Universidad ORT Uruguay

Facultad de Ingeniería

Gestión de Finanzas Personales

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Bruno Benzano - 168500

Guillermo Bergengruen - 168786

Adrián Claverí - 169118

Juan Ignacio Galán - 164559

Tutor: Martín Solari

2015

Declaración de autoría

Nosotros, Bruno Benzano, Guillermo Bergengruen, Adrián Claverí, y Juan Ignacio Galán, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el proyecto de grado de fin de curso de la carrera Ingeniería en Sistemas;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Bruno Benzano

Guillermo Bergengruen

Adrián Claverí

Juan Ignacio Galán

Agradecimientos

En primer lugar, nos gustaría agradecer a Martín Solari, quien en su rol de tutor nos acompañó en toda la tesis, aportando su conocimiento y gran apoyo, siempre encontrando tiempo para dedicarnos y proveer consejos de gran valía.

En segundo lugar a Rodrigo Freire, arquitecto de Paganza, que en incontables ocasiones dedicó tiempo libre para apoyarnos en la construcción del producto. Su conocimiento del dominio de Paganza, su experiencia técnica y su visión de los objetivos a futuro hicieron indispensable su aporte.

También agradecemos a la empresa Paganza por su decisión de presentarse a la iniciativa de la Universidad ORT, que sin ellos este proyecto no existiría. Además agradecemos todos los recursos que pusieron a nuestra disposición, facilitando en gran medida la realización del proyecto.

Por último, a nuestras familias y amigos, que supieron entender que este proceso conlleva mucho tiempo y nos acompañaron para hacerlo más leve, sin recriminar los días que a causa del proyecto estuvimos ausentes.

Abstract

El proyecto tiene como objetivo desarrollar una herramienta para permitir al usuario registrar, analizar y tomar decisiones con respecto a sus finanzas personales. Se entiende por finanzas personales aquellos datos relacionados a transacciones económicas que afectan a una persona o a un núcleo familiar. Este proyecto fue presentado por la empresa Paganza, con la finalidad de integrar esta herramienta a su sistema actual. Paganza permite a los usuarios pagar diferentes servicios a través de la web y a través de su aplicación móvil. El principal objetivo de esta aplicación es simplificar proceso de pago haciendo énfasis en la usabilidad.

Para el desarrollo de esta herramienta se utilizará como principal fuente de datos los registros correspondientes a los pagos realizados mediante el uso de Paganza. El objetivo de la herramienta es organizar la información ya ingresada en el sistema permitiendo al usuario clasificar, visualizar y analizar los datos.

Las finanzas personales abarcan otras transacciones o conceptos que no están comprendidas dentro del sistema de pagos de Paganza, como por ejemplo compras diarias de alimentos, gastos de transporte o gastos ocasionales. La herramienta desarrollada deberá permitir el registro de estas otras transacciones.

Esto brindará al usuario un panorama más completo de sus finanzas y permitirá mejorar su capacidad de control y análisis. La herramienta enfatizará la experiencia de usuario facilitando el ingreso de los datos y la visualización de los mismos.

Palabras clave

Finanzas personales, Paganza, Análisis de gastos, Billetera electrónica, Gestión de gastos, Scrum, Reconocimiento de texto, Códigos QR, Ley de inclusión financiera.

Glosario

API: Interfaz de programación de aplicaciones (*Application Programming Interface*) es un conjunto de reglas (código) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas.

Azure: Es una plataforma ofrecida como servicio y alojada en los Data Centers de Microsoft.

Branch: Rama, es el duplicado de un objeto bajo control de revisión que permite que las modificaciones sucedan en ambas ramas en paralelo.

Breadcrumb: Sistema de navegación auxiliar utilizada en interfaces gráficas de usuarios.

Buxfer: Herramienta online para gestión de gastos.

CSS: Hoja de estilo en cascada (*Cascading Style Sheets*). Es un lenguaje usado para definir y crear la presentación de un documento estructurado escrito en HTML.

Code first: Método que permite definir un modelo usando clases de C#, generando luego la base de datos.

Commit: Acción de consignar un conjunto de cambios en una herramienta de gestión de versionado.

Cross-platform: Multi-plataforma. Todo software o dispositivo capaz de poder utilizarse en diferentes plataformas.

CUTCSA: Compañía Uruguaya de Transportes Colectivos S.A. Es la empresa de transporte más grande de Uruguay, en cuanto a cantidad de unidades de transporte colectivo.

DTO: Objeto de Transferencia de Datos. Es un objeto que transporta datos entre procesos.

Database first: Método que permite primero crear la base de datos con sus tablas (y otras estructuras) y luego incorporarlas a la aplicación.

Dirección General Impositiva: Es la Unidad Ejecutora responsable de la administración de los impuestos internos del país.

Early adopters: Usuarios que prueban primero los productos más novedosos en busca de nuevas experiencias o para mejorar sus empresas o sus procesos.

Entity framework: Es un conjunto de tecnologías de ADO.NET que permiten el desarrollo de aplicaciones de software orientadas a datos.

Expensify: Herramienta online para gestión de gastos.

Feature: Características o funcionalidades de una aplicación.

Git: Software de control de versiones.

Gitbash: Consola de Git.

HTML: Lenguaje de marcas de hipertexto (*HyperText Markup Language*). Hace referencia al lenguaje de marcado para la elaboración de páginas web.

Hosting: Alojamiento web. Servicio que provee a los usuarios de Internet un sistema para poder almacenar información, imágenes, vídeo, o cualquier contenido accesible vía web.

IVA: Impuesto al Valor Agregado, tributo o impuesto que deben pagar los consumidores al Estado uruguayo por el uso de un determinado servicio o la adquisición de un bien.

JSON: Es un acrónimo de JavaScript Object Notation, un formato ligero originalmente concebido para el intercambio de datos en Internet.

JavaScript: Es un lenguaje de programación interpretado.

KITE: Herramienta de generación de scripts para la medición de performance de sitios web.

Look and feel: La apariencia del sistema.

MVC: El modelo–vista–controlador es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones.

MVP: Producto mínimo viable (Minimum Viable Product) se define como el producto de mayor retorno de inversión, que contiene las partes clave de lo que necesita el cliente.

Merge: Unión, proceso para unificar 2 *branches*.

Migrations: Conjunto ordenado de pasos que describen cómo actualizar (y degradar) el esquema de base de datos.

OCR: Reconocimiento óptico de caracteres (*Optical Character Recognition*) es un sistema computarizado de análisis que permite escanear un documento de texto.

PDF: Formato de documento portátil (*Portable Document Format*) es un formato de almacenamiento para documentos digitales independiente de plataformas de software o hardware.

POS: Punto de venta (*Point of Sale*) terminal electrónica utilizada para cobrar con tarjeta de crédito o débito.

Product Backlog: Listado de los ítems o características del producto a construir.

QR: Código de respuesta rápida (*Quick Response Code*) es un módulo para almacenar información en una matriz de puntos o en un código de barras bidimensional.

REST: Transferencia de Estado Representacional (*Representational State Transfer*) Interfaz entre sistemas que utilice directamente HTTP para obtener datos.

RUT: El registro único tributario es un mecanismo para identificar, ubicar y clasificar a las personas y entidades que tengan la calidad de contribuyentes.

Razor: Sintaxis para crear aplicaciones web.

***Refactoring*:** Técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

Release: Versión de un producto que se le entrega a un cliente.

SCM: Gestión de Configuración de Software (*Software Configuration Management*) Es el conjunto de procesos destinados a asegurar la calidad de todo producto a través del control de cambios.

SQA: Aseguramiento de calidad de software (*Software Quality Assurance*) Conjunto de actividades para asegurar que los procesos y productos de software cumplen con los requerimientos, estándares y procedimientos.

Scrum: Marco de trabajo relacional e iterativo empleado en metodologías ágiles.

Sprint: Referido a las iteraciones en Scrum.

SQLite: Sistema de gestión de bases de datos relacional.

Stakeholders: Refiere a trabajadores, organizaciones sociales, accionistas y proveedores, que se ven afectados por las decisiones de una empresa.

Storyboard: conjunto de ilustraciones o elementos mostrados en secuencia con el objetivo de servir de guía para entender una historia.

UI: Interfaz de usuario (*User Interface*) Es el medio con que el usuario puede comunicarse con una máquina comprendido por todos los puntos de contacto entre el usuario y el equipo.

UserVoice: Herramienta web para recolectar comentarios de usuarios.

XAML: Lenguaje Extensible de Formato para Aplicaciones (*eXtensible Application Markup Language*) Es el lenguaje para la interfaz de usuario para WPF y Silverlight.

Xamarin Forms: Herramienta que crea una abstracción sobre la interfaz de usuario de Android, iOS y Windows Phone permitiendo desarrollarla una única vez con código C#.

Xamarin: *Framework* libre de la plataforma de desarrollo.NET para dispositivos Android, iOS y GNU/Linux.

Xpenditure: Herramienta online para gestión de gastos.

Índice

1. Introducción

1.1. Contexto del proyecto

El proyecto surge como iniciativa planteada por Paganza [1] para extender sus funciones actuales. La idea propuesta por Paganza fue desarrollar una herramienta enfocada en las finanzas personales, dada la relación que existe entre esta área y las necesidades que cubre con sus funcionalidades actuales. Las finanzas personales están compuestas por el conjunto de transacciones económicas realizadas por las personas derivadas de las actividades cotidianas de las mismas.

Uno de los factores que hacen que las finanzas personales sea un área de interés para el proyecto es el creciente uso de los dispositivos móviles y de la tecnología aplicada a las tareas cotidianas. Además del aumento notorio del uso de medios de pago electrónico en el país, incentivado por la aplicación de la ley de inclusión financiera [2][3]. Esta ley, entre otras cosas, dictamina el uso obligatorio de cuentas bancarias para el pago de salarios y estimula el uso de los medios de pago electrónicos a través de devoluciones impositivas. La nueva ley también prevé la creación de una figura legal para la emisión de dinero electrónico. Paganza se encuentra actualmente realizando las gestiones requeridas por la regulación para establecerse como uno de estos emisores de dinero electrónico.

1.2. Definición de Paganza

Paganza es un servicio que permite a usuarios pagar facturas y contratos a través de su página web o de su aplicación para *smartphones*. El servicio que brinda Paganza consiste en simplificar los pagos de las cuentas a los usuarios realizando débitos de su cuenta bancaria asociada. Los usuarios, a medida que reciben los comprobantes de consumo, pueden ingresar en el sistema los datos necesarios para que Paganza efectúe el pago. La mayoría de estos comprobantes contienen un código de barras que puede ser escaneado desde la aplicación móvil para facilitar el ingreso de estos datos. Para los casos en los que este código no esté disponible, el registro puede hacerse manualmente ingresando los datos necesarios. En resumen, Paganza permite el pago de cuentas generadas por servicios contratados, evitando a los usuarios concurrir físicamente al lugar de pago previsto por los proveedores de estos servicios.

1.3. Objetivos del producto

Las actividades cotidianas de las personas hacen que estas realicen con frecuencia diferentes transacciones económicas que constituyen los movimientos de sus finanzas personales. Uno de los problemas más importantes que se les presentan a las personas para poder gestionar las finanzas personales es el registro de los datos para poder controlar y analizar de forma eficaz sus finanzas, y tomar decisiones para mejorar la administración de las mismas. Por lo tanto, el objetivo del producto a desarrollar es crear una herramienta para ordenar los datos que los usuarios tienen en el sistema actual e ingresar nuevas fuentes de gastos, para permitirles analizarlos y tomar decisiones con respecto a sus finanzas personales.

Esta herramienta debe mejorar la forma en la que los datos de los pagos se visualizan actualmente, mostrando los movimientos que se realizan sobre la cuenta bancaria y cómo se relacionan con los pagos efectuados. Finalmente, se deberá facilitar el ingreso de los gastos en los que incurren las personas, que no pueden ser pagados a través de Paganza, con el fin de poder tener una visión completa de los egresos de las finanzas personales.

1.4. Objetivos del proyecto

Uno de los objetivos del proyecto es mantener el estándar de calidad de desarrollo de Paganza. Otro de los objetivos del proyecto es poder administrar la posible puesta en producción de funcionalidades dentro del marco de un producto que se encuentra en uso por un gran grupo de usuarios. Por otro lado, se debe hacer énfasis en la innovación y la usabilidad.

Dentro de los objetivos del equipo se encuentra, por un lado, obtener conocimiento al desarrollar un producto en el marco de un proyecto de grado. Además, superar el desafío que genera trabajar con un cliente real que cuenta con un producto estable con 45.000 usuarios activos [4]. Por último, la posibilidad de generar contactos profesionales a través de la interacción con el mercado laboral del sector.

1.5. Marco de desarrollo del proyecto

El proyecto se desarrolló en el contexto académico del laboratorio Software Factory de la Universidad ORT Uruguay. La Software Factory provee un marco de aprendizaje enfocado

en la transferencia tecnológica de productos de calidad a la industria. Para lograr estos objetivos se aplican prácticas de ingeniería de software en un contexto real. Se promueve que los equipos sean auto-gestionados y realicen evaluaciones periódicas de su proceso de desarrollo. Se provee a los estudiantes de un marco de tutorías y revisiones periódicas para facilitar los procesos de aprendizaje.

El proyecto fue realizado con la colaboración continua de la empresa Paganza. El arquitecto de software y desarrollador principal de esta empresa tuvo amplia disponibilidad para validaciones conceptuales, así como para responder dudas técnicas en distintos niveles. Los fundadores de la empresa participaron en las reuniones de validación de alto nivel. La empresa también facilitó lugares de reunión y acceso a recursos de software para el desarrollo de este proyecto.

1.6. Descripción del equipo

El equipo está compuesto por cuatro estudiantes de ingeniería de sistemas. Como necesidad de organización y para establecer claramente responsabilidades dentro del equipo, se buscó designar roles para cada uno de los integrantes. A lo largo de la ejecución del proyecto y como resultado de la asignación y ejecución de las tareas, estos roles fueron enfatizándose más.

Los roles definidos fueron: Arquitecto, *Scrum Master*, Desarrollador, Encargado de SQA, *Tester* y Encargado de SCM. Dentro del rol de desarrollador, que fue asignado a todos los integrantes, se distinguen diferentes perfiles de desarrollo. Estos perfiles fueron asignados teniendo en cuenta la experiencia de cada uno de los integrantes y sus conocimientos sobre las plataformas. Los mismos fueron Desarrollador Web *Frontend*, Desarrollador Web *Backend* y Desarrollador Móvil.

Dado que los roles superan en cantidad a los integrantes del equipo, se asignarán varios roles por integrante. De todas maneras, la filosofía del equipo es mantener la posibilidad de que todos los integrantes participen en todas las áreas, aportando para satisfacer las metas propuestas. La asignación de estos roles se realizó como señala la Tabla 1-1.

Nombre	**Roles**		
Adrián	*Scrum Master*	*Tester*	Desarrollador Web *Frontend*
Bruno	Encargado de SQA	Desarrollador Móvil	
Guillermo	Encargado de SCM	Desarrollador Móvil	Desarrollador Web *Frontend*
Juan Ignacio	Arquitecto	Desarrollador Web *Backend*	

Tabla 1-1: Asignación de los roles en el equipo

2. Problema

Las finanzas personales están compuestas por las transacciones económicas cotidianas. La gestión eficiente de las finanzas personales implica la administración de los recursos disponibles para satisfacer los objetivos planteados. Estos objetivos pueden ser, por ejemplo, mejorar la capacidad de ahorro o minimizar gastos innecesarios. Independientemente de cuáles sean estos objetivos, que en caso de las finanzas personales dependen de las personas o familias, para llevar a cabo esta administración es necesario disponer de determinada información. Esta información permite conocer qué transacciones se han realizado y en qué estado se encuentran, para poder evaluar si los objetivos pueden cumplirse o no, y qué acciones deben tomarse para alcanzarlos. Para ejercer este control, las personas se encuentran con ciertos problemas que hacen difícil esta tarea. Algunos de los factores identificados que generan estos problemas son los siguientes.

2.1. Diferentes orígenes de gastos

Los gastos en los que incurren las personas son originados por las actividades que realizan. Algunos de ellos, obedecen al mantenimiento de su vivienda, de su vehículo, sus gastos en salud, educación, impuestos, transporte o entretenimiento. Además, las personas contraen gastos determinados por su actividad laboral o generados por su familia. El hecho de que estos gastos sean diversos en su origen hace más difícil su registro y administración.

2.2. Diferente periodicidad de los gastos

Otro factor importante es la frecuencia con la que estos gastos se producen. Algunos de estos gastos se generan de forma mensual, semestral o aperiódica. Esta característica de los gastos hace más difícil su seguimiento.

2.3. Diferentes formas de pago

A medida que los gastos son generados, las personas deben hacer efectivos los pagos de los mismos, pero para hacerlo disponen de diversas modalidades. Algunos de estos pagos se realizan en efectivo, otros con débitos automáticos de las cuentas bancarias o con tarjetas de débito o crédito, entre otros métodos. Esta diversidad de modalidades hace que la información se encuentre en distintos lugares y sea más complejo el análisis de estos pagos. Otro aspecto de este factor, es que la aplicación de la ley de inclusión financiera, establece

devoluciones impositivas con valores diferentes e instrumentadas con diversos mecanismos dependiendo del medio de pago electrónico [2].

2.4. Diferentes tipos de comprobantes

Otra de las fuentes de información sobre los gastos que los usuarios disponen son los comprobantes y documentos asociados a los mismos. El inconveniente que genera el análisis de estos documentos es definir un criterio único para documentos totalmente distintos como *tickets*, facturas o comprobantes digitales. Estos documentos imponen la dificultad de ser almacenados físicamente de forma ordenada. A esto se le debe sumar la complejidad que implican las transacciones que no generan ningún tipo de comprobante.

2.5. Información desorganizada y difícil de consultar

En el caso de las personas que guardan los comprobantes para las transacciones que realizan, la consulta de los mismos debe hacerse de forma manual, lo que requiere tiempo y dedicación. Además, si el dato que quiere analizarse se compone de varios comprobantes, del mismo tipo o diferentes, esta tarea resulta más compleja. Si bien es posible llevar un registro organizado, uniforme y con los datos necesarios de las transacciones, resulta tedioso el ingreso de estos datos en algún medio que permita luego la consulta y el fácil manejo de la información.

2.6. Digitalización de transacciones financieras

Cada vez con más frecuencia, las transacciones se realizan utilizando medios de pago electrónicos. Estos medios comprenden las tarjetas de débito y crédito, las transferencias electrónicas de fondos y los instrumentos que permiten efectuar pagos electrónicos a través de cajeros automáticos, internet u otras vías reglamentadas correspondientemente. La ley de inclusión financiera [2] incentiva el uso de estos medios, pero de todas maneras los medios de pago no electrónicos siguen siendo usados. La coexistencia de medios no electrónicos, y medios electrónicos diversos forma parte del problema de la gestión de las finanzas personales.

3. Solución

Contemplando la problemática planteada, se propone como solución el desarrollo de una herramienta que permita el análisis de los gastos ingresados, haciendo énfasis en la automatización o simplificación del proceso de registro en la mayor medida posible. Entre otras cosas, esta herramienta permite a los usuarios visualizar y controlar cómo se distribuyen sus gastos en los diferentes servicios, detectar variaciones y tomar decisiones a partir de la información de sus egresos.

El desarrollo de esta herramienta implica extender algunas de las funciones actuales de Paganza, utilizar los datos que el sistema registra actualmente, permitir nuevas formas de ingreso y otorgar la posibilidad de analizar los datos de forma integral. Este proyecto consiste en tomar como entrada los pagos ingresados al sistema y permitir al usuario ver, categorizar y analizar sus gastos de forma simple. Algunos ejemplos de esto son: ver la evolución de los egresos por mes, observar cómo se distribuyen los gastos en los diferentes servicios, hasta crear etiquetas para organizar y consultar los egresos.

La solución planteada se basa en resolver los siguientes puntos para permitir la gestión eficiente de las finanzas personales, manteniendo el nivel de los estándares para poder integrarse a Paganza.

3.1. Registro

Para gestionar es necesario disponer de la información organizada eficientemente, para ello es necesario registrarla o recogerla automáticamente. Como se ha mencionado, el registro de la información requiere tiempo y conducta, por lo tanto el ingreso de la información debe ser fácil de realizar en la mayor medida posible. Deberán proveerse mecanismos de ingreso que faciliten la tarea del registro de los datos, disminuyendo el tiempo o el esfuerzo necesario. Los diferentes comprobantes disponen de información que puede ser analizada y extraída de forma automática, como el caso de los códigos de barras que ya utiliza Paganza para recoger la información del pago de facturas. Además, la Dirección General Impositiva se encuentra en la implantación del régimen de documentación de operaciones mediante comprobantes fiscales electrónicos. Estos documentos incluyen un código QR que contiene información de la operación que puede ser decodificado.

3.2. Organización

Con el fin de consultar la información ingresada y poder entender lo que representa, es necesario poder establecer criterios para ordenar los datos. La solución propuesta deberá proveer al usuario estructuras para definir estos criterios de organización y aplicarlos con sus datos. Por ejemplo, los gastos pueden ser agrupados bajo diferentes conceptos, los cuales pueden ser asociados con los gastos en forma de etiquetas, para luego poder consultarlos más fácilmente organizados de esta manera.

3.3. Consulta y análisis

Debido a que la gestión requiere la consulta de los datos ingresados, el sistema deberá permitir al usuario contar con los datos necesarios para analizar el estado de sus finanzas personales. Dentro de este análisis se encuentra la consulta de movimientos sobre la cuenta del usuario en Paganza. Esto permite al usuario observar los débitos realizados de su cuenta bancaria, los pagos correspondientes a los servicios contratados, las devoluciones impositivas y el saldo de su cuenta de Paganza. Esta consulta de movimientos también es importante para Paganza considerando sus perspectivas de consolidarse como emisor de dinero electrónico. Además, es de interés para los usuarios observar cómo están conformados sus gastos, tanto consultando los montos de los pagos efectuados como comparando el porcentaje que representan con respecto al resto de sus gastos. También es necesario observar cómo estos gastos se distribuyen en el tiempo para entender su evolución o posibles anomalías y encontrar sus causas. Como por ejemplo el aumento de consumo de energía eléctrica en invierno. Con esta información el usuario podrá analizar la situación y tomar decisiones que le permitan favorecer el cumplimiento de sus objetivos.

3.4. Planificación y asistencia para el control

Finalmente, para satisfacer los objetivos planteados es necesario pensar en qué acciones favorecerán en el futuro estos objetivos. Por lo tanto es necesario planificar estas acciones y controlar su ejecución. El sistema deberá facilitar esta planificación y su control. Por ejemplo, el usuario podrá establecer presupuestos y establecer valores de referencia para ser alertado cuando sean alcanzados. También se podrá consultar cuál es el valor actual de consumo sobre el presupuesto y presentarlo de forma visual.

4. Ingeniería de requerimientos

4.1. Proceso de ingeniería de requerimientos

Desde el inicio del proyecto se determinó que el enfoque del proceso de requerimientos es ágil, siguiendo con la metodología general del proyecto. Esto lo permite la cercanía con el cliente, además de la disponibilidad para realizar validaciones con el equipo. Otro punto a considerar es la posibilidad de acceso a usuarios actuales de Paganza, tanto virtualmente como personalmente. De esta manera se pueden recoger solicitudes directamente y realizar experimentos y entrevistas con los usuarios.

Al inicio del proyecto se comenzó con una etapa de extracción de requerimientos, para la cual se realizaron diversas reuniones con Paganza. En las mismas el cliente desarrolló su idea sobre finanzas personales y las funcionalidades básicas deseadas, por lo que el cliente determinó las primeras historias y un claro criterio de aceptación.

Para comenzar con el proyecto y generar un vínculo de trabajo, se realizaron estos requerimientos que ya estaban validados por el cliente y el usuario. El cambio en el contexto bancario, impulsado por la aplicación de la ley de inclusión financiera, motivó los primeros requerimientos. El estímulo del uso de medios de pago electrónicos y las devoluciones impositivas provocó un aumento en este medio de pago. Debido a esto, los usuarios necesitaban tener más información acerca de los movimientos y devoluciones al pagar sus cuentas utilizando Paganza. Este nuevo contexto motivó a agregar nuevas funcionalidades, permitiendo una diferenciación estratégica tanto para los usuarios de Paganza como para los clientes (instituciones bancarias). Por lo que dichos requerimientos necesitaban estar en producción lo antes posible, con los estándares de calidad requeridos para una aplicación que ya cuenta con usuarios activos.

Las reuniones con Paganza y el análisis de solicitudes de usuarios no fueron suficientes para especificar los requerimientos. Por lo tanto, de forma paralela se comenzó con el desarrollo de las primeras funcionalidades, mientras que se validó con usuarios el resto de los requerimientos.

Fue así que se decidió ajustar el proceso de ingeniería de requerimientos, pasando el equipo a relevar requerimientos más activamente y especificando los requerimientos que serán

siempre validados por Paganza. Por lo que el cliente y los usuarios pasaron a ser una fuente distinta de requerimientos. De esta manera, se utilizó el gran conocimiento de Paganza sobre el negocio y el *feedback* temprano de los usuarios.

Utilizando una herramienta de *feedback online*, se continuó detectando nuevas necesidades de los usuarios. Además, mediante entrevistas sencillas se pudo entender como un usuario registraba sus gastos y en qué momento, para luego comenzar a construir prototipos sobre distintas ideas. Todos estos conceptos se validaban y priorizaban con Paganza, ya que la solución debía mantenerse en el marco del proyecto contemplando las expectativas del cliente.

El equipo se encargó de especificar los requerimientos en forma de historias de usuario. Las mismas incluían una definición de terminado que surgió del *feedback* de los prototipos y las reuniones con Paganza.

En la etapa inicial del proyecto se comenzó con el desarrollo en la plataforma web debido a que el cliente lo solicitó de esta manera. Aun así, las necesidades de los usuarios abarcaban también mejorar la experiencia móvil. Por lo que Paganza encontró interés en realizar prototipos para validar tecnologías y funcionalidades que pueden incluirse en la aplicación ya existente. A mitad de proyecto, y utilizando las mismas metodologías del proceso, se cambió el enfoque a dispositivos móviles, haciendo énfasis en la experimentación mediante prototipos funcionales, poniendo la interfaz de usuario en segundo plano.

4.2. Metodologías utilizadas

4.2.1. Design Thinking

Design Thinking es una metodología para generar ideas innovadoras que centra su eficacia en entender y dar solución a las necesidades reales de los usuarios [5]. Dicha metodología puede ser utilizada en cualquier campo, desde desarrollo de software a servicios. En este proyecto, se encontró utilidad en algunos conceptos, ya que la metodología no fue implementada en su totalidad.

De las características principales de la metodología, se hizo énfasis en las siguientes:

- Generación de empatía: Entender los problemas, necesidades y deseos de los usuarios implicados en la solución.
- Trabajo en equipo.
- Generación de prototipos: Define que toda idea debe ser validada antes de asumirse como correcta. De esta manera se identifican los fallos con antelación, identificándolos antes de dar con la solución.

Las otras características apuntan a generar una atmosfera lúdica durante el proceso. Por último, el desarrollar las técnicas con un gran contenido visual y plástico. Por más que se tomaron en cuenta estas características, no se hizo hincapié en ellas.

El proceso de Design Thinking comienza por generar empatía con los usuarios, comprendiendo las necesidades y el entorno de los mismos. Luego continúa con una etapa de definición, donde se filtran todos los conceptos surgidos en la etapa de empatía, identificando problemas cuya solución es clave para obtener el resultado deseado. Con una lista refinada de conceptos se pasa a una etapa de generación de ideas, donde se evalúan todas las posibles soluciones a los problemas antes planteados. Para convertir esa idea en realidad se pasa a una etapa de prototipación, visualizando las posibles soluciones y encontrando oportunidades de mejora. Por último, se entra en una etapa de pruebas con usuarios implicados en la solución sobre la cual se está desarrollando. En esta etapa se identifican posibles mejoras, fallos a resolver y carencias de la solución. En la Figura 4-1 se puede observar un diagrama del proceso completo de Design Thinking.

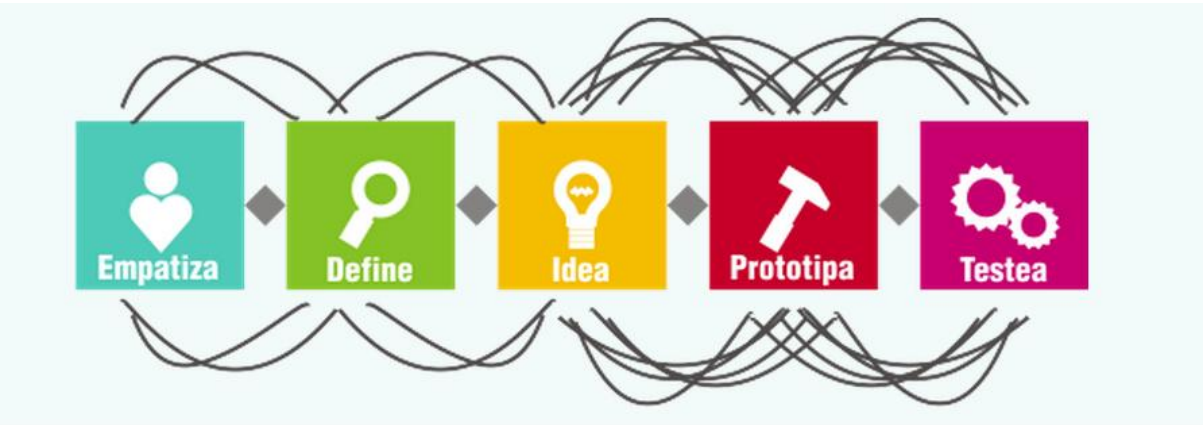

Figura 4-1: Proceso de Design Thinking [5].

El proceso de Design Thinking no es lineal, se pueden saltar etapas así como volver a una anterior en caso de ser necesario. La idea es partir de varios conceptos desordenados hasta tener un objetivo claro que colmen las expectativas u objetivos del equipo. Contando también con el concepto de iteración, repitiendo las etapas para obtener mejores resultados.

Para el proceso de detección y análisis de requerimientos, se tomaron algunos conceptos de dicha metodología, haciendo énfasis en las etapas de empatía y prototipación. Por lo que se realizaron diversas entrevistas con usuarios para entender el problema y diseñar posibles soluciones. En todas se trató de generar empatía y llegar a los aspectos emocionales. Se realizaron 4 entrevistas que fueron grabadas donde se realizaban preguntas sobre posibles funcionalidades a realizar. En las mismas se intentaba que el usuario guíe la entrevista en lugar de responder preguntas. Luego se realizaron otras entrevistas (4 realizadas por cada integrante del equipo) donde el foco sí fue responder una serie de preguntas. Un ejemplo de estas entrevistas puede verse en el Anexo 9.1. Se trató que los entrevistados pertenezcan a distintos sectores, con distinta experiencia en gestión de gastos, sin importar si eran usuarios de Paganza o no. De esta manera, se abarcaron diferentes aristas del problema, además de las reuniones semanales con Paganza, en las que se discutieron estos problemas y se pensó en qué prototipos utilizar. Los resultados de dichas entrevistas determinaron nuevos requerimientos para ser analizados y priorizados por el cliente.

Para los usuarios se generaban prototipos conceptuales y sencillos de realizar, mientras que a Paganza se le entregaban prototipos de bajo nivel, más enfocados en la solución final. De esta manera, se capturaron las necesidades reales de los usuarios, permitiendo dar una solución dentro del marco existente de Paganza.

Los requerimientos relevados utilizando esta metodología comprenden el registro de gastos no ingresados mediante Paganza y todo lo relacionado a dispositivos móviles. En la mayoría de las entrevistas los usuarios planteaban la necesidad de registrar los gastos que no se pagan con Paganza, con el fin de controlarlos y categorizarlos. Además, cuando se mencionaba la aplicación móvil de Paganza, se solicitaban ciertas funcionalidades sobre la gestión de los gastos, algunas de las cuales estaban contempladas en la plataforma web. Del

resultado de estas entrevistas se derivaron los requerimientos desarrollados. El detalle de los mismos, será explicado en la sección 4.5.

4.2.2. MVP

El MVP (*Minimum Viable Product*) se define como el producto de mayor retorno de inversión, que contiene las partes clave de lo que necesita el cliente [6]. Un MVP tiene solamente las *features* claves que permiten al producto ser puesto en producción. El público objetivo se compone por *early adopters* que tienden a proporcionar *feedback* del producto. Es una estrategia destinada a evitar construir productos que los clientes no quieren, maximizando el retorno de la inversión.

El proceso consiste en determinar cuáles son las *features* claves para el cliente, para eso es necesario responder la siguiente pregunta: ¿Cuál es el conjunto mínimo de *features* que se debe entregar para satisfacer al cliente? Toda idea, *feature* y trabajo realizado debe ser enmarcado con la misma pregunta. Por lo que, al tener la respuesta, se pueden separar las historias que son necesarias del *backlog* y agruparlas en la lista del MVP.

Dicha lista, es un subconjunto del *product backlog*, que está priorizada, estimada y mejor definida. Si bien el enfoque es trabajar en historias catalogadas para el MVP se puede trabajar sobre otras historias con el beneplácito del cliente y del equipo.

Para el proyecto se determinó un MVP con las funcionalidades claves para el desarrollo web. El mismo fue evolucionando hasta capturar el mayor valor en el menor tiempo posible. En lugar de concretar un segundo MVP, se siguió trabajando sobre lo entregado en el primero y el resto de las historias del *backlog*. Durante el transcurso del MVP se fueron confeccionando y refinando las historias restantes del *backlog*. Por lo que al entregar dicho MVP, se tiene un *backlog* priorizado con el cual se siguió trabajando.

Como se mencionó anteriormente, se utilizaron historias de usuarios como herramienta para especificar y validar requerimientos como se ve en la Figura 4-2. Además de la importancia de las historias en paradigmas ágiles, existen otros beneficios importantes, tales como dividir proyectos en pequeñas entregas o mantener una relación cercana con el cliente. Las historias se especificaron con un título, un criterio de finalización e imágenes

como prototipos para ejemplo. De esta manera se validaron los requerimientos a la hora de realizar la funcionalidad y al momento de la entrega.

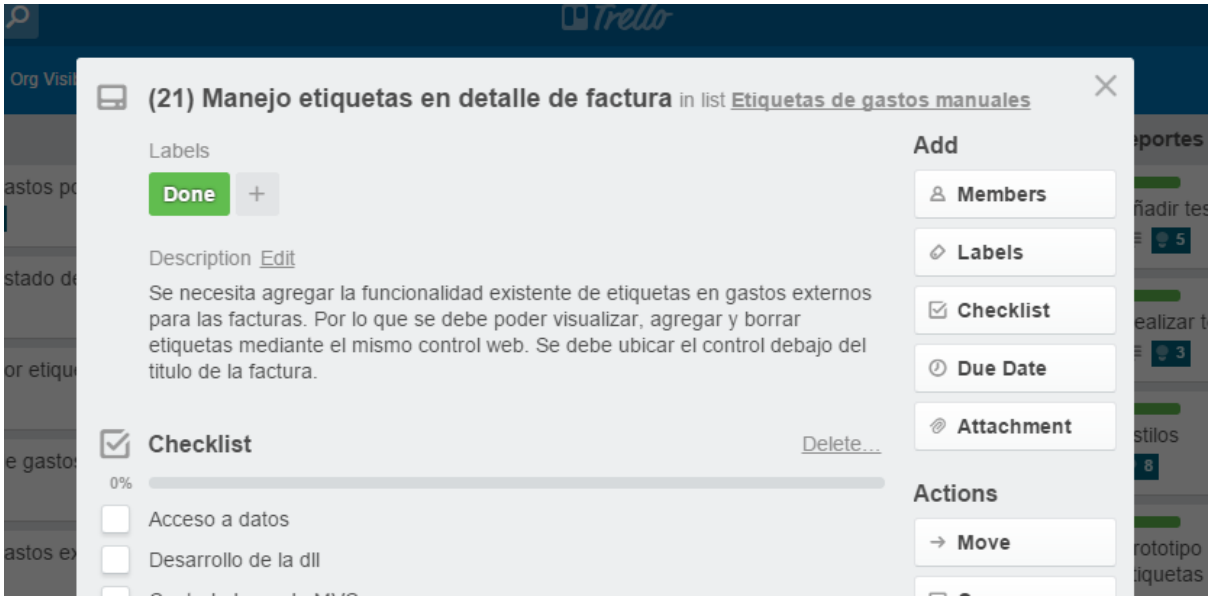

Figura 4-2: Captura de pantalla de historia de usuario

4.2.3. Resultado de las metodologías utilizadas

Se considera por parte del equipo que el resultado de la aplicación de ambas metodologías fue satisfactorio. En el caso de Design Thinking, se tomó la más útil de la metodología y se aplicó al proceso. Se consideraron los valores y procesos principales como apoyo al proceso, haciendo foco en conceptos como empatía y prototipación. Se utilizaron técnicas como el mapa de empatía o entrevistas, así como *storyboard* y prototipos en imagen, que determinaron los requerimientos claves para el proyecto.

En cuanto al MVP, se encontraron 2 grandes beneficios en el empleo de dicha técnica. En primer lugar, poder entregar valor rápidamente al cliente, cumpliendo con los tiempos para la puesta en producción, tanto como para validar los conceptos. Esto permitió contar con el *feedback* de usuarios y de Paganza lo antes posible. Por otro lado, generar un vínculo de confianza con el cliente, demostrando progreso tempranamente y comenzando a trabajar en conjunto.

4.3. Requerimientos relevados

4.3.1. Requerimientos iniciales de Paganza

En la primera reunión con Paganza para relevar requerimientos, realizada el 16 de octubre de 2014, surgieron los primeros conceptos de gestión de gastos personales. Además, el cliente acercó comentarios de usuarios realizados a través de UserVoice. Esta herramienta permite relevar comentarios a usuarios a través de la página web. La mayoría de los comentarios, sobre herramientas para análisis de gastos y transacciones dentro de la cuenta, solicitaban las mismas funcionalidades. Por ejemplo, un usuario comentó: "… ¿Dónde puedo ver o utilizar mi saldo por devolución de los puntos del IVA?". Otro comentario de un usuario: "Hola, quisiera saber de qué forma se utiliza el importe a mi favor ya que sólo veo el descuento correspondiente al mes".

Entre la primera reunión con Paganza y los comentarios de los usuarios, los requerimientos relevados son los siguientes:

- Etiquetas de gastos: Personalización de etiquetas para determinar categorías de los gastos.
    - Automáticas
    - Manuales
    - Contratos
- Alertas: Mensaje o mail ante un pago fuera del promedio de los usuarios del sistema.
    - "Pagos Raros"
- Reportes: Análisis gráfico de los gastos de un usuario en Paganza.
    - Gastos por mes
    - Por contrato
    - Por servicio
    - Estado de cuenta
    - Por etiqueta
    - Comparar con otros usuarios de Paganza
- Registrar gastos externos en Paganza: Permitir el ingreso de gastos por fuera de las facturas al sistema, utilizando alguna de las siguientes alternativas:

    - API
    - Aplicación externa o desde la web
    - Integrar notificaciones de bancos (tarjetas de crédito)
    - Reconocimiento de estados de cuenta en PDF
- Exportar datos de pagos de Paganza: Facilitar la exportación de datos para poder ser utilizado por otras aplicaciones o por el usuario, mediante alguna de las siguientes opciones:
    - API
    - Excel
    - Otro formato
- Empresa: Funcionalidades para clientes especiales de Paganza.
    - Doble firma para la aprobación de pagos
    - Usuarios relacionados
    - Personalización de UI

4.4. Análisis de requerimientos

4.4.1. Product Backlog

Una vez relevados los requerimientos de Paganza y usuarios, se comenzó a escribir el *product backlog*. A continuación se enumeraran las *features* que componen dicho *backlog*, cuya especificación se encuentra en la sección 4.5. En este caso solo se hará una breve reseña de cada funcionalidad.

Backlog inicial:

- Etiquetas para gastos manuales: Categorización manual de los gastos.
- Etiquetas para gastos automáticos: Categorización automática de gastos, basado en etiquetas previas y el concepto de contrato.
- Alerta de pagos raros: Mail alertando sobre un pago fuera de lo común para el resto de los usuarios.
- Reporte de gastos por mes: Gráfica indicando el monto gastado por mes.
- Reporte de estado de cuenta: Reporte indicando ingresos, egresos y devoluciones del IVA a la billetera de Paganza.

- Reportes por contrato: Comparación de gastos por mes y contra otros contratos.
- Reportes por servicio: Comparación de gastos por mes y contra otros servicios.
- Reportes por etiquetas: Gráfica mostrando qué gastos cuentan con cuál o cuáles etiquetas.
- Presupuesto mensual: Indicador gráfico de los ingresos de una persona comparados con la suma de los gastos para un mes.
- Presupuesto por categorías: Indicador gráfico de los gastos para un servicio determinado, pudiendo ser mensual o único.
- Alertas de presupuesto: Al alcanzar un monto determinado en el presupuesto, lanzar una alerta al usuario.
- Registrar gastos externos: Permitir el ingreso de gastos que no se puede registrar por Paganza.

4.4.2. Priorización de requerimientos y definición del MVP

Partiendo del *product backlog* anterior fue que se determinó el MVP. Para ello se estableció junto con Paganza y usuarios qué *features* tenían un mayor valor. La medida de valor está dada por el impacto en la capacidad de gestionar los gastos personales, además de lo que Paganza considere más cercano a estar en producción. Por consiguiente, el MVP priorizado se detalla a continuación:

- Análisis de gastos por mes.
- Reporte de estado de cuenta (Billetera).
- Etiqueta de gastos manuales.
- Registro de gastos externos.
- Reporte de etiquetas.

4.4.3. Prototipos realizados

A continuación se hará una reseña de los prototipos empleados para validar los requerimientos. Tanto para la web al inicio del proyecto como para dispositivos móviles, se utilizaron prototipos para todas las funcionalidades del proyecto, validando requerimientos y realizando pruebas de concepto sobre tecnologías.

4.4.3.1. Prototipos web:

Se realizaron prototipos para las 5 funcionalidades del MVP, abarcando desde prototipos de interfaz de usuario hasta funcionales. Para las funcionalidades más próximas a estar en producción, se realizaron solamente prototipos de interfaz de usuario. En cambio, para los restantes requerimientos se realizaron ambos tipos de prototipos. Los prototipos funcionales se utilizaron de forma evolutiva, obteniendo *feedback* temprano de Paganza y de los usuarios para poder validarlos. A continuación se adjuntan algunos ejemplos de requerimientos prototipados.

Análisis de gastos por mes:

En la Figura 4-3 se puede observar el primer boceto sobre el análisis. En este se indican conceptos clave como los gastos del mes, los gastos de un servicio y la comparación entre un servicio y el resto de los gastos del mes.

Figura 4-3: Primer boceto de análisis de gastos

Etiquetas:

En la Figura 4-4 se puede observar la primera aproximación sobre la funcionalidad, indicando en un gasto existente las etiquetas que contiene, además de cómo agregar o remover etiquetas.

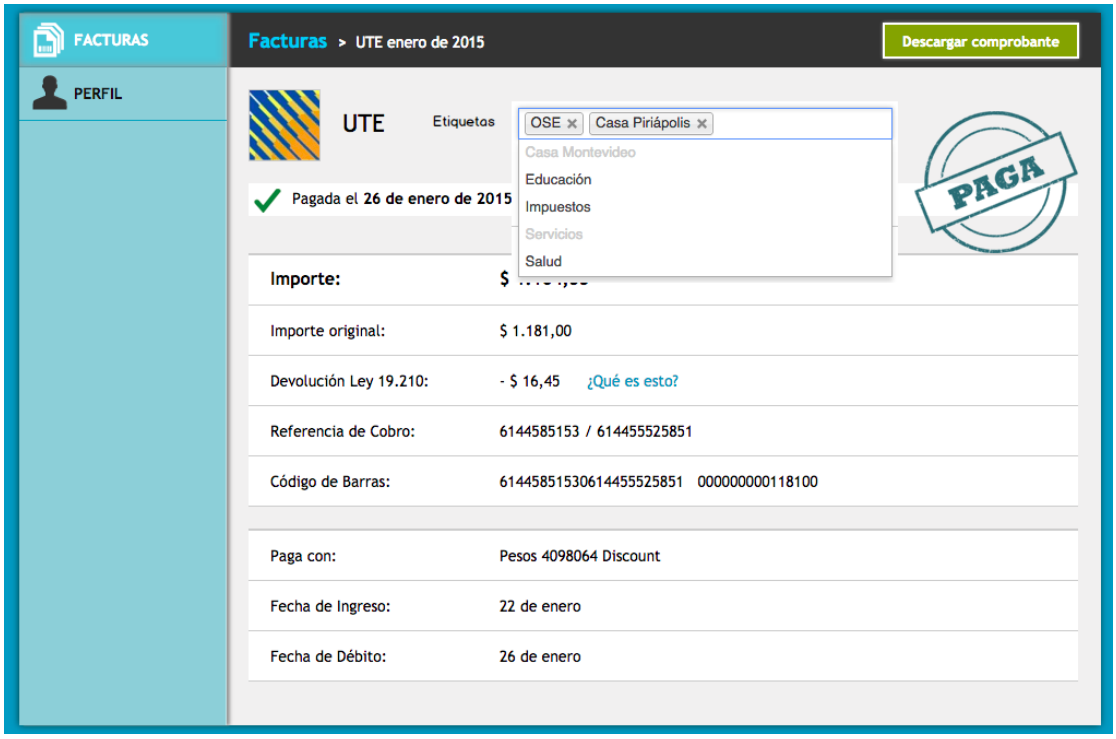

Figura 4-4: Primer versión de etiquetas

Gastos manuales:

En la Figura 4-5, se muestra un boceto de alto nivel sobre el ingreso de un nuevo gasto. Observando conceptos sobre los campos que determinar un gasto, y el flujo de la aplicación.

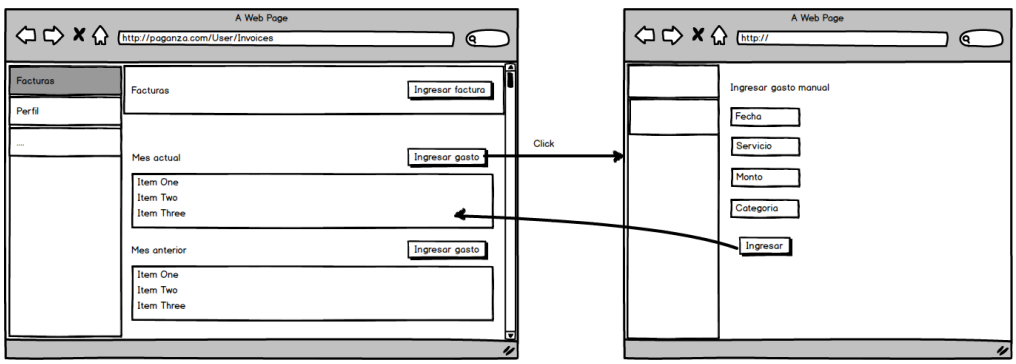

Figura 4-5: Boceto sobre ingreso de gasto manual

4.4.3.2. Prototipos móviles:

En este caso, todas las funcionalidades para dispositivos móviles fueron realizadas con el fin de validar los conceptos y la tecnología. Por lo que se comenzó con prototipos funcionales desechables sobre algunos conceptos. Luego se generaron nuevos prototipos pero evolutivos de las distintas funcionalidades. Se hizo hincapié en validar el contenido de los prototipos y no tanto en la interfaz de usuario. La validación se hizo tanto con Paganza como con los usuarios, generando instancias de *testing* y entrevistas con los mismos. En el Anexo 9.2 se puede ver con más detalle los prototipos realizados.

4.5. Especificación de requerimientos

En esta sección se dará un listado de las historias de usuarios determinadas para el proyecto, tanto para la web como para la aplicación móvil.

4.5.1. Desarrollo web

Análisis de gastos por mes: El objetivo es visualizar de forma simplificada la evolución de los gastos a lo largo de un período de tiempo utilizando una gráfica de columnas tomando una columna para cada mes. Por otro lado se deberá mostrar un listado con los servicios comprendidos en dicho período con sus correspondientes montos ordenados de forma descendente. Para comprender la proporción de estos servicios en el período, se incluirá una gráfica de torta para los servicios y sus montos asociados. Dentro de esta funcionalidad deberá incluirse la posibilidad de indicar las fechas de inicio y fin del período seleccionado y los servicios incluidos.

Reporte de estado de cuenta (Billetera): Similar al estado de cuenta proporcionado por un banco. El objetivo es mostrar las transacciones de la cuenta de Paganza, para un mes dado. Los movimientos de esta cuenta pueden ser del tipo débito de cuenta, pago de servicio o devolución del IVA. En caso de que se realicen varios pagos con un mismo débito se deberán mostrar agrupados y el usuario podrá ver el detalle de los mismos.

Gastos externos: Permitir el ingreso de gastos ajenos a los ingresados por Paganza, relacionados a gastos que no sean facturas, como por ejemplo el supermercado. Dicho ingreso puede ser mediante un formulario web como a través de la importación de un

archivo Excel. La información requerida debe ser la misma para ambas modalidades. Se debe presentar un listado de los gastos de manera similar al listado de facturas.

Etiqueta de gastos manuales: Permite asociar tanto gastos de Paganza como externos a una etiqueta. Dichas etiquetas corresponden a un concepto o categoría libre para el usuario, que le permite organizar mejor los gastos registrados. Etiquetar un gasto es posible tanto al momento de crear o editar. Las etiquetas tienen un nombre y color asociado. Se creará una página de administración para el manejo de dichos nombres y colores. Las etiquetas se muestran tanto en los listados como en el detalle del gasto.

Reporte de etiquetas: El objetivo es poder comparar la suma de los montos de los gastos que presenten una etiqueta ingresada por el usuario. Deberá incluir tanto gastos de Paganza como externos, permitiendo seleccionar una etiqueta y comprar contra el resto de los gastos para un período determinado.

Integrar listado de gastos con etiquetas: Utilizando el listado existente en Paganza, integrar la nueva funcionalidad de etiquetas, mostrando las mismas en el listado, así como permitir el filtrado.

Filtrar en listado de gastos externos: Utilizar la misma lógica que el filtrado en facturas de Paganza, para el nuevo listado de gastos externos, permitiendo tener las mismas funcionalidades en ambos listados.

Integrar gastos externos al análisis de gastos: Añadir como fuente de datos los gastos externos al análisis de gastos por mes, permitiendo mostrar solamente un tipo de gastos o ambos. De esta manera contemplar todos los orígenes de gastos ingresados en el sistema para usuario, pudiendo realizar las mismas comparaciones entre los distintos gastos a lo largo del período.

Integrar reporte de etiquetas al análisis de gastos: Similar a la historia anterior, pero considerando las etiquetas. De esta manera se reutiliza la experiencia del análisis, brindado mayor control sobre gastos etiquetados. Permitiendo ver los gastos correspondientes a una etiqueta en el mes, comparado contra el total y otras etiquetas, discriminando en un mes el total de gastos etiquetados y sin etiquetar.

Presupuesto mensual por rubro: Brindar la posibilidad de establecer montos por rubros, donde se sumen los gastos de ese rubro en ese mes. Se puede definir la periodicidad del presupuesto (mensual o único). Se asigna un monto máximo del presupuesto para indicar cuándo se llegó al límite.

Etiquetas automáticas de gastos: Sugerir etiquetas en gastos tanto de Paganza como externos. Generar un sistema de reglas para determinar cuándo corresponde agregar una etiqueta.

Alertas de presupuesto: Para un presupuesto dado, alertar al usuario en 2 momentos. Cuando se llegue a un monto mínimo para notificación y cuando se llegue al máximo monto del presupuesto. El usuario deberá poder establecer estos 2 valores.

Alertas pagos raros: Tomar como referencia pagos (personales) anteriores o el promedio, y alertar al usuario cuando el monto de un pago sea extraordinario.

4.5.2. Desarrollo móvil

Login: Vincular la cuenta existente de Paganza con la aplicación de gestión de gastos personales, mediante los mecanismos de seguridad existentes.

Ingreso de gastos externos manualmente: Permitir al usuario ingresar gastos externos completando un formulario.

Ingreso de gastos externos usando QR: Mediante el escaneo de un código QR, tomar todos los datos posibles para ingresar un gasto externo, autocompletando así los campos del formulario de ingreso.

Ingreso de gastos externos reconociendo texto: Mediante la captura de una foto de un comprobante, reconocer el texto para así ingresar un gasto externo, autocompletando los campos del formulario de ingreso.

Listado de gastos externos: Proveer un listado con todos los gastos registrados, tanto en el dispositivo como en la web de Paganza.

Reporte de billetera: Muestra un desglose de las transacciones del sistema utilizando Paganza, para poder visualizar cuánto dinero es debitado de la cuenta y cuánto corresponde a la devolución del IVA.

Geolocalización de servicios: Sugiere servicios según la posición del usuario y los gastos que tenga registrados hasta el momento, facilitando el ingreso de gastos externos al tener que completar menos campos.

Seguridad en la app: Proveer un servicio de seguridad para datos sensibles, al agregar un código que se solicita al iniciar la aplicación y cada vez que se accede a la misma.

Manejo de presupuestos: Permite establecer un presupuesto para un rubro, fijando un mínimo para notificación y un máximo. Al alcanzar alguno de los montos determinados, se lanza una notificación al dispositivo.

Búsqueda en el listado de gastos: Búsqueda avanzada y filtrado en el listado de gastos.

Reporte de gastos mensuales: Reporte mensual de los gastos registrados en Paganza.

4.5.3. Requerimientos no funcionales

Tecnologías Plataforma Web

Debido a la naturaleza del proyecto, se deberán utilizar todas las herramientas y tecnologías utilizadas por Paganza para el desarrollo del mismo:

- Se deberá utilizar ASP.NET MVC con el *framework* 4.0 de .NET, utilizando como IDE de desarrollo Visual Studio .NET 2013.
- En lo que respecta al servidor donde se despliega la aplicación, se debe utilizar Microsoft Azure.
- Se deberá utilizar Entity Framework versión 6.1.0.
- La base de datos a utilizar también se ubicará en Azure y debe utilizar el motor Microsoft SQL Server.
- Para la comunicación a través de APIs, se utilizará JSON como lenguaje de comunicación entre las diferentes plataformas.

Tecnologías Plataforma Móvil

Debido a la naturaleza del proyecto y las necesidades del cliente, la aplicación será desarrollada en Xamarin, plataforma ya utilizada por Paganza para realizar sus aplicaciones. Se deberá hacer uso de la tecnología Xamarin Forms, la cual Paganza desea investigar con fines de evaluar esta nueva tecnología teniendo en cuenta una futura renovación de la aplicación actual.

Debido a que el proyecto en el aspecto móvil se basa en la creación de prototipos, y sólo pretende realizar pruebas de concepto sobre los mismos, se seleccionó la plataforma Android para el desarrollo de los prototipos, más específicamente Android 4.4 KitKat.

Además de la necesidad de crear pruebas de concepto, la selección de Android también se debe a que dicha plataforma permite hacer deploy de aplicaciones sobre dispositivos físicos de manera más fácil que otras plataformas, como lo son iOS o Windows. Desarrollar para Android permite crear dichas pruebas de concepto de manera rápida y sencilla, pudiendo probar en dispositivos físicos sin mayores complicaciones.

Disponibilidad

Debido a que Paganza es un sistema que se encuentra actualmente en producción, se debe tener en cuenta que las funcionalidades a desarrollar no deben impactar en la actual disponibilidad del sitio. Esto es, se debe hacer especial énfasis en la introducción de código al sistema, controlando la mayor cantidad de excepciones posibles que se puedan generar, así como también permitir que el sistema se recupere y continúe funcionando.

Mantenibilidad

A la hora de diseñar y desarrollar las nuevas funcionalidades se debe tener en cuenta que la funcionalidad desarrollada podría ser mantenida y/o modificada por Paganza en un futuro, por lo cual debe ser lo suficientemente flexible y clara. Se debe tener en cuenta los posibles cambios que el cliente quiera realizar a futuro sobre las funcionalidades desarrolladas. La misma consideración se debe tener en cuenta a la hora de desarrollar la aplicación móvil.

Según lo relevado, se pretende realizar una extensión de la aplicación de Paganza, por lo cual se deberá utilizar las mismas herramientas tecnológicas que se están utilizando actualmente. Esto permitirá que la integración de los requerimientos desarrollados por el equipo con la solución actual sea lo más fácil y menos conflictiva posible. De cualquier manera, se deberá desarrollar una arquitectura que sea fácilmente adaptable a la solución actual, y a la vez deberá reutilizar los componentes actuales de Paganza para tener una solución coherente a través de todo el sistema. Por otro lado, todo tipo error generado debe ser auto-explicativo y debe proveerle a la entidad que consume dicho servicio la causa del error.

Performance

Las funcionalidades a desarrollar no deben impactar en la performance actual del sistema, por lo cual se debe realizar un estudio previo de la misma para las funcionalidades que se encuentran directamente afectadas por el alcance de este proyecto. Para esas funcionalidades, se debe tomar en cuenta el tiempo actual de respuesta sin las funcionalidades a desarrollar y controlar dicha performance a lo largo del proyecto para evaluar el impacto. Se deberán establecer métricas que permitan evaluar el impacto de las nuevas funcionalidades sobre el sistema actual, las cuales se encuentran en la sección 6.2.4.1. En caso de que las funcionalidades sean totalmente nuevas, se deberán establecer estándares de performance a partir de aplicaciones similares y tiempos de respuesta promedio del sistema actual.

Seguridad

Debido a que Paganza es un sistema de pago de facturas, y que permite a sus usuarios ingresar al sistema y manejar datos sensibles, se debe tener en cuenta que las funcionalidades a desarrollar no generen ningún tipo de brecha de seguridad. Se debe construir una arquitectura que se acople al sistema de autenticación actual de Paganza, reutilizando dicha funcionalidad. Por otro lado, las funcionalidades a desarrollar no deben permitir que se realicen cambios sobre los datos de los usuarios, así como también de las entidades existentes del sistema.

Testeabilidad

Tomando en cuenta que las funcionalidades a desarrollar podrían ser puestas en producción por Paganza, se debe crear una arquitectura que permita probar dichas funcionalidades de manera simple y acorde a los estándares y prácticas de *testing* realizadas actualmente por Paganza. Se debe tener en cuenta de qué manera se encuentran actualmente realizadas las pruebas del sistema por Paganza y tomar esto en cuenta a la hora de desarrollar el sistema.

Usabilidad

Considerando que Paganza es un sistema que actualmente es utilizado por una gran cantidad de usuarios que ya se encuentran familiarizados con el mismo, se debe tomar y respetar las mismas prácticas de usabilidad que ya se encuentran en uso. Las funcionalidades a desarrollar deben ser construidas para que los usuarios no las perciban como un sistema aparte, sino que forman parte de una misma solución.

Desde el punto de vista móvil, la aplicación debe ser rápida y sencilla, permitiendo al usuario crear gastos o consultar datos con un alto nivel de eficiencia. Se debe considerar el uso de tareas asíncronas para evitar que el usuario visualice ventanas en donde la interfaz no muestre el estado de la aplicación.

5. Diseño y arquitectura

5.1. Introducción

5.1.1. Organización del Documento de Arquitectura

La documentación de la arquitectura del sistema se organiza de la siguiente manera:

- ***Stakeholders***: se detallan qué entidades están interesadas en la documentación de la arquitectura del sistema, para determinar qué vistas detallar.
- **Principales Requerimientos del Sistema:** sección que explica qué problema se desea resolver y qué atributos de calidad y requerimientos funcionales son los que más influenciaron en la arquitectura.
- **Solución:** descripción a grandes rasgos de la arquitectura diseñada y de qué manera se atacó cada una de las necesidades.
- **Vista de Módulos:** las vistas de módulos explican a nivel de paquetes y clases los elementos del software y la relación entre ellos.
- **Vista de Componentes:** las vistas de componentes explican a nivel de componentes y conectores de qué manera interactúan las diferentes partes del sistema.
- **Vista de Asignación:** las vistas de asignación describen de qué manera se encuentran físicamente ubicados los diferentes componentes del software.
- **Plan de Implementación:** se describe el ciclo de implementación utilizado para el desarrollo del sistema.

En el Anexo 9.3 se explica de qué manera se encuentra documentada cada vista.

5.1.2. Stakeholders

A continuación se identifican qué entidades se encuentran interesadas en la arquitectura del sistema, con el fin de realizar una documentación que satisfaga las necesidades de dichas entidades. Se identificaron los siguientes *stakeholders* y sus necesidades en lo que respecta a la documentación de la arquitectura:

- *Scrum Master*: desde el punto de vista del *framework* utilizado, es necesario generar de manera iterativa diagramas de alto nivel que permitan observar la complejidad del sistema a desarrollar así cómo también qué tareas se deben realizar para

completar una funcionalidad. Esto lleva a que sea necesario una documentación de la arquitectura general del sistema a alto nivel, que permita realizar estimados a grandes rasgos.

- *Testers*: dada la necesidad de entregar un producto de calidad al cliente, será necesario crear *testing* unitario post entrega para cumplir con los estándares de Paganza. Para ello, será necesario tener identificado y documentado las áreas claves del sistema y los puntos de interacción entre los diferentes componentes, haciendo necesario la documentación de una vista de componentes.
- Desarrolladores: se necesita una documentación de la interacción entre las diferentes capas del sistema para poder separar responsabilidades de desarrollo y poder avanzar de una manera más rápida y fácil. Para lograr dicha flexibilidad, la documentación de vistas de componentes, así cómo también vistas de módulos, permitirán al equipo desarrollar de manera independiente siempre y cuando se respete lo definido en dichas vistas.
- Equipo de Mantenimiento: debido a que el sistema será entregado a Paganza, se debe tener una buena documentación para que dicho equipo pueda realizar el mantenimiento necesario. Debido a esto, se necesita la misma información que los desarrolladores pero con mayor énfasis en las vistas de módulos y componentes y conectores, donde se pueda distinguir fácilmente que entidades interactúan con cada funcionalidad y dónde se deben realizar los cambios necesarios. Por otro lado, para los procesos que se consideren de alta importancia en el sistema, será necesario una explicación del comportamiento del mismo.
- Cliente: Paganza como cliente estará interesado en los recursos necesarios que deben adquirir para que la aplicación funcione, y qué ambiente debe tener para que la aplicación se encuentre en producción. Se deberán crear vistas de despliegue y referencias a otros servicios que son necesarios para la aplicación.
- Usuarios finales: se debe tener en cuenta que los usuarios finales de la web podrán estar interesados en cómo funcionan las funcionalidades desarrolladas, por lo que se deberá documentar las principales operaciones del sistema de manera que un usuario las pueda entender, así como las restricciones necesarias para que la aplicación pueda ser utilizada.

En la Tabla 5-1 se puede observar el interés de cada *stakeholder* para cada vista.

	Módulos	C&C	Despliegue	Comportamiento
Scrum Master	Alto	Bajo	Medio	Bajo
Testers	Medio	Alto	Bajo	Medio
Desarrollador	Alto	Alto	Alto	Alto
Equipo de Mantenimiento	Alto	Alto	Bajo	Medio
Cliente	Bajo	Bajo	Alto	Medio
Usuarios finales	Bajo	Medio	Medio	Alto

Tabla 5-1: Interés de cada *stakeholder* por las vistas

Teniendo en cuenta la tabla anterior, se concluye que será necesario:

- Descripción detallada de los módulos, haciendo énfasis en la complejidad de las funcionalidades a desarrollar, así como también en la manera en la cual interactúan con otros componentes y cómo se puede agregar/modificar funcionalidad.
- Descripción detallada de los componentes y conectores del sistema, indicando los principales contratos del mismo, señalando los principales componentes que interactúan en las funcionalidades más importantes del sistema.
- Una vista de despliegue en donde se muestre dónde se ubica cada componente desarrollado y qué otros sistemas son necesarios para que el sistema funcione.
- Se deberá documentar las principales funcionalidades con énfasis en el comportamiento, haciendo uso de diagramas de actividad que le permitan a los desarrolladores saber de qué manera se ejecutan, y a los usuarios finales cómo funcionan las funcionalidades desarrolladas. Se decide que el uso de diagramas de actividad será más amigable para los usuarios finales, y útil para los desarrolladores,

dejando el uso de diagramas de secuencia para mecanismos más técnicos como acceso a base de datos o acceso a *backend*.

5.2. Principales Requerimientos del Sistema

5.2.1. Problema a Resolver

En primer lugar, el sistema actual de Paganza se encuentra pensado y diseñado para realizar pagos de las facturas de sus clientes. Esto resulta en que su arquitectura esté diseñada para atacar dicha necesidad, desde el ingreso de una factura hasta el procesamiento de dichos pagos, y no se encuentre enfocada en el análisis de las facturas existentes ni en el manejo de otro tipo de gastos. Se debe tener en cuenta este aspecto a la hora de diseñar la solución debido a que esta debe interactuar con el sistema actual.

En segundo lugar, los componentes a desarrollar deben ser completamente compatibles con la solución actual de Paganza. Esto comprende:

- Versión de .NET utilizada para desarrollar los componentes.
- Reutilización del código existente de Paganza.
- Flexibilidad en caso de que la estructura del código de Paganza sea modificado.
- Coherencia de código con la solución actual de Paganza, teniendo en cuenta la forma de programación, estándar de codificación, imitación de acceso a datos, etc.

Los componentes y funcionalidades a desarrollar deben tener un bajo impacto sobre la solución de Paganza, intentando minimizar los cambios realizados sobre la solución actual, enfatizando el uso de interfaces y confeccionado los módulos de tal manera que las funcionalidades desarrolladas se encuentren encapsuladas acordemente. También se debe considerar que es posible que las extensiones realizadas sean reutilizadas desde la aplicación móvil.

Teniendo en cuenta que es altamente probable que el código a desarrollar sea puesto producción en un futuro por Paganza, los componentes deben ser testeados acordemente y deben tener un alto nivel de prolijidad y claridad del código. En el Anexo 9.4 se detallan las buenas prácticas de programación que se deben aplicar.

En lo que respecta a la mantenibilidad del sistema, se debe buscar la manera de crear clases y extender comportamiento existente, y no modificar el código que Paganza actualmente tiene, a los efectos de minimizar el impacto sobre la solución actual. Debido a esto, los cambios a realizar sobre el proyecto existente deberán hacer énfasis en la reutilización de código y en creación de nuevas clases y archivos. En caso de que sea necesario realizar cambios profundos sobre el sistema, se deberá analizar con Paganza dichos cambios para poder minimizar el impacto sobre la aplicación actual.

En lo que respecta a la aplicación móvil se debe enfatizar el uso de la tecnología Xamarin Forms y sus diversos componentes, explorando las variables que tiene dicha tecnología, así como de qué manera se puede integrar con otros componentes y cómo desarrollar ciertas funcionalidades de manera nativa.

La aplicación debe permitir el ingreso de gastos de una manera fácil, sencilla y rápida. Se debe tener en consideración que el registro de gastos usualmente es tedioso para los usuarios y la aplicación debe permitir que el usuario pueda registrar los gastos sin mayores problemas y en el menor tiempo posible.

Luego, debido a que esta aplicación tiene un gran componente de *backend*, se debe crear un mecanismo que permita la fácil extensión y modificación del acceso a datos del *backend*. De la misma manera, la aplicación debe hacer uso de los datos que se encuentran localmente, a modo de mejorar los tiempos de respuesta para enriquecer la experiencia de usuario. Debido a esta razón, el sistema debe crear un mecanismo para el acceso a base de datos local que sea reutilizable para todas las entidades del sistema.

En términos generales se debe tener en cuenta que se está desarrollando una aplicación orientada a múltiples plataformas, por lo cual se debe abstraer la mayor cantidad de lógica posible para favorecer dichas plataformas.

Por último, en lo que respecta al reconocimiento de imágenes, el sistema debe permitir que se puedan registrar diferentes tipos de gastos, y que sea fácil de agregar nuevas estrategias de reconocimiento de gastos en el futuro. También se debe tener en cuenta el mecanismo a utilizar para reconocer imágenes. Por un lado, si se desea implementar el mecanismo de reconocimiento de imágenes por cuenta propia, se debe analizar dónde se ubicará dicho

servicio y el impacto que puede tener sobre la performance. Si se utilizara un servicio externo, se debe tener en cuenta el impacto que puede tener esto y de qué manera se puede implementar.

5.2.2. Componentes del Sistema Actual

A continuación se describen los componentes que serán útiles a la hora del diseño e implementación de la solución de gestión de gastos, así como también un diagrama de paquetes (Figura 5-1) para demostrar la dependencia entre ellos:

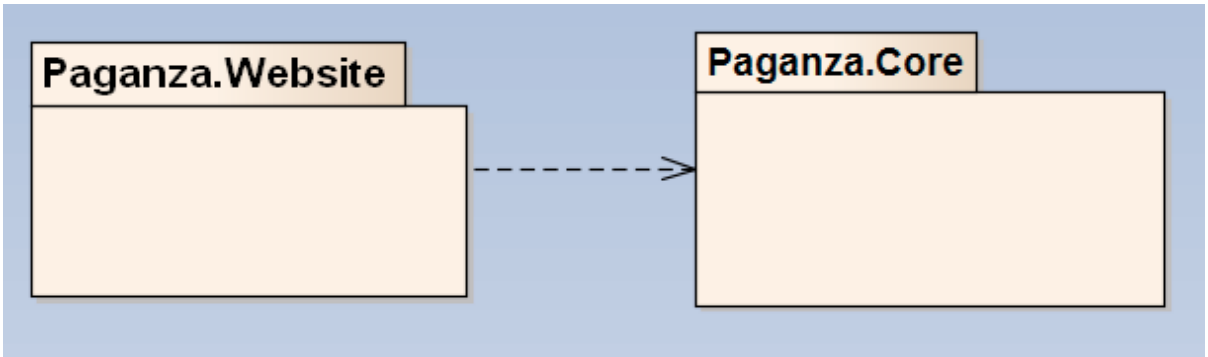

Figura 5-1: Componentes más importantes de la aplicación actual de Paganza

- Paganza.Website: solución MVC donde se encuentra toda la lógica que engloba la interacción con el usuario por medio de la web. Esta solución será importante para el equipo del proyecto, ya que será necesario hacer cambios sobre esta solución para poder integrar las extensiones realizadas.
- Paganza.Core y Paganza.Core.Async: son librerías binarias donde se encuentra la lógica del negocio, acceso a datos y manejo de los mismos. Desde el punto de vista del proyecto de gestión de gastos, son bibliotecas que se reutilizarán para mantener la coherencia con el sistema actual y poder usar la lógica existente

5.2.3. Características Técnicas Actuales

Plataforma Web: El sistema actual de Paganza se encuentra desarrollado en C# utilizando MVC y la tecnología Razor. Esta aplicación se encuentra desplegada en la nube utilizando Microsoft Azure como servidor. En dicho servicio también se encuentra la base de datos del sistema, siendo esta un Microsoft SQL Server. Por otro lado, la integración con la base de datos se realizó utilizando Entity Framework.

Plataforma Móvil: En lo que respecta al desarrollo de la aplicación móvil, en primer lugar Paganza ya cuenta con una aplicación para sus usuarios, desarrollada utilizando el framework Xamarin como herramienta de desarrollo multiplataforma. También se debe destacar que Paganza se encuentra actualmente considerando un rediseño de su aplicación móvil, intentando utilizar la nueva tecnología propuesta por Xamarin, siendo esta Xamarin Forms.

5.3. Solución

5.3.1. Solución General

Por un lado, se buscará el desarrollo basado en componentes que se integrarán con la actual aplicación web de Paganza, de manera que el actual usuario no se vea impactado por las nuevas funcionalidades. Dichos componentes se ubicarán entre el núcleo de la lógica actual de Paganza y la capa de presentación del usuario, cómo se muestra en la Figura 5-2.

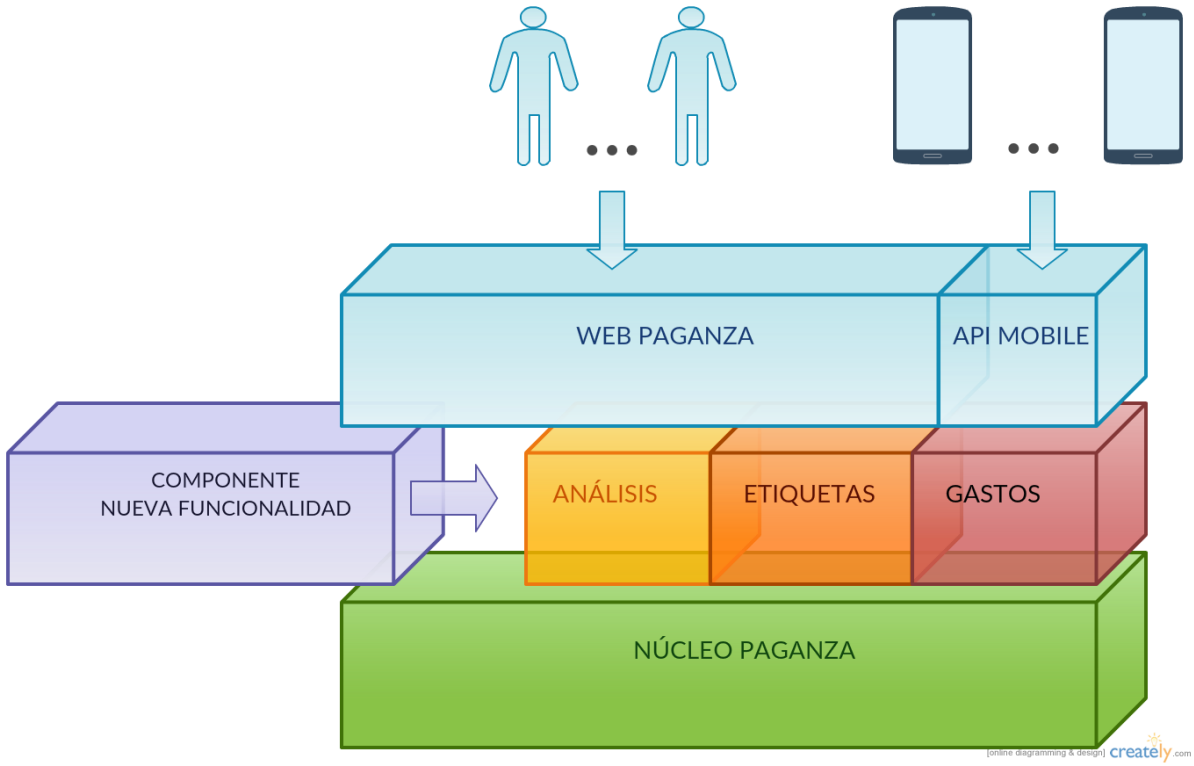

Figura 5-2: Solución arquitectónica del sistema

Para la aplicación móvil, se creará una API que haga uso de los componentes desarrollados, y permita el acceso a esta funcionalidad desde cualquier dispositivo móvil.

5.3.2. Plataforma Web

Teniendo en cuenta las necesidades del negocio, se decidió que la mejor manera de crear extensiones fáciles de implementar e integrar es haciendo uso de componentes, lo que en el ambiente C# se traduce a librerías binarias. Dentro de estas librerías, se encapsulará toda la lógica necesaria para:

- Obtener los datos necesarios para la extensión, ya sea acceder a la base de datos o procesar información.
- Realizar las validaciones necesarias en caso de que la obtención de datos se haga utilizando ciertos parámetros del negocio.
- Procesar la información y retornarla a quienes estén haciendo uso del componente de manera amigable para que sólo sea necesario desplegar la información y no realizar más procesamiento sobre dicha información.
- Proveer interfaces que servirán como punto de acceso a la extensión, las cuales serán implementadas dentro de dicha extensión.

A partir de esto, se define que los componentes deben, en primer lugar, tener una capa de acceso a la extensión. Esta capa estará compuesta por interfaces que ofrecen los servicios necesarios para ejecutar la extensión. Dentro de esa capa también se deberá encontrar la implementación de dicha interfaz. Por otro lado, debe existir una capa de acceso a la base de datos. Esta capa tendrá una fuerte relación y dependencia con el proyecto Paganza.Core y Paganza.Core.Async, ya que se reutilizarán las clases ya generadas y la lógica de acceso a datos realizada por Paganza. Luego, deberá existir una capa de procesamiento de los datos recolectados por la capa de acceso, para poder generar información amigable y fácil de ser desplegada al cliente.

Por último, dependiendo del caso y de la extensión a realizar, se tendrá una capa donde se realizarán operaciones de utilidad, como validación de parámetros, manejo de excepciones, funcionalidades de utilidad, etc.

A modo general, el funcionamiento de estos componentes debería seguir los siguientes pasos:

- Una entidad externa invocará uno de los métodos definidos en la interfaz expuesta.

- La clase que implementa dicha interfaz deberá atender el pedido.
- En caso de necesitar realizar algún tipo de validación de los parámetros recibidos, se utilizará la capa de utilidades.
- Se realizará el acceso a la base de datos.
- Se obtiene un resultado bruto, el que será retornado a la clase que implementa la interfaz expuesta.
- Dicha clase se encarga del procesamiento de la información para generar datos amigables.
- La información será retornada a la entidad que realizó el pedido.

La arquitectura diseñada presenta las siguientes ventajas:

- Bajo impacto sobre la solución actual de Paganza: se desarrollan componentes fuera de la aplicación de Paganza, aunque fuertemente relacionados con las entidades del negocio. El código a desarrollar se realizará sobre la librería binaria de Paganza.Core, pero todo tipo de acceso a datos, procesamiento y respuesta se realizará dentro del componente, lo que implica no tener que modificar las clases y métodos ya existentes en Paganza para agregar estas funcionalidades.
- Facilidad de desarrollo: la arquitectura permite desarrollar componentes de forma independiente, sin necesidad de trabajar sobre el proyecto de Paganza. Sólo se toma la librería binaria Paganza.Core, y se crea un sistema aparte para realizar los componentes. Una vez realizado el prototipo, la integración con Paganza simplemente debería implicar agregar el componente al proyecto y ajustar el *frontend* para poder consumir el nuevo servicio.
- El desarrollo en componentes enfocado a servicios permite que se puedan reutilizar estos componentes lógicos para la aplicación móvil, ya que la funcionalidad se encontrará encapsulada en dicho componente, aunque pueden existir cambios sobre las clases del negocio. Esto dependerá del componente que se esté desarrollando. Para permitir que la aplicación móvil se comunique con el sistema actual y con las nuevas funcionalidades a desarrollar, se deberá generar una API en la capa de presentación actual de Paganza para exponer los servicios creados.
- Responsabilidad Única: cada componente se focaliza en una funcionalidad en particular y toda la lógica de dicha funcionalidad se encuentra englobada en ese

componente. Esto permite centralizar el código de una cierta funcionalidad y fácil mantenimiento en caso de ser necesario las modificaciones sobre el código.

- Facilidad de cambio de componente: debido a que la dependencia entre Paganza y el nuevo componente está dada por una sola interfaz, es fácil cambiar dicho componente por otro, permitiendo modificar comportamiento de manera sencilla y rápida.
- Favorece la mantenibilidad: la encapsulación de funcionalidad y la definición de los servicios a través de interfaces permite que Paganza pueda identificar fácilmente qué componentes interactúan en las funcionalidades desarrolladas y en caso de ser necesario, realizar cambios de manera fácil y con el menor impacto posible.

Por otro lado, se pueden identificar los posibles riesgos:

- Puede ocurrir que los cambios necesarios para poder aplicar el *add-on* sean lo suficientemente complejos que entren en conflicto con la solución actual de Paganza. Se deberá tener en cuenta que la solución de Paganza debe cambiar lo menos posible, y que todo tipo cambio que se necesite hacer sobre el sistema actual tiene que ser absolutamente necesario.
- Performance: puede suceder que los nuevos componentes que se vayan a desarrollar impacten en la performance del sistema. Se debe tener en cuenta la cantidad de solicitudes realizadas al servidor, así como el tiempo de procesamiento y el acceso a datos.

5.3.3. Plataforma Móvil

Observando los problemas y atributos de calidad a satisfacer, se decidió utilizar, como estructura general del sistema, el patrón MVC (*Model-View-Controller*), siguiendo como ejemplo el patrón MVC orientado a la programación web, similar al patrón MVP (*Model-View-Presenter*), donde se encuentra una capa de presentación (*View*) la cual envía pedidos al *Presenter*, para luego interactuar con las entidades del negocio (*Model*).

A partir de esto, se puede separar la interfaz de la lógica del negocio, permitiendo que el sistema sea mantenible y reusable a partir de dicha independencia. Más aún, debido a que el equipo tiene poca experiencia sobre Xamarin, es altamente probable que la interfaz varíe

a lo largo del proyecto, así como también puede llegar a variar cuando el producto sea entregado a Paganza. Separando el sistema usando MVC permite que se desarrolle de forma paralela la interfaz del modelo del negocio, y permite que en un futuro, se pueda cambiar la interfaz de usuario o la lógica del negocio, ya sea por Paganza o un tercero sin mayores problemas, teniendo en cuenta que siempre se respetarán los contratos definidos.

Se define que el sistema tendrá 3 capas:

- Capa de presentación: debido a la naturaleza de la tecnología, la cual busca desarrollar aplicaciones *cross-platform*, se tiene una capa superior donde se almacena la lógica de interfaz de usuario. Luego, se divide dicha capa en varias subcapas, una capa base donde se ubica la lógica común a todas las plataformas para que luego diferentes capas consuman sus servicios. Para este proyecto, debido a que solo se desarrolló para Android, se tiene la capa base de lógica, y luego una capa específica para Android.
- Capa de servicios: para favorecer la usabilidad y la mantenibilidad, se desarrolló una capa de servicios que permite separar la interfaz de la lógica del negocio. Esta capa se encarga de contener los controladores que manejan los pedidos de la interfaz y definir de qué manera responder a dichos pedidos. Se debe respetar que la capa de servicios exponga interfaces las cuales serán la manera de comunicarse con el sistema, y que estas interfaces reciban parámetros de tipos primitivos, para minimizar la dependencia Dominio/Interfaz.
- Capa de lógica de negocio: en esta capa se ubicará toda la lógica del negocio, siendo estas las clases de modelo de negocio, así como el acceso a base de datos y acceso al *backend*.
- Por último, se tendrá una capa transversal al sistema, donde se tendrán clases de utilidades.

La separación en capas permite crear diferentes niveles de quiebre del sistema, donde se pueden intercambiar dichas capas en caso de que sea necesario una modificación sobre la lógica implementada.

Interfaz gráfica

En lo que respecta a la interfaz gráfica, se buscó hacer el mayor uso de componentes de Xamarin Forms, permitiéndole al cliente tener la mayor cantidad de ejemplos y prototipos posibles sobre dicha tecnología, agregándole valor al producto entregado.

Acceso a datos

Para el acceso a base de datos local, se busca ubicar dicha funcionalidad en la capa de negocio del sistema, pero dentro de un componente independiente de las demás entidades de dicha capa. Esto permite que dicho componente sea intercambiable en caso de que Paganza desee cambiar el motor de base de datos.

Para lograr el acceso a base de datos, se utiliza el motor de base de datos SQLite, compatible con las principales plataformas de desarrollo móvil. Luego, para un manejo rápido y genérico de las entidades de la base de datos, se creó una clase que permite agregar nuevas entidades de forma rápida y sencilla.

Se realizó una clase genérica que permite el acceso a diferentes servicios de *backend* utilizando diferentes métodos de HTTP. Esto permite reutilizar el código desarrollado para todas las solicitudes que se realizan al *backend*. Por otro lado, se define que la comunicación con el *backend* se realizará a través de respuestas en formato JSON.

Uso de herramientas físicas

Por otro lado, funcionalidades que utilizan herramientas físicas del dispositivo, como lo es la cámara, debieron ser resueltas a partir de código nativo, debido a que el manejo de la cámara y los archivos/imágenes necesitan ser accedidos a través de código específico para cada plataforma.

Para el reconocimiento de texto a partir de imágenes, se decidió utilizar un servicio de terceros [7] para procesar las imágenes y obtener el texto reconocido. Se consideró que la aplicación debe enfocarse en el reconocimiento de palabras claves orientadas a crear un gasto, y no enfocarse en el procesamiento de la imagen a partir de texto.

En lo que respecta al manejo de la imagen, se investigó que cada plataforma maneja de formas diferentes la entidad imagen, por lo que este procesamiento debe realizarse de forma nativa. Por esta razón, y manteniendo el concepto de favorecer la mantenibilidad, el servicio implementado para convertir las imágenes a texto recibe como parámetro la ubicación física de la imagen. De esta manera, se mantiene la regla de recibir parámetros de tipos primitivos, y se mantiene la independencia entre la interfaz y la lógica del negocio.

Luego, debido a que se reconocen facturas y recibos específicos, como lo son los recibos POS y los boletos de CUTCSA, se decidió utilizar esto para mejorar las habilidades de reconocimiento de la aplicación. Teniendo en cuenta que los algoritmos de reconocimiento de datos son más fieles para reconocer números sobre letras, se buscó números en dichos recibos que permitan deducir otra información valiosa del recibo.

Por un lado, los recibos POS contienen el RUT (registro único tributario) de la empresa que emite el recibo. A partir de ese RUT, que es único para cada empresa, se puede deducir el nombre de dicha empresa, permitiendo autocompletar el nombre del gasto a partir de dicho RUT. Cuando se reconoce la palabra RUT en el ticket, se realiza un procedimiento que permite detectar el nombre de dicha empresa, creando un gasto con ese nombre asociado. Por otro lado, los boletos de CUTCSA poseen dos números que permiten determinar otra información del recorrido tomado, como lo son la línea y el número de parada.

En lo que respecta al servicio de reconocimiento de texto, se decidió que dicho servicio sea gestionado por parte del *backend*, y que la aplicación móvil no dependa directamente de dicho servicio. Esto permite:

- En caso de que el servicio consumido no se encuentre disponible o deje de brindar dicho servicio, el impacto de cambiar esto se realiza solo en el *backend* de la aplicación, y no existe la necesidad de cambiar ni recompilar la aplicación.
- Es posible que en el futuro, Paganza decida implementar su propio OCR y no consumir un servicio de un tercero. Ubicando dicho procesamiento en el *backend* permite que sea fácil de cambiar el servicio externo por uno propio, ya que no tendrá impacto sobre la aplicación móvil.

A continuación se muestra (Figura 5-3) un diagrama de actividad para el proceso de reconocimiento de texto:

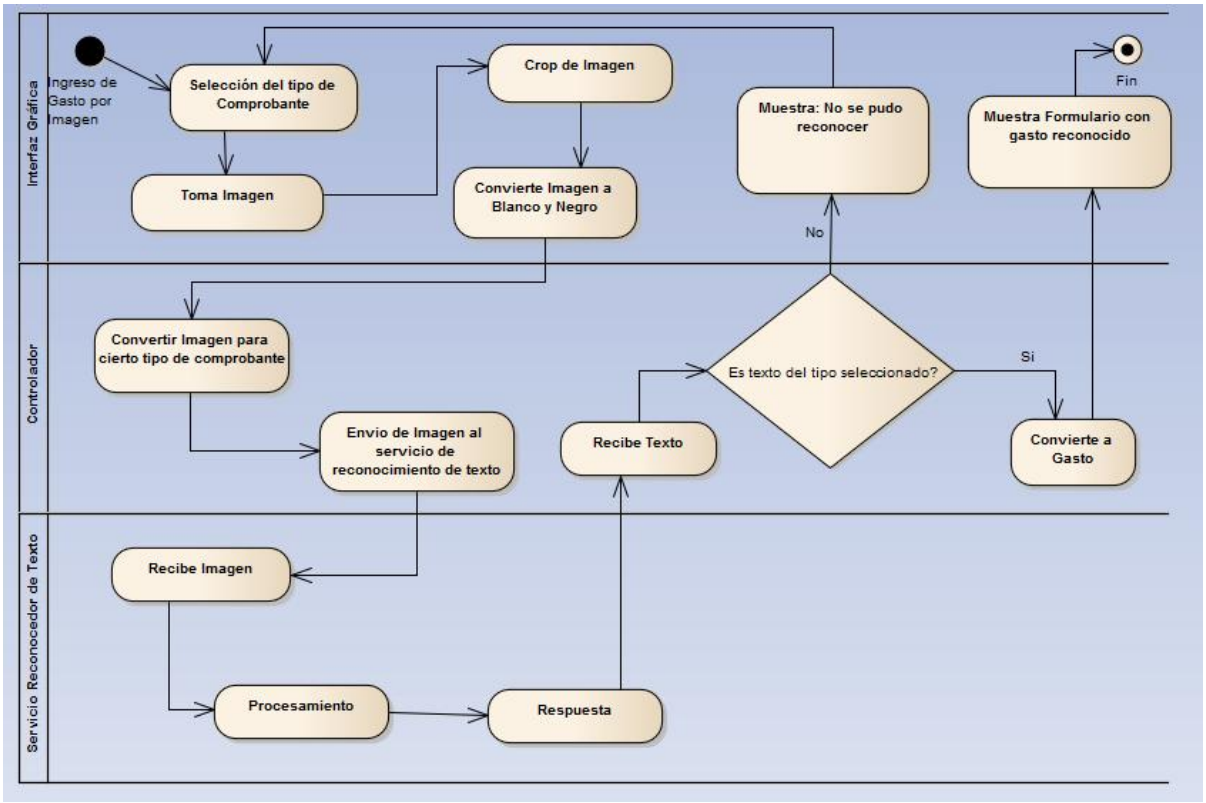

Figura 5-3: Diagrama de actividad – reconocimiento de texto

Usabilidad

Por último, para favorecer la usabilidad de la aplicación, se buscó que todo tipo de interacción con la capa de servicios de la aplicación se realice de manera asincrónica, lo que permite no bloquear la interfaz de usuario. De esta manera, se le puede mostrar al usuario ventanas y mensajes de qué está sucediendo en la aplicación mientras se ejecutan las tareas que requieren mucho procesamiento e interacción con el *backend*.

Performance

En lo que respecta a performance, se realizaron las siguientes decisiones de diseño:

- Cacheo de rubros: para obtener los posibles rubros a los cuales se le puede asociar un gasto, se decidió tener dicha información en el backend, y consultar esa

información la primera vez que se los necesita. En caso de que no sea posible comunicarse con el servidor, se utilizará la última versión de los rubros.

- Manejo de imágenes: para que el *upload* de las imágenes al servicio de reconocimiento de texto sea lo más rápido posible se decidió:
    - Implementar una funcionalidad que permita cortar la imagen, para que el usuario omita las esquinas innecesarias de la imagen.
    - Convertir la imagen a blanco y negro, disminuyendo el peso de la misma, además de facilitar el reconocimiento de texto.
- Manejo de gastos: para facilitar el ingreso de los gastos, se tomaron las siguientes decisiones:
    - En caso de que el dispositivo no tenga acceso a internet, el gasto se guarda de manera local, con una bandera que indica que no fue subido al *backend*.
    - Cada vez que se abre la aplicación, la aplicación consulta la base de datos local y valida que no existan gastos que estén *offline.*
    - En caso de que existan gastos que estén *offline* y se tenga acceso a internet, la aplicación de manera asíncrona sube esos gastos al backend y actualiza la base de datos local.
    - Por otro lado, en la aplicación local solo se guardan aquellos gastos que, o bien no se haya ingresado al backend, o sean del mes y año actual.
    - Tomando lo anterior en consideración, cuando el usuario consulta sus gastos a través del listado puede ocurrir que:
        - No se tenga acceso a internet, por lo cual se mostrarán sólo los gastos del mes y año actual, además de los gastos que no se han subido a la web.
        - Si el usuario tiene acceso a internet, la aplicación busca nuevos gastos en el *backend* y actualiza la base de datos con dichos gastos.
- Manejo local de RUT y líneas de CUTCSA:
    - Se utilizaron el RUT y los números de parada y línea para determinar otra información de los recibos, para que el reconocimiento de texto sea lo más completo posible. Para favorecer la performance, se tomaron las siguientes decisiones de diseño:

- Por un lado, se creó una tabla que mapea el número de RUT con el nombre de la empresa ubicada en el *backend*. De esta manera, se genera una base de datos de RUT-Nombres, que será formada por los usuarios. En caso de que algún usuario ya haya ingresado un gasto de cierta empresa, dicho nombre podrá ser consultado y obtenido.
- Luego, se asume que el usuario realiza muchas compras sobre un mismo local, por lo cual obtener el nombre de dicho local al *backend* sería poco eficiente. Por lo cual se decidió crear una entidad en la base de datos local para guardar los nombres de los gastos generados por el usuario. De esta manera, si el usuario repite cierto gasto en cierto local, obtener el nombre de dicho gasto sólo consiste en consultar la base de datos local.
- De manera análoga se obtiene la información para los boletos de CUTCSA a partir del número de parada y el número de línea

5.3.4. Alternativas Consideradas

En lo que respecta a las alternativas consideradas para el diseño arquitectónico del sistema, se tuvieron en cuenta dos diseños alternativos para el desarrollo de las funcionalidades web. Por un lado, se intentó crear un sistema completamente independiente al de Paganza hoy en día.

Se analizó la posibilidad de crear una nueva aplicación web que mantenga el mismo *look and feel* que tiene Paganza en su aplicación actual, pero incluiría solamente las nuevas funcionalidades contempladas en este proyecto. Por un lado, esta posibilidad le permitía al equipo desarrollar de una manera más independiente del sistema de Paganza, y no presentaba la curva de aprendizaje que sí tiene la solución seleccionada, lo cual permitía comenzar el desarrollo de manera más rápida. Si bien presentaba las ventajas mencionadas, se descartó dicha posibilidad debido a que, en primer lugar, el equipo de Paganza debía construir una gran cantidad de servicios para que la nueva aplicación consuma, a modo de obtener todos los datos que se necesitan de los usuarios, como lo son sus credenciales, sus facturas, etc. En segundo lugar, esta nueva aplicación implicaría mayores gastos para el cliente, debido a que sería necesario obtener un nuevo dominio, gastos relacionados al

almacenamiento de datos así como también *hosting* y gastos relacionados al desarrollo de los servicios necesarios para la transferencia de datos.

Teniendo en cuenta la posibilidad anterior, finalmente se decidió que las funcionalidades a desarrollar se implementarían sobre la aplicación existente de Paganza. Con esto en mente, se buscó crear un diseño arquitectónico que le permita a los integrantes del equipo la independencia suficiente del código de Paganza para facilitar el desarrollo y minimizar el impacto sobre la solución actual. Por lo cual, el diseño se enfocó en la utilización de componentes auto contenidos. Si bien este enfoque se mantuvo para el diseño de la arquitectura final, en un principio se busco que toda la lógica correspondiente a una funcionalidad en particular estuviese contenida dentro de dicho componente, desde las entidades del sistema, hasta la capa de presentación. Finalmente se desechó dicha idea debido a que Paganza en un futuro deseaba utilizar las funcionalidades y las entidades desarrolladas en este proyecto con otros fines, sobre todo para integración con nuevas funcionalidades propuestas por ellos.

5.4. Vistas de Módulos

5.4.1. Vista de Descomposición/Uso del Sistema

Para brindar una mayor claridad sobre la representación del sistema, se crean dos diagramas. El primer diagrama muestra cómo se descompone el sistema web, mientras que por otro lado se muestra la descomposición de la aplicación móvil Figura 5-4 y Figura 5-5 respectivamente.

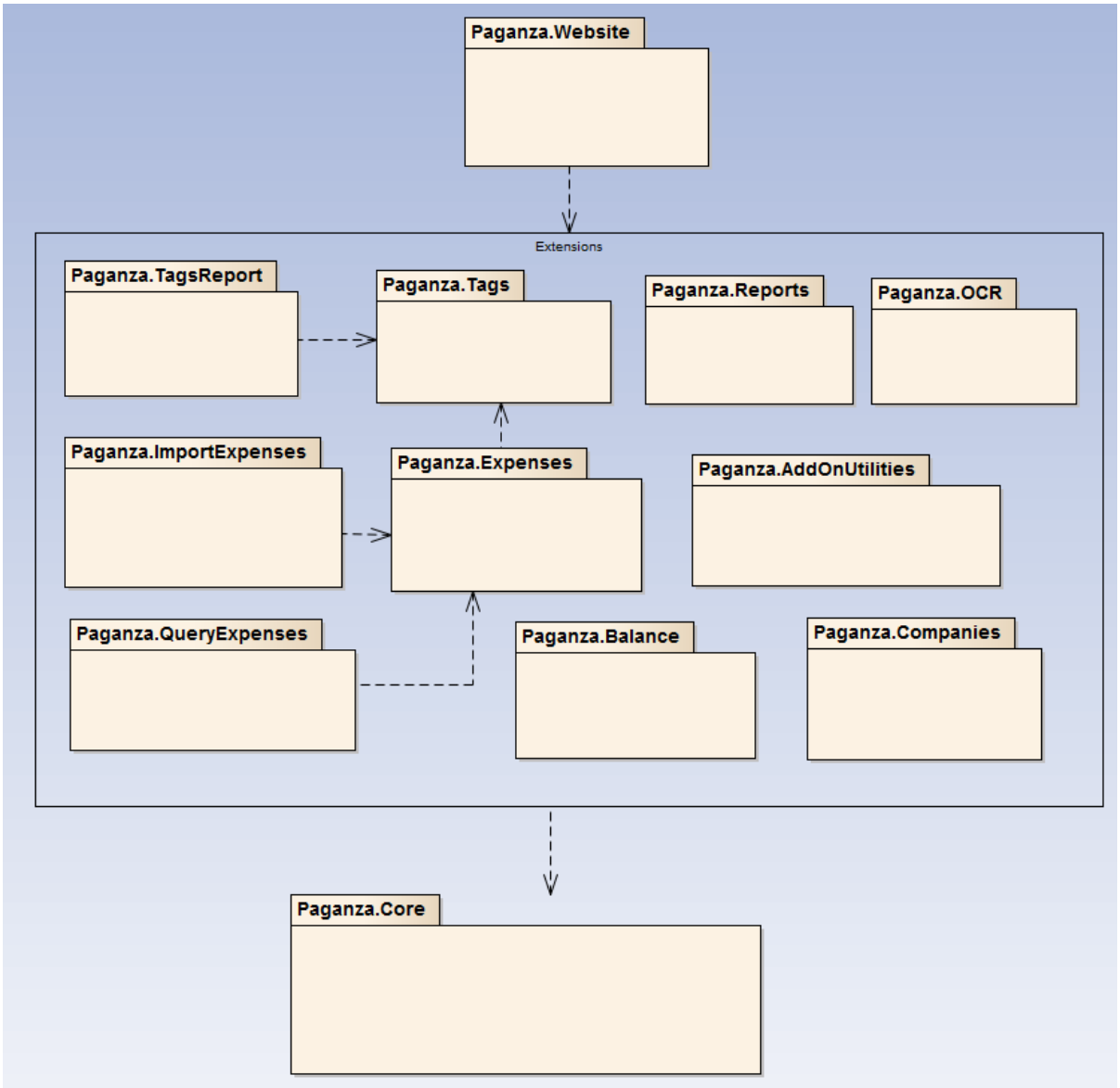

Figura 5-4: Diagrama de Paquetes – Sistema Web

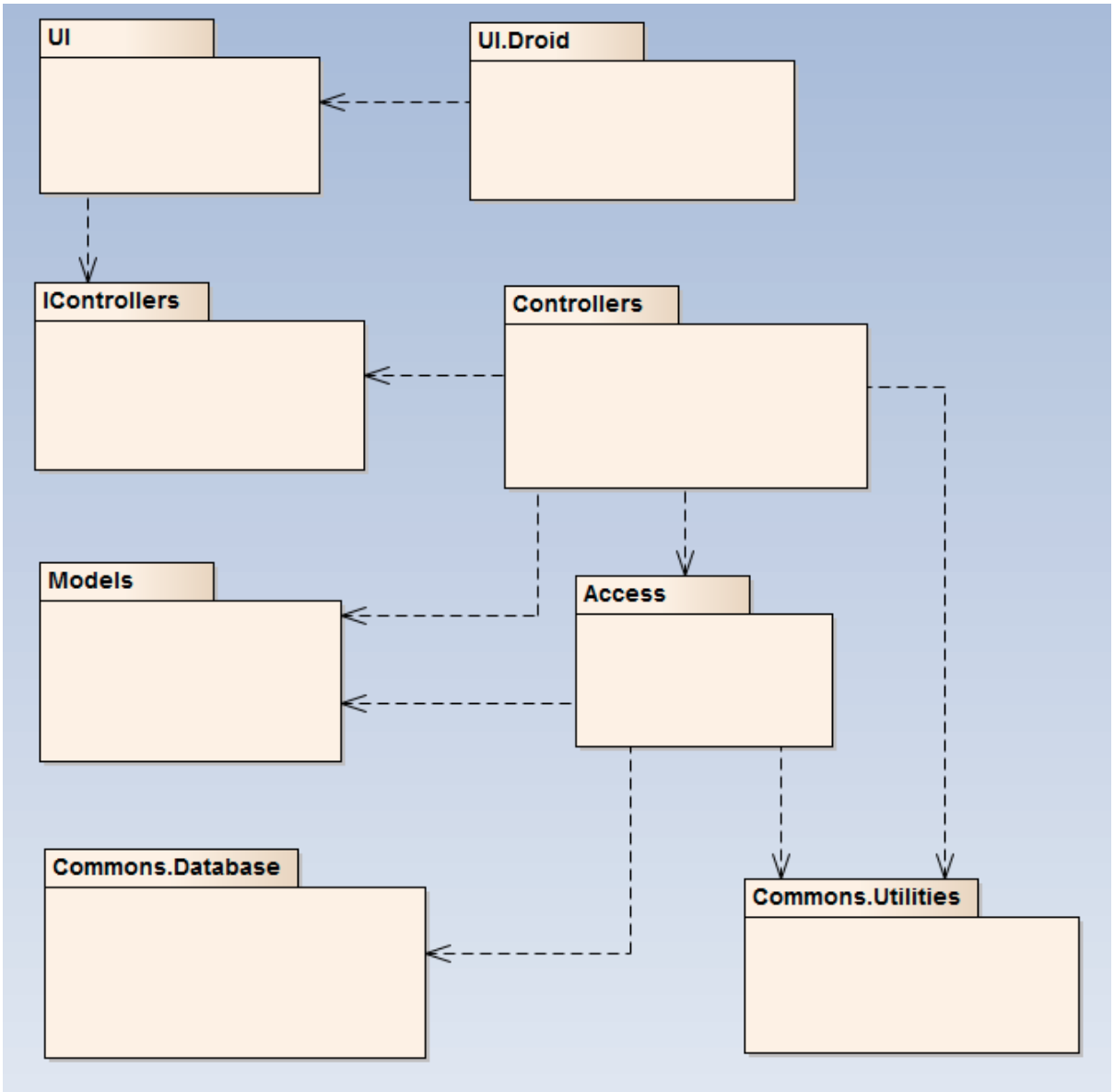

Figura 5-5: Diagrama de Paquetes – Sistema Móvil

Catálogo de Elementos

- Paganza.Website: paquete que contiene la interfaz gráfica para el sitio web. En dicho paquete se encuentra la lógica original de Paganza para su web actual, así como también la nueva lógica agregada debido a las extensiones realizadas.
- Paganza.Companies: paquete donde se define la funcionalidad del manejo de empresas.
- Paganza.TagsReport: paquete donde se define la funcionalidad del reporte de etiquetas.

- Paganza.Tags: paquete donde se define la funcionalidad de etiquetas, siendo este la creación/edición de etiquetas, así como también la consulta de las mismas.
- Paganza.ImportExpenses: paquete donde se define la funcionalidad de importación de gastos.
- Paganza.Expenses: paquete donde se ubica la lógica correspondiente al manejo de gastos.
- Paganza.QueryExpenses: paquete donde se ubica la funcionalidad que permite consultar los gastos de un usuario.
- Paganza.Balance: paquete donde se encuentra la creación del reporte de Balance o Billetera.
- Paganza.OCR: paquete donde se realiza el manejo de reconocimiento de texto para la creación de gastos desde la aplicación móvil.
- Paganza.AddOnUtilities: paquete donde se ubican todas las funcionalidades de utilidad para los componentes desarrollados.
- Paganza.Core: paquete donde se ubican las clases del dominio de Paganza, y donde se agregaron nuevas entidades debido a las extensiones desarrolladas.
- UI: paquete que contiene toda la lógica base relacionada con la interfaz de Usuario. Aquí se ubican las clases de la aplicación orientadas a la interfaz de usuario que son comunes para todas las plataformas.
- UI.Droid: paquete que contiene la lógica de interfaz de usuario orientada a la plataforma Android.
- IControllers: paquete que define los contratos necesarios para la interacción entre los controladores del sistema y la interfaz gráfica.
- Controllers: paquete que contiene la lógica de interacción entre la interfaz y el modelo de datos. Será el encargado de recibir los pedidos de la interfaz, procesarlos y devolver una respuesta a la interfaz. También será el encargado del manejo de errores de la aplicación.
- Access: será la capa de acceso a los datos del sistema así como el acceso al backend.
- Models: paquete donde se ubicarán los modelos de datos del sistema, siendo estos todas las clases del dominio.

- Commons.Database: paquete donde se ubican las clases encargadas de estandarizar el manejo de la base de datos local de la aplicación. Se definirán interfaces a implementar para la interacción con la base de datos (guardar, crear, borrar, etc.) así como una clase abstracta que definirá los atributos básicos que deberán tener las clases del dominio.
- Commons.Utilities: paquete de utilidades de la aplicación.

Justificaciones de Diseño

Como se había mencionado previamente, el enfoque a la hora de diseñar el sistema y separar la funcionalidad fue sobre la creación de componentes separados por funcionalidad. Como se puede ver, se crearon diferentes paquetes para cada una de las funcionalidades a desarrollar, permitiendo de esta manera aumentar la mantenibilidad del sistema debido a la claridad y a la separación de responsabilidades, creando interfaces bien definidas de comunicación con cada componente. Además, esta separación en componentes permitió la reutilización de código a lo largo del proyecto, permitiendo utilizar los componentes desarrollados para crear nuevas funcionalidades (como por ejemplo los gastos y la importación de los mismos) o para la aplicación móvil.

Entrando más en detalle, se creó un paquete de Utilidades (Paganza.AddOnUtilities) para agrupar la funcionalidad que es utilizada en todos los componentes. Dicha funcionalidad engloba manejos de excepciones, control de parámetros para todas las interfaces expuestas y clases genéricas para manejo de errores. Para encontrar un desglose de las decisiones de diseño sobre la plataforma web tomadas para las funcionalidades más importantes, ver el Anexo 9.5.

Por otro lado, la aplicación móvil se puede separar en 4 capas:

- Capa de presentación: incluye los paquetes
    - PaganzaMobile.UI.Droid
    - PaganzaMobile.UI
- Capa de servicios:
    - PaganzaMobile.IControllers
    - PaganzaMobile.Controllers

- Capa de lógica de negocio:
    - PaganzaMobile.Access
    - PaganzaMobile.Models
    - PaganzaMobile.Commons.Database
- Capa transversal:
    - PaganzaMobile.Commons.Utilities

La separación de dichas secciones en capas permite que el equipo pueda desarrollar de manera paralela, basando la comunicación entre la lógica del negocio y la interfaz gráfica a través de interfaces bien definidas en el paquete IControllers.

5.4.2. Vista de Descomposición/Uso de Extensiones

Representación Primaria

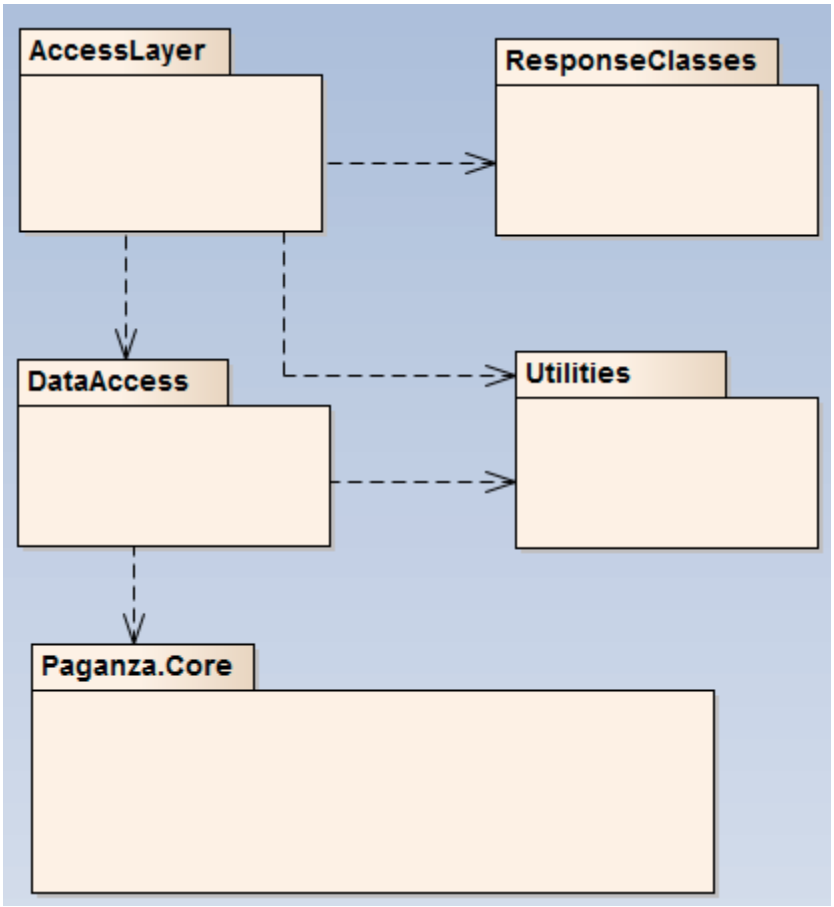

Figura 5-6: Diagrama de Paquetes - Extensiones

Diagrama de Clases

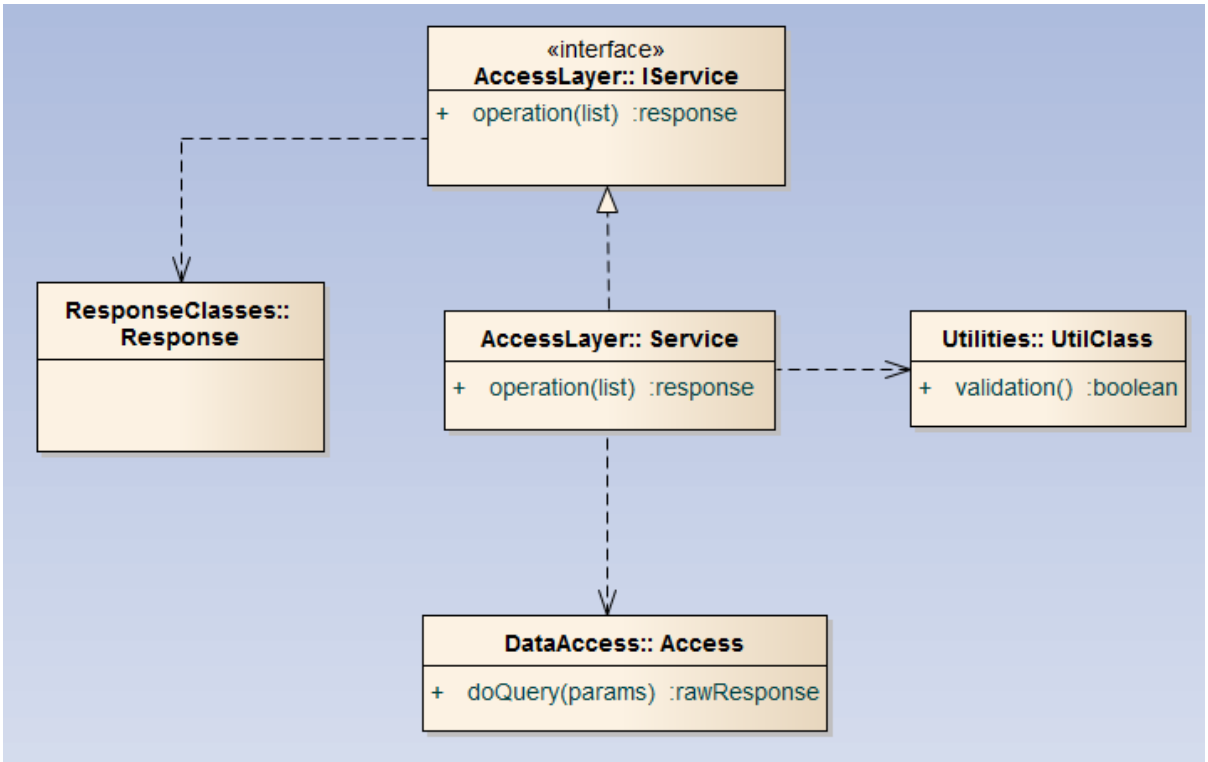

Figura 5-7: Diagrama de Clases - Extensiones

Diagrama de Secuencia

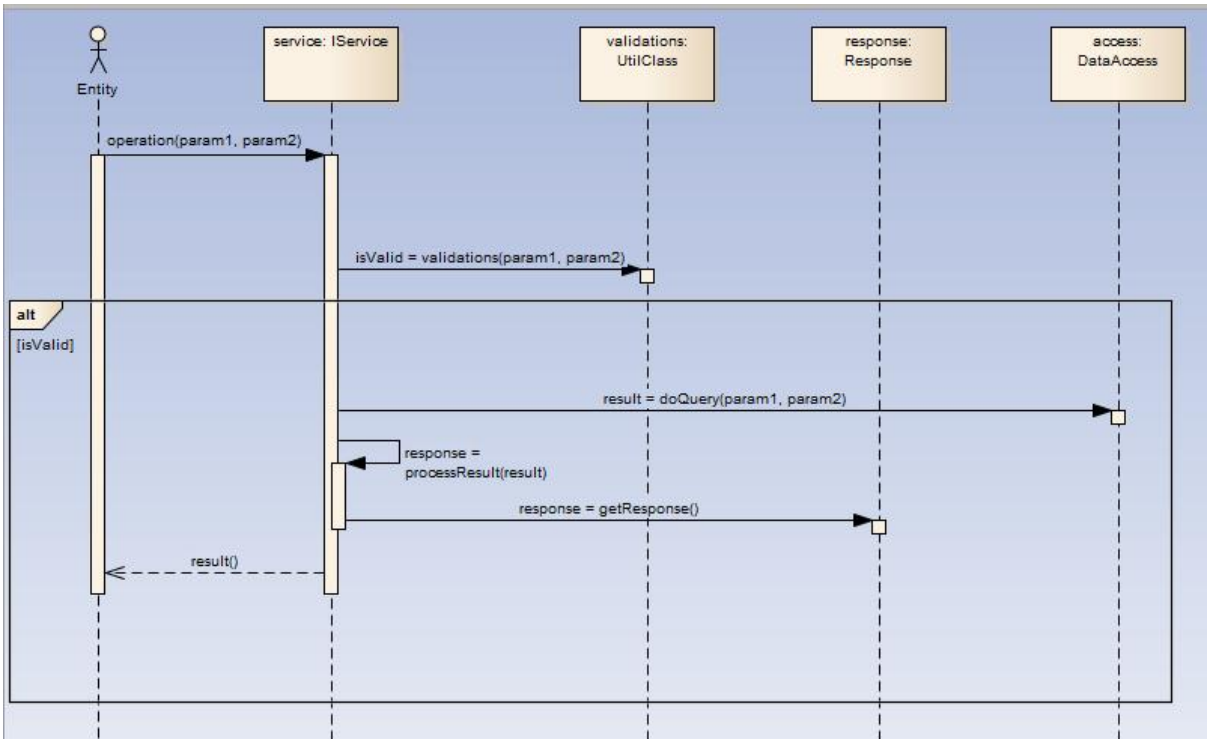

Figura 5-8: Diagrama de Secuencia - Extensiones

En la Figura 5-8, se puede ver la interacción genérica que el sistema existente tendrá con los componentes desarrollados. En primer lugar, la entidad que realiza un pedido a dicho componente invocará cierta función a través de una de las interfaces provistas por el componente. Dentro de la implementación de dicha interfaz, el componente realiza las validaciones necesarias sobre los parámetros que se recibieron. En caso de que los parámetros sean válidos, el componente pasará a realizar la consulta correspondiente sobre la base de datos, para luego devolver los datos crudos de dicha consulta. Una vez recibidos esos datos, el componente realizará una conversión de los datos crudos resultantes de la consulta a clases DTO (*Data Transfer Object)* las cuales serán el resultado que reciba la entidad que realizó el pedido en primer lugar.

Catálogo de Elementos

- AccessLayer: Capa de acceso al componente.
- DataAccess: Capa de acceso a los datos del sistema de Paganza.
- Utilities: Capa vertical que deberá contener todas las operaciones extra que se realizan en el componente, que son de utilidad para el sistema.
- ResponseClasses: Capa donde se ubican las clases de respuesta o DTO del componente.
- IService: interfaz que se proveerá desde el componente, para atender los pedidos. Es quien define el contrato en la interacción entre el componente y el sistema de Paganza. Esta clase se encontrará dentro de la capa AccessLayer.
- Service: clase que implementará la interfaz provista. Esta clase se encontrará dentro de la capa AccessLayer.
- UtilClass: clase de ejemplo para la implementación de utilidades. Se encuentra dentro de la capa Utilities.
- ResponseClass: clase que contiene la información de las entidades de Paganza de una manera simple y fácil de mostrar.
- DataAccess: clase que interactúa con la base de datos de Paganza, y que devuelve el resultado de la consulta para que esta sea procesada. Esta clase hereda de la clase BusinessComponent, para poder simular y realizar las consultas a la base de datos tal como lo hace Paganza.

Justificaciones de Diseño

Para las extensiones a realizar, se buscó crear un diseño que permita al equipo una base de la cual partir a la hora de desarrollar. Por un lado, se crea la capa de acceso al componente, donde se encontrarán las interfaces que serán provistas, así como también la implementación de las mismas. Dicha capa será el punto de acceso para que las entidades externas puedan consumir los servicios del componente.

Estas interfaces permiten definir los contratos que se tendrán para interactuar con las funcionalidades a desarrollar, permitiendo que quien necesite hacer uso de dicha funcionalidad sepa qué métodos existen y de qué manera se pueden invocar. Por otro lado, estas interfaces permiten que el equipo desarrolle de forma paralela entre el desarrollo de funcionalidad de las entidades del sistema así como el desarrollo de la interfaz de usuario. Por último, estas interfaces definen el contrato utilizado para la extensión desarrollada, lo que permite que Paganza pueda modificar la funcionalidad base en un futuro, siempre y cuando se acoplen a las interfaces definidas. Esto genera puntos de variación de las nuevas funcionalidades a lo largo del sistema, favoreciendo la mantenibilidad del mismo.

En segundo lugar, la capa de acceso a datos tendrá una fuerte dependencia al *Core* de Paganza, debido a que será necesario acceder y utilizar las entidades del negocio para acceder a los datos. Las clases que se encuentran en dicha capa deberán seguir el comportamiento y la estructura de las clases del negocio de Paganza, para mantener la coherencia en el sistema, y realizar todas las operaciones de la misma manera que lo hace Paganza. Para esto, las clases que realicen el acceso a datos *per se*, deberán heredar de la clase BusinessComponent, ubicada en Paganza.Core. A partir de dicha clase, se puede realizar el acceso a base de datos y obtener la información necesaria.

En lo que respecta a las utilidades del componente, se busca crear una capa donde se realicen todo tipo de validaciones del componente, manejo de excepciones, etc. Al comenzar con el desarrollo de los componentes y con el transcurso del proyecto, se busca desarrollar una capa de utilidades que sea reutilizable para todos los componentes. En caso de que dicha capa comience a ser reutilizada, se creará un nuevo componente para ubicar

estas funcionalidades de utilidades, para evitar código duplicado, además de centralizar el código.

Por último, dentro de la capa de respuesta se encontrarán las clases que se utilizarán para responder a las entidades que realicen los pedidos. Debido a que las entidades de Paganza son complejas, además de que no están orientadas a ser utilizadas por los componentes a desarrollar, se deberán crear clases que simplifiquen la información de las entidades de Paganza y permitan retornar solo la información necesaria.

5.5. Vista de Componentes

En la siguiente sección se describe el sistema a nivel de componentes, mostrando de qué manera se comunican los diferentes módulos del sistema. Para generar diagramas de componentes claros y entendibles, se decidió separar dichos diagramas según la plataforma, creando un diagrama de componentes para la plataforma web (Figura 5-9), así como también creando un diagrama para la plataforma móvil (Figura 5-10).

5.5.1. Plataforma Web

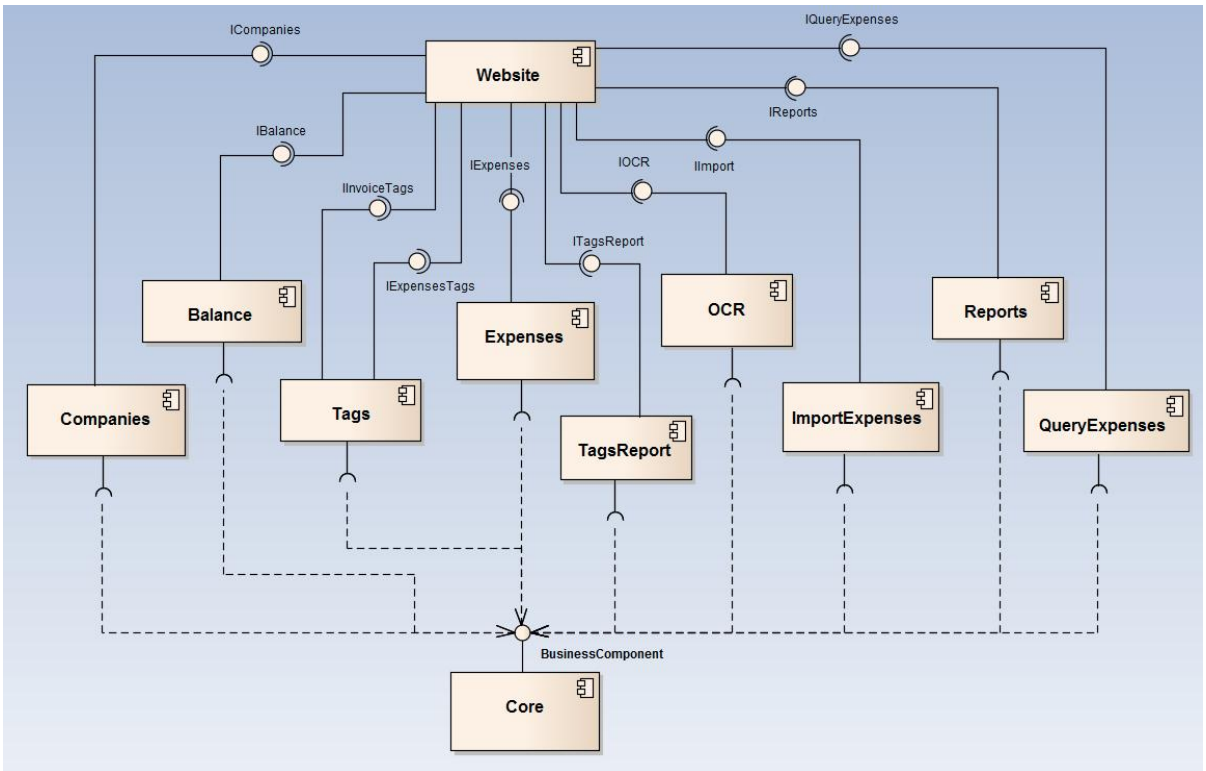

Figura 5-9: Diagrama de Componentes – Plataforma Web

5.5.2. Plataforma Móvil

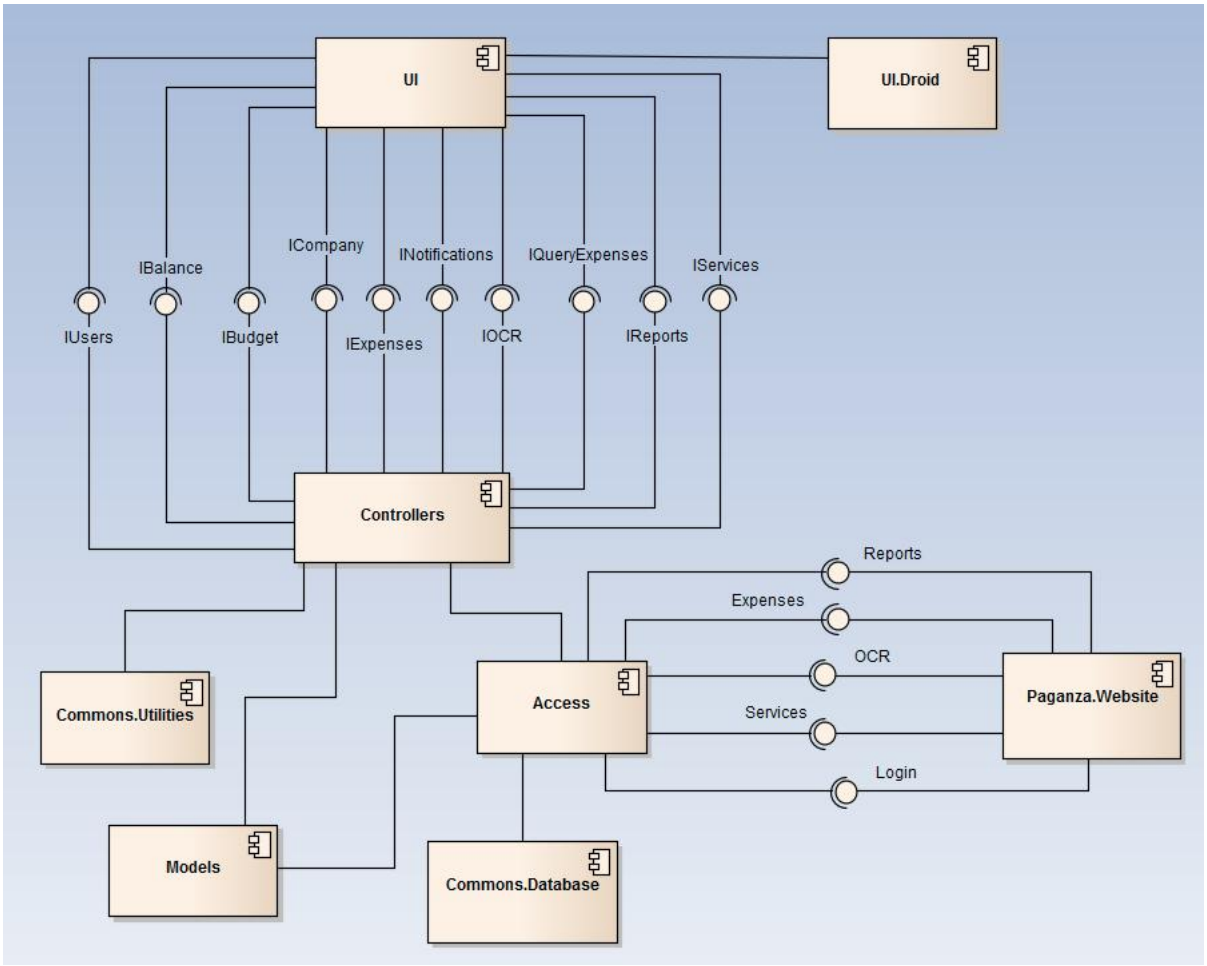

Figura 5-10: Diagrama de Componentes – Plataforma Móvil

5.5.3. Catálogo de Elementos – Plataforma Web

- Website: componente que contiene la interfaz gráfica para el sitio web. En él, se realizan todos los pedidos hacia los nuevos componentes desarrollados. Requiere, como se muestra en el diagrama de componentes, que se implementen las interfaces correspondientes a cada funcionalidad.
- Companies: componente donde se define la funcionalidad del manejo de empresas. Implementa la interfaz ICompanies, encargada de manejar los pedidos correspondientes a la asociación de Empresas con números de RUT.
- TagsReport: componente donde se define la funcionalidad del reporte de etiquetas. Implementa la interfaz ITagsReport para permitir el análisis de los gastos diferenciados por etiquetas.

- Tags: componente donde se define la funcionalidad de etiquetas, siendo esta la creación/edición de etiquetas, así como también la consulta de las mismas. Se implementan dos interfaces IInvoiceTags y IExpensesTags, siendo estas las responsables del manejo de etiquetas para facturas y para gastos respectivamente.
- ImportExpenses: componente que implementa la interfaz IImport que permite importar gastos desde un archivo Excel.
- Expenses: componente donde se ubica la lógica correspondiente al manejo de gastos. Implementa la interfaz IExpenses que permite el manejo de gastos.
- QueryExpenses: componente donde se ubica la funcionalidad que permite consultar los gastos de un usuario, implementando la interfaz IQueryExpenses.
- Balance: componente donde se encuentra la creación del reporte de Balance o Billetera. En este componente se ofrece la interfaz IBalance que permite obtener el reporte de Balance para cierto usuario.
- OCR: componente donde se realiza el manejo de reconocimiento de texto para la creación de gastos desde la aplicación móvil, implementando la interfaz IOCR.
- Core: componente donde se ubican las clases del dominio de Paganza, y donde se agregaron nuevas entidades debido a las extensiones desarrolladas. Dicho componente ofrece la interfaz BusinessComponent, interfaz que define el comportamiento básico para las clases que realizan la interacción con la base de datos.

5.5.4. Catálogo de Elementos - Plataforma Móvil

- UI: componente que contiene toda la lógica base relacionada con la interfaz de usuario. Aquí se ubican las clases de la aplicación orientadas a la interfaz de usuario que son comunes para todas las plataformas.
- UI.Droid: componente que contiene la lógica de interfaz de usuario orientada a la plataforma Android.
- Controllers: componente que contiene la lógica de interacción entre la interfaz y el modelo de datos. Será la encargada de recibir los pedidos de la interfaz, procesarlos y devolver una respuesta a la interfaz. También será la encargada del manejo de errores de la aplicación.

- Access: componente donde se ubica la lógica que permite obtener datos desde el *backend*, así como también obtener datos desde la base de datos local.
- Models: componente donde se ubicarán los modelos de datos del sistema, siendo estos todas las clases del dominio.
- Commons.Database: componente donde se ubican las clases encargadas de estandarizar el manejo de la base de datos local de la aplicación.
- Commons.Utilities: componente de utilidades de la aplicación.
- Paganza.Website: componente donde se publican los diferentes servicios necesarios para que la aplicación móvil pueda interactuar con el *backend*.
- Expenses: servicio que permite crear gastos, así como también consultarlos
- Company: servicio que permite obtener un nombre a partir de un número RUT
- Reports: servicio que se utiliza para obtener el reporte de gastos mensuales
- OCR: componente externo que recibe la imagen tomada para obtener el texto reconocido
- Services: servicio que se utiliza para obtener los posibles rubros a los cuales se le puede asociar un gasto

5.6. Vista de Despliegue

Representación Primaria

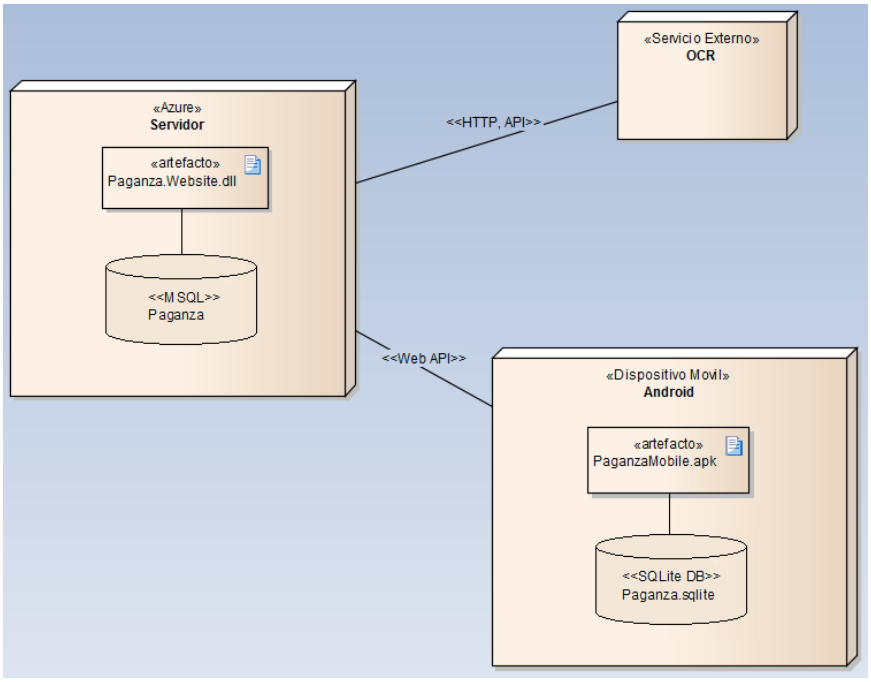

Figura 5-11: Diagrama de Despliegue

Catálogo de Elementos

- Servidor: servidor ubicado en *Azure* donde se ubica físicamente la aplicación desplegada.
- Paganza.Website.dll: librería binaria generada para desplegar la aplicación en el servidor.
- Paganza (MSQL): base de datos ubicada en *Azure*, utilizando el motor de base de datos Microsoft SQL Server.
- Android: dispositivo móvil donde se instala la aplicación generada.
- PaganzaMobile.apk: instalador generado para desplegar la aplicación en el dispositivo.
- Paganza.sqlite (SQLITE): base de datos ubicada en el dispositivo móvil, utilizando SQLite como motor de base de datos.
- Servicio Externo: se utiliza un servicio desarrollado por terceros para realizar el reconocimiento de texto. Dicho servicio se encuentra ubicado en un servidor externo, y la comunicación con dicho servicio se realiza a través de una API provista.

5.7. Plan de Implementación Web

Debido a que el proceso de desarrollo incluye la construcción de extensiones, el proceso de implementación deberá permitir:

1. Rápida generación de una extensión para ser validada por el cliente.

2. Integración y prueba con las entidades de existentes sin necesidad de incluir la extensión dentro de Paganza.

3. Se debe tener en cuenta los estilos que se usan actualmente en Paganza, para que las extensiones se encuentren alineadas con la aplicación existente.

4. Observando las necesidades del cliente y la metodología a implementar, se deberán generar extensiones de manera incremental, y en cada etapa del proceso se le agregará lógica más compleja hasta tener un componente integrable con Paganza.

Las etapas de evolución de las extensiones son las siguientes:

Código Estático: el primer prototipo de la extensión será realizado sólo con código JavaScript y HTML/CSS estático. Se reutilizarán los estilos y las librerías ya existentes en Paganza, y se comenzará a desarrollar un prototipo con sólo código estático de *frontend*. Esto permitirá comenzar a validar los requerimientos con el cliente de forma rápida y fácil, mientras que se puede comenzar a diseñar el código de backend y las modificaciones necesarias.

Prototipo de Backend: Una vez validado el requerimiento, se comienza con la etapa de desarrollo del componente de *backend*. El desarrollo de estos prototipos incluye la integración con el sistema de Paganza. Esto se realizará creando nuevas clases del dominio de Paganza para este componente en particular, utilizando la base de datos del ambiente de desarrollo local y realizando DataBase First con Entity Framework. Esto permite realizar pruebas de concepto sobre la estructura de la base de datos actual de Paganza sin impactar sobre el sistema en ambiente de producción.

Aplicación MVC: Luego de tener validado el *frontend*, y tener un prototipo funcional del componente de *backend*, se comenzará a trabajar en una Aplicación MVC nueva, independiente de Paganza. En ella, se integrará el código de *frontend* y se comenzará a realizar la integración con el componente de *backend* desarrollado. Se realizarán ajustes tanto en *frontend* como en *backend*, dependiendo del resultado de la integración.

Integración con Paganza.Core: Luego de finalizada la integración de ambos prototipos se integrará el código de *backend* de Paganza. En caso de haber realizado cambios sobre la base de datos, se consultará primero con Paganza para saber la mejor manera de realizar los cambios propuestos, para poder implementarlos en el ambiente de producción en un futuro.

5.8. Plan de implementación móvil

La implementación de la aplicación móvil está basada en la construcción de prototipos funcionales y pruebas de concepto. Por este motivo, el principal objetivo del desarrollo sobre la plataforma móvil, es desarrollar funcionalidades que aporten valor en la experimentación, ya sea desde lo conceptual o desde lo técnico. Otro aspecto que favoreció esta decisión fue la incertidumbre por parte del cliente frente al futuro estratégico de estas

nuevas funcionalidades. Paganza, al momento del desarrollo de la aplicación móvil, no había tomado una decisión con respecto a incluir funciones relativas a la gestión de gastos dentro de su actual aplicación, o crear una nueva aplicación con este fin.

Por los motivos explicados anteriormente, se decidió desarrollar las funcionalidades como prototipos que serán entregados como resultado del proyecto. De esta manera, se agilizó el proceso de implementación y liberación para poder ejecutar la aplicación mediante pruebas beta. También gracias a esta independencia se consiguió implementar funcionalidades diferentes desde lo conceptual a las funcionalidades móviles de Paganza.

Por otro lado, al realizar pruebas desde el aspecto técnico, respondiendo a una iniciativa de Paganza, se utilizó la herramienta de desarrollo de interfaz de usuario *cross-platform* Xamarin Forms. Como la aplicación móvil existente se encuentra desarrollada en Xamarin y recientemente fue liberada la herramienta Xamarin Forms, se decidió implementar las funcionalidades utilizándola. Xamarin Forms es un enfoque de desarrollo dentro de Xamarin que permite desarrollar aplicaciones nativas compartiendo gran parte del código. Como parte del proceso de implementación, el equipo realizó una investigación de la herramienta ya que era desconocida para todos los integrantes. Esta investigación, junto con el producto desarrollado y las conclusiones del desarrollo fueron entregadas a Paganza como transferencia de conocimiento. Un análisis más profundo y concreto sobre Xamarin y Xamarin Forms como tecnologías y sus resultados de aplicación en el proyecto pueden ser encontrados en el Anexo 9.6.

Dentro del proceso de desarrollo móvil se realizaron pruebas de concepto técnicas cuya finalidad fue garantizar la factibilidad de implementación de las funcionalidades necesarias. Estas pruebas fueron satisfactorias, aunque en caso de haber encontrado impedimentos tecnológicos ocasionados por la inmadurez del producto, pudieron haberse implementado utilizando Xamarin.Android y Xamarin.iOS que son utilizados por Paganza actualmente. Las pruebas de concepto desarrolladas durante el proyecto fueron las siguientes:

Geolocalización: Se realizó una aplicación con manejo elemental de las características de posicionamiento provistas por el dispositivo. Esta prueba incluye la obtención de las coordenadas geográficas actuales del dispositivo, mostrándolas en un componente visual.

También se incluye la utilización de marcadores para indicar puntos referenciados geográficamente.

Implementación de interfaz de usuario: Se realizó una prueba para validar la utilización de los diferentes componentes de interfaz de usuario como botones, campos de ingreso de texto, selección de opciones, etc. Dentro de esta implementación se utilizaron las dos formas que provee Xamarin Forms para desarrollar la interfaz de usuario, siendo estas XAML o por código.

Consumo de APIs REST: Debido a que para establecer comunicaciones entre el servidor y la aplicación móvil se utilizarían servicios REST, se desarrolló una aplicación básica que realizará solicitud a una API e interpretará los resultados.

Manejo de base de datos local: La necesidad de almacenar información en el entorno local determinó la creación de una base de datos en SQLite. Para asegurar su correcto funcionamiento se implementaron las operaciones de lectura y escritura en la base de datos.

Utilización de cámara: Como se necesitaba interpretar datos a partir de imágenes, como por ejemplo los códigos QR de los comprobantes o texto incluido en los mismos, se realizó una prueba que utilizaba la cámara para reconocer códigos QR o enviarla a un servidor para análisis OCR.

Notificaciones: Dentro de los requerimientos analizados se incluía la posibilidad de utilizar notificaciones para alertar al usuario ante la ocurrencia de ciertos eventos, se realizó una prueba para utilizar la funcionalidad de notificaciones locales.

Ciclo de vida de la aplicación: Durante la ejecución de aplicaciones móviles ocurren eventos que afectan dicha ejecución, como la transición a segundo plano, la reanudación de la ejecución, etc. Para el correcto manejo de estos eventos se desarrolló una aplicación básica contemplando este comportamiento.

6. Procesos de apoyo

6.1. Gestión del proyecto

En este capítulo se tratarán los temas relativos a la gestión del proyecto, indicando la metodología y ciclo de vida elegidos, además del registro del esfuerzo y herramientas utilizadas. Por último se explicará la gestión de riesgos, indicando principales riesgos encontrados y sus planes de contingencia.

6.1.1. Metodología utilizada

Por las características del proyecto se utilizó una metodología ágil. El mismo se basa en un desarrollo iterativo e incremental donde los requerimientos y soluciones evolucionan durante el transcurso del proyecto [8], además de aprovechar el *feedback* continuo y la buena comunicación con el cliente.

Las características del proyecto son las adecuadas para la aplicación de una metodología con dicho ciclo de vida. Los objetivos del proyecto están claros pero no están detallados todos los requerimientos, ni determinadas todas las posibles soluciones. Otro punto a tener en cuenta es que actualmente el equipo es pequeño y altamente motivado, además de contar con un cliente altamente integrado al proyecto y con disponibilidad para reunirse frecuentemente con el equipo.

Para llevar a cabo el proyecto, se decidió adoptar el *framework* Scrum. Un marco de trabajo que genera un contexto relacional e iterativo de inspección y adaptación constante. Esto permite que los involucrados en el proyecto generen su propio proceso, lo que diferencia un *framework* de una metodología. Se resolvió utilizar Scrum ya que el equipo tenía previa experiencia con el *framework* y un miembro del equipo tiene experiencia como Scrum *Master*. Por otro lado, el cliente conoce el *framework* y se adaptó a su rol de *Product Owner* tanto como a las ceremonias determinadas.

Ya que el equipo nunca trabajó junto previamente utilizando Scrum, se decidió adoptar un enfoque de adaptación llamado Scrum orgánico [9], que utiliza como base el mismo *framework* iterativo incremental que propone Scrum. La idea es no hacer un uso de “Todo o

nada" de las ceremonias y artefactos, sino emplearlas a medida que el equipo lo encuentre necesario, y analizar su impacto paulatinamente.

6.1.2. Ceremonias y artefactos

A continuación se detallarán todas las ceremonias realizadas dentro del marco de trabajo, además de explicar las herramientas del *framework* y cómo se utilizaron para el proyecto.

Product Backlog: Es un listado de los ítems o características del producto a construir. Debe ser priorizado por el *Product Owner*. El equipo puede influir y expresar su opinión acerca de la priorización, pero es el *Product Owner* quien tiene la última palabra. La priorización se puede determinar por el valor de negocio de cada ítem, por el retorno de inversión o por importancia y riesgo de cada ítem.

El alcance en Scrum es variable. Esto se debe a que no es posible conocer de manera anticipada todas las características del sistema que se pretende construir. Por lo que el tiempo y el costo se fijan para el proyecto, mientras que la variable es el alcance.

A modo de manejo de contingencia, se aprovecha el tener un *backlog* priorizado y contar con un alcance variable. Esto determina que las historias menos prioritarias, que representan menor valor, son las que se pueden dejar de lado si hay imprevistos. Es decir que al respetar tiempos y costos, el alcance de menor prioridad es el que paga el precio de retrasos e imprevistos.

En este caso, el *backlog* fue priorizado tanto al inicio del proyecto como en su desarrollo. Esto permitió determinar un MVP, y continuar iterando sobre el *backlog* para tener claro con qué tareas continuar una vez terminado dicho MVP. Sobre la etapa final del proyecto, fue necesario negociar para definir si se comenzaba con historias nuevas o se mejoraba la calidad de las existentes con el objetivo de ponerlas en producción.

Sprint Backlog: Es un conjunto de ítems que fueron seleccionados para trabajar en ellos durante un cierto *sprint*. También forman parte las tareas que el equipo de desarrollo debe realizar para entregar un incremento funcional potencialmente entregable.

Se decidió agregar al *backlog* las tareas consideradas relevantes para el proyecto que se originen en las retrospectivas. Esto se refiere a tareas de investigación o mejoras en el proceso que indirectamente mejoran la calidad del producto que se está desarrollando.

Incremento funcional: El resultado de cada *sprint*. Esto se debe a que es una característica funcional nueva (o modificada) de un producto que está siendo construido de manera evolutiva. El producto crece con cada *sprint*.

Sprint: Refiere a las iteraciones en Scrum. Al ser un marco de trabajo incremental e iterativo, el producto se construye en incrementos funcionales de periodos cortos para obtener *feedback* frecuente. Scrum recomienda una duración entre 1 a 4 semanas, siendo entre 2 y 3 semanas lo más habitual. La volatilidad del negocio es lo que fija lo largo del sprint. No se recomienda modificar el largo del mismo muy frecuentemente, aunque se pueden hacer excepciones para observar si la calidad del trabajo mejora.

Para nuestro proyecto se decidió junto con el cliente comenzar con *sprints* de 3 semanas, ya que al comienzo los requerimientos estaban definidos y se trabajaba con el fin de poner el producto en producción. Luego se modificó a 2 semanas, ya que se necesitaba tener productos antes para poder validarlos con usuarios y el cliente.

Planning meeting (Planificación de Sprint): Al comienzo del *sprint* se realiza una reunión de planificación, donde el equipo se pone de acuerdo en qué tareas realizará. Esta reunión generalmente se divide en 2 partes, enfocadas en el qué y el cómo se realizara el trabajo.

En la primer parte de la reunión, se revisa el *backlog* determinando qué historias serán trabajadas por el equipo. Para esto deben estar presentes los interesados necesarios para evacuar cualquier tipo de duda. Si una historia no cuenta con la descripción necesaria o la definición de hecho, la historia no está en condiciones de comenzar. Para determinar cuántas historias se trabajarán, se debe realizar una estimación y considerando la velocidad se determina el contenido del *sprint*.

La segunda parte comienza cuando ya se tiene determinado el alcance del *sprint*. Es donde se determina un diseño de alto nivel de la solución para el sprint, identificando las actividades que el equipo debe llevar a cabo. Todas las historias del *sprint* deben detallarse

en tareas a completar, dando una visión completa y transparencia sobre el trabajo realizado.

Para el proyecto, la estimación de las historias se realizó tomando como unidad un punto. El mismo se determinó en una historia que se considerara atómica y sea sencillo la comparación con otras. Las reuniones de planificación contaban solamente con el equipo de desarrollo y el *Scrum Master*, ya que previamente estaban todas las historias detalladas y priorizadas para trabajarse. La velocidad promedio del equipo fue aproximadamente de 80 puntos, esto será detallado en el registro del esfuerzo en el Anexo 9.7.

Daily meeting (Scrum diario): Los objetivos de esta reunión son incrementar la comunicación, explicitar los compromisos del equipo y dar visibilidad de los impedimentos. Estas reuniones deben ser diarias y no tomar más de 15 minutos. Cada integrante del equipo de desarrollo debe expresarse con el fin de cumplir esos objetivos. Generalmente se responden 3 preguntas para abarcar esos temas. No se busca que la reunión sea un reporte de avance o estado, sino que es un espacio de comunicación dentro del equipo. Es aquí donde se ve el avance del *sprint*, indicando el estado de las tareas del *sprint backlog*.

Por las características del marco del proyecto, no fue posible que el equipo realizara reuniones diarias presenciales. Dichas reuniones se realizaban cada 2 o 3 días, por conferencia y presencialmente 1 vez por semana.

Sprint Review Meeting (Revisión de Sprint): Al finalizar el *sprint* se realiza una reunión de revisión, donde se evalúa el incremento funcional construido. En dicha reunión, el equipo y los interesados revisan el producto entregado, donde se pueden aceptar o rechazar las funcionalidades construidas. Todo el *feedback* obtenido, que determine un cambio o una nueva funcionalidad debe ser registrado e ingresado como ítem en el *backlog*.

Para el proyecto, las reuniones de revisión se llevaban a cabo al finalizar el sprint de ser posible. En caso de no ser posible, se enviaba un correo electrónico con el detalle de lo construido y luego con el *feedback* se hacían las modificaciones necesarias. En este caso la reunión presencial se realizaba lo antes posible.

Figura 6-1: Reunión de revisión con Paganza

Sprint Retrospective Meeting (Retrospectiva): El origen de esta reunión se basa en la mejora continua y las prácticas emergentes. El equipo reflexiona la forma de trabajo con el fin de mejorar sus prácticas. Dicha reunión ocurre inmediatamente después de la reunión de revisión. Mediante técnicas de facilitación y análisis de causas, se buscan fortalezas como oportunidades de mejora. Luego el equipo mediante consenso decide qué acciones concretas tomar.

En un principio las reuniones de retrospectiva tenían el fin de mejorar la comunicación entre el equipo y con el cliente, además de evaluar las actividades de Scrum realizadas como lo propone Scrum orgánico. Luego el objetivo de las reuniones fue analizar el proceso de desarrollo y calidad del producto, como por ejemplo agregando tareas de revisión y *testing*.

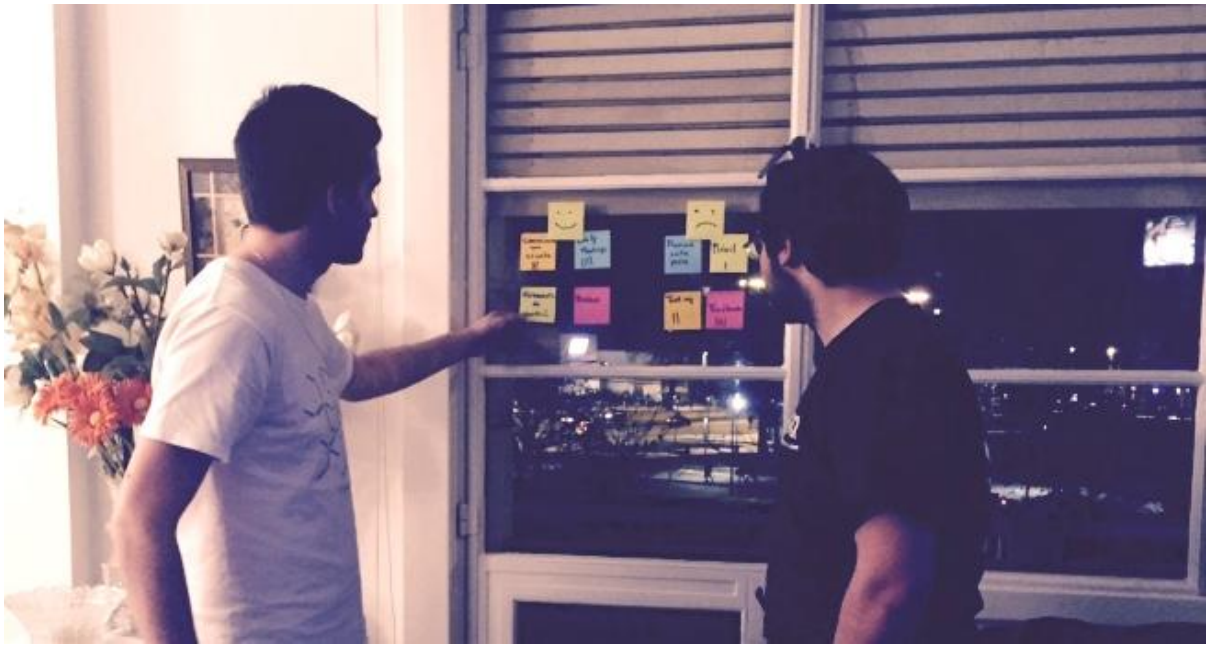

Figura 6-2: Ejemplo de retrospectiva

Sprint Refinement Meeting (Refinamiento): Es una actividad constante durante el *sprint*, o bien puede ser una reunión dedicada. Su objetivo es analizar los ítems en el *backlog*, para poder detallar aquellos en los que falte información. Además, se pueden estimar los siguientes ítems para ahorrar tiempo en la reunión de planificación.

El equipo realizaba esta reunión cuando se obtenía *feedback* de usuarios o del cliente, de esta manera se preparaban los siguientes ítems y se validaban lo antes posible.

6.1.3. Roles

Product Owner: Es el responsable del éxito del producto desde el punto de vista de los interesados, además de determinar la visión del producto, recolectar requerimientos y ser experto en el negocio. El foco debe ser maximizar la rentabilidad del producto, su principal herramienta es la priorización de los ítems del *backlog*.

Para el proyecto, el *Product Owner* determinado fue Paganza. El conocimiento de la metodología, el negocio y del rol facilitó esta asignación. Su principal actividad radicó en la priorización de los ítems del *backlog*, cómo de la validación de los incrementos generados.

Equipo de desarrollo: Conformado por los 4 integrantes del proyecto. Es el único responsable por la construcción y calidad del producto. El equipo de desarrollo debe ser

auto-organizado. No necesita un líder externo para asignar tareas ni determinar la forma con la que serán resueltos los problemas. El objetivo principal es transformar las funcionalidades comprometidas en software funcionando y con calidad productiva. Además de proveer una estimación de las tareas del *backlog* y asumir el compromiso de trabajo al inicio de cada *Sprint*.

La asignación de tareas en este proyecto fue determinada por la experiencia de cada integrante en el área solicitada, aunque se hizo el ejercicio de tomar tareas en las que el integrante no se siente cómodo y así mejorar el aprendizaje. Por otro lado, se tomó ventaja del conocimiento técnico del cliente para poder resolver tareas que necesitaban un mayor nivel de calidad.

Scrum Master: Es el rol encargado de ayudar al equipo en la utilización del *framework*, alcanzando el máximo nivel de productividad posible, velando por el correcto empleo y evolución de *Scrum*, así como facilitando su uso y protegiendo al equipo de distracciones y trabas externas al proyecto, además de asegurar la cooperación y comunicación en el equipo. Otra habilidad necesaria para el rol es el de facilitador, ayudando al equipo en las ceremonias de *Scrum*.

El rol fue tomado por un integrante del equipo, con una certificación y previa experiencia como *Scrum Master*. En el contexto de este proyecto, no se pudo contar con un *Scrum Master* dedicado completamente al rol, ya que se necesitaba como miembro del equipo de desarrollo, por lo que las tareas se centraron en la facilitación en distintas reuniones y el manejo de los artefactos de *Scrum*. A medida que avanza el proyecto, el rol es cada vez menos necesario ya que el equipo se acostumbró a las actividades del *framework*.

6.1.4. Registro del esfuerzo

Al comienzo del proyecto se utilizó una planilla para indicar las tareas realizadas y el tiempo consumido. De esta manera se agregó transparencia al *backlog* de modo que el equipo es consciente del trabajo de cada uno de los integrantes. Esta medida no estaba alineada con la metodología elegida, pero fue lo más sencillo de utilizar. Mediante las retrospectivas y el uso de *Scrum* orgánico, se comenzó con la utilización de un *sprint taskboard*. Partiendo del

nuevo cuadro, junto con la estimación de las tareas, se pudo comprobar el esfuerzo realizado y el compromiso del equipo.

Como fue antes mencionado la asignación de tareas es individual, tomando la responsabilidad de una historia y realizando todas las tareas necesarias. Existe la posibilidad de compartir tareas con otro integrante, pero siempre hay un responsable por la historia que se encarga de verificar que se termine en tiempo y forma. En las reuniones diarias de Scrum se indica el avance en las tareas o se comunica cuando una historia se encuentra en condiciones de pasar a *testing* cruzado.

Como forma de gestión, el *Scrum master* llevó un registro de los puntos planificados para un *sprint* y la capacidad del equipo, además de completar los puntos realizados al finalizar el *sprint* y los bugs resueltos. Esto permite realizar controles en las reuniones retrospectivas, analizando el trabajo completado y comparar con otros *sprints*. De esta manera se pudo determinar si la velocidad del equipo es la correcta, con qué tanta precisión se planifica y tener un control básico sobre *bugs* o errores resueltos. En la Figura 6-3 se puede observar una gráfica con la velocidad normalizada del equipo.

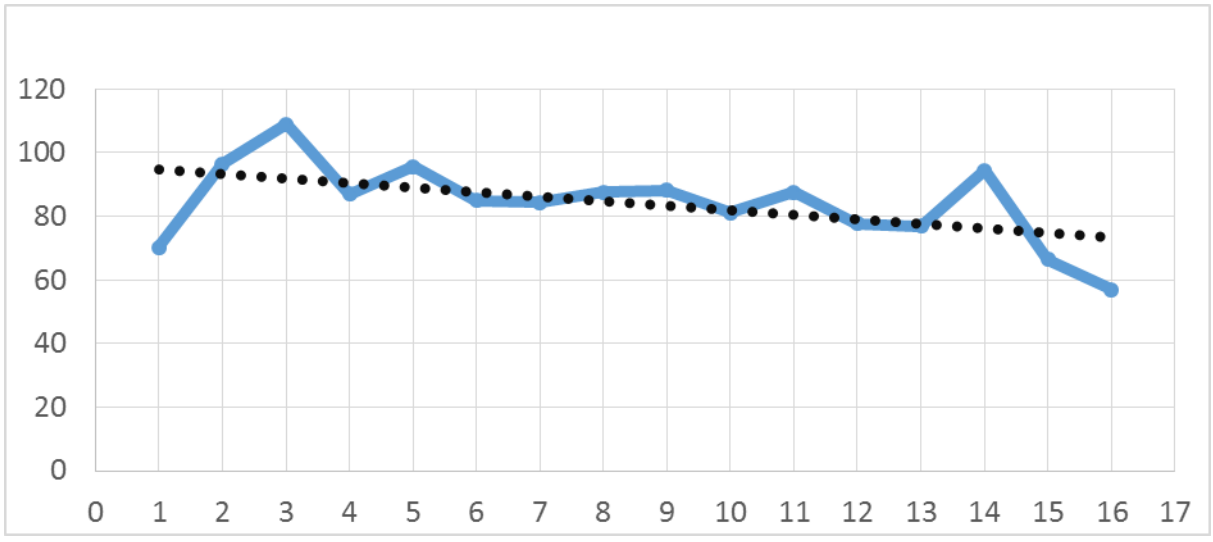

Figura 6-3: Velocidad normalizada del equipo

En esta gráfica se pueden apreciar 2 valores, la velocidad normalizada por *sprint* (línea azul) y el promedio de los puntos (línea punteada). La velocidad normalizada se calcula tomando los puntos realizados en el *sprint* dividido entre la capacidad del equipo. Por lo que dicha velocidad representa fielmente la verdadera velocidad del equipo si el mismo trabajara al

100%. La velocidad normalizada del equipo es de 84 puntos, por lo que se puede observar que desde el *sprint* 4 al 13 se mantuvo constante en esa velocidad. En las etapas iniciales del proyecto, el equipo no tenía total registro del trabajo realizado, había muchas tareas de aprendizaje, errores en la estimación y los *sprints* eran de 3 semanas. A lo largo del proyecto estos errores se fueron corrigiendo, determinando una velocidad y mejorando la estimación de las historias, lo que permitió asegurar el compromiso de las historias dentro de un *sprint* y poder estimar qué historias se pueden realizar en futuros *sprints*. Al final del proyecto se puede observar un descenso en la velocidad, debido a que se decidió terminar las historias comenzadas para no dejar historias inconclusas. Sobre el final del proyecto se observa un descenso en la velocidad debido a que no se contemplaron los datos de esfuerzo dedicado a documentación. En la Figura 4-1 se puede observar la relación entre puntos planificados, realizados y cantidad de bugs por *sprint*.

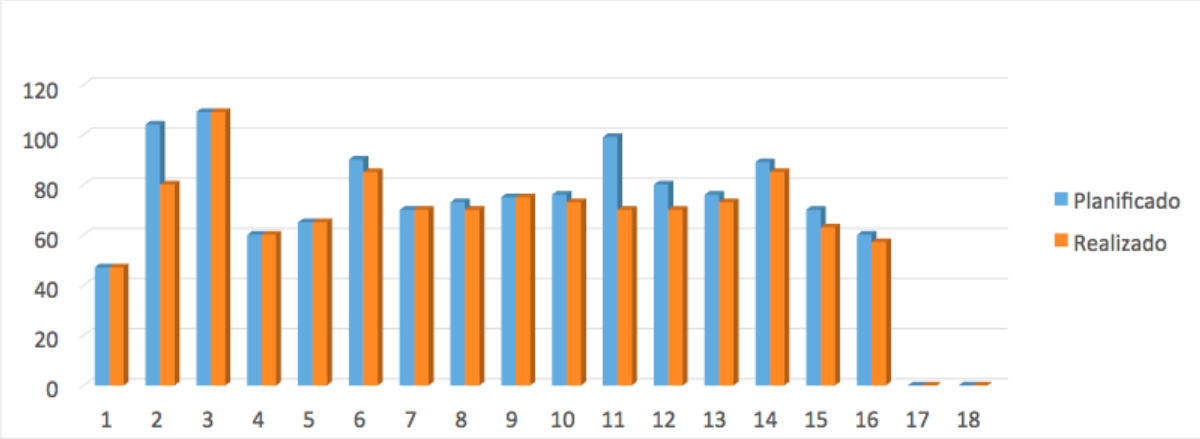

Figura 6-4: Gráfica de Planificado vs Realizado

Sprint	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Bugs	0	10	8	10	12	10	8	5	4	6	6	5	3	4	8	2	2	0

Tabla 6-1: Tabla de bugs resueltos por sprint

En esta gráfica se puede determinar los puntos planificados al iniciar el *sprint* y los puntos de las historias realizadas y aprobadas al final del *sprint*. En general la relación entre planificado y realizado fue cercana, lo que se considera satisfactorio. A la hora del análisis, se debe tener en cuenta la evolución del proceso de estimación y el esfuerzo dedicado a tareas que no están estimadas. Para el primer punto, el *sprint* 2 es un claro ejemplo de un *sprint* con una gran cantidad de puntos estimados, los cuales no se pudieron realizar debido

a que la velocidad del equipo era menor y no se tuvo en cuenta la disponibilidad para el sprint. A lo largo del proyecto la estimación es más precisa, teniendo en consideración la Tabla 6-1 de bugs realizados en el *sprint*. Se tomó la decisión se no estimar los *bugs* ya que no es una tarea sencilla de realizar, simplemente si al resolverlo se determina que el impacto es grande, el *bug* se agrega como historia. Por lo que teniendo la cantidad de los puntos realizados y los *bugs* resueltos en un *sprint*, se puede determinar con qué precisión se estima y se planifica dicho *sprint*. Otro punto notorio es el *sprint* 11, el cual representó el final del MVP y se dedicó a realizar *refactoring* sobre lo entregado, dedicando sólo ese *sprint*. Como fue explicado anteriormente, en los últimos *sprints* se ve un descenso de los puntos realizados, lo que se debe a que se decidió terminar con las historias pendientes, disminuyendo la capacidad del equipo, además de que se comenzó con tareas de documentación.

Considerando los resultados obtenidos a fin de cada sprint, se puede determinar que en este proyecto se consumieron 2200 horas. Un 65% fue dedicado a tareas relativas a implementación y *testing*, mientras que el 35% restante fue dedicado a reuniones, entrevistas y relevamiento de requerimientos.

6.1.5. Herramientas utilizadas

En cuanto a las herramientas a utilizar, la premisa fue encontrar herramientas gratuitas, web y con un respaldo para mitigar problemas de disponibilidad. Para el registro inicial del *backlog* y las tareas se comenzó a utilizar Visual Studio Online como se ve en la Figura 6-5, herramienta provista por Microsoft para la gestión de proyectos ágiles. El equipo no se adaptó totalmente a la herramienta, por lo que durante una reunión de retrospectiva se decidió cambiar por Trello como se observa en la Figura 6-6. Dicha herramienta online permite generar listas que representaron los estados de las tareas, y agregar tarjetas que representan historias y tareas. Mediante un *plugin* llamado Scrum for Trello se pudo guardar la estimación de las historias y controlar los puntos realizados en un sprint. Para la planilla de gestión se utilizó una hoja de cálculo de Excel, que se puede observar en el Anexo 9.7.

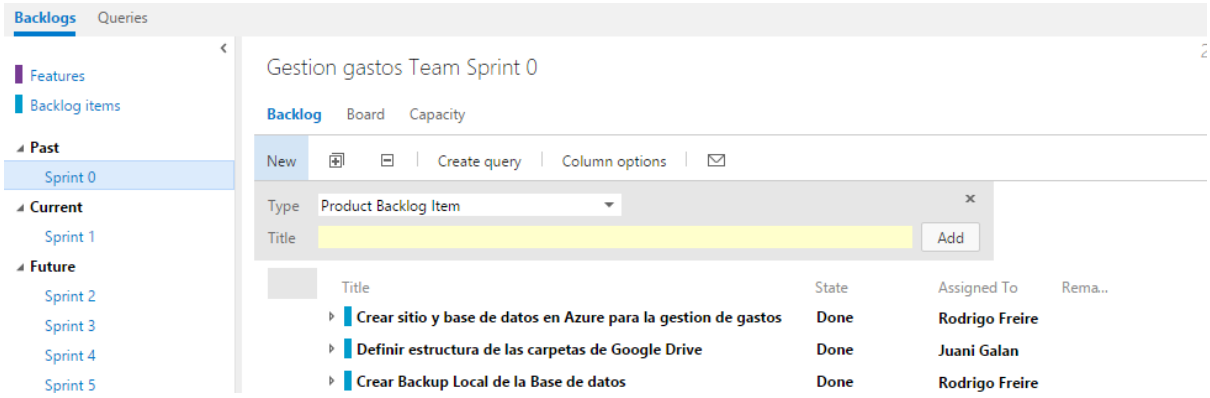

Figura 6-5: Captura de pantalla de las primeras historias en Visual Studio Online

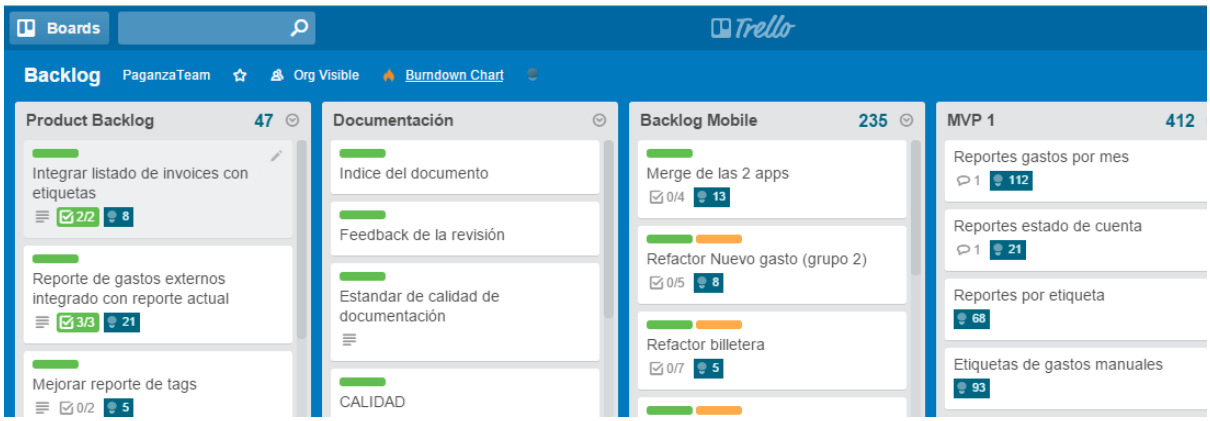

Figura 6-6: Captura de pantalla del backlog en Trello

6.1.6. Gestión de riesgos

Durante la ejecución del proyecto se han identificado los riesgos asociados, y se han asignado los correspondientes planes de respuesta de acuerdo a la valoración. También se han definido indicadores para el correcto monitoreo de los mismos y poder determinar la aplicación de estos planes en caso de ser necesario. Se encontraron los siguientes riesgos, clasificados como positivos o negativos.

<u>Positivos:</u>

1- Popularización de la aplicación y nuevas funcionalidades: En caso de que ocurra en una etapa de madurez y desarrollo del proyecto, se calificó como positivo. Se considera que es una oportunidad para obtener *feedback* y potenciar el desarrollo del proyecto.

Negativos:

2- Popularización de la aplicación y nuevas funcionalidades: Si esto ocurre en una etapa temprana del proyecto, se calificó como un riesgo negativo, ya que el equipo no estaría preparado para afrontar la demanda del producto con un proceso que no está definido completamente.

3- Interés de Paganza: El equipo necesita de la participación de Paganza activamente en el proyecto, dado que es un aporte importante al mismo tanto de manera tangible como intangible como el caso del conocimiento del dominio del problema, intereses de usuarios, etc.

4- Inexperiencia del equipo en aspectos concretos de gestión: Dado que el equipo no ha gestionado proyectos de tal envergadura y con este grado de responsabilidad en el pasado, se debe considerar como un posible riesgo. Deben ser constantemente monitoreados los imprevistos, errores de estimación, y demás factores asociados a la gestión del proyecto.

5- Interesados externos que puedan convertirse en competencia: Dentro del dominio del problema, se encuentran actores que pueden estar interesados en desarrollar productos similares y que podrían competir con el producto de finanzas personales desarrollado en este proyecto.

6- Adaptación a un sistema existente: Teniendo en cuenta que el producto que se desarrollará como módulo dependiente del sistema que Paganza tiene implementado hoy, se deberá tener en consideración la adaptación que requiere por parte de los usuarios, o por parte del sistema existente al momento de desarrollar el producto.

7- Limitaciones del entorno de Paganza: Como consecuencia de ser un módulo dependiente de un sistema existente debe considerarse las limitaciones que puedan presentarse debido a características o implementaciones particulares del sistema actual.

8- Disponibilidad del equipo: Miembros del equipo con dedicación parcial, no constante. Existe la posibilidad de viajes, vacaciones, enfermedad, etc.

9- Disponibilidad de recursos físicos y ambientes: El proyecto requiere materiales, servicios contratados, etc. Estos materiales pueden dejar de estar disponibles, averiarse o perderse.

10- Disponibilidad del cliente: Ausencia por viaje o negocios del cliente que afecte la disponibilidad de respuesta y reunión del cliente con el equipo.

A partir de los riesgos identificados se podrá realizar un análisis de probabilidades de impacto en el proyecto. Se toma como posibles probabilidades entre 20%, 40%, 60% y 80%. Además de esto se debe de establecer la escala del impacto que tiene el riesgo hacia el proyecto. Se decidió utilizar una escala del 1 al 3. Por lo que utilizando las escalas propuestas, se plantea la Tabla 6-2.

Probabilidad / Impacto	**1**	**2**	**3**
20%	Baja	Baja	Media
40%	Baja	Media	Alta
60%	Media	Alta	Alta
80%	Alta	Alta	Alta

Tabla 6-2: Índice de probabilidad e impacto

Se considera que si un riesgo tiene bajo impacto (1) y baja probabilidad (20%), se considerará como de baja magnitud. Por el contrario, se considera un riesgo de alta magnitud, si tiene altas probabilidades de que suceda (80%) y alto impacto (3). Teniendo esto en consideración, se pasan a categorizar los riesgos identificados para así realizar un análisis cualitativo, como se puede observar en la Tabla 6-3.

Factor de riesgo	**Impacto**	**Probabilidad**	**Magnitud**
1-Popularización de la aplicación y nuevas funcionalidades (Positivo)	2	60%	Alta
2-Popularización de la aplicación y nuevas funcionalidades (Negativo)	2	40%	Media
3-Interés de Paganza	3	20%	Media
4-Inexperiencia del equipo en aspectos concretos de gestión	1	40%	Baja
5-Interesados externos que puedan convertirse en competencia	2	40%	Media

6-Adaptación a un sistema existente	3	80%	Alta
7-Limitaciones del entorno de Paganza	1	60%	Media
8-Disponibilidad del equipo	3	80%	Alta
9-Disponibilidad de recursos físicos y ambientes	2	40%	Media
10-Disponibilidad del cliente	2	20%	Baja

Tabla 6-3: Análisis cualitativo de riesgos

Por lo que a raíz de la tabla anterior, se listarán los riesgos según la priorización, seleccionando indicadores y medidas de contingencia.

1- Disponibilidad del equipo: A modo de indicador, en cada planificación se comunicará la disponibilidad de cada integrante, la cual quedará registrada en una planilla. El plan de contingencia consiste en aumentar la dedicación del resto de los integrantes del equipo para no impactar en la velocidad.

2- Adaptación a un sistema existente: El indicador para este riesgo consta en analizar el impacto de cambio de Paganza y del equipo. El plan contingencia se compone de reestructurar la arquitectura propuesta del proyecto o reconsiderar las posibilidades de integración realizando una aplicación aparte.

3- Popularización de la aplicación y nuevas funcionalidades (positivo): Como indicador se utilizó la cantidad de visitas a las nuevas secciones registradas por Google Analytics provistas por Paganza, y la cantidad de menciones en UserVoice. Como plan de contingencia, hacer uso del *feedback* y movimiento generado para retroalimentar el proceso y relevar nuevos requerimientos junto con solicitudes de cambio.

4- Limitaciones del entorno de Paganza: A modo de indicador, se tomará en cuenta la necesidad de utilizar herramientas que puedan estar fuera del entorno que Paganza otorgó. Por lo que el plan de contingencia consta de ofrecer a Paganza el desarrollo, modificación o inversión necesaria para que se pueda utilizar. Otro plan es solicitar a Paganza que provea estas herramientas.

5- Popularización de la aplicación y nuevas funcionalidades (negativo): Se utilizó el mismo indicador que para la parte positiva. El plan de contingencia en este caso se

compone de comunicar a los usuarios que las funcionalidades están en fase beta, recabando todo el *feedback* y errores reporteados para mejorar el producto.

6- Interesados externos que puedan convertirse en competencia: A modo de indicador, se utilizaron las encuestas para relevar posibles herramientas utilizadas por la gestión de gastos, además de noticias de negocios y comunicados de Paganza como experto de negocio. El plan de contingencia consta de priorizar la salida a producción frente al desarrollo de nuevas características para competir rápidamente en el mercado.

7- Disponibilidad de recursos físicos y ambientes: A modo de indicador, tanto en las retrospectivas como en las reuniones diarias, se informará de inconvenientes o trabas que surgen por falta o fallas de recursos. Como contingencia se planificó la solicitud de apoyo a Paganza, como también el aporte de recursos por parte del equipo para reparar o sustituir equipos.

8- Interés de Paganza: Los indicadores para este riesgo se componen por la cantidad de reuniones mantenidas y correos electrónicos intercambiados entre el equipo y el cliente. Como plan de contingencia se consideró ofrecer mayor compromiso en el proyecto, realizando supuestos o sustituyendo *feedback* con investigación. Por último considerar el desarrollo de la herramienta como sistema *standalone*.

9- Disponibilidad del cliente: Como indicador se determinó que en las reuniones de planificación o revisión se le comunique al equipo la disponibilidad del cliente para el próximo sprint. El plan de contingencia se compone de comunicarse con otro integrante del cliente o realizar una reunión extraordinaria a modo de refinamiento para evacuar dudas existentes.

10- Inexperiencia del equipo en aspectos concretos de gestión: Los indicadores constan de errores en estimación, capacidad del sprint, determinación de la velocidad o estándar de calidad alcanzado que puede ser controlado en las reuniones de retrospectiva. A modo de contingencia se planificó realizar una restructuración del proyecto, ajustes dentro del plan y tareas de gestión además de considerar una capacitación.

6.1.6.1. Análisis de riesgos ocurridos

A lo largo del proyecto se ejecutaron los planes de contingencia a raíz de la ocurrencia de algunos de los riesgos. Para responder a problemas en la disponibilidad del equipo, tanto por vacaciones como por exámenes o parciales, los integrantes del equipo que estaban disponibles aumentaron su capacidad en el sprint para no impactar en la velocidad. En cuanto a la adaptación a un sistema existente, luego de completar la capacitación y diseño inicial de la solución se realizó un primer prototipo probando la interacción con Paganza. Tratando de generar la menor cantidad de cambios posibles, se ejecutó el plan de contingencia cambiando la arquitectura diseñada, adaptándola al sistema existente y necesidades del cliente. En cuanto a las limitaciones del entorno de Paganza, surgió la necesidad de utilizar la herramienta Xamarin Forms como también generar un foro separado de UserVoice. Dichas herramientas no estaban disponibles al comienzo del proyecto, y el plan de contingencia empleado fue proponer al cliente que realice una inversión para obtener las herramientas, las cuales fueron adquiridas. Por último en cuanto a la disponibilidad de recursos físicos y ambientes ocurrieron 2 eventos para los que se emplearon los planes de contingencia. El primero fue la pérdida de una computadora de un integrante del equipo por 40 días, por lo que el plan empleado fue que el equipo aportó el recurso faltante consiguiendo otra computadora hasta reparar la original. El segundo evento ocurrido fue la imposibilidad de acceder al servicio de Azure de desarrollo, la cual se solicitó a Paganza que repare y así ocurrió.

6.2. Aseguramiento de la calidad

6.2.1. Introducción

El proyecto Finanzas Personales se originó con el fin de ser parte del actual producto Paganza. Este sistema se encuentra en actual uso por más de 45.000 personas [4]. Esto impulsó a obtener, a raíz del proceso y de actividades enfocadas en el producto, un resultado de calidad adaptado a las necesidades de Paganza.

El uso de Scrum, con sus artefactos y ceremonias, facilita el aseguramiento de la calidad. Por otro lado, se siguieron estándares de codificación y convenciones para entregar un producto de calidad para Paganza.

Los estándares y procesos son importantes para obtener productos de calidad, pero también es necesario adoptar una cultura de calidad en el equipo de desarrollo. Esto implica que cada uno sienta como suyo lo que crea y también, estar alineado con los objetivos de calidad del proyecto. Por ejemplo estar alineados con el cliente en realizar actividades de verificación como parte del proceso entendiendo que esto mejora la calidad del producto. Todos los integrantes del equipo están enfocados en generar un producto de calidad y asumieron la responsabilidad de esto en cada producto de software que realizaban.

6.2.2. Actividades preventivas

6.2.2.1. Scrum

Cómo fue nombrado, se utilizó Scrum para gestionar el proceso. A continuación se describirán las ceremonias del framework que fomentan el aseguramiento de la calidad.

Mediante la reunión de planificación se asegura que el equipo entiende correctamente las necesidades del cliente, permitiendo a los interesados agregar información a la hora de comenzar con la tarea. Al hacerse *sprints* cortos (2 o 3 semanas) se puede ver reflejado fácilmente el trabajo realizado en comparación con lo planificado. Con el criterio de terminado de una tarea y una correcta priorización, se asegura que se trabaja en lo que el cliente solicita.

Al final del sprint, dos ceremonias aseguran la calidad tanto del producto como del proceso. En la reunión de revisión se muestra al cliente lo realizado y este aporta *feedback* de primer nivel. Se realizaron varias reuniones de revisión con el cliente a lo largo del proyecto, que permitieron detectar errores y posibilidades de mejora, favoreciendo la calidad del producto. La reunión de retrospectiva tiene como objetivo analizar lo realizado en el sprint, puntos altos y cuestiones a mejorar. Además, es una instancia donde el equipo se concentra en actividades específicas del proceso, cómo asegurar la calidad en distintas áreas, repasar tareas de *testing* o mejorar documentación. Por ejemplo, en estas reuniones se decidió acortar la duración del sprint de 3 a 2 semanas, para dinamizar la relación con el cliente. Otro ejemplo fue comenzar la aplicación de *testing* cruzado sobre historias terminadas.

6.2.2.2. Estándares de codificación

Debido a que el equipo necesita trabajar sobre un producto que está en constante evolución, se definió realizar una ingeniería reversa del código para extraer un estándar de codificación. El proceso realizado para la generación de este estándar requirió un estudio meticuloso del código que generó en los integrantes conocimiento valioso para la posterior implementación. Una vez finalizado el desarrollo del proyecto, facilitará la integración con el producto actual y la futura mantenibilidad del mismo. Algunos de los puntos importantes extraídos de este estudio son los siguientes:

- Todos los métodos que exponen funcionalidad sobre la base de datos deben ser asíncronos.
- Todas las clases que realicen acceso a la base de datos deben implementar la clase *BusinessComponent.*
- Todas las clases que representan nuevas entidades en el sistema deben heredar de la clase abstracta *EntityBase.*

Los lineamientos de codificación obtenidos permitieron también la incorporación de *Clean Code* [10], al ser también directrices generales para generar código fácil de leer, expresivo, sencillo y mantenible. Algunos de estos preceptos que fueron aplicados son los siguientes:

- Usar variables mnemotécnicas, utilizando nombres que describan que significa la variable. En caso de ser booleanos, escribirlos como si fuesen preguntas.
- Los métodos no pueden sobrepasar las 30 líneas.

El detalle del estándar de codificación y uso de *Clean Code* serán explicados en el Anexo 9.4.

6.2.2.3. Estándares de documentación

Se definió utilizar una plantilla de documentación basada en el documento 302 provisto por la Universidad ORT para generar una estructura base de los documentos escritos durante el proyecto. Esto genera uniformidad y facilidad para los lectores, en casos que deban leer más de un documento. Esta plantilla asegura que los documentos incluyan la fecha de creación,

los autores y las versiones generadas con sus respectivas fechas de edición además del sprint en el que fue realizado.

6.2.2.4. Capacitación

Al ser un producto de grandes dimensiones y gran complejidad técnica, se hicieron sesiones de transferencia de conocimiento por parte de Paganza hacia el equipo. Al comienzo del proyecto se realizaron reuniones en las cuales fue explicado al equipo el dominio del negocio de Paganza y de qué manera resuelve la problemática sobre la que se enfoca. Posteriormente se realizaron sesiones donde el arquitecto de Paganza explicó los principales componentes de la aplicación y sus interacciones. Luego se realizaron instancias de transferencia de conocimiento sobre el despliegue de los componentes mencionados anteriormente. Finalmente se realizaron nuevas reuniones en donde Paganza explicó el uso de las herramientas de desarrollo (Visual Studio, SQL Server Management Studio, Entity Framework utilizando *Code First*), en estas reuniones también se trataron los temas de uso del repositorio de código y base de datos. Esto fue importante ya que al trabajar al mismo nivel que el equipo de Paganza, era necesario mantener coherencia para posteriores integraciones. Se encuentra en el Anexo 9.8 un ejemplo de un documento generado para la configuración del ambiente de desarrollo local.

6.2.3. Validación y verificación

Como parte del proceso de verificación y validación se realizaron revisiones de documentación que incluyeron requerimientos, arquitectura y diseño. También se realizaron revisiones de código y de los resultados de *testing*. Además como parte de este proceso se realizaron actividades de prueba tanto manuales como automatizadas. Las actividades incluyen *testing* unitario, de sistema y de aceptación. Dentro de este proceso se incluyeron evaluaciones heurísticas y experimentación con usuarios a través de pruebas beta con el fin de validar prototipos.

6.2.3.1. Revisiones de documentación

Como parte del proceso de verificación se realizaron revisiones de documentación entre pares. Estas revisiones se realizaron sobre los documentos generados como entregable de

una actividad, como el caso de los resultados de *testing*, conclusiones sobre entrevistas o documentación de diseño. El propósito de estas revisiones, realizadas por otro integrante que no haya participado en la confección del documento a revisar, es encontrar errores para que sean corregidos.

Por otro lado, se realizaron revisiones sobre documentos particulares como es el caso de la documentación de diseño y arquitectura del sistema. Estas revisiones eran necesarias debido a las características del proyecto, de integración sobre un producto existente, y eran conducidas por el arquitecto de Paganza. Cabe destacar la rigurosidad de estas revisiones teniendo en cuenta la formación académica y técnica del representante del cliente encargado de realizarlas.

6.2.3.2. Testing

A lo largo del proyecto se realizó *testing* unitario, de integración, de sistema y de aceptación utilizando *alpha testing* y *beta testing*. A nivel de test unitario se desarrollaron *tests* que cubren casos básicos para todos los métodos expuestos por las interfaces de cada componente. Cada vez que se agregaba un nuevo componente a la solución, se creaba un proyecto de *testing* unitario que probaba las funcionalidades que este componente exponía, generando así un conjunto de pruebas automáticas. En la Figura 6-7 se muestra un ejemplo de prueba unitaria para uno de los módulos desarrollados. Estas pruebas generadas eran ejecutadas durante el proceso de pruebas de regresión como se explica más adelante en esta sección.

```
[TestMethod]
public async Task ExpensesTags_DeleteTagForExpense_TwoTags()
{
    var factory = new Mock<ICoreFactory>();
    var dbContext = new Mock<IDbContext>();
    var unitOfWork = new UnitOfWork(dbContext.Object);
    var dbSet = new FakeDbSet<ExternalExpense>();
    var dbSet2 = new FakeDbSet<Tag>();

    factory.SetupGet(f => f.UnitOfWork).Returns(unitOfWork);
    dbContext.Setup(j => j.Set<ExternalExpense>()).Returns(dbSet);
    dbContext.Setup(j => j.Set<Tag>()).Returns(dbSet2);

    IExpensesTags tA = new Tags.AccessLayer.ExpensesTags(factory.Object);
    dbSet.Add(new ExternalExpense() { Id = 1, OwnedById = 1, Tags = new List<Tag>(), Name = "NewExpense" });

    ResponseClasses.ExpensesTags exp1 = await tA.CreateTagForExpense("Tag1", 1, 1);
    ResponseClasses.ExpensesTags exp2 = await tA.CreateTagForExpense("Tag2", 1, 1);

    Assert.IsTrue(exp2.Tags.Count.Equals(2));

    int tagId = exp2.Tags.Where(i => i.Name.Equals("Tag1")).Select(i => i.Id).FirstOrDefault();
    ResponseClasses.ExpensesTags expD = await tA.DeleteTagFromExpense(tagId, 1, 1);

    Assert.IsTrue(expD.Tags.Count.Equals(1));
}
```

Figura 6-7: Ejemplo de prueba unitaria

Como parte del proceso de pruebas de integración se decidió que dentro de la duración del sprint, cada vez que se terminaba una historia, ésta se pasaba para un estado de prueba, donde se comprobaba a nivel de *testing* funcional. Aquí otro integrante del equipo, diferente del que había realizado la tarea, prueba el correcto funcionamiento y verifica que la definición de completado se cumpliera mediante la aplicación de pruebas exploratorias. En caso de que la tarea impactara directamente sobre el *Core* de la aplicación de Paganza, la misma también era sometida a una revisión de código por otro integrante. Los errores encontrados en esta etapa se registraban en una planilla como la que puede verse en el Anexo 9.9.

Al terminar el sprint se mostraban las historias completadas en ese sprint que aportaban valor al cliente. En esta instancia se hacía *testing* de aceptación, donde el cliente definía si lo realizado estaba aprobado o si necesitaba alguna modificación. También como parte del *testing* de aceptación se realizaron pruebas *alpha* llevadas a cabo por Paganza para constatar el correcto funcionamiento de los componentes desarrollados.

Finalmente se realizaron pruebas *beta* mediante la habilitación de las funcionalidades a usuarios seleccionados utilizando criterios otorgados por Paganza, tales como la frecuencia de utilización de la aplicación y de la web. Para obtener los resultados de estas pruebas se utilizó la herramienta UserVoice, que permite la recolección de *feedback* directamente

desde la web. Algunas de estas pruebas con usuarios seleccionados, se realizaron bajo la modalidad de reuniones de observación. En estas reuniones se solicitaba al usuario realizar tareas específicas utilizando el sistema, para registrar las observaciones y detectar posibilidades de mejora.

En el caso de la aplicación móvil se realizaron reuniones para observar el comportamiento de los usuarios, teniendo en cuenta que las funcionalidades eran de carácter experimental. Además se consideró importante poder evaluar el contexto de uso de la aplicación por lo que algunas de estas evaluaciones se realizaron bajo situaciones simuladas de uso cotidiano, como se ve en la Figura 6-8, donde se solicitó a un usuario ingresar un gasto escaneando un código QR inmediatamente después de haber pagado en un supermercado. Los resultados de estas sesiones se registraron utilizando planillas como por ejemplo la que se encuentra en el Anexo 9.10.

Figura 6-8: Usuario ingresando un gasto escaneando un código QR

Dentro del contexto de automatización de pruebas además de las pruebas unitarias, que son automatizadas, se utilizó Selenium. Esta herramienta orientada a la plataforma web permite grabar sucesiones de eventos desde la perspectiva del usuario para reproducirlos luego, y evaluar su resultado comparándolo con el resultado esperado [11]. En la Figura 6-9 se puede ver un ejemplo de la utilización de esta herramienta.

```
[TestMethod]
public void WalletLoadTest()
{
    driver.Navigate().GoToUrl(baseURL);
    driver.FindElement(By.LinkText("Ingresar")).Click();
    driver.FindElement(By.Id("username")).Clear();
    driver.FindElement(By.Id("username")).SendKeys("33866");
    driver.FindElement(By.Id("password")).Clear();
    driver.FindElement(By.Id("password")).SendKeys([redacted]);
    driver.FindElement(By.CssSelector("#formLogin > div.popUpBottom > #formSubmit")).Click();
    driver.FindElement(By.CssSelector("div.itemBalance.contentMenuItem")).Click();
    driver.Manage().Timeouts().ImplicitlyWait(TimeSpan.FromSeconds(10));
    Assert.AreEqual(true, IsElementPresent(By.CssSelector("#wallet_tabs_container")));
}
```

Figura 6-9: Ejemplo de prueba de Selenium

La automatización fue realizada debido a que dentro del marco del proyecto se iban a hacer nuevos módulos de forma periódica, por lo tanto era necesario evitar la reaparición de defectos corregidos o la introducción de nuevos defectos a causa de estas modificaciones. De esta manera, al momento de integrar un nuevo componente se ejecutaban las pruebas de regresión compuestas por las pruebas unitarias y las pruebas de interfaz web desarrolladas con Selenium.

6.2.3.3. Registro de bugs

Para el registro de bugs se utilizó en las etapas iniciales del proyecto la herramienta Workflowy ya que el cliente está familiarizado con esta herramienta. Más adelante se decidió utilizar TFS otorgando acceso al cliente de forma de permitir el ingreso directo de bugs encontrados. Esta decisión se debió a la necesidad de mayor complejidad en el manejo y la organización de los datos ingresados. TFS satisface esta necesidad ya que incluye características específicas para el manejo de bugs.

Algunos de los bugs o posibilidades de mejora fueron reportados directamente por los usuarios utilizando la herramienta de *feedback* online que Paganza ya empleaba, UserVoice.

6.2.3.4. Evaluaciones heurísticas

Este método de evaluación sin participación del usuario, consiste en verificar ciertos lineamientos que un sistema debería cumplir a nivel de usabilidad. Se basaron en las heurísticas de Nielsen [12], las cuales son 10 recomendaciones sobre elementos y comportamientos que un sistema web debe contener. Como ejemplo de aplicación de esta técnica, bajo la heurística de "Control y libertad del usuario" se encontró que en la funcionalidad agregar etiqueta a un gasto, la cual implica ingresar el detalle del gasto, sería útil agregar un *breadcrumb* que mejore la navegabilidad dentro del sitio. En el Anexo 9.11 se encuentra el detalle de la aplicación de esta metodología.

6.2.4. Métricas

6.2.4.1. Definición de Métricas

Con el fin de apoyar la toma de decisiones y controlar el proceso, se buscará definir un conjunto de métricas que permitan brindar la información suficiente para dicho análisis. Para la definición de estas métricas el equipo se ha basado en la técnica GQM [13], la cual permite definir diferentes metas las cuales permitieron determinar, a través de preguntas realizadas, las métricas más relevantes al proyecto. A continuación se pasará a describir las metas definidas para este proyecto, así como también las preguntas y las métricas correspondientes.

Observando la naturaleza del proyecto, así como también las necesidades del cliente y de los usuarios, se determinó, que sería necesario realizar una medición de la usabilidad del sistema. Debido a que Paganza es un producto que se enfoca en la facilidad y simpleza de su sistema, se decidió que una de las metas sería analizar el producto entregado con el propósito de evaluar la usabilidad desde el punto de vista del usuario. Con esto, se realizaron las preguntas y métricas que se consideraron necesarias:

- ¿Cuánto tiempo requiere hacer uso de las funcionalidades más importantes?
    - Tiempo para ingresar un gasto.
    - Tiempo que lleva ver gastos de un mes en particular.
    - Tiempo que lleva agregar una etiqueta.

    - Tiempo que lleva ver el listado de gastos.
    - Tiempo de reconocimiento de texto.
- ¿Qué tan fáciles de usar son las funcionalidades desarrolladas?
    - Cantidad de *clicks* se requieren para agregar una etiqueta.
    - Cantidad de *clicks* para ver los gastos de un mes en particular.
    - Cantidad de *clicks* para crear un gasto.
    - Cantidad de *clicks* para crear gastos de manera múltiple.
- ¿Qué tan usadas son las funcionalidades desarrolladas?
    - Cantidad de visitas a Paganza / Cantidad de visitas a funcionalidades implementadas.

Por otro lado, debido a la necesidad de que el proyecto sea fácilmente mantenible por el equipo de Paganza, se define que otra meta sea analizar el producto entregado con el fin de evaluar la mantenibilidad desde el punto de vista del desarrollador de Paganza. Para esta meta, se definieron las siguientes preguntas y métricas:

- ¿Qué tan fácil de mantener es el código desarrollado?
    - Índice de Mantenibilidad [14][15].
- ¿Cuántos cambios fueron introducidos al código existente?
    - Cantidad de cambios sobre código existente.

En tercer lugar, se define que será necesario medir la eficiencia de tanto el sistema web como la aplicación móvil, a modo de obtener un indicador de cómo se esta desarrollando el sistema y cómo se compara el sistema desarrollado con las aplicaciones similares del mercado. También será necesario validar el impacto sobre la eficiencia actual de Paganza, validando que las funcionalidades que se desarrollaron sobre funcionalidades existentes no hayan impactado de manera crítica. Para esto, es necesario definir una meta que permita mejorar y controlar la eficiencia desde el punto de vista del sistema. Para esta meta, se definen las siguientes preguntas y métricas:

- ¿Cuál es el tiempo de respuesta de las funcionalidades en comparación con herramientas análogas?
    - Tiempo para ingresar un gasto.
    - Tiempo que lleva ver gastos de un mes en particular.
    - Tiempo que lleva en agregar una etiqueta.

    - Tiempo que lleva ver el listado de gastos.
    - Tiempo de reconocimiento de texto.
- ¿Cuál es el impacto de eficiencia sobre las funcionalidades existentes?
    - Tiempo de respuesta para el listado de facturas sin etiquetas.
    - Tiempo de respuesta para el listado de facturas con etiquetas.
    - Tiempo de respuesta para el detalle de una factura sin etiquetas.
    - Tiempo de respuesta para el detalle de una factura con etiquetas.

Por último, se considera que será necesario definir una meta que permita mejorar y controlar el proceso de ingeniería de software desde el punto de vista del equipo, para poder controlar la efectividad en las tareas del proceso, así como también controlar los tiempos del proyecto y evaluar la dedicación. Para esto, se definen las siguientes preguntas y métricas:

- ¿Cuántas solicitudes de cambio fueron hechas por el cliente?
    - Solicitudes de cambio por funcionalidad hechas por el cliente.
- ¿Cuántos errores fueron encontrados por revisiones internas?
    - Cantidad de errores detectados por revisiones.
- ¿Cuántos errores fueron encontrados por usuarios?
    - Cantidad de errores detectados por usuarios.
- ¿Cuántos errores fueron encontrados por Paganza?
    - Cantidad de errores detectados por Paganza.
- ¿Cuántas de las funcionalidades resuelven problemas experimentados por usuarios?
    - Solicitudes de cambio sobre requerimientos planteados por los usuarios.
- ¿Cuánto tiempo se dedica por semana al proyecto?
    - Horas por semana.
    - Horas totales.
- ¿Qué tan eficientemente estima el equipo?
    - SP planificados por sprint / SP completados por sprint.

6.2.4.2. Registro de Métricas

A continuación se realiza una agrupación de la metodología a utilizar a la hora de registrar las métricas definidas previamente.

Por un lado, las métricas correspondientes a eficiencia, *feedback* de usuarios y mantenibilidad serán relevadas cada 2 *sprints,* para obtener métricas que consideren los cambios realizados sobre el sistema. Dichas métricas serán registradas utilizando la herramienta Google Sheets. En lo que respecta a métricas de eficiencia, para tener una comparación con otras herramientas de gestión de gastos, se realizan las mismas pruebas de eficiencia sobre las herramientas Expensify, Xpenditure y Buxfer. A continuación se detallan las métricas que serán recolectadas utilizando este enfoque, además de aclaraciones sobre cómo se registra cada métrica:

- Tiempo para ingresar un gasto.
- Tiempo que lleva ver gastos de un mes en particular.
- Tiempo que lleva agregar una etiqueta.
- Tiempo que lleva ver el listado de gastos.
- Tiempo de reconocimiento de texto.
- Cantidad de *clicks* que se requieren para agregar una etiqueta.
- Cantidad de *clicks* para ver los gastos de un mes en particular.
- Cantidad de *clicks* para crear un gasto.
- Tiempo de respuesta para el listado de facturas sin etiquetas.
    - Esta métrica será relevada al comienzo del desarrollo de la funcionalidad de etiquetas y se tendrá como referencia a lo largo del desarrollo.
- Tiempo de respuesta para el listado de facturas con etiquetas.
- Tiempo de respuesta para el detalle de una factura sin etiquetas.
    - Esta métrica será relevada al comienzo del desarrollo de la funcionalidad de etiquetas y se tendrá como referencia a lo largo del desarrollo.
- Tiempo de respuesta para el detalle de una factura con etiquetas.
- Cantidad de visitas a Paganza / Cantidad de visitas a funcionalidades implementadas.
    - Esta métrica será otorgada por Paganza debido a que son ellos quienes tienen acceso al administrador de *Google Analytics* del sitio, así como también acceso a *Azure*. En la Figura 6-10 se puede observar dicha métrica.

Page	Pageviews	Unique Pageviews	Avg. Time on Page	Entrances	Bounce Rate	% Exit	Page Value
	8,101 % of Total: 7.20% (112,583)	5,346 % of Total: 7.62% (70,181)	00:01:00 Avg for View: 00:01:11 (-15.99%)	1,231 % of Total: 4.35% (28,290)	25.84% Avg for View: 39.44% (-34.49%)	14.74% Avg for View: 25.13% (-41.34%)	<$0.01 % of Total: 22.30% ($0.02)
1. /Reports/Balance	4,245 (52.40%)	3,084 (57.69%)	00:00:41	95 (7.72%)	48.42%	9.33%	<$0.01 (126.07%)
2. /Reports/Analysis	3,856 (47.60%)	2,262 (42.31%)	00:01:23	1,136 (92.28%)	23.94%	20.70%	<$0.01 (64.46%)

Show rows: 10 Go to: 1 1 - 2 of 2

Figura 6-10: Porcentaje de visitas de las funcionalidades en producción

- Índice de Mantenibilidad
    - Se utilizará la herramienta integrada en Visual Studio 2013 para evaluar el índice de mantenibilidad del sistema, como se muestra en la Figura 6-11. Si bien la documentación de dicha métrica [14] declara que un buen índice se encuentra dentro del rango 20 – 100, se considera que debido a que se debe entregar un proyecto altamente mantenible, se considerará un buen índice de mantenibilidad aquel que se encuentra en el rango 85 – 100.

Code Metrics Results

Filter: None Min: Max:

Hierarchy	Maintainability Index
ExternalExpenses\Paganza.Companies (Debug)	84
ExternalExpenses\Paganza.Expenses (Debug)	88
ExternalExpenses\Paganza.ImportExpenses (Debug)	85
ExternalExpenses\Paganza.QueryExpenses (Debug)	69
Reports\Paganza.Balance (Debug)	89
Reports\Paganza.Reports (Debug)	90
Reports\Paganza.TagsReport (Debug)	88
Tags\Paganza.Tags (Debug)	85

Figura 6-11: Medición de Índice de Mantenibilidad

- Cantidad de cambios sobre código existente.
    - Se evaluará la cantidad de cambios realizados sobre las clases existentes de Paganza.
- Solicitudes de cambio por funcionalidad hechas por el cliente.
- Solicitudes de cambio sobre requerimientos planteados por los usuarios.
- Cantidad de errores detectados por revisiones.
- Cantidad de errores detectados por usuarios.
- Cantidad de errores detectados por Paganza.

Para el registro de tiempo en la aplicación móvil, se realizará un promedio del tiempo que le lleva a un usuario experto realizar la operación. Un ejemplo de dicho registro se muestra en

la Figura 6-12. Para realizar la medición de eficiencia en la aplicación web, se crearan *scripts* de automatización utilizando la herramienta KITE (Figura 6-13) y se comparará los resultados de cada herramienta. Se utilizará una máquina virtual para realizar las pruebas de eficiencia para mantener un ambiente de prueba intacto e idéntico a lo largo del proyecto.

Mobile	1	2	3	4	5	Avg
Crear un Gasto Manualmente	20.18	16.86	15.31	15.33	15.4	16.616
Listado de Gastos	2.8	2.53	2.57	2.22	2.48	2.52
Reconocimiento de Texto	32.04	30.04	43.02	35.06	29.06	33.844

Web	Resultados
Crear un Gasto	4.975
Listado de Gastos	2.474
Filtrado de Gastos	1.256
Agregar Etiqueta a Gasto	0.565

Figura 6-12: Medición de performance

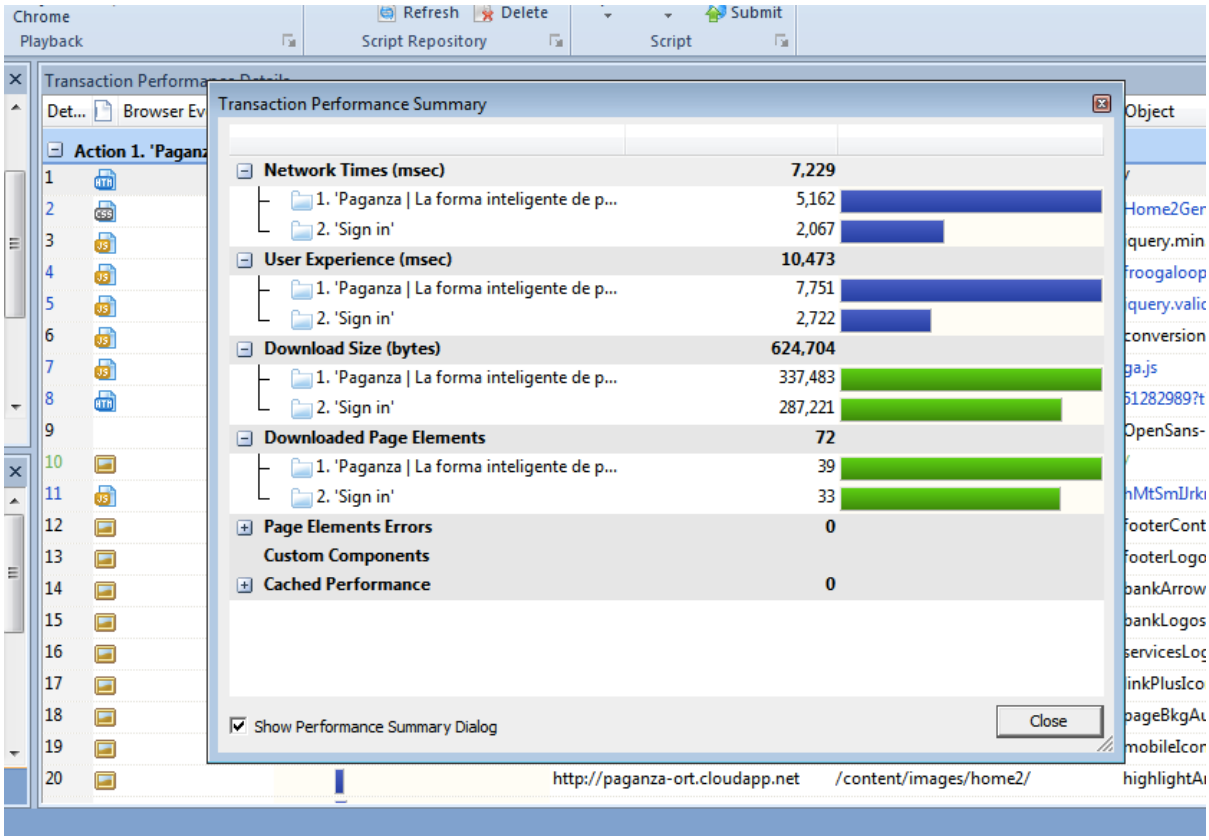

Figura 6-13: Resultado de medición de eficiencia utilizando KITE

En lo que respecta a las solicitudes de cambio y al registro de errores, se pone a disposición de Paganza y del equipo la herramienta TFS para el registro de los mismos, un ejemplo

puede verse en la Figura 6-14. Para ello, se crea una guía de reporte de errores que se encuentra en el Anexo 9.12.

22	Analisis	Product Backlog Item	Mejoras en Reporte de Gastos	New	Gestion gastos\Paganza
23	Analisis	Bug	No se puede ocultar el filtro en el reporte de gastos	New	Gestion gastos\Paganza
24	Billetera	Bug	Fallo en reporte de billetera	New	Gestion gastos\Paganza
25		Product Backlog Item	Mejoras en codigo para uso de async	New	Gestion gastos\Paganza
26	Etiquetas	Bug	Agregar mensaje cuando se quiere agregar la sexta tag. Ahora dice Incorrect Invoice	New	Gestion gastos\Equipo
27	Gastos Externos	Bug	ServiceId de expenses se puede superponer con los serviceId de invoices	New	Gestion gastos\Equipo
28	Facturas	Bug	Loading de listado de facturas no ocupa toda la pantalla	New	Gestion gastos\Equipo
29	Gastos Externos	Bug	No hay forma de volver al listado de gastos	New	Gestion gastos\Equipo
30	Gastos Externos	Bug	Al borrar una etiqueta, la llamada al servicio devuelve un error y no se borra la etiqueta	New	Gestion gastos\Equipo
31	Gastos Externos	Bug	No se muestran las etiquetas previamente ingresadas	New	Gestion gastos\Equipo
32	Analisis	Bug	La grafica de analisis se corta	New	Gestion gastos\Usuarios

Figura 6-14: Errores y Oportunidades de mejora registradas en TFS

Por otro lado, las métricas que se enfocan en el registro de horas consumidas, se registrarán en cada sprint, para poder tener un seguimiento de las horas consumidas por el equipo con el fin de realizar reajustes o planificar los *sprints*. Se registrará en Google Sheets las horas consumidas por *sprint*, y a partir de ellas, se pueden derivar las siguientes métricas:

- Horas totales.
- Horas por semana.
- SP planificados por sprint / SP completados por sprint.

Un resumen del registro de dichas métricas se puede encontrar en el Anexo 9.13.

6.2.4.3. Análisis de Métricas

A partir de la métricas registradas, se realizó un análisis a lo largo del proyecto que permitió realizar acciones correctivas y fundamentar la toma de decisiones. A continuación se muestra un análisis de las métricas registradas y de qué manera impactaron en el proyecto.

Por un lado, en lo que refiere a las métricas de eficiencia, se pudo observar que los tiempos de ingreso de gastos, tanto en la aplicación web como en la móvil, eran más altos que los tiempos de ingreso para las aplicaciones competidoras (Anexo 9.13). Esto se debió a que los campos de Rubro y Fecha de ingreso eran auto completados por las aplicaciones competidoras, mientras que en la aplicación desarrollada se le obligaba al usuario a ingresar las mismas. Para mejorar esto, se decidió seleccionar el primer rubro por defecto, y la fecha de ingreso como la del día actual. Esto, como se puede observar entre el sprint 12 y 14 del Anexo 9.13, mejoró considerablemente el tiempo de ingreso de gastos. Además, dicha

mejora permitió mejorar la usabilidad de la aplicación, como se puede observar en la cantidad de *clicks* y cantidad de campos a completar en el Anexo 9.13.

Otra mejora que es importante destacar es el tiempo de carga del listado de facturas. Debido al desarrollo de la funcionalidad de etiquetas al listado de facturas, la performance de dicha funcionalidad fue impactada con un aumento del 7% del tiempo de respuesta. Luego del correspondiente análisis, se observó que se accedía, por cada factura, a la base de datos para obtener las etiquetas de dichas facturas. Esto se mejoró agregando la información de las etiquetas a la tabla accedida para obtener la información de las facturas.

En lo que respecta a la mantenibilidad, se puede observar que hasta el sprint 10, ninguno de los paquetes desarrollados cumplía con la meta de índice de mantenibilidad de 85. A partir de esto, se consideró que era necesario realizar una etapa de *refactoring* del código que aumentará la mantenibilidad de las funcionalidades desarrolladas y que permita mejorar la claridad y disminuir la complejidad del código. Haciendo uso de interfaces para los servicios expuestos dentro de los componentes, así como también la aplicación de Clean Code de una manera más rigurosa, se logró que dicho índice aumentara, logrando el objetivo preestablecido.

Sobre la detección de errores, se puede observar que la cantidad de errores detectados por Paganza hasta el sprint 8 representaba un 43% de los errores encontrados en total, lo que demostraba que el proceso de revisión del equipo no era lo suficientemente efectivo. Para mejorar esto, se decidió aumentar el número y la rigurosidad de las revisiones de código. Esto resultó en la disminución de la cantidad de errores reportados por Paganza.

Un aspecto importante a destacar es que en varias funcionalidades del sistema desarrollado la usabilidad y eficiencia del mismo es igual e incluso mejor que las aplicaciones que se encuentran actualmente en el mercado. Por un lado, se puede observar que el tiempo necesario para reconocer texto para el alta de gastos por parte del sistema desarrollado es unas 6 veces más rápido que el reconocimiento de texto por parte de la herramienta Expensify, y unas 12 veces más rápido que la herramienta Xpenditure. Por otro lado, la importación de gastos a través de un archivo Excel generó una mejor usabilidad y eficiencia que las aplicaciones del mercado, debido a que para importar gastos en Paganza, solo es

necesario crear un archivo Excel y subir dicho archivo, mientras que en las aplicaciones del mercado es necesario ingresar cada gasto de manera individual a una planilla generada en la página, como se puede ver en la Figura 6-15.

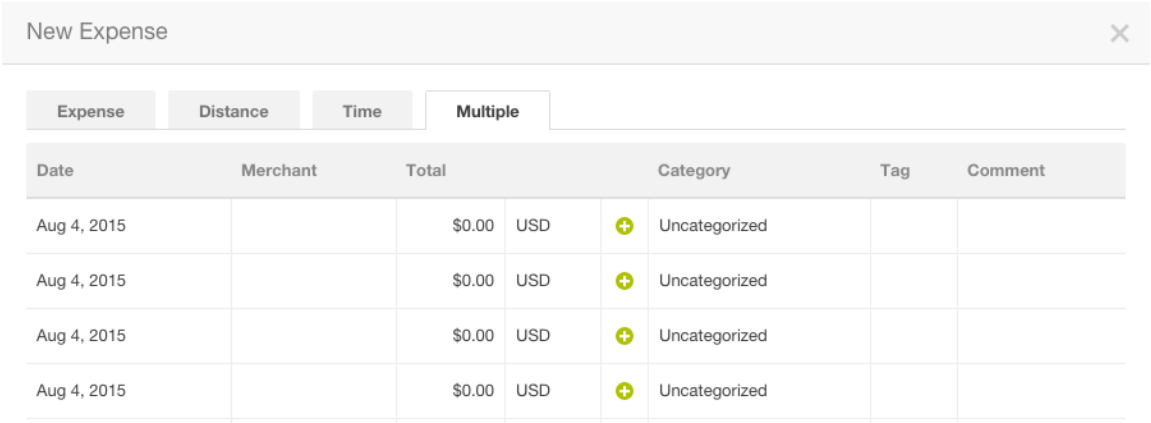

Figura 6-15: Ingreso de múltiples gastos utilizando Expensify

Por último, se pudo observar que a mediados del *sprint* 6, la velocidad del equipo estaba tendiendo a unos 80 puntos por sprint. Esto determinó que el equipo comenzara a realizar las planificaciones de los *sprints* a partir de dicha unidad, tomando como cantidad máxima de puntos a desarrollar como 80.

6.3. SCM

Dentro de la gestión de configuración se considera la definición y gestión de los repositorios, tanto de código como de documentación. También incluye la administración del proceso y herramientas de desarrollo, el apego al proceso de desarrollo y la administración del software y hardware necesario para el funcionamiento del sistema.

6.3.1. Repositorios de código

Para el almacenamiento y el versionado del código se utilizaron repositorios remotos con el sistema de versionado Git. Esta decisión fue tomada debido a que Paganza al momento de comenzar el proyecto utilizaba este tipo de repositorio. Para el equipo fue sencillo adecuarse a esta herramienta dado que era conocida por todos los integrantes.

6.3.1.1. Repositorio Web

Para el desarrollo de las funcionalidades web, dado que el sistema está completamente integrado con Paganza desde el punto de vista de la implementación, se comparte un repositorio único. Cuando comenzó el desarrollo fue concedido el acceso a este repositorio a los miembros del equipo y fue creada una rama para el trabajo de las funcionalidades correspondientes al proyecto. El equipo publica el código directamente en el repositorio remoto administrado por Paganza que se encuentra almacenado en Bitbucket. El cliente dio acceso a esta rama, pero no adjudicó permisos sobre el repositorio en general. Esto ocasionó que no se pudieran crear más ramas para el desarrollo web.

6.3.1.2. Repositorio Móvil

En el caso de la aplicación móvil, como el propósito del desarrollo era realizar prototipos y pruebas de concepto, cuya integración al producto de Paganza no estaba en el horizonte cercano, se creó un nuevo repositorio. Este repositorio también se almacena utilizando el servicio de Bitbucket, pero es administrado por el equipo. Para la organización del repositorio se crean ramas por grupo de funcionalidad como fue explicado en el proceso de implementación. Cada una de las ramas respetaba el flujo estricto llamado *Gitflow Workflow*. En ella se contempla que se tengan 2 ramas, una llamada master y otra desarrollo. En la primera se mantiene el código de los *releases*, mientras que en la segunda se mantiene el código principal, y a partir de esta se crean las nuevas ramas que contendrán las nuevas funcionalidades. Esto es más entendible en formato gráfico, como se muestra en la Figura 6-16 de [16].

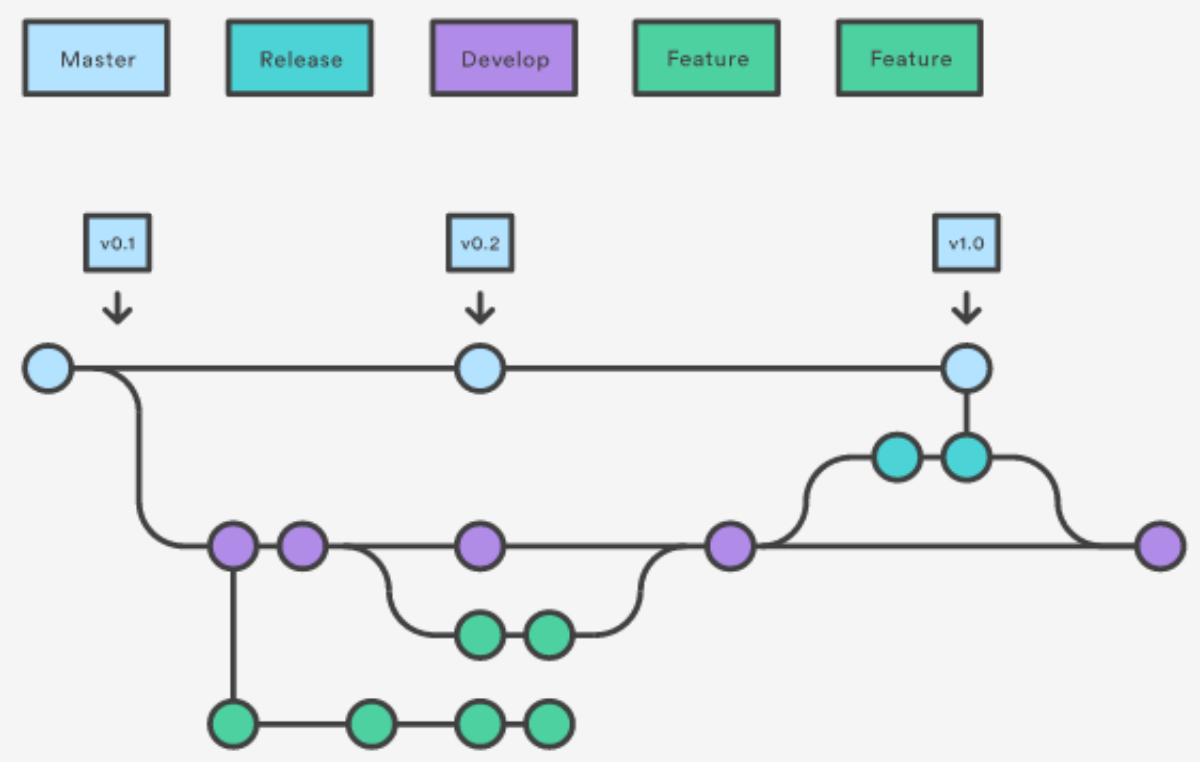

Figura 6-16: Diagrama de estructuración del repositorio

En la rama principal (master), es recomendado utilizar etiquetas para marcar los *releases*. Luego en la rama de desarrollo, se tienen todos los *commits* de los desarrolladores, y como fue comentado, a partir de esta rama es que se crean nuevas *branches* para desarrollar nuevas funcionalidades. Cuando lo nuevo es aprobado, se hace *merge* con la rama de desarrollo y luego de un determinado número de funcionalidades nuevas, se hace *merge* con la rama principal, generando así un nuevo *release*.

6.3.2. Manejo de documentación

Para mantener la documentación accesible a todo el equipo y poder tener una forma fácil de manipularlos en conjunto, se decidió utilizar la herramienta Google Drive. Esto se decidió, ya que era una herramienta conocida por los integrantes, y por el cliente, facilitando el tiempo de aprendizaje de la herramienta. Esta herramienta también gestiona automáticamente el versionado de los documentos.

6.3.3. Estructura

Las carpetas dentro de la unidad de Google Drive, se crearon con el fin de organizar y agrupar por temática. La estructura de carpetas de la unidad de Google Drive es la siguiente:

- Documentación general
    - Proceso de ingeniería de software
    - Procesos de apoyo
- Sprints
    - Sprint Nº - Fecha de comienzo – Fecha de fin
- Reuniones

Dentro de documentación general, se encuentran todos los documentos relacionados con el proceso de ingeniería de software y los procesos de apoyo. Siendo estos ingeniería de requerimientos, diseño, desarrollo y testing, y dentro de los procesos de apoyo, gestión de configuración del software, aseguramiento de calidad y gestión de proyectos. Dentro de la carpeta Sprints, se encuentran cada uno de los Sprints realizados con los entregables resultantes de la ejecución del Sprint. Por último, dentro de Reuniones, se encuentran documentos que se utilizaron para recabar cierta información en reuniones con el cliente, con el tutor, o con otros actores.

6.3.4. Proceso de Despliegue

En lo que respecta al proceso de despliegue, se debe tomar en cuenta los siguientes aspectos:

- Los funcionalidades desarrolladas fueron implementadas en un sitio de desarrollo, creado por Paganza para este proyecto en especial
- Los funcionalidades desarrolladas pueden haber impactado en el código original de Paganza, así como también en la base de datos del sistema.
- Todo código desarrollado por el equipo debe cumplir con los requerimientos de calidad establecidos por Paganza.

Teniendo en cuenta estos puntos, se determinó, en conjunto con Paganza, el proceso de despliegue de las funcionalidades desarrolladas. En primer lugar, se llevará a cabo una pre

validación por parte del equipo, donde se ejecutarán las pruebas unitarias para evaluar que las nuevas funcionalidades no hayan introducido errores en el sistema. En caso de que se encuentren errores, se genera un reporte y se cancela el despliegue. En caso de que no se encuentren, se despliega la funcionalidad en el sitio de desarrollo y se le notifica a Paganza, quienes realizan la validación de dicha funcionalidad. Una vez aprobada la funcionalidad, el equipo pasa a actualizar el repositorio con la nueva funcionalidad para que Paganza realice una inspección de código que determinara si la funcionalidad se integra al sistema en producción o si debe mejorar en cuanto a la calidad. Este proceso se puede observar en la Figura 6-17.

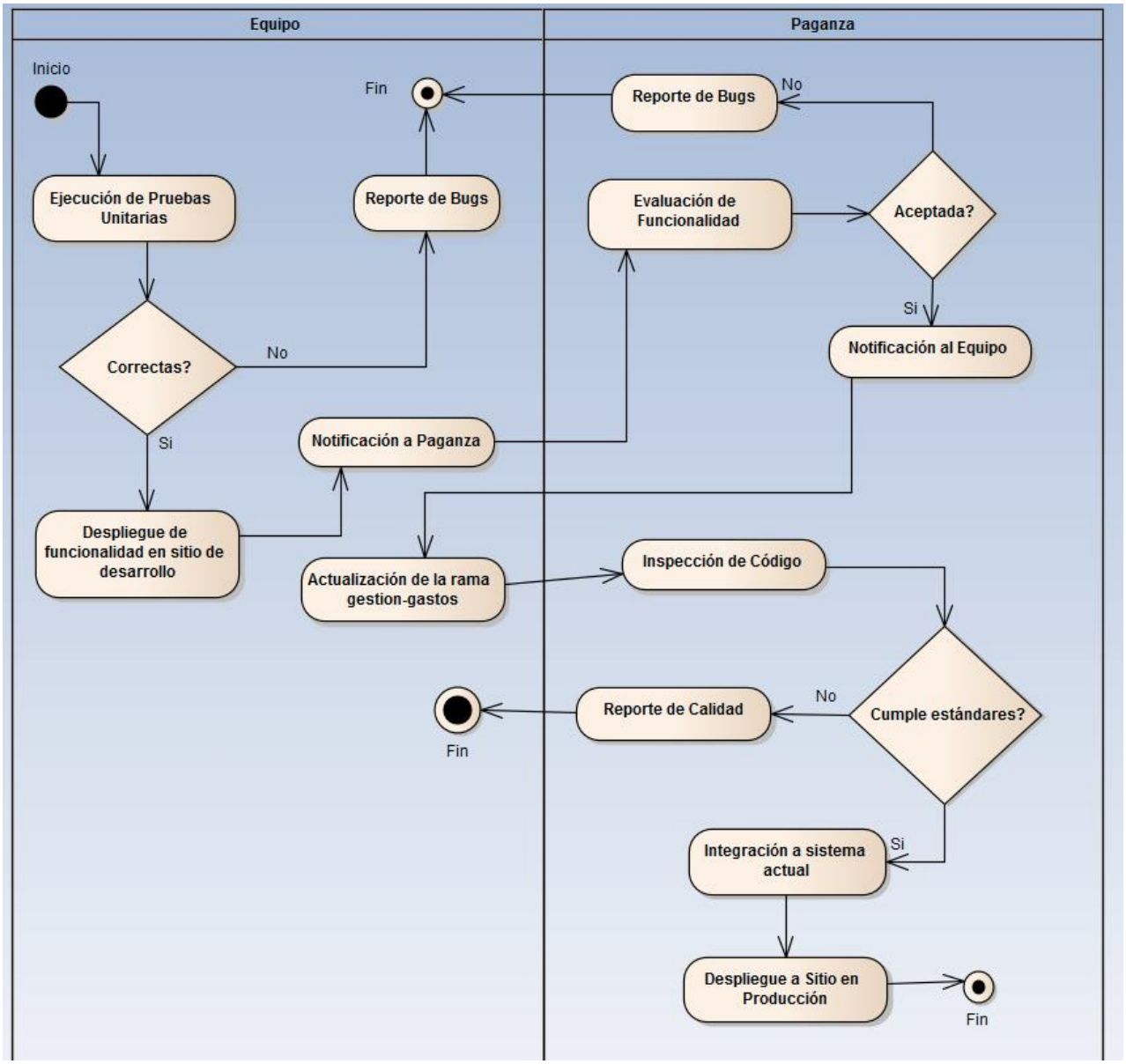

Figura 6-17: Diagrama de Actividad para el proceso de despliegue

7. Reflexión final

En primer lugar es importante destacar los resultados obtenidos como consecuencia de la ejecución del proyecto. Se logró construir una herramienta que permite la gestión de los gastos de forma eficiente e integrada con el producto de Paganza, logrando así uno de los objetivos más importantes del proyecto. Por otro lado, se valoró positivamente la obtención de conclusiones y resultados concretos mediante la investigación realizada utilizando prototipos funcionales. Además, la integración de sistemas desarrollados en diferentes plataformas es considerada como un desafío superado exitosamente.

En segundo lugar a lo largo de la ejecución del proyecto se produjeron ciertos acontecimientos que se considera oportuno mencionar. Uno de los eventos más importantes ocurridos fue la puesta en producción de las funcionalidades de análisis y billetera en una etapa temprana del proyecto, demostrando el cumplimiento de los objetivos de calidad propuestos. Esto además permitió que las funcionalidades mencionadas estuvieran accesibles para la totalidad de los usuarios de Paganza. Otro evento destacado durante el desarrollo del proyecto fue la finalización del MVP planificado, permitiendo así satisfacer el objetivo de entregar valor al cliente. Finalmente, una vez concluido el trabajo sobre la prototipación en la plataforma móvil, se realizó una presentación formal con el cliente donde fue transferido el conocimiento y los resultados obtenidos. Con esto el equipo consiguió cerrar esta etapa del proyecto satisfactoriamente.

En lo que respecta a la experiencia obtenida por parte del equipo es importante hacer mención a los conocimientos adquiridos y la satisfacción personal alcanzada, gracias a la superación de desafíos en forma conjunta. Por otro lado se considera valiosa la generación de contactos profesionales y relacionamiento con Paganza. A su vez, la conformación de un equipo integrado por compañeros cuya relación es estrecha desde el comienzo de la carrera contribuyó al desarrollo exitoso del proyecto.

Por último, considerando las etapas futuras del proyecto se espera que las funcionalidades restantes sean también integradas a Paganza, y que a partir de ellas el producto pueda satisfacer necesidades de los usuarios aumentando el valor percibido.

8. Referencias bibliográficas

[1] "Paganza | La forma inteligente de pagar tus cuentas." [En línea]. Disponible en: http://www.paganza.com/. [Accedido: 25-ago-2015].

[2] "Ley 19.210." [En línea]. Disponible en: http://www.parlamento.gub.uy/leyes/AccesoTextoLey.asp?Ley=19210&Anchor=. [Accedido: 23-ago-2015].

[3] "Más uso de pagos electrónicos y menos de los tradicionales | Noticias Uruguay y el Mundo actualizadas - Diario EL PAIS Uruguay." [En línea]. Disponible en: http://www.elpais.com.uy/economia/noticias/mas-pagos-electronicos-menos-tradicionales.html. [Accedido: 23-ago-2015].

[4] "El público se adueñó del producto." [En línea]. Disponible en: http://www.elpais.com.uy/el-empresario/publico-se-adueno-producto.html. [Accedido: 22-ago-2015].

[5] "Design Thinking en Español." [En línea]. Disponible en: http://www.designthinking.es/inicio/index.php. [Accedido: 12-ago-2015].

[6] M. O'Neill, "Scrum, Minimum Viable Product and Operation Overnight", *GeoVoices*. [En línea]. Disponible en: http://geovoices.geonetric.com/2013/10/scrum-minimum-viable-product-and-operation-overnight/. [Accedido: 23-ago-2015].

[7] "Ocr Api Service - Iphone - Android - Blackberry - Windows Phone - Web apps." [En línea]. Disponible en: http://ocrapiservice.com/. [Accedido: 26-ago-2015].

[8] M. Alaimo y M. Salias, *Proyectos Ágiles con Scrum: Flexibilidad, aprendizaje, innovación y colaboración en contextos complejos*, 2 edition. Kleer, 2013.

[9] "Scrum Orgánico: Mi primera vez | Scrum Orgánico." [En línea]. Disponible en: http://www.scrumorganico.com/article/scrum-org%C3%A1nico-mi-primera-vez. [Accedido: 23-ago-2015].

[10] R. C. MARTIN, *Clean code: A Handbook of Agile Software Craftsmanship*, Edición: 1. Upper Saddle River, NJ: PRENTICE HALL, 2008.

[11] "Introduction — Selenium Documentation." [En línea]. Disponible en: http://www.seleniumhq.org/docs/01_introducing_selenium.jsp. [Accedido: 24-ago-2015].

[12] "10 Heuristics for User Interface Design: Article by Jakob Nielsen." [En línea]. Disponible en: http://www.nngroup.com/articles/ten-usability-heuristics/. [Accedido: 19-ago-2015].

[13] R. V. Solingen y E. Berghout, *Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. London; Chicago: McGraw-Hill Inc.,US, 1999.

[14] "Code Metrics Values." [En línea]. Disponible en: https://msdn.microsoft.com/en-us/library/bb385914.aspx. [Accedido: 24-ago-2015].

[15] "Maintainability Index Range and Meaning - Code Analysis Team Blog - Site Home - MSDN Blogs." [En línea]. Disponible en: http://blogs.msdn.com/b/codeanalysis/archive/2007/11/20/maintainability-index-range-and-meaning.aspx. [Accedido: 24-ago-2015].

[16] "Gitflow Workflow", *Atlassian Git Tutorial*. [En línea]. Disponible en: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. [Accedido: 23-ago-2015].

[17] “Introduction to Mobile Development - Xamarin.” [En línea]. Disponible en: http://developer.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development/. [Accedido: 24-ago-2015].
[18] “Xamarin.Forms - Xamarin.” [En línea]. Disponible en: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/. [Accedido: 24-ago-2015].

9. Anexos

9.1. Ejemplo de entrevista para usuarios

Si es usuario actual de Paganza:

1- ¿Cómo utiliza Paganza?

    a. ¿Qué paga con Paganza?

    b. ¿Cuándo agenda los pagos?

2- ¿Qué datos de sus gastos le gustaría ver en Paganza?

3- ¿Pagaría más gastos con Paganza si pudiera controlar sus gastos más eficientemente?

4- ¿Ingresaría otros gastos si se integraran a los ya registrados?

5- ¿Qué gastos que actualmente no se pueden registrar le gustaría agregar?

Si es usuario actual y tiene otras formas de registrar gastos

1- ¿Cómo registra sus gastos?

2- ¿Cuáles considera son las ventajas y desventajas?

3- ¿Qué gastos registra y por qué?

4- ¿Cuáles gastos no registra y por qué?

Si es usuario actual de Paganza pero no registra sus gastos

1- ¿Por qué no registra sus gastos?

2- ¿Le gustaría registrarlos si fuera fácil?

Si no son usuarios de Paganza

1- ¿Cómo paga sus facturas?

2- ¿Cómo registra sus gastos?

3- ¿Si Paganza le permitiera registrar y analizar sus gastos, la utilizaría?

9.2. Prototipos de baja resolución realizados para plataforma móvil

9.2.1. Menú

Figura 9-1: Boceto de menú móvil

9.2.2. Registro de Gasto Manual

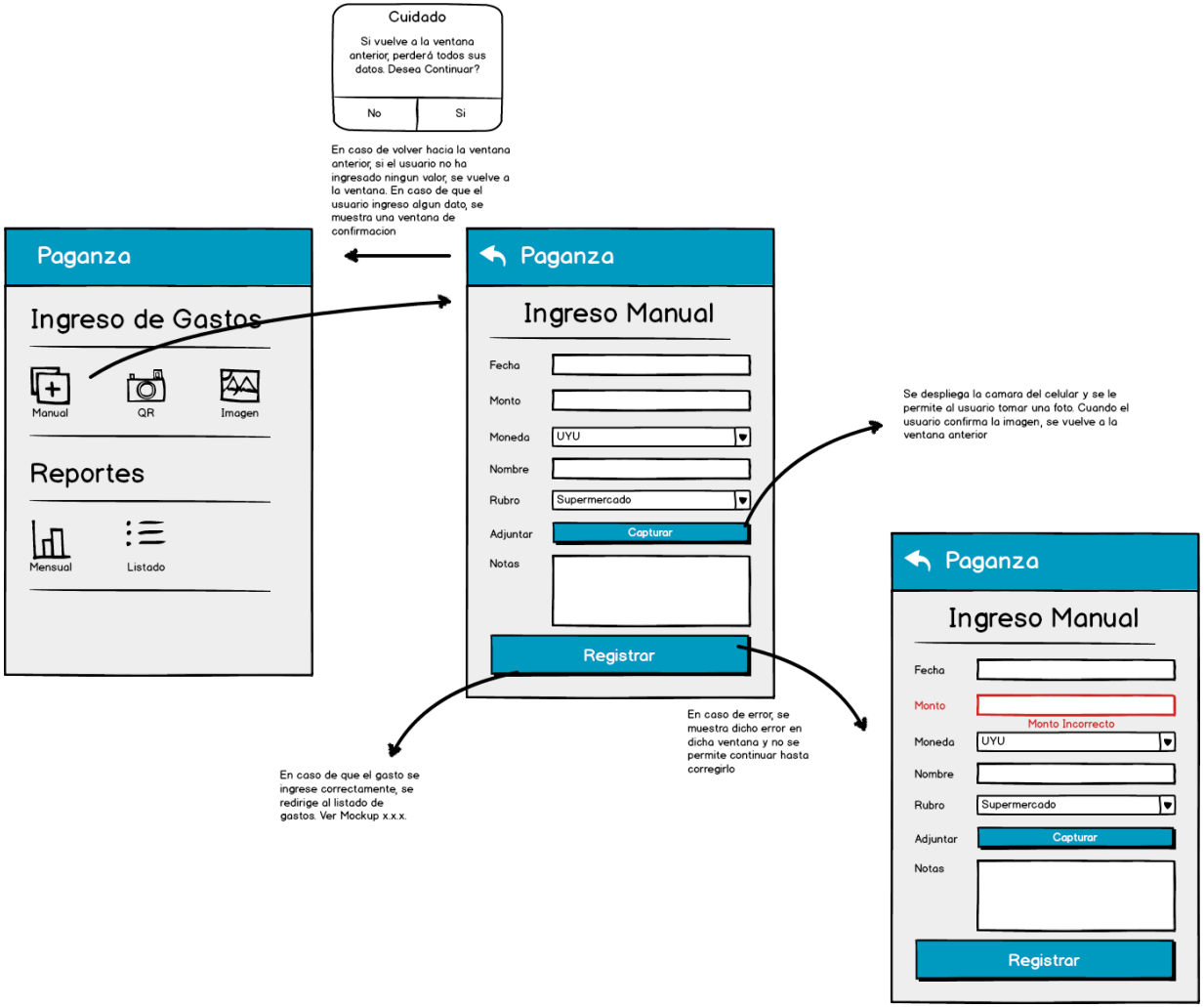

Figura 9-2: Boceto de flujo de ingreso de gasto

9.2.3. Reporte Mensual de gastos

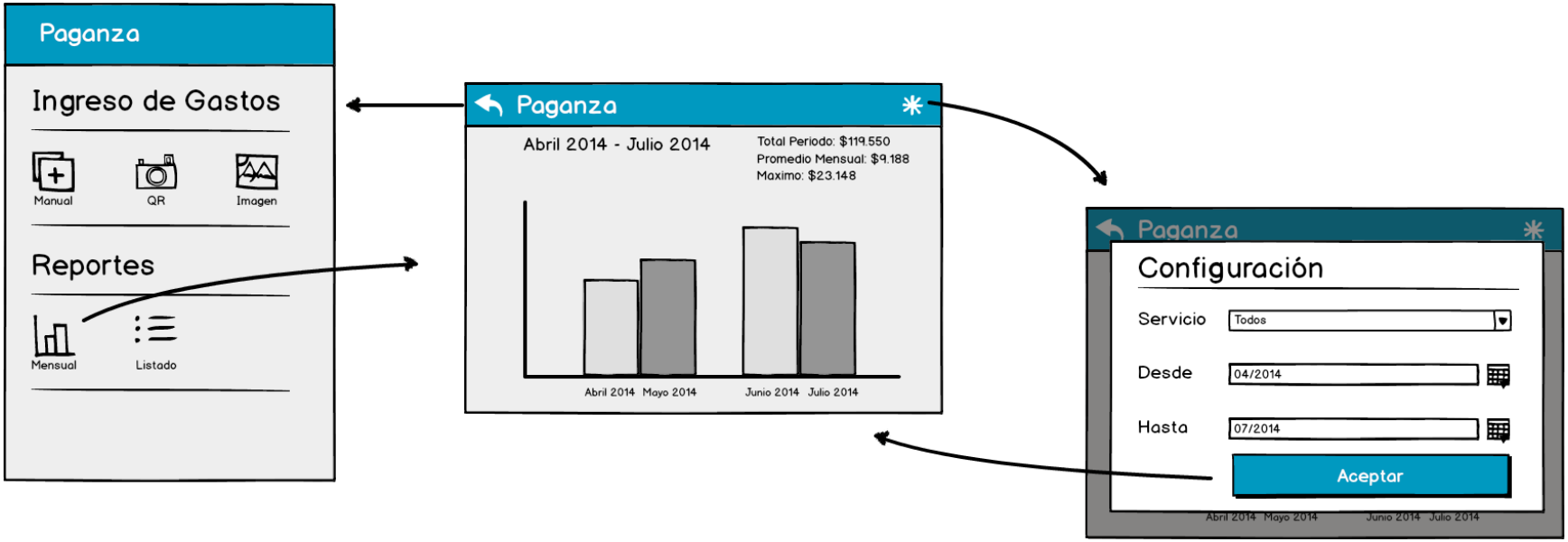

Figura 9-3: Boceto de visualización de reporte mensual

9.2.4. Listado de Gastos

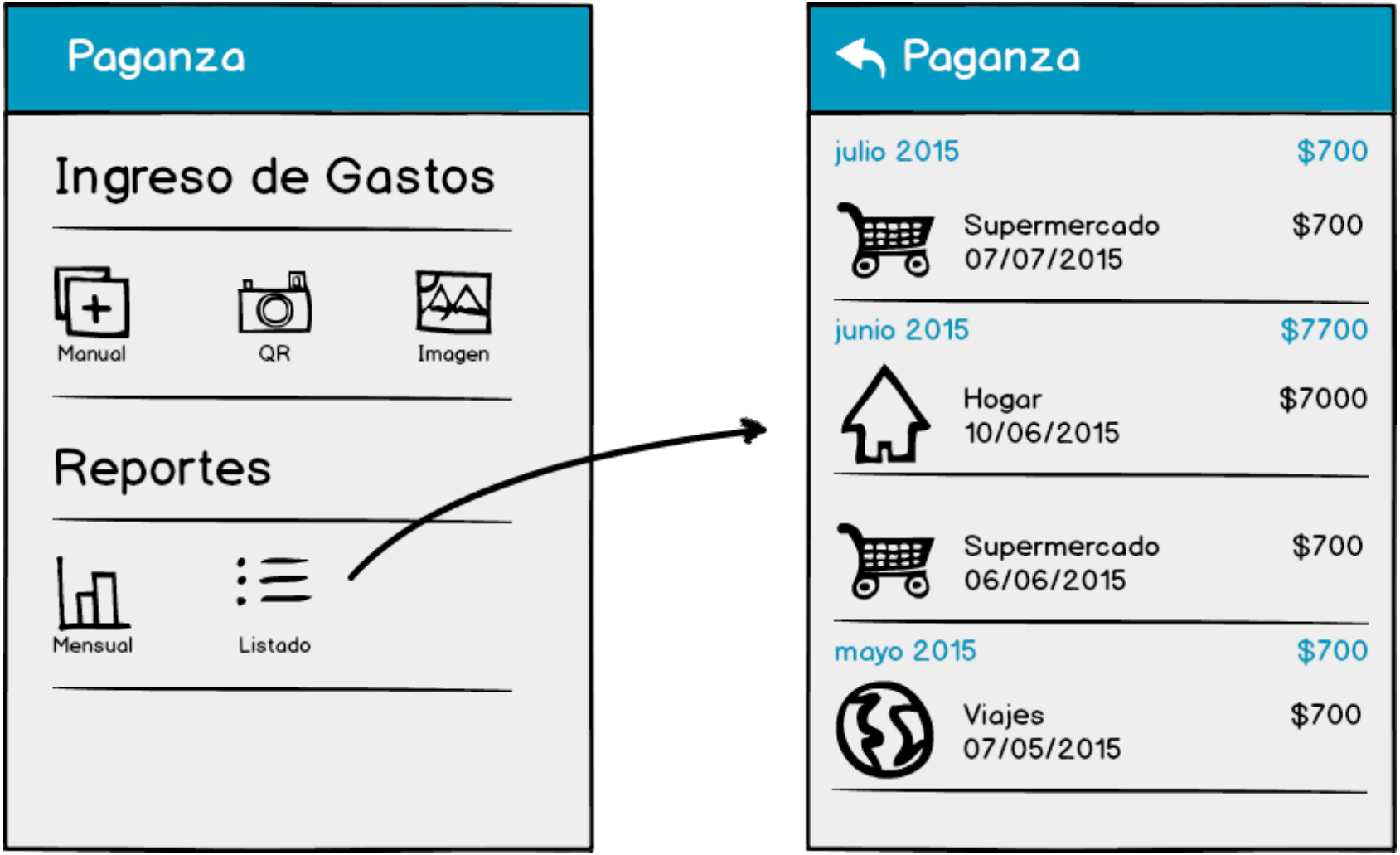

Figura 9-4: Boceto de listado de gastos

9.2.5. Login

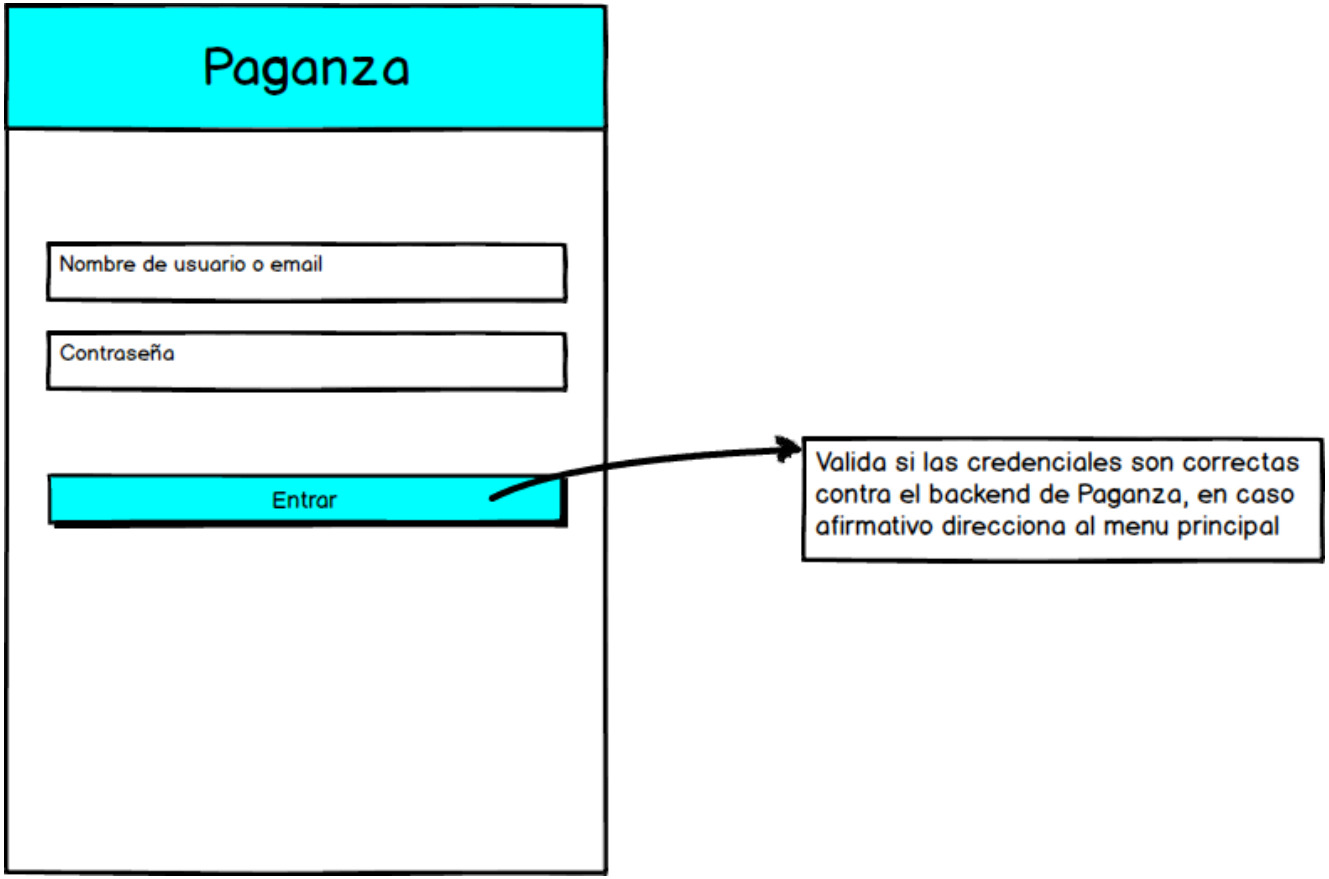

Figura 9-5: Boceto de ingreso de sesión

9.2.6. Seguridad

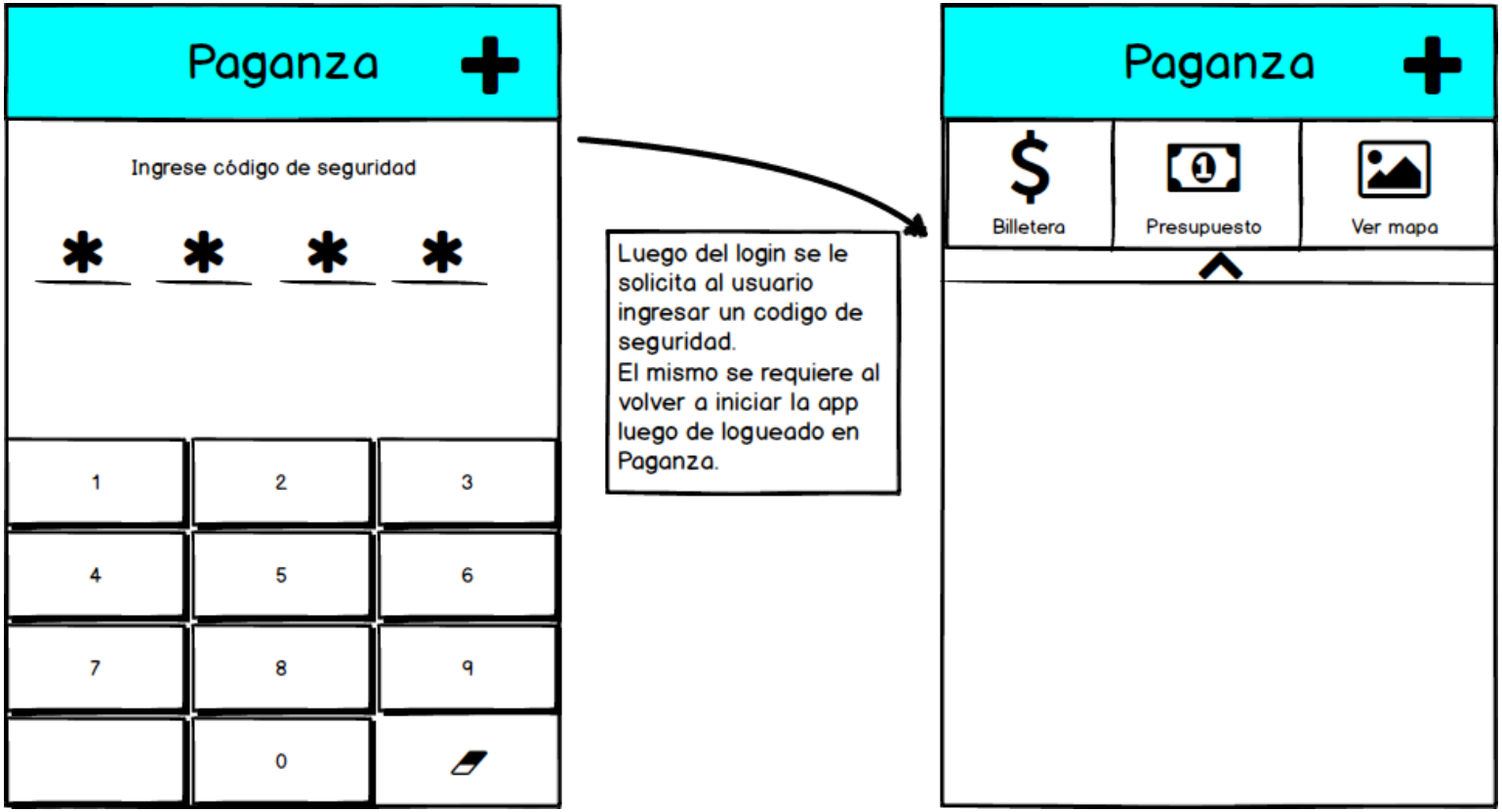

Figura 9-6: Boceto de seguridad móvil

9.2.7. Home

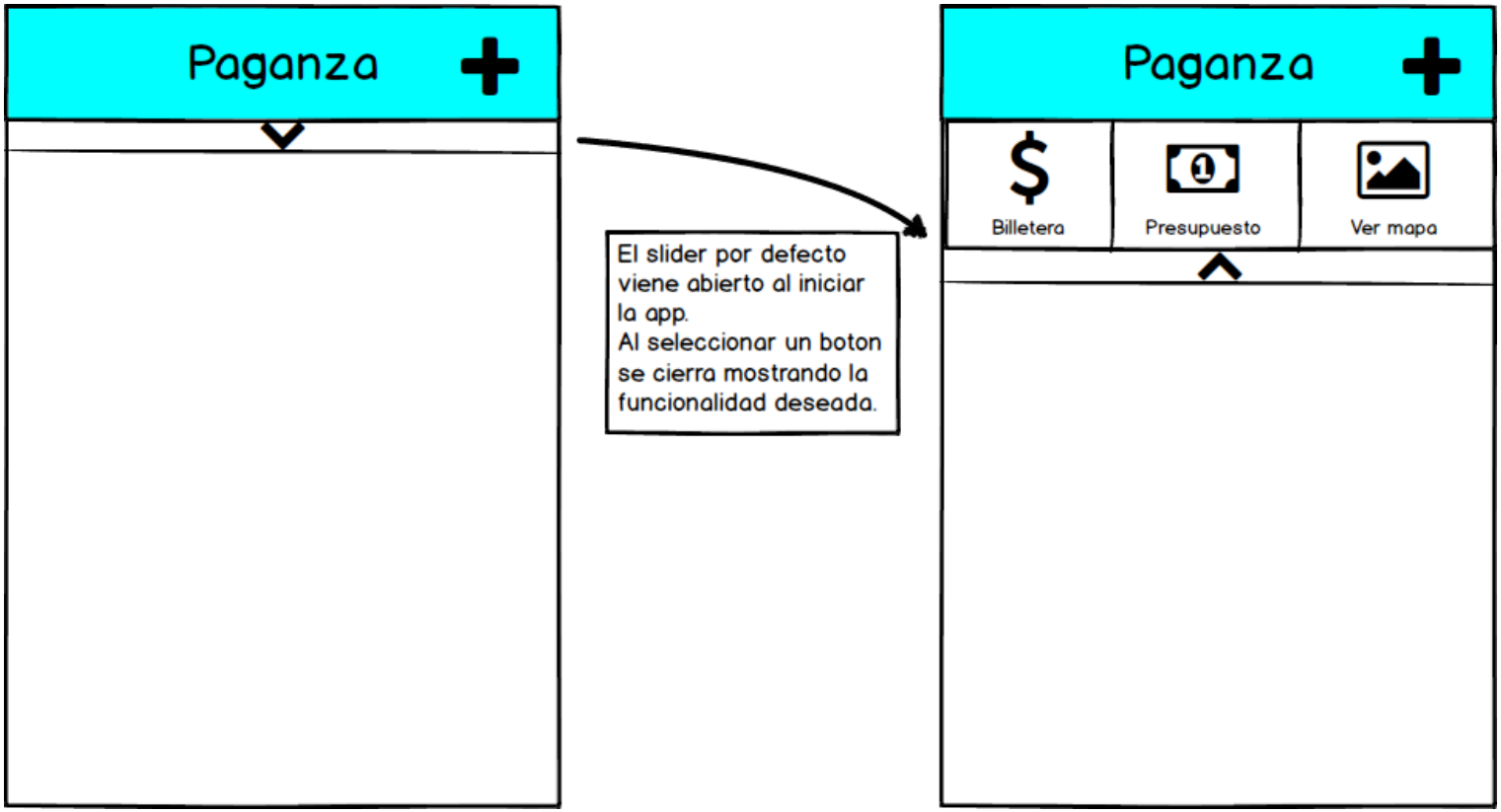

Figura 9-7: Boceto de pantalla de inicio

9.2.8. Mapa

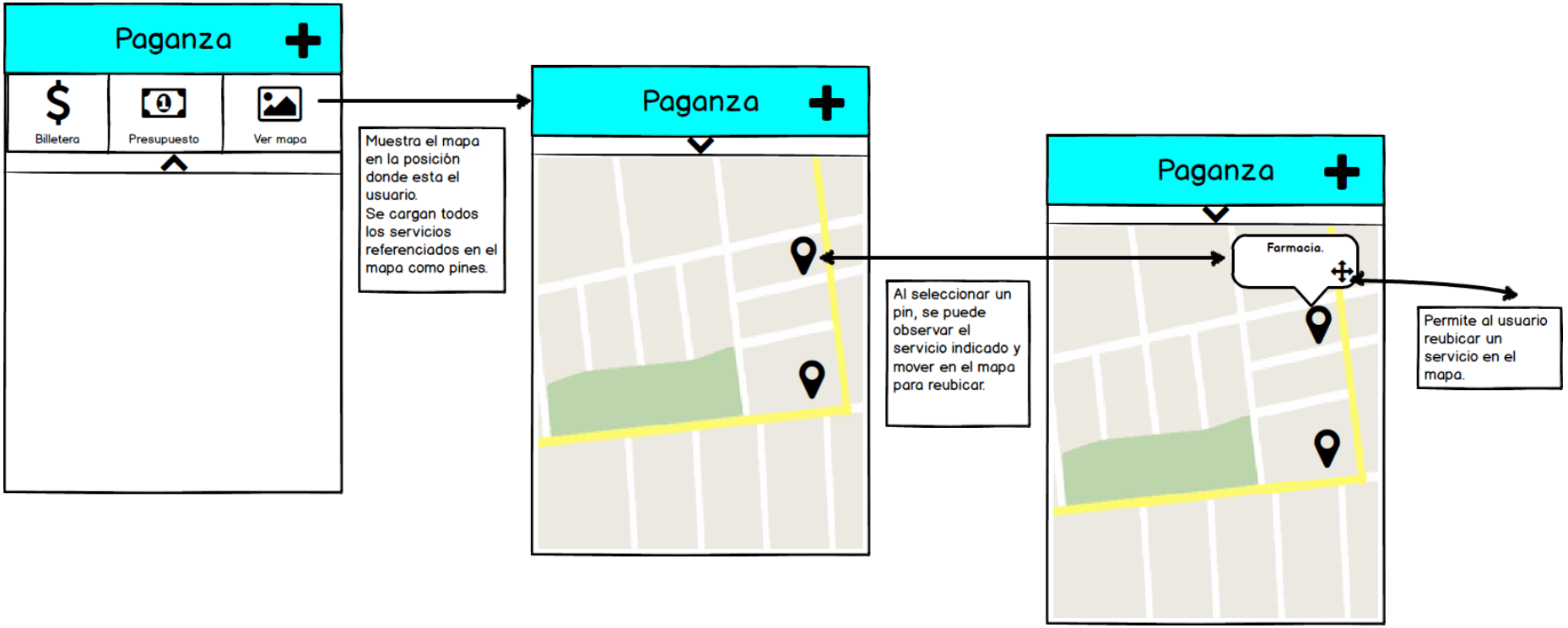

Figura 9-8: Boceto de funcionalidad de geolocalización

9.3. Guía de documentación de vistas

En esta guía se detalla de qué manera se documentó las vistas del sistema. Cada una de las vistas sigue la siguiente convención:

1. Contiene un nombre.

2. Representación primaria: se incluyen los diagramas pertinentes a la vista donde se ubican las entidades más relevantes para dicha vista. En caso de que aplique, se separan los diagramas por plataforma, siendo estas web y móvil.

3. Catálogo de Elementos: enumera las entidades que fueron representadas en la representación primaria, con nombre y responsabilidad. En caso de que aplique, se separan los elementos por plataforma, siendo estas web y móvil.

4. Justificaciones de diseño: se detallan las justificaciones de diseño pertinentes a la vista actual.

9.4. Estándares de codificación

9.4.1. Paganza

- Todos los métodos que exponen funcionalidad sobre la base de datos deben ser asíncronos, por lo cual deben:
    - Hacer uso de async.
    - Deben retornar Task<T>.
- Todas las clases que realicen acceso a la base de datos deben implementar la clase BusinessComponent.
- En caso de tener que agregar entidades o atributos a entidades existentes a la base de datos, se debe utilizar como ORM la herramienta Entity Framework, con la metodología Code First. Deben generar Migrations auto explicativas para cada uno de los cambios realizados.
- Todas las clases que representan nuevas entidades en el sistema deben heredar de la clase abstracta EntityBase.
- A la hora de realizar testing unitario, se debe reutilizar la manera en que Paganza ya realiza dicho testing, haciendo uso de las clases Mock, ICoreFactory y IDbContext.
- Se debe respetar la estructura de carpetas de la solución actual, en donde cada carpeta engloba una funcionalidad en particular.
- Todos los métodos creados a nivel de Controllers deben detectar, de la misma manera que Paganza ya lo realiza, si un usuario a iniciado sesión antes de realizar cualquier operación.
- A la hora de desarrollar una nueva funcionalidad, se debe la existencia de una funcionalidad similar en el sistema y observar de que manera esta desarrollado e intentar desarrollarlo de la misma manera. Ejemplos:

    - Listado de Gastos externos debe estar implementado de manera similar al listado de facturas.
    - Los nuevos links de la barra de navegación de la izquierda del sitio deben estar implementados de la misma manera que los links actuales.
    - El filtrado de gastos externos se debe realizar de la misma manera que el listado de facturas.

- Hacer énfasis en el uso de Partial Views.
- Se debe mantener la nomenclatura de los paquetes de Paganza. Esto es, a la hora de crear un nuevo Paquete, se lo debe nombrar Paganza.<nombre_paquete>
- Se debe hacer reuso de la mayor cantidad de entidades posibles de Paganza. Prestar importante atención al uso de Enums, mayoritariamente incluidos en la clase Enums y Currencies, así como también el uso de Países ubicados en la clase Countries.

9.4.2. Clean Code

- Consistencia: si se realiza algo de cierta manera, hacer cosas similares de la misma manera.
- Usar variables auto explicativas, utilizando nombres que describan que significa la variable. En caso de ser booleanos, escribirlos como si fuesen preguntas.
- Constructores en vez de setters.
- Los métodos no pueden sobrepasar las 30 líneas.
- Mantener todo lo relacionado a la misma feature en el mismo namespace.
- Evitar evaluaciones booleanas de más de una condición. En caso de existir, pasarlas a un método.
- Devolver excepciones en vez de nulls.

9.5. Justificaciones de diseño web

9.5.1. Reportes

9.5.1.1. Frontend

Para el despliegue del reporte de análisis se utilizó la librería Google Charts. Esta librería permite la visualización de datos en representación gráfica, en forma de gráfica de tortas, gráfica de columnas y más. Existe una extensiva documentación que permite una fácil utilización de la librería, y una gran comunidad detrás.

Esta librería necesita ser provista con listas de datos organizados de cierto modo para su correcto despliegue. Estos datos se obtienen a través de una API con la que se comunica el código interpretado JavaScript, para su posterior formateo para producir la entrada que necesita la librería para desplegar los datos de forma correcta.

9.5.1.2. Arquitectura diseñada

A partir de la arquitectura diseñada para las extensiones, se creó el componente llamado Paganza.Reports. A continuación, se detalla de qué manera se implementó la arquitectura para dicha funcionalidad. Analizando los requerimientos de Paganza, se identificaron las siguientes características de esta funcionalidad:

- Se trata de una consulta de datos al sistema, por lo cual no será necesario realizar modificaciones sobre la base de datos
- Debido a la necesidad de mostrar la información haciendo uso de gráficas, será necesario utilizar una librería de terceros para poder renderizar la información. A partir de esto, se puede concluir que será necesario utilizar APIs para poder obtener la información necesaria desde el *backend*
- Como se trata de un nuevo requerimiento sobre una funcionalidad no existente, para la integración con Paganza se creará un nuevo controlador en Paganza.Website.UI.

- Existen parámetros que se deben tener en cuenta a la hora de realizar la consulta, por lo cual el componente tendrá que realizar las validaciones correspondientes de esos parámetros

- Debido a que se trata de una funcionalidad para usuarios que hayan iniciado sesión en el sistema, toda operación que se realice en el nuevo controlador a crear deberá validar que exista un usuario en sesión.

9.5.1.3. Access Layer

Analizando los requerimientos, se puede observar que existen dos tipos de reportes a realizar, siendo uno obtener los gastos entre ciertas fechas agrupando por servicio, y el otro un reporte de los gastos mensuales, agrupando por servicio. Para satisfacer estas necesidades, se creó la interfaz IReports que provee los siguientes servicios:

- GetExpensesReport: método que será invocado para obtener los gastos entre ciertas fechas agrupados por servicio. Recibe como parámetros:

    - UserId: id del usuario de Paganza .

    - Year: año desde el cual se desea hacer la consulta.

    - Month: mes desde el cual se desea hacer la consulta.

    - MonthTo: mes de hasta el cual se desea hacer la consulta. Este parámetro puede llegar a ser nulo. En caso de que serlo, la consulta se realizara hasta 12 meses después del mes y año establecido en los dos parámetros anteriores.

    - YearTo: año de hasta el cual se desea hacer la consulta. Este parámetro puede llegar a ser nulo. En caso de que serlo, la consulta se realizara hasta 12 meses después del mes y año establecido en los dos parámetros anteriores.

    - ServiceId: id del servicio que se desea consultar. Este parámetro será usualmente nulo, en caso de serlo se devuelven todos los servicios.

    - Currency: string que representa el tipo de moneda sobre la que se quiere hacer la consulta.

- GetMonthlyReport: método que será invocado para obtener los gastos mensuales agrupados por servicio.
    - UserId: id del usuario de Paganza.
    - Year: año desde el cual se desea hacer la consulta.
    - Month: mes desde el cual se desea hacer la consulta.
    - MonthTo: mes de hasta el cual se desea hacer la consulta. Este parámetro puede llegar a ser nulo. En caso de que serlo, la consulta se realizara hasta 12 meses después del mes y año establecido en los dos parámetros anteriores.
    - YearTo: año de hasta el cual se desea hacer la consulta. Este parámetro puede llegar a ser nulo. En caso de que serlo, la consulta se realizara hasta 12 meses después del mes y año establecido en los dos parámetros anteriores.
    - ServiceId: id del servicio que se desea consultar. Este parámetro será usualmente nulo, en caso de serlo se devuelven todos los servicios.
    - Currency: string que representa el tipo de moneda sobre la que se quiere hacer la consulta.

9.5.1.4. DataAccess Layer

En esta capa se crea la lógica para realizar el acceso a base de datos. Se crea la clase AnalysisReport, que hereda de BusinessComponent, y que define dos métodos, cada uno para poder satisfacer las dos funcionalidades de reportes.

9.5.1.5. ResponseClasses Layer

Debido a la naturaleza de la funcionalidad, y ya que será necesario utilizar librerías de terceros para exponer, se llegó a la conclusión de que será necesario implementar APIs para poder acceder a la información y poder realizar diferentes consultas y recargar la información que se le despliega al usuario. Por lo tanto, el componente deberá retornar información en un formato simple y que permita ser convertido a información de *frontend*. Como se decidió utilizar Google Charts librería que recibe como datos de entrada un JSON,

se concluye que la información que debe devolver el componente debe ser convertible a JSON.

Para la representación de la información para el reporte de gastos agrupados por servicio, se retorna una instancia de la clase MonthResponse, donde se tiene:

- Data: atributo que devuelve la información necesaria para desplegar la grafica. Se trata de una lista de MonthlyReport, que contiene toda la información de los gastos por mes para cada servicio.

- Logos: diccionario en donde se incluye los nombres de los archivos que contienen los logos de los servicios, para ser desplegados en *frontend.*

- Services: diccionario con información sobre los servicios que permite manejar de una manera más sencilla la información en *frontend*.

Luego, ExpenseReport será utilizada para devolver la información necesaria para mostrar el reporte de gastos mensuales.

Por último, en caso de que el componente reciba información inválida, se creó una clase llamada ErrorResponse, a modo de que el *frontend* reciba una respuesta amigable en caso de que se envíe un parámetro incorrecto.

9.5.1.6. Utilities Layer

Debido a que el servicio expuesto recibe varios parámetros, sobre todo parámetros que representan fechas, se debe hacer una validación de que estos parámetros sean correctos. Para ello se crea la clase ParamsValidation, donde se valida que el año y el mes sean correctos.

9.5.2. Etiquetas

9.5.2.1. Backend

Para el desarrollo de las etiquetas, se siguió la arquitectura diseñada para los componentes del proyecto. Se desarrolló un componente con las 4 capas propuestas previamente, las cuáles se detallan a continuación:

9.5.2.2. Access Layer

En esta capa, se definen dos tipos de contratos:

- ITags: interfaz que provee métodos para alta, baja y modificación de etiquetas. Además, permite asociar etiquetas a facturas existentes.
- IQueryTags: interfaz que provee los métodos para consultar las etiquetas, así como obtener las etiquetas de cierta factura.

9.5.2.3. DataAccess Layer

En esta capa, se crea la clase TagsAccess. Dicha clase será la encargada de recibir todos los pedidos de las implementaciones de las interfaces de la capa de acceso, y procesarlas.

En resumen, crea las etiquetas según los parámetros recibidos y guarda la información en la base de datos. Así mismo, realiza baja y modificación de dichas etiquetas. Por otro lado, tiene los métodos que permiten consultar las etiquetas, donde se hacen consultas a la base de datos utilizados LINQ para obtener esas etiquetas.

9.5.2.4. ResponseClasses Layer

Para simplificar la respuesta al *frontend* de los datos de las etiquetas, se crearon dos clases de ayuda:

- SingleTag: clase que representa una etiqueta en particular. Contiene el identificador de dicha etiqueta, así como también el nombre.
- InvoiceTags: clase que representa las etiquetas que tiene una factura en particular. Contiene una lista de SingleTags.

9.5.2.5. Interacción con *Frontend*

Para realizar la interacción con el *frontend*, se creó una API la cual permite dar de alta etiquetas así como también modificarlas y eliminarlas. También permite agregar etiquetas a facturas existentes y consultar las mismas.

Dicha API se creó dentro de un controlador en el proyecto Paganza.Website.UI, y expone estos servicios para que el *frontend* los consuma. En resumen, desde *frontend* se invocan dichos servicios utilizando AJAX, y dependiendo de la respuesta (JSON), se muestran las etiquetas creadas para esa factura.

9.5.3. Gastos

9.5.3.1. Backend

Para el desarrollo de los gastos externos, se siguió la arquitectura diseñada para los componentes del proyecto. Se desarrolló un componente con las 4 capas propuestas previamente, las cuáles se detallan a continuación:

9.5.3.2. Access Layer

En esta capa, se definen dos tipos de contratos:

- IExpenses: interfaz que provee métodos para alta, baja y modificación de gastos externos.
- IExpenseService: interfaz que provee métodos para alta, baja y modificación de servicios o rubros.
- IExpenseFiles: interfaz que provee métodos para alta, baja y modificación de archivos asociados a los gastos.
- IQueryExpenses: interfaz que provee los métodos para consultar los gastos externos.

9.5.3.3. DataAccess Layer

En esta capa, se crean las clases ExpensesAccess y ExpenseServiceAccess. Dichas clases serán las encargadas de recibir todos los pedidos de las implementaciones de las interfaces de la capa de acceso, y procesarlas.

En resumen, crean los gastos y servicios según los parámetros recibidos y guarda la información en la base de datos. Así mismo, realiza baja y modificación de dichos gastos o

servicios. Por otro lado, tienen los métodos que permiten consultar los gastos externos, donde se hacen consultas a la base de datos utilizado LINQ para obtener esos gastos.

9.5.3.4. ResponseClasses Layer

Para simplificar la respuesta al *frontend* de los datos de los gastos, se crearon dos clases de ayuda:

- DetailedExpense: clase que tiene la información detallada de un gasto. Se utiliza para mostrar toda la información relevante a un gasto.
- SimpleExpense: clase que tiene información abreviada de un gasto para que este sea mostrado una lista.

9.6. Xamarin

9.6.1. Introducción

El propósito de este documento es describir las características técnicas de Xamarin como plataforma de desarrollo móvil. También se explicará en este documento las características del producto Xamarin Forms sobre el cual Paganza planteó particular interés y que la empresa no había investigado aún. Además se incluirán conclusiones específicas de la utilización de esta tecnología dentro del marco del proyecto, como por ejemplo resultados de reutilización de código y desafíos encontrados durante el desarrollo.

9.6.2. Definición de Xamarin

Xamarin es una plataforma de desarrollo de aplicaciones móviles *cross-platform*. En cuanto al desarrollo de aplicaciones móviles, *cross-platform* implica la capacidad de poder ejecutar la misma aplicación en diferentes ambientes de ejecución móvil. Esto está directamente relacionado con los sistemas operativos en los cuales puede ser ejecutada. En el caso de Xamarin, es posible desarrollar y ejecutar aplicaciones para los tres sistemas operativos más representativos a nivel móvil, Android, iOS y Windows Phone.

La ventajas más importantes de esta tecnología es la posibilidad de reutilización de código, que impacta directamente sobre la mantenibilidad. Es notorio que requiere menos esfuerzo mantener una única aplicación para todas las plataformas que mantener tantas aplicaciones como plataformas sea necesario soportar. Dentro del desarrollo *cross-platform* existen otras alternativas que permiten alcanzar este objetivo de reutilización, como Sencha, PhoneGap o Appcelerator. Sin embargo, frente a estas alternativas Xamarin tiene algunas ventajas relacionadas mayormente con la performance, gracias al uso de código compilado y no interpretado. Estas características propias de Xamarin y que diferencian esta tecnología otros métodos de desarrollo *cross-platform* son:

- Amplia cobertura de las SDKs nativas dentro de su implementación, utilizando relacionamiento fuertemente tipados con estas SDKs. Esto permite explorarlas más fácilmente y evitar errores en tiempo de ejecución mediante la detección de errores en tiempo de compilación.

- Posibilidad de uso directo de código nativo. Xamarin permite la invocación de código desarrollado en librerías de terceros o previamente implementadas en código nativo.

- IDEs específicos multiplataforma. Xamarin utiliza Microsoft Visual Studio en Windows y Xamarin Studio para Mac OS.

9.6.3. Aplicación en el marco del proyecto

El equipo no encontró dificultades significativas en la adopción de la tecnología. Se reconoció la existencia de una curva de aprendizaje pronunciada, pero una vez familiarizado con los mecanismos adecuados para implementar las funcionalidades se pudieron resolver todos los requerimientos técnicos planteados. Se encontró valioso haber realizado pruebas de concepto técnicas para garantizar la factibilidad de las funcionalidades a desarrollar, como por ejemplo una aplicación básica con interacción con la cámara del dispositivo o con las funcionalidades de geolocalización. Fue un factor positivo para el equipo que el desarrollo se realice utilizando el lenguaje C#, que era conocido por todos los integrantes. Se consideró útil la posibilidad de desarrollar y probar en los simuladores integrados a la plataforma, con capacidades de *debugging* completas y que funcionan correctamente. Se realizó de forma fácil y exitosa la instalación de la aplicación desarrollada en diferentes dispositivos con diferentes versiones del sistema operativo y diferentes resoluciones de pantalla.

9.6.4. Xamarin Forms

Forms es una herramienta nueva de Xamarin, posterior a su producto de desarrollo *cross-platform*. Esta herramienta está centrada en el desarrollo compartido de la interfaz de usuario entre plataformas. Esto implica mejorar el aspecto de mantenibilidad ya mencionado, llevándolo al marco de la interfaz de usuario, que sin el uso de Forms debe hacerse de forma específica para cada plataforma. Xamarin Forms es un conjunto común de controles de interfaz de usuario, que permite manejar las diferentes plataformas con una única implementación. El uso de Xamarin Forms es una enorme ventaja desde el punto de vista de la reutilización del código, pero como contrapartida tiene limitada la interacción con código nativo y la flexibilidad y calidad del diseño de la interfaz de usuario.

9.6.5. Aplicación en el marco del proyecto

Dentro del marco del proyecto todo el desarrollo móvil se realizó con Xamarin Forms, siendo esta la forma en la que el equipo decidió realizar la investigación sobre la plataforma. Concretamente la experiencia se valora como positiva, obteniendo un alto porcentaje de reutilización de código. Además de una aplicación con un diseño de interfaz de usuario satisfactorio, considerando que no era el foco del desarrollo. Las pruebas de concepto y prototipos desarrollados fueron satisfactorios en cuanto a factibilidad técnica, ya sea mediante librerías provistas por terceros o características propias del *framework*, todos los desafíos planteados pudieron ser resueltos. Se encontraron inconvenientes durante el desarrollo en dos grandes aspectos. Por un lado son limitados y poco documentados los complementos desarrollados por terceros disponibles. Por otro lado algunas librerías que permiten interacciones con uso de características nativas están en fase de pruebas beta, las que se pudo constatar su inestabilidad.

9.7. Registro histórico del tiempo

	Planificado		Real				
Sprint	Puntos Estimados	Capacidad	Realizado	Velocidad Normalizada	Nº Bugs resueltos	Inicio	Fin
1	47	67%	47	70	0	21/11/14	11/12/14
2	104	83%	80	96	10	12/12/14	01/01/15
3	109	100%	109	109	8	12/01/15	01/02/15
4	60	69%	60	87	10	02/02/15	13/02/15
5	65	68%	65	96	12	16/02/15	27/02/15
6	90	100%	85	85	10	02/03/15	13/03/15
7	70	83%	70	84	8	16/03/15	27/03/15
8	73	80%	70	88	5	30/03/15	10/04/15
9	75	85%	75	88	4	13/04/15	24/04/15
10	76	90%	73	81	6	27/04/15	08/05/15
11	99	80%	70	88	6	11/05/15	22/05/15
12	80	90%	70	78	5	25/05/15	05/06/15
13	76	95%	73	77	3	08/06/15	19/06/15
14	89	90%	85	94	4	22/06/15	03/07/15
15	70	95%	63	66	8	06/07/15	17/07/15
16	60	100%	57	57	2	20/07/15	31/07/15
17	0	100%	0	0	2	03/08/15	14/08/15
18	0	100%	0	0	0	17/08/15	26/08/15

Tabla 9-1: Registro histórico del tiempo

9.8. Guía de configuración del entorno local de desarrollo

A continuación se incluyen los pasos necesarios para configurar la base de datos en el ambiente local.

1. Instalar SQL Management Studio. Puede descargarse la versión gratuita desde el siguiente link. http://msdn.microsoft.com/en-us/evalcenter/dn434042.aspx.
2. Se recomienda el uso de **LocalDb** como motor de base de datos local. LocalDb es similar a SQL Express, con la diferencia que en lugar de ser un servicio que esta constantemente corriendo, se "prende" a demanda, cuando se conecta a la base de datos, de modo que es aún más "económico".
3. Luego en el SQL Management Studio importar el backup (.bacpac). Este archivo está almacenado en Google Drive.
4. Para importar la base de datos seguir los siguientes pasos:

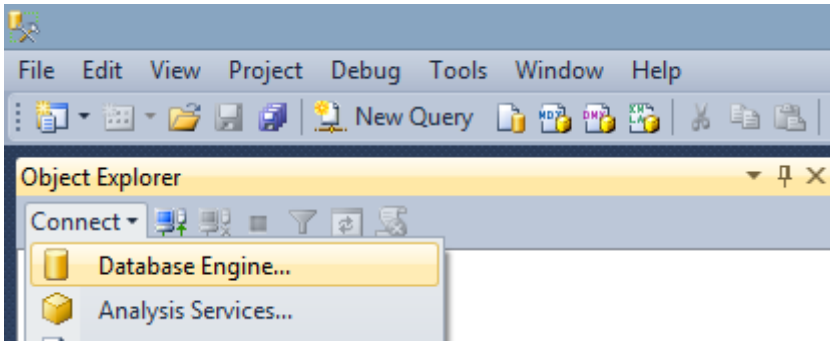

Figura 9-9: Paso 1

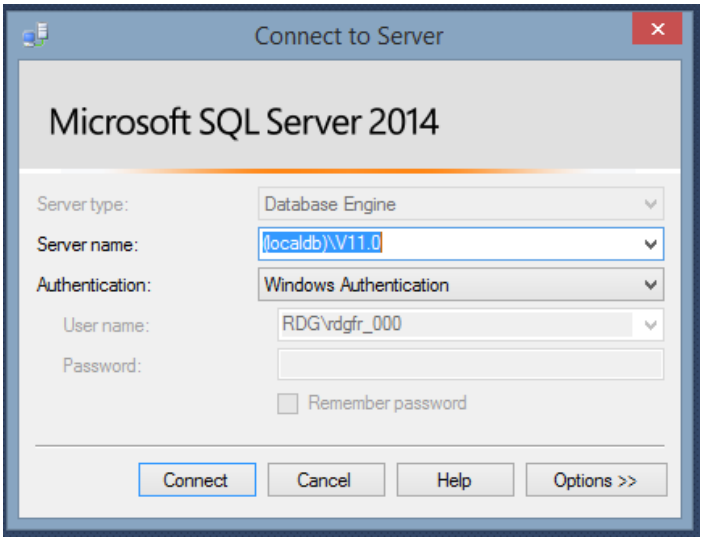

Figura 9-10: Paso 2

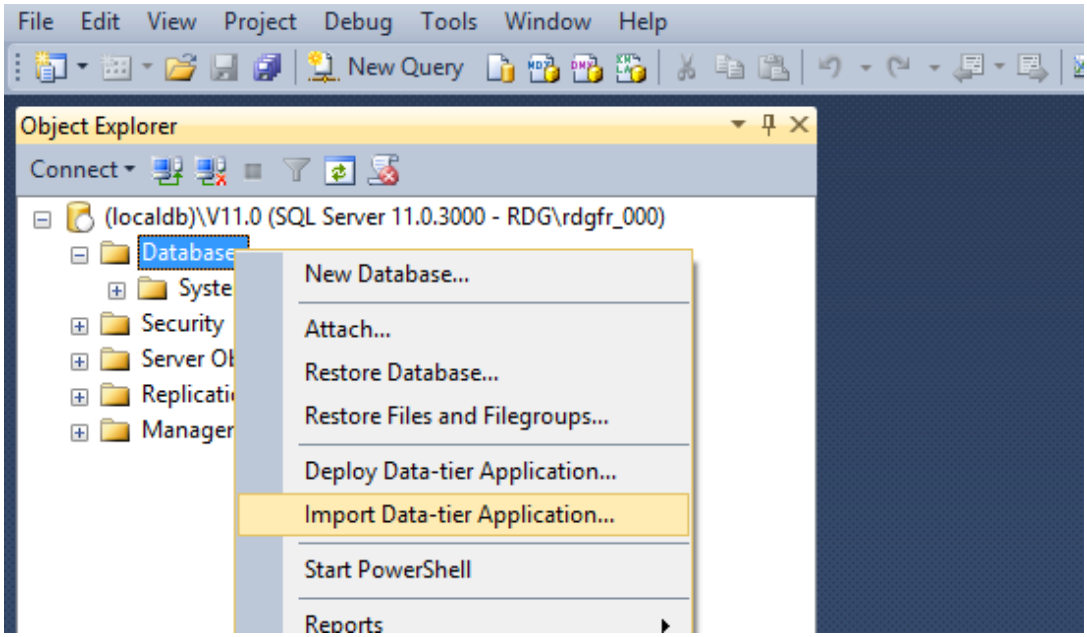

Figura 9-11: Paso 3

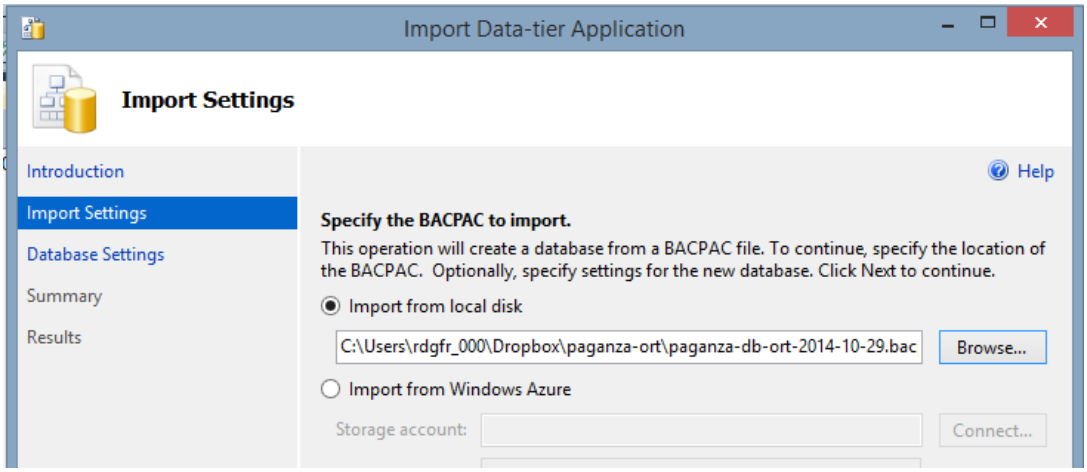

Figura 9-12: Paso 4

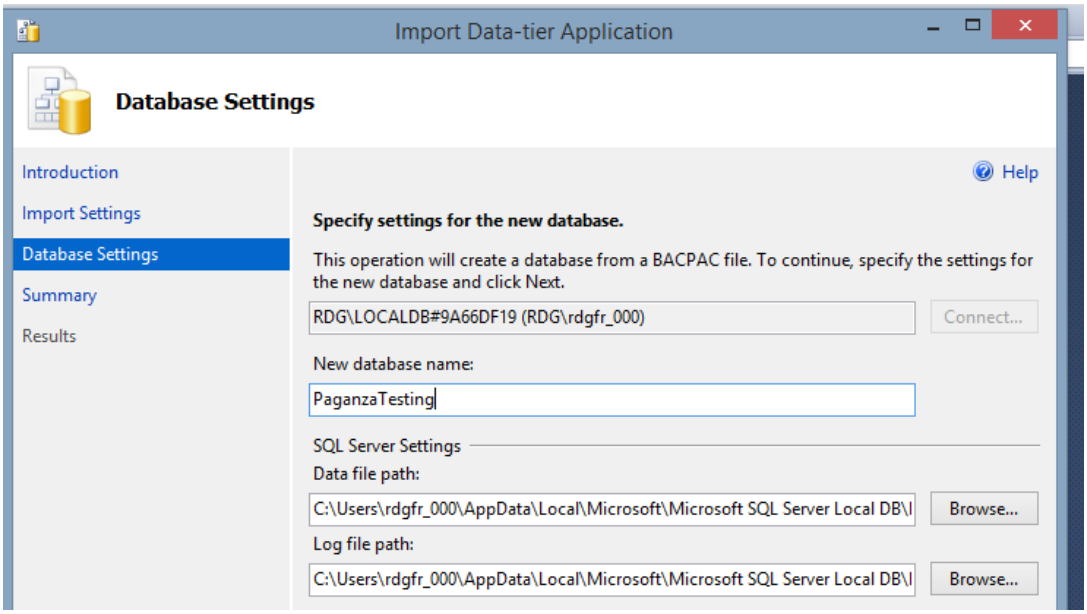

Figura 9-13: Paso 5

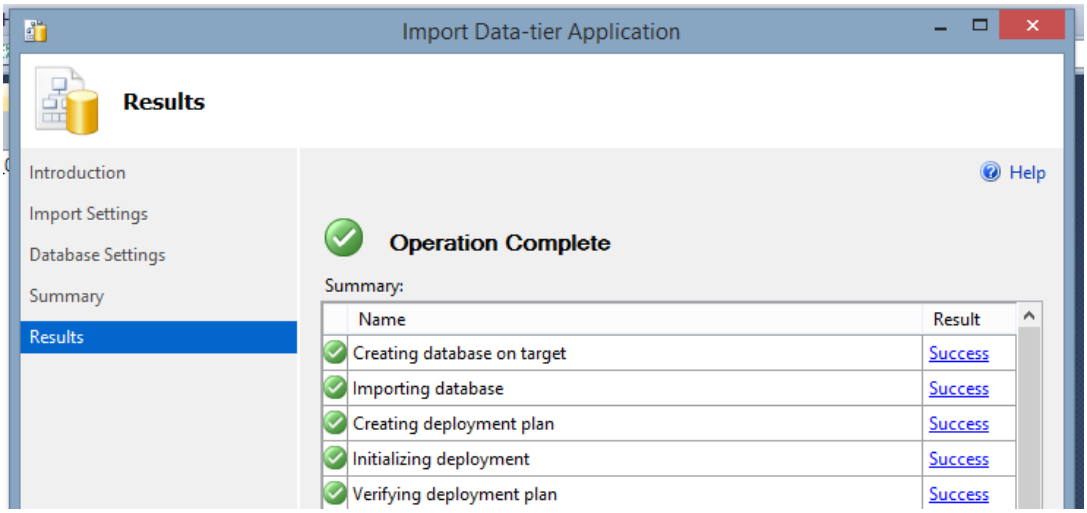

Figura 9-14: Paso 6

A continuación se describen los pasos necesarios para descargar el repositorio de código y los sub módulos referenciados.

1. Crear la carpeta "Paganza" en el root (C:\Paganza).

2. Dentro de la nueva carpeta "Paganza", usando gitbash ejecutar el siguiente comando dentro de /c/Paganza:

$ git clone -b gestion-gastos --recursive https://bitbucket.org/paganza/server

Recomendación: Incluir las claves ssh en el gitbash como lo recomienda bitbucket, y cambiar las rutas de los repos de https a git.

Luego se procede igual. Ejemplo:

$ git clone -b gestion-gastos --recursive git@bitbucket.org:paganza/server.git

Acá se puede usar el siguiente comando para bajar los sub módulos:

$ git submodule update --init

3. Esto debería crear la carpeta /server dentro de Paganza y debería bajar todo el código necesario incluyendo los sub módulos Paganza.Communication y Paganza.Parser.
4. Si se entra a la carpeta /server el branch actual debería ser 'gestion-gastos'.
5. Opcionalmente, se puede volver a la carpeta /Paganza y obtener los repos /communication y /parser independientemente:
/c/Paganza

$ git clone https://bitbucket.org/paganza/communication

$ git clone https://bitbucket.org/paganza/parser

Esto quedaría de la siguiente forma utilizando git:

$ git clone git@bitbucket.org:paganza/communication.git

$ git clone git@bitbucket.org:paganza/parser.git

Finalmente, se debería de tener una estructura de carpetas como se ve en la Figura 9-15 y en la Figura 9-16.

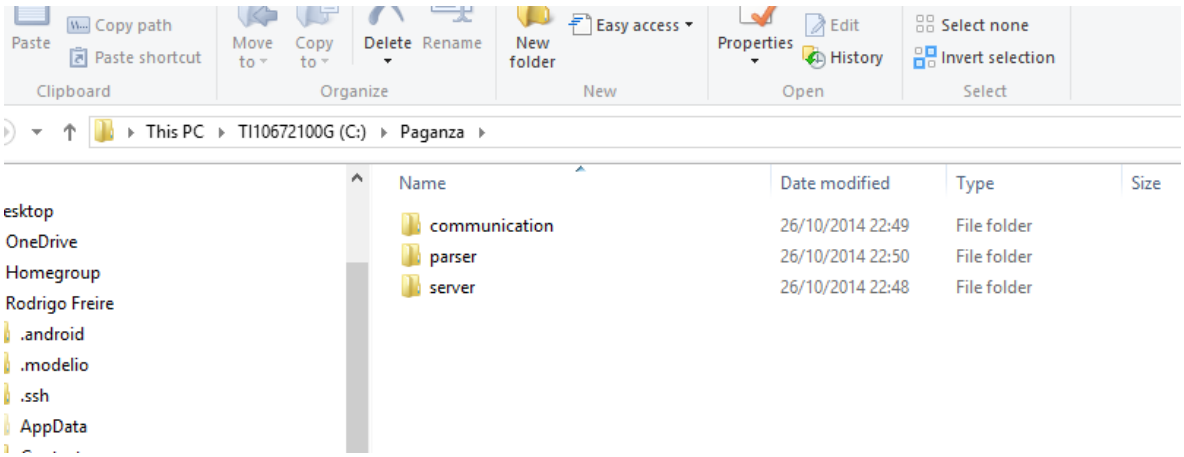

Figura 9-15: Estructura de carpetas

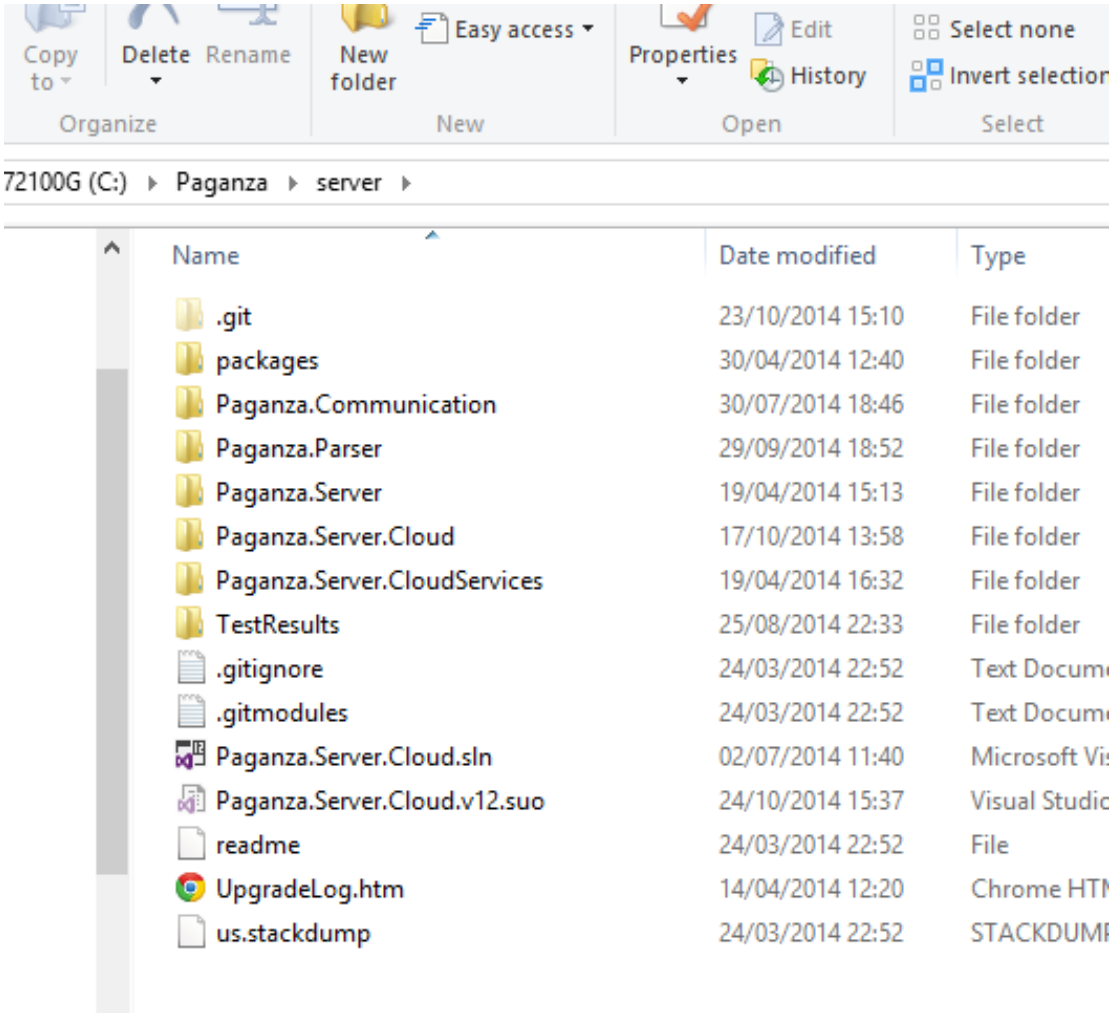

Figura 9-16: Contenido del repositorio

9.9. Ejemplo de planilla de testing

A continuación se muestra un ejemplo de una planilla de testing, donde se describe cómo se registran pruebas ejecutadas por un integrante del equipo. Aquí se desglosa qué módulo se probaba, qué requisitos eran necesarios que se cumplan y la descripción de la prueba a realizar. Luego de ser ejecutada por el integrante del equipo, se completaba el resto de la tabla. Aquí se introducía el estado resultante de la prueba, si ésta cumplía con el objetivo o no. Además en casos que ameriten, se agregan comentarios, y por último se completaba el nombre del conductor de la prueba. Esto se refleja en la siguiente tabla.

Módulo	Requisito	Descripción	Estado	Comentarios	Tester
Reporte de gastos	Usuarios 61,68,70	Ninguna factura paga	OK	-	Adrián Claverí
Reporte de gastos	Usuario 78458	Servicio con nombre más largo	NOK	(1)	Adrián Claverí
Reporte de gastos	Usuario con más de 1 servicio	Probar el filtrado en la gráfica por servicio	OK	-	Adrián Claverí
Reporte de gastos	Usuario con más de 1 servicio	Probar filtrado por fecha	OK	-	Adrián Claverí
Reporte de gastos	Usuario con más de 5 servicios	Verificar el paginado de servicios y la gráfica	OK	-	Adrián Claverí

Tabla 9-2: Extracto de planilla de testing

1. Al entrar en el reporte de gastos con el usuario 78458, se puede observar que en el listado de servicios, dentro de la tabla inferior, el nombre del mismo excede el largo estipulado en el contenedor.

Cómo se puede observar, en esta planilla de ejemplo, existieron pruebas que fueron satisfactorias, pero otra no. En el caso que una prueba no cumpliera con su objetivo, ésta era reportada al equipo y era agregada como bug para ser resuelta en un posterior sprint.

Se intenta que el comentario escrito sea lo más descriptivo posible, mostrando los pasos para reproducirlo, así en el futuro cuando vaya a ser arreglado, se puede identificar claramente el defecto.

9.10. Ejemplo de sesión de beta testing

En esta planilla se encuentran los comentarios y notas relevadas en una sesión de beta *testing* con un usuario. Los comentarios están categorizados por funcionalidad.

	Área	Bug	*Enhancement*	*Feedback*
Reporte de Gastos	Gráfica		Redondear gráfico, que no llegue al tope. Forma genérica de ver	Muy bueno como se muestra todo, queda claro. Pensé que me abría otra página pero me lo muestra ahí y está bueno.
	Filtro			Hizo *click* en el botón de calendar y no le abrió. *click* en el mes y no cierra el filtro. El botón filtrar no se sabe que es, el título no indica que es lo que se filtra, está como escondido. "El término está bien pero la gente no tiene que saber que es lo que filtra ahí"
	Gastos del periodo		No sabía que al clickear en la gráfica arriba, carga en los gastos del periodo. Indicar esto de alguna manera.	OK. No se dio cuenta que al hacer click te resalta en la torta.
	Torta	Se corta el resaltado contra el panel de gastos		Está bueno ver el valor en el hover.
Billetera	Saldos			Entiende que es.
	Reporte		"Me llegó el recibo de BSE que ya habíamos pago. Chequeamos en Paganza para saber que la había pagado"	Entiendo que es pago porque hay un menos, que es un débito porque conozco la palabra.
	Botón de más (Expandir)			A la primera no lo vi.

Tabla 9-3: Ejemplo de planilla de *beta testing*

	Área	Bug	*Enhancement*	*Feedback*
Expenses	Ingresar un expense	Agregamos un archivo adjunto, da error dice: Datos incorrectos en rojo.	El monto, al hacer click dice 0, debería borrarse el número.	Al ingresar uno nuevo, entré a editar, para volver atrás toco salir. Estaría bueno que indique que es notas (al leerlo entendió)
	Importar gasto		Mostrar ayuda con opción de mostrar sólo una vez.	Explicar, no se entiende qué hace. Importar gasto a dónde? de dónde?. Poner explicación arriba. Importar piensa que es importar a la computadora desde Paganza.
	Editar una expense			Se pueden editar todos los campos?
Etiquetas	Administración de etiquetas	No puedo editar el color de una etiqueta		
	Reporte			Es confuso el no ver datos de entrada y que aparezcan elementos mientras selecciona otras cosas
	Etiquetas un expense			Está buena la forma de ingresar expenses.

Tabla 9-4: Continuación de planilla de *beta testing*

	Comentarios generales
Reporte de Gastos	Explicar, no se entiende qué hace. Importar gasto a donde? de donde?. Poner explicación arriba. Importar piensa que es importar a la computadora desde Paganza.
Billetera	Queda bastante claro el concepto, aunque se podría explicar un poco mejor (colores) para la gente que no esté acostumbrada
Expenses	Está bueno el concepto de ingresar gastos por fuera de Paganza. Quedan detalles por mejorar.
Etiquetas	No queda claro para qué quiero etiquetar gastos y visualmente no se ven muy bien.

Tabla 9-5: Comentarios general sobre lo relevado en el *beta testing*

9.11. Ejemplo de aplicación de heurísticas de Nielsen

Heurísticas	Análisis de Gastos	Billetera	Listado de Gastos
Visibilidad del estado del sistema	Cuando se aplica el filtro, no se muestra ningún cargando. Estaría bueno que el botón de filtrar se cambie por un gif.	--	--
Control y libertad del usuario	Soportar Deshacer--> se podría tener un botón que diga "Aplicar filtro anterior". En el listado de los servicios, se debería tener un botón que diga "deseleccionar todos"	--	--
Consistencia y estándares	Fechas: mostrar en el formato que lo hace Paganza. El color de los *datepicker* debería ser el mismo verde de los botones. al hacer *click* en la barra acumulativa, debería hacer algo? La barra del promedio debería indicar cuanto es el promedio. El *dropdown* de servicios, en la misma sección que las fechas confunde, porque al seleccionar varios servicios, luego seleccionar fechas y hacer *click* en aplicar, para que después borre los servicios no es coherente	Poner el formato de fechas de Paganza. agregar colores que permitan mostrar de una mejor manera los pagos, débitos y devoluciones de IVA. si ya se sabe que se está mirando en pesos, porque aparece la moneda? Sacar el *hover* con la manito sobre las filas que no tienen ninguna acción	Poner el formato de fechas de Paganza

Tabla 9-6: Aplicación de heurísticas de Nielsen

Heurísticas	Análisis de Gastos	Billetera	Listado de Gastos
Prevención de errores	Se podría agregar al *Viewbag* el primer mes y año en el cual se registro un gasto, para evitar que el filtro de fecha pueda ir hasta el infinito para atrás. De cualquier manera, se debería poder seleccionar una fecha desde el 2010 en adelante o similar	--	--
Reconocimiento antes que recuerdo	Se podría poner un *tooltip* cuando se carga por primera vez la página, indicando donde esta el filtrar y que pasa cuando se hace *click*	--	--
Ayuda y documentación	Se podría poner algunas animaciones en *css*, con algunos *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa	Se podría poner algunas animaciones en *css*, con algunos *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa	Se podría poner algunas animaciones en *css*, con algunos *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa

Tabla 9-7: Continuación de aplicación de heurísticas de Nielsen

Heurísticas	Editar/Crear un Gasto	Agregar etiqueta a un gasto	Agregar una etiqueta a una factura	Administrar Etiquetas
Visibilidad del estado del sistema	--	Falta cargando para las modificaciones de las etiquetas, ya sea agregar una o borrar	Falta cargando para las modificaciones de las etiquetas, ya sea agregar una o borrar	Falta cargando para las modificaciones de las etiquetas, ya sea agregar una o borrar
Control y libertad del usuario	--	Faltaría un *breadcrumb* para volver fácilmente al listado de gastos	Faltaría un *breadcrumb* para volver fácilmente al listado de gastos	--
Consistencia y estándares	Botón de ingresar gasto debería ser verde. Color del *datepicker* debería ser verde	Agregar los colores a las etiquetas. El texto "Etiquetas" puede quedar mejor al lado del input, y no arriba. "Editar Gasto" queda bastante repetitivo	Agregar los colores a las etiquetas. El texto "Etiquetas" puede quedar mejor al lado del input, y no arriba.	Poner en "apagado" a las etiquetas que no se están editando cuando se esta editando una etiqueta. Los links de editar, guardar y cancelar podrían ser o iconos o botones verdes
Prevención de errores	Faltan validaciones de *javascript* en *frontend*	--	--	--
Flexibilidad y eficiencia de uso	--	Falta filtrar los gastos. según etiqueta, según servicio, según fecha, según monto	Falta filtrar las facturas según etiqueta	Podría existir un "ordenar etiquetas por nombre"
Ayuda y documentación	Se podría poner algunas animaciones en *css*, con *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa	Se podría poner algunas animaciones en *css*, con *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa	Se podría poner algunas animaciones en *css*, con *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa	Se podría poner algunas animaciones en *css*, con *tooltips* que expliquen que hace cada sección, que muestra cada parte y como se usa

Tabla 9-8: Continuación de aplicación de heurísticas de Nielsen

9.12. Guía de ingreso de bugs

Con el fin de agrupar los bugs encontrados por los distintos actores del proyecto, se decidió utilizar TFS. Una herramienta gratuita de Microsoft que permite la gestión online de incidentes.

Durante el proyecto se encontraron 3 posibles fuentes de incidentes:

- El equipo de desarrollo
- Paganza
- Los usuarios

Tanto el equipo como Paganza, tienen acceso directo al repositorio de incidentes, pero los usuarios no. Para esto se brindó, mediante UserVoice, la posibilidad de dar *feedback*, reportando un bug y luego el equipo se encarga de subirlo al sitio.

Con el fin de mantener trazabilidad en los reportes y para obtener métricas, se agregó una categorización para diferenciar de qué fuente proviene el incidente.

Template de incidente

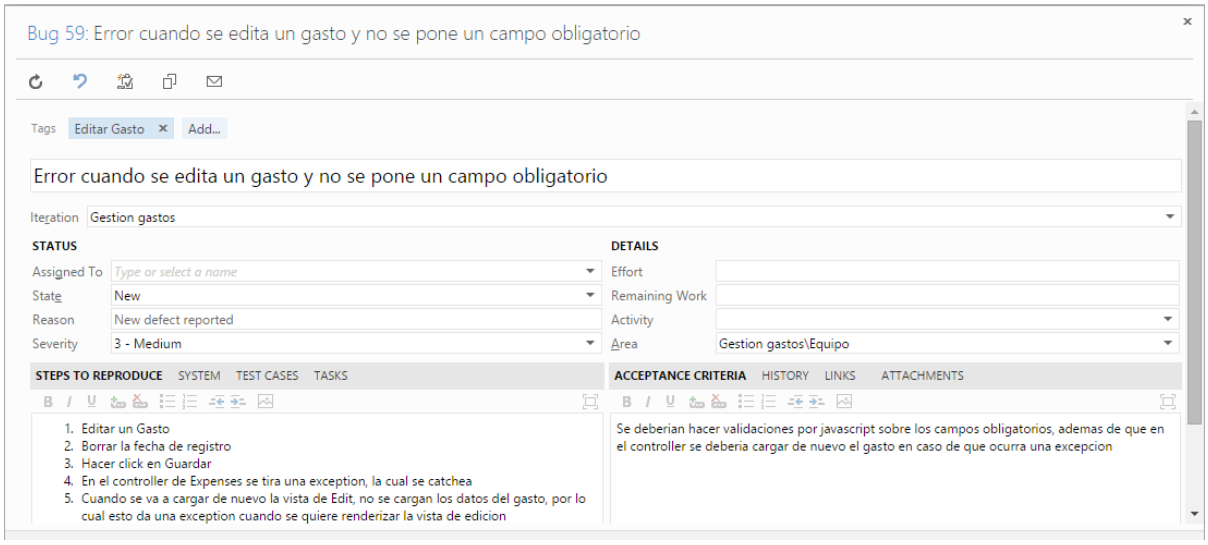

Figura 9-17: Template de incidentes

Los campos necesarios para un correcto reporte de incidentes son los siguientes:

- *Tags*: Para su mejor visualización y con el fin de obtener métricas, se agregan etiquetas según la funcionalidad que se ve afectada por el bug. Por ejemplo: Gastos, Facturas, Análisis

- Título: Campo descriptivo del error.

- *State*: Se debe dejar el valor que viene por defecto (*New*). Este campo ayuda a la trazabilidad del bug a lo largo del proceso de corrección de incidentes. El estado cambia al comenzar a trabajarse en el mismo (*Committed*), terminado o removido en caso que no aplique.

- *Severity*: En caso de ser un incidente importante, que influya en el funcionamiento normal de la aplicación debe ser reportado con un alto nivel de severidad.

- Área: Actor que reportó el error.

- *Steps to reproduce*: Una serie de pasos a seguir para de un funcionamiento normal, reproducir el bug reportado.

- *Acceptance criteria*: Campo opcional, en el cual se especifica el correcto funcionamiento que debería tener la aplicación.

- *Attachments*: Lugar donde se pueden adjuntar imágenes del bug encontrado o cualquier archivo adjunto que aporte al incidente.

Para mejor visualización de los incidentes en el TFS, se crearon 3 vistas según el área de reporte y ordenados por severidad.

Criterio para crear un bug

Se entiende como bug o incidente un error en el programa que desencadena en un resultado no deseado. La herramienta elegida permite la distinción entre un *bug* y una *feature* (oportunidad de mejora). Por lo que se manejan como *bugs* simplemente errores en el curso normal de una funcionalidad, mientras que nuevas funcionalidades o cambios en el comportamiento existente se deben reportar de manera distinta.

En caso de que un reporte de *bug* determine que hay que realizar un *refactor* grande o involucre otras funcionalidades, el mismo será estimado e incluido en el *backlog* de mejoras.

Asignación de *bugs*

A medida que transcurre el *sprint* y se reportan los *bugs*, los mismos se van priorizando según la importancia y los miembros del equipo los toman según su disponibilidad. En caso de un bug muy importante, se prioriza por sobre otra funcionalidad que no esté comenzada.

Al momento de comenzar a trabajar en un bug, el estado del mismo pasa a "*Committed*", y el encargado de resolverlo se marca en "*Assigned to*".

Luego de corregido el incidente, se debe marcar el *ticket* como terminado y en la pestaña "*History*" se deben indicar los cambios realizados, subiendo de ser necesario evidencia del arreglo del *bug*.

Comunicación de corrección de *bug*

Al final del *sprint*, además de comunicar las *features* realizadas a Paganza, se envía una lista de *bugs* resueltos y que se comenzaron a trabajar. Además de en esta instancia revisar la lista en caso de tener repetidos o incidentes que no apliquen más (arreglados en otro momento).

9.13. Resultados de métricas

9.13.1. Resumen de métricas registradas

	SP2	SP4	SP6	SP8	SP10	SP12	SP14	SP16	SP18
Tiempo para ingresar un gasto	--	--	--						
Web (segundos)	--	--	--	6.08	5.95	6.13	5.34	4.975	5.122
Móvil (segundos)	--	--	--	--	--	20.13	18.54	16.616	15.562
Tiempo que lleva ver gastos de un mes en particular	--	--	--						
Web (segundos)	--	--	--	--	--	1.345	1.45	1.256	1.78
Móvil (segundos)	--	--	--	--	--	1.86	1.45	1.352	1.67
Tiempo que lleva en agregar una etiqueta	--	--	--	1.53	1.45	1.67	0.856	0.565	0.654
Tiempo que lleva ver el listado de gastos	--	--	--						
Web (segundos)	--	--	--	3.456	3.33	3.78	2.87	2.52	2.474

Móvil (segundos)	--	--	--	--	--	3.01	2.93	2.52	2.5
Tiempo de reconocimiento de texto (segundos)	--	--	--	45.38	42.32	40.3	38.53	33.844	32.05
Cantidad de *clicks* se requieren para agregar una etiqueta	--	--	--	5	5	3	3	3	3
Cantidad de *clicks* para ver los gastos de un mes en particular	--	--	--	--	--	--			--
Web	--	--	--	1	1	1	1	1	1
Móvil	--	--	--	--	--	3	3	3	3
Cantidad de *clicks* para crear un gasto	--	--	--						
Web	--	--	--	3 Click 3 Campos	3 Clicks 4 Campos	2 Clicks 4 Campos	2 Clicks 2 Campos	2 Clicks 2 Campos	2 Click 2 Campos
Móvil	--	--	---	--	--	5 Clicks 3 Campos	5 Clicks 2 Campos	5 Clicks 2 Campos	5 Clicks 2 Campos

Cantidad de *clicks* para ingresar 5 gastos de manera múltiple	---	--	---	---	---	7	5	5	5
Tiempo de respuesta para el listado de facturas sin etiquetas (segundos)	--	--	--	2.898	2.898	2.898	2.898	2.898	2.898
Tiempo de respuesta para el listado de facturas con etiquetas (segundos)	--	--	--	3.127	3.105	3.023	2.74	2.402	2.513
Tiempo de respuesta para el detalle de una factura sin etiquetas (segundos)	--	--	--	0.568	0.568	0.568	0.568	0.568	0.568
Tiempo de respuesta para el detalle de una factura con etiquetas	--	--	--	1.078	0.987	1.02	0.74	0.701	0.641

(segundos)									
Cantidad de cambios sobre código existente	---	---	1	1	1	1	1	---	---
Solicitudes de cambio por funcionalidad hechas por el cliente	---	---	2	3	---	---	3	3	3
Solicitudes de cambio sobre requerimientos planteados por los usuarios	---	---	4	3	---	---	3	3	2
Cantidad de errores detectados por revisiones	7	12	13	7	6	4	8	6	6
Cantidad de errores detectados por usuarios	---	---	---	---	4	1	2	2	2
Cantidad de errores detectados por Paganza	3	9	13	3	---	2	2	3	---

Figura 9-18: Resumen de métricas registradas

9.13.2. Eficiencia de otras Herramientas

Web	Expensify	Xpenditure	Buxfer
Crear un gasto			
Campos a completar	2	6	3
Cantidad de *clicks*	2	4	2
Importar múltiples gastos (5)	15	16	14
Listado de gastos	1	2	1
Filtrado de gastos (filtrar por fecha)	2	5	6
Agregar etiqueta a gasto	3	-----	4

Figura 9-19: Comparación de usabilidad en cantidad de *clicks*

Móvil	Expensify	Xpenditure	Buxfer
Crear un gasto manualmente			
Campos a completar	2	3	2
Cantidad de *clicks*	4	5	4
Crear un gasto reconocimiento			
Campos a completar	1	0	---
Cantidad de *c*licks	5	6	---
Listado de gastos (incluido traer mas gastos)	1	0	1
Filtrado de gastos	---	---	---

Tabla 9-9: Comparación en cantidad de *clicks* de la plataforma móvil

Web	Expensify	Xpenditure	Buxfer
Crear un Gasto (segundos)	1.69	7.727	2.827
Listado de Gastos (segundos)	8.033	6.165	4.68
Filtrado de Gastos (segundos)	1.24	1.35	1.153
Agregar Etiqueta a Gasto (segundos)	0.615	-----	2.09
Reconocimiento de Texto (minutos)	3:18.37	6:04.32	------

Tabla 9-10: Comparación en medida de tiempo de las funcionalidades web

9.13.3. Registro de Índice de Mantenibilidad

	SP2	SP4	SP6	SP8	SP10	SP12	SP14	SP16	SP18
Índice de Mantenibilidad									
Web									
Paganza.Companies							85	84	84
Paganza.Expenses				84	84	87	88	88	88
Paganza.ImportExpenses						85	85	85	85
Paganza.OCR									65
Paganza.QueryExpenses							69	69	69
Paganza.Balance		83	83	83	83	89	89	89	89
Paganza.Reports	83	83	83	80	80	88	90	91	91
Paganza.TagsReport					86	88	88	88	88
Paganza.Tags			87	84	84	85	85	85	85
Mobile									
Commons.Database					80	80	82	84	84
Commons.Utilities					82	90	90	90	90
Controllers					69	77	77	77	77
IControllers						100	100	100	100
Models					91	91	91	91	91
Access					82	90	92	93	93

Tabla 9-11: Índice de mantenibilidad