación interpretado, reflexivo y orientado a objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995. Combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidad con otros lenguajes de programación como Lisp, Lua, Dylan y CLU. Ruby es un lenguaje de programación interpretado en una sola pasada y su implementación oficial es distribuida bajo una licencia de software libre.

En este apartado lo hemos usado para crear nuestro servicio web. La razón principal es la sencillez de este lenguaje interpretado y la rapidez con la que se puede desarrollar.

6.3.1 Instalación de Ruby

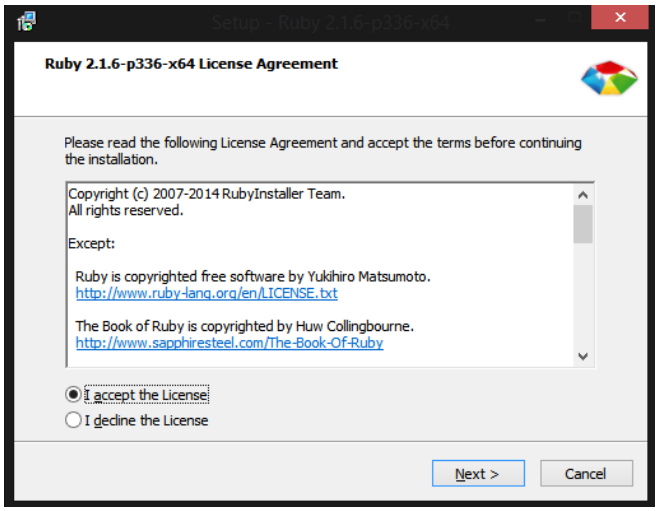
PASO1 :Vamos a la página http://rubyinstaller.org/downloads/

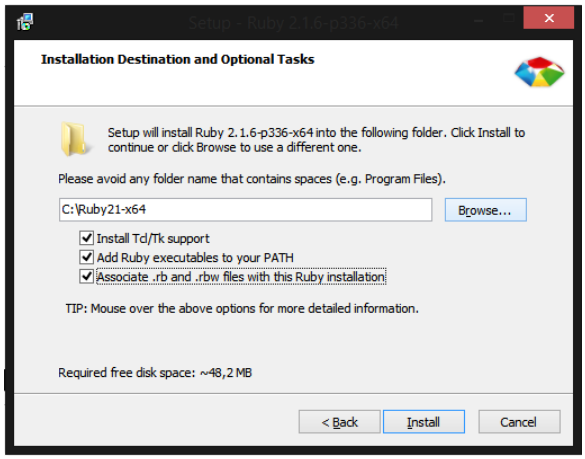
PASO2: Descargamos la versión devkit que nos interese.



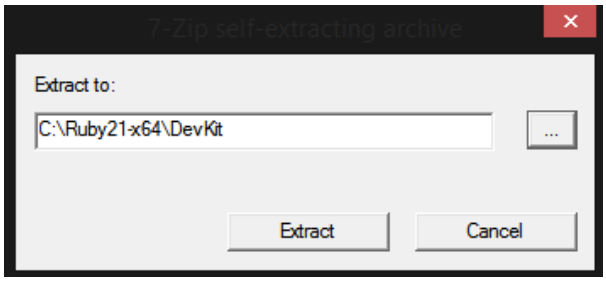
PASO3: Descargamos la misma versión del installer.

PASO4: Instalamos RubyInstaller.





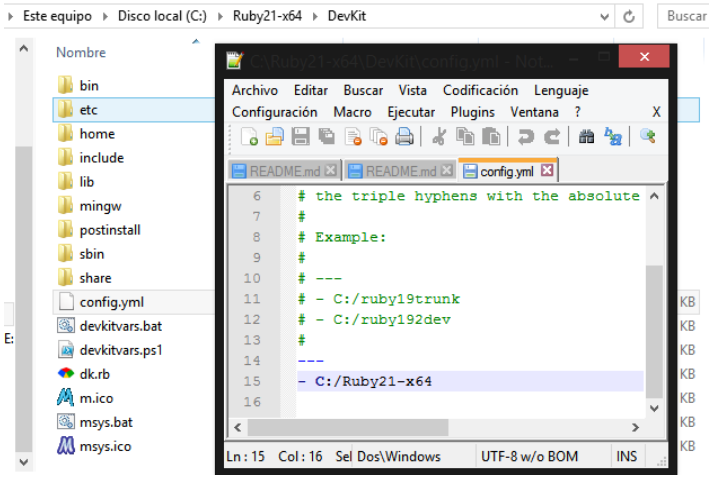
PASO5: Extraemos ele kit de desarrollador en la carpeta donde se instaló Ruby.



PASO6: Instalamos DevKit.

```
C:\Ruby21-x64\DevKit>ruby dk.rb init
C:\Ruby21-x64\DevKit>ruby dk.rb install
```

PASO7: Si lo extraemos en otro sitio y lo instalamos hemos de asegurarnos que en el archivo config.yml se encuentra la dirección a la carpeta donde está instalado Ruby.



6.3.2 Instalacion de RubyOnRails

Una vez instalado Ruby, para instalar el framework RubyOnRails solo necesitaremos instalarlo como una 'gema' y solo tendremos que ejecutar el siguiente comando.

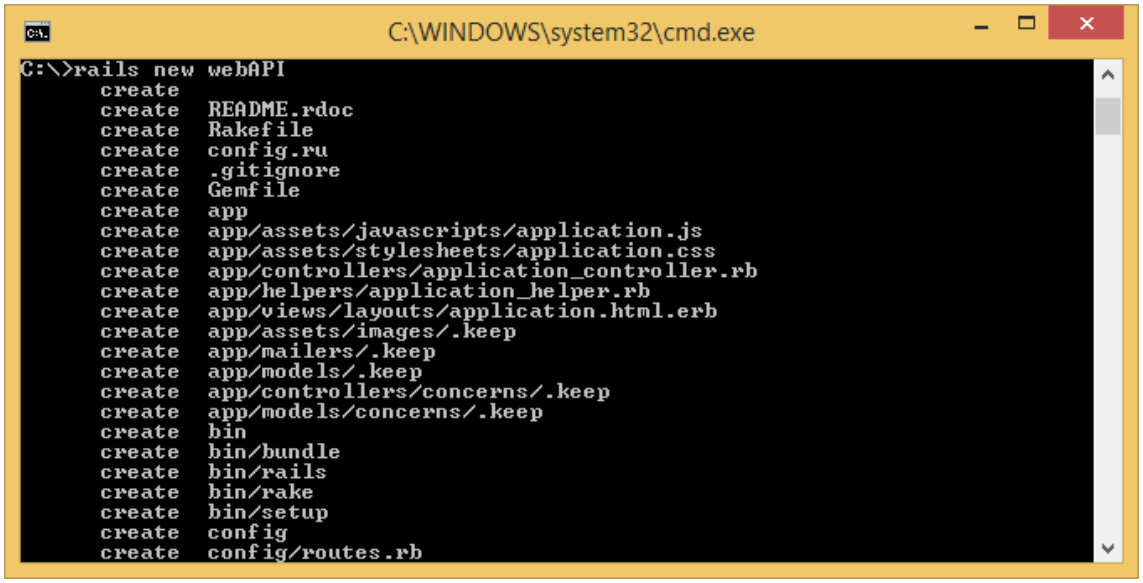
```
C:\Ruby21-x64\DevKit>gem install rails
```

Una vez instalado tenemos dos importates gemas de Ruby:

- El gestor de dependencias Bundler.
- El gestor de tareas y automatización Rake.

6.3.3 Creación del proyecto webAPI

Tan solo tendremos que ejecutar el siguiente comando y ya tendremos el proyecto base sobre el que trabajaremos.



Como se puede observar nos crea un árbol de directorios con la configuración básica en una arquitectura MVC.

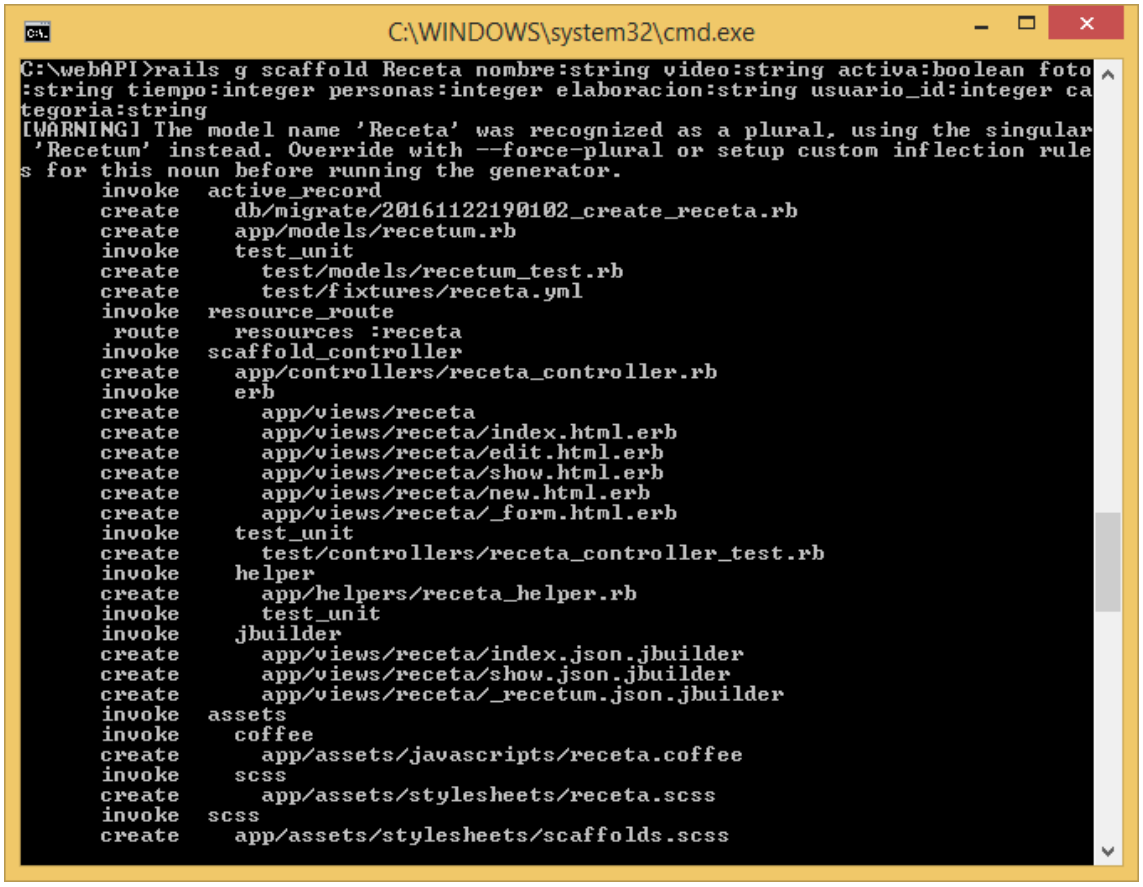
6.4 Creación de la base de datos.

Crearemos la base de datos tabla por tabla con el siguiente comando “rails g scaffold <nombreTabla> <nombreAtributo1>:<tipoAtributo1>…<nombreAtributoN>:<tipoAtributoN>”

Lo que va a hacer este comando es:

- Generar el modelo básico de la tabla sin asociaciones.
- Generar el controlador básico de la tabla CRUD.
- Generar las vistas por defecto de la tabla CRUD.
- Generar un archivo con las migraciones que deben hacerse a la base de datos.

Ejemplo con tabla Receta

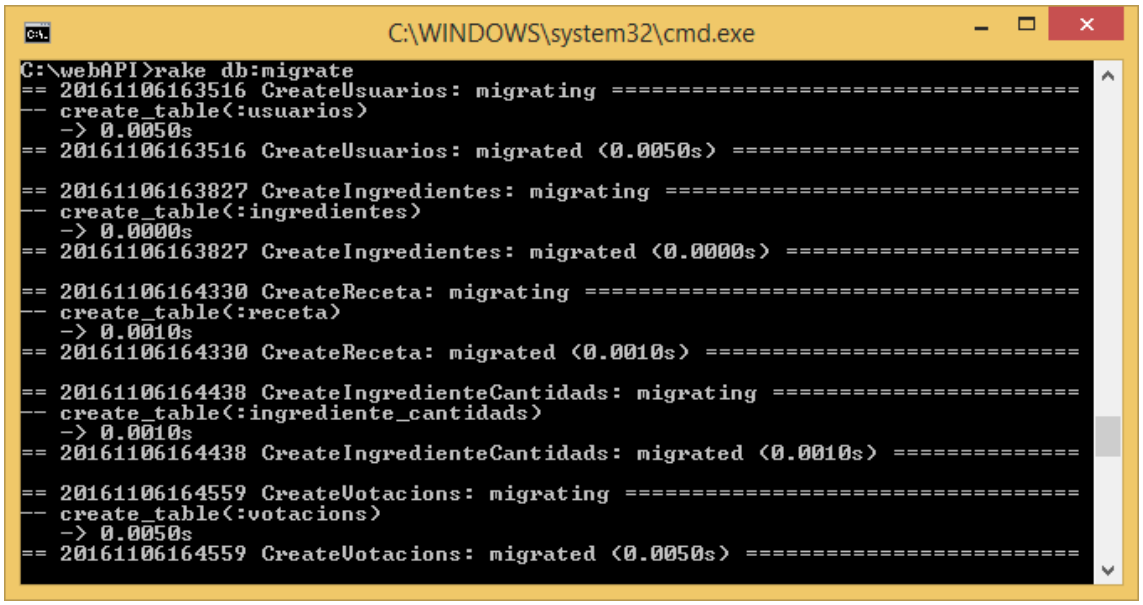


En la imagen de arriba se muestra el comando para crear la tabla receta y sus atributos, así como el resultado que proporciona, generando el modelo, el controlador y la vista, así como la migración que debe hacer a la base de datos.

6.4.1 Migrando las tablas a la base de datos

Una vez hemos ejecutado este comando tantas veces como tablas tengamos, procedemos a la migración de estas tablas a la base de datos. Por defecto RubyOnRails vuelca las tablas en SQLITE pero si quisiéramos hacerlo en mongo o en SQLServer, tan solo tendríamos que cambiar la configuración por defecto e instalar la gema de la base de datos que queramos.

Para migrar las tablas se utiliza el siguiente comando ("rake db:migrate").



Como se puede observar nos hace las migraciones a la base de datos sin problemas, si surgiera algún error fallaría la migración.

6.4.2 Asociaciones de la base de datos

En este caso las asociaciones se gestionan desde Rails por tanto es en el modelo donde tendremos que añadir las asociaciones. Para las asociaciones en Rails existen dos tipos por modelo que pueden ser:

- Belongs_to: aquellas asociaciones con belongs_to son aquellas que son 1 a 1.
- Has_many: aquellas asociaciones con has_many son aquellas que son 1 a muchos.

Por tanto un modelo que viene de una tabla tendrá tantas asociaciones entre tablas como sean necesarias para cubrir todas las relaciones entre las tablas.

Sobre belongs_to decir que en el hash que lo acompaña debe estar el nombre de la tabla en singular (Hay que tener cuidado ya que Rails a veces cambia los nombres de las tablas si cree que están en plural).

Sobre has_many decir que en el hash que lo acompaña debe estar el nombre de la tabla en plural (Hay que tener cuidado ya que Rails a veces cambia el nombre de la tabla si cree que esta en plural, de tal forma que no cambiaría para has_many).

class Ingrediente

```
# File app/models/ingrediente.rb, line 2

has_many :ingrediente_cantidads
```

class IngredienteCantidad

```
# File app/models/ingrediente_cantidad.rb, line 2
belongs_to :recetum
belongs_to :ingrediente
```

class Recetum

```
# File app/models/recetum.rb, line 2
belongs_to :usuario
has_many :ingrediente_cantidads
has_many :votacions
```

class Usuario

```
# File app/models/usuario.rb, line 2
belongs_to :recetum
has_many :votacions
```

class Votacion

```
# File app/models/votacion.rb, line 2
belongs_to :usuario
belongs_to :recetum
```

6.4.3 Modificando las vistas

Una vez que ya tenemos las asociaciones vamos a cambiar el código base de las vistas como sigue para que todas las tablas muestren la información como queramos.

En este caso vamos a tener 5 vistas para cada una de las tablas que tengamos, estas serán:

- **index.html.erb**, que mostrara un listado de todos los elementos de la tabla.
  Esta vista contendrá este código**:**
  <%=@elementos%>
  En Ruby este código significa que va a mostrar el valor de la variable "elementos", el "@" quiere decir que es pública.
- **show.html.erb**, que mostrará uno a uno todos los elementos de la tabla.
  Esta vista contendrá este código**:**
  <%=@elemento%>
  En Ruby este código significa que va a mostrar el valor de la variable "elemento", el "@" quiere decir que es pública.
- **create.html.erb**, que mostrará el elemento recién creado en la tabla.
  Esta vista contendrá este código**:**
  <%=@creado%>
  En Ruby este código significa que va a mostrar el valor de la variable "creado", el "@" quiere decir que es pública.
- **update.html.erb**, que mostrará el elemento recién modificado de la tabla.
  Esta vista contendrá este código**:**
  <%=@actualizado%>
  En Ruby este código significa que va a mostrar el valor de la variable "actualizado", el "@" quiere decir que es pública.
- **destroy.html.erb**, esta mostrará "true" si el elemento se ha eliminado correctamente o "false" en caso contrario.
  Esta vista contendrá este código**:**
  <%=@del%>
  En Ruby este código significa que va a mostrar el valor de la variable "del", el "@" quiere decir que es pública.

6.4.4 Modificando los controladores

Una vez sepamos las tablas que tenemos, sus asociaciones y qué va a mostrar cada vista, procedemos a modificar los controladores de cada tabla para que se muestre en la vista lo que necesitamos. En las siguientes páginas se mostrará el código de los controladores y posteriormente se hablara sobre el mismo, explicando cada aspecto de este código.

class IngredienteCantidadsController

create()

```
# File app/controllers/ingrediente_cantidads_controller.rb, line 11
def create
 ing=JSON.parse(params[:json].gsub('++','.'))
 @ingredienteCantidad = IngredienteCantidad.new
 @ingredienteCantidad.cantidad=ing["cantidad"];
 @ingredienteCantidad.ingrediente_id=ing["ingrediente_id"];
 @ingredienteCantidad.recetum_id=ing["recetum_id"];

 @ingredienteCantidad.save

 @creado=@ingredienteCantidad.to_json(:except=>[:updated_at,:created_at])
 end
```

destroy()

```
# File app/controllers/ingrediente_cantidads_controller.rb, line 36
def destroy
  @ingredienteCantidad = IngredienteCantidad.find(params[:id])
  @ingredienteCantidad.destroy
  @del=true
end
```

index()

```
# File app/controllers/ingrediente_cantidads_controller.rb, line 3
def index
  @elementos = IngredienteCantidad.all.to_json(:except=>[:updated_at,:created_at])
end
```

show()

```
# File app/controllers/ingrediente_cantidads_controller.rb, line 7
def show
    @elemento = IngredienteCantidad.find(params[:id]).to_json(:except=>[:updated_at,:created_at])
end
```

update()

```
# File app/controllers/ingrediente_cantidads_controller.rb, line 23
def update

ing=JSON.parse(params[:json].gsub('++','.'))
  @ingredienteCantidad = IngredienteCantidad.find(params[:id])
      @ingredienteCantidad.cantidad=ing["cantidad"];
  @ingredienteCantidad.ingrediente_id=ing["ingrediente_id"];
  @ingredienteCantidad.recetum_id=ing["recetum_id"];

  @ingredienteCantidad.save

  @actualizado=@ingredienteCantidad.to_json(:except=>[:updated_at,:created_at])
end
```

update()

```
# File app/controllers/ingredientes_controller.rb, line 33
def update

ing=JSON.parse(params[:json].gsub('++','.'))
  @ingrediente = Ingrediente.find(params[:id])
  @ingrediente.nombre=ing["nombre"];
  @ingrediente.proteinas=ing["proteinas"];
  @ingrediente.grasa=ing["grasa"];
  @ingrediente.carbohidratos=ing["carbohidratos"];
  @ingrediente.calorias=ing["calorias"];
  @ingrediente.calcio=ing["calcio"];
  @ingrediente.hierro=ing["hierro"];
      @ingrediente.potasio=ing["potasio"];
  @ingrediente.magnesio=ing["magnesio"];
  @ingrediente.sodio=ing["sodio"];
  @ingrediente.fosforo=ing["fosforo"];
  @ingrediente.ioduro=ing["ioduro"];
  @ingrediente.selenio=ing["selenio"];
  @ingrediente.zinc=ing["zinc"];
  @ingrediente.save

  @actualizado=@ingrediente.to_json(:except=>[:updated_at,:created_at])
end
```

class IngredientesController

create()

```
# File app/controllers/ingredientes_controller.rb, line 11
def create
  ing=JSON.parse(params[:json].gsub('++','.'))
  @ingrediente = Ingrediente.new
  @ingrediente.nombre=ing["nombre"];
  @ingrediente.proteinas=ing["proteinas"];
  @ingrediente.grasa=ing["grasa"];
  @ingrediente.carbohidratos=ing["carbohidratos"];
  @ingrediente.calorias=ing["calorias"];
  @ingrediente.calcio=ing["calcio"];
  @ingrediente.hierro=ing["hierro"];
  @ingrediente.potasio=ing["potasio"];
  @ingrediente.magnesio=ing["magnesio"];
  @ingrediente.sodio=ing["sodio"];
  @ingrediente.fosforo=ing["fosforo"];
  @ingrediente.ioduro=ing["ioduro"];
  @ingrediente.selenio=ing["selenio"];
  @ingrediente.zinc=ing["zinc"];
  @ingrediente.save

  @creado=@ingrediente.to_json(:except=>[:updated_at,:created_at])
  end
```

destroy()

```
# File app/controllers/ingredientes_controller.rb, line 56
def destroy
  @ingrediente = Ingrediente.find(params[:id])
  @ingrediente.destroy
  @del=true
end
```

index()

```
# File app/controllers/ingredientes_controller.rb, line 3
def index
  @elementos = Ingrediente.all.to_json(:except=>[:updated_at,:created_at])
end
```

show()

```
# File app/controllers/ingredientes_controller.rb, line 7
def show
    @elemento = Ingrediente.find(params[:id]).to_json(:except=>[:updated_at,:created_at])
  end
```

class RecetaController

create()

```
# File app/controllers/receta_controller.rb, line 11
def create
  receta=JSON.parse(params[:json].gsub('++','.'))
  @recetum = Recetum.new
  @recetum.nombre=receta["nombre"];
  @recetum.video=receta["video"];
  @recetum.activa=receta["activa"];
  @recetum.foto=receta["foto"];
  @recetum.tiempo=receta["tiempo"];
  @recetum.personas=receta["personas"];
  @recetum.elaboracion=receta["elaboracion"];
  @recetum.usuario_id=receta["usuario_id"];
  @recetum.categoria=receta["categoria"];
  @recetum.save

  @creado=@recetum.to_json(:include => {
  :ingrediente_cantidads =>
  {:include => {:ingrediente =>{:except=>[:updated_at,:created_at]}},
  :except=>[:updated_at,:created_at]}},:except=>[:updated_at,:created_at])
end
```

destroy()

```
# File app/controllers/receta_controller.rb, line 47
def destroy
  @recetum = Recetum.find(params[:id])
  @recetum.destroy
  @del=true
end
```

index()

```
# File app/controllers/receta_controller.rb, line 3
def index
  @elementos = Recetum.to_json(:include => {
  :ingrediente_cantidads =>
  {:include => {:ingrediente =>{:except=>[:updated_at,:created_at]}},
  :except=>[:updated_at,:created_at]}},:except=>[:updated_at,:created_at])
end
```

show()

```
# File app/controllers/receta_controller.rb, line 7
def show
  @elemento = Recetum.find(params[:id]).to_json(:include => {
  :ingrediente_cantidads =>
  {:include => {:ingrediente =>{:except=>[:updated_at,:created_at]}},
  :except=>[:updated_at,:created_at]}},:except=>[:updated_at,:created_at])
end
```

update()

```
# File app/controllers/receta_controller.rb, line 28
def update

receta=JSON.parse(params[:json].gsub('++','.'))
  @recetum = Recetum.find(params[:id])
  @recetum.nombre=receta["nombre"];
  @recetum.video=receta["video"];
  @recetum.activa=receta["activa"];
  @recetum.foto=receta["foto"];
  @recetum.tiempo=receta["tiempo"];
  @recetum.personas=receta["personas"];
  @recetum.elaboracion=receta["elaboracion"];
  @recetum.usuario_id=receta["usuario_id"];
  @recetum.categoria=receta["categoria"];
  @recetum.save

  @actualizado=@recetum.to_json(:include => {
  :ingrediente_cantidads =>
  {:include => {:ingrediente =>{:except=>[:updated_at,:created_at]}},
  :except=>[:updated_at,:created_at]}},:except=>[:updated_at,:created_at])

end
```

class VotacionsController

create()

```
# File app/controllers/votacions_controller.rb, line 11
def create
  ing=JSON.parse(params[:json].gsub('++','.'))
  @votacion = Votacion.new
  @votacion.voto=ing["voto"];
  @votacion.usuario_id=ing["usuario_id"];
  @votacion.recetum_id=ing["recetum_id"];
  @votacion.save
  @creado=@votacion.to_json(:except=>[:updated_at,:created_at])
  end
```

destroy()

```
# File app/controllers/votacions_controller.rb, line 36
def destroy
  @votacion = Votacion.find(params[:id])
  @votacion.destroy
  @del=true
end
```

index()

```
# File app/controllers/votacions_controller.rb, line 3
def index
  @elementos = Votacion.all.to_json(:except=>[:updated_at,:created_at])
end
```

show()

```
# File app/controllers/votacions_controller.rb, line 7
def show
    @elemento = Votacion.find(params[:id]).to_json(:except=>[:updated_at,:created_at])
end
```

update()

```
# File app/controllers/votacions_controller.rb, line 23
def update

ing=JSON.parse(params[:json].gsub('++','.'))
  @votacion = Votacion.find(params[:id])
      @votacion.voto=ing["voto"];
  @votacion.usuario_id=ing["usuario_id"];
  @votacion.recetum_id=ing["recetum_id"];
  @votacion.save
  @actualizado=@votacion.to_json(:except=>[:updated_at,:created_at])
end
```

class UsuariosController

create()

```
# File app/controllers/usuarios_controller.rb, line 11
def create
user=JSON.parse(params[:json].gsub('++','.'))
@usuario = Usuario.new
@usuario.nombre=user["nombre"];
@usuario.nick=user["nick"];
@usuario.apellidos=user["apellidos"];
@usuario.email=user["email"];
@usuario.clave=user["clave"];
@usuario.faltas=user["faltas"];
@usuario.administrador=user["administrador"];
@usuario.save

@creado=@usuario.to_json(:except=>[:updated_at,:created_at])
end
```

destroy()

```
# File app/controllers/usuarios_controller.rb, line 42
def destroy
  @usuario = Usuario.find(params[:id])
  @usuario.destroy
  @del=true
end
```

index()

```
# File app/controllers/usuarios_controller.rb, line 3
def index
  @elementos = Usuario.all.to_json(:except=>[:updated_at,:created_at])
end
```

show()

```
# File app/controllers/usuarios_controller.rb, line 7
def show
    @elemento = Usuario.find(params[:id]).to_json(:except=>[:updated_at,:created_at])
  end
```

update()

```
# File app/controllers/usuarios_controller.rb, line 26
def update

user=JSON.parse(params[:json].gsub('++','.'))
  @usuario = Usuario.find(params[:id])
  @usuario.nombre=user["nombre"];
  @usuario.nick=user["nick"];
  @usuario.apellidos=user["apellidos"];
  @usuario.email=user["email"];
  @usuario.clave=user["clave"];
  @usuario.faltas=user["faltas"];
  @usuario.administrador=user["administrador"];
  @usuario.save

  @actualizado=@usuario.to_json(:except=>[:updated_at,:created_at])
end
```

Como se puede observar todos los controladores tienen los mismos métodos y estos tienen el mismo nombre que las vistas en cada uno de ellos.

En el método Index se obtienen todos los elementos de la tabla, y se parsea a json quitando los atributos :updated_at y :created_at que genera rails por defecto en todas las tablas. En index de Receta por ejemplo se puede observar cómo podemos incluir en el parseo a json, aquellas tablas asociadas que se requieran.

En el método Show se obtiene un elemento de la tabla el cual se encuentra con el método find(:id) y volvemos a parsear a json.

En el método create se obtiene primero un json con los datos que se quieren añadir a la tabla.

Para crear el elemento primero se parsea el json para convertirlo en formato hash o diccionario en el que los elementos sean accesibles, luego se crea una instancia del elemento que se vaya a añadir a la tabla, añadimos los parámetros del json y guardamos el elemento en la base de datos con el método save. Luego mostramos el elemento creado en la vista.

Para actualizar un elemento primero se busca el elemento en la tabla, entonces se parsea el json para convertirlo en formato hash o diccionario en el que los elementos sean accesibles, luego se modifican los atributos del elemento por los otorgados en el json y guardamos el elemento con save, lo que actualizara el elemento en la base de datos. Luego mostramos el elemento creado en la vista.

Para eliminar un elemento se busca el elemento en la tabla a través de su id, si existe se elimina y se devuelve un True en la vista si no se muestra en la False a la vista.

6.4.5 Acceso por URL

Ahora ya tenemos las vistas, los controladores y los modelos, pero estos no son accesibles porque no tienen una ruta definida para acceder a ellos por lo tanto vamos a entrar a config/routes.rb en el proyecto y para poder acceder a los datos como se requiera.

GET	/	receta#index
GET	/receta	receta#index
GET	/receta/:id	receta#show
GET	/receta/crear/:json	receta#create
GET	/receta/actualizar/:id/:json	receta#update
GET	/receta/eliminar/:id	receta#destroy
GET	/usuario	usuarios#index
GET	/usuario/:id	usuarios#show
GET	/usuario/crear/:json	usuarios#create
GET	/usuario/actualizar/:id/:json	usuarios#update
GET	/usuario/eliminar/:id	usuarios#destroy
GET	/ingrediente	ingredientes#index
GET	/ingrediente/:id	ingredientes#show
GET	/ingrediente/crear/:json	ingredientes#create
GET	/ingrediente/actualizar/:id/:json	ingredientes#update
GET	/ingrediente/eliminar/:id	ingredientes#destroy
GET	/ingrediente_cantidad	ingrediente_cantidads#index
GET	/ingrediente_cantidad/:id	ingrediente_cantidads#show
GET	/ingrediente_cantidad/crear/:json	ingrediente_cantidads#create
GET	/ingrediente_cantidad/actualizar/:id/:json	ingrediente_cantidads#update
GET	/ingrediente_cantidad/eliminar/:id	ingrediente_cantidads#destroy
GET	/votacion	votacions#index
GET	/votacion/:id	votacions#show
GET	/votacion/crear/:json	votacions#create
GET	/votacion/actualizar/:id/:json	votacions#update
GET	/votacion/eliminar/:id	votacions#destroy

Y esta sería la codificacion

```
# File config/routes.rb

Rails.application.routes.draw do

  root 'receta#index'
get 'receta'=>  'receta#index'
 get 'receta/:id'=>  'receta#show'
get 'receta/crear/:json'=>  'receta#create'
get 'receta/actualizar/:id/:json'=>  'receta#update'
get 'receta/eliminar/:id'=>  'receta#destroy'
 get 'usuario'=>  'usuarios#index'
get 'usuario/:id'=>  'usuarios#show'
get 'usuario/crear/:json'=>  'usuarios#create'
get 'usuario/actualizar/:id/:json'=>  'usuarios#update'
get 'usuario/eliminar/:id'=>  'usuarios#destroy'
 get 'ingrediente'=>  'ingredientes#index'
get 'ingrediente/:id'=>  'ingredientes#show'
get 'ingrediente/crear/:json'=>  'ingredientes#create'
get 'ingrediente/actualizar/:id/:json'=>  'ingredientes#update'
get 'ingrediente/eliminar/:id'=>  'ingredientes#destroy'
 get 'ingrediente_cantidad'=>  'ingrediente_cantidads#index'
get 'ingrediente_cantidad/:id'=>  'ingrediente_cantidads#show'
get 'ingrediente_cantidad/crear/:json'=>  'ingrediente_cantidads#create'
get 'ingrediente_cantidad/actualizar/:id/:json'=>  'ingrediente_cantidads#update'
get 'ingrediente_cantidad/eliminar/:id'=>  'ingrediente_cantidads#destroy'
 get 'votacion'=>  'votacions#index'
get 'votacion/:id'=>  'votacions#show'
get 'votacion/crear/:json'=>  'votacions#create'
get 'votacion/actualizar/:id/:json'=>  'votacions#update'
get 'votacion/eliminar/:id'=>  'votacions#destroy'

end
```

6.4.6 Inicialización del servicio.

Para inicializarlo tan solo habrá que ejecutar el siguiente comando.

Rails s –p <puerto>

Ahora ya se puede acceder por localhost:2000, para acceder a través de internet, se tendrá que descargar un servidor web ya sea Apache[17] o Nginx[18] y configurar un virtual host con proxypass y reverseproxypass al puerto 2000.

RUTA	DESRIPCION
GET	
URL/usuario/1	Nos devuelve el usuario con id= 1
URL/ingrediente/1	Nos devuelve el ingrediente con id= 1
URL/receta/1	Nos devuelve la receta con id=1
GET LISTADOS	
URL/usuario	Nos devuelve todos los usuarios
URL/ingredientes	Nos devuelve todos los ingredientes
URL/receta	Nos devuelve todas las recetas
POST	
URL/usuario/crear/JSON	Nos crea un usuario nuevo a partir de un JSON
URL/ingrediente/crear/JSON	Nos crea un ingrediente nuevo a partir de un JSON
URL/receta/crear/JSON	Nos crea una receta nueva a partir de un JSON
UPDATE	
URL/usuario/modificar/1/JSON	Modificamos el usuario con id=1 a partir de un JSON modificado
URL/ingrediente/modificar/1/JSON	Modificamos el ingrediente con id= 1 a partir de un JSON modificado
URL/receta/modificar/1/JSON	Modificamos la receta con id= 1 a partir de un JSON modificado
DELETE	
URL/usuario/delete/1	Eliminamos el usuario con id=1
URL/ingrediente/delete/1	Eliminamos el ingrediente con id=1
URL/receta/delete/1	Eliminamos la receta con id=1

Y nuestra clase **Repositorio** albergará los siguientes métodos:

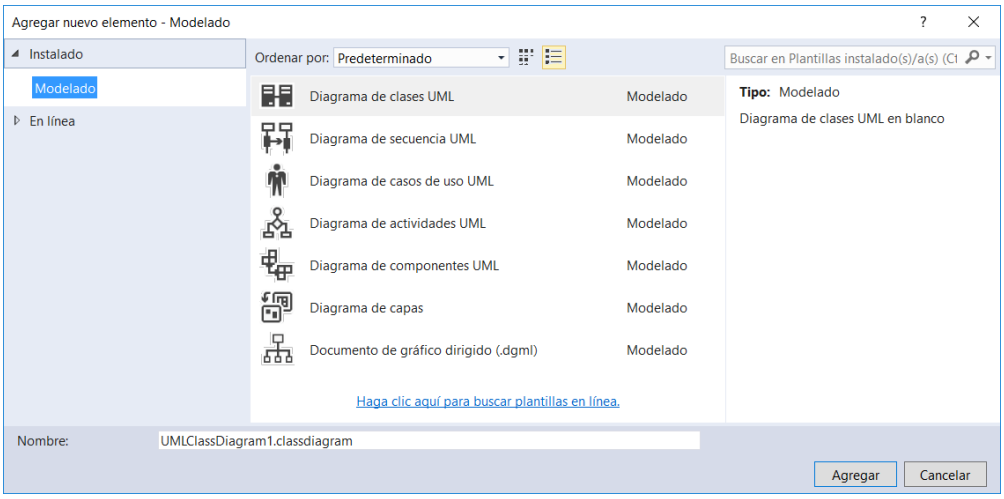
- **public Usuario getUsuario(int id):** Devuelve el objeto usuario con el id pasado por parámetro.
- **public Receta getRecetas(int id):** Devuelve el objeto usuario con el id pasado por parámetro.
- **public Ingrediente getIngredientes(int id):** Devuelve el ingrediente con el id pasado por parámetro.
- **pulic List<Usuario> getListUsuarios(int id):** Devuelve una lista de todos los usuarios
- **public List<Ingrediente> getListIngredientes(int id):** retorna una lista de todos los ingredientes.
- **public List<Receta> getListRecetas(int id):** retorna una lista de todas las recetas.
- **public void crearUsuario(string json):** crea un usuario a partir de un JSON.
- **public void crearIngrediente(string json):** crea un usuario a partir de un JSON.
- **public void crearReceta(string json):** crea una receta a partir de un JSON.
- **public void modificarUsuario(int id, string json):** modifica un usuario con un id.
- **public void modificarIngrediente(int id, string json):** modifica un ingrediente con un id.
- **public void modificarReceta(int id, string json):** modifica una receta con un id.
- **public void borrarUsuario(int id):** borra el usuario id.
- **public void borrarIngrediente(int id):** borra el ingrediente id.
- **public void borrarReceta(int id):**borra la receta id.

7 Aplicación de la Metodología de Modelado de Software a Visual Studio con Xamarin

7.1 Introducción

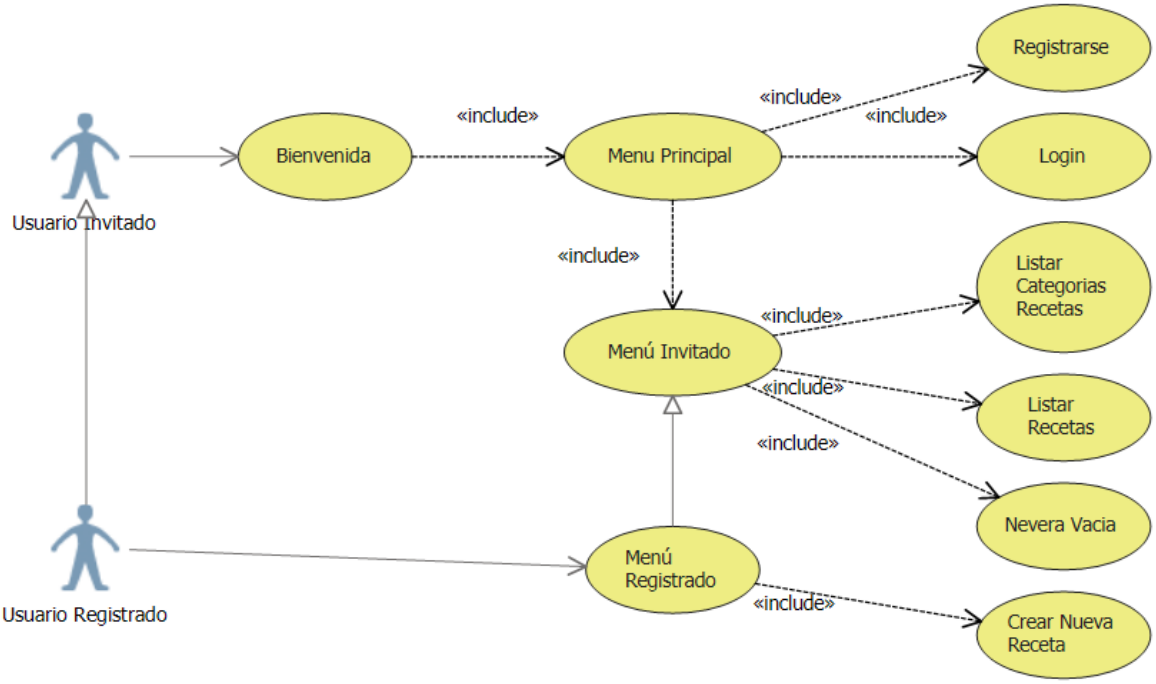
Visual Studio 2015 Enterprise nos brinda una serie de herramientas de modelado suficientes para hacer un proyecto de modelado completo. Sin embargo, por el momento dichas herramientas no están del todo integradas en todos los tipos de proyectos, es decir, Visual Studio 2015 nos deja crear proyectos específicos de modelado de manera individual y no nos deja adjuntar los resultados de dichos proyectos a otro tipo de Proyecto. Si por ejemplo generamos de un diagrama de clases, código a través de generador de código, nos genera una carpeta interna al proyecto con las clases generadas. Dichas clases son clases en C# básico y no se asemejan en nuestro caso a las clases Activity, que en primer lugar, precisan heredar de la clase Activity para poder ser funcional a nivel de Android. Este mismo problema lo encontramos con la integración de Visual Paradigm en Visual Studio, en el que no podía integrar un proyecto de modelado con otro de Android. Así pues, llegamos a la solución de ese problema usando una técnica ajena al proyecto de modelado y propia del proyecto de Android. De esta manera, se podía constituir un modelado completo siguiendo la metodología de modelado elegida, usando varias herramientas diferentes.

Aplicando esta técnica, perdemos el concepto de trazabilidad entre casos de uso y clases. Con Visual Paradigm era posible hacer que la trazabilidad fuese semiautomática. Podíamos definir de donde venía un caso de uso desde la especificación preliminar, en qué clase derivaba y viceversa, y además todo ello podía ser visualizado en un informe gracias a la Herramienta Grid que generaba una tabla con todos los detalles que quisiéramos. En nuestro caso y con nuestras herramientas no va a ser posible. Seguidamente mostraremos con unos ejemplos recogidos de nuestro proyecto, como vamos a ir avanzando progresivamente en la generación de todos los diagramas exigidos.



7.2 Diagrama de Casos de Uso

Respecto a los diagramas de casos de uso, al estar ausente la trazabilidad automática y no tener que ser dependientes de ella, podríamos haberlo hecho con cualquier herramienta de modelado. Podríamos haberlo hecho con Visual Paradigm u otra herramienta diferente. En este caso, hemos escogido la propia herramienta de Visual Studio y hecho un Diagrama de Casos de uso informal, justo para entender el concepto de la aplicación sin entrar en detalles de qué componentes internos utilizaríamos. Empezaremos explicando este pequeño recorte de nuestro diagrama.



Este diagrama nos muestra la interacción que tendría un usuario invitado con el sistema. El **Usuario Invitado** puede acceder a la sección de **Bienvenida** desde la cual puede acceder al **Menú Principal** en el que puede elegir si **Registrarse**, hacer **Login** o acceder al **Menú Invitado**. En el Menú Invitado solo se le ofrecen las opciones de **Listar las Categorías de las Recetas**, **Listar las Recetas** y **Acceder a la aplicación Nevera Vacía**.

Seguidamente vemos que el **Usuario Registrado** hereda todas las acciones que puede realizar el **Usuario Invitado**, sin embargo, vemos que añade un nuevo caso de uso Llamado **Menú Registrado** que hereda a su vez de Menú Invitado adquiriendo todas las acciones que se dan lugar en dicho menú. Pero con una acción más, **Crear Nueva Receta.**

En definitiva, el **Usuario Registrado** puede hacer todo lo que hace el **Invitado** pero además en su menú le da la opción de poder **Crear una Receta.** Este diagrama simple nos daría pie a crear nuestro diagrama de clases.

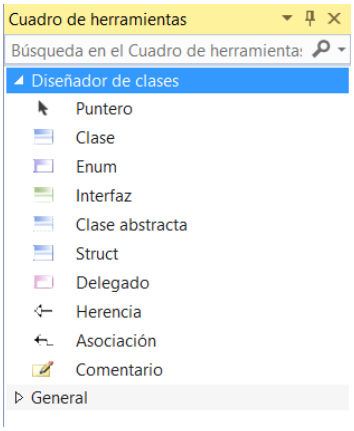
7.3 Diagrama de Clases

Como se mencionó en la Introducción de esta sección, no veíamos funcional hacer un diagrama de clases utilizando las herramientas de modelado propias de un proyecto de modelado en Visual Studio. Sin embargo y como consta anteriormente también, se encontró una alternativa a la herramienta de modelado muy útil y con muchas ventajas.

Al igual que ventajas tiene sus desventajas respecto a un diagrama de clases clásico.

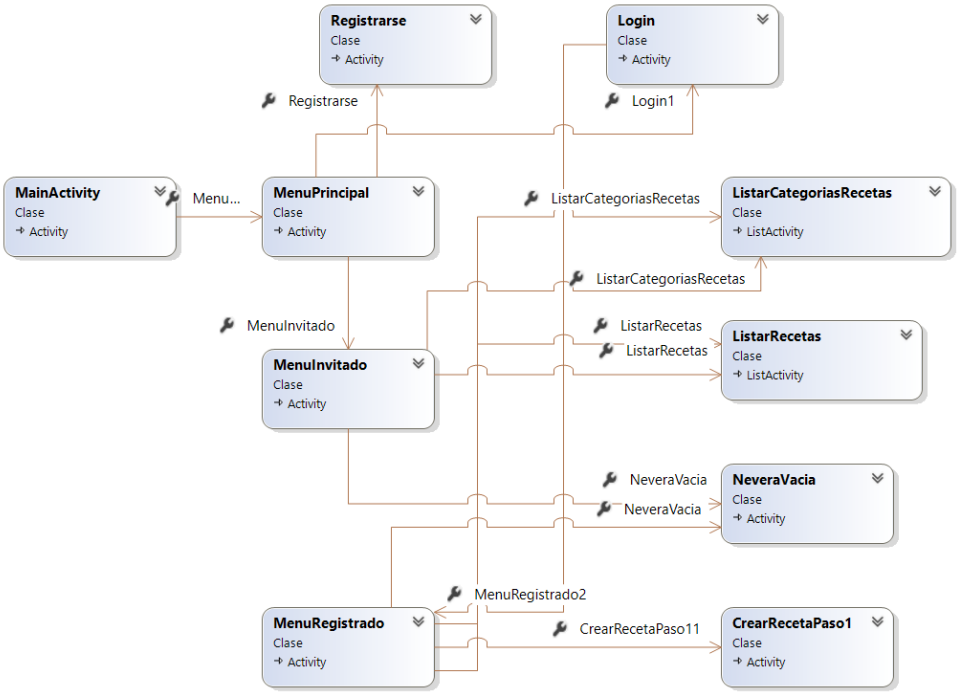
Para iniciar la herramienta tan solo necesitamos irnos a una clase activity, dar al botón derecho del ratón y se nos desplegará un menú donde debemos seleccionar la opción “Ver Diagrama de Clases”.

Y nos aparecerá un Layout con nuestra clase dibujada y donde podremos ir añadiendo las clases que queramos. Hay dos opciones, la primera es crear la clase y después arrastrarla al diagrama y la otra es en el diagrama crear una nueva clase que automáticamente se genera y se actualiza en tiempo real en nuestra carpeta de clases. Es decir, los cambios son bidireccionales, cualquier cosa que variemos en el diagrama de clases se modificará en nuestras clases físicas y viceversa. Es una ventaja con respecto de Visual Paradigm ya que para hacer ingeniería inversa o directa teníamos que hacerlo por medio de selección de la operación concreta. Sin embargo, esto hacernos perder mucha información sin darnos cuenta. Si borramos una clase en el diagrama se borra el archivo de nuestra carpeta.



En Visual Paradigm teníamos un cuadro de Herramientas más completo que albergaba relaciones de composición, agregación, etc. Sin embargo, con las herramientas que tenemos nos es suficiente para modelar estas aplicaciones en Android basadas en Activities.

Nuestro Diagrama resultando es el siguiente



Trazabilidad

CASO DE USO	CLASE
Bienvenida	MainActivity
Menú Principal	MenuPrincipal
Menú Invitado	MenuInvitado
Menú Registrado	MenuRegistrado
Registrarse	Registrarse
Login	Login
Listar Categorias Recetas	ListarCategoriasRecetas
Listar Recetas	ListarRecetas
Nevera Vacia	NeveraVacia
Crear Receta Paso 1	CrearRecetaPaso1

Vemos que menos la primera clase a la que hemos respetado el nombre que le asigna Xamarin cd MainActivity, todas las clases se asemejan en nombre a su caso de uso correspondiente.

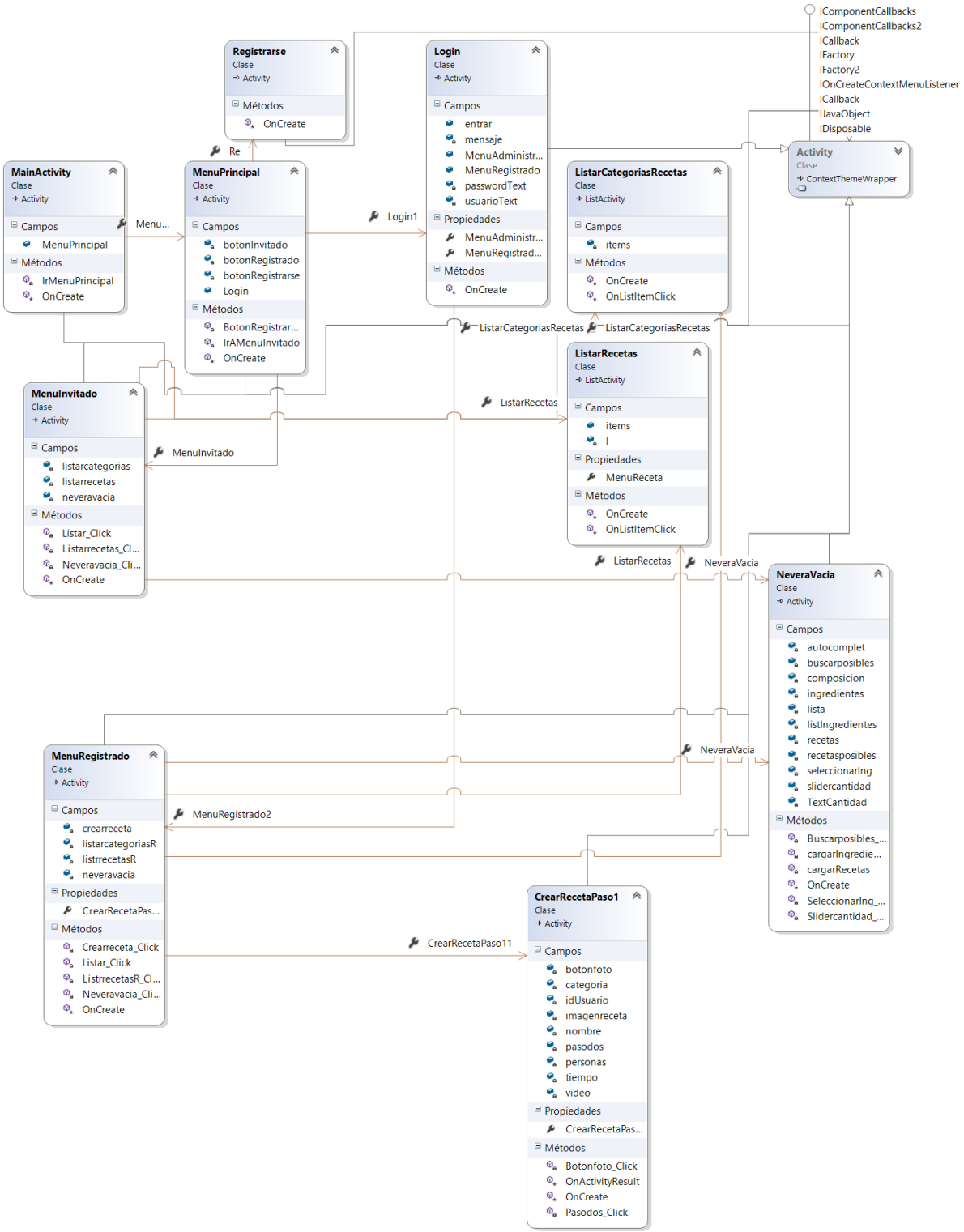
La razón principal por lo que la herencia que teníamos en nuestro diagrama de casos de uso anteriormente haya desaparecido es porque, aunque creemos una herencia a nivel de activities, no podemos crearla a nivel de vistas. Es decir, si hacemos la herencia entre Menú Invitado y Menú registrado. Heredamos todos los métodos que este alberga, al igual que las instancias a componentes. Tendríamos dos clases iguales, pero según nuestro diagrama de casos de uso, Menú registrado difiere con respecto a Menú Invitado en que añade otro caso de uso que es CrearReceta. Eso simboliza que en la vista de Menú Registrado existirá un nuevo botón que nos lleve a la ventana CrearReceta una vez pulsado. So colocamos ese botón en la vista MenuInvitado aparecerá inevitablemente al usuario Invitado y nuestro objetivo es que solo esté presente si el usuario está identificado como registrado. Por lo que la única solución que tenemos en este momento es simbolizar la herencia en el diagrama de casos de uso, pero en el diagrama de clases replicar las relaciones que tenga la clase de la que se hereda.

En cuanto a las relaciones. Estas relaciones solo sirven a modo simbólico ya que, al crear una relación entre una clase y otra, en el código de la clase que lanza la relación, se crea una instancia de la clase destino. Esto no nos importa a nivel funcional, pero si a nivel estético del código puesto que, si esas instancias no las usamos nunca, estamos creando código residual, código que no sirve para nada.

No se usan ya que cuando instanciamos una Activity desde otra, lo hacemos a partir de una secuencia de código muy particular y no podemos hacerlo de otra manera.

Existe una posible solución usando el método getType() de la clase Object y creando una instancia de Intent del mismo tipo que este método nos devuelva, sin embargo, continuamos usando más código del necesario.

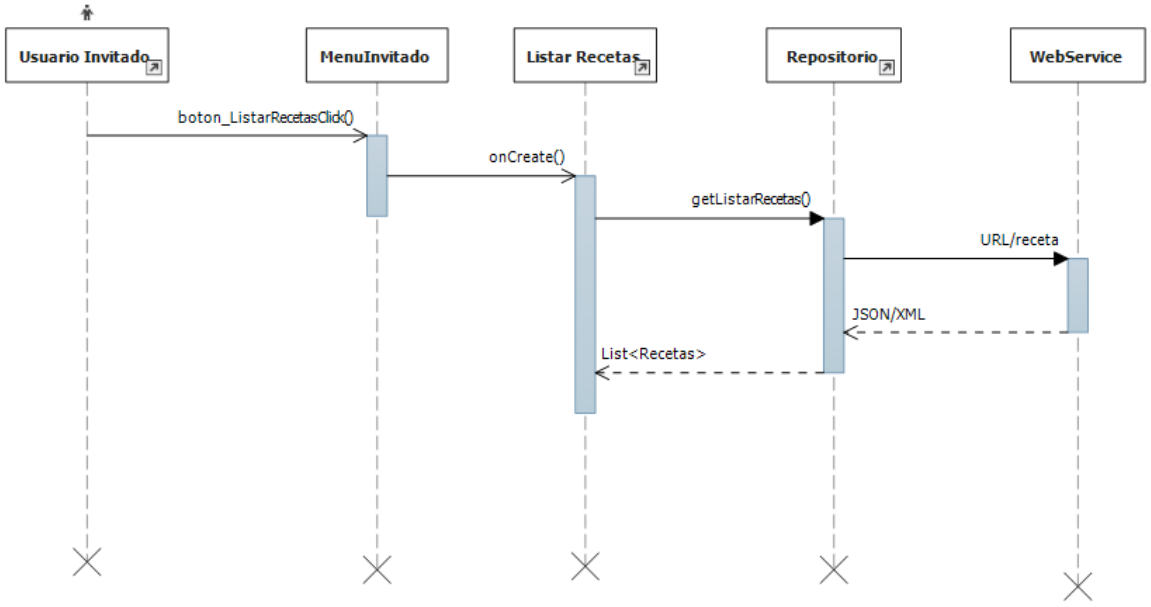
En el siguiente diagrama extendido vemos todas las clases expandidas mostrando todos sus atributos y sus métodos al igual que veíamos con Visual Paradigm. Sin embargo, sabemos cómo se llaman los atributos, pero no sabemos de qué tipo de atributos estamos hablando a menos que pichemos sobre el *in situ* y veamos sus detalles en la ventana de Propiedades.


IComponentCallbacks
IComponentCallbacks2
ICallback
IFactory
IFactory2
IOnCreateContextMenuListener
ICallback
IJavaObject
IDisposable
Activity
Clase
ContextThemeWrapper
Registrarse
Clase
Activity
Métodos
OnCreate
Re
Login
Clase
Activity
Campos
entrar
mensaje
MenuAdministr...
MenuRegistrado
passwordText
usuarioText
Propiedades
MenuAdministr...
MenuRegistrad...
Métodos
OnCreate
MainActivity
Clase
Activity
Campos
MenuPrincipal
Métodos
IrMenuPrincipal
OnCreate
Menu...
MenuPrincipal
Clase
Activity
Campos
botonInvitado
botonRegistrado
botonRegistrarse
Login
Métodos
BotonRegistrar...
IrAMenuInvitado
OnCreate
Login1
ListarCategoriasRecetas
Clase
ListActivity
Campos
items
Métodos
OnCreate
OnListItemClick
ListarCategoriasRecetas
ListarCategoriasRecetas
ListarRecetas
Clase
ListActivity
Campos
items
I
Propiedades
MenuReceta
Métodos
OnCreate
OnListItemClick
ListarRecetas
MenuInvitado
Clase
Activity
Campos
listarcategorias
listarrecetas
neveravacia
Métodos
Listar_Click
Listarrecetas_Cl...
Neveravacia_Cli...
OnCreate
MenuInvitado
ListarRecetas
ListarRecetas
NeveraVacia
NeveraVacia
Clase
Activity
Campos
autocomplet
buscarposibles
composicion
ingredientes
lista
listIngredientes
recetas
recetasposibles
seleccionarIng
slidercantidad
TextCantidad
Métodos
Buscarposibles_...
cargarIngredie...
cargarRecetas
OnCreate
SeleccionarIng_...
Slidercantidad_...
NeveraVacia
MenuRegistrado
Clase
Activity
Campos
crearreceta
listarcategoriasR
listrrecetasR
neveravacia
Propiedades
CrearRecetaPas...
Métodos
Crearreceta_Click
Listar_Click
ListrrecetasR_Cl...
Neveravacia_Cli...
OnCreate
MenuRegistrado2
CrearRecetaPaso11
CrearRecetaPaso1
Clase
Activity
Campos
botonfoto
categoria
idUsuario
imagenreceta
nombre
pasodos
personas
tiempo
video
Propiedades
CrearRecetaPas...
Métodos
Botonfoto_Click
OnActivityResult
OnCreate
Pasodos_Click

7.4 Diagrama de Secuencias

Los Diagramas de Secuencias nos sirven para determinar en cada caso qué llamadas a métodos y cuando se hacen desde una ventana a nuestro Repositorio y éste a nuestra base de datos.

El siguiente diagrama de secuencias corresponde a la ventana **Listar Receta,** nuestro usuario invitado interacciona con la ventana y a partir de aquí se crea el proceso antes mencionado:

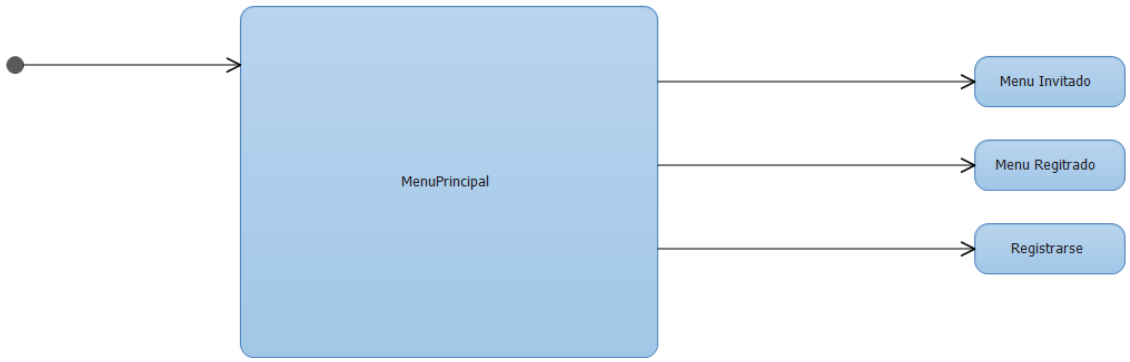


Estos diagramas los hemos creado con la herramienta de modelado propia de Visual Studio. La herramienta dispone de lo necesario para describir las llamadas a métodos que se hacen en muchas de las clases. En este ejemplo vemos que el Usuario Invitado hace Click() en el botón de Listar Recetas del menú Invitado , se instancia una clase Listar Recetas y actúa el método onCreate() presente en todas las Activities. Este método instancia un objeto de clase Repositorio que a su vez contiene el método getListarRecetas() que obtiene el contenido en formato XML de una ruta de nuestro Servicio Web y lo des-serializa en un objeto tipo List<Receta> que devuelve a Listar Recetas.

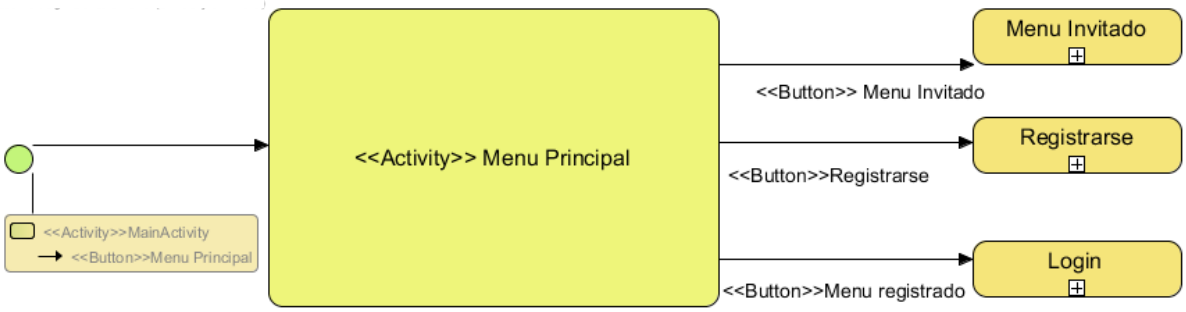
7.5 Diagrama de Estados/Actividades/Negocio

Los diagramas los hemos elaborado con **Visual Paradigm**. No hemos usado la herramienta propia de Modelado de Diagramas de Actividades de Visual Studio porque primer lugar no podemos vincular una tarea con un sub diagrama y en segundo lugar, tampoco podemos nombrar los conectores.

En Visual Studio (Diagramas de Actividades):



En Visual Paradigm (Diagramas de Proceso de Negocio):



8 Tuplas DCU/DCL/DSQ/GUID/DST

Diagramas de Casos de Uso/Clases/Secuencias/Ventanas/Negocio

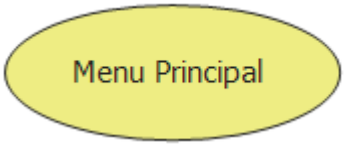
En este Apartado mostraremos dependiendo de cada caso una imagen de un caso de uso, con su clase relacionada, el diagrama de secuencias de dicha clase, la ventana de interfaz de usuario o vista y el diagrama de Proceso de Negocio. Esto nos hará ver de forma detallada la trazabilidad de cada uno de los caminos seguidos en el desarrollo de nuestro proyecto.

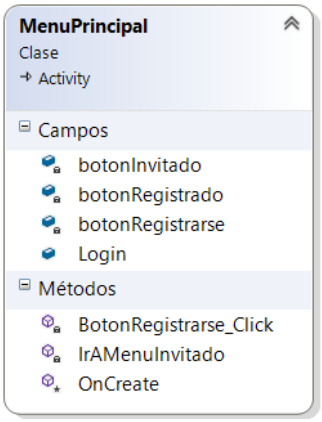
Habrá casos en los que no mostraremos Diagramas de Secuencias ya que sólo en los casos en los que haya intercambio de datos con fuentes externas (repositorio) serán necesarios para visualizar los métodos correspondientes. También habrá casos en los que nuestras Activities recibirán datos de otras Activities que han hecho llamadas al repositorio, estos datos serán pasados a estas Activities por serialización de objetos como antes hemos detallado.

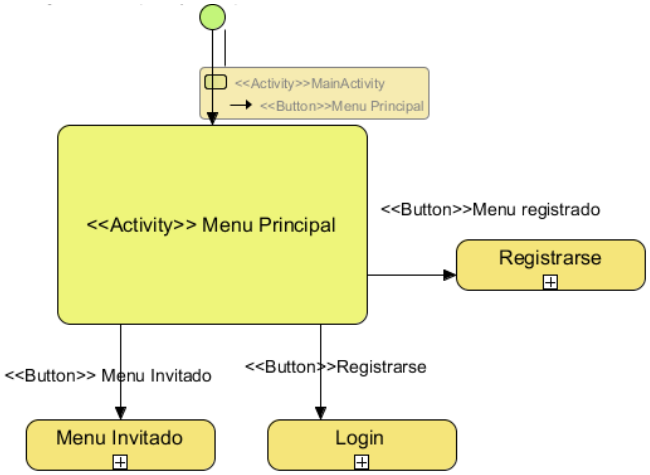
8.1 Bienvenida

En esta primera pantalla, vemos tan solo un botón que al pulsarlo nos lleva al **Menu Principal**. Al carecer aquí de llamadas a base de datos, prescindimos de diagrama de secuencias.

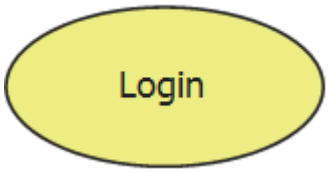
8.2 Menú Principal

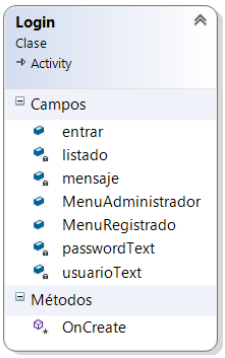


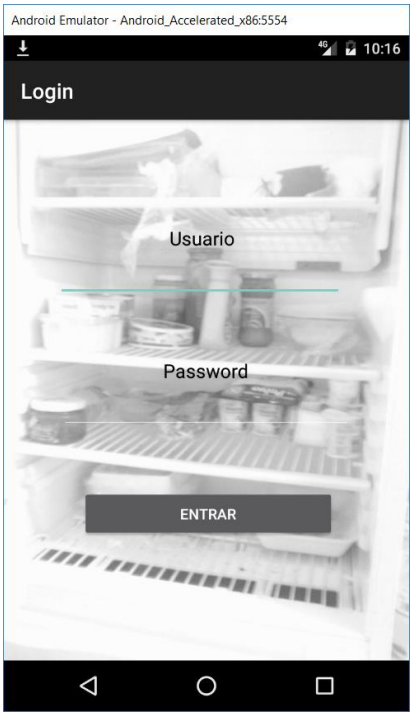


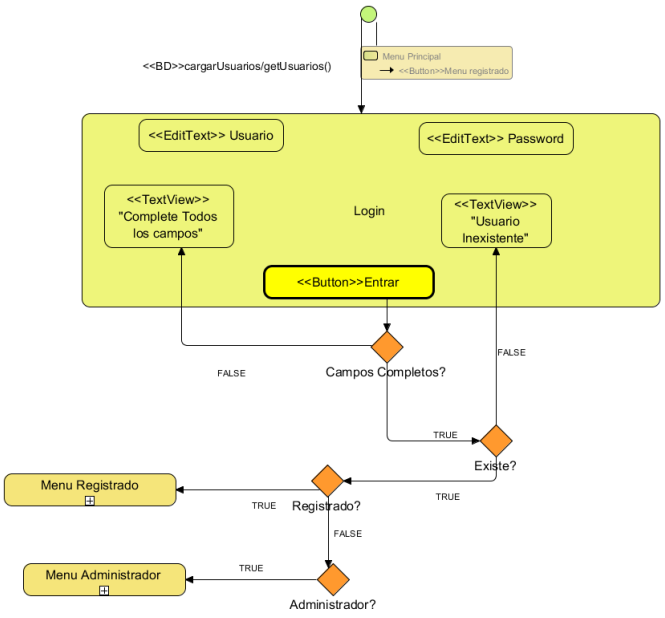


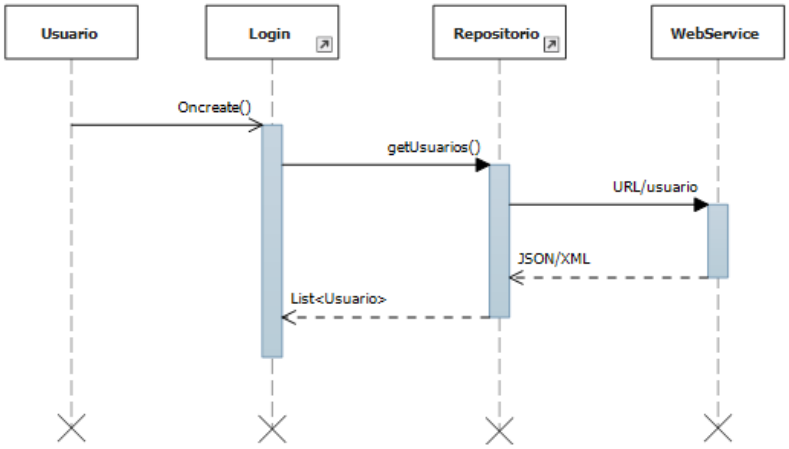
8.3 Login


Login

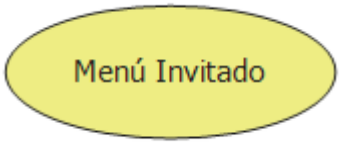

Login
Clase
Activity
Campos
entrar
listado
mensaje
MenuAdministrador
MenuRegistrado
passwordText
usuarioText
Métodos
OnCreate

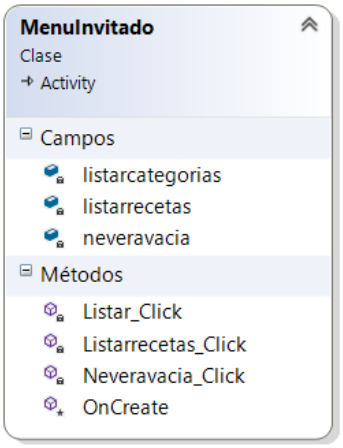

Android Emulator - Android_Accelerated_x86:5554
10:16
Login
Usuario
Password
ENTRAR

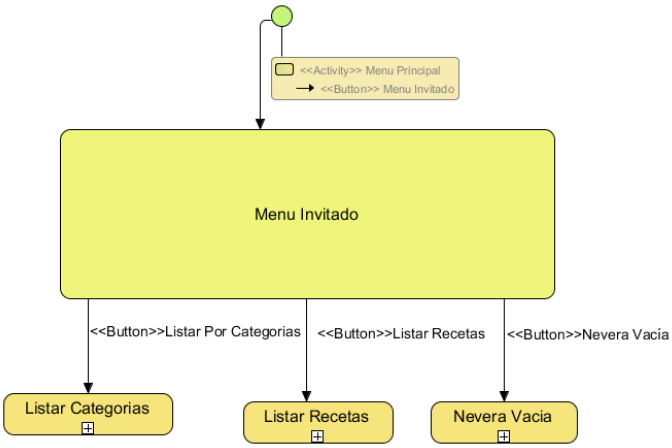

<<BD>>cargarUsuarios/getUsuarios()
Menu Principal
<<Button>>Menu registrado
<<EditText>> Usuario
<<EditText>> Password
<<TextView>>
"Complete Todos
los campos"
Login
<<TextView>>
"Usuario
Inexistente"
<<Button>>Entrar
FALSE
Campos Completos?
FALSE
TRUE
Existe?
Menu Registrado
TRUE
Registrado?
TRUE
FALSE
Menu Administrador
TRUE
Administrador?


Usuario
Login
Repositorio
WebService
Oncreate()
getUsuarios()
URL/usuario
JSON/XML
List<Usuario>

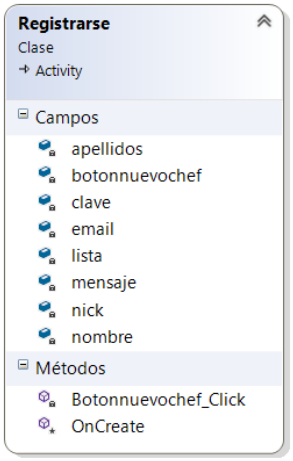
8.4 Menu Invitado

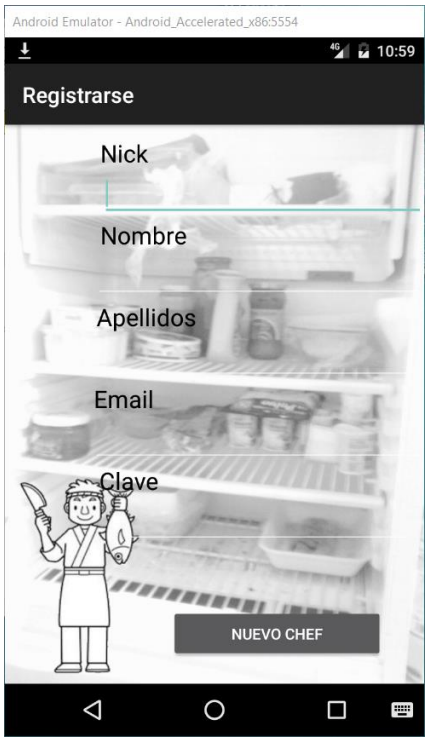


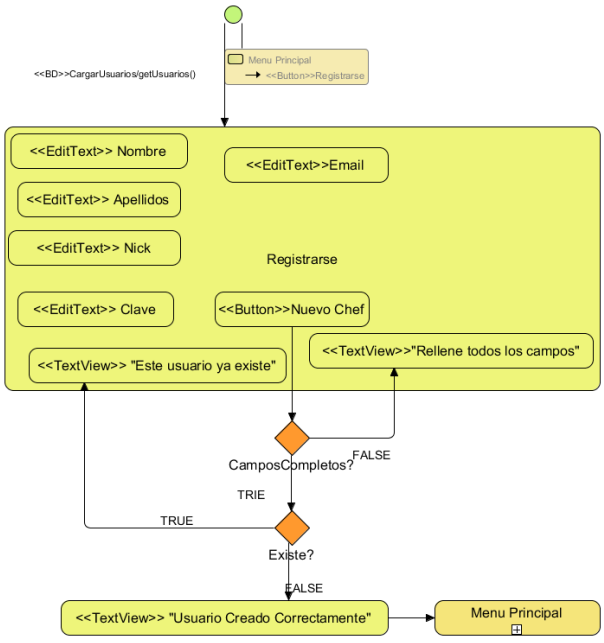


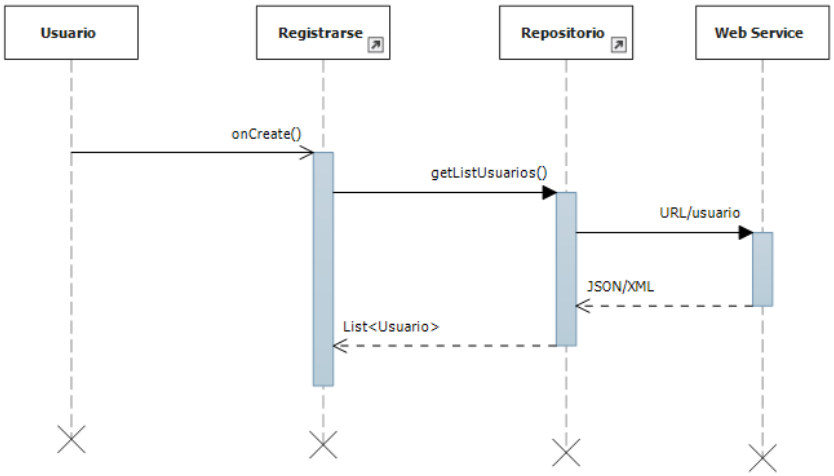


8.5 Registrarse

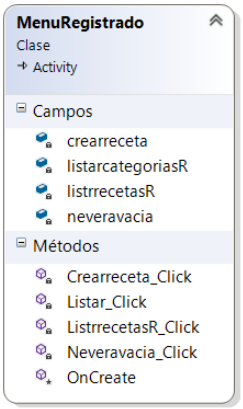


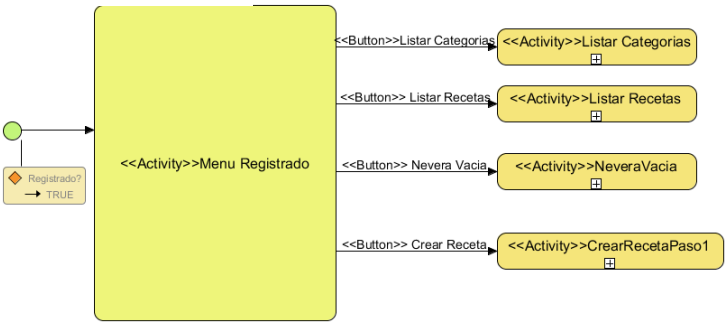




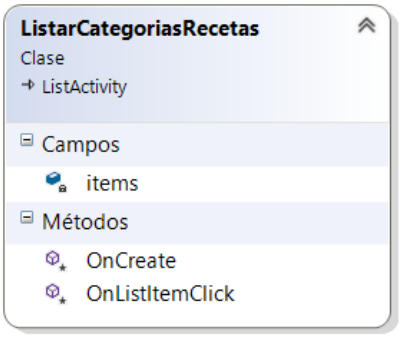


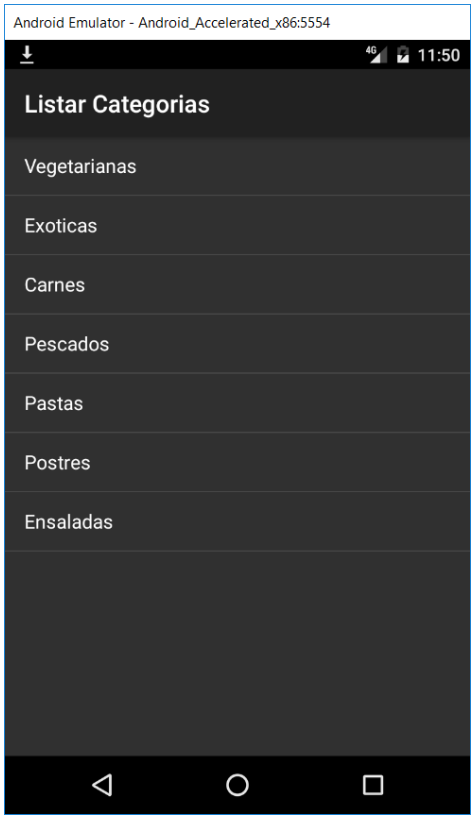
8.6 Menú Registrado

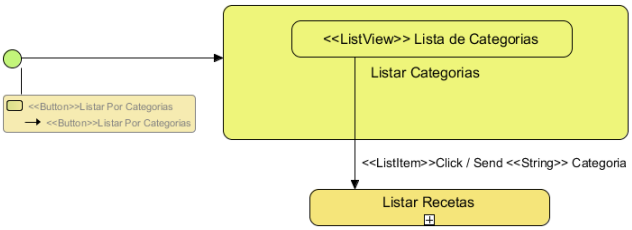




8.7 Listar Categorías Recetas

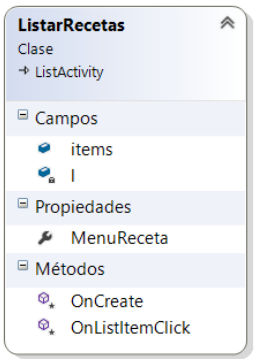


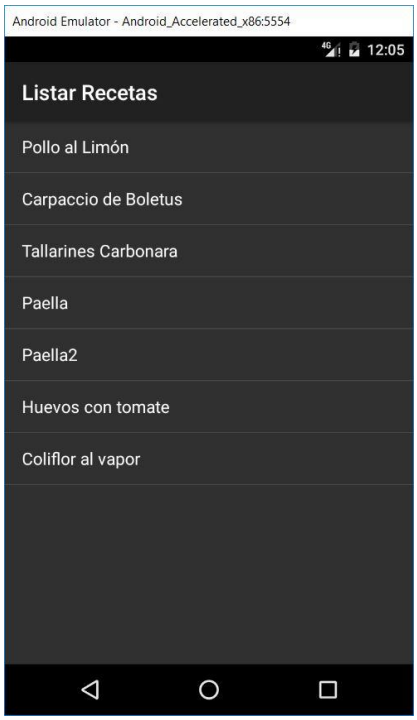


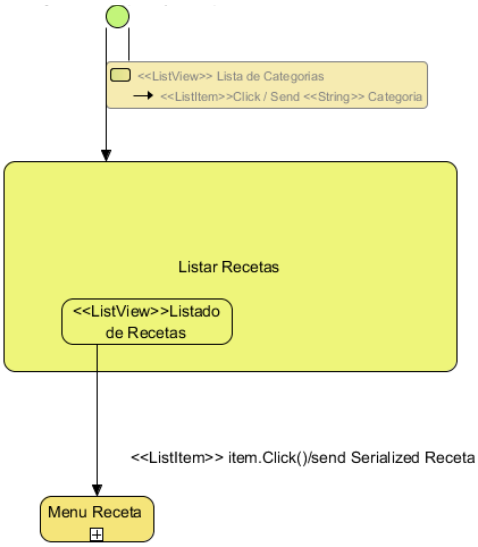


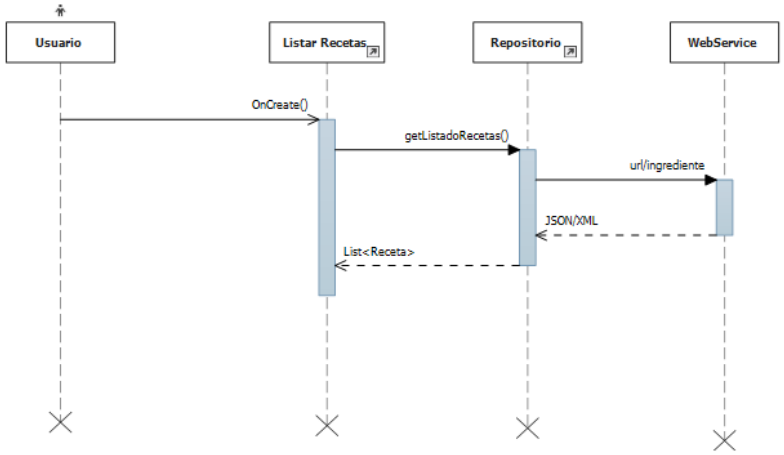
Esta ListActivity muestra un listado prefijado de las posibles categorías que existen. Al pulsar en un ítem de la lista, nos manda a la ListActivity ListarRecetas y envía a esta tarea un String por JSON como dato. ListarRecetas se encagará de listar las recetas oportunas filtrando por la categoría elegida.

8.8 Listar Recetas



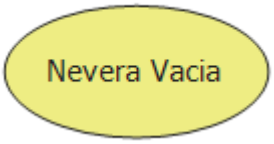


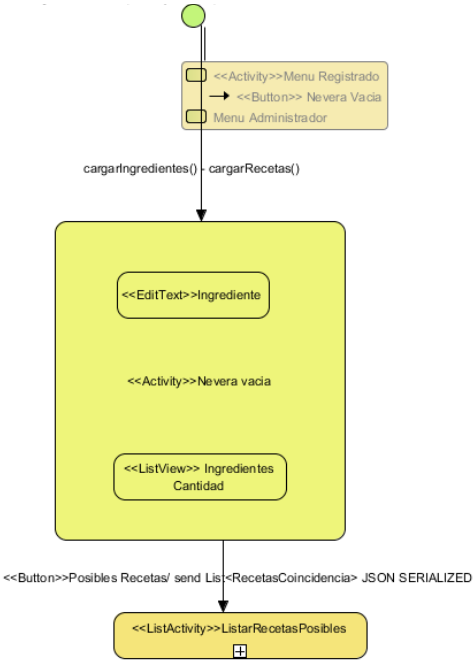


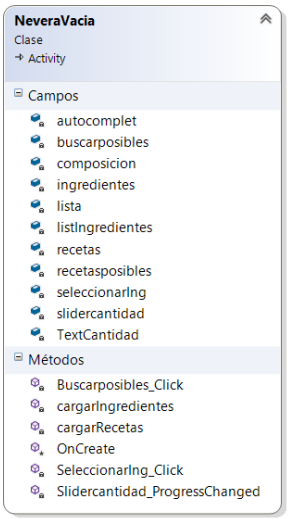


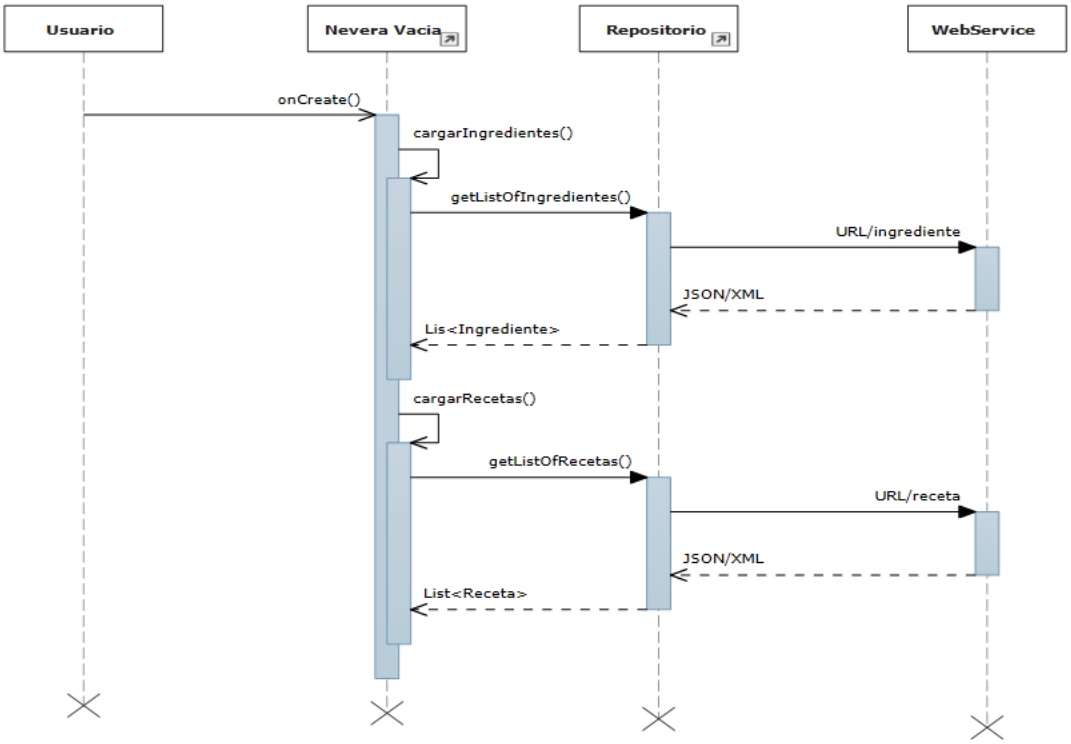
La ListActivity ListarTareas se encarga de visualizar una lista de Recetas. Para ello llama al método getListadoRecetas() del repositorio y así recibe en formato List<Receta> todas ellas. Cuando el Usuario selecciona un ítem de la lista pasamos al MenuReceta el cual recibe de la presente Actividad el objecto Receta serializado en JSON. Posteriormente en destino se des-serializar.

8.9 Nevera Vacía

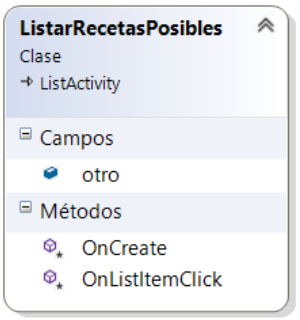


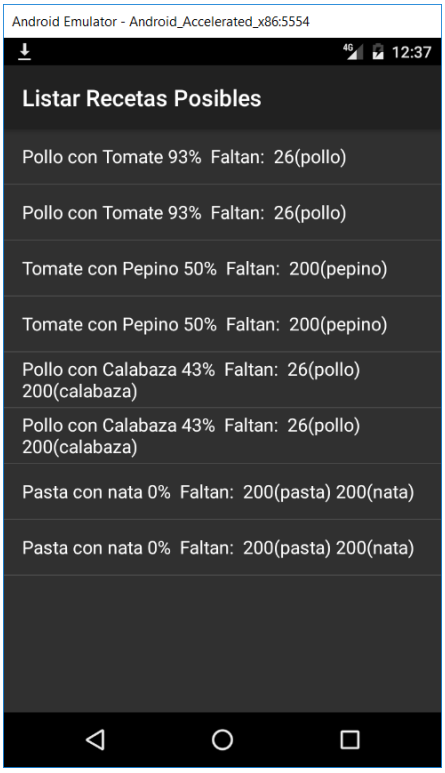


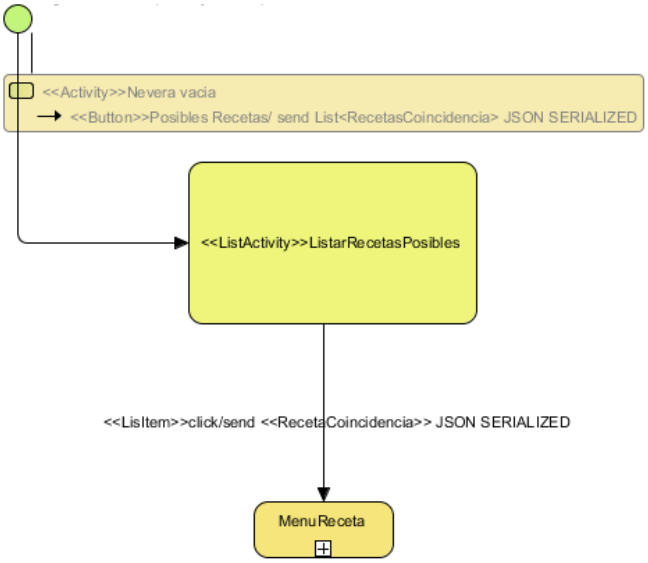




8.10Listar Recetas Posibles

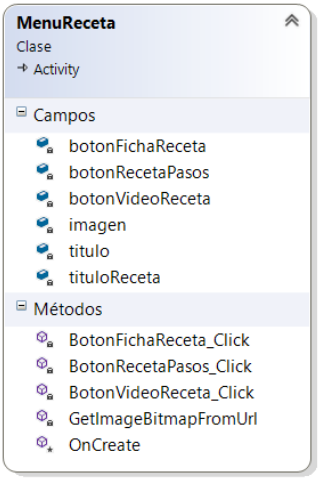


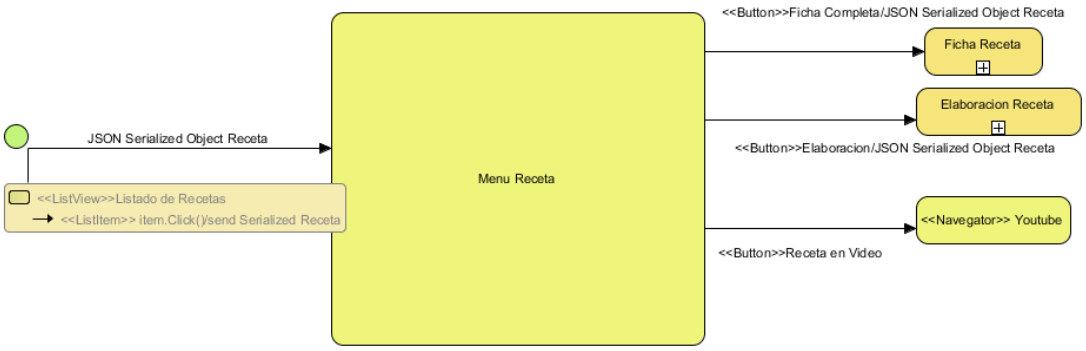




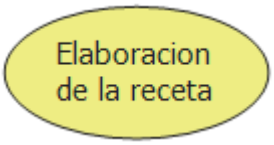
Aquí, el usuario visualiza el resultado que nos da la aplicación Nevera vacía, un listado de recetas con un índice % de coincidencia respecto de los ingredientes que el usuario dispone y en su caso la cantidad de cada ingrediente faltante.

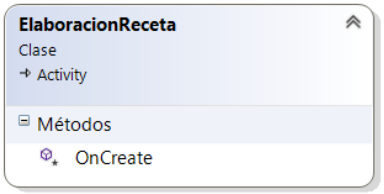
8.11Menú Receta

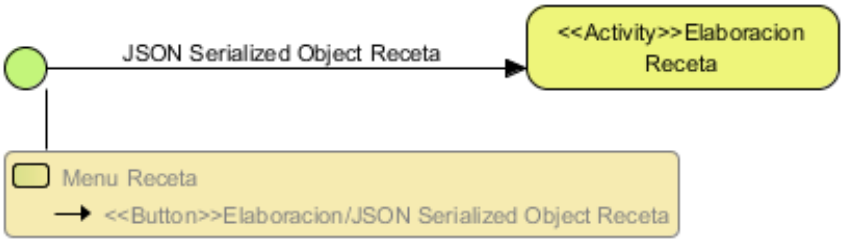




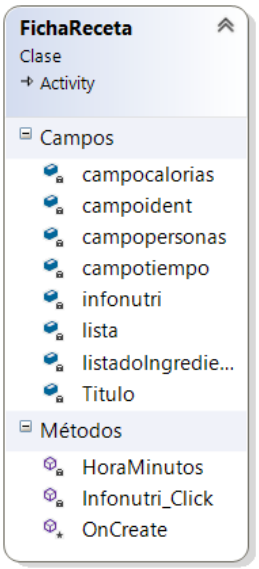
8.12 Elaboración Receta

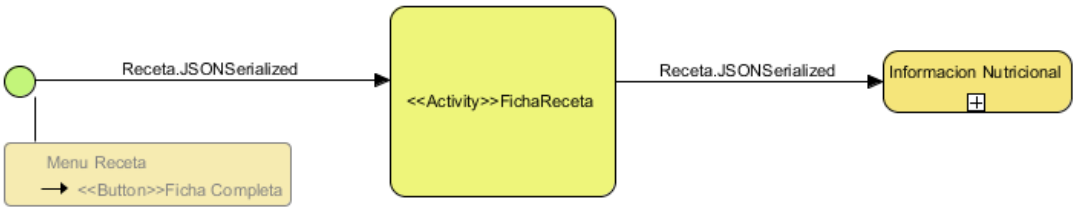




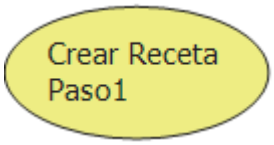


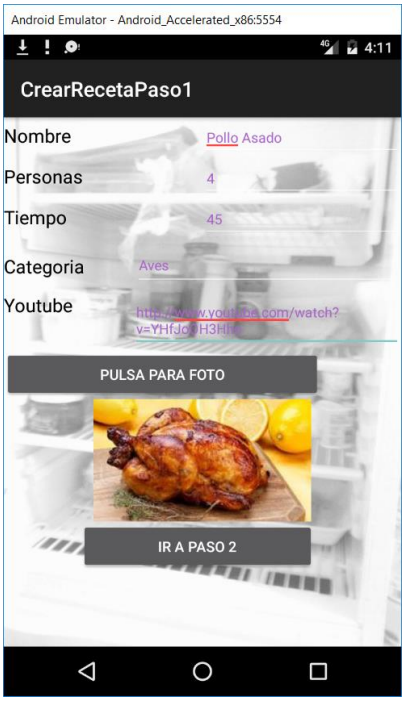
8.13Ficha Receta

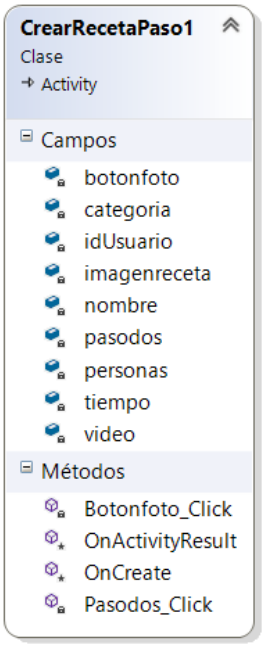


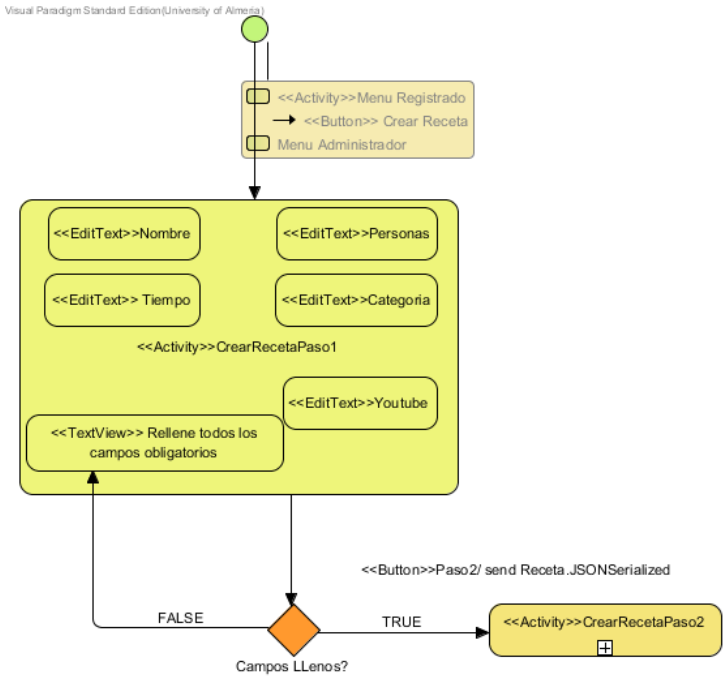


8.14Crear Receta Paso1

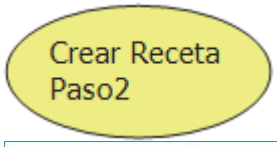


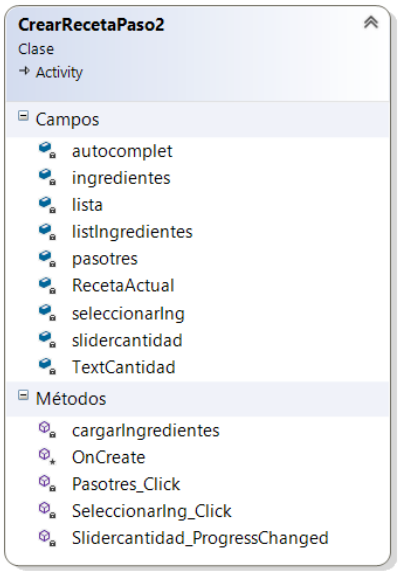


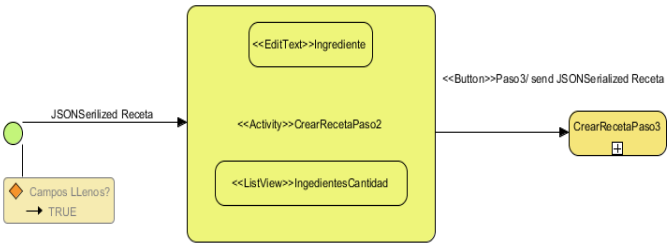


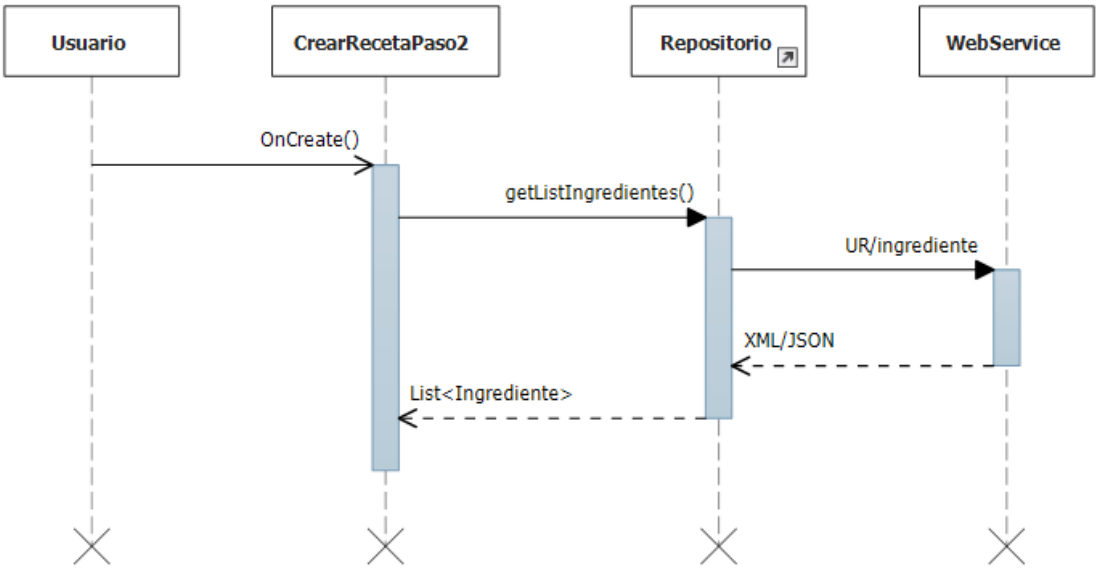


8.15Crear Receta Paso2

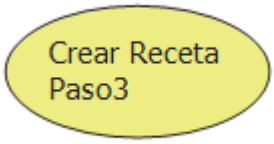


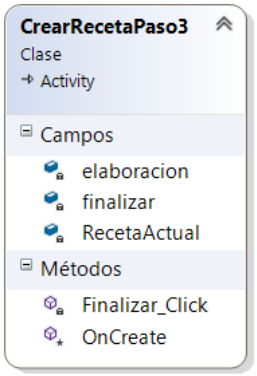


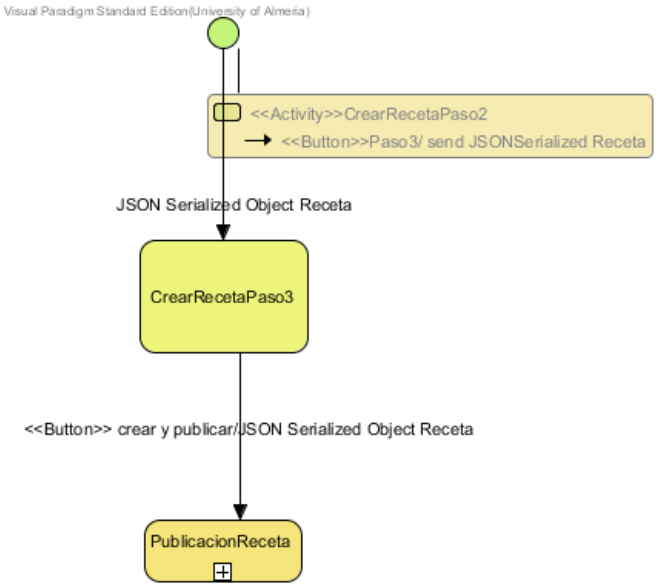




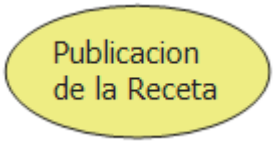
8.16Crear Receta Paso3

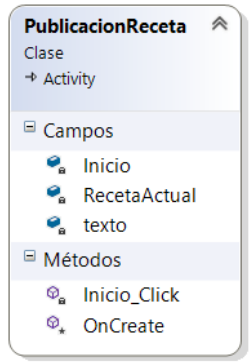


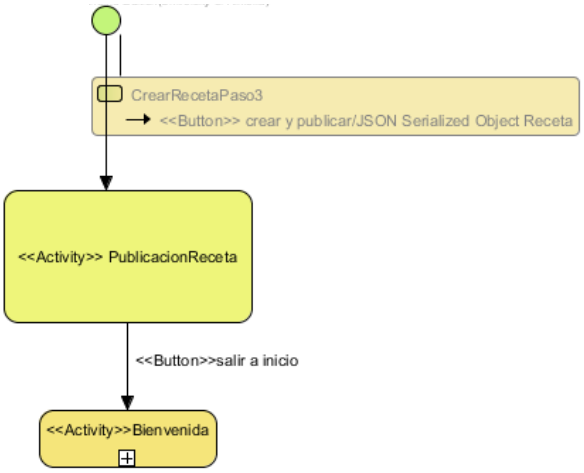


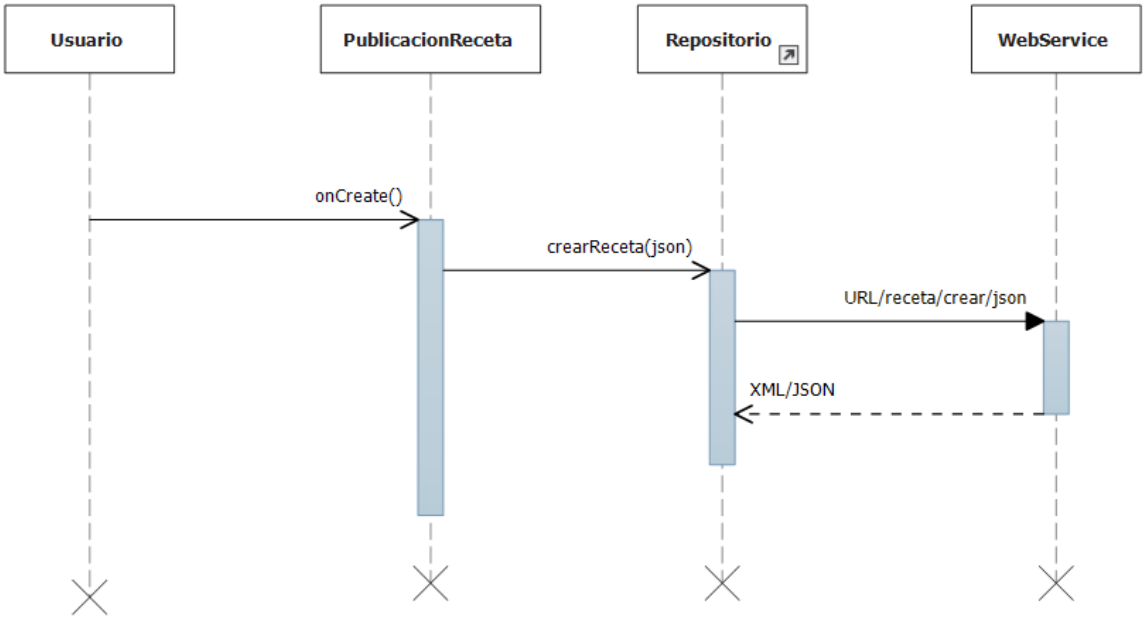


8.17Publicación Receta

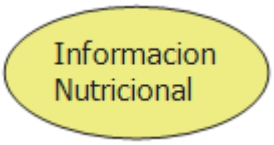


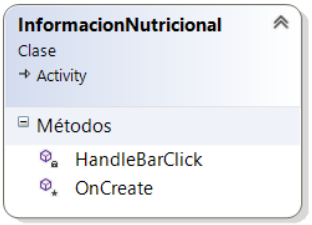


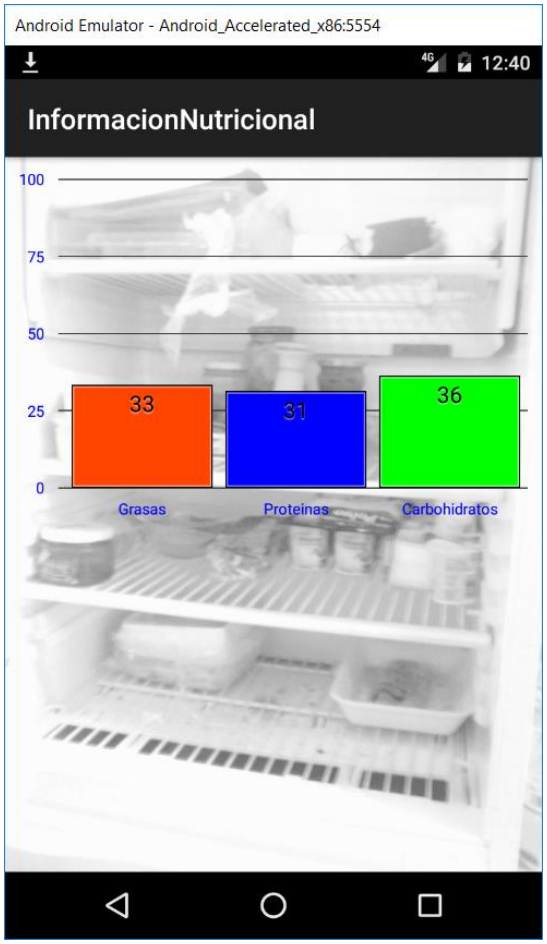


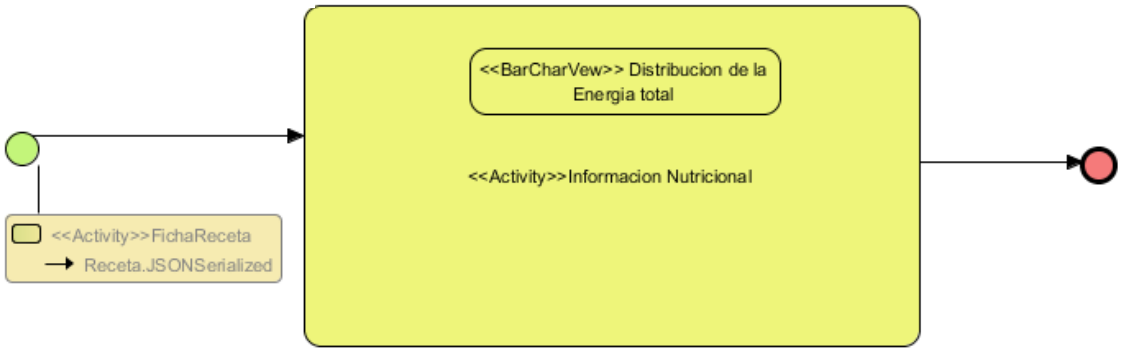


8.18Información Nutricional

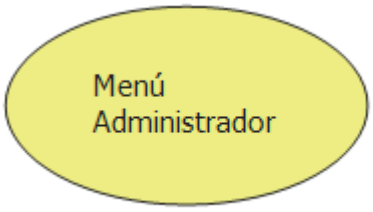


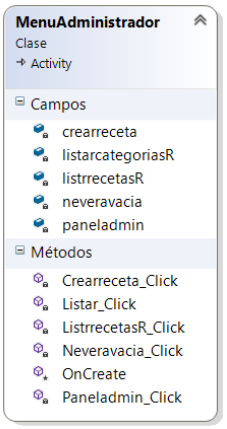


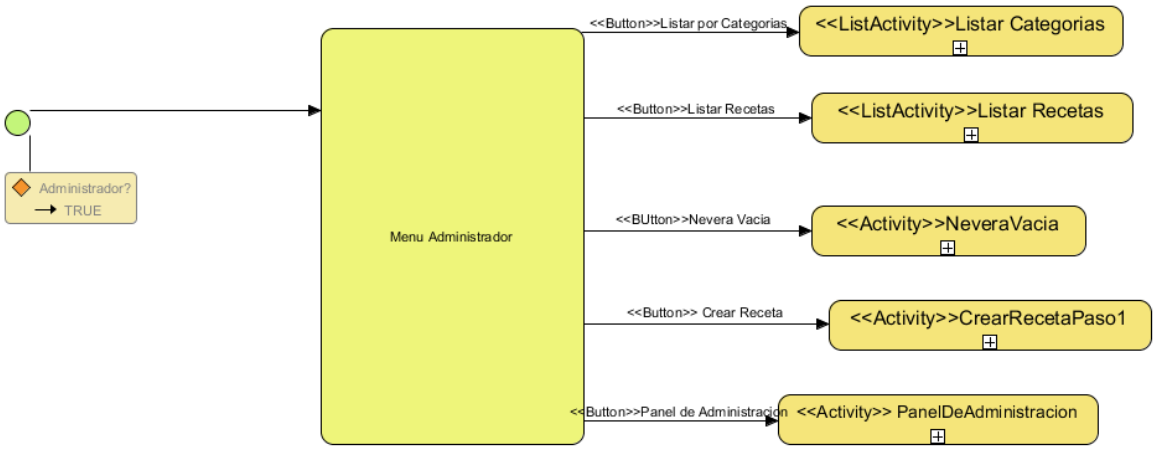




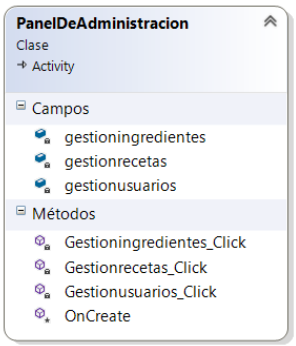
8.19 Menú Administrador

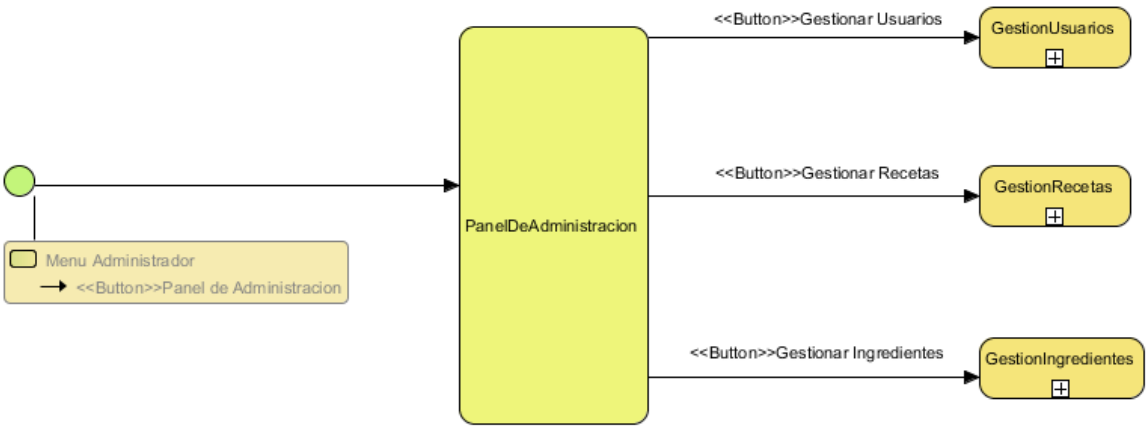




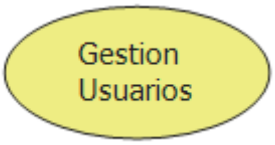


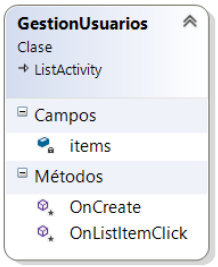
8.20 Panel de Administración

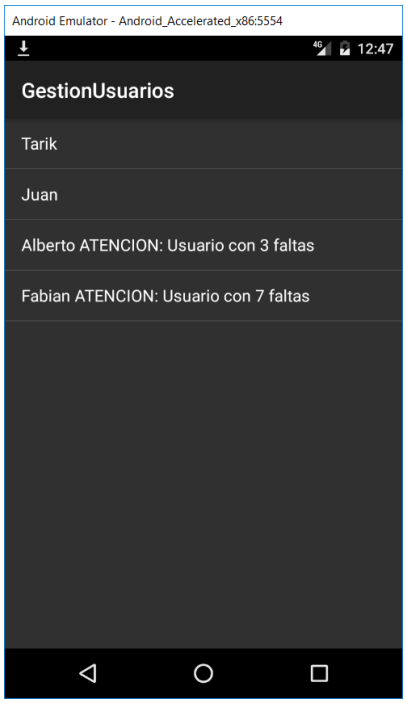


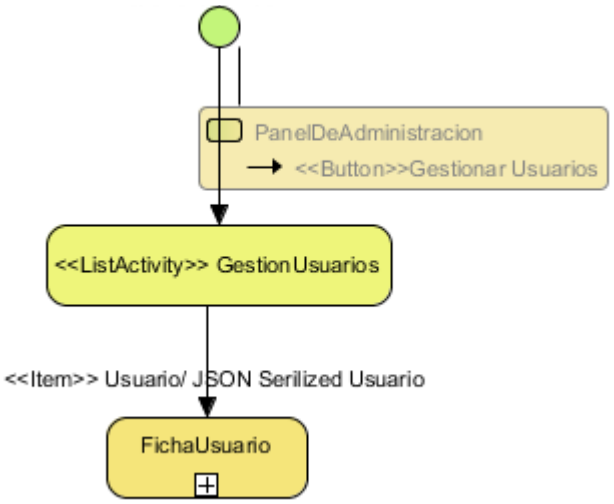


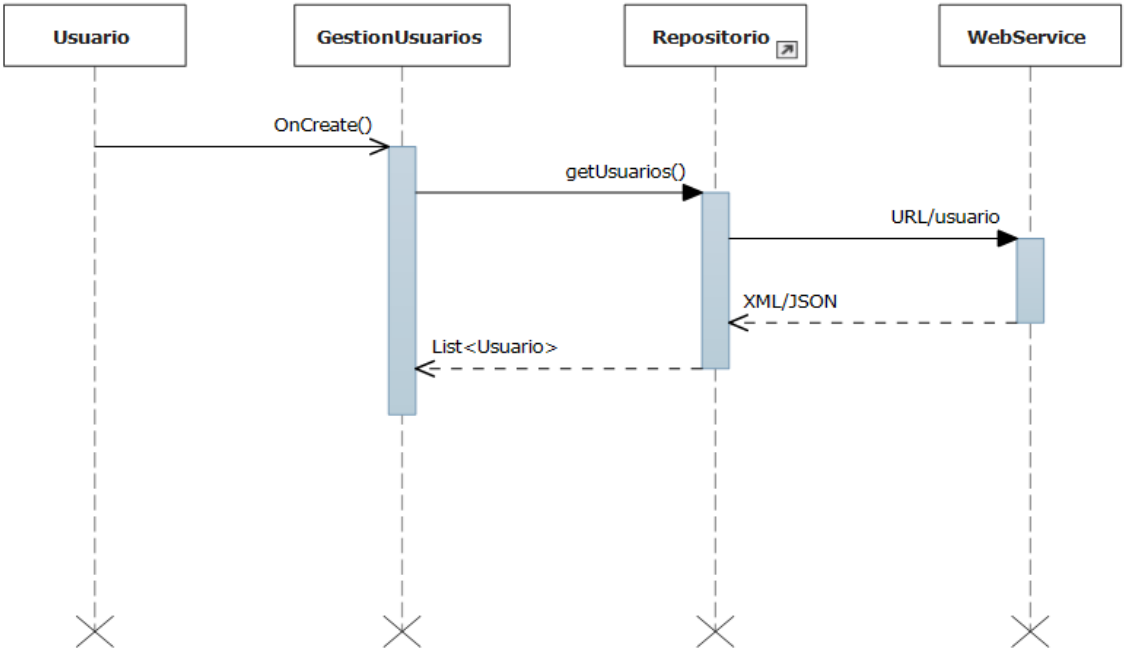
8.21Gestión Usuarios



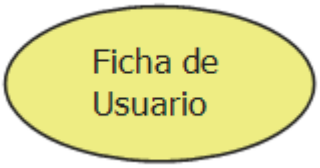


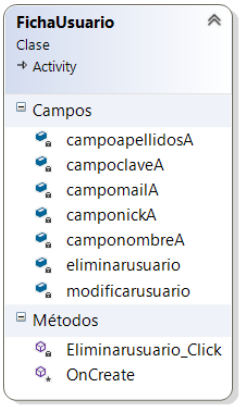


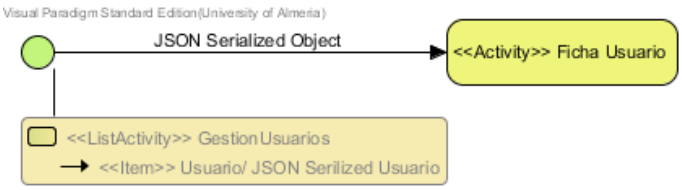


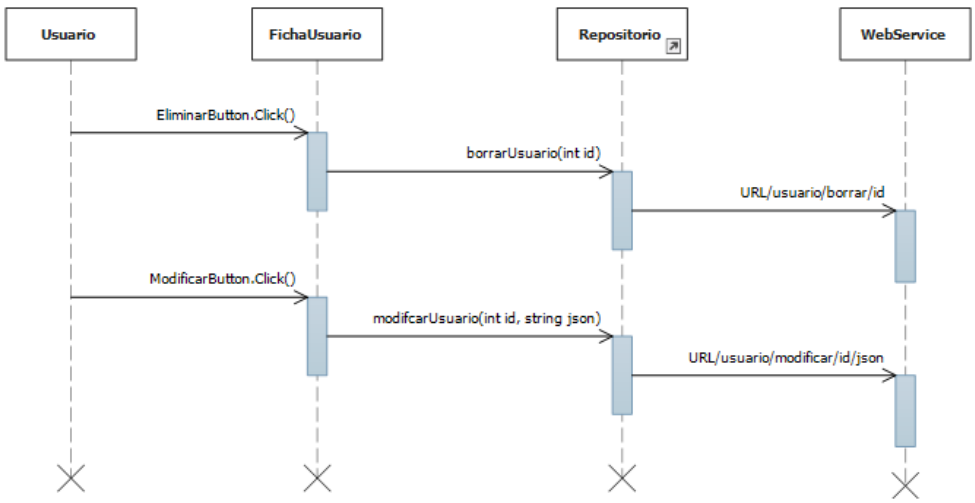


8.22 Ficha Usuario

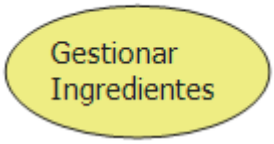


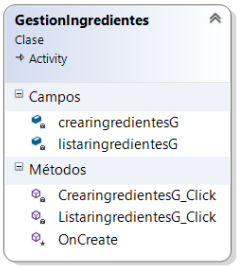


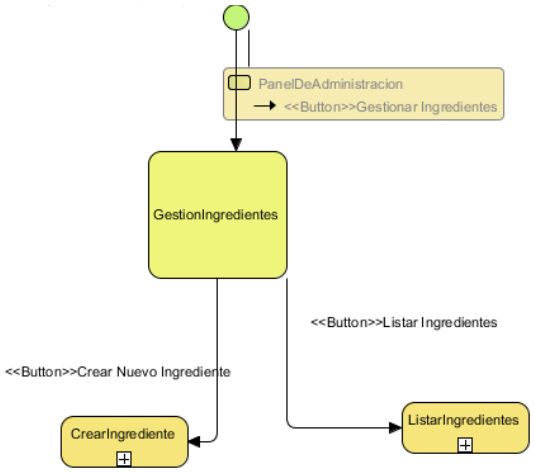




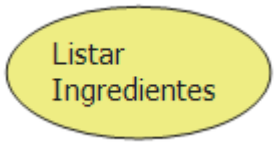
8.23Gestión Ingredientes

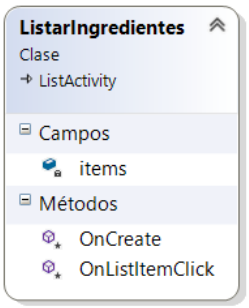


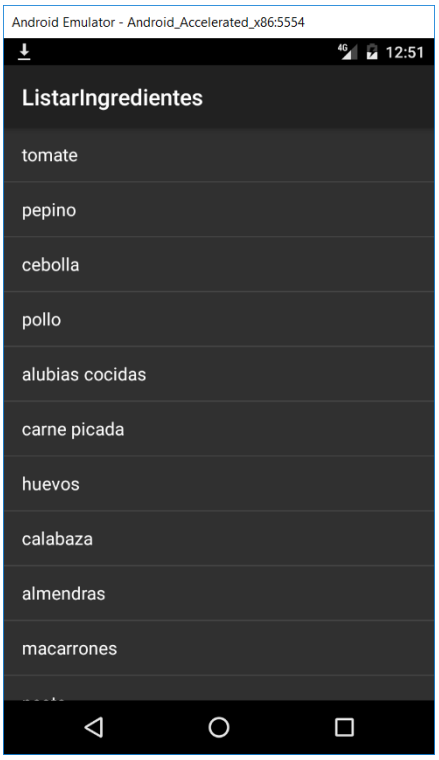


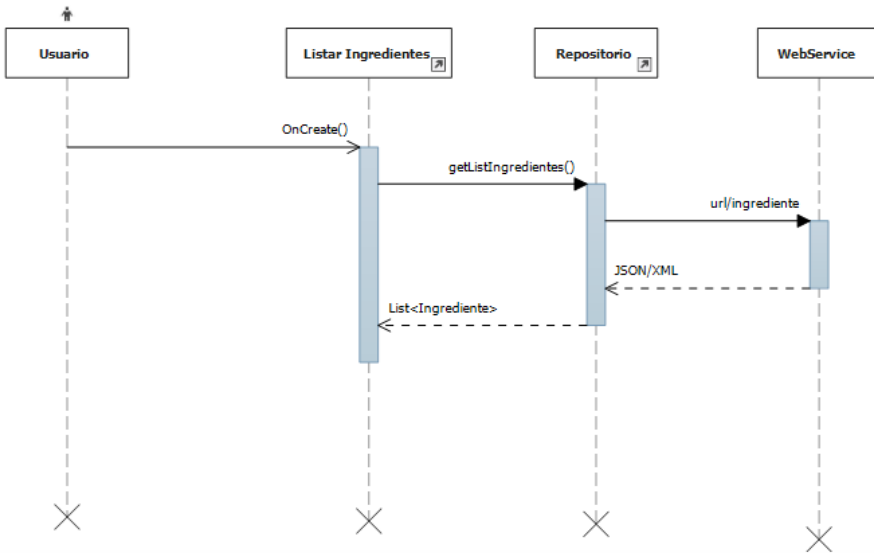


8.24Listar Ingredientes

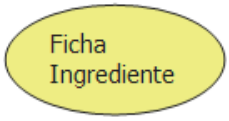


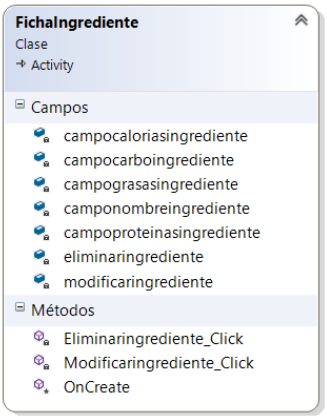


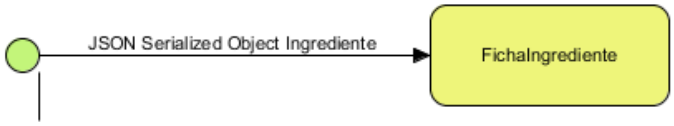


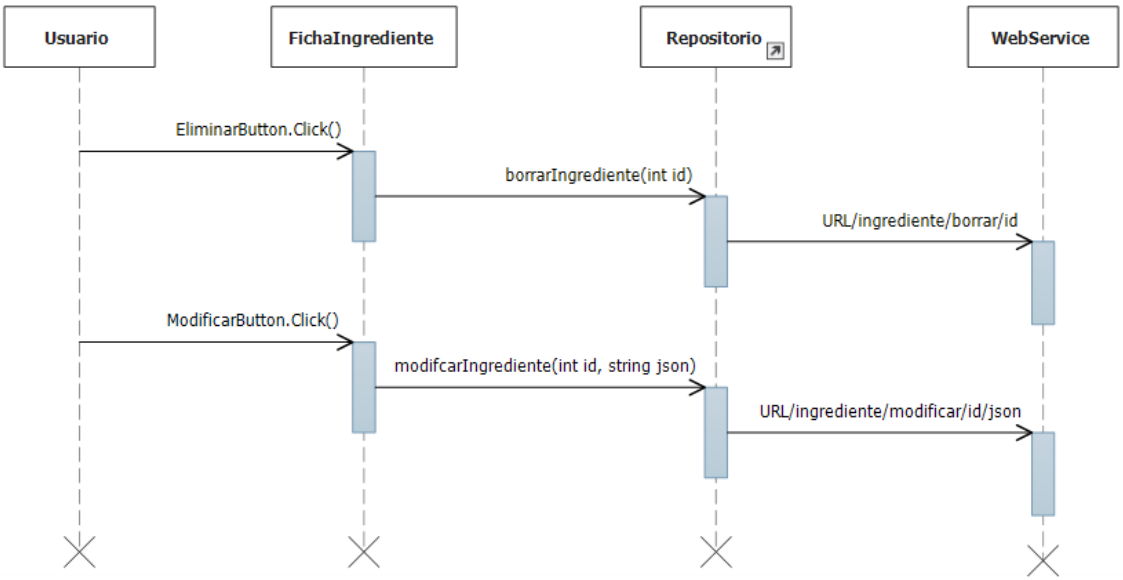


8.25Ficha Ingrediente

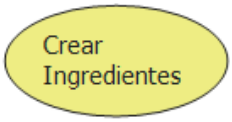


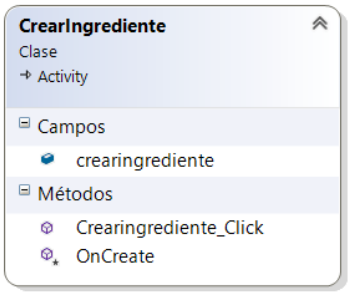


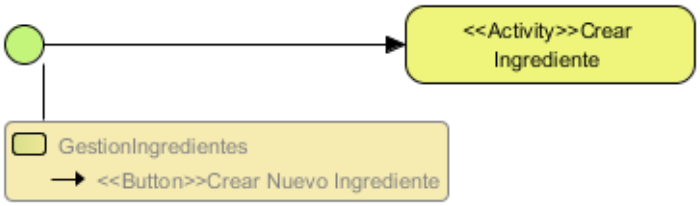


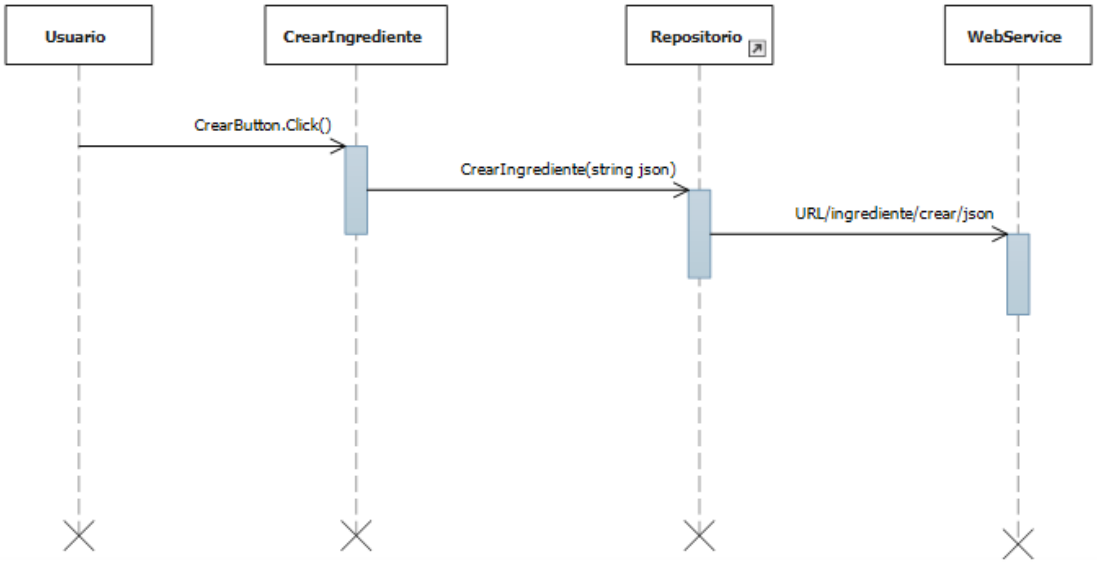


8.26Crear Ingrediente

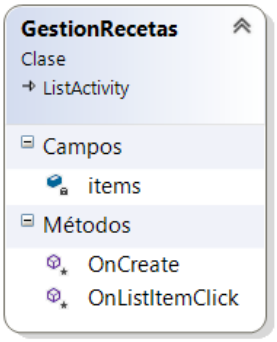


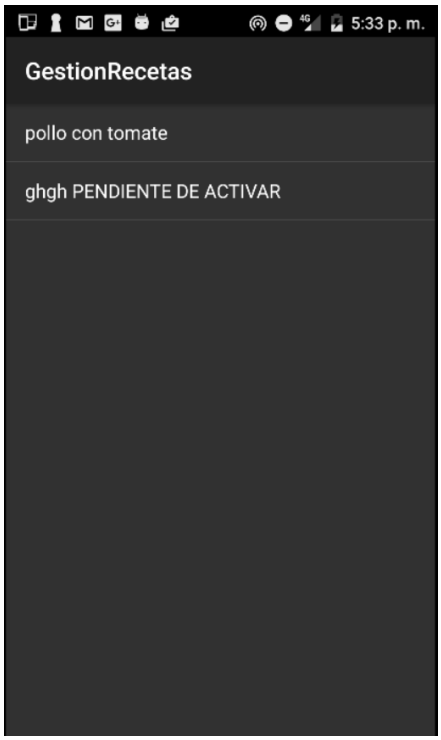


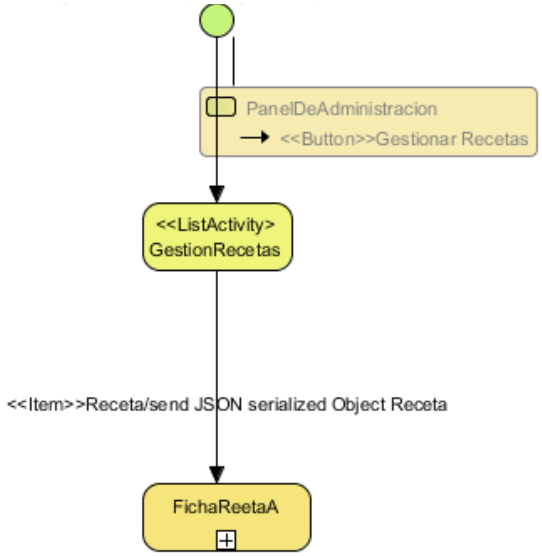


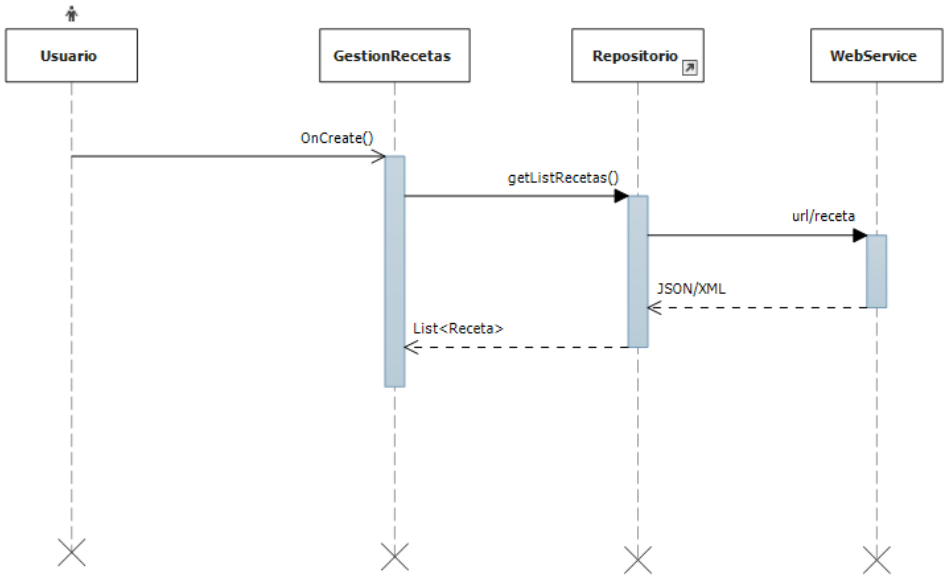


8.27Gestión Recetas

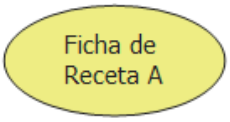


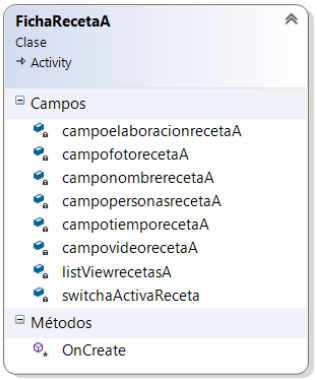


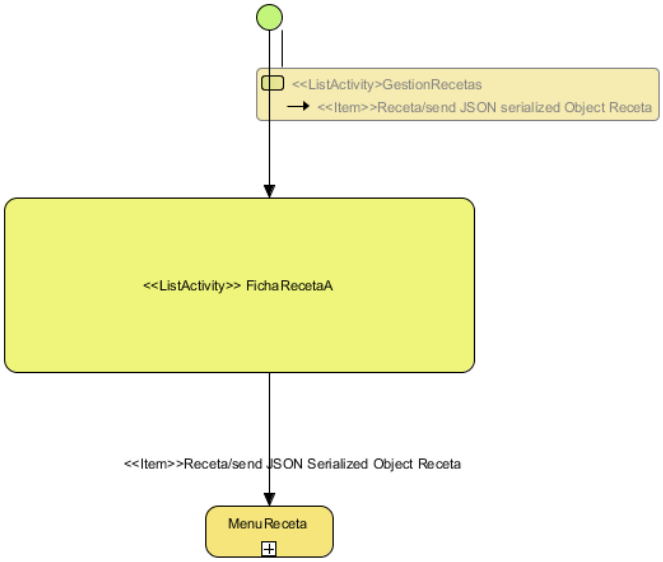


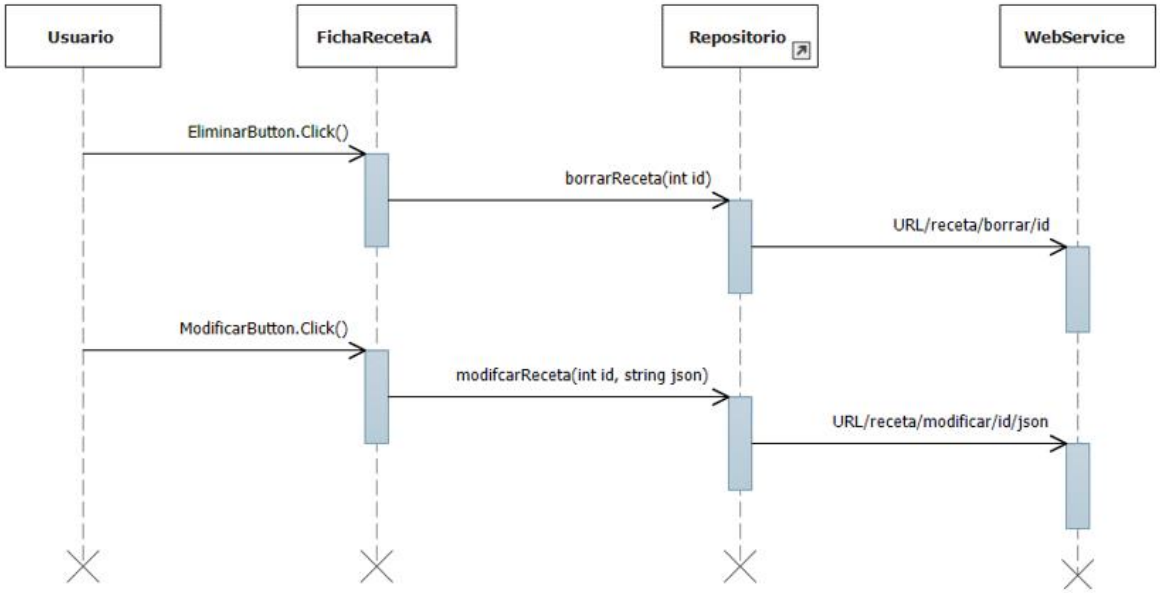


8.28Ficha Receta A

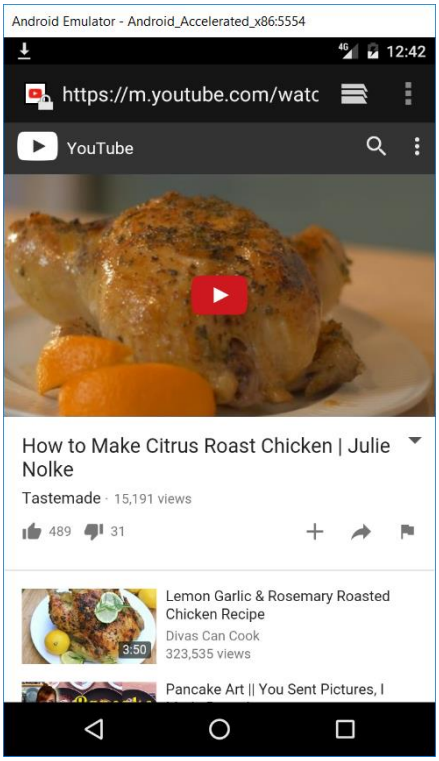








8.29Receta En Video



9 Conclusiones y Líneas Futuras

9.1 Conclusiones

Una vez finalizado este proyecto, las conclusiones a las que hemos llegado a lo largo de todo el desarrollo son varias y las encuadraremos en dos categorías

9.1.1 A nivel de Modelado

Dentro de este apartado hay varios aspectos a mencionar ya que, aun teniendo la Metodología de Modelado a aplicar suficientemente clara, el terreno para aplicarla era inexplorado. En ningún foro hemos encontrado información relativa al modelado de aplicaciones móviles propio de Visual Studio y siendo realistas, digamos que, aunque satisfactorio haya sido el resultado para nosotros, aún dista bastante de lo que esperábamos obtener. Sin embargo, lo bueno del modelado es que da la libertad de poder usar diferentes herramientas y poder conjugarlas para lograr el objetivo, sin embargo, aquí el problema real ha sido el desacople de las herramientas, la no posible integración entre ellas hace que no podamos decir que haya un puente entre una fase con la siguiente, entre un diagrama y otro, entre el comienzo y el fin.

En definitiva, la conclusión a la que hemos llegado es que es imperativo para que se dé como resultado un buen proyecto de modelado que exista una trazabilidad de principio a fin y que no dependa sólo de documentar esa trazabilidad como he hecho en algunos puntos del proyecto, sino que las herramientas ayuden a ello. Prueba clara de que es posible es que aplicando esta misma metodología en entornos de Eclipse con Visual Paradigm para proyecto de escritorio en Java se llega a un resultado sobresaliente.

Conclusión: Hay que seguir investigando y apostando por la integración a todos los niveles.

9.1.2 A nivel de Implementación

La tarea más dura después del modelado ha sido sin duda la implementación. No habiendo nunca desarrollado una aplicación móvil en una plataforma nativa, no se podía comparar si era peor o mejor. Tan solo podemos hablar de lo difícil que ha sido desarrollar en esta plataforma híbrida, sobre todo al comienzo y antes de saber de la existencia de la depuración en dispositivos móviles. Donde realmente tienen una desventaja Xamarin es justamente en lo que debería ir más fluido, Compilación y Depuración. También hemos de añadir que dependiendo de la configuración que se le haya dado a nuestro entorno funciona mejor o peor, por lo que se deduce que todo el tiempo que hemos perdido con problemas varios, podríamos habérnoslo ahorrado con una buena configuración de nuestro entorno. Sin embargo, en busca de si era o no posible mejorar la configuración, estimábamos que nos iba a costar más tiempo y posiblemente la pérdida de la configuración a la que estábamos ya acostumbrados.

A parte de nuestros posibles propios fallos, existen realmente problemas que vienen de casa con nuevas actualizaciones que no van del todo bien y que también retrasan en gran medida el desarrollo. En definitiva, a este y otros niveles, hemos experimentado de primera mano las desventajas que supone trabajar en el desarrollo de aplicaciones híbridas y estamos de acuerdo con el material aportado por la empresa Inteligenz que mostramos en el apartado 3 de este proyecto, en el que se hace las comparativas de las aplicaciones Híbridas y Nativas.

10 Líneas Futuras

Las posibles líneas futuras de mejora de mi proyecto las dividiré igual que antes en dos categorías, a nivel de modelado y a nivel de implementación. La diferencia entre una y otra sería en que a nivel de modelado interesaría investigar de qué manera podría mejorarse la adaptación de la metodología dada usando las mismas herramientas o diferentes para el objetivo de modelar una aplicación móvil. A nivel de Implementación y servicios incluiría las posibles líneas de productos que se podrían llevar a cabo en un futuro, o la mejora de los presentes y/o el estudio de posibilidades de que esta aplicación pasara de ser un mero proyecto a nivel académico a una aplicación que se pudiese lanzar al público y en su caso poder venderse o usarla lucrativamente.

10.1 A nivel de Modelado

- **Investigar cómo acoplar un proyecto de modelado de Visual Studio a un proyecto de Android:**
  Leyendo por encima en material de Microsoft, parece ser que la línea de posible investigación estaría orientada a traducir clases estereotipadas como Activities en el diagrama de clases a clases propias de un proyecto Android. Hasta donde hemos llegado se puede hacer desde clases con estereotipos clásicos de C#.
- **Investigar cómo acoplar un proyecto de modelado de Visual Paradigm a un proyecto de Android:**
  Lo mismo que en el punto anterior, sin embargo, con Visual Paradigm. Un problema con mucha similitud
- **Investigar las posibles mejoras en el modelado de clases:**
  A lo largo del proyecto hemos echado de menos recursos clásicos del modelado tales como la Herencia entre clases. Investigar si es posible o no, y si lo es, ver si nos beneficia finalmente.
- **Usar metodologías ágiles de Gestión de Proyectos a la Metodología de modelado:**
  En mi particularidad modo de ver, la gestión de proyectos llevada a cabo en la Metodología de modelado y desarrollo del software en este proyecto, podría mejorarse siguiendo patrones de metodologías ágiles como SCRUM u otra variación acorde.

10.2 A nivel de Implementación y Servicios

- **Añadir la visualización de los componentes nutricionales completos de cada Ingrediente:**
  Supondría añadir a la clase Ingrediente más atributos como, por ejemplo, la composición en Hierro, Magnesio, Calcio, Ioduro, Selenio, Fósforo, Sodio, etc. Para tener una visión mucho más completa a cerca de la receta en el punto de vista nutricional.
- **Añadir la visualización de los componentes nutricionales completos de cada Receta:**
  A su vez, los datos anteriormente citados podríamos mostrarlos a partir de tablas o gráficos tal que hicimos anteriormente con las proteínas, grasas y carbohidratos. Una gráfica extra.
- **Intentar hacer Scraping a listas de ingredientes y sus atributos directamente a través de internet:**
  Quizás una de las cosas que hay que mejorar es el automatizar la inserción de datos básicos, como los ingredientes que pueden conseguirse de tablas en páginas web de internet para poder actualizar nuestra base de datos. Quedaría pendiente estudiar cómo hacer Scraping desde C#.
- **Hacer búsqueda de ingredientes o productos directamente de internet y traernos los precios y en qué Supermercados están disponibles**:
  El Scraping nos serviría también para tal propósito. Podría ser útil para los clientes saber por ejemplo dónde comprar los productos que le faltan para elaborar una receta y comparar precios entre distintas tiendas y supermercados.
- **Intentar adaptar nuestro modelado al desarrollo de aplicaciones en Windows Phone e iOS:**
  Xamarin nos da la oportunidad de aparte de crear aplicaciones en Android, hacerlo simultáneamente para Windows Phone e iOS. Para iOS se necesitaría una máquina de Apple conectada a nuestra red para poder conectar Xamarin al entorno ce desarrollo de XCode.
- **Crear búsquedas filtradas de recetas:**
  Aquí podríamos añadirle más opciones al usuario para que pudiese elegir recetas que tuviesen o no algún compuesto. Por ejemplo, para que los hipertensos pudiesen prescindir de recetas que tuviesen un alto índice de Sodio, elegir recetas ricas más en proteínas que en grasas o carbohidratos.
- **Publicar la aplicación en GooglePlay:**
  Esta sería quizás la línea futura más alejada de nosotros en el tiempo puesto que para ello se debería pasar de una aplicación de prueba como la que estoy mostrando a una aplicación profesional y requeriría además de un replanteamiento de análisis de requisitos, más desarrolladores para acelerar el desarrollo, establecer metodologías de trabajo ágiles, en definitiva, no hacer una aplicación de aficionados, una aplicación robusta y preparada para ser modular, actualizable, etc.

En definitiva, hemos listado solo unas posibles líneas que se podrían desarrollar en un futuro, sin embargo, el hecho de haber aprendido esta metodología junto con este entorno nos ha abierto la puerta a una inmensidad de otras posibles opciones, ya que, aparte de lo mostrado en este documento, existen otras particularidades de las herramientas de Xamarin que ofrecen una alta gama aplicativa.

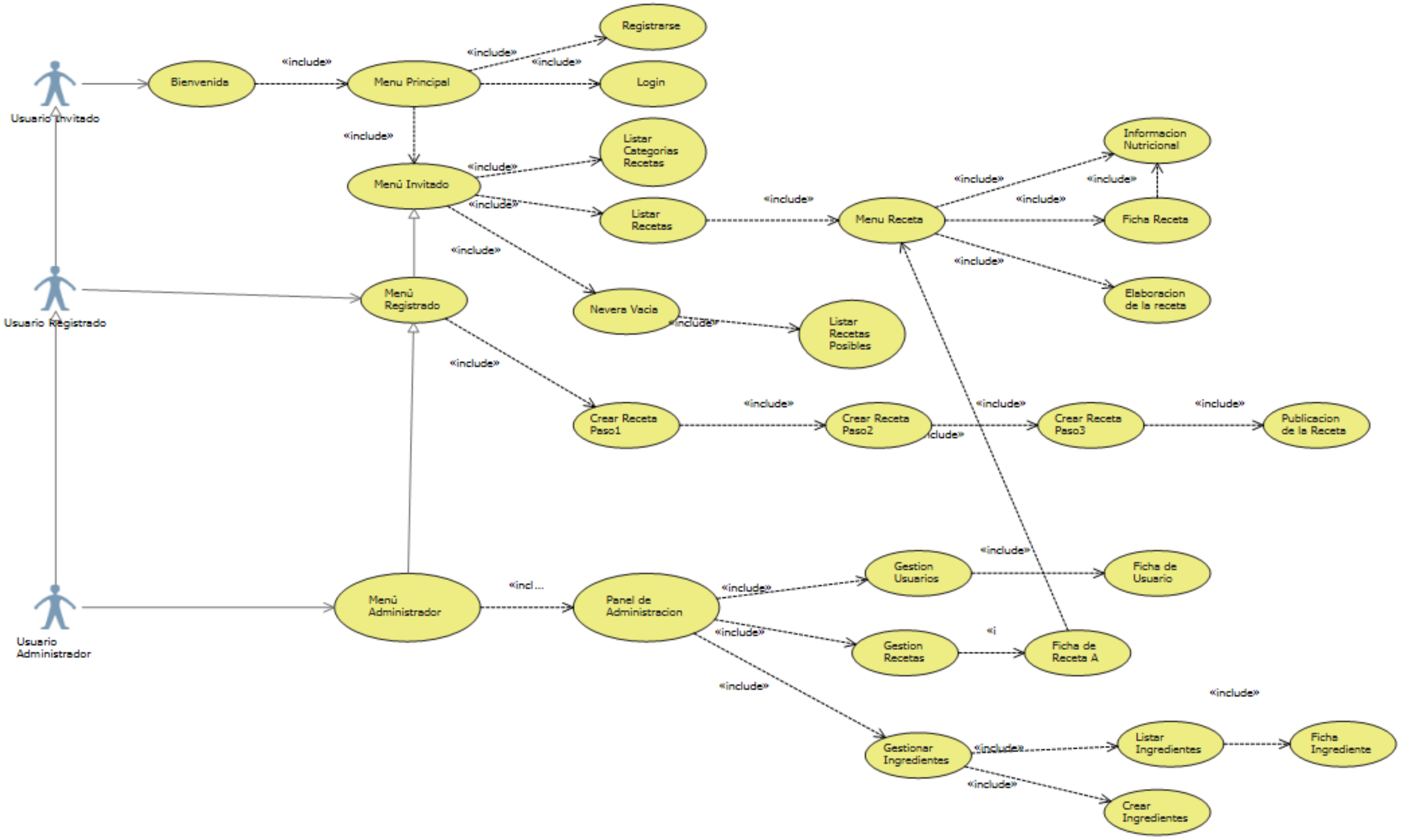
11 Bibliografía

- http://indalog.ual.es/WWW/compjournaldbipress.pdf
- http://indalog.ual.es/mdd/udbi/
- https://developer.xamarin.com/guides/cross-platform/application_fundamentals/web_services/
- https://forums.xamarin.com/
- https://developer.xamarin.com/guides/
- https://msdn.microsoft.com/en-us/library/dd409445.aspx
- http://www.intelygenz.es/
- **Scraping en C#:** https://www.youtube.com/watch?v=4cPPD-MFadQ
- **Web Services Xamarin de Juan Carlos Zuluaga:**
  - *Parte1:* https://www.youtube.com/watch?v=hAPoKgZA5Rw
  - *Parte2*: https://www.youtube.com/watch?v=Ic_XTcIO3SQ
  - *Parte3:* https://www.youtube.com/watch?v=Bq-of7cLQOs
  - *Parte4:* https://www.youtube.com/watch?v=vTYG8z81kqM
- **Partes 1 a 11 de videos enlazados de Juan Carlos Zuluaga:** https://www.youtube.com/watch?v=OgW_6HO6whI4

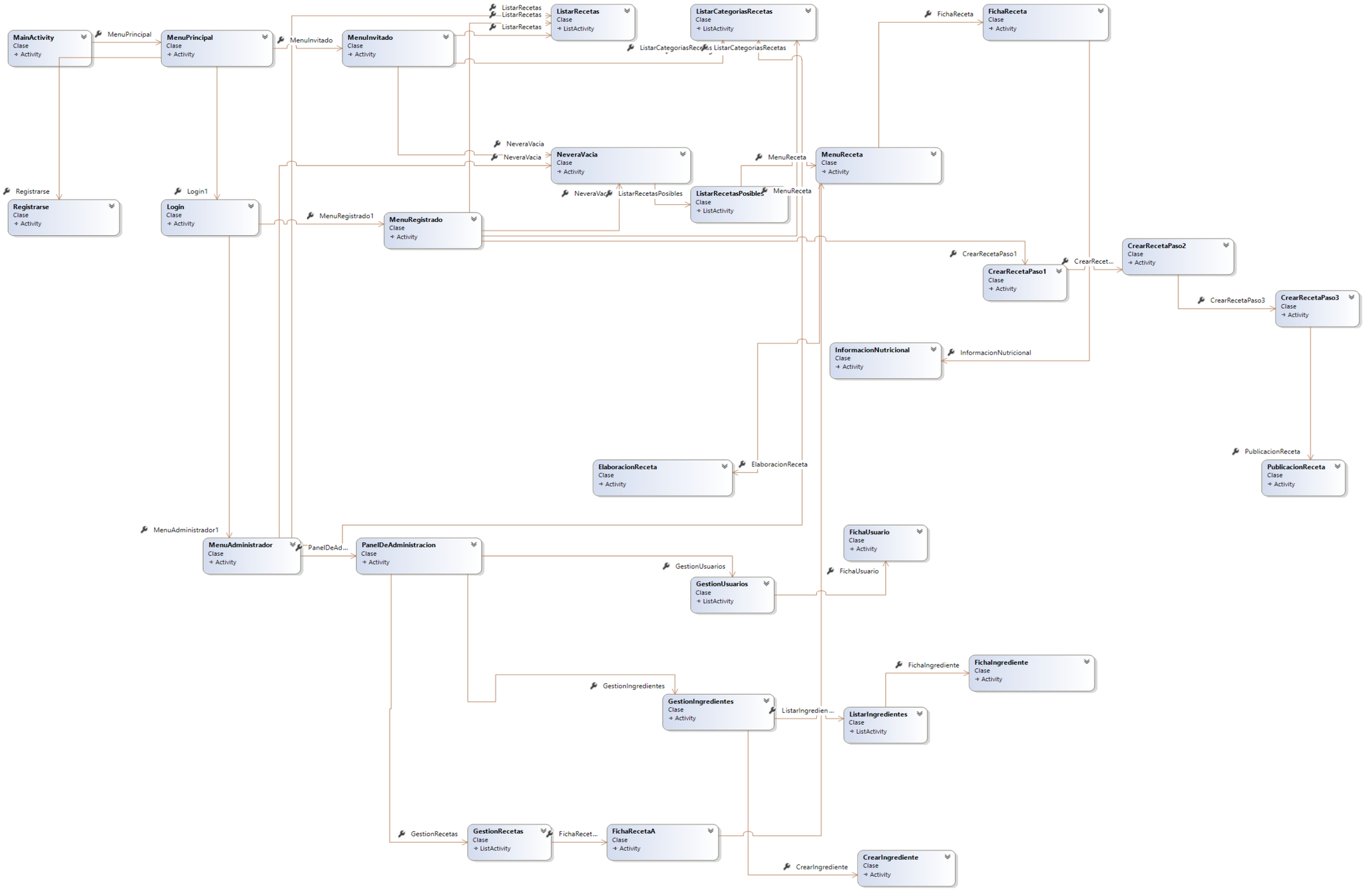
- **Materiales en PDF de la empresa InteligenceZ cedidos a terceros.**

12 Diagramas Completos

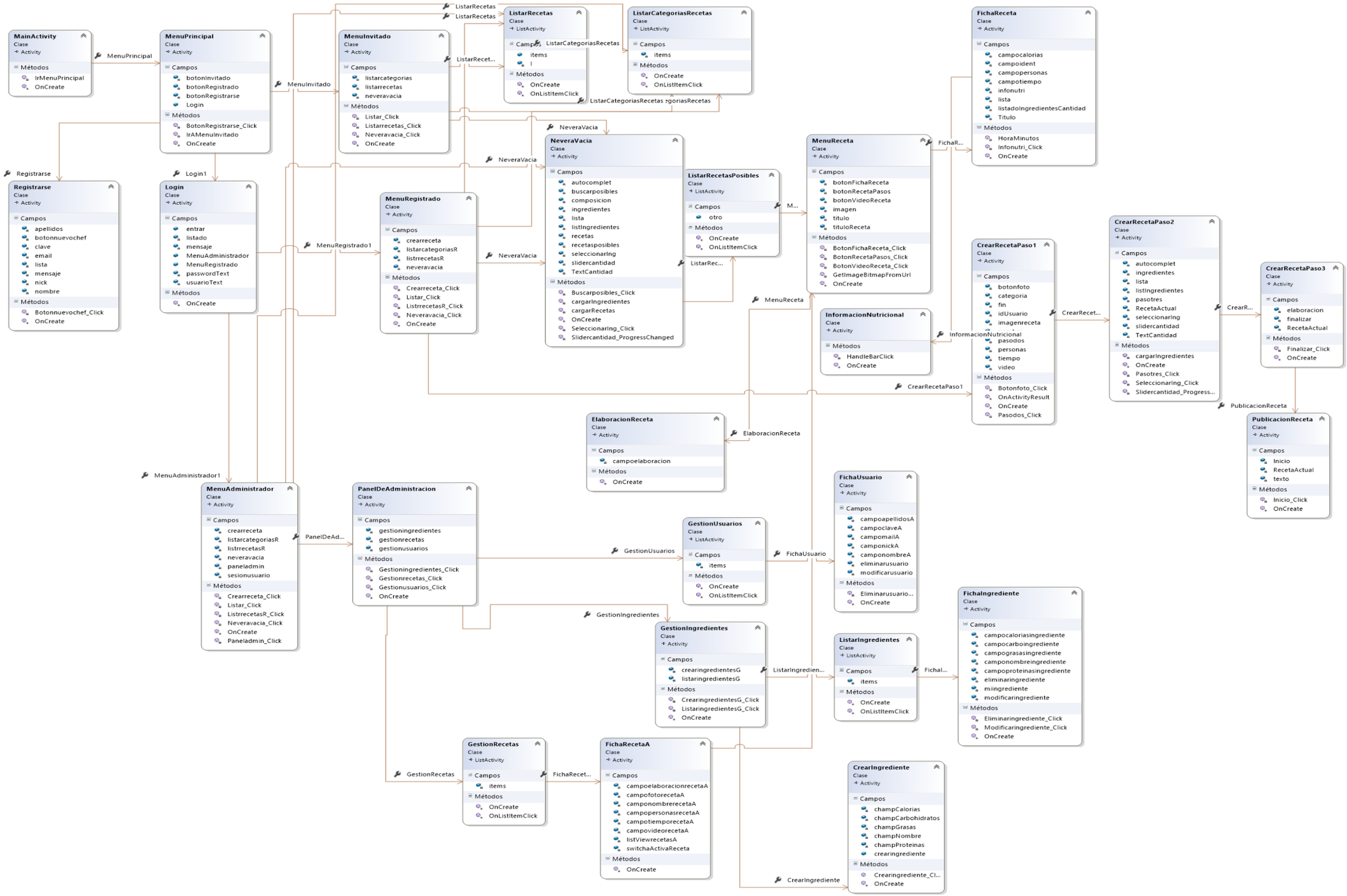
12.1Diagrama de Casos de Uso Completo



12.2 Diagrama de Clases Contraído



12.3Diagrama de Clases Extendido



13 Glosario de Términos

[1] **Android Studio**: Es un entorno de desarrollo integrado para la plataforma Android

[2] **Xamarin**: Xamarin es una compañía establecida por los ingenieros que crearon Mono, una implementación libre de la plataforma de desarrollo .NET para dispositivos Android, iOS y GNU/Linux. Cuando en este proyecto nos referimos a él, nos referimos al componente que se acopla a VS con ese mismo nombre y nos sirve para desarrollar apps para dispositivos móviles multiplataforma.

[3] **C#:** pronunciado C sharp en inglés, es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270)

[4] **Patrones de Diseño:** Los **patrones de diseño** son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al **diseño** de interacción o interfaces. Un **patrón de diseño** resulta ser una solución a un problema de **diseño**.

[5] **UML:** El lenguaje unificado de modelado (**UML**, por sus siglas en inglés, Unified Modeling Language) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el Object Management Group (OMG).

[6] **Datos Persistentes:** Como el propio nombre indica son datos que persisten, datos albergados por ejemplo en una base de datos, lo contrario son datos que se producen en un momento dado pero no se albergan indefinidamente.

[7] **HTML5:** HyperText Markup Language, versión 5, es la quinta revisión importante del lenguaje básico de la World Wide Web, HTML.

[8] **JavaScript**: abreviado comúnmente JS, es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

[9] **CSS:** Hojas de estilo en cascada (o CSS, siglas en inglés de Cascading Stylesheets) es un lenguaje de diseño gráfico para definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado

10 **iOS:** Sistema operativo para teléfonos Apple

[11] **APIS:** Una API (siglas de 'Application Programming Interface') es un conjunto de reglas (código) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas: sirviendo de interfaz entre programas diferentes de la misma manera en que la interfaz de usuario facilita la interacción humano-software.

[12] **Profiling:** En ingeniería de software el análisis de rendimiento, comúnmente llamado profiling o perfilaje, es la investigación del comportamiento de un programa de computadora usando información reunida desde el análisis dinámico del mismo. El objetivo es averiguar el tiempo dedicado a la ejecución de diferentes partes del programa para detectar los puntos problemáticos y las áreas dónde sea posible llevar a cabo una optimización del rendimiento (ya sea en velocidad o en consumo de recursos).

[13] **Herramientas de Testing:** Herramientas de testeo de aplicaciones

[14] **Cordova:** Apache Cordova es un framework de desarrollo de apliaciones móviles.

[15] **Frameworks:** En el desarrollo de software, un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software

[16] **Adapter**: El patrón Adapter (Adaptador) se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda

Servidor Apache: El **servidor** HTTP **Apache** es un **servidor** web HTTP de código abierto, para plataformas Unix (BSD, GNU/Linux, etc.), MicrosoftWindows, Macintosh y otras, que implementa el protocolo HTTP/1.1 y la noción de sitio virtual.

[18] **Nginx:** pronunciado en inglés “engine X”, es un servidor web/proxy inverso ligero de alto rendimiento y un proxy para protocolos de correo electrónico (IMAP/POP3)

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Este proyecto se enmarca dentro del campo de "Desarrollo dirigido por Modelos" que ha tomado auge tanto a nivel académico como industrial. Se centra en una Metodología de Modelado y Diseño del Software basada en UML e impartida en las asignaturas de MDS1 y MDS2 de la carrera de Ingeniería Informática de la Especialidad de Ingeniería del Software. Esta metodología ha sido usada en dichas asignaturas en entornos Java para el desarrollo de aplicaciones de escritorio.
Aquí vamos a adaptarla para que se pueda integrar con entornos de desarrollo de aplicaciones para dispositivos móviles. Más concretamente en entornos de Visual Studio y en Lenguaje C# utilizando Xamarin. El objetivo final a parte de demostrar que se puede adaptar esta metodología a un entorno diferente, es el de mostrar un ejemplo de un ciclo completo de desarrollo en el que hemos modelado, diseñado y programado una aplicación de Android a la que hemos llamado Empty Fridge (Nevera Vacía). Esta aplicación ofrece a los usuarios la posibilidad entre otros servicios de consultar las recetas detalladas de cocina que pueden elaborar ellos mismos con los ingredientes de que disponen en su nevera o despensa.

GRADO EN INGENIERÍA INFORMÁTICA, 2016/2017