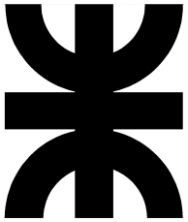


Universidad Tecnológica Nacional

Facultad Regional Santa Fe

Proyecto Final de Carrera

Plataforma web para la gestión de estudio contable - ***DiaryBooster***

Integrantes:

- Chamorro, Lucas Gabriel.

Director:

- Ing. Lisandro Vrancken.

Universidad Tecnológica Nacional – Santa Fe

2021

Contenido

1. Introducción

El proyecto final de carrera, abordado en este informe, consiste en el desarrollo y la implementación de un sistema web ideado para el registro y soporte de las tareas administrativas realizadas en un estudio contable. El alcance de este trabajo no son tareas contables en sí mismas, como pueden ser la liquidación de impuestos, el asiento de registro contables, entre otras; sino más bien proveer una herramienta de soporte robusta y confiable para las tareas de gestión derivadas de las mismas.

El producto obtenido como resultado de este proyecto fue denominado DiaryBooster. Se trata de una herramienta diseñada para el estudio contable "Baldomir & Gaudiano", que tiene como objetivo maximizar y organizar la realización de tareas diarias. Funciona como una agenda completa, en donde el cliente podrá realizar el seguimiento estricto de todas las tareas y actividades que se llevan a cabo diariamente. Además, tiene la posibilidad de contemplar otro tipo de aspectos como por ejemplo finanzas, reuniones, carga de clientes, empleados, manejo de sueldos, etcétera.

En este documento explicaremos todos los procesos realizados desde la captura de requerimientos hasta la puesta en producción y mantenimiento del sistema, metodologías utilizadas, herramientas, problemas encontrados en el desarrollo, riesgos asumidos, pruebas realizadas, etcétera.

2. Problemática

El origen de este proyecto se debe fundamentalmente a los problemas detectados por el dueño del estudio contable en el funcionamiento cotidiano del mismo. A grandes rasgos, estos inconvenientes se podrían resumir en la pérdida de eficiencia del tiempo de trabajo, debido a la gran carga generada por las actividades administrativas, irregularidades y pérdidas ocasionales de información por la falta de una única fuente confiable y actualizada de datos, y pérdidas económicas derivadas de un difícil seguimiento de las cobranzas a clientes.

Inicialmente, el cliente nos indicó que todo su accionar se reflejaba en múltiples planillas de Excel y, con el consecuente crecimiento de la organización, la complejidad diaria incrementó, provocando que el uso de las mismas se torne insostenible.

La solución que se implementó para el cliente realiza un abordaje integral de las necesidades que surgen de la problemática planteada. Al proveer un único sistema en el cual encontrar toda la información relativa a la gestión del estudio, resolvemos las dificultades asociadas al acceso a la información, la integridad y actualidad de la misma, y la falta de sincronía.

Uno de los aspectos relevantes del sistema es permitir dividir el trabajo realizado por el estudio en tareas y actividades que representen porciones más reducidas. Esto permite llevar adelante un mejor seguimiento de las labores requeridas por cada cliente, saber el nivel de avance de cada tarea, y al estandarizar la forma de trabajar, elevar los niveles de productividad.

Por otro lado, la automatización de vencimientos basada en los calendarios de AFIP y las bandejas de entrada de cada usuario, ayudan a reducir la carga de trabajo. De tal manera, los empleados pueden gestionar su propio trabajo.

Por otro lado, el proyecto también aborda la gestión de los clientes del estudio, abarcando no solo el acceso a los datos actualizados de los mismos, sino también el seguimiento de sus requerimientos y de sus estados de cuenta.

El trabajo del proyecto implicó por lo tanto comprender el dominio involucrado en el universo del estudio contable, modelarlo, e implementar un sistema que resolviera las necesidades del cliente, y además gestionar los requerimientos y las expectativas del mismo, acompañándolo y ayudándolo a identificar qué funcionalidades eran respuesta a necesidades reales, y cuáles no eran parte necesaria del desarrollo. Finalmente, alcanzamos una solución que logró dar respuesta a las problemáticas planteadas en esta sección.

3. Objetivos

El objetivo general del proyecto fue desarrollar e implementar un sistema web que permita gestionar y dar seguimiento a las tareas diarias de un estudio contable.

Detallamos los objetivos particulares que esperamos brindar con la utilización de DiaryBooster:

- Obtener una trazabilidad completa del trabajo realizado por la organización.
- Acceder a una agenda automatizada que detalle:
    - El trabajo que se debe realizar y su nivel de urgencia.
    - La visualización de todas las tareas asignadas (vencidas, del día, por vencer).
- Automatizar tareas administrativas que actualmente consumen tiempo y esfuerzo.
- Lograr un aumento en los ingresos a partir del seguimiento detallado de las tareas ejecutadas, y los pagos percibidos por las mismas.
- Alcanzar un mayor valor entregado al cliente.
- Facilitar el seguimiento de clientes, una tarea que muchas veces se torna complicada por la falta de un soporte adecuado para la misma.
- Nuclear en un mismo sitio, información que hoy se encuentra dispersa en varios medios, archivos e incluso equipos que dificultan su acceso, actualización e integridad.
- Alcanzar un resguardo de información periódico y automatizado.
- Lograr un acceso remoto a la información y organización del estudio.
- Optimizar el tiempo en base a la organización laboral.

4. Metodología de desarrollo

Llevar a cabo cualquier proyecto de esta magnitud requiere un arduo trabajo de investigación por parte del equipo. Es de gran importancia contar con una *metodología de desarrollo* que nos encamine hacia resultados fiables. Aquí explicaremos en base a nuestra experiencia qué metodologías utilizamos y cómo lo llevamos a cabo.

Para el desarrollo de DiaryBooster, la metodología elegida fue una adaptación del Proceso Unificado de Desarrollo (RUP), en la que nos basamos principalmente en sus fortalezas, pero a la cual intentamos imprimir un enfoque ligero. Evitamos apegarnos estrictamente a la especificación formal del proceso y sus entregables, incluyendo algunas adaptaciones y prácticas características de las metodologías ágiles.

El primer punto de consideración respecto de las propiedades de RUP, corresponde a un tratamiento de un proceso iterativo e incremental. Desde la primera reunión con el cliente se estableció que el sistema a desarrollar contaría con un nivel de complejidad considerables, y de un considerable tamaño para el equipo que lo gestionaría. En consecuencia, poder generar una solución incremental, que fuera agregando valor y funcionalidades por etapas, se convertiría en una ganancia tanto para el cliente como para el equipo de trabajo, dado que se podría retornar valor de forma temprana, y evitar arduas revisiones.

El segundo factor relevante para la decisión fue que el proceso unificado se encuentra dirigido por casos de uso. La cantidad de funcionalidades a desarrollar era importante y los casos de uso son un medio adecuado para dejarlas asentadas, reflejar sus interacciones, etcétera. También, permiten detallar adecuadamente qué se espera de cada caso de uso, especialmente para aquellos que son más sensibles y tienen una mayor complejidad.

En relación a la arquitectura del sistema, existía una idea previa respecto de las tecnologías a utilizar (las cuales brindaban un marco de trabajo definido, por más que no fuera obligatorio) y de los atributos de calidad que se esperaban alcanzar, por lo cual no lo consideramos una ventaja importante. A su vez, el enfoque en los riesgos críticos del proyecto nos parecía importante, pero no encontramos inicialmente que se tratara de un factor decisivo para el caso. Esta apreciación inicial ocasionó múltiples contratiempos a medida que avanzó el proyecto, y ciertamente podríamos haber aprovechado en mayor beneficio esta característica propia del método de desarrollo.

Por último, el otro elemento que valoramos a la hora de seleccionar RUP fue la documentación propuesta por el propio modelo. Si bien no utilizamos todos los documentos que se detallan en la especificación del mismo, incluimos aquellos que consideramos más importantes y que nos permitieron darle un enfoque organizado al proyecto. Además, UML (Lenguaje Unificado de Modelado) da un marco de referencia estándar para escribir e interpretar la documentación, de forma que no importa el momento en que accedamos a la misma; igualmente obtendremos una idea clara de cómo se estructura y comporta nuestro sistema.

Este último punto fue especialmente importante debido a que, si bien el proyecto surgió como un desarrollo para un cliente particular, determinamos que era conveniente

construirlo de forma tal que fuera modificable, escalable y extensible, para poder convertirse en un producto de software estándar y que pudiera ser vendido nuevamente en el mercado. Para lograr este objetivo no solo es relevante la calidad del código y de la arquitectura del sistema, sino también el conocimiento que se logre plasmar en los documentos de forma que se vuelva más sencillo el diseñar y construir nuevas soluciones o modificar las existentes.

4.1 Consideración de prácticas de metodologías ágiles

A pesar de todas las ventajas que nos guiaron a la selección de esta metodología de desarrollo, también reconocimos ciertas características que podían derivar en contratiempos importantes para el proyecto. Es por ello que decidimos adaptar RUP, e incorporar determinadas prácticas de las denominadas metodologías ágiles, en pos de disminuir estos riesgos potenciales, e incluso incorporar nuevas ventajas al proceso de desarrollo a utilizar.

Uno de los puntos que más inquietud nos generaba era la relación con el cliente. Si bien la utilización de casos de uso establece especificaciones claras respecto de las funcionalidades del sistema, y de cuáles deberían ser los criterios de aceptación correspondientes, las iteraciones de RUP involucran bastante al cliente en las etapas más tempranas de las mismas, para luego dejar paso a las tareas del equipo de trabajo. Esto podría generar diferencias entre las necesidades reales del cliente y las que fueran relavadas inicialmente por múltiples factores. Hacer iteraciones cortas en el tiempo y que incluyan pocas funcionalidades es una opción para enfrentar este tipo de problemas, pero en este caso particular, el core de funcionalidades básicas era bastante grande y complejo, por lo cual no era posible dividir tan fácilmente los casos de uso en iteraciones reducidas, y al mismo tiempo brindar valor relevante para el cliente. Para solucionar esto, optamos por incorporar al cliente tanto como fuera posible en el equipo de trabajo. Resultaría imposible tener al cliente in situ, como plantea Xtreme Programming, pero organizamos reuniones periódicas para mostrar avances conseguidos, compartir decisiones respecto a la implementación de requerimientos que pudieran afectar de alguna forma a otros casos de uso. También, planeábamos los pasos a seguir de acuerdo a las prioridades del negocio, pero sin dejar de prever cómo las decisiones más adecuadas hoy pueden afectar las necesidades del mañana.

Otra de las características generalmente reconocidas a las metodologías ágiles es utilizar la documentación justa y necesaria. Como resaltamos anteriormente, este era un punto importante para nosotros, pero tampoco era nuestra intención alcanzar el final del proyecto y encontrar que teníamos una documentación excesivamente detallada y precisa, y un software que no cumpliera con todos los requerimientos funcionales y no funcionales debido a que no le dedicamos la suficiente atención. Por esto decidimos tomar como base la documentación planteada por RUP y a partir de allí comenzar a seleccionar los documentos que consideramos más relevantes para la gestión, el desarrollo y el posterior mantenimiento. Al final de la selección nos encontramos con un conjunto de artefactos que constituían una cantidad razonable, intermedia, y adecuada a las necesidades del proyecto.

En relación a este punto, también buscamos tomar las decisiones en el momento adecuado. No renunciamos a una visión global que guiara el desarrollo del proceso desde el inicio, o que fuera capaz de levantar las alertas necesarias, pero sí decidimos que las definiciones se fueran detallando progresivamente, de acuerdo al grado de avance del proyecto y de las necesidades puntuales a resolver en cada momento.

Por último, otra característica importante que intentamos incorporar al proyecto fue utilizar buenas prácticas de codificación, empleando prácticas de código limpio, variables claras, estructuras simples (en lo posible) y reutilizables, y especialmente mantener consistencia en todas las instancias que tuvieran cierto grado de similitud. En este sentido la refactorización del código e incluso de la estructura del proyecto fue una constante a lo largo del mismo. A medida que avanzamos, fue mejorando nuestro conocimiento de las tecnologías utilizadas, y por consiguiente pudimos establecer patrones reutilizables en diferentes secciones del proyecto. En estos casos, tal como sugiere Xtreme Programming, aplicamos las refactorizaciones necesarias para mantener la consistencia del código, y de tal manera obtener un producto de calidad.

5. Determinación de la arquitectura

Uno de los aspectos fundamentales de cualquier proyecto es la base del mismo; es decir, se requiere realmente de un análisis previo riguroso para determinar de qué manera se construirá y cuál es el camino a seguir una vez en desarrollo. Aquí, abarcaremos la estrategia utilizada para la elaboración del "Documento de Arquitectura de Software", los modelos empleados y los resultados obtenidos en consecuencia.

El establecimiento de la arquitectura sobre la cual se construyó DiaryBooster fue uno de los puntos al cual le dimos mayor relevancia desde el inicio del proyecto. El mismo fue determinante, como se comentó en la sección anterior, hasta para la elección de la metodología de desarrollo.

5.1 Documento de Arquitectura de Software

Del Proceso Racional Unificado se desprende uno de los documentos clave para la implementación de un proyecto de software: el Documento de Arquitectura de Software. La confección del mismo contó con una versión inicial que se fue refinando posteriormente con los elementos más determinantes entre las versiones 1.0 y 2.0, y la inclusión de pequeñas modificaciones o actualizaciones a medida que se iban tomando decisiones con un mayor nivel de detalle, pero que no implicaban cambios estructurales o profundos en las determinaciones preestablecidas.

El alcance abarcado por el documento incluye la implementación del sistema en el estudio contable Gaudiano, de la ciudad de Santa Fe. De acuerdo con ello, los atributos de calidad esperados se establecieron a partir de necesidades detectadas por el equipo de desarrollo respecto de las buenas prácticas de ingeniería de software, las normativas legales vigentes y otras por requisito del cliente.

La representación de la arquitectura de DiaryBooster se realizó mediante vistas pertenecientes al modelo V&B (Views & Beyond). Para la construcción de los gráficos se prescindió de la notación formal de UML, dando lugar a la utilización de las representaciones gráficas informales más extendidas para cada tipo de vista. Si bien esta decisión parece ir en perjuicio de RUP, consideramos que estas vistas informales brindarían una facilidad de comprensión mayor, más rápida, flexible, y que no obligaría a quien deba trabajar y modificar el sistema (en caso de que no fuéramos los autores) a aprender las reglas de modelado UML para comprender las vistas.

El Documento de Arquitectura de Software forma parte de los anexos del Informe final del proyecto; por consiguiente, puede ser consultado en caso de requerir un mayor nivel de detalle. A continuación, expondremos los puntos más significativos del mismo.

5.2 Atributos de calidad

Considerando los requerimientos funcionales, y primordialmente los requerimientos no funcionales o restricciones del sistema, se estableció que los principales atributos de calidad esperados serían los siguientes:

- Modificabilidad
- Seguridad
- Usabilidad
- Accesibilidad

No cabe duda de que, así como los mencionados, existen muchos otros atributos de calidad de software que pueden ser valiosos tanto para el proyecto en estudio como para cualquier otro. Sin embargo, sabemos que los atributos de calidad están fuertemente interrelacionados entre sí, y que no interactúan impulsándose unos a otros de una forma directamente proporcional, sino que suelen operar en forma contraria, inversamente proporcional, donde si un atributo aumenta, una contraparte se verá disminuida. En consecuencia, debemos encontrar en el espacio de soluciones un punto de equilibrio o balance entre fuerzas, priorizando aquellos que consideramos brindan un mayor valor al sistema, sobre otros menos relevantes. Con estas consideraciones, fueron escogidos los cuatro atributos ya detallados.

A su vez, para identificar tácticas arquitectónicas que resulten apropiadas y útiles para la solución requerida, se desarrollaron dos escenarios para cada atributo de calidad detallado (excepto para accesibilidad, para el cual solo se desarrolló un escenario).

Todos ellos pertenecen a un único atributo de calidad buscado, y están compuestos por: una descripción general del escenario, una fuente, un estímulo, un artefacto, el entorno en que se producen, la respuesta al estímulo y la medida esperada de ella, las tácticas propuestas para solucionar el planteo del escenario, y las interrelaciones que plantean con otros atributos de calidad.

5.3 Tácticas Propuestas

El resultado de cada uno de los escenarios planteados (los cuales no serán detallados para evitar extendernos) derivó en una o más tácticas propuestas, aplicadas posteriormente tanto para la determinación de la arquitectura, como para la construcción del sistema. En conjunto, las soluciones propuestas fueron las siguientes:

- Transformamos las tipificaciones del sistema que puedan variar en el tiempo en clases del modelo y tablas de BD, y brindar acceso a los ABM correspondientes al usuario.
- Utilizamos un *framework* de desarrollo que permita la fácil instalación y acoplamiento de nuevas librerías sin afectar las existentes.
- Utilizamos el patrón MVC, de forma tal que se separen correctamente la lógica del sistema, las vistas de usuario y el modelo. Así, en caso de requerir una nueva funcionalidad o modificar una existente, sólo se requerirá identificar las entidades del

modelo involucradas, y desarrollar la nueva funcionalidad, aislando el impacto que podría tener en otras funciones del sistema.
- Utilizamos el método de autenticación de usuarios.
- Dividimos las funcionalidades del sistema en roles de usuario. Los roles responden a una jerarquía de roles.
- Buscamos codificar los registros sensibles antes de introducirlos a la base de datos, y decodificarlos cuando se presenten al usuario por medio del sistema.
- Diseñamos las pantallas de la vista aplicando prácticas de diseño web *responsive*, que adapta el contenido de acuerdo al tamaño de la pantalla que lo está visualizando.
- Utilizamos el patrón MVC para adaptar las vistas sin necesidad de influir en los demás componentes.
- Utilizamos plantillas estandarizadas para la presentación de las interfaces de usuario, que podrán ser reutilizadas en todas las pantallas del sistema (TWIG).
- Utilizamos *frameworks* de diseño de interfaces web (*Bootstrap*, por ejemplo).
- Construimos un sistema web que será accesible a través de internet.

5.4 Especificación de la arquitectura mediante vistas

La aplicación de las tácticas arriba enumeradas, en conjunto con decisiones relacionadas a la tecnología escogida para el desarrollo, la implementación del proyecto, y otras consideraciones del equipo de trabajo que no surgieran necesariamente de los escenarios planteados, condujeron a un diseño arquitectónico que se representa mediante un conjunto de vistas. Las mismas se seleccionaron de acuerdo al modelo Views & Beyond, por lo que las escogidas fueron: Vista módulo, Vista componente conector, y Vistas de instalación y de despliegue para esquematizar la vista de asignación. Cada una de ellas representa diferentes tipos de componentes, desde elementos de software, sistemas, subsistemas, librerías, hardware, conexiones, relaciones, etcétera. También se presentan las relaciones entre estos elementos, poniendo de manifiesto una noción básica acerca del comportamiento esperado del sistema.

Vista Módulo

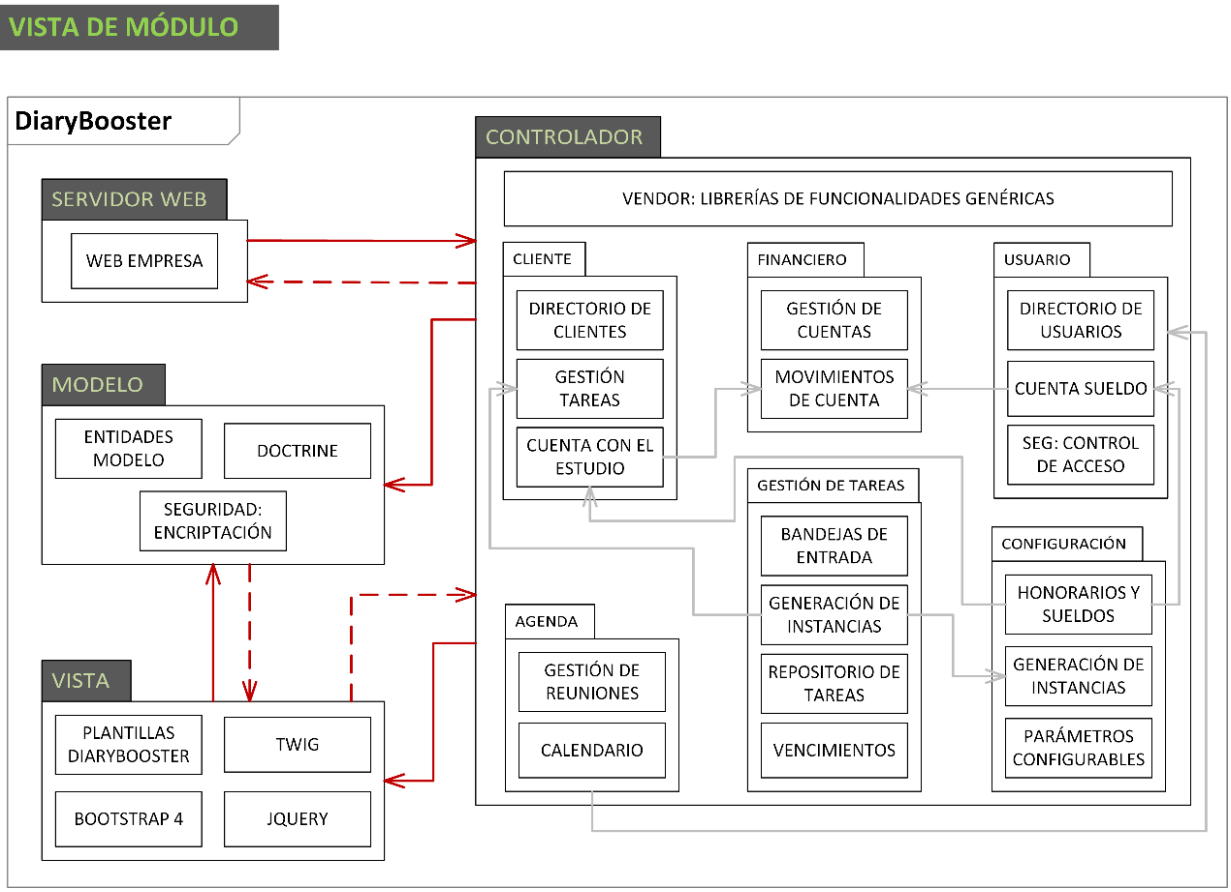


IMAGEN 001. Diagrama de arquitectura – Vista de módulo del sistema. Representación de los distintos módulos que componen el sistema, y las responsabilidades y funcionalidades de cada uno.

En la vista módulo podemos apreciar claramente la aplicación del patrón de diseño Modelo – Vista – Controlador, una de las propuestas más reiteradas para la resolución de los problemas planteados por los diferentes escenarios. También se puede ver, en uno de los submódulos de configuración, la transformación de las diferentes tipificaciones del sistema en parámetros configurables. Este módulo es el responsable de todas estas variables configurables y su funcionamiento.

Por último, también podemos visualizar la implementación en un submódulo de la gestión de usuarios el control de acceso requerido para aumentar la seguridad del sistema; y en la vista, la implementación de distintas librerías, con sus propias funcionalidades y objetivos, mencionadas en las tácticas propuestas (como ser el caso de TWIG y de Bootstrap).

Vista componente & conector

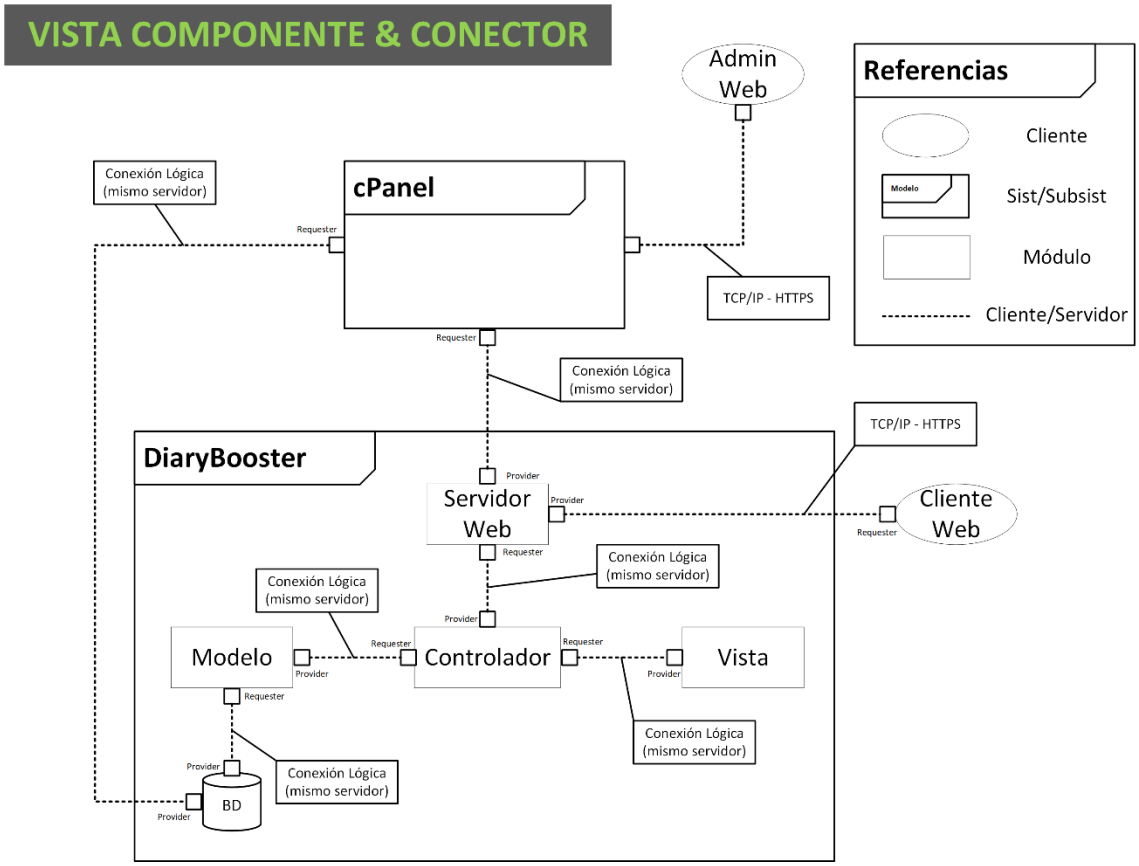


IMAGEN 002. Diagrama de arquitectura – Vista de componente Conector. Representación de los comportamientos y las interacciones entre los componentes del sistema.

La vista componente conector nos permite visualizar cómo se conectan e interactúan los distintos elementos que se hacen presentes en tiempo de ejecución; llámense procesos, objetos, clientes, servidores, medios de almacenamiento, etc.

Como se puede observar, los principales componentes del sistema están conectados lógicamente entre sí, no requieren una interfaz especial de conexión, sino que todos forman parte del mismo componente superior en la jerarquía, que se encarga de comunicarlos entre sí. El controlador consume servicios tanto de la vista como del modelo, los cuales brindan respuesta a sus peticiones, para que este pueda contestar las solicitudes del usuario. A su vez, el servidor web que aloja el sitio web, enruta las request enviadas por el usuario para que sean procesadas por el controlador; el cual le devolverá posteriormente el response de forma que el servidor web pueda contestar al usuario. Queda a la vista que todas las relaciones se establecen con un esquema cliente-servidor.

Por último, el servidor es administrado de forma remota por una interfaz provista por el web hosting llamada cPanel. A través de la misma el administrador web puede modificar las configuraciones del servidor web, acceder a la base de datos, o modificar la versión del sistema que es ejecutada actualmente.

Vista de Instalación

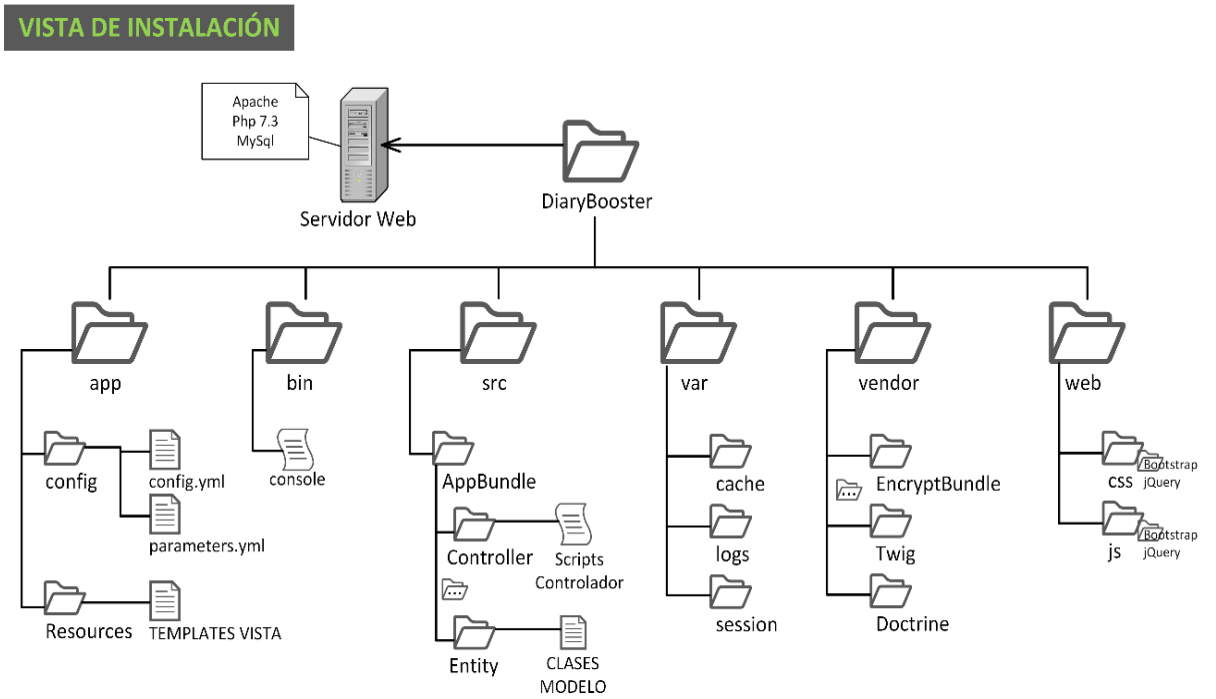


IMAGEN 003. Diagrama de arquitectura – Vista de instalación. Representación de cómo se distribuyen los principales archivos y directorios que componen el código.

La vista de instalación presenta la distribución de los principales componentes en el sistema de ficheros de DiaryBooster. No se muestra la estructura completa del filesystem, solamente la ubicación de aquellos ficheros o módulos considerados más relevantes.

A grandes rasgos, se trata de la distribución estándar de un proyecto de software construido con el framework de desarrollo Symfony v3.4, con algunos detalles particulares, como la inclusión de algunas librerías gráficas específicas, o el bundle de encriptación utilizado para codificar los datos sensibles antes de introducirlos a la base de datos.

A raíz de lo expuesto en el gráfico, podemos señalar que aquí se están incluyendo las siguientes tácticas propuestas anteriormente: la utilización de un framework de desarrollo que permite una fácil instalación y acoplamiento de nuevas librerías sin afectar las existentes (en el caso de Symfony, las mismas se instalan generalmente en el vendor); así como la codificación de los registros sensibles antes de introducirlos a la base de datos, y solo decodificarlos para su presentación al usuario, lo cual se aprecia con la presencia del EncryptBundle.

Vista de despliegue

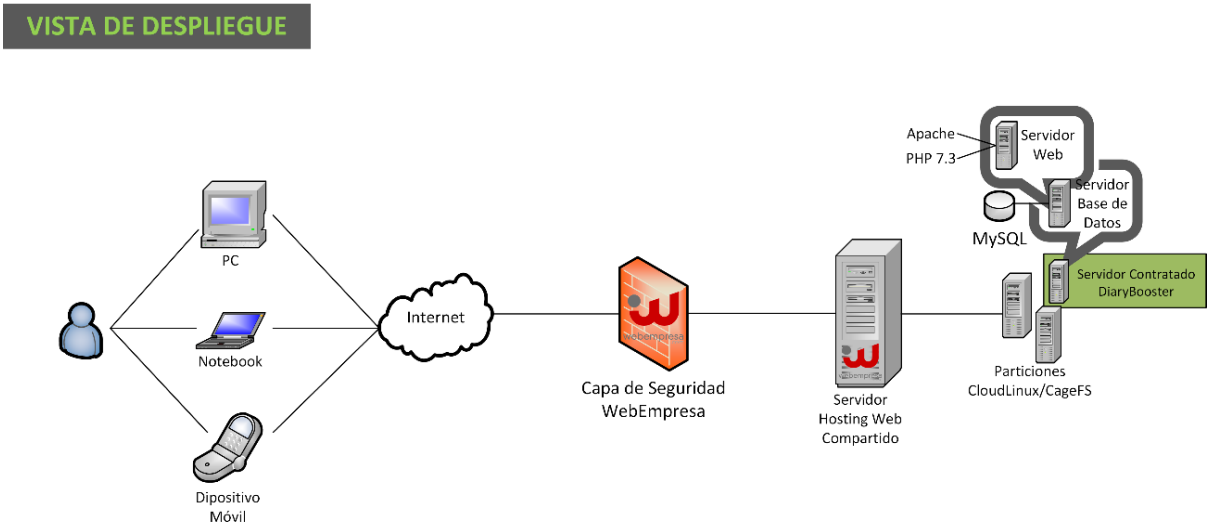


IMAGEN 004. Diagrama de arquitectura – Vista de despliegue. Topología de los componentes sobre los cuales opera el *software*.

A partir de la vista de despliegue podemos observar claramente cómo se estructura el sistema en cuanto a los componentes físicos que lo componen e intervienen en su utilización.

Teniendo en cuenta la envergadura del proyecto, se escogió utilizar un servicio web hosting de tercero, ya que sería improductivo montar un servidor propio para ser expuesto a internet debido a que sería utilizado por una cantidad de entre 5 y 50 usuarios. Al mismo tiempo, otro factor preponderante fue el costo que implica contar con infraestructura propia, en comparación con las tarifas anuales de Web Empresa (el proveedor escogido), y la facilidad que brinda tanto en tiempo como facturación. Sumado a los puntos ya nombrados, las soluciones preestablecidas que brindan los servicios de este tipo referentes a seguridad, disponibilidad, configuración, entre otras, nos posibilitaron enfocarnos principalmente en la construcción del software y no tanto en requisitos de infraestructura. Con esta consideración, pudimos entregarle valor al cliente en una etapa temprana.

6. Gestión del proyecto

En esta sección, abordaremos la puesta en práctica del plan de proyecto, especificaremos cuáles fueron los pasos realizados, las distintas etapas en las que se dividió tanto temporalmente como a nivel de tareas e iteraciones, los principales puntos a tener en cuenta a la vista del ciclo de vida de un sistema de información (en sus primeras etapas), y el detalle del trabajo realizado.

Conforme a lo establecido por la metodología de desarrollo escogida, el proyecto se realizó de forma incremental a partir de iteraciones. Concretamente, fueron empleadas tres iteraciones de distinta duración. Además del tiempo de extensión, también fueron caracterizadas por la preponderancia de algunas de las etapas del proceso unificado, por las tareas abarcadas y los entregables obtenidos. Para explicar en mayor detalle las particularidades de cada una, primero profundizaremos un poco acerca del marco de trabajo propuesto por RUP.

La metodología Proceso Unificado consta de cuatro ***etapas*** fundamentales:

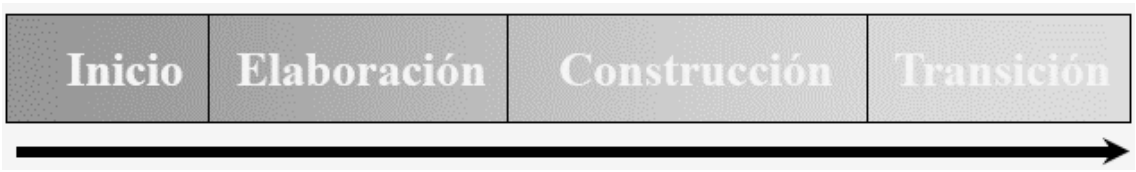


IMAGEN 005. Fases que componen el proceso unificado de desarrollo.

Inicio: aquí hemos definido:

- Alcance del proyecto.
- Visión preliminar del sistema.
- Casos de negocio.

Elaboración: afinamos la visión, especificamos en detalle la mayoría de los casos de uso y diseñamos la arquitectura del sistema.

Construcción: los objetivos para esta fase fueron los siguientes:

- Un plan de proyecto para la fase de transición.
- Un producto listo para ser entregado a los usuarios finales.
- El *software* integrado en una plataforma adecuada.
- Una descripción de la versión actual.

Transición: en esta fase entregamos el producto a los usuarios para su validación y despliegue.

Todas estas etapas atravesaron en mayor o menor medida las distintas iteraciones que componen el proyecto en su conjunto. Más allá de las particularidades que puedan existir, siempre existe una versión productiva del sistema al final de una iteración.

Para llevar adelante la construcción de DiaryBooster, alcanzar el éxito en el desarrollo del proyecto y abarcar todas las necesidades acordadas con el cliente, se planificaron

desde el comienzo tres iteraciones con distinta duración y cargas de trabajo. La estipulación inicial de los tiempos de cada etapa no fue la que aconteció finalmente, por motivos que trataremos en detalle más adelante. No obstante, las cargas de trabajo asignadas sí se cumplieron acorde al plan, y si ponderáramos los tiempos de acuerdo a las tareas incluidas en cada una, si bien no fueron los planeados, vemos que se encuentran en proporción a lo establecido al comienzo.

Iteración 1

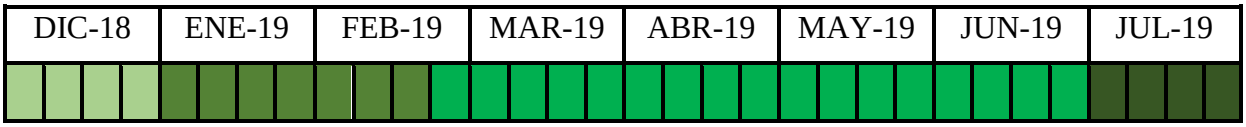


IMAGEN 006. Distribución temporal de la primera iteración, marcando el período abarcado por cada fase: Inicio, Elaboración, Construcción y Transición.

Iteración 2

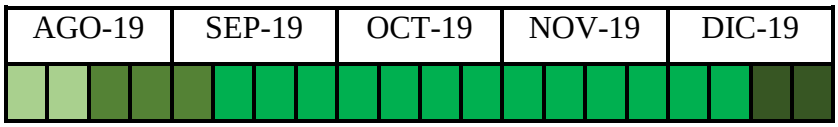


IMAGEN 007. Distribución temporal de la segunda iteración, marcando el período abarcado por cada fase: Inicio, Elaboración, Construcción y Transición.

Iteración 3

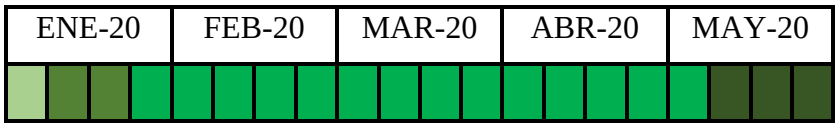


IMAGEN 008. Distribución temporal de la última iteración, marcando el período abarcado por cada fase: Inicio, Elaboración, Construcción y Transición.

Las tareas ejecutadas en cada una de las iteraciones fueron seleccionadas de forma acorde a los objetivos de la misma, a la versión del sistema que se estaba por construir, y a los diferentes cambios o adaptaciones que surgían. Podremos observar en las siguientes tablas el trabajo realizado en cada iteración.

Iteración 1	
Objetivo	Tareas realizadas
Construir la ***versión inicial*** del sistema, abarcando todas las funcionalidades core (módulos Clientes, Reuniones, Gestión de Tareas y Agenda) necesarias para la gestión diaria de las tareas de cada empleado.	Etapa Inicio: ● Escribimos el plan de iteración. ● Confeccionamos un modelo de casos de uso inicial. ● Generamos una evaluación inicial de riesgos. ● Creamos un modelo de dominio inicial. Etapa Elaboración: ● Refinamos el modelo de casos de uso. ● Listamos los principales atributos de calidad buscados y definimos escenarios para cada uno de ellos. ● Diseñamos y documentamos la arquitectura del sistema. ● Confeccionamos los casos de prueba más importantes. ● Generamos el modelo de Diseño. Etapa Construcción: ● Codificamos funcionalidades seleccionadas para la iteración. ● Generamos los *tests* y corrimos la *suite* de pruebas. ● Iteramos la codificación de acuerdo a los resultados de pruebas que se consideraron relevantes. Etapa Transición: ● Realizamos el plan de transición. ● Montamos la aplicación dentro del servidor. ● Realizamos las pruebas de validación del sistema con el cliente. ● Actualizamos y resguardamos todos los documentos que describían la versión del sistema.

TABLA 001. Resumen de las tareas realizadas en la iteración 1, detallado por fase.

Iteración 2	
Objetivo	Tareas realizadas
Mejorar la versión inicial de las funcionalidades *core*, adaptando a los nuevos requerimientos que surgieron del cliente, y añadiendo las funcionalidades que se dejaron fuera en la primera versión (sólo de los módulos incluidos en la versión 1.0).	Etapa Inicio: ● Realizamos el plan de iteración. ● Actualizamos el plan de riesgos. ● Actualizamos el modelo de dominio incluyendo las nuevas entidades. ● Relevamos los nuevos casos de uso. Etapa Elaboración: ● Refinamos el modelo de casos de uso, detallando cada uno. ● Revisamos que la definición de la arquitectura se mantenga acorde a los cambios introducidos. ● Confeccionamos los casos de prueba más importantes. ● Actualizamos el modelo de diseño. Etapa Construcción: ● Codificamos funcionalidades seleccionadas para la iteración. ● Generamos los *tests* y corrimos la *suite* de pruebas. ● Iteramos la codificación de acuerdo a los resultados de pruebas que se consideraron relevantes. Etapa Transición: ● Realizamos el despliegue de la nueva versión en el servidor web. ● Corrimos los comandos y los *scripts* de actualización de bases de datos. ● Realizamos las pruebas de validación del sistema con el cliente. ● Actualizamos y resguardamos todos los documentos que describían la versión del sistema.

TABLA 002. Resumen de las tareas realizadas en la iteración 2, detallado por fase.

Iteración 3	
Objetivo	Tareas realizadas
Añadir las últimas modificaciones a las bandejas de entradas, incluir la vista del resumen de trabajo del cliente, y construir los módulos Financiero/Configuración, que son transversales a los demás.	Etapa Inicio: ● Realizamos el plan de iteración. ● Actualizamos el plan de riesgos. ● Actualizamos el modelo de dominio incluyendo las nuevas entidades. ● Relevamos los nuevos casos de uso. Etapa Elaboración: ● Refinamos el modelo de casos de uso, detallando cada uno. ● Revisamos que la definición de la arquitectura se mantenga acorde a los cambios introducidos. ● Confeccionamos los casos de prueba más importantes. ● Actualizamos el modelo de diseño. ● Maquetamos las pantallas más importantes. Etapa Construcción: ● Codificamos funcionalidades seleccionadas para la iteración. ● Generamos los tests y corrimos la suite de pruebas. ● Iteramos la codificación de acuerdo a los resultados de pruebas que se consideraron relevantes. Etapa Transición: ● Realizamos el despliegue de la nueva versión en el servidor web. ● Corrimos los comandos y los scripts de actualización de bases de datos. ● Realizamos las pruebas de validación del sistema con el cliente. ● Actualizamos y resguardamos todos los documentos que describían la versión del sistema.

TABLA 003. Resumen de las tareas realizadas en la iteración 3, detallado por fase.

Finalmente, al acabar todas las iteraciones, contamos con los siguientes entregables que conforman DiaryBooster, que constituyen el resultado del proceso de desarrollo:

- Presupuesto Inicial.
- Maquetas de pantallas principales.
- Especificación de requerimientos.
- Modelo de dominio.
- Modelos de diseño:
    - Diagrama de clases.
    - Diagramas de secuencia.
- Documento de especificación de arquitectura.
- Casos de prueba.
- Código ejecutable.
- Documento de despliegue.

Hasta aquí, hemos descrito los pasos seguidos para llevar adelante el desarrollo del sistema web DiaryBooster, en conjunto con el lapso temporal abarcado por estas tareas, y los resultados obtenidos. No obstante, en las siguientes secciones describiremos con mayor nivel de detalle los puntos que consideramos más relevantes para una mejor comprensión acerca de cómo fue resuelto el proyecto.

6.1 Elicitación de requerimientos

En un proceso de Ingeniería de Software, el relevamiento y la determinación de los requerimientos de un sistema a construir es una de las etapas más importantes del ciclo de vida de cualquier sistema.

Esta parte del proceso, requiere un conocimiento profundo acerca del negocio y las necesidades del cliente. Existen diferentes alternativas para alcanzar este conocimiento, así como para plasmarlo posteriormente en un documento que permita al equipo de desarrollo construir el sistema correspondiente.

Trataremos a continuación cuáles fueron las estrategias utilizadas en el proceso de elicitación de requerimientos, así como los resultados conseguidos.

6.1.1 Acercamiento Inicial

El primer paso a seguir fue reunirnos con el cliente para conocer cuál era el proyecto a realizar, analizar la factibilidad del mismo, y si resultaba de interés para nosotros. El objetivo de la reunión inicial era presentar a todos los participantes, sus intereses, y conocer un poco más en profundidad las necesidades del negocio, y las posibles soluciones que podíamos aportar como equipo de trabajo. El resultado consistió en la siguiente evaluación inicial: el cliente requería un sistema de gestión de tareas y recursos de su negocio. La solución más apropiada para sus requisitos era un sistema web, y el equipo de trabajo estaba en condiciones de llevar adelante el proyecto planteado.

A continuación, se acordaron dos nuevas reuniones con el cliente. La primera, para detallar cuáles eran las funcionalidades principales y las restricciones que operarían sobre el sistema. A partir de este trabajo en conjunto, el equipo de desarrollo estaría en condiciones de generar un presupuesto inicial. En un segundo encuentro se acordaría el monto final del presupuesto.

Luego de la tercera reunión con el cliente, acordamos dar inicio al proyecto de desarrollo de software, considerando un desarrollo a medida para el estudio contable Gaudiano, pero con la cesión completa de los derechos sobre el software al equipo de desarrollo, de forma tal que este pudiera ser adaptado y revendido a otros clientes.

6.1.2 Presupuesto

En el presupuesto se establecieron las solicitudes del cliente que fueron abarcadas en el proyecto, y algunas que se podrían implementar en futuros desarrollos. Dicho presupuesto constituyó el alcance inicial acordado. Posteriormente, el resultado del proyecto se alejó un poco de lo establecido en este plan inicial, porque determinadas funcionalidades y su complejidad asociada no habían sido detalladas en su totalidad, y el desarrollo completo de las mismas afectarían las estimaciones iniciales. Por ello, finalmente se eliminaron algunos de los módulos contemplados en el presupuesto inicial, que no eran tan relevantes para el cliente como las funcionalidades que se fueron añadiendo en el proceso de desarrollo.

6.1.3 Entrevistas

Las entrevistas o reuniones con el cliente fueron el principal medio de relevamiento de requerimientos y validación de avance. Estas reuniones se llevaban a cabo con intervalos de tres semanas, con ciertas modificaciones en base al desarrollo del proyecto o a eventuales problemas.

En las entrevistas iniciales, conversamos respecto a las necesidades del cliente y las íbamos transformando en requerimientos funcionales, que posteriormente originaron los casos de uso. Con el paso del tiempo y el avance del proyecto, los progresos realizados se fueron haciendo presentes, y en base a estos desarrollos proseguía el diálogo con el cliente acerca de los refinamientos necesarios, o de las nuevas funcionalidades a añadir. Al igual que en las reuniones iniciales, esto derivaba en la actualización de casos de uso existentes, o en la creación de nuevos.

La medida de avance real era el código generado por el equipo de trabajo y la funcionalidad que le mostrábamos al cliente en cada entrevista. Sin embargo, también se exponían los casos de uso relevados y el detalle escrito sobre los mismos. Esto nos sirvió de soporte en determinadas ocasiones en las que surgieron disputas respecto de lo acordado previamente.

6.1.4 Documento de especificación de requerimientos

Todos los detalles y acuerdos generados en las entrevistas con el cliente se dejaron asentados en notas que el equipo de trabajo tomó en el transcurso de las mismas. A continuación, estas notas se pasaron en limpio en el documento de especificación de requerimientos, y se convirtieron en uno o más casos de uso. Conforme a lo establecido previamente en la metodología de desarrollo, se generó un modelo de casos de uso,

contemplando los distintos actores que interactúan con el sistema, y todas las funcionalidades que se definieron para DiaryBooster.

6.2 Gestión de riesgos

Una de las partes fundamentales de cualquier proyecto es analizar qué tipos de riesgos pueden presentarse al momento de comenzar a trabajar. Esto nos permite ser conscientes del panorama que se puede presentar frente a diferentes acciones para evitar tener escenarios más difíciles de afrontar.

Aquí expresaremos los riesgos detectados durante la planificación del proyecto y que gestionamos durante el mismo. En la siguiente tabla, comentaremos cómo llevamos a cabo cada uno de los riesgos detectados, qué acción tomamos en base a eso y qué impacto tuvo realmente en base al calculado:

Id Riesgo	Riesgo Identificado	Categoría	Subcategoría	Impacto	Probabilidad	ER
1	Estimaciones de tiempo incorrectas en la etapa de planificación	Dirección de Proyecto	Estimación	50	50%	25
2	Usuarios no interesados	Recursos Humanos	Comportamiento	75	25%	18,75
3	Demoras debido a la interrupción de un servicio de terceros	Externos	Subcontratistas y proveedores	75	10%	7,5
4	Caída de servidores de aplicaciones y bases de datos	Externos	Subcontratistas y proveedores	90	10%	9
5	Falta de servicios básicos para la utilización del sistema (computadoras, internet, energía	Externos	Cliente	50	10%	5

	eléctrica, etcétera.)					
6	Incumplimiento del cronograma del proyecto	Organizacional y Planeación	Autocontrol	25	25%	6,25
7	Velocidad de desarrollo menor a la planificada	Organizacional y Planeación	Autocontrol	50	50%	25
8	Falta de control en el seguimiento por parte del director del proyecto	Organizacional y Planeación	Direccionamiento estratégico	75	10%	7,5
9	Pérdida de interés del cliente en llevar adelante el proyecto	Externos	Cliente	90	10%	9
10	La tecnología disponible no es adecuada para realizar el proyecto	Procesos y Sistemas	Tecnología	75	10%	7,5
11	Imposibilidad de cumplir con la cantidad de horas de trabajo semanales de cada recurso	Recursos Humanos	Comportamiento	50	10%	5
12	Falta de experiencia en tecnologías	Recursos Humanos	Conocimiento, experiencia y destrezas	75	75%	56,25
13	Un integrante del equipo abandona el proyecto.	Recursos Humanos	Disponibilidad de personal	90	10%	9
14	Surgimiento de obligaciones laborales o aumento en la carga horaria de	Recursos Humanos	Disponibilidad de personal	75	75%	56,25

	los integrantes del proyecto					
15	Disminución de la calidad del producto generada por la utilización de una tecnología desconocida.	Recursos Humanos	Conocimiento, experiencia y destrezas	75	50%	37,5
16	Ausencia temporal de integrantes	Recursos Humanos	Disponibilidad de personal	10	50%	5
17	Dificultad de aprendizaje del sistema	Técnicos	Diseño	75	25%	18,75
18	*Performance* por debajo de los niveles aceptables	Técnicos	Diseño	90	50%	45
19	El sistema no cumple los objetivos planteados	Técnicos	Requisitos	90	10%	9
20	Filtración de información sensible	Técnicos	Fiabilidad	75	25%	18,75
21	La arquitectura seleccionada no se ajusta a las necesidades del proyecto	Técnicos	Diseño	90	25%	22,5
22	Ausencia de funcionalidades críticas en las primeras implementaciones	Técnicos	Requisitos	90	10%	9
23	Cancelación del proyecto por falta de fondos	Recursos Físicos y Financieros	Recursos Financieros	90	50%	45

24	Pérdida de los avances del proyecto por rotura o extravío de recursos informáticos	Recursos Físicos y Financieros	Instrumentos y Equipos	75	25%	18,75
25	Conflicto en el equipo de desarrollo debido a la falta de un *team leader*	Organizacional y Planeación	Autocontrol	75	10%	7,5
26	Modificación de requerimientos por parte del cliente en etapas tempranas del desarrollo	Técnicos	Requisitos	25	50%	12,5
27	Modificación de requerimientos por parte del cliente en etapas avanzadas del desarrollo	Técnicos	Requisitos	90	25%	22,5
28	Falta de compromiso por parte del cliente en brindar *feedback* del producto entregado	Externo	Clientes	50	50%	25
29	Inserción de requerimientos importantes en etapas avanzadas del proyecto	Técnicos	Requisitos	90	25%	22,5
30	Falta de pago por parte del cliente	Externos	Cliente	50	50%	25

TABLA 004. Riesgos gestionados.

En base a lo expuesto anteriormente, podemos clasificar los riesgos de la siguiente manera:

Nivel de Riesgo	ID riego
Aceptable	2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 16, 17, 19, 20, 22, 24, 25, 26
Poco riesgoso	1, 7, 15, 21, 27, 28, 29, 30
Riesgoso	12, 14, 18, 23
Muy Riesgoso	-

TABLA 005. Clasificación de riesgos utilizada en el proyecto.

6.2.1 Plan de acción para el tratamiento de riesgos

La idea fundamentalmente fue determinar qué acciones llevaríamos a cabo en caso de que ocurra algún riesgo. Para ello, definiremos tres acciones que nos permitirán gestionar cada uno:

- Asumir el riesgo.
- Evitar el riesgo.
- Reducir el riesgo.

Es de gran importancia aclarar que esto lo aplicaremos a aquellos riesgos que se encuentren entre los niveles:

- Riesgoso.
- Muy riesgoso.

En la siguiente tabla, se muestra el plan de acción para cada riesgo:

Id Riesgo	Riesgo	Estrategia	Respuesta
12	Falta de experiencia en Tecnologías	Evitar el riesgo	Cada uno de los integrantes del equipo, antes de empezar con el desarrollo, se capacitará en las diferentes tecnologías a utilizar.
14	Surgimiento de obligaciones laborales o aumento en la carga horaria de los integrantes del proyecto	Asumir el riesgo	Cada integrante se compromete a cumplir con las horas planteadas en la planificación del proyecto de *software*. En caso de no poder hacerlo se recuperarán en otra semana de trabajo.
18	*Performance* por debajo de los niveles aceptables	Evitar el riesgo	Para ello, se deberá prestar especial atención a la definición de la arquitectura del sistema, como también evitar que la complejidad del código sea muy elevada.
23	Cancelación del proyecto por falta de fondos	Reducir el riego / Asumir el riesgo	Para evitar que este riesgo se convierta en un problema para el proyecto, ofrecimos facilidad de pago a la hora de presupuestarlo (distintos planes de cuotas, y pago contra entrega – reduciendo el riesgo). Por otro lado, asumimos el riesgo ya que, al quedarnos con los derechos intelectuales del sistema, y considerar que es potencialmente comercializable, nos interesa llevar adelante el proyecto.

TABLA 006. Plan de acción para el tratamiento de los riesgos considerados más relevantes.

6.2.2 Impacto Real

En este apartado, comentaremos cómo se desarrolló el análisis de riesgos, qué impacto tuvo cada aspecto y una breve explicación de cuán importante es este análisis en base a nuestra experiencia.

A continuación, detallaremos nuestra experiencia sobre aquellos riesgos que hemos clasificado de gran impacto para el proyecto.

Falta de experiencia en Tecnologías

Impacto Estimado: 75

Probabilidad de ocurrencia: 75%

Impacto Real: 50

Al no tener gran conocimiento en las tecnologías utilizadas, este riesgo ha tomado lugar a lo largo del inicio del proyecto y en algunas situaciones en las cuales decidimos insertar nuevas tecnologías o funcionalidades. El mismo se podría haber reducido drásticamente si hubiésemos seleccionado una tecnología conocida, pero consideramos factible el hecho de tomar un riesgo con Symfony porque nos permitiría tener un panorama de negocios y tecnologías más amplio en un corto lapso de tiempo.

Surgimiento de obligaciones laborales o aumento en la carga horaria de los integrantes del proyecto

Impacto Estimado: 75

Probabilidad de ocurrencia: 75%

Impacto Real: 75

Sabíamos que este riesgo podría tener un gran impacto debido a la situación personal de cada uno de los miembros del equipo. Ocurrieron cambios -en gran medida previstos- que hicieron extender los tiempos iniciales. Hemos pasado por situaciones que han generado una gran pérdida de tiempo para que el proyecto se lleve a cabo en el tiempo planeado.

Performance por debajo de los niveles aceptables

Impacto Estimado: 90

Probabilidad de ocurrencia: 50%

Impacto Real: 40

Si tenemos en cuenta nuestra falta inicial de experiencia con dicha tecnología, podemos decir que hemos evitado el impacto de este riesgo en un gran porcentaje. Hemos utilizado buenas prácticas para el desarrollo, logrando así, buenos resultados para el cliente.

Cancelación del proyecto por falta de fondos

Impacto Estimado: 90

Probabilidad de ocurrencia: 50%

Impacto Real: 0

Es un riesgo que puede tomar lugar por varios motivos tales como: falta de fondos, pandemia, cierre del estudio. A pesar de ello, el cliente hasta el día de la fecha sigue pidiendo funcionalidades y pensando en grandes mejoras para el proyecto.

Al comienzo del proyecto no dimensionamos el impacto de los riesgos. Actualmente, podemos determinar que elaborar un estudio de riesgos puede ayudar a cualquier organización a cumplir con gran parte de sus estimaciones. A su vez, consideramos de gran importancia generar una revisión constante de esto para lograr pulir cada plan de acción y asumir que pueden aparecer nuevas contingencias.

7. Listado de requerimientos funcionales

Para generar el modelo de caso de uso se utilizó como entrada el listado de requerimientos funcionales establecido en conjunto con el cliente. Estos se estructuraron en diferentes módulos, con la intención de facilitar su comprensión e intentar proporcionar una visión global e integrada del sistema.

Funcionalidades	
Módulo	***Descripción***
Clientes	– Información centralizada de los clientes del estudio. – Repositorio centralizado de todas las personas que interactúan con el estudio, tanto personas humanas como jurídicas. – Personalización del servicio brindado a cada cliente, desde las tareas brindadas, las actividades que componen cada una, los honorarios percibidos y la gestión realizada para cada cliente.
Gestión de tareas	– Repositorio centralizado de las tareas llevadas adelante por el estudio y de las actividades que las componen, estandarizando la forma de trabajo. – Generación automática de tareas, de acuerdo a los vencimientos determinados anualmente por AFIP. – Especificación de los trabajos realizados para cada cliente, determinada por la asignación de tareas. – Distribución automatizada de los trabajos a realizar por todos los empleados del estudio. – Documentación y seguimiento de labores realizada por cada tarea de cada cliente, mediante el seguimiento de las actividades que las componen, y una bitácora de notas asociadas a cada una de las instancias de estas tareas. – Carga de los calendarios de vencimientos para cada tarea a realizar mensual o anualmente. – Bandeja de trabajo de las tareas asignadas a cada usuario, permitiéndole conocer los pendientes con cada cliente. – Bandeja de trabajo de las actividades asignadas a cada usuario, permitiéndole conocer los trabajos que debe realizar en cada día. – Personalización de las tareas para cada cliente. – Cargar guías de trabajo para ayudar a los empleados en sus labores diarias.

	– Generación de tareas rápidas, que no requieran programación, y sean realizadas ocasionalmente y de forma única por cada usuario o para cada cliente. – Posibilidad de replicar tareas rápidas previamente confeccionadas, o de convertirlas en tareas programables, con su propio calendario de vencimientos.
Reuniones	– Generación de reuniones como un tipo especial de tarea. Deben figurar junto con las instancias de tareas en las bandejas de entrada, y también deben tener su propio apartado. – Calendario en formato mensual para la visualización de las reuniones diarias y la programación semanal de las mismas. – Las reuniones deben contar también con una bitácora de notas sobre las cuales dejar registro de lo trabajado.
Finanzas	– Registro de los trabajos realizados para cada cliente, con su correspondiente compensación económica, en formato contable. – Registro de los pagos de clientes, en formato contable. – Registro de las cuentas en uso en el estudio. – Registro de los flujos de cuenta generados por el estudio, en formato contable. – Generación de las comisiones correspondientes a cada empleado por las tareas realizadas. – Asiento contable de los créditos y débitos generados para cada empleado.
Usuarios	– Gestión de usuarios para acceso y uso del sistema. – Activación/desactivación de usuarios, para mantener el histórico de sus trabajos, pero al mismo tiempo la posibilidad de dar de baja el acceso al sistema, y eliminar los parámetros de configuración que les corresponden a cada uno.

Configuración	– Parametrización de los tiempos de generación de tareas, siempre de acuerdo a los calendarios de vencimientos, pero permitiendo personalizar la cantidad de tiempo de anticipación. – Parametrización de la distribución de tareas entre los usuarios, para su asignación al momento de la generación automática de las instancias. – Configuración de los salarios mensuales de los empleados, y de los honorarios de los clientes. – Parametrización de distintas entidades agrupadoras/clasificadoras del sistema,tales como: Tipos de cuenta, Divisas, Formas de pago, Tipos de Ingreso y Egreso, Tipos de tarea.

TABLA 007. Listado de funcionalidades provistas por el sistema, estructuradas en módulos que las agrupan.

En el Documento de especificación de requerimientos, incluimos el listado de requerimientos no funcionales.

8. Modelo de Casos de uso

Sobre la base del listado recién expuesto se procedió a construir el modelo de casos de uso. La complejidad del sistema derivó en un modelo bastante grande, con más de 100 casos de uso y 3 actores para esta etapa del proyecto (si se continuara con los módulos que se dejaron fuera en el presupuesto inicial, aumentarían tanto los casos como los actores).

En el comienzo del proyecto se contabilizaron un poco más de 60 casos de uso. Esta cantidad creció considerablemente debido a que esta primera definición se hizo a un alto nivel de abstracción, y al trabajar sobre ellos y refinarlos, muchos se transformaron en múltiples casos de uso. También, se fueron añadiendo algunos nuevos requerimientos de acuerdo a lo solicitado por el cliente. Para que no se produjera un gran desajuste en cuanto a la cantidad de trabajo contemplado, lo que se hizo fue eliminar otros módulos del plan de trabajo. A pesar de ello, la complejidad de las nuevas funcionalidades resultó considerablemente mayor que aquellas eliminadas del alcance del proyecto.

Los usuarios resultantes del modelo de casos de uso fueron los siguientes: administrador, accede a todas las funcionalidades del sistema, empleado, acceso más restringido, especialmente en lo que respecta a la configuración y el módulo financiero; y el tiempo, que no es en sí un usuario, pero dispara casos de uso mediante comandos CRON.

Para documentar la especificación de cada caso de uso se utilizó una plantilla común que se fue completando de acuerdo a lo requerido por cada uno. Dicha tabla estaba compuesta por los siguientes campos: Número (CU XXX), Nombre del caso de uso (título, de qué se trata), Actor (quien es el usuario que lo instancia/inicia), Descripción (breve detalle de lo que ocurre en el caso de uso), Pre-condiciones (requisitos que se deben cumplir para poder iniciar el flujo), Flujo normal (descripción mediante una serie de pasos ordenados del flujo de acciones que deben realizarse, tanto por el usuario como por el sistema, para que el caso de uso transcurra de inicio a fin por el camino de éxito o normal esperado), Flujos alternativos (descripción de los distintos caminos alternativos que pueden suceder a partir del flujo principal, incluyendo la causa que generó este flujo secundario y los pasos que se suceden hasta finalizar el mismo), y Postcondiciones (estado posterior del sistema una vez que ha finalizado el caso de uso).

Debido a la cantidad de casos de uso implementados, no expondremos las especificaciones de cada uno en este documento. El detalle de los mismos se encuentra en el Documento de requerimientos de software, el cual forma parte de los anexos del informe.

Sin embargo, sí presentaremos los diagramas de caso de uso, que mostrarán visualmente la división que se hizo del listado de funcionalidades definidas a un alto nivel de abstracción, en requerimientos más específicos implementados por el sistema. También podremos observar las distintas relaciones que existen entre los distintos casos de uso, tanto con los actores, como entre sí.

La notación elegida para la construcción de estos diagramas fue establecida por UML. En consecuencia, los elementos con los que nos encontraremos en los diagramas son los

siguientes: actores, casos de uso y relaciones. No utilizamos subsistemas, y tampoco todos los tipos de relaciones. Existirá una única relación de herencia, utilizada solamente para establecer que el usuario de tipo administrador, es también un empleado del estudio, por lo cual hereda todos los comportamientos que puede realizar este último. Luego, entre los casos de uso y los actores, observaremos relaciones de tipo asociación, indicando que estos se comunican entre sí. Por último, entre los casos de uso existen dos tipos de relaciones: de extensión y de inclusión. El primero, indica que el comportamiento de un caso de uso puede ser incorporado a partir del siguiente. Es una relación direccional, y el origen puede ser incorporado en el destino. Por otro lado, la inclusión (también direccional) representa que el caso de uso destino debe ser incorporado en el origen para que este pueda cumplir con la funcionalidad esperada.

8.1 Diagramas de caso de uso

Jerarquía de usuarios

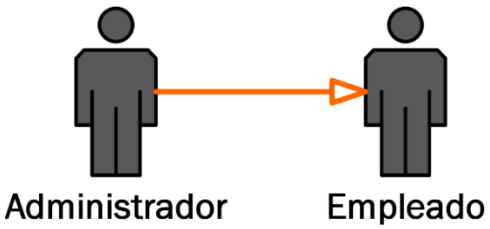


IMAGEN 009. Jerarquía de usuarios del sistema y herencia de comportamiento y funcionalidades de acuerdo al rol asignado.

Casos de uso Cliente/Persona

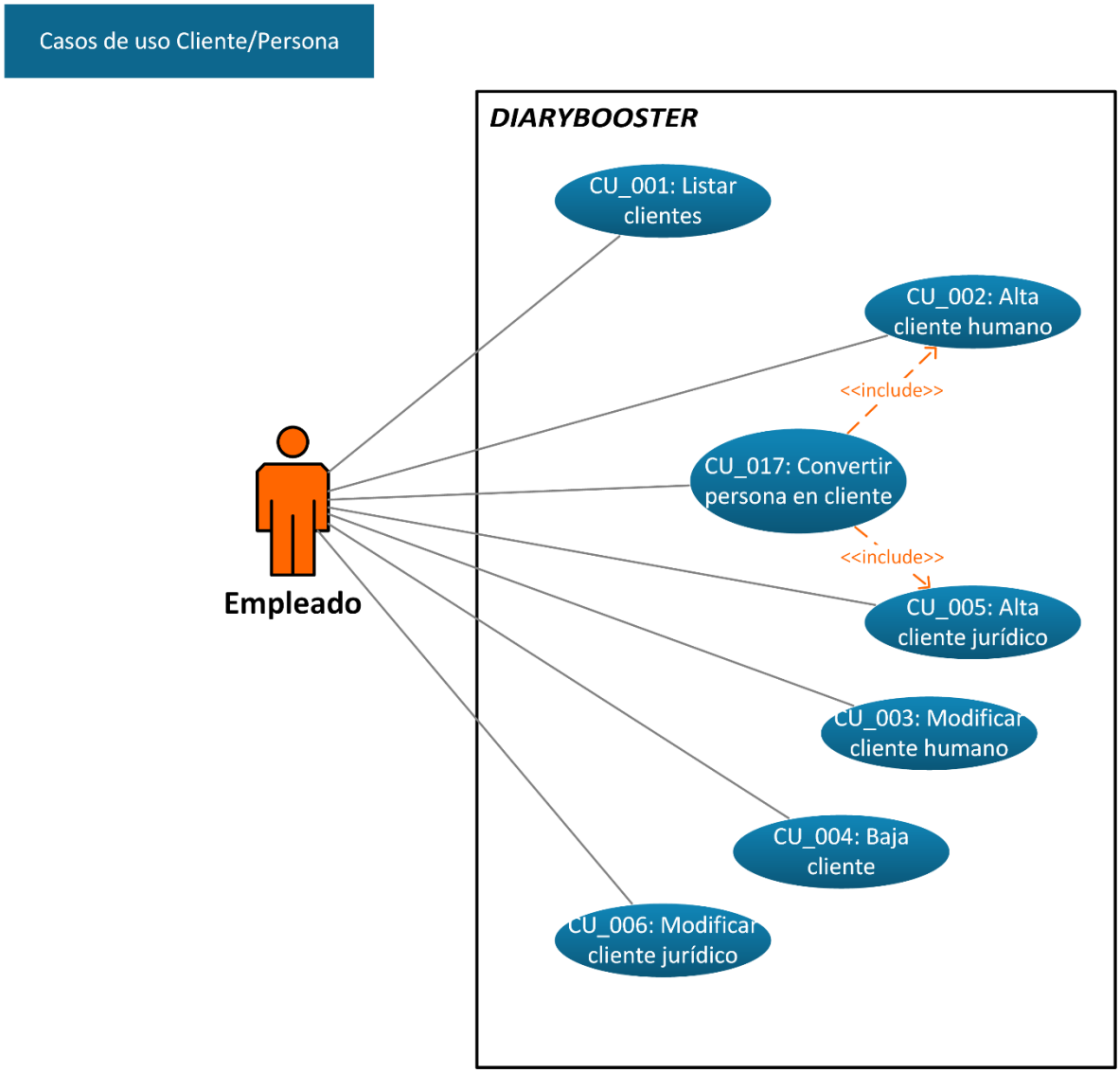


IMAGEN 010. Casos de uso relacionados a la gestión de clientes y personas. Estos se han dividido en tres imágenes, de forma que puedan ser visualizados más fácilmente. Imagen 1.

Casos de uso Cliente/Persona

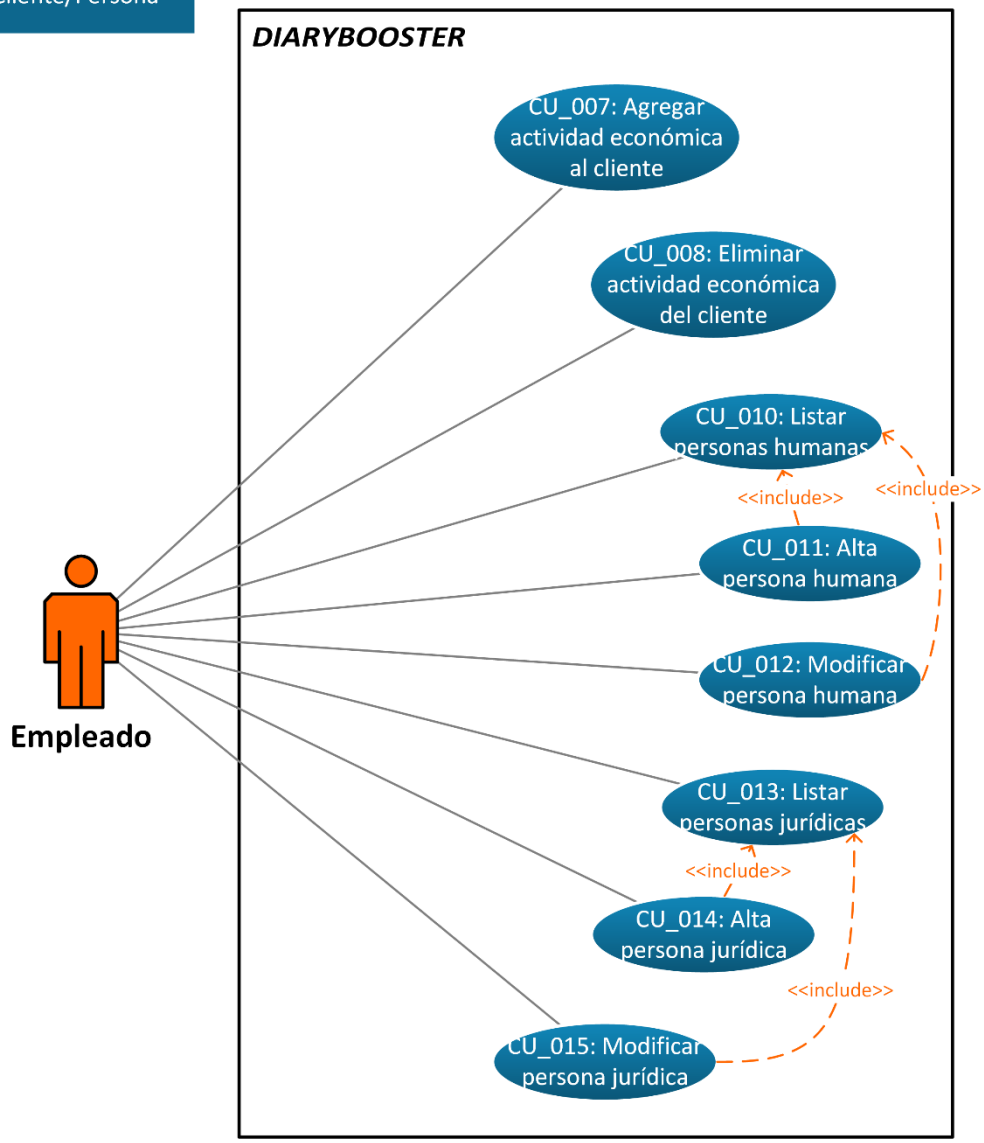


IMAGEN 011. Casos de uso relacionados a la gestión de clientes y personas. Estos se han dividido en tres imágenes, de forma que puedan ser visualizados más fácilmente. Imagen 2.

Casos de uso Cliente/Persona

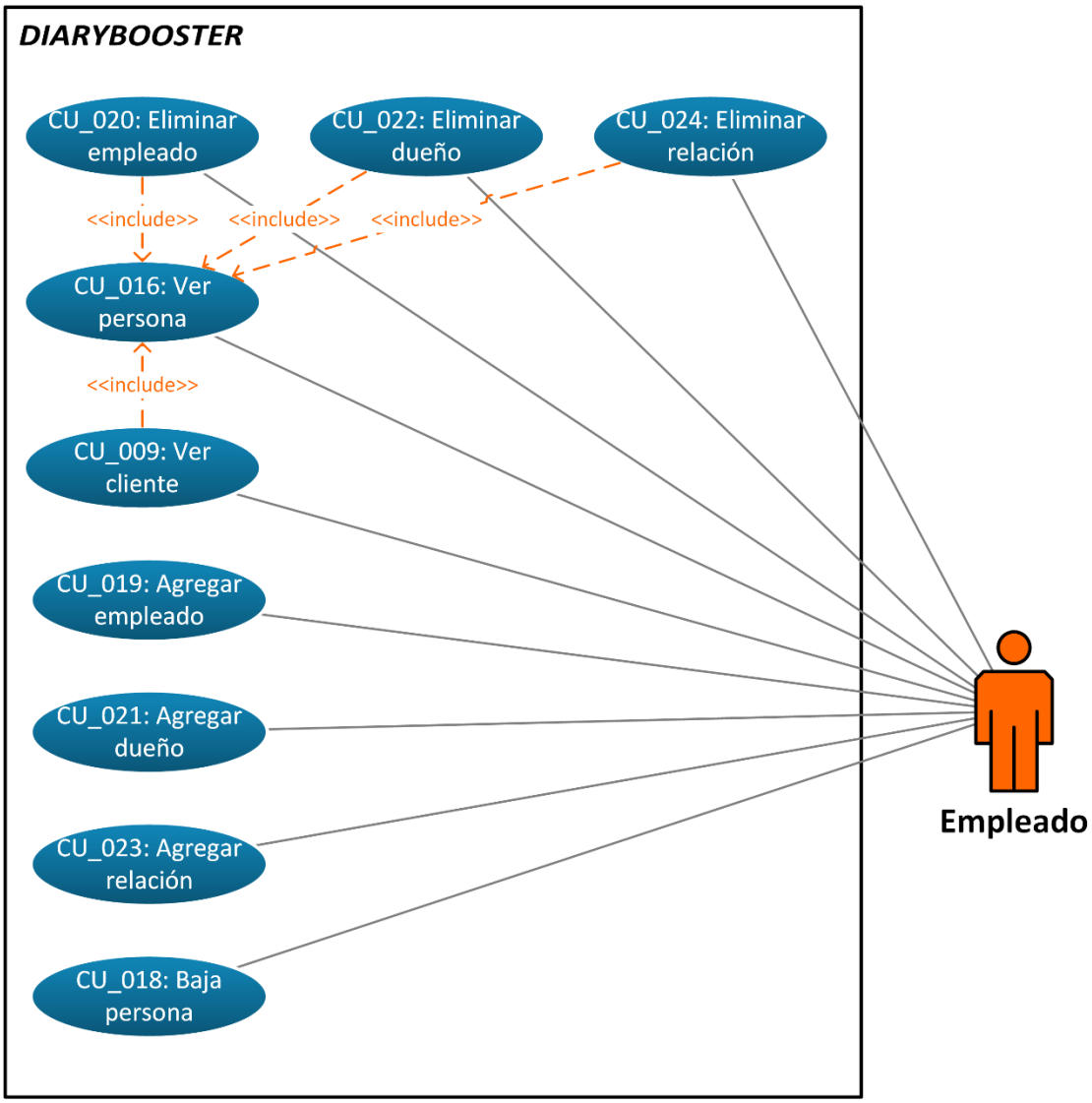


IMAGEN 012. Casos de uso relacionados a la gestión de clientes y personas. Estos se han dividido en tres imágenes, de forma que puedan ser visualizados más fácilmente. Imagen 3.

Casos de uso para gestión de tareas

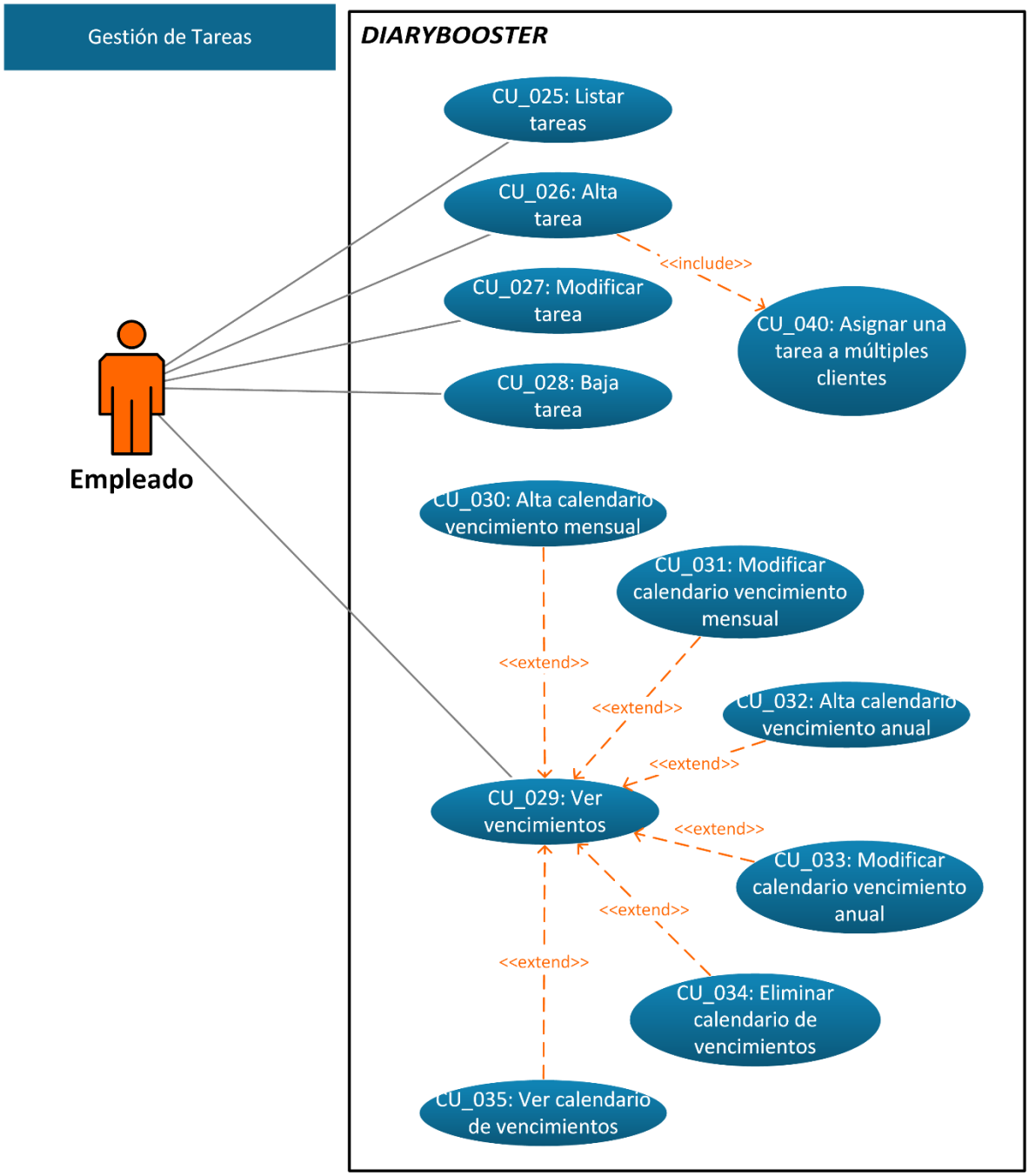


IMAGEN 013. Casos de uso relacionados a la gestión de la entidad Tarea. Incluye desde el ABM de la misma, hasta la gestión posterior de los calendarios de vencimiento.

Casos de uso de gestión de tareas cliente

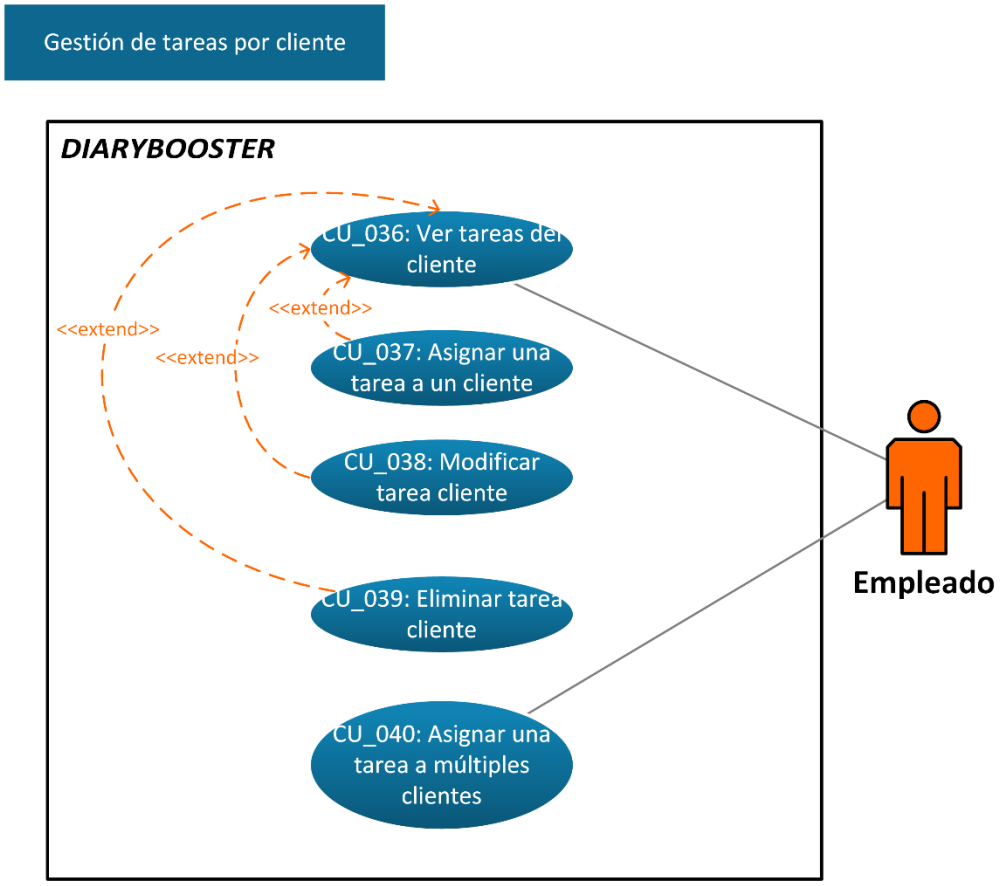


IMAGEN 014. Casos de uso relacionados a la asignación y especialización de tareas para uno o más clientes.

Casos de uso de gestión de instancias de tareas

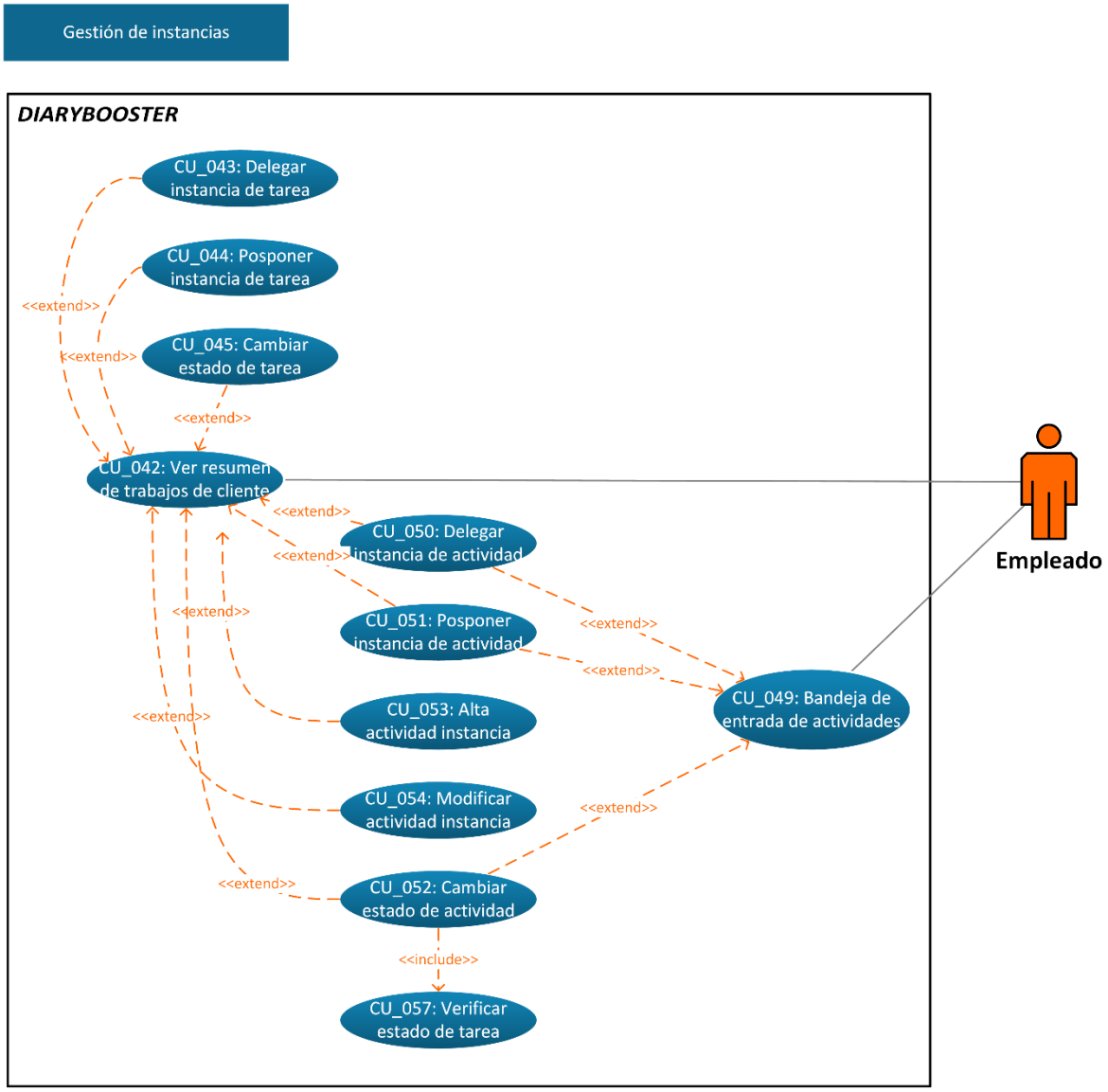


IMAGEN 015. Casos de uso relacionados a la gestión de las instancias de tarea. Estos se han dividido en dos imágenes, de forma que puedan ser visualizados más fácilmente. Imagen 1.

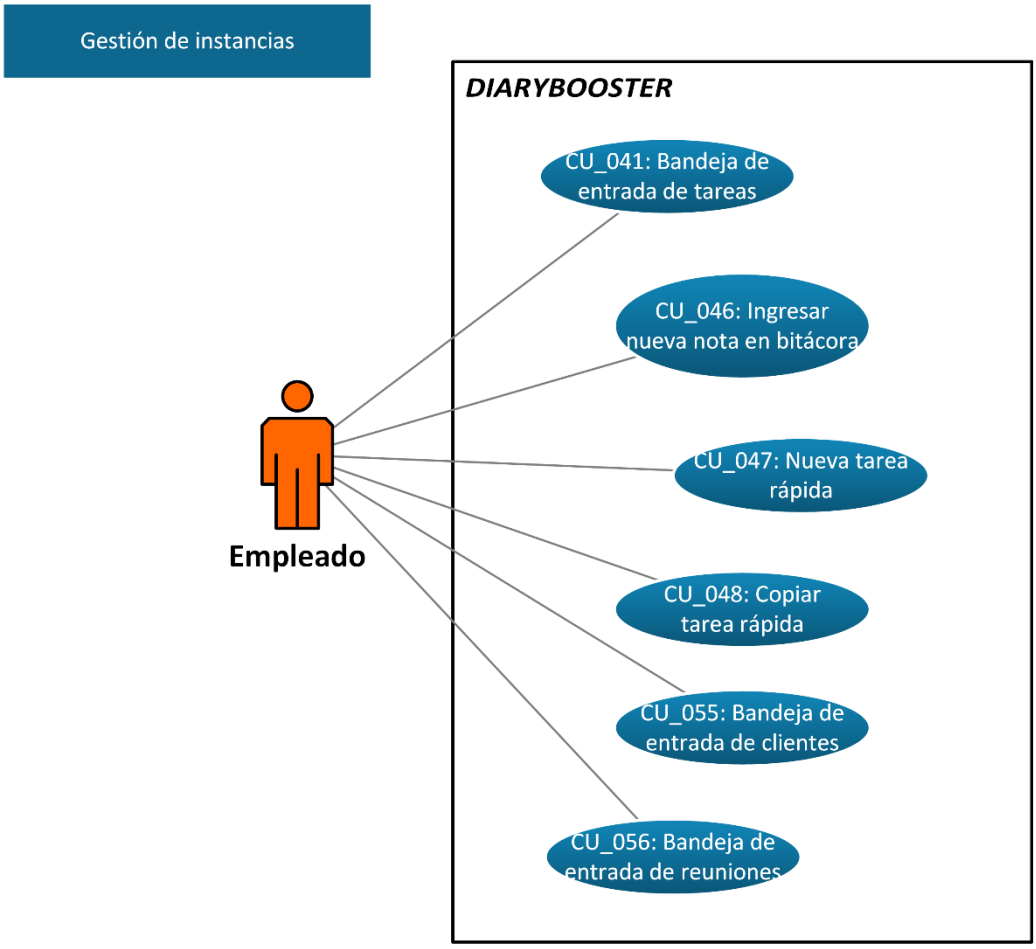


IMAGEN 016. Casos de uso relacionados a la gestión de las instancias de tarea. Estos se han dividido en dos imágenes, de forma que puedan ser visualizados más fácilmente. Imagen 2.

Casos de uso de gestión de reuniones

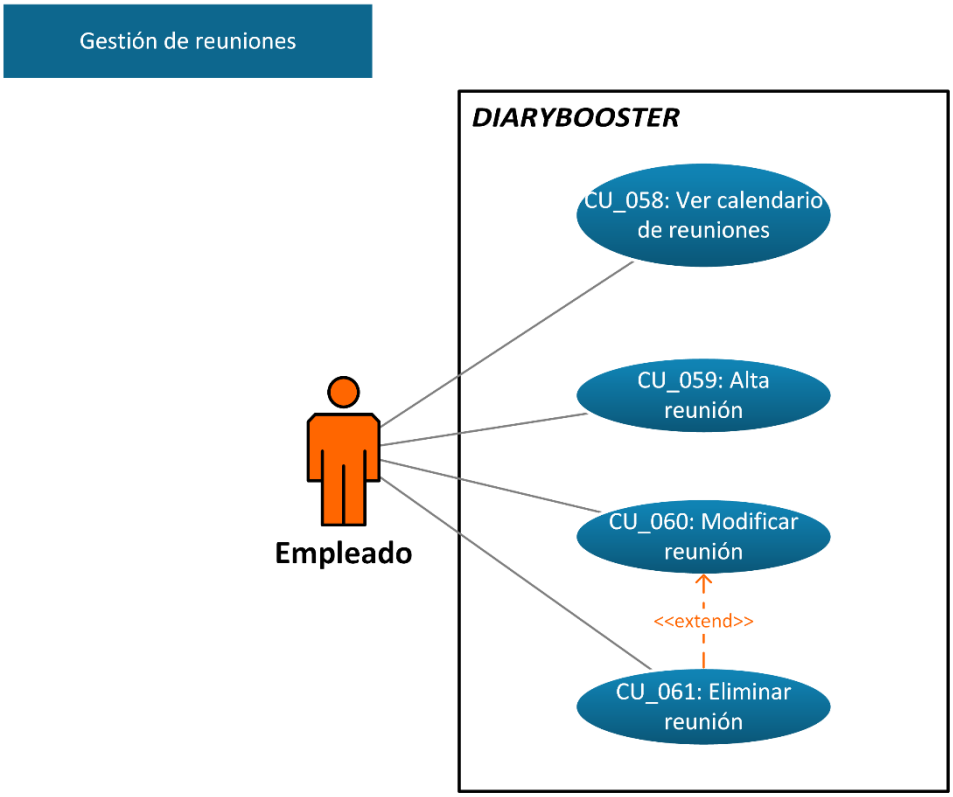


IMAGEN 017. Casos de uso que pertenecen al módulo de reuniones.

Casos de uso de gestión de cuentas

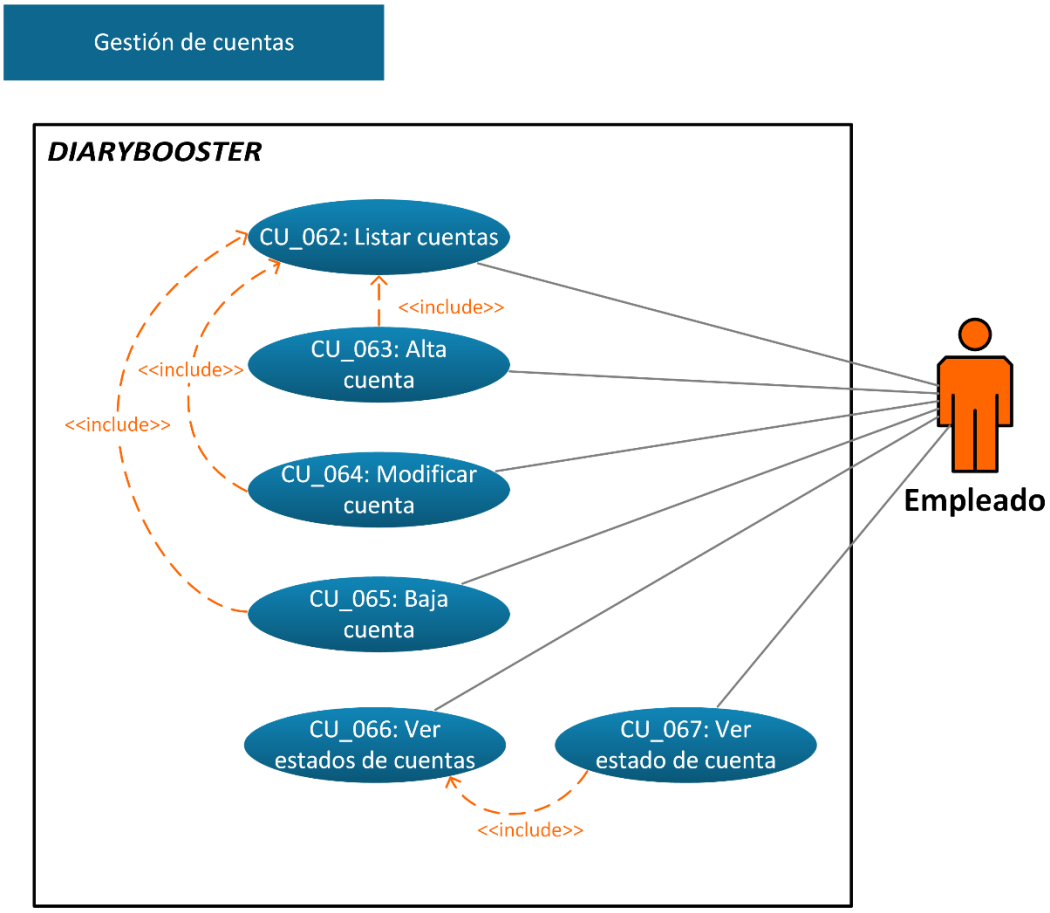


IMAGEN 018. Casos de uso que pertenecen al módulo financiero, y corresponden a la gestión de la entidad cuenta.

Gestión de cuentas

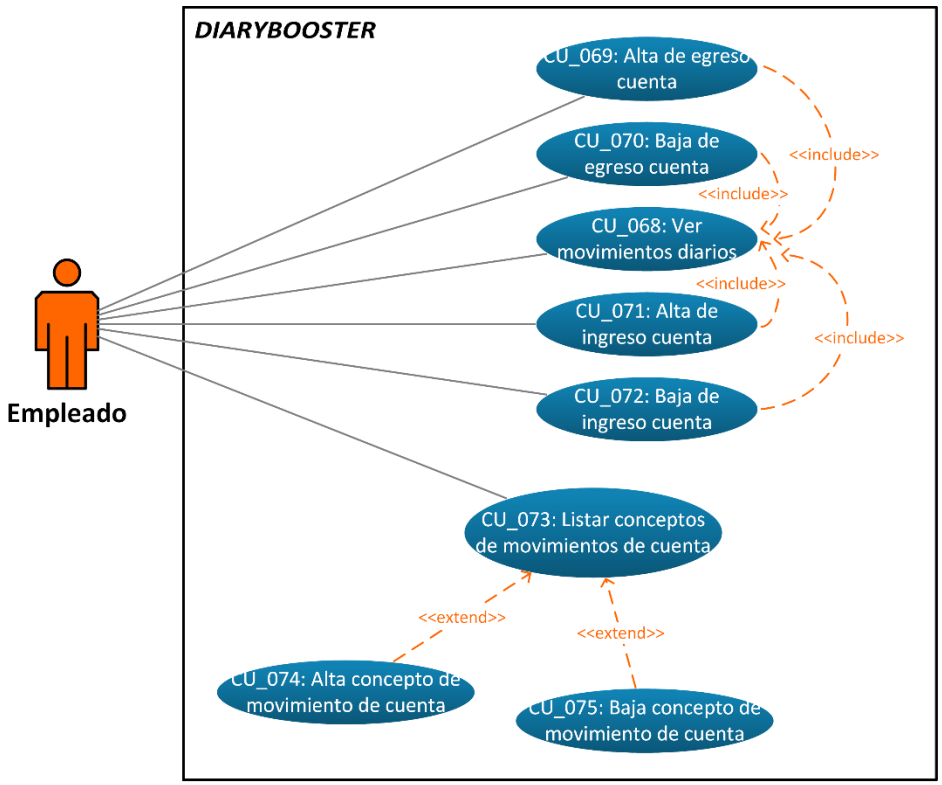


IMAGEN 019. Casos de uso que pertenecen al módulo financiero, y corresponden a la creación manual de movimientos de cuenta, y sus tipificaciones.

Casos de uso de gestión de cuentas de clientes

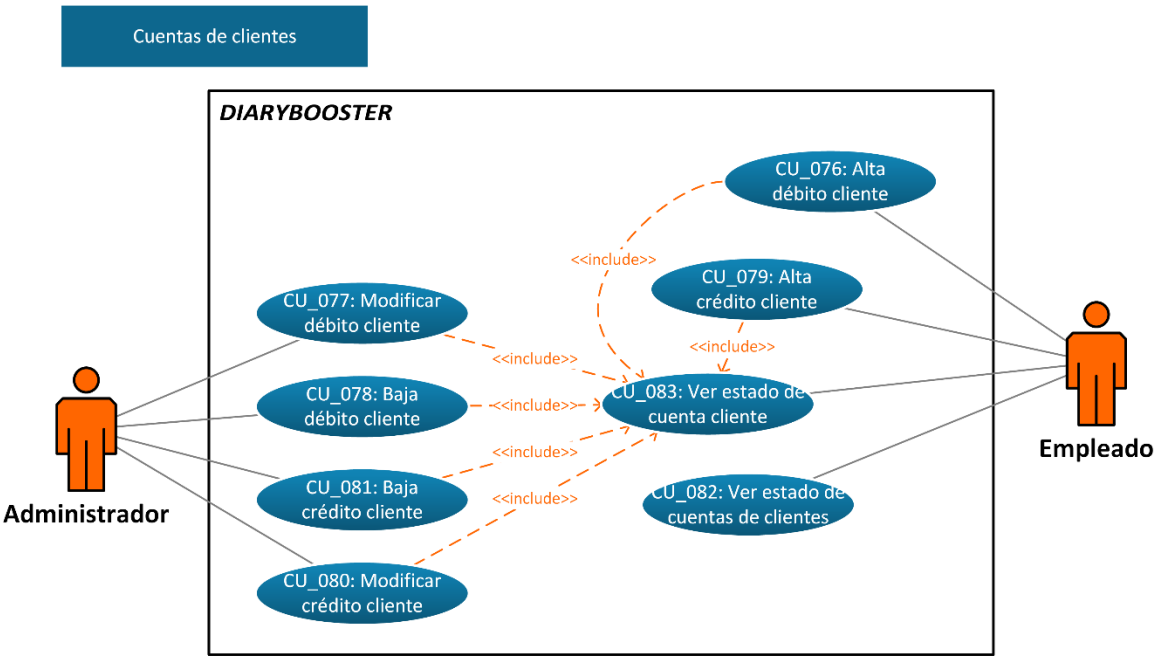


IMAGEN 020. Casos de uso que pertenecen al módulo financiero, y corresponden a la gestión de los movimientos de cuenta de cliente.

Casos de uso de gestión de cuentas de empleados

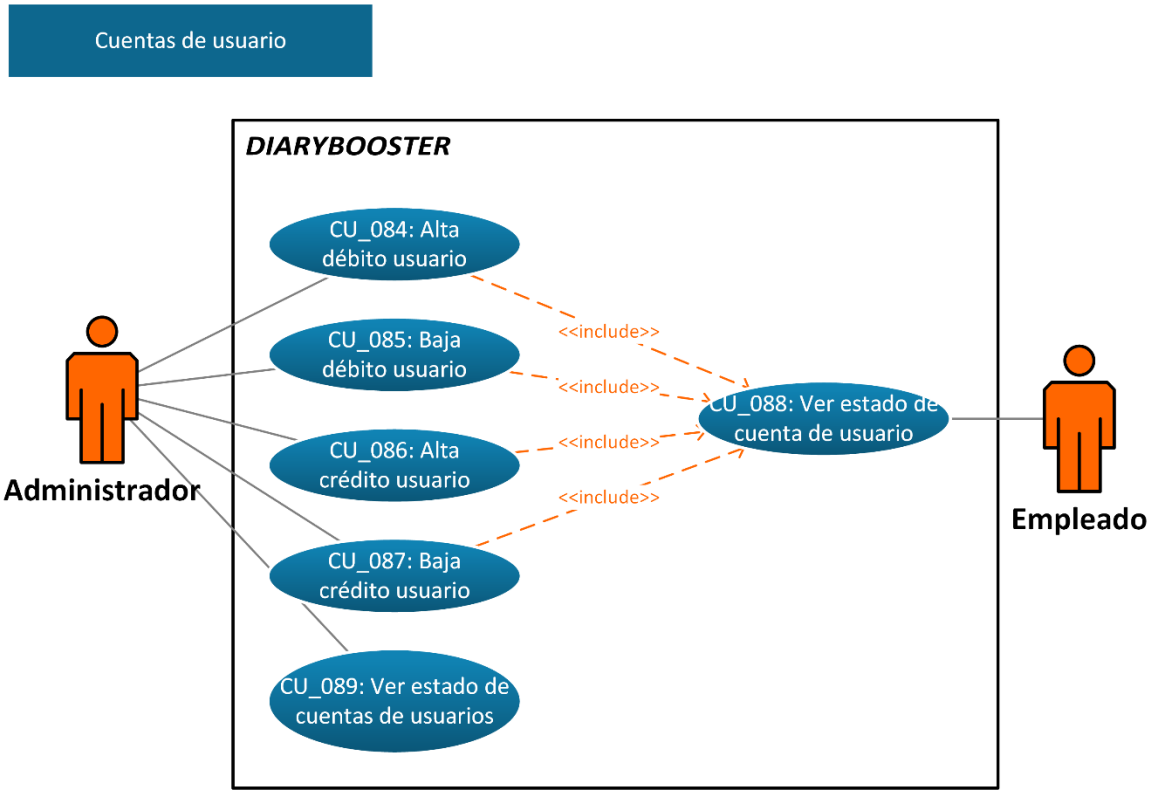


IMAGEN 021. Casos de uso que pertenecen al módulo financiero, y corresponden a la gestión de los movimientos de cuenta de usuario.

Casos de uso de configuración del sistema

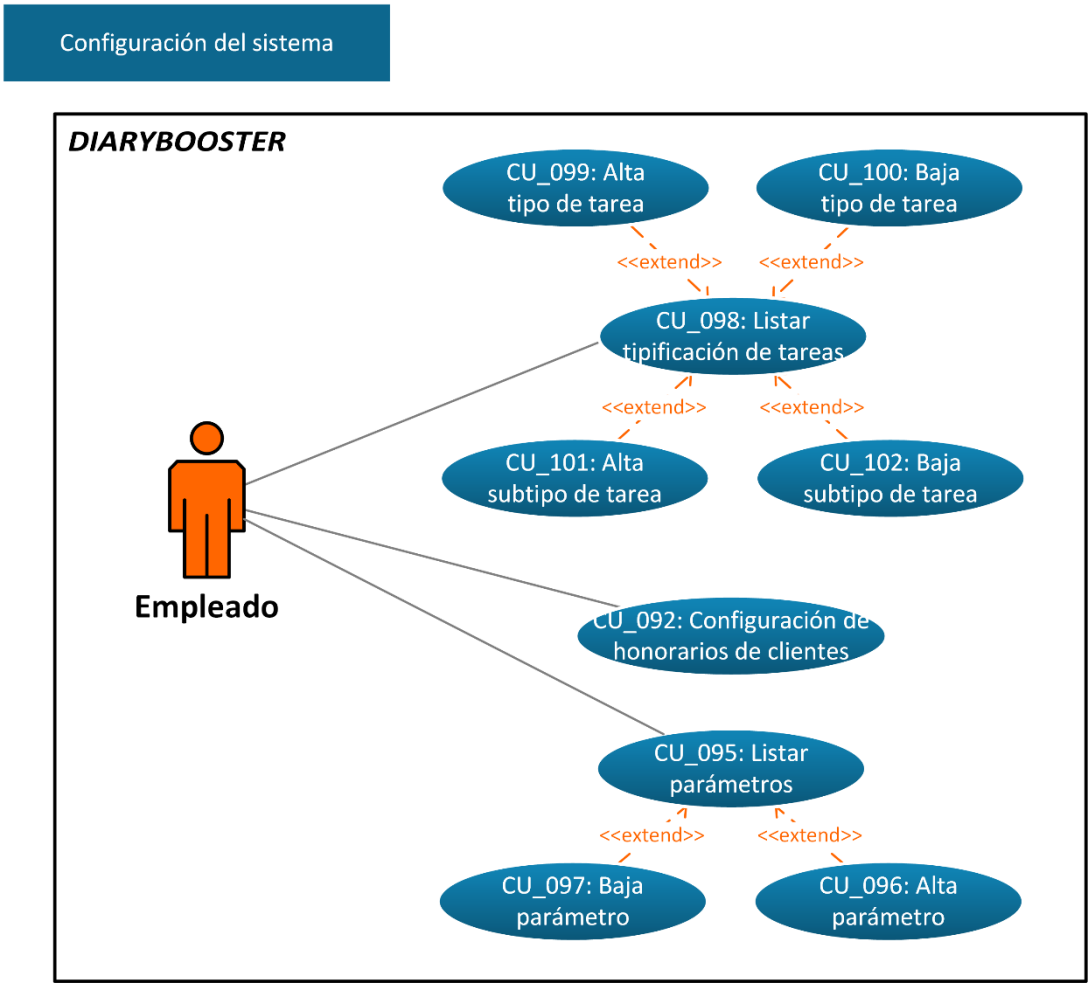


IMAGEN 022. Casos de uso que pertenecen al módulo de configuración y pueden ser instanciados por cualquier usuario.

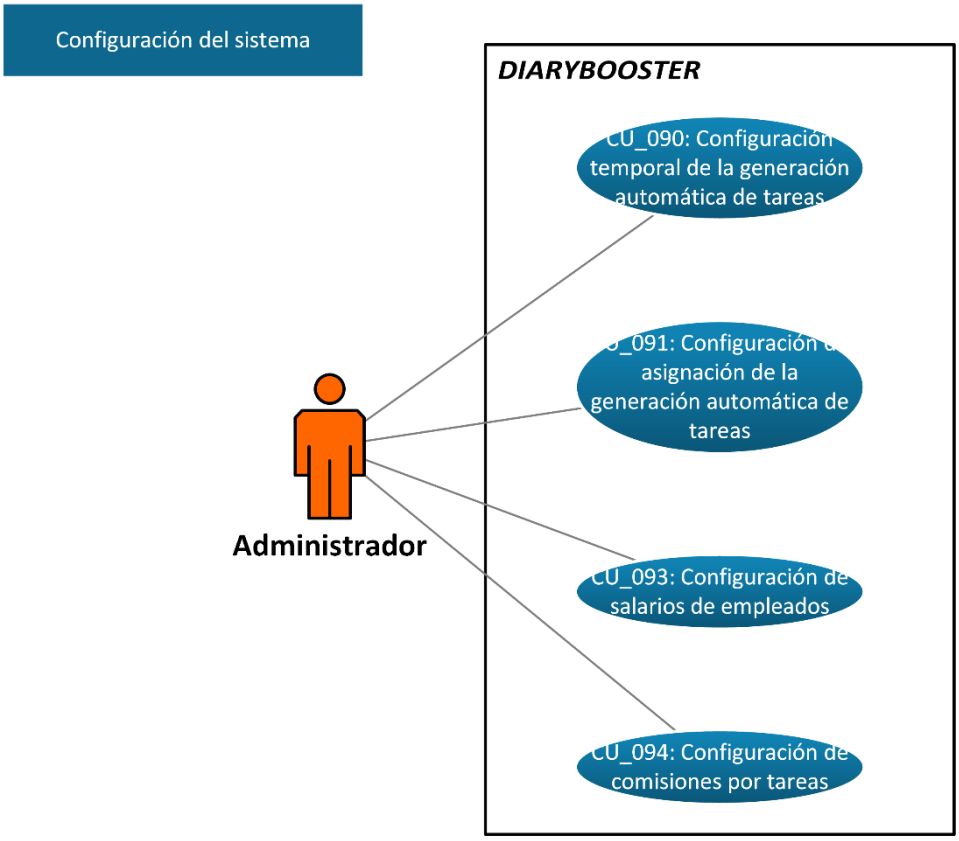


IMAGEN 023. Casos de uso que pertenecen al módulo de configuración a los cuales tiene acceso el usuario administrador exclusivamente.

Casos de uso instanciados temporalmente

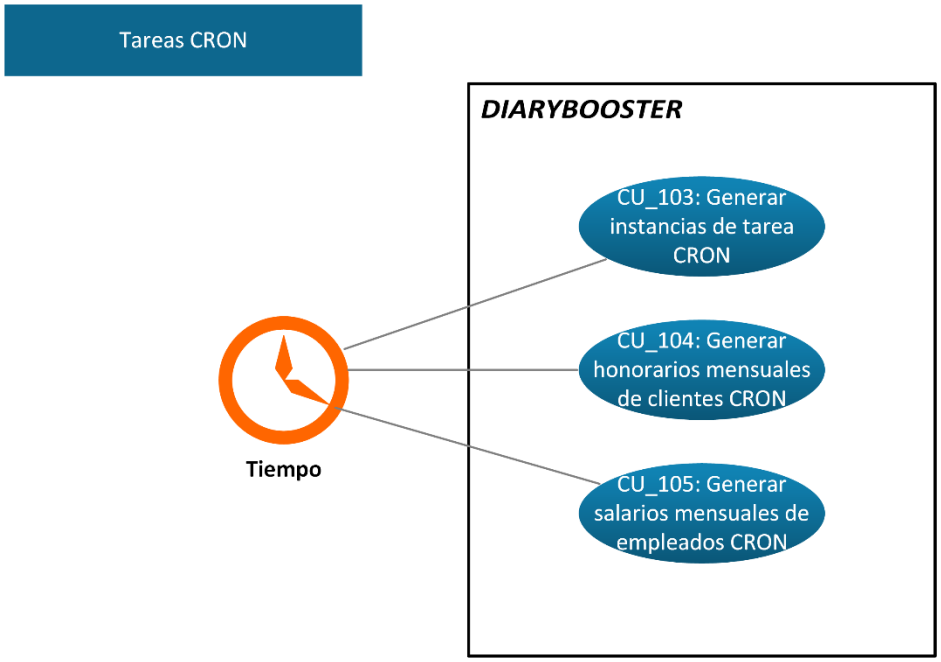


IMAGEN 024. Casos de uso que se ejecutan automáticamente cada período predeterminado de tiempo.

Casos de uso de gestión de usuarios

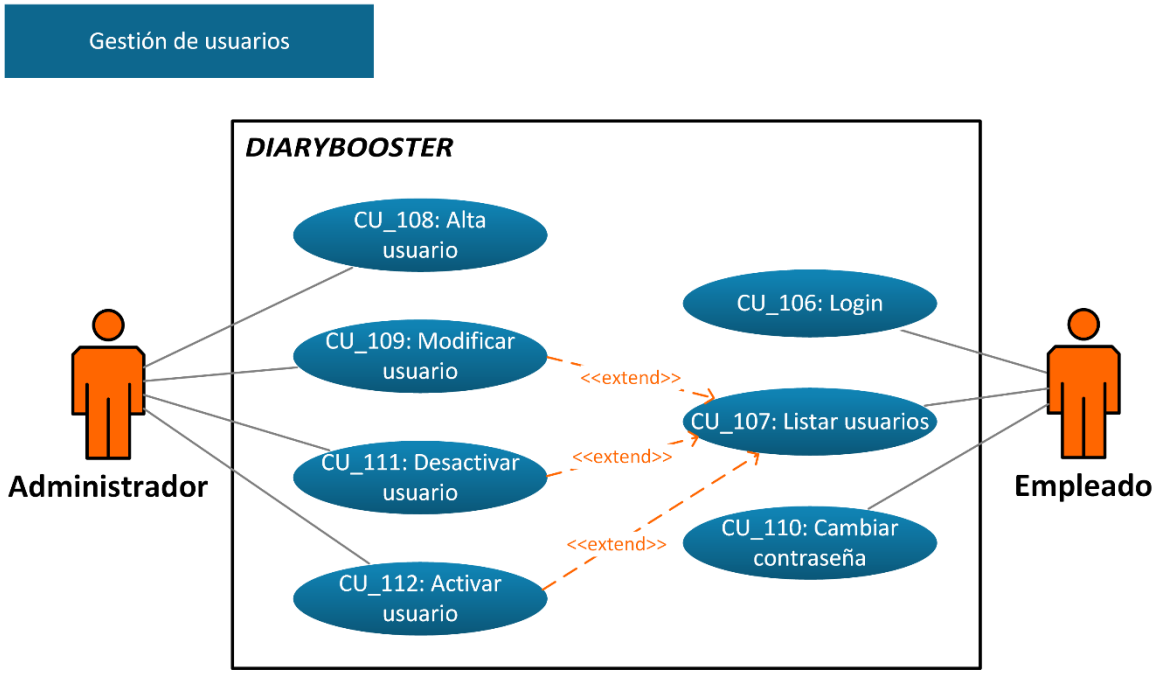


IMAGEN 025. Casos de uso que pertenecen al módulo de usuarios.

9. Modelo de Diseño

Para el desarrollo del proyecto, no confeccionamos un único Documento de Diseño que agrupe todos los modelos que detallan cuál es el diseño seguido en la construcción de DiaryBooster. En esta sección, detallaremos cuáles fueron las representaciones escogidas, lo que se buscó visibilizar con cada una y los resultados obtenidos. Se podrán observar modelos que representan tanto los aspectos estáticos como dinámicos del sistema.

El documento de arquitectura de software, que se explicó anteriormente, podría formar parte de un documento más amplio que contemple todos los aspectos del diseño de DiaryBooster. Por ello, lo tomaremos como referencia para argumentar o explicar algunas decisiones en los modelos que veremos a continuación, pero no entraremos en mayor nivel de detalle respecto a la arquitectura en sí misma.

Al mismo tiempo, existe otra decisión previa que condiciona de manera significativa el diseño del sistema: la elección del framework de desarrollo. Si bien se puede utilizar para construir otro tipo de soluciones, Symfony es una herramienta que surgió para construir aplicaciones web basadas en el modelo vista controlador. Como surgió en el documento de arquitectura, optamos por implementar una solución basada en este modelo de diseño, por los múltiples beneficios que encontramos a partir de los escenarios planteados.

Teniendo en cuenta estas consideraciones iniciales, procederemos a referir aquellas decisiones de diseño que nos parecen más relevantes.

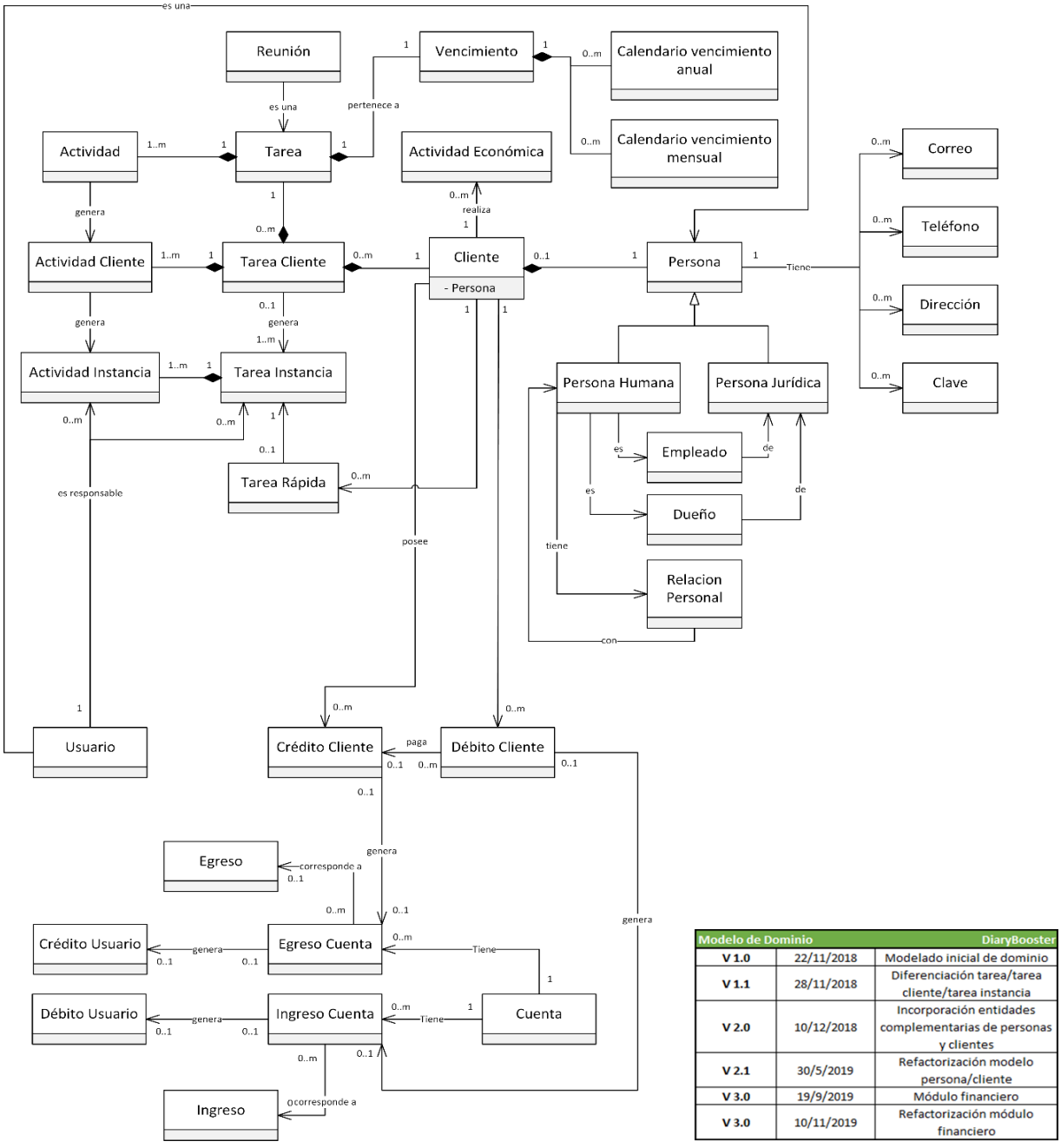
9.1 Modelo de dominio

Si bien el modelo de dominio forma parte de las etapas más tempranas del desarrollo de software, perteneciendo al análisis, lo introduciremos en esta parte del informe. Esto se debe a que, en nuestro proceso, ambos modelos estuvieron íntegramente relacionados, y se fueron retroalimentando el uno al otro.

El modelo de dominio no se construyó completamente desde el principio. Si bien ya contábamos con todas las definiciones semánticas y conceptuales por parte del cliente, decidimos hacerlo de esta manera para tener mayor flexibilidad al ir agregando los nuevos módulos en etapas posteriores. Al mismo tiempo, nos sirvió para ir repitiendo en los nuevos módulos las prácticas que encontramos beneficiosas de las implementaciones previas. Vale recalcar, que también nos sirvió como un primer acercamiento a un modelo de clases muy básico. Se pueden observar algunos detalles importantes: la herencia de las personas humanas y jurídicas, una relación directa entre una clase abstracta persona y una única entidad cliente (que puede ser humano o jurídico), el desglose de las tareas y actividades en tres niveles distintos (solución adoptada por el equipo de desarrollo, ya que el cliente siempre habló de tareas y actividades), entre otros pequeños aspectos que representan decisiones incipientes de diseño. Todo este proceso nos sirvió para validar el modelo con el cliente, presentándole los

beneficios y restricciones del mismo, y luego continuar profundizándolo para obtener el modelo de clases final, pero ya sobre una base más firme.

En la siguiente imagen, podemos visualizar la versión final del modelo de dominio.



Modelo de Dominio		DiaryBooster
V 1.0	22/11/2018	Modelado inicial de dominio
V 1.1	28/11/2018	Diferenciación tarea/tarea cliente/tarea instancia
V 2.0	10/12/2018	Incorporación entidades complementarias de personas y clientes
V 2.1	30/5/2019	Refactorización modelo persona/cliente
V 3.0	19/9/2019	Módulo financiero
V 3.0	10/11/2019	Refactorización módulo financiero

IMAGEN 026. Diagrama que representa el modelo de dominio del sistema. El mismo muestra las principales entidades y las relaciones existentes entre las mismas, a un alto nivel de abstracción.

Cada una de estas, se convirtieron en una clase del diagrama de clases, y en una entidad del Modelo.

9.2 Diagrama de clases

En base a lo detallado en el diagrama de dominio, generamos todas las clases que representaban objetos del modelo, sus atributos y las relaciones entre ellas. Es importante aclarar que, en el diagrama de clases, solamente detallamos las clases del Modelo que representaban entidades del mismo, casi en una relación uno a uno con las tablas de la base de datos.

Lo decidimos de esta forma dado que la mayoría de las otras clases del sistema se pueden entender e interpretar fácilmente con un pequeño conocimiento de Symfony. Por ello, consideramos que construir otros diagramas para las mismas, o inundar el diagrama actual con todas estas otras clases que componen tanto la Vista, como el Controlador e incluso otras partes del Modelo, solamente serviría para decorar la documentación del sistema y no reportaría un valor agregado al equipo de trabajo.

La estructura del framework, se basa mayoritariamente en 5 tipos de clases/archivos para construir un sistema web completamente funcional. Existen desde luego, otros objetos que permiten brindar un funcionamiento más refinado, o con algunas particularidades, pero siempre en torno a estos componentes principales:

- *Controlador*: Clases que implementan el controlador del modelo MVC. Utilizan el resto de las clases para disponibilizar al usuario, el comportamiento esperado del sistema.
- *Entidad*: Es un objeto del modelo. Se almacena en una tabla de base de datos y tiene relación con otras entidades. Son las clases documentadas en el diagrama.
- *Repositorio*: Clase que permite al controlador acceder a los datos almacenados en la base y transformarlos fácilmente en entidades.
- *FormType*: Clase de Symfony que permite construir y manipular fácilmente formularios en base a entidades del modelo.
- *Plantillas Twig*: Plantillas en las cuales se implementa la GUI. Si bien no es obligatorio el uso de Twig, es la opción predeterminada del framework, y la escogida por el equipo de trabajo. Están optimizadas para ser renderizadas fácilmente desde cualquier controlador y utilizar los formularios Symfony con muy poco código.

Como se puede observar, el componente sobre el cual estructuraremos los demás son las entidades del modelo. Por ejemplo, si tenemos una entidad que tiene un ABM en el sistema, seguramente contará con un EntityType, que modelará su formulario. Al mismo tiempo, la vista tendrá una carpeta para la entidad (o el módulo al que pertenece) con un formEntity.html.twig para el alta y la edición, y un indexEntity.html.twig para el listado. El repositorio podrá existir o no, dependiendo del nivel de complejidad de consultas que se realicen con el mismo. Doctrine construye un repositorio por defecto, y si no se necesita más funcionalidad que la que brinda esta clase inicial, no es siquiera necesario crear una clase EntityRepository para el mismo.

Los controladores no tienen una relación de uno a uno con las entidades (aunque el diseño se podría haber planteado de esa forma). La opción elegida fue crear clases controladores

relacionados a los distintos módulos, que contuvieran las acciones (funciones que implementan una funcionalidad) relacionadas con estos módulos.

Las clases controladores creadas fueron:

Controlador	*Módulo*
ClienteController	Cliente
PersonaController	Cliente
CuentaClienteController	Financiero
CuentaUsuarioController	Financiero
CuentaController	Financiero
IngresoEgresoController	Financiero
ReunionController	Agenda
TareaClienteController	Gestión Tareas
TareaController	Gestión Tareas
TareaInstanciaController	Gestión Tareas
VencimientoController	Gestión Tareas
UsuarioController	Usuario
ConfiguraciónController	Configuración
UtilidadesController	Utilidades
ValidaciónController	Utilidades

TABLA 008. Descripción de los distintos controladores que se crearon para implementar la lógica del sistema, divididos de acuerdo al módulo al que pertenecen.

En las siguientes páginas, presentaremos dos diagramas de clases. Ambos pertenecen a DiaryBooster, pero se han dividido en dos partes para facilitar la lectura y comprensión del diseño del sistema. En el primer diagrama, están las clases relacionadas a la gestión de tareas y clientes; en el segundo, se encuentran las clases relacionadas al módulo financiero. Existen algunas clases que figuran repetidas para poder ilustrar correctamente las relaciones existentes.

El orden en que presentamos los diagramas fue el orden en que fueron creados Se puede observar cuáles entidades se escogieron como base del sistema y cuáles se fueron acoplando, sumando funcionalidad y coherencia al modelo a medida que avanzaba el proyecto.

DIAGRAMA DE CLASES – MODULO GESTION

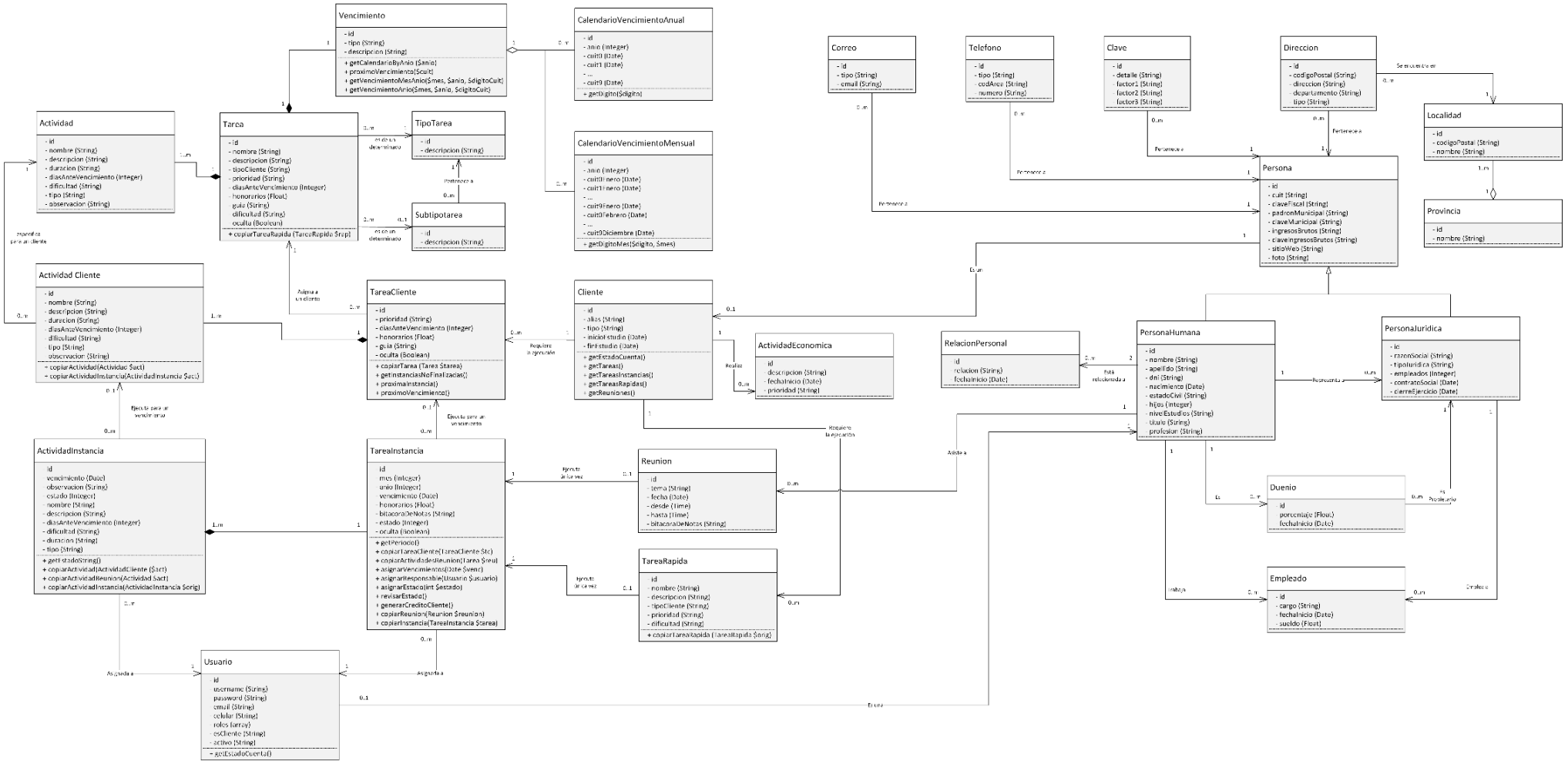
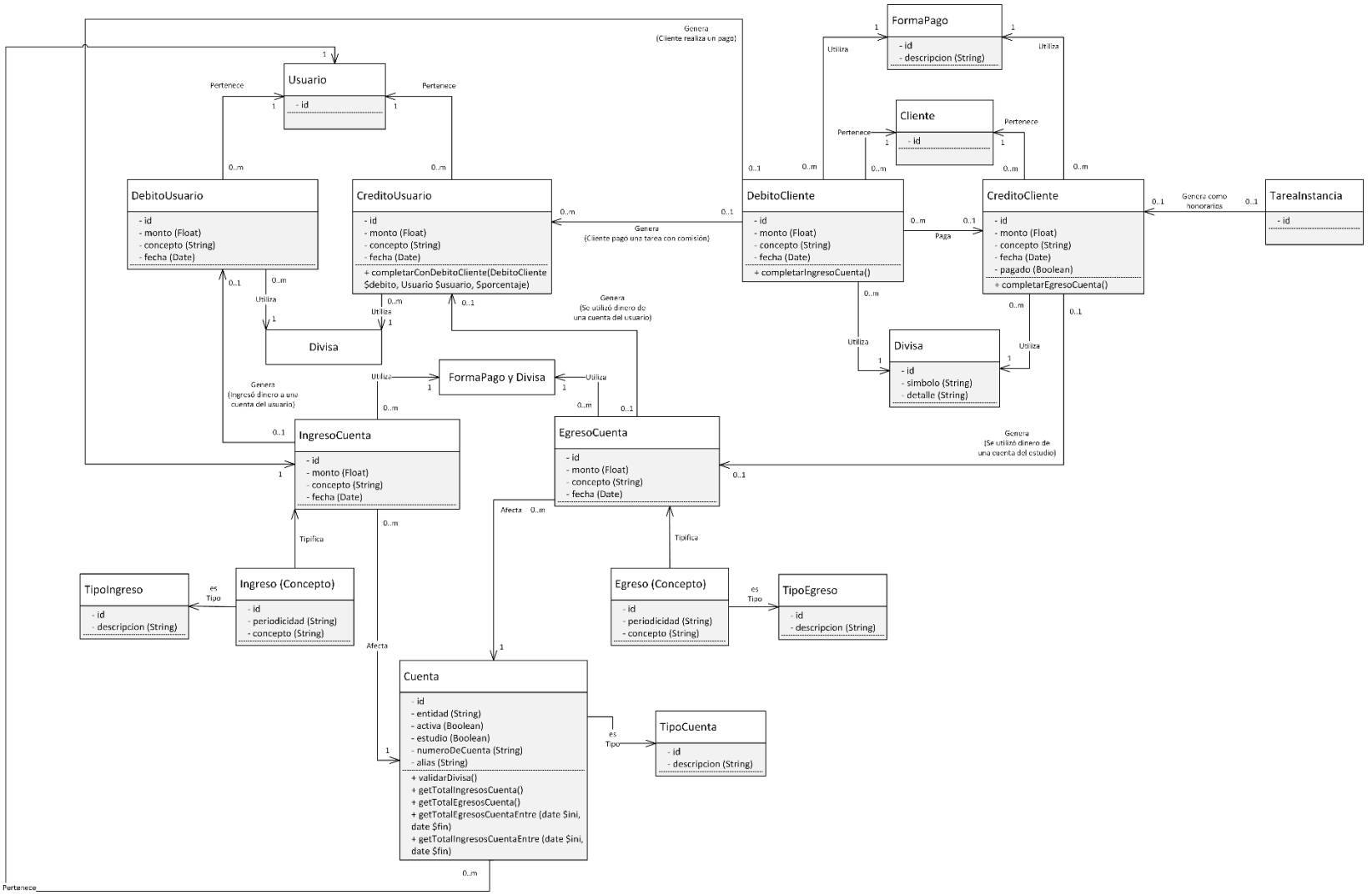


DIAGRAMA DE CLASES – MODULO FINANCIERO



9.3 Validación de entidades

En este apartado detallaremos qué metodología utilizamos para la declaración de entidades, cómo funciona y qué beneficios nos brinda a la hora de trabajar. También, indicaremos cómo han sido declarados los atributos de cada una de ellas en relación a sus respectivos tipos de datos.

Un aspecto importante a la hora de hablar de las entidades que componen el modelo es qué tipos de datos corresponden a cada atributo, cuáles son las condiciones que los vuelven válidos y aquellas que hacen que sean incorrectos. Sin embargo, no creamos un documento escrito para dejar registro de estas decisiones, debido al método utilizado para definir las clases de entidad y su relación con la base de datos. Doctrine ofrece una sintaxis particular, las anotaciones. Si queremos saber, por ejemplo, qué tipo de dato corresponde a un campo, o cuál es su longitud, simplemente abrimos la clase correspondiente a la entidad, buscamos el nombre del campo y obtendremos toda la información relacionada. Como todo el equipo de trabajo tenía acceso al código, no consideramos necesario escribir un documento aparte.

```php
/**
 * @var integer
 *
 * @ORM\Column(name="id", type="integer", nullable=false)
 * @ORM\Id
 * @ORM\GeneratedValue(strategy="IDENTITY")
 */
private $id;

/**
 * @var string
 *
 * @ORM\Column(name="nombre", type="string", length=255, nullable=false)
 */
private $nombre;

/**
 * @var string
 *
 * @ORM\Column(name="descripcion", type="string", length=150, nullable=true)
 */
private $descripcion;
```

IMAGEN 027. Ejemplo de las anotaciones utilizadas en el proyecto para definir los tipos de datos correspondientes a cada entidad, y cómo el mapeador objeto relacional sabrá a partir de las mismas cómo debe comunicarse con la base de datos.

Por otra parte, hemos documentado la validación de los formularios que se incluyeron en el sistema. Si bien está relacionada con la definición de los atributos que recién mencionamos, esta validación puede ser un poco más específica (por ejemplo, en cuanto a restricción de formato, u otras restricciones que pueden generarse con respecto al largo del campo, cuando se trata de un valor encriptado que ocupa más lugar en la base que el texto ingresado).

Estas validaciones se incluyeron en el documento de especificación de requerimientos, ya que es el que utilizamos para acordar con el cliente las funcionalidades específicas de cada caso de uso. Incluimos: tipo de campo, obligatoriedad, longitud

mínima y máxima (según correspondiera), y otras restricciones tales como: los caracteres aceptados, o un formato particular (hh:mm, por ejemplo). Por ello, cada vez que íbamos agregando un caso de uso que incluía un formulario, este se iba adicionando a la parte final del documento con todas las validaciones correspondientes, y se utilizaba para establecer los acuerdos alcanzados entre el equipo de trabajo y el cliente, dejando claro para todos el funcionamiento de cada campo.

9.4 Diagramas de secuencia

Los modelos detallados hasta ahora son empleados, en su gran mayoría, para brindar una perspectiva estática de la estructura del sistema. Sin embargo, elegir únicamente este tipo de visión nos impide comprender correctamente cuál es su funcionamiento, cómo debería comportarse, cómo interactúan cada uno de sus componentes, entre otros aspectos relevantes.

Para poder representar los aspectos dinámicos de DiaryBooster, decidimos emplear diagramas de secuencia, que nos permiten observar la interacción entre los distintos componentes mediante el intercambio de mensajes. Partiendo del hecho de que utilizamos un paradigma orientado a objetos (mayoritariamente), entendemos que estos diagramas son especialmente relevantes para modelar nuestro sistema. En estos veremos las clases incluidas en el diagrama de clases, como también los otros tipos de clases que fuimos detallando: controladores, formularios, plantillas, etc.

Además, no hemos construido un diagrama de secuencia por cada caso de uso. La utilización del framework de desarrollo escogido y los estándares de comportamiento establecidos por el equipo de trabajo nos permitirán mostrar en pocos diagramas de secuencia el comportamiento general del sistema. Luego de observar el gráfico, podremos aplicarlo a la gran mayoría de casos de uso detallados, reemplazando simplemente a los actores genéricos que se utilizaron en el diagrama, por los específicos de cada caso de uso (por ejemplo, para el CU 001 – Listar clientes, reemplazando el controller por el ClienteController).

9.5 Funcionamiento normal de un formulario

En la siguiente imagen, se puede observar el diagrama de secuencia que modela el normal funcionamiento de un formulario. Los elementos involucrados son: usuario, que interactúa con el sistema mediante la interfaz mostrada en su navegador web y controlador, encargado de procesar todas las peticiones del usuario, implementar la lógica esperada del caso de uso, y llamar a los componentes del modelo y vista necesarios para retornar la respuesta al usuario. Junto con ellos podemos encontrar el Form Type, que es la clase que gestiona el formulario correspondiente a la entidad; el Entity Manager, que es la clase de Doctrine que se conecta con la base de datos y persiste las entidades; y Twig, que es el sistema de renderizado de plantillas cuya tarea es generar las vistas para presentar al usuario.

Cada uno de estos elementos abstractos se reemplazará por el correspondiente de acuerdo al caso de uso a implementar. En el caso del alta de un cliente humano, se tratará del controlador ClienteController, el form ClienteHumanoType, y la plantilla clienteHumanoForm.html.twig.

Otra particularidad de este escenario es que todas las peticiones se realizan mediante peticiones http directamente desde el navegador e implican redirecciones. En este caso no se realizan peticiones Ajax al backend para el procesamiento del formulario, por lo tanto, la lógica sigue esta secuencia de procesamiento, con peticiones y respuestas directas al navegador, y una única interacción con el usuario. Por otro lado, siempre la respuesta consiste en un sitio html completo.

Por último, no se muestra al usuario completando el formulario. Esto se decidió para no agregar mayor complejidad a este diagrama, considerando que luego de la respuesta del First Request, este escenario vuelve a comenzar, luego de que el usuario complete el formulario y pulse el botón “guardar”. Esto refleja de forma más representativa el comportamiento del controlador, no la interacción real, considerando que facilita la construcción de los controladores de creación/edición.

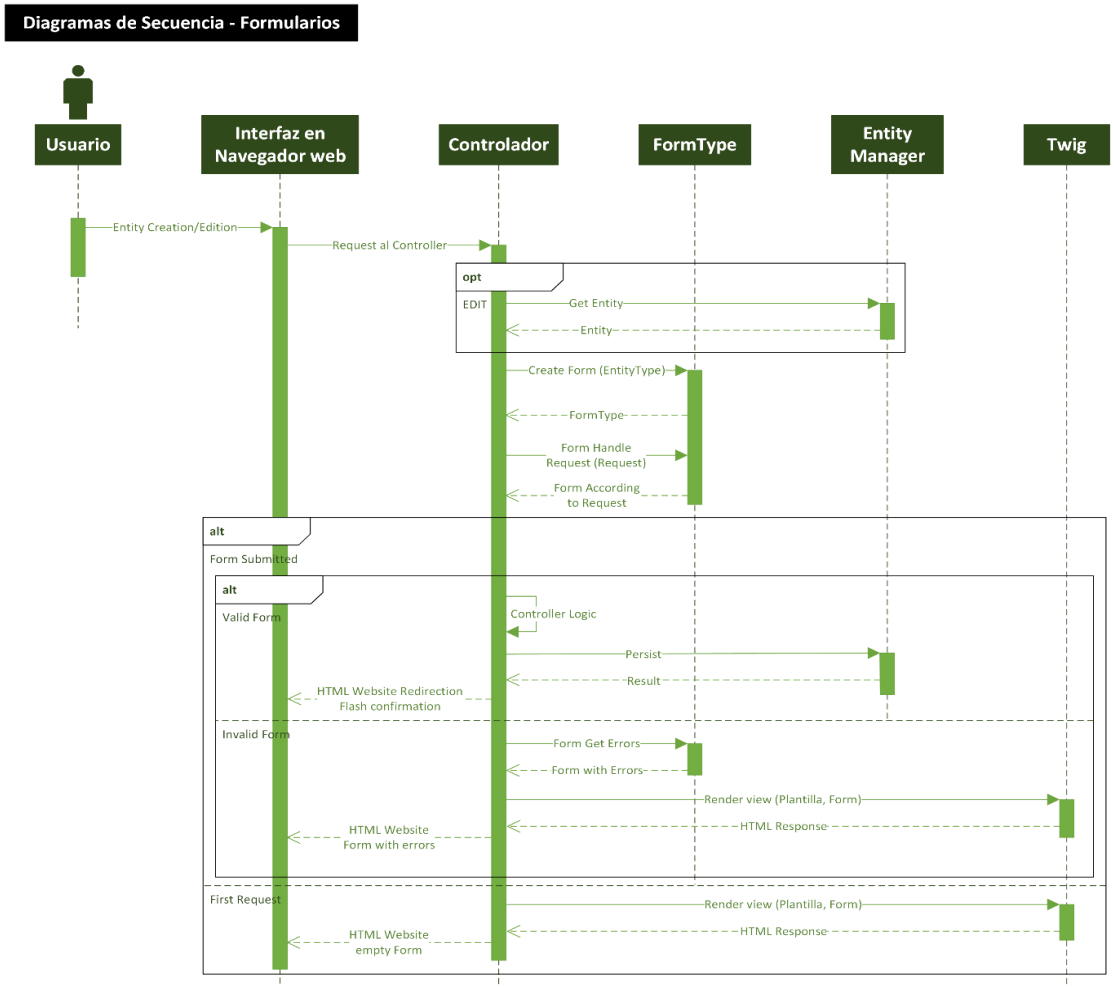


IMAGEN 028. Diagrama de secuencia que describe el comportamiento genérico que presentan los formularios del sistema.

9.6 Funcionamiento de un formulario embebido en un modal

El segundo diagrama también presenta un formulario, pero en un escenario distinto al primero. En este caso, este no se manipula mediante redirecciones http al backend, sino que se embebe en un modal Bootstrap dentro de otra pantalla (generalmente un listado), y se trabaja mediante peticiones Ajax al backend. Estas peticiones también generan respuestas HTML, pero no devolverán un sitio web completo que se mostrará en el navegador, sino que es una porción de texto HTML que reemplazará el contenido actual del modal. Por esta razón, el procesamiento del formulario se estructuró de otra forma. Se mostró en diferentes instancias la petición inicial del formulario y el tratamiento luego de que el usuario lo completa y envía nuevamente al backend. En este caso, hay mucho más trabajo del lado del cliente, que debe procesar las peticiones del usuario, construir la ruta, el payload necesario y hacer la petición al backend para reemplazar la respuesta en el modal al recibirla.

En caso que el formulario sea válido, se redirigirá al usuario a otra pantalla actualizando la información recientemente ingresada.

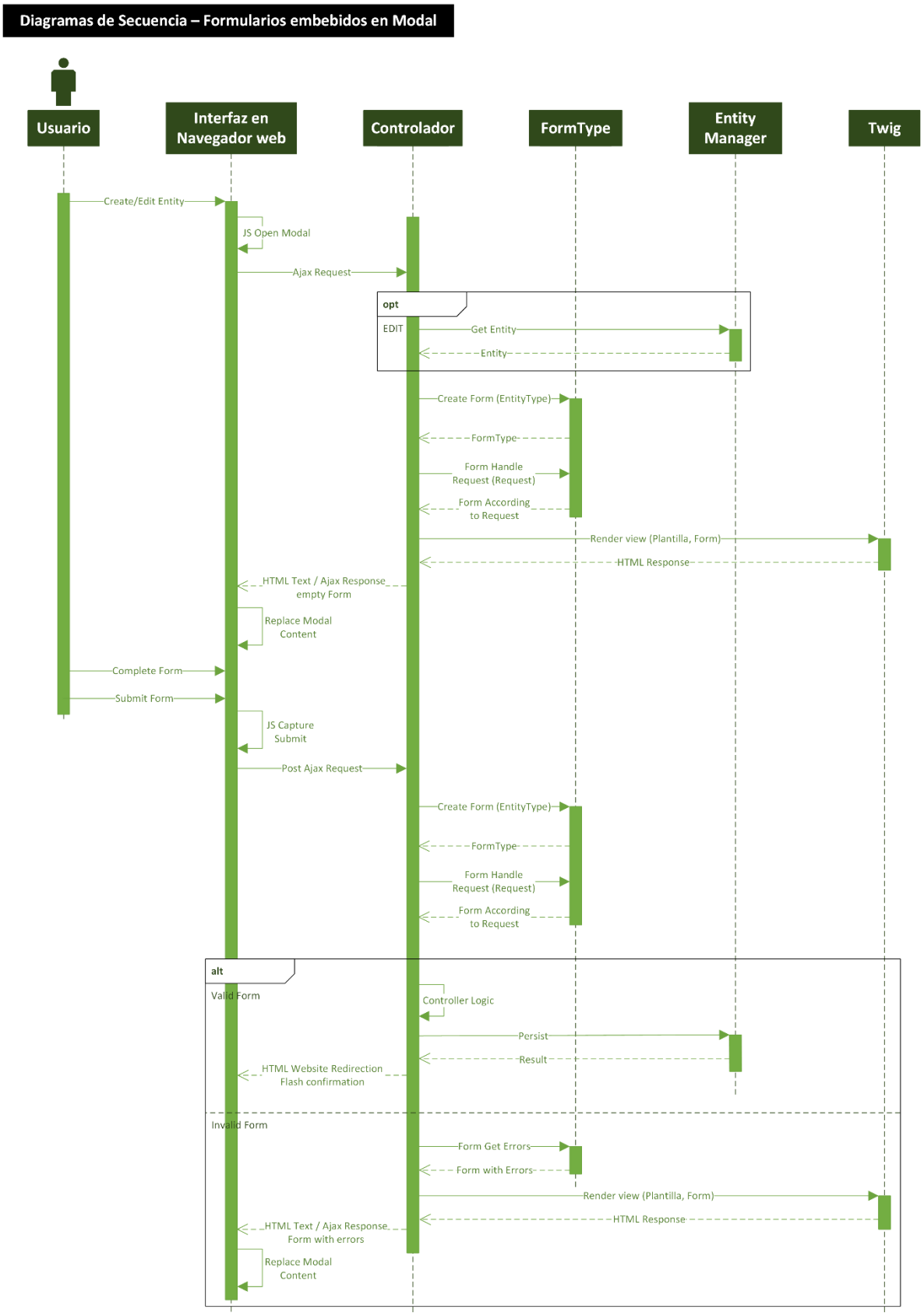


IMAGEN 029. Diagrama de secuencia que describe el comportamiento genérico que presentan los formularios del sistema que se encuentran embebidos en un modal.

9.7 Funcionamiento de un listado

Otro caso particular diagramado es el de los listados. Ha sido modelado de manera estandarizada para que se siguiera el mismo patrón a lo largo de todo el sistema.

Existen algunos casos particulares que modifican un poco este flujo, como algún formulario que pueda existir dentro de un modal, o un listado que incluya filtros mediante un formulario Symfony. Sin embargo, al no ser comportamientos tan extendidos sino más bien de casos puntuales, aplicamos estas pequeñas modificaciones únicamente a la hora de generar el código.

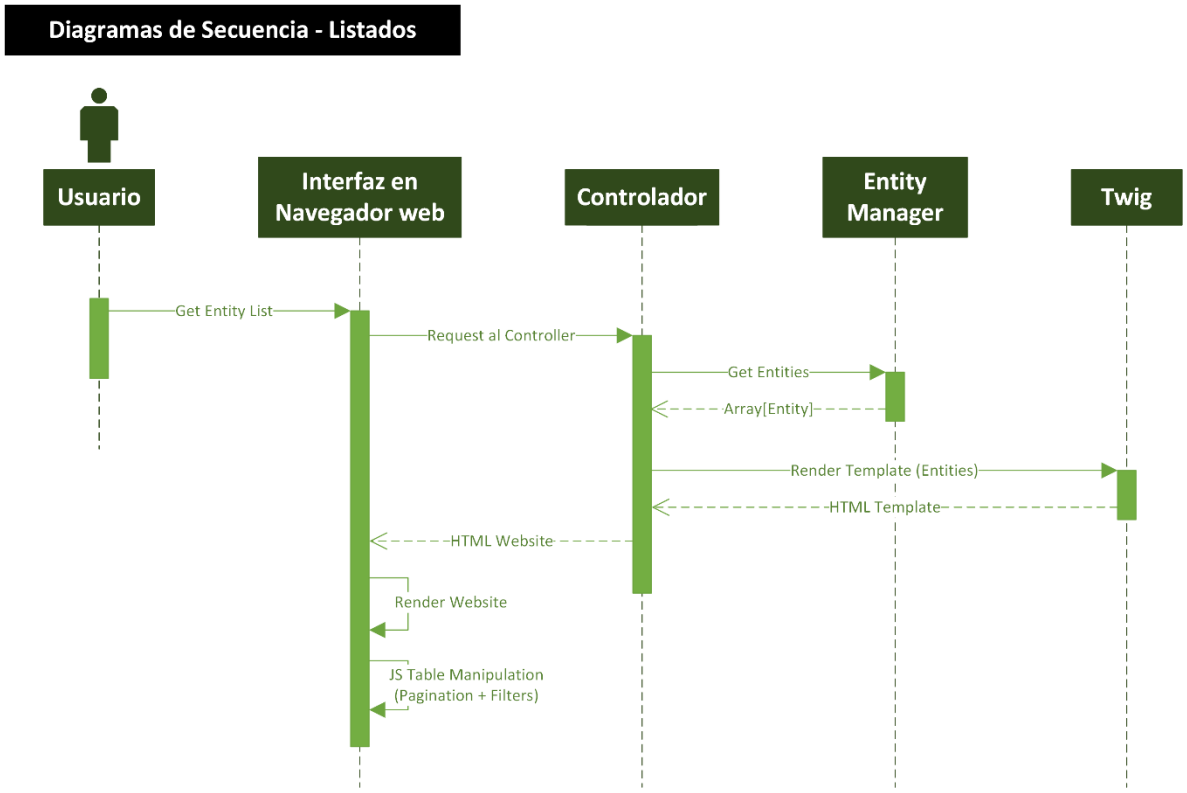


IMAGEN 030. Diagrama de secuencia que describe el comportamiento genérico que presentan los listados de las diferentes entidades principales del sistema.

9.8 Funcionamiento de los procesos programados

El último diagrama de secuencia que construimos fue el que corresponde a la ejecución de tareas programadas o CRONs. Estas tareas son comandos que se han configurado para ejecutarse a intervalos regulares de tiempo.

El sistema cuenta con tres comandos Symfony que fueron construidos para ejecutarse de esta forma, cada uno con un objetivo específico. Todos son muy importantes para el correcto funcionamiento del sistema, e spor esto que cualquier alerta generada por dichas tareas envía un mensaje a nuestros correos para estar al tanto de cualquier inconveniente. De esta manera, podemos verificar que se hayan ejecutado correctamente, y en caso contrario, corregir el error y asegurarnos de que corran de forma adecuada.

El primer comando es ejecutado diariamente por el web server, y consiste en la creación de las instancias de tareas, en base a las tareas genéricas que se han creado en el sistema, y a las asignaciones que se ha hecho a cada cliente. Una configuración global del sistema es la cantidad de días antes de que caduque la primera actividad de una tarea, es por esto que debe generarse la instancia que la contiene. Por lo tanto, en base a los calendarios de vencimientos cargados en el sistema por los usuarios de acuerdo a lo dispuesto por AFIP, y a esta configuración, se determina diariamente qué instancias de tarea deben crearse, y cuáles esperarán. Así, cada día los usuarios verán en sus bandejas de entrada las tareas y actividades que deban recibir tratamiento.

Los otros dos consisten en la generación mensual de honorarios a cobrar a cada cliente por los servicios mensuales prestados y débitos que se adeudan a cada empleado en concepto de sueldo. Cada uno de estos parámetros tiene una sección específica en la configuración del sistema, y se almacenan en un registro de la Entidad Parámetro.

A continuación, detallaremos un nivel de comportamiento general de las tareas CRON:

Diagramas de Secuencia - CRON

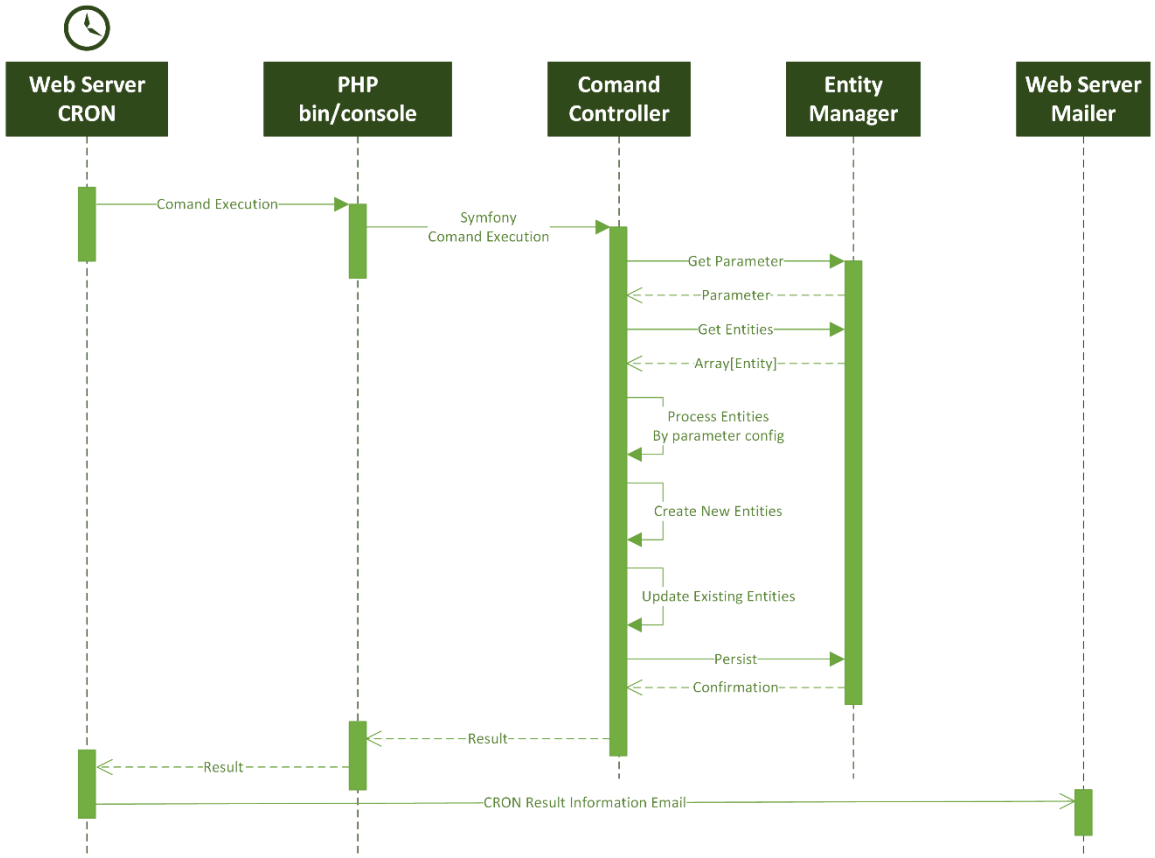


IMAGEN 031. Diagrama de secuencia que describe el funcionamiento de las tareas programadas.

9.9 Diagrama de estados Tarea Instancia

Dentro de las entidades básicas del sistema, además de los clientes, podemos encontrar las tareas y actividades. En conjunto, representan los trabajos que el personal del estudio debe realizar para cada uno de los clientes.

Existen diversos trabajos que se realizan cotidianamente en un estudio contable. Estos trabajos se repiten cada cierto período de tiempo (anual o mensual, o incluso algunos de única vez), y siempre tienen un flujo de subtareas (las actividades) similar. Generalmente, cuentan con un calendario de vencimientos dispuestos por AFIP, basados en el último dígito del CUIT. Estas descripciones generales acerca de los quehaceres del estudio, es lo que se constituyó en el sistema como una tarea. Entonces, podemos definir a una entidad Tarea como un marco de trabajo, compuesto por actividades, un tipo de vencimiento y sus calendarios, que se asignará a los clientes para repetirse a lo largo del tiempo, relegando esta repetición en la capacidad del sistema de generar instancias de tarea automáticamente. Como mencionamos, cada una de estas tareas se asignará a uno o más clientes. Esta asignación genera una nueva entidad: Una Tarea Cliente, compuesta por Actividades Clientes. En sí, son una réplica de la descripción de sus respectivas entidades padre. Sin embargo, es importante para los usuarios contar con la posibilidad de especificar las tareas que tienen que realizar para cada cliente. Por ejemplo, podría existir un cliente con dos actividades extras que no aparecen en ningún otro, o una modificación de honorarios, con un vencimiento previo (porque en la práctica los empleados han comprobado que se demora más en presentar la documentación pertinente), etcétera. Es por ello que, cuando se realiza una asignación, la Tarea se replica en una Tarea Cliente independiente de las demás, que permite personalizar completamente los trabajos a realizar para cada cliente.

En último lugar, contamos además con la clase Tarea Instancia. Esta clase corresponde al trabajo real efectuado por el estudio en beneficio de un cliente. Supongamos una liquidación de IVA, que debe realizarse de manera mensual, acorde al calendario de AFIP. La tarea “Liquidación de IVA” establece el marco general, cómo se realiza esta tarea para todos los clientes del estudio. Por otro lado, la asignación de Liquidación de IVA para el cliente X, explicita cómo debe realizarse la misma para este cliente particular. Y finalmente, la instancia corresponderá a la liquidación de IVA realizada para X en el mes de Marzo 2020, con vencimiento el día 15, efectuada por el empleado Y, y que ya se encuentra en estado finalizada. Así, para una tarea mensual, mes a mes se generarán nuevas instancias acordes a cada Tarea Cliente existente.

Con estas consideraciones, podemos concluir que tanto las entidades Tarea como Tarea Cliente, son especificaciones de trabajo, una a nivel general, y la otra a nivel de cliente. Por último, la clase Tarea Instancia representa una ocurrencia específica del conjunto de actividades, para un cliente y un lapso de tiempo particular. Las Tareas Instancias que surgen a partir de Tareas Clientes anuales o mensuales, serán generadas por el sistema de manera automática.

Ya explicada la diferencia entre estas tres entidades (y sus componentes, actividades, actividades cliente, y actividades instancia), podemos proceder a explicar el

siguiente gráfico.Como ya fue mencionado anteriormente, el trabajo realizado realmente está expresado a partir de las instancias de tarea y de actividad. Por ende, lo que el usuario visualizará en sus bandejas de entrada, serán estas instancias. Pero ¿cómo sabe cuándo un trabajo está pendiente, o ya está finalizado? ¿Cómo puede llevar registro de ello? Para eso, se introduce en ambas clases el concepto de estado. Los valores posibles para el estado son los mismos tanto para la tarea como para la actividad instancia: No Iniciada, En Proceso, Esperando Cliente, y Finalizada. La principal diferencia entre la tarea y la actividad instancia es que el estado de la actividad se setea de forma manual y solo depende de esta asignación. Sin embargo, en el caso de una tarea instancia, el estado puede setearse manualmente y modificar el estado de la tarea y todas sus actividades o puede cambiar en base al estado de las actividades que la componen. En el primer caso, siempre que se setee un estado a una tarea manualmente, se pasarán todas las actividades no finalizadas al estado seleccionado, pudiendo darla finalizada completamente, o pasarla a otro estado. Por otro lado, cuando no se modifica el estado de la tarea en sí misma, sino el estado de sus actividades, es necesario realizar una comprobación para asegurarse que no haya cambiado el estado global, o actualizarlo si correspondiera.

Diagrama de estado – Tarea Instancia

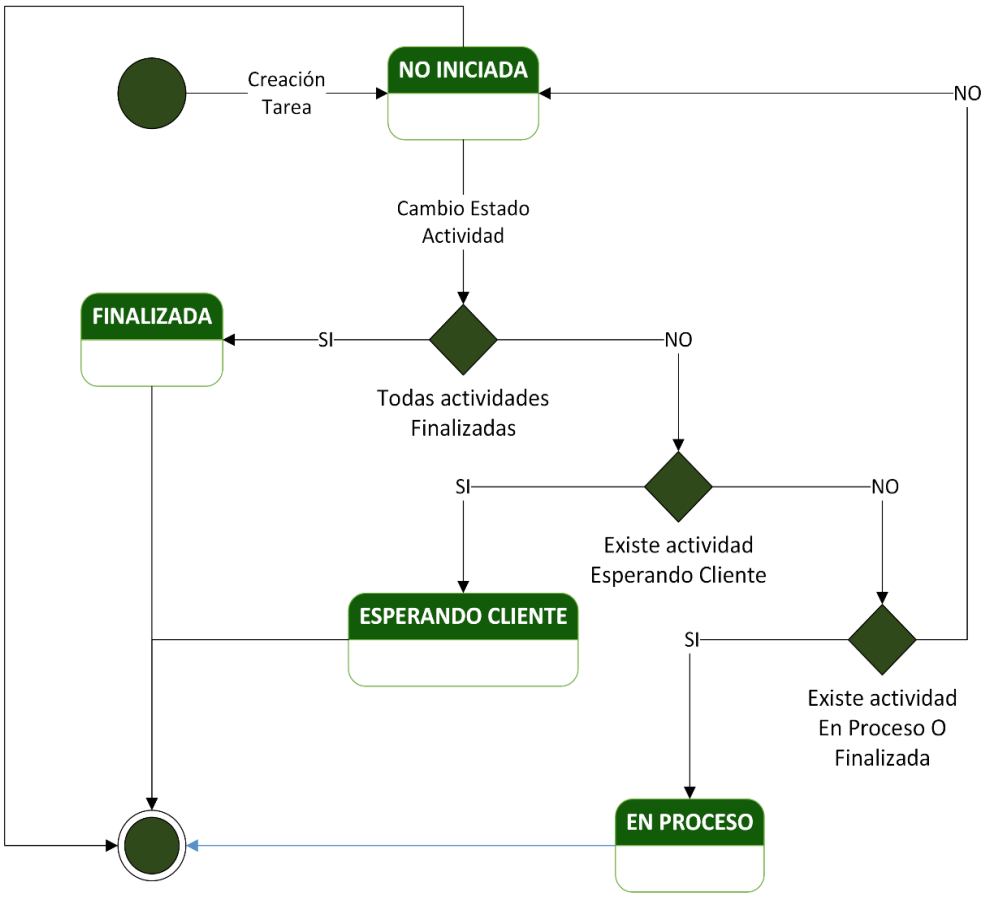


IMAGEN 032. Diagrama de estados de la entidad tarea instancia. La finalidad de este diagrama es describir cuál será el estado resultante de una tarea, al modificar el estado de una o más de sus actividades instancia.

El diagrama de estados tarea instancia se construyó para clarificar cómo debe funcionar esta comprobación, y qué estado corresponderá a la tarea en base al estado de sus actividades. Esto se implementa en el caso de uso CU 057: Verificar estado de tarea instancia. Si bien en el código no se implementó de la misma forma que se observa en el diagrama por cuestiones de performance y buenas prácticas, se obtuvo el mismo comportamiento que el descrito en el gráfico.

10. Seguridad

Uno de los requerimientos no funcionales más importantes de este proyecto corresponde a la seguridad. Esto no solamente concierne a una solicitud particular del cliente, limitando el acceso a su información por parte de terceros, sino que también responde a la naturaleza de la información que será almacenada en el sistema.

Si bien, como ya se ha mencionado previamente, no se almacenará información contable en el sistema, sino administrativa, existe información personal de los clientes necesaria para la gestión diaria de sus tareas por parte de los empleados del estudio contable. Además, uno de los requerimientos del sistema es poder almacenar claves relacionadas a la entidad cliente (por ejemplo, una clave de AFIP), por lo cual esta información requiere un tratamiento especial para garantizar que no sea utilizada de forma indebida por personas ajenas al estudio.

Para poder resolver de forma adecuada este requerimiento del sistema, a la hora de plantear la arquitectura del mismo, se ha abordado la seguridad como un atributo de calidad a alcanzar. Esto se encuentra reflejado en los escenarios planteados en el Documento de arquitectura de software, que forma parte de los anexos de este trabajo.

Presentamos a continuación un resumen de las principales medidas de seguridad que se han seleccionado para la implementación del proyecto:

- ***Navegación HTTPS***. Es la medida básica de seguridad. Si bien para los entornos locales de cada desarrollador no utilizamos certificados SSL, el sitio web productivo solo puede ser accedido mediante el protocolo HTTPS. Al intentar acceder mediante HTTP, el usuario será redirigido automáticamente.
- ***Autenticación de usuarios***. No existe ninguna pantalla del sistema que pueda ser accedida de forma anónima (más allá del Login). Cualquier intento de acceso a una funcionalidad específica por parte de un usuario anónimo, será redirigido al Login.
- ***Utilización de Symfony como framework de desarrollo***. La utilización de un framework de amplio uso a través de toda internet, en conjunto con una gran comunidad de soporte y mantenimiento, da como resultado que cualquier vulnerabilidad que pueda existir en la capa de seguridad del mismo sea detectada y corregida rápidamente, debido a que no formará parte exclusiva de nuestro proyecto, sino de toda la comunidad de usuarios.
- ***Algoritmo de cifrado BCrypt y condiciones mínimas para contraseñas***. A la hora de dar de alta un usuario, es necesario establecer una contraseña con al menos 8 caracteres, una mayúscula, una minúscula y un número. Por otro lado, al utilizar un campo *username* para acceder al sistema, solamente el administrador que lo dio de alta y el empleado dueño de dicho usuario lo conocerán y podrán utilizarlo para acceder (no se utiliza una dirección de correo, por ejemplo, que es un dato de público conocimiento).

  Por otra parte, se utiliza el algoritmo de cifrado *BCrypt* con un número alto de saltos para las contraseñas de los usuarios.

- ***Hosting web de calidad y trayectoria en el mercado***. Para alojar la solución desarrollada, se ha escogido el hosting *WebEmpresa*. El mismo cuenta con más de 20 años de trayectoria, y una gran base de buenas referencias a lo largo de la comunidad de usuarios de internet.

  Para su selección se tuvo en cuenta la facilidad de acceso a soporte que brinda, y la buena reputación con la que cuenta. Provee servicios de configuración automatizados (como el certificado de SSL), y revisiones periódicas respecto a las configuraciones de seguridad de cada sitio que aloja. También herramientas automatizadas para verificar la configuración de seguridad del sitio.
- ***Jerarquía de roles de usuario***. Para acceder a las diferentes funcionalidades del sistema, se ha creado una jerarquía de roles, que solo permite determinadas acciones a un usuario de tipo administrador. Por ejemplo, solamente el administrador puede acceder a las configuraciones financieras, o al estado de cuenta de todos los empleados. Un usuario común solo podrá ver el estado de su cuenta, pero estará imposibilitado de ver el de sus compañeros.
- ***Encriptación de campos sensibles en la base datos***. Para todos los datos referidos a claves de los clientes, se ha procedido a utilizar un bundle específico para *Symfony* que permite encriptar/desencriptar campos determinados configurados previamente antes de que *Doctrine* los recupere de la base de datos. De esta forma, en caso de que la seguridad del hosting web sea vulnerada, y terceros accedieran a nuestra base de datos, no contarían con la información necesaria para poder ver dicha información, ya que esta se encuentra encriptada.
  Esto también es útil para que, en caso de que los administradores del sistema debieran acceder a la base de datos para solucionar algún problema, no se encontraran con dicha información en un formato legible para ellos.

11. Tecnologías:

A continuación, explicaremos las tecnologías utilizadas en el desarrollo del proyecto. Estas tecnologías permitieron que el completo desarrollo del sistema se realice de manera eficaz y ordenada, lo que condujo a resultados más claros y ordenados para futuras modificaciones.

Inicialmente, no teníamos definidos todas los frameworks a utilizar para el desarrollo de la plataforma DiaryBooster. Basándonos en el conocimiento previo y en el análisis efectuado, decidimos optar por las siguientes herramientas:

11.1 Base de datos:

PhpMyAdmin es un software de código abierto escrito en PHP, diseñado para manejar la administración y gestión de bases de datos *MySQL* a través de una interfaz gráfica de usuario.

11.2 Desarrollo Backend:

Symfony es un *framework* diseñado para desarrollar aplicaciones web basadas en el patrón Modelo-Vista-Controlador. Sus ventajas radican en que separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web. Proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja. Además, automatiza las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación. Está desarrollado completamente en PHP, es compatible con la mayoría de gestores de bases de datos, como *MySQL*, *PostgreSQL*, *Oracle* y *Microsoft SQL Server*. Se puede ejecutar tanto en plataformas *nix (Unix, Linux, etc.) como en plataformas *Windows*.

11.3 Desarrollo FrontEnd:

Twig es un motor de plantillas moderno para PHP. Posee las siguientes características:

Rapidez: compila las plantillas hasta código PHP regular optimizado. El costo general en comparación con código PHP regular se ha reducido al mínimo.

Seguridad: tiene un modo de recinto de seguridad para evaluar el código de plantilla que no es confiable. Esto te permite utilizar *Twig* como un lenguaje de plantillas para aplicaciones donde los usuarios pueden modificar el diseño de la plantilla.

Flexibilidad: es alimentado por flexibles analizadores léxicos y sintácticos. Esto permite al desarrollador definir sus propias etiquetas y filtros personalizados, y crear su propio DSL.

Bootstrap es un conjunto de herramientas de código abierto para desarrollar proyectos responsive (adaptado a cualquier dispositivo) con HTML, CSS y *JavaScript*.

HTML5 es un lenguaje de marcado de hipertexto (Hyper Text Markup Language) usado para estructurar y presentar el contenido para la web. Funciona a través de etiquetas y es la herramienta fundamental que permite a los navegadores interpretar el código y permitirnos ver imágenes, párrafos, estructuras, etc.

CSS (Cascading Style Sheets) es un lenguaje de diseño gráfico que posibilita definir y crear la presentación de un documento estructurado escrito en un lenguaje de marcado. Es muy usado para establecer el diseño visual de los documentos web e interfaces de usuario escritas en HTML.

JavaScript es un lenguaje de programación interpretado. Se utiliza principalmente del lado del cliente para crear efectos atractivos y dinámicos en las páginas web. Los navegadores modernos interpretan el código JavaScript integrado en las páginas web. También se puede utilizar en el lado del servidor (por ejemplo, con *NojeJs*).

JQuery es una librería de *JavaScript*. Esta librería es de código abierto, simplifica la tarea de programar en *JavaScript;* además, permite agregar interactividad a un sitio web sin tener conocimientos del lenguaje. A nivel mundial, existe una cantidad de *plugins* creados para desarrolladores con el fin de resolver situaciones concretas dentro del maquetado de un sitio. Para que puedan funcionar, necesitan la importación de *JQuery* en el proyecto en cuestión.

11.4 Versionado:

GitHub es un sistema de gestión de proyectos y control de versiones de código, así como una plataforma de red social diseñada para desarrolladores. Permite trabajar en colaboración con otras personas de todo el mundo, planificar proyectos y realizar un seguimiento del trabajo.

GitHub es también uno de los repositorios *online* más grandes de trabajo colaborativo en todo el mundo.

Git es un sistema de control de versiones distribuido gratuito y de código abierto diseñado para manejar todo tipo de pryectos, desde proyectos pequeños hasta muy grandes, con velocidad y eficiencia. Incluye las funcionalidades de crear ramas y fusiones, y reescribir historiales de repositorios, lo cual ha dado como resultado muchas herramientas y flujos de trabajo innovadores y eficaces. Las solicitudes de incorporación de cambios son una de esas populares herramientas con las que los equipos pueden colaborar en las ramas de *Git* y revisar con eficacia el código de los demás. *Git* es el sistema de control de versiones más utilizado en el mundo hoy en día y se le considera el estándar actual de desarrollo de software.

11.5 Sistema operativo

Ubuntu es un sistema operativo de *software* libre y código abierto. La razón principal de su elección fue la facilidad de uso para desarrolladores y porque la fluidez del uso de aplicaciones es óptima.

11.6 Entorno de desarrollo

Visual Studio Code es un editor de código fuente desarrollado por Microsoft para *Windows*, *Linux* y *macOS*. Es compatible con varios lenguajes de programación tales como C#, *Java*, *JavaScript*, SQL, PHP, HTML, CSS, *React*, etc.

NetBeans es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación *Java*. Existe además

un número importante de módulos para extenderlo. *NetBeans* IDE es un producto libre y gratuito sin restricciones de uso.

11.7 Testing:

Selenium WebDriver es una suite de herramientas y librerías de libre distribución que permiten soportar la automatización de navegadores web. Provee extensiones para emular la interacción de usuarios reales con los navegadores. Esta característica es la que se utilizó para poder implementar el conjunto de pruebas funcionales sobre DiaryBooster. Es un proyecto abierto, desarrollado por numerosos colaboradores de todo el mundo. Hasta el presente continúa actualizándose e incorporando mejoras constantemente. Provee drivers para distintos navegadores, siendo los más desarrollados para *Chrome* y *Mozilla Firefox* (dado que son los más extendidos globalmente). En el proyecto utilizamos *Google Chrome* como navegador para la implementación de las pruebas. Al mismo tiempo, el proyecto está desarrollado para ser multiplataforma, y también para implementarse a través de múltiples lenguajes de programación: *Java*, *Python*, C#, *Ruby*, *JavaScript* y *Kotlin.*

En este proyecto, utilizamos ***Java 11*** para trabajar. Esta selección se basó en los conocimientos previos que poseíamos con este lenguaje de programación. Para trabajar con *Java*, seleccionamos el IDE *IntelliJ Idea*, de la compañía *JetBrains*. El mismo, requiere la compra de una licencia, pero también tiene la opción de un período de prueba, el cual aprovechamos en este caso. Transcurrido dicho período, accedimos a la versión comunitaria, que nos continuó dando soporte para la mayoría de herramientas necesarias. Esta selección se debió en parte a la suite de herramientas utilizadas en nuestra principal fuente de formación en automatización de pruebas con *Selenium*. Sin embargo, también influyó la facilidad de uso de la herramienta, la optimización de la misma para trabajar con *Java*, y la calidad de los productos desarrollados por *JetBrains*, que incluyen una variedad de detalles que facilitan enormemente la tarea del programador.

Uno de estos detalles es la integración nativa del IDE con *Maven*, una herramienta de gestión y construcción de proyectos *Java*, que permite automatizar distintos procesos del mismo. El aspecto que más nos interesaba del trabajo con *Maven* era la fácil gestión de dependencias e importación de paquetes.

12. Estrategia de pruebas

Este capítulo presenta las principales decisiones tomadas por el equipo respecto a la estrategia considerada para llevar adelante las pruebas del sistema. Se especifican: el tipo de pruebas construidas, el alcance de las mismas, el patrón de diseño utilizado y la implementación de la suite de pruebas funcionales automatizadas.

Una de las primeras decisiones respecto a la estrategia de pruebas surgió luego de decidir la metodología de desarrollo a utilizar. Como mencionamos anteriormente, no nos basamos en una implementación estricta del Proceso Unificado de Desarrollo, sino que intentamos adaptarlo incorporando determinadas prácticas derivadas de las metodologías ágiles. Un aspecto que analizamos detenidamente para incorporar al proyecto fue el desarrollo basado en pruebas (TDD). Consideramos que es una buena práctica, y que podría resultar de utilidad para el proyecto. Sin embargo, desistimos de su incorporación especialmente a raíz de dos características del proyecto: la falta de experiencia del equipo para trabajar con este tipo de práctica y la escasez de recursos humanos para la envergadura del proyecto. Probablemente si hubiéramos tenido algún integrante con experiencia en QA hubiéramos podido incluir TDD en la metodología de desarrollo, y nos hubiera reportado gran valor. Siendo otra la situación, desistimos de su incorporación.

En instancias siguientes, las decisiones a tomar se centraron en el tipo de test a realizar, y las herramientas a utilizar. Debido a que la mayor parte de la lógica de cada caso de uso se encontraba en los controladores, una de las primeras decisiones fue no implementar tests unitarios, dado que probar las clases del modelo no reportaba mayor valor, y para probar las clases de los controladores era necesario simular las entradas de datos preseteando los payloads de cada petición. Consideramos que esto podía llevar a no detectar errores introducidos en modificaciones posteriores a la escritura del test, dado que se perdía una parte importante que era la interacción del usuario con el sistema, y la transformación de la interacción en el set de datos a procesar.

Es por esto que decidimos utilizar como estrategia de testing las pruebas funcionales, que nos permitieran probar directamente el comportamiento completo de una función esperada del sistema, todas las opciones que esta pone a disposición del usuario, y el resultado obtenido.

Para que estos test resultaran fáciles de aplicar, y se pudieran hacer pruebas de regresión al introducir un cambio en el sistema y tener la seguridad de que todo funcionaba de acuerdo a lo esperado, concluimos que era necesario automatizar este tipo de pruebas.

La primera opción para implementar esta automatización fue escribirlas dentro del mismo proyecto, con la suite de pruebas que es provista por el framework Symfony. Sin embargo, para la versión utilizada (3.4) resultaba muy difícil simular determinadas acciones del usuario, como la selección de un representante para un cliente jurídico, o

elegir la ciudad de una dirección, por el tipo de campo empleados para su ingreso en los formularios.

Por este motivo, decidimos indagar un poco más y escoger una herramienta específica para automatización de pruebas funcionales. Luego de investigar en la web, y de escuchar el consejo de otros profesionales desempeñados en el área de QA, concluimos que, de acuerdo a nuestra acotada experiencia en el rubro, y el alcance de las pruebas que requeríamos realizar, sería una buena opción utilizar Selenium Web Driver, el cual nos proveía de un conjunto de herramientas más que suficientes para nuestros requerimientos, y nos permitía abordar el proyecto cómodamente desde cualquiera de los lenguajes de programación que ponía a nuestra disposición. En este sentido, escogimos trabajar con Java, dado que no solo era un lenguaje conocido por el equipo de trabajo, sino que también nos permitiría ir un poco más a fondo con el testeo del sistema. Si bien las pruebas funcionales por lo general solo evalúan los resultados a partir de la interacción con el sistema en sí mismo, trabajar con Java nos permitiría conectar el proyecto de pruebas directamente a la base de datos, y no solo comprobar lo que nos presenta la interfaz de usuario, sino también indagar un poco más a fondo y asegurarnos que cada acción del usuario genere la correspondiente modificación del modelo de datos en la base de pruebas.

12.1 Cobertura del Código

Debido a la dimensión del proyecto, y a la cantidad de casos de uso incluidos, acordamos con el cliente acotar la cobertura de los casos de prueba, incluyendo las principales funcionalidades o aquellas consideradas más críticas.

Sería una buena práctica que cada caso de uso tuviera su propio caso de prueba como mínimo (aunque dependiendo de la complejidad del mismo, podrían ser varios escenarios más), y tener una cobertura de código con testing automatizado lo más alta posible.

Sin embargo, dada la envergadura del proyecto y el acotado equipo de trabajo, seguir ese camino hubiera tomado demasiado tiempo, el cual retrasaría la entrega de valor al cliente y a la par elevaría los costos del desarrollo. Además de este aspecto económico, también es real que muchos de los casos de uso son CRUDs sencillos, por lo cual la evaluación del beneficio/coste de trabajo que implica desarrollar los casos de prueba automatizados a nivel funcional decantaron en la decisión de posponer estos puntos para etapas más avanzadas del proyecto.

Esto permitió poner más énfasis en las funcionalidades principales (a nivel de negocio y complejidad), y hacer testeos más completos sobre las mismas. Como consecuencia, cada nueva entrega que se hacía al cliente resultaba en mayor confianza tanto para él como para el equipo de trabajo, dado que se podían correr regresiones que aseguraran una alta calidad en los aspectos claves del proyecto.

Aquellos requerimientos que no generaron tests automatizados fueron probados en su totalidad bajo pruebas manuales, contemplando los diversos flujos posibles de acuerdo a la descripción del caso de uso.

Para no generar documentación extensa pero que reportara poco valor, solo se detallaron los casos de prueba que fueron automatizados. Dada la baja complejidad de aquellos probados manualmente consideramos que los escenarios se pueden determinar con facilidad a partir de la descripción del caso de uso.

El resultado final obtenido de las pruebas fue bastante satisfactorio para el equipo de trabajo, dado que luego de la entrega de cada incremento, obtuvimos un número proporcionalmente bajo de incidencias o bugs detectados por el cliente.

Los casos de prueba desarrollados con testing automático fueron:

Num	*Prueba*	*Entidad*
1	Crear un cliente humano con datos mínimos	Cliente Humano
2	Crear un cliente humano con todos los datos	Cliente Humano
3	Editar un cliente humano	Cliente Humano
4	Crear un cliente jurídico con datos mínimos	Cliente Jurídico
5	Crear un cliente jurídico con todos los datos	Cliente Jurídico
6	Editar un cliente jurídico	Cliente Humano
7	Crear una tarea con los datos mínimos	Tarea
8	Crear una tarea con todos los datos	Tarea
9	Editar una tarea	Tarea
10	Asociar una tarea nueva a un cliente	Tarea Cliente
11	Asociar una tarea a varios clientes	Tarea Cliente
12	Generar las instancias de dichas tareas	Tarea Instancia
13	Especificar (modificar) una tarea para un cliente	Tarea Cliente
14	Crear una tarea rápida con datos mínimos	Tarea Rápida
15	Crear una tarea rápida con todos los datos	Tarea Rápida
16	Crear una reunión	Reunión
17	Eliminar una reunión	Reunión
18	Finalizar una tarea - Generación de entidades financieras	Tarea Instancia
19	Finalizar una tarea cambiando estado de actividad	Tarea Instancia
20	Cambiar estado a no iniciada	Tarea Instancia
21	Cambiar estado a en proceso	Tarea Instancia

22	Cambiar estado a en espera del cliente	Tarea Instancia
23	Delegar una tarea	Tarea Instancia
24	Posponer una tarea	Tarea Instancia
25	Eliminar tarea	Tarea Instancia
26	Crear un ingreso cuenta	Ingresos/Egresos
27	Crear un egreso cuenta	Ingresos/Egresos
28	Crear un crédito usuario	Cuenta Usuario
29	Crear un débito usuario	Cuenta Usuario
30	Verificar los movimientos y el resultado en tabla de empleado	Cuenta Usuario

TABLA 009. Listado de los casos de prueba desarrollados.

12.2 Patrón de diseño

Para este nuevo proyecto menor paralelo a la construcción del sistema general, establecimos un patrón de diseño propio de un proyecto de testing: Page Object Model. El mismo consiste, básicamente, en separar la interacción del driver con el sitio HTML del test en sí mismo, en pos de favorecer la adaptación de un test a los cambios que se producen a lo largo del tiempo en el sistema, modificando simplemente las clases de tipo página.

Profundizando un poco más en este patrón de diseño encontramos dos grandes paquetes: Los test y las clases de tipo página. La función de las clases de tipo página es contener toda la información de la página, y el acceso a la misma. Por ejemplo, una clase de un listado de clientes, contendrá como atributos los id de la tabla, de los campos de filtro, de los botones o enlaces; y los métodos para setear estos campos, para acceder al contenido de la tabla o una fila en particular, y para clickear en los distintos enlaces o botones.

De esta forma, un test no manipula directamente el HTML, sino que utiliza las clases Object Page para poder interactuar con el navegador. En el caso de que sea necesario modificar el contenido de una página, porque este cambió en el proyecto (se agregaron/eliminaron campos, se agregaron/modificaron secciones en el sitio, se modificó el nombre, id o las clases de los distintos elementos) solamente es necesario modificar la clase de esta página, y el test continuará funcionando.

```java
package Modals;

import Pages.CuentasEmpleadosPage;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.ui.Select;

public class FormDebitoUsuarioModal {

    private WebDriver driver;
    private Actions actions;
    // Ids
    private String idConcepto = "debito_usuario_concepto";
    private String idDivisa = "debito_usuario_divisa";
    private String idMonto = "debito_usuario_monto";
    private String idFecha = "debito_usuario_fecha";

    // Botones
    private By guardar = By.id("guardar");

    // Genericas
    public FormDebitoUsuarioModal(WebDriver driver) {
        this.driver = driver;
        this.actions = new Actions(this.driver);
    }
    private WebElement getElementById(String text) {
        WebElement element = driver.findElement(By.id(text));
        moveToElement(element);
        return element;
    }
    public void moveToElement(WebElement element) { this.actions.moveToElement(element).perform(); }

    // Setters
    public void setConcepto(String text){
        WebElement clave = getElementById(idConcepto);
        clave.clear();
        clave.sendKeys(text);
    }
    public void setDivisa(String text){
        Select select = new Select(getElementById(idDivisa));
        select.selectByVisibleText(text);
```

IMAGEN 033. Ejemplo de una clase de tipo Page Object. Representa el modal de débito usuario.

En el ejemplo anterior, podemos observar un ejemplo de un Page Object, en este caso del formulario de un Débito de Usuario. En el mismo encontraremos los id de cada input que el usuario debe completar, un acceso para el botón de guardado, funciones que nos permiten acceder a un elemento web, desplazarnos a este elemento web (que corresponden a métodos privados, ya que son acciones que realiza y encapsula esta misma clase), y luego los métodos públicos que nos permiten setear o interactuar con los distintos elementos web que componen esta página.

De esta forma, el test de crear un nuevo Débito para un usuario no necesitará conocer cómo se compone la estructura HTML del sitio, solo necesitará llamar al método setConcepto para indicar que el nuevo débito es en concepto de un “Pago de haberes”, para poner un caso de ejemplo.

12.3 Implementación del proyecto de test funcional

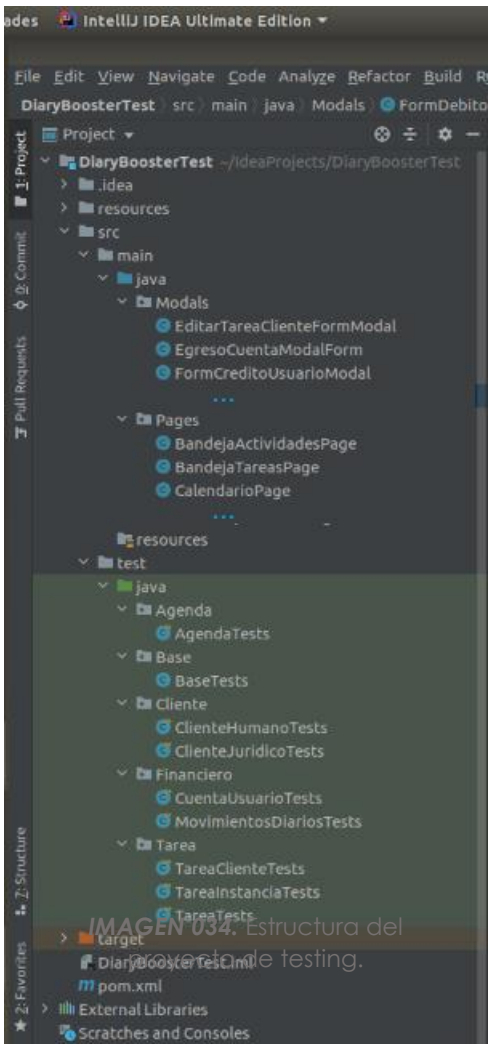


IMAGEN 034. Estructura del proyecto de testing.

De acuerdo a lo planteado por el Page Object Model, la estructura del proyecto de testing quedó tal como se puede apreciar en la imagen: dentro del proyecto Java, encontramos dos paquetes: Modals y Pages. Ambos paquetes contienen clases de tipo Page Object; sin embargo, los separamos en dos grupos diferentes de acuerdo a si eran una página web completa en sí mismos, o si se trataba de una parte del sitio web embebida en un modal. Si bien cada uno de estos Page Objects modales podrían ir incluidos cada uno dentro de su propia página, al estar tan intrínsecamente relacionado su contenido con una funcionalidad específica, decidimos que era mejor que tuvieran su propia clase. Por otro lado, esto también ayuda a la reutilización del código, dado que muchas veces estos modales son llamados desde diferentes páginas de origen, por lo cual si los modeláramos dentro de la misma Page Object que los contienen, deberíamos repetir su contenido en diferentes lugares.

Por otro lado, también podemos apreciar el sector de test del proyecto, donde se encuentran almacenados todos los casos de prueba detallados en la sección anterior. Cada uno de ellos tiene su propio test, reconocible por su propia función antecedida por la anotación @Test. Sin embargo, se encuentran agrupados en paquetes y clases, de acuerdo al módulo del sistema al que corresponden, y la entidad principal que están testando.

El BaseTest es la clase desde la cual se extienden cada uno de nuestros test particulares. El mismo se encarga de inicializar el WebDriver de selenium, dirigirse a la URL del proyecto, e iniciar sesión. Luego, cada uno de los tests particulares puede ser ejecutado individualmente, en conjunto con los demás test de la clase, o del paquete al que pertenecen. También pueden ejecutarse todos cómo un único conjunto (la forma en que lo usamos como pruebas de regresión, para verificar que todo continuara funcionando correctamente).

Cada uno de ellos es independiente, y se pueden ejecutar en cualquier orden. La única clase que tiene test con dependencias internas es la de CuentaUsuarioTests, sin embargo, también puede ejecutarse dentro de todo el conjunto sin ningún problema, dado que las dependencias internas se encuentran correctamente indicadas dentro de las

anotaciones de los mismos. Si bien el ideal para todo set de pruebas es que estas sean independientes entre sí, vimos la oportunidad de extender nuestro aprendizaje e implementamos una funcionalidad que está presente en la suite de test de Java, considerando que no afecta al correcto funcionamiento de la suite de pruebas.

Por último, para poder realizar las pruebas es necesario hacer unas modificaciones mínimas en nuestro entorno local de desarrollo. Para no generar dependencias entre las pruebas, y poder iniciar rápidamente el proceso de testing, generamos una base de datos específica para poder correr las pruebas, que contiene los valores iniciales esperados por cada uno de los test. De esta forma, cuando vamos a iniciar las pruebas de regresión, simplemente importamos esta base con la configuración inicial de los datos, y apuntamos el proyecto en nuestro entorno local a esta nueva imagen de la base de datos. Corremos las pruebas, verificamos el correcto funcionamiento de todo el conjunto, y luego, volvemos a apuntar la base de datos del proyecto a la versión de desarrollo normal. Este trabajo extra en realidad no significa más que un breve período de tiempo entre modificaciones y limpieza de caché, pero es necesario dado que no contamos con más entornos que nuestro propio local en el cual desarrollamos, y el entorno productivo.

13. Extensibilidad

El software DiaryBooster ha sido diseñado a partir de la solicitud realizada por el estudio contable Gaudiano. Sin embargo, si bien se ha creado un producto a partir de los requerimientos planteados por este cliente particular, se ha acordado que la propiedad del software es del grupo de desarrollo, pudiendo disponer del mismo para futuros proyectos.

Considerando el resultado final obtenido, este producto de software es completamente funcional y valioso para cualquier estudio contable. Incluso, puede ser utilizado por otro tipo de organizaciones, como estudios jurídicos, por ejemplo, siempre que encuentren práctica la forma de funcionar que plantea el sistema, dividiendo los trabajos a realizar en tareas y estas a su vez en actividades repetitivas, programables y con vencimientos, a ser asociadas a múltiples clientes y distribuidas entre uno o varios usuarios.

Además, existen ciertas funcionalidades a ser desarrolladas que pueden brindar mucho valor tanto al cliente actual como a futuros potenciales,por ejemplo:

1. un módulo de estadísticas que explote los datos generados,
2. módulo de comunicaciones con los clientes del estudio,
3. una vista de acceso para dichos módulos de comunicación, de forma que puedan conocer el estado actual de sus encargos, su estado de cuenta, etc.

Desde un punto de vista comercial, también se podría refactorizar el sistema de forma que el acceso a cada módulo no dependa solo del perfil del usuario (Empleado o Administrador), sino también de los módulos disponibles, y vender cada uno de estos por separado, de acuerdo a las necesidades de cada cliente. Se estima este cambio como un trabajo de esfuerzo moderado, con un alto impacto en la cantidad de líneas de código afectadas, pero de una complejidad moderada, que abre la posibilidad de implementar un nuevo servicio de control de acceso, que incorpore esta nueva variable (módulos disponibles).

El otro aspecto a considerar es la inclusión de nuevos clientes al sistema. Exploramos dos posibles soluciones a este aspecto:

1. La primera considera un único punto de acceso al sistema, un *login* genérico, en el cual cada cliente accediera al sistema, y fuera redireccionado a su propia instancia del producto, donde solo viven sus datos.

La otra opción considerada consiste en añadir al sistema productivo actual un subsistema de control de propiedad, añadiendo un propietario a cada entidad del sistema, y de esta manera proveer acceso a las mismas mediante la adhesión de una jerarquía organizacional a la administración de usuarios (cada usuario pertenece a una organización, pudiendo acceder solo a los registros que le pertenecen a la misma).

14. Conclusiones

La puesta en funcionamiento de DiaryBooster ha implicado un cambio de alto impacto en el quehacer diario del estudio contable Gaudiano. El sistema final obtenido ha logrado cubrir exitosamente los requerimientos planteados inicialmente por el cliente, y ha logrado brindar soluciones a otros problemas o necesidades que fueron descubiertas a medida que se avanzó en el proceso de desarrollo.

Los principales aportes del proyecto para el estudio se resumen en:

- Facilitar el seguimiento de las tareas de sus clientes, evitando pérdidas de información u olvidos involuntarios.
- Automatización de vencimientos de tareas y actividades de acuerdo a los calendarios de AFIP.
- Disminución de la carga de trabajo dedicada a tareas administrativas.
- Registro efectivo y fiable para el seguimiento de los estados de cuenta de los clientes respecto del estudio.
- Consolidar una fuente de información centralizada, única y de fácil acceso para cada uno de los miembros del estudio.
- Facilitar las tareas de control y seguimiento de la labor de los empleados.

Esperamos que, a lo largo del tiempo, estas características permitan que los clientes perciban una mejora en la calidad del servicio brindado, y a su vez, posibiliten un incremento de las operaciones del estudio, permitiéndoles gestionar de forma eficiente mayor cantidad de clientes.

Uno de los aspectos que más resaltamos como equipo de trabajo es la experiencia obtenida al llevar adelante el proyecto. Ser capaces de aplicar los conocimientos adquiridos durante el cursado de la carrera, tanto los saberes teóricos como las experiencias prácticas, en un trabajo de índole profesional pero que también cuenta con el abordaje académico de un proyecto universitario, ha sido sin duda una de las experiencias formativas más relevantes durante nuestro paso por la universidad. No se trataba solamente de llevar adelante un proyecto de software, sino de hacerlo aplicando de la mejor manera posible todo lo aprendido, seleccionando las herramientas correctas y más adecuadas para el mismo, sabiendo que este aspecto sería evaluado más adelante. Esto nos ayudó a valorar las herramientas adquiridas durante los años de estudio, y a darle la importancia real que tienen a la hora de utilizarse en el mundo laboral.

Cabe mencionar como puntos importantes la experiencia en el trato con el cliente, la planificación de las tareas y los esfuerzos estimados para cada una, la carga de retrabajo, el manejo de los plazos y expectativas del cliente y del propio equipo de trabajo. Al mismo tiempo, considerando más los aspectos técnicos, nos sirvió para aplicar conocimientos aprendidos en procesos de desarrollo de software, elicitación de requerimientos, patrones y modelos de diseño, de arquitectura, profundizar en sus defectos y virtudes, y ganar en experiencia en relación al momento de aplicarlos, o, en su defecto, cuándo se impone la búsqueda de una alternativa.

Finalmente, pero no menos importante, las habilidades adquiridas a nivel de tecnologías aplicadas constituyen una gran ganancia para el equipo de trabajo.

Para ambos integrantes del equipo de trabajo fue la primera experiencia con un cliente particular, con todas las novedades, características y complicaciones que implican estos tipos de desarrollo. Gran parte de los inconvenientes surgieron de la gestión de la relación con el cliente. Las necesidades del estudio fueron presentadas de una forma sencilla durante las primeras reuniones, pero a medida que se fueron refinando y detallando en casos de uso, la complejidad de la solución se incrementó rápidamente. Al tratarse de un desarrollo que no correspondía simplemente a una relación laboral, sino que también sería utilizado como nuestro proyecto final de carrera, muchas veces nos encontramos en la tensión entre establecer límites de acuerdo a lo ya acordado, y por otro lado, mantener el proyecto en marcha.

Otra dificultad con la que nos encontramos al avanzar en el proyecto, fue el nivel de experiencia que tenía el equipo de trabajo. Si bien ambos trabajábamos en desarrollo web, no contábamos con un gran nivel de seniority.

En última instancia, uno de los factores que más nos afectó a nivel de la distribución del tiempo fueron aspectos personales de cada uno de los integrantes del equipo de trabajo, e incluso del cliente, que generaron demoras en el proyecto.

En consecuencia, por todos los puntos expuestos en esta sección el proyecto sufrió retrasos con respecto a la planificación inicial. Aun así, esto no generó inconvenientes en ningún momento respecto de la relación con el cliente, sino que se lograron acuerdos que permitieron que el proyecto cumpliera con las expectativas y necesidades de ambas partes. Si bien fallamos en las estimaciones iniciales y con ello en la planificación, y también reconocemos aspectos a mejorar a nivel de gestión, considerando que fue nuestra primera experiencia, la satisfacción del cliente, los saberes y destrezas adquiridas y el producto de software resultante, consideramos que el proyecto fue exitoso.

15. Referencias

[1] Use Case Diagrams Reference:

https://www.uml-diagrams.org/use-case-reference.html

[2] UML Sequence Diagrams:

https://www.uml-diagrams.org/sequence-diagrams.html

[3] Test Automation University:

https://testautomationu.applitools.com/

[4] The Selenium Browser Automation Project:
https://www.selenium.dev/documentation/en/

[5] Contributing to the Selenium Site & Documentation:
https://www.selenium.dev/documentation/en/contributing/

[6] PageObject:

https://martinfowler.com/bliki/PageObject.html

[7] Overall Software:

https://getbootstrap.com

https://symfony.es/documentacion

https://twig.symfony.com/doc/2.x/

https://api.jquery.com/

https://docs.oracle.com/en/java/javase/11/

https://maven.apache.org/guides/

https://www.phpmyadmin.net/docs/

[8] Ian Sommerville: Ingeniería de Software. 9na Edición. Pearson (2011)

[9] Roger S. Pressman. Ingeniería de Software, Un Enfoque Práctico. Mc Graw-Hill. 7ma Edición, 2010.