Tabla de contenido

1. Capítulo 1: Introducción

El presente Proyecto Final de Carrera consiste en el desarrollo e implementación de un sistema de información web orientado a dar soporte a las tareas que se desarrollan en un consultorio de kinesiología. Esto incluye la gestión de pacientes y sus historias clínicas, la gestión de turnos, sesiones y tratamientos, el registro de movimientos de caja, entre otras actividades.

El problema a resolver, que impulsó la ejecución del proyecto se centró en la ineficiencia en la gestión de tareas diarias, la pérdida de información y las irregularidades surgidas producto de realizar los procesos sin soporte informático. Debíamos comprender el dominio, modelarlo, organizar y validar los conceptos, para luego implementar un sistema que permita llevar a cabo las actividades mencionadas antes. Además, y con similar importancia, se presentaban determinados atributos no funcionales a atender, como la facilidad de uso, seguridad de la información y portabilidad.

Si bien el sistema desarrollado como parte de este proyecto final brinda soporte a los procesos que la clínica posee, provee además a los especialistas de ciertas capacidades que mejoran su experiencia con los pacientes y con la información manejada. Entre estas características podemos mencionar las *estadísticas*, que brindan la posibilidad de realizar un seguimiento de información relevante y dar soporte a la toma de decisiones. Asimismo, también contribuyen las *notificaciones*, que permiten dar aviso a los especialistas acerca de ciertos eventos y recordatorios; y los *recordatorios* para los pacientes, enviados para que éstos no olviden los turnos previamente reservados. De esta manera, el sistema de información mejora la experiencia tanto de los especialistas que trabajan en la clínica como de los pacientes que acuden a ella.

El objetivo general establecido para el proyecto fue desarrollar un sistema de información web que brinde las funcionalidades necesarias para dar soporte integrado a las actividades diarias del consultorio de kinesiología, que fue llevado a cabo exitosamente a través del cumplimiento de los distintos objetivos particulares definidos. A lo largo del informe se describen las actividades y resultados de cada uno de los procesos de desarrollo de software, así como también los cambios de alcance y riesgos manejados como parte de la gestión integral del proyecto.

En los siguientes capítulos buscamos describir los procesos seguidos desde un aspecto práctico y de realización del proyecto. En el capítulo 2, se encuentran definiciones del dominio del problema, las herramientas y metodología utilizadas para resolverlo, además de la fundamentación de su utilización. Más adelante en el documento, en el capítulo 3, se pueden encontrar detalles del desarrollo del proyecto: los procesos seguidos y los resultados obtenidos en las etapas de análisis, diseño, pruebas e implementación del sistema, así como también detalles acerca de la gestión del proyecto. Finalmente concluimos el documento con reflexiones, conclusiones y aportes del proyecto.

2. Capítulo 2: Dominio, métodos y herramientas

2.1. Dominio

El proyecto se sitúa dentro del dominio de la salud, más precisamente en la gestión de un consultorio donde concurren pacientes para realizar tratamientos ambulatorios de fisiokinesioterapia. Se describen a continuación, actividades y conceptos claves del dominio en estudio que fueron obtenidos como parte del relevamiento realizando en el proyecto.

2.1.1. Atención de un paciente

Cuando un paciente desea atenderse en el consultorio, se comunica al mismo para pedir un turno (fecha y hora de atención). Desde el consultorio, teniendo en cuenta las características del tratamiento y la agenda de los profesionales, se proponen fechas y horarios posibles y se llega a un acuerdo con el paciente sobre cuándo asistir.

Una vez que el paciente asiste al consultorio se procede a registrar los datos personales del mismo en una ficha clínica (también llamada ficha kinésica). Si la ficha del paciente no existe, se crea una nueva; caso contrario, se amplía la ficha del paciente con el nuevo tratamiento que va a realizar y se actualizan sus datos personales. A partir de este momento el paciente inicia un tratamiento para atender su problema de salud.

Los tratamientos, en su mayoría, son de tipo kinesioterapia y fisioterapia, e implican la realización de 1 o más sesiones. Cada sesión implica programar una nueva asistencia del paciente al consultorio (un nuevo turno). Así el paciente realiza las sesiones planificadas hasta alcanzar el fin del tratamiento.

El consultorio atiende pacientes tanto particulares como por obra social. En este último caso se requiere que el paciente presente una o más órdenes médicas destinadas a cubrir las sesiones que va a realizar. Los profesionales del consultorio hacen un seguimiento de estas órdenes a fin de poder facturar. Este seguimiento implica la generación de planillas mensuales que el profesional debe presentar ante el Círculo de Kinesiólogos.

A continuación se describen con más detalle algunos de los conceptos mencionados anteriormente.

2.1.2. Tratamientos

Se denomina *tratamiento* a un conjunto de sesiones que comparten el mismo tipo de tratamiento (por ejemplo: fisioterapia, kinesioterapia) y apuntan a resolver el mismo diagnóstico (por ejemplo: esguince). Puede decirse que un tratamiento relaciona y agrupa un

conjunto de sesiones destinadas a resolver un problema de salud de un paciente. Asimismo, un *tipo de tratamiento* describe las características de la práctica que se le va a realizar al paciente en la serie de sesiones a realizar; el profesional abordará el problema de salud del paciente con distintas estrategias dependiendo el tipo de tratamiento. Ejemplos de tipos de tratamiento son: fisioterapia, kinesioterapia, kinesioterapia respiratoria, gimnasia terapéutica.

2.1.3. Sesiones

Como se dijo anteriormente, por cada sesión, el paciente asiste al consultorio para realizar un tratamiento en particular. Por lo general, el paciente dispone de ciertos días y horarios en la semana donde puede asistir. Esto brinda la posibilidad de programar uno o más turnos del tratamiento desde un principio. Es decir, en la primera sesión (o alguna de las primeras) se puede programar los turnos de futuras sesiones a realizar. Un tratamiento de fisiokinesioterapia está compuesto habitualmente por un conjunto de 5 a 10 sesiones.

- Turno vs sesión: un turno se describe por la fecha y hora en la que un paciente asiste al consultorio. La sesión, por su parte, extiende este concepto asociando el paciente y su turno al tipo de tratamiento que el mismo esté realizando. Una sesión tiene una determinada duración (en minutos) la cual varía dependiendo del tipo de tratamiento.
- Asistencia a las sesiones: los pacientes ocasionalmente pueden ausentarse a una sesión. Si el paciente da aviso con anticipación de esta situación, el consultorio considera que la sesión no transcurrió, es decir, que ésta no cuenta y el paciente puede re-programarla en un futuro. En caso de no dar aviso con anticipación, el paciente pierde la sesión y ésta no puede reprogramarse. En este caso la sesión cuenta como si el paciente hubiese asistido a la misma.

2.1.4. Ordenes médicas

- Los tipos de tratamiento, por un lado, pueden ser prestaciones cubiertas por una obra social, o bien, actividades (como gimnasia o masajes) que deben realizarse de forma particular. En aquellos casos en los cuales el tipo de tratamiento es cubierto por la obra social del paciente, éste debe presentar la orden médica correspondiente.
- Las prestaciones que un profesional de la medicina realiza a sus pacientes tienen asociado un código que permite identificarlo; dicho código es importante porque debe estar presente en la planilla que se entrega al círculo de Kinesiólogos todos los meses (la cual es necesaria para facturar las sesiones trabajadas).

- Autorizaciones: las órdenes médicas deben autorizarse (salvo casos excepcionales) antes de presentarse al Círculo de Kinesiólogos. Algunas de las obras sociales (IAPOS por ejemplo) brindan el servicio de autorizaciones on-line. La tarea de autorizarlas recae sobre la administración del consultorio. El proceso de autorización da como resultado un número o código que debe incluirse en la planilla ya mencionada. Para algunas obras sociales existen prestaciones que no requieren autorización (Ej. Sancor - Fisiokinesioterapia); sin embargo el código se incluye en la planilla de todas formas.

2.1.5. Estudios de diagnóstico por imágenes

A los profesionales del consultorio les interesa poder guardar o disponer de una copia de los estudios de diagnóstico por imágenes de sus pacientes. En ocasiones se les realiza una captura fotográfica.

2.1.6. Consentimiento informado

Cada vez que un paciente inicia un nuevo tratamiento en el consultorio, debe firmar un documento llamado *Consentimiento Informado*. La *Ley 26529 - Derechos del paciente, historias clínicas y consentimiento informado*, enuncia que el paciente debe dejar constancia por escrito que está de acuerdo con el tratamiento que va a comenzar a realizar con el kinesiólogo. La presencia de dicho consentimiento informado es importante ya que de lo contrario el seguro no cubre la mala praxis del profesional.

2.2. Metodología de desarrollo

El proceso de desarrollo utilizado en el proyecto está dentro del *framework* que plantea el Proceso Unificado. Desde una perspectiva resumida, puede decirse que el Proceso Unificado hace referencia a un contexto que sirve como *plantilla* que puede reutilizarse para crear instancias de ella, en forma análoga a una clase de la Programación Orientada a Objetos, que puede utilizarse para crear objetos [1]. La instancia o refinamiento más conocido es RUP (Rational Unified Process), de donde también tomamos lineamientos.

Nuestro proceso

En general seguimos las recomendaciones y lineamientos que plantea el Proceso Unificado, aunque en ciertos aspectos adaptamos el proceso a las características del equipo, incorporando prácticas y quitando artefactos.

A partir del Proceso Unificado pueden desprenderse o instanciarse desde un proceso en cascada hasta una metodología ágil. Por este motivo, explicamos a continuación una visión general de la metodología que empleamos.

2.2.1. Descripción general

El Proceso Unificado es una metodología basada en la arquitectura, guiada por casos de uso, iterativa e incremental, lo cual permite enfocarse en la arquitectura y aspectos de calidad, en las funcionalidades requeridas desde el punto de vista del usuario, y en desarrollar el software realizando iteraciones que permiten ajustar el proceso de una iteración a la siguiente. El Proceso Unificado se repite a lo largo de una serie de ciclos. Al final de cada ciclo se obtiene como resultado una versión externa del sistema, es decir, una versión expuesta a los clientes y usuarios. Se planifican nuevos ciclos a medida que surjan nuevos requerimientos o cambios significativos.

Cada ciclo está compuesto por cuatro fases: Inicio, Elaboración, Construcción y Transición. A su vez, cada fase se descompone en una o más iteraciones. En cada una de las cuatro fases se planifica realizar, con distinto esfuerzo, actividades de Análisis, Diseño, Desarrollo y Pruebas. En la fase de Elaboración predominan el análisis y el diseño, incorporando algunas tareas de desarrollo de la arquitectura de alto nivel del sistema y la preparación de casos de prueba funcionales. Por otro lado, en la fase de Construcción predominarán la codificación y las pruebas, planificando al principio de cada iteración una tarea de análisis y diseño para trabajar sobre los potenciales cambios de requerimientos desde la iteración previa.

En la siguiente imagen vemos un ejemplo de ciclo del Proceso Unificado, como se compone y como se distribuyen las actividades en las distintas fases. Se puede apreciar que las fases a su vez se subdividen en iteraciones.

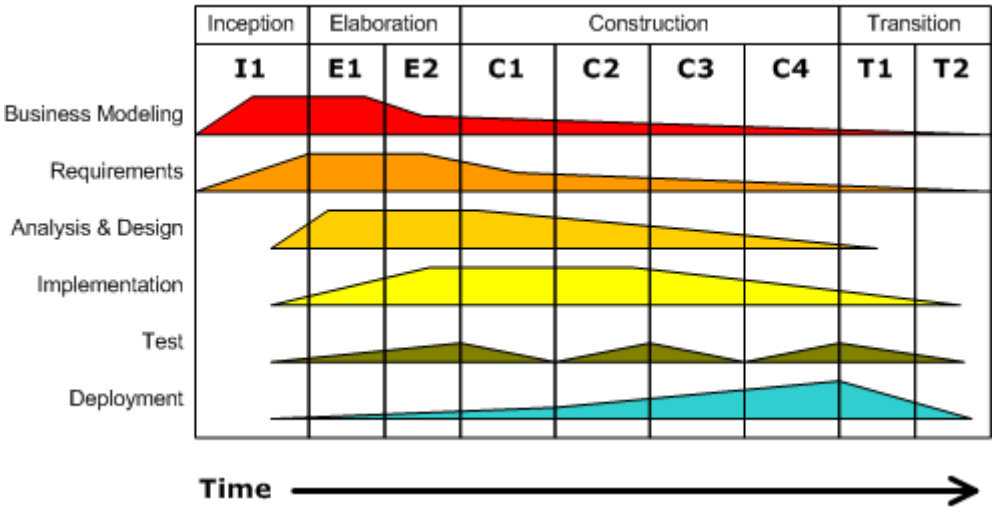

Ilustración 1.1: Ejemplo de un ciclo del Proceso Unificado.

Cada iteración puede considerarse como un miniproyecto que se asemeja a un ciclo de vida en cascada debido a que se desarrolla a través de actividades en cascada. Podríamos llamar una *minicascada* a cada iteración [1].

En cada iteración se producen resultados tangibles en forma de versiones internas (versión preliminar, no expuesta a clientes y usuarios), y cada una de ellas aporta un incremento y demuestra la reducción de los riesgos con los que se relaciona. Sin embargo, estas versiones pueden ser presentadas a los clientes, en cuyo caso proporcionan una retroalimentación valiosa [1].

2.2.2. Basado en la arquitectura

La arquitectura nos da una clara perspectiva del sistema completo, necesaria para controlar el desarrollo. Necesitamos una arquitectura que describa los elementos más importantes del modelo. Estos elementos significativos, arquitectónicamente hablando, incluyen subsistemas, dependencias, interfaces, colaboraciones, nodos y clases activas. Describen los cimientos del sistema, que son necesarios como base para comprenderlo, desarrollarlo y producirlo.

Es por esto que el primer objetivo de la fase de elaboración es establecer una arquitectura sólida de forma que sea una arquitectura base. Como resultado, se entra en la fase de construcción con unos fundamentos sólidos para construir el resto del sistema.

La arquitectura se representa mediante vistas del modelo de diseño, de despliegue, etc. Este conjunto de vistas se representan utilizando la Estrategia 4+1 descrita en el siguiente capítulo. Dado que una vista consiste en un determinado modelo analizado desde una cierta

perspectiva, por ejemplo si tomamos una vista del modelo de casos de uso esta se parecerá al propio modelo de casos de uso, pero estará compuesta solamente de aquellos actores y casos de uso que son arquitectónicamente significativos.

Existen varios motivos por los cuales debemos centrarnos en la arquitectura:

- Comprender el sistema: Para desarrollar el sistema, este debe ser comprendido por todos los que vayan a intervenir en él. La descripción de la arquitectura debe capacitar a los desarrolladores, directivos, clientes y otros usuarios para que comprendan lo que se está haciendo con suficiente detalle como para facilitar su participación. Los modelos y diagramas UML ayudan a esto, y deben utilizarse para describir la arquitectura.
- Organizar el desarrollo: Dividiendo el sistema en subsistemas, con las interfaces claramente definidas y con un responsable o un grupo de responsables establecido para cada subsistema, el arquitecto puede reducir la carga de computación entre los grupos de trabajo de los diferentes subsistemas. De este modo, se permite que el software en las distintas partes (subsistemas) progrese independientemente.
- Fomentar la reutilización: Los componentes de software reutilizables están diseñados y probados para encajar, y así el tiempo de construcción y el coste son menores.

  Una buena arquitectura ofrece a los desarrolladores un andamio estable sobre el que trabajar. El papel de los arquitectos es definir ese andamiaje y los subsistemas reutilizables que el desarrollador pueda utilizar.

  En nuestro caso planteamos la arquitectura de forma genérica para poder adaptar el sistema a otros dominios similares y poder reutilizar gran parte de la arquitectura en un futuro.

- Hacer evolucionar el sistema: El sistema debe ser fácil de modificar; esto quiere decir que los desarrolladores podrán modificar el diseño o la implementación sin tener que preocuparse por la repercusión en el sistema. En la mayoría de los casos, deberían ser capaces de implementar nuevas funcionalidades (es decir, casos de uso) sin tener que pensar en un impacto dramático en el diseño e implementación existentes.

2.2.3. Guiado por casos de uso

Existen varios motivos por los cuales los casos de uso son buenos y hemos optado por su utilización:

- Capturar requisitos funcionales: La idea de caso de uso permite la identificación del software que cumple con los objetivos del usuario. Los casos de uso son las funciones que proporciona un sistema para añadir valor a sus usuarios. Tomando la perspectiva de cada tipo de usuario, podemos capturar los casos de uso que necesitan para hacer su trabajo. Por otro lado, si comenzamos pensando en un conjunto de buenas funciones del sistema sin pensar en lo casos de uso que emplean los usuarios concretos, será difícil decir si esas funciones son importantes o incluso si son buenas.

  Además, los casos de uso son intuitivos. Los usuarios y los clientes no tienen que aprender una notación compleja. En su lugar, se puede utilizar lenguaje natural, lo que hace más fácil la lectura de los casos de uso y la propuesta de cambios.

- Dirigir proceso de desarrollo: El que un proyecto esté dirigido por casos de uso significa que progresa a través de una serie de flujos de trabajo que inician a partir de los casos de uso. Los casos de uso ayudan a los desarrolladores a encontrar las clases de diseño del modelo de desarrollo. Los desarrolladores intentan encontrar clases que realicen los casos de uso a medida que los leen. Los casos de uso también nos ayudan a desarrollar interfaces de usuario que le permiten a los usuarios desempeñar sus tareas de manera más sencilla y eficiente.
  Los casos de uso no solo inician un proceso de desarrollo sino que lo enlazan. Son un importante mecanismo para dar soporte a la trazabilidad a través de todos los modelos. Un caso de uso en el modelo de requisitos es trazable a su especificación en el análisis y en el diseño, a todas las clases participantes en su especificación, a componentes y finalmente, a los casos de prueba que lo verifican. Esta trazabilidad es un aspecto importante de la gestión de un proyecto. La trazabilidad entre los casos de uso y el resto de los elementos del modelo hace más fácil mantener la integridad del sistema y conservar actualizado al sistema en su conjunto cuando tenemos requisitos cambiantes.

- Estimar esfuerzo: Los casos de uso nos ayudaron a planificar, asignar y controlar muchas de las tareas a realizar. Por cada caso de uso se pueden identificar un grupo de tareas. Cada caso de uso a especificar es una tarea, cada caso de uso a diseñar es una tarea, y cada caso de uso a probar es una tarea. Podemos incluso estimar el esfuerzo y el tiempo necesario para llevar a cabo esas tareas. Las tareas identificadas a partir de

los casos de uso nos ayudan a estimar el tamaño del proyecto y los recursos necesarios.

La técnica de estimación basada en puntos de casos de uso utilizada se encuentra en el Anexo A.

- Para idear la arquitectura: Los casos de uso también nos ayudan a idear la arquitectura del sistema. Mediante la selección del conjunto correcto de casos de uso (los casos de uso arquitectónicamente significativos) para llevarlo a cabo durante las primeras iteraciones, podemos implementar un sistema con una arquitectura estable que pueda utilizarse en muchos ciclos de desarrollo subsiguientes.

2.2.4. Iterativo e incremental

Uno de los factores más influyentes en el diseño del Proceso Unificado es el esfuerzo por reducir los riesgos. Jacobson *et al.* describen: "Identificamos, priorizamos, y llevamos a cabo las iteraciones sobre la base de los riesgos y su orden de importancia. Esto se hace así cuando evaluamos tecnologías nuevas, y cuando trabajamos para cumplir con las necesidades de los usuarios [...]. También es así cuando, en las primeras fases, vamos estableciendo una arquitectura que será robusta [...]. Sí, organizamos las iteraciones para conseguir una reducción del riesgo [...]. El objetivo es acabar con los riesgos en una iteración temprana." [1]

La cita anterior plantea un escenario similar al nuestro: los principales riesgos son técnicos asociados al uso de una nueva tecnología, donde buscamos establecer una arquitectura a la que puedan incorporarse cambios sin un mayor esfuerzo, ya que estamos sujetos a requerimientos cambiantes que necesita el cliente.

Los autores de *The Unified Software Development Process* también hacen una observación que nos guió en la planificación de las fases: "[...] en principio, todos los riesgos técnicos pueden hacerse corresponder con un caso de uso o un escenario de un caso de uso. Aquí, correspondencia quiere decir que el riesgo se atenúa si se desarrolla el caso de uso con sus requisitos funcionales y no funcionales. Esto es cierto no sólo para los riesgos relativos a los requisitos y a la arquitectura, sino también para la verificación del hardware y software subyacentes. Mediante una cuidadosa selección de los casos de uso, podemos probar todas las funciones de la arquitectura subyacente." [1]

La gestión de riesgos planteada por el Proceso Unificado marca una de las diferencias con procesos más lineales como el *cascada*. Si realizarnos un gráfico del riesgo

comparándolo con el tiempo de desarrollo, el proceso iterativo empieza a reducir riesgos importantes en las primeras iteraciones. En el momento en que el trabajo alcanza la fase de construcción, quedan pocos riesgos importantes, y el trabajo prosigue sin problemas. En contraste, si utilizamos el modelo en cascada, los riesgos importantes no se tratan hasta la fase de integración. La siguiente imagen muestra un ejemplo de lo mencionado.

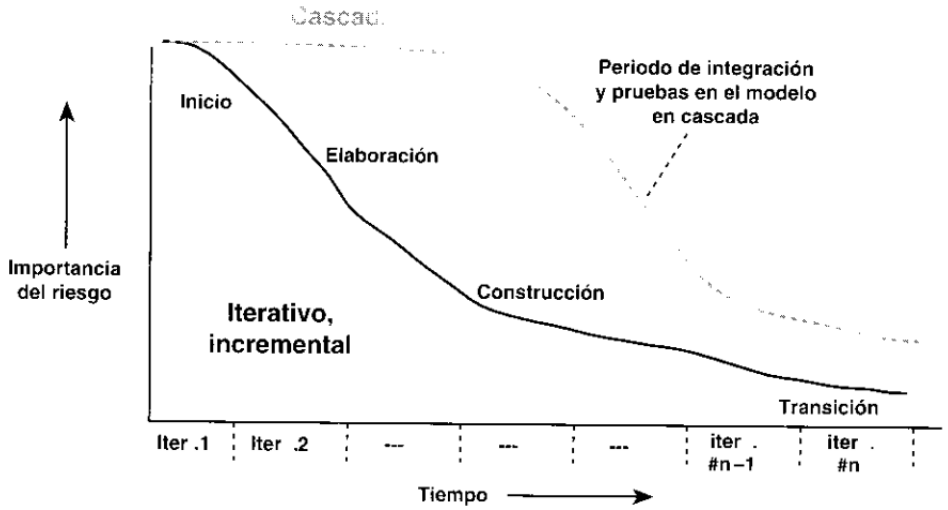

Ilustración 1.2: Los riesgos importantes se identifican y se reducen al principio del desarrollo iterativo, a diferencia del desarrollo en cascada. En este último, los riesgos más importantes permanecen hasta que la integración y las pruebas se ocupan de ellos, como lo indica la línea discontinua.

Por otro lado, cada construcción puede ocasionar problemas y llevar a una parada temporal del avance. El siguiente gráfico nos ayuda a explicar estos conceptos.

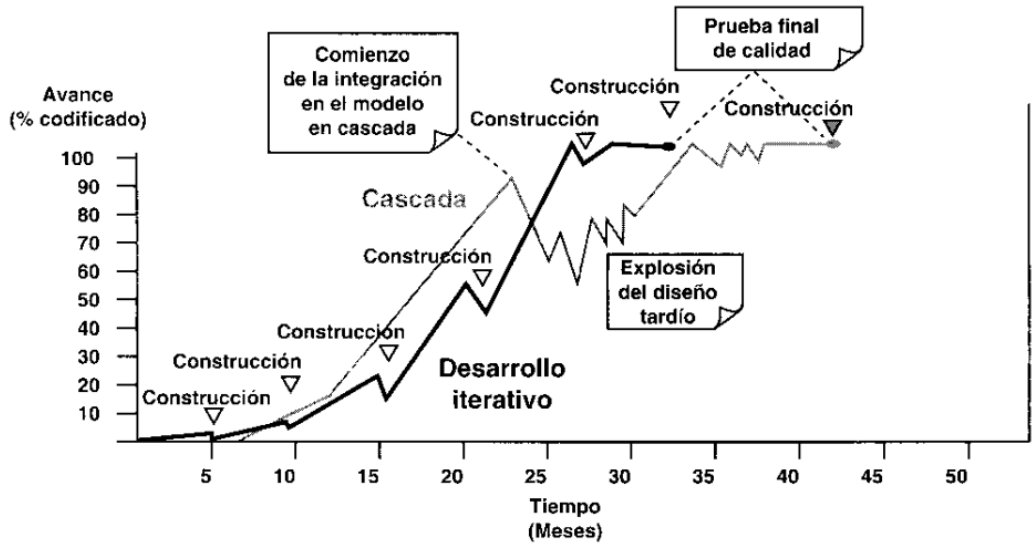

Ilustración 1.3: Integración y recuperación ante problemas en un proceso iterativo vs un proceso en cascada.

Debido a que los incrementos en una metodología iterativa son relativamente pequeños (comparados con la integración final del producto entero), la recuperación se lleva

a cabo rápidamente (línea más oscura). En el método en cascada (la línea fina), los desarrolladores no comienzan la implementación hasta haber terminado los requisitos, y el diseño. Informan de un buen progreso en la implementación porque no tienen construcciones intermedias que les indiquen otra cosa. Los problemas están ocultos hasta que la integración y las pruebas los descubran todos a la vez (explosión del diseño tardío). En el desarrollo iterativo, la implementación comienza más temprano y las construcciones frecuentes no sólo descubren pronto los problemas, sino que también los solucionan en pequeños lotes más fáciles de manejar. [1]

2.2.5. Prácticas de otras metodologías

2.2.5.1. Programación en parejas

Esta práctica propuesta por la metodología Extreme Programming (XP) sugiere que se aplique a la totalidad del código. Si bien no habíamos planificado seguir esta práctica, consideramos que la aplicamos a lo largo del proyecto. Se programó en parejas, sobre todo el código que resolvía asuntos del negocio, de lógica del sistema, y aquellas cuestiones de presentación (interfaz de usuario) y persistencia que resultaban complejas. Notamos ciertos beneficios al aplicar la programación en parejas:

- Reducción del tiempo requerido para resolver problemas y situaciones que se presentaban como complejas, donde se necesitaba esfuerzo mental.
- Cada integrante se siente dueño de todo el código, y se elimina casi por completo el rechazo a tocar código ajeno.
- La revisión del código que escribe el compañero se hace en el momento, facilitando el entendimiento del código en revisiones futuras y reduciendo el tiempo de resolución de bugs o cambios.

2.2.5.2. Refactoring

Refactorizar implica mejorar el código ya existente. Desde el principio de la codificación se tuvieron en cuenta los patrones, buenas prácticas y características que hacen a un código legible y fácil de mantener. Sin embargo hay ciertas situaciones en las que se genera código de mala calidad:

- En la implementación de casos de uso orientados a mitigar riesgos, donde se quiere saber si el CU puede ser resuelto de una determinada manera, con una determinada tecnología, o hay que encararlo de otra forma. Por lo general son cosas complejas, que requieren de una buena estructuración, que se postergan para una refactorización.

Aquí pueden aparecer escasa documentación, clases o métodos muy grandes, código duplicado.

- Desconocimiento de cierta sintaxis o librerías del lenguaje que resuelven de manera más *prolija* ciertas circunstancias. Por ejemplo en el manejo de fechas, o lectura de los parámetros de una solicitud web. Es necesario refactorizar el código generado sin este conocimiento.
- Simplemente porque en el momento, al desarrollador o pareja que se encontraba codificando, no se le ocurrió una mejor forma de resolver la situación.

2.2.5.3. Integración continua

La integración continua es una práctica en la que cada vez que un desarrollador (o pareja de desarrolladores) finaliza una sesión de trabajo, debe integrar sus cambios al resto del código del sistema. Integrar implica que las modificaciones y nuevas funcionalidades se acoplen al sistema y que a su vez los test de unidad sigan funcionando al 100%. La integración continua facilita la corrección de los problemas de integración ya que si algo falla, solo una cosa ha cambiado: los nuevos cambios integrados. Es responsabilidad del desarrollador que incorpora los cambios dejar la nueva versión del sistema (con sus cambios incorporados) funcionando al 100%.

Nosotros nos apoyamos en los test de integración y en la herramienta de versionado. A pesar que usamos un sistema de versionado distribuido como es Git, al final de cada sesión de trabajo se subían los cambios hechos al repositorio remoto, procurando siempre dejar consistente (que compile y sin errores aparentes) el código del sistema.

2.3. Herramientas

Las herramientas soportan los procesos de desarrollo de software modernos y, a su vez, los procesos se ven influidos fuertemente por las herramientas. Las herramientas son buenas para automatizar procesos repetitivos, mantener el trabajo estructurado, gestionar grandes cantidades de información y para guiarnos a lo largo de un camino de desarrollo concreto.

Con un soporte pobre de herramientas, el proceso debe sostenerse sobre gran cantidad de trabajo manual y será por tanto menos formal y consistente. Sin herramientas que favorezcan la consistencia a lo largo del ciclo de vida, será difícil mantener actualizados los modelos y la implementación. Las herramientas están ahí para automatizar el proceso tanto como sea posible.

Un proceso soportado por herramientas permite el trabajo concurrente del conjunto completo de trabajadores, y proporciona una manera de comprobar la consistencia de todos los artefactos [1].

Las herramientas utilizadas para dar soporte al ciclo de vida completo son:

- Gestión de requerimientos: Se utiliza para almacenar, examinar, revisar, hacer el seguimiento y navegar por los diferentes requisitos de un proyecto de software.
- Modelado visual: Se utiliza para automatizar el uso de UML, es decir, para modelar y ensamblar una aplicación visualmente. Con esta herramienta conseguimos la integración con entornos de programación y aseguramos que el modelo y la implementación siempre sean consistentes.
- Herramientas de programación: Proporciona una gama de herramientas, incluyendo editores, compiladores, depuradores, detectores de errores y analizadores de rendimiento.
- Aseguramiento de la calidad: Se utiliza para probar aplicaciones y componentes, es decir, para registrar y ejecutar casos de prueba.

Además de estas herramientas orientadas a la funcionalidad, hay otras herramientas que abarcan todo el ciclo de vida. Estas herramientas incluyen:

- Control de versiones.
- Seguimiento de defectos.
- Documentación.
- Gestión de proyecto.
- Automatización de procesos.

2.3.1. Listado de herramientas utilizadas

Enterprise Architect: Herramienta Case de diseño y modelado visual basada en UML. La misma fue utilizada para documentar diagramas de casos de uso y su especificación, requerimientos funcionales y no funcionales, diagramas de clase, diagramas de tablas, diagramas de arquitectura y dependencias entre interfaces de usuario. Se utiliza a lo largo de todo el ciclo de vida de desarrollo del software.

GitHub: Plataforma de desarrollo colaborativo de software. Permite alojar proyectos utilizando el sistema de control de versiones de Git. Nos sirve principalmente como

repositorio del código fuente, para hacer un seguimiento de los defectos, para visualizar rápidamente el código de alguna versión antigua y para analizar el pulso del proyecto en base a los gráficos que proporciona. Por otro lado, elegimos Git por su buena integración con el IDE NetBeans. A su vez, instalamos un plugin llamado **Github Issues** con el propósito de poder manejar desde el mimo Netbeans IDE los bugs, improvements e historias de GitHub, haciendo más fácil el trackeo y seguimiento de los mismos.

Microsoft Project (o MSP): Software de administración de proyectos utilizado para asignación de recursos a tareas, dar seguimiento al progreso y analizar cargas de trabajo.

Google Drive y Google Docs: Se utilizaron estos servicios para compartir archivos en la nube entre los miembros del equipo, y, sobre todo, para trabajar de forma comunitaria en la edición diferentes documentos relacionados con el proyecto.

Maven: Es una herramienta que automatiza el proceso de compilación, ejecución de pruebas, gestión de dependencias, armado de los paquetes .jar, e instalación de dichos paquetes. Utiliza un Project Object Model (POM) para describir el proyecto de software a construir (módulos), sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Una característica clave de Maven es que puede dinámicamente descargar plugins de un repositorio que provee acceso a muchas versiones de diferentes proyectos en Java. Esto último nos facilitó la gestión de dependencias en el equipo, y además del empaquetado y despliegue, nos facilitó la ejecución de las pruebas de unidad e integración

NetBeans: Es un IDE que permite trabajar, entre otros lenguajes, con Java. Es de uso libre y proporciona una integración nativa (sin necesidad de plugins) con la mayoría de las otras herramientas asociadas al lenguaje Java: Glassfish, Java EE, Maven, JUnit y Git.

PgAdmin III: Es una herramienta libre de administración para PostgreSQL a través de interfaz gráfica. Fue utilizada para facilitar la el almacenamiento, modificación y extracción de la información en la base de datos y la configuración de la misma.

PostgreSQL: Motor de bases de datos relacional de uso libre.

Java EE 7: Provee un entorno (conjunto de estándares) para desarrollar aplicaciones empresariales. Este concepto incluye muchas tecnologías; las utilizadas en el proyecto fueron:

- Desarrollo de páginas web. JavaServer Faces
- Mapeo objeto-relacional. Java Persistence API
- Desarrollo de aplicaciones multicapa. Enterprise JavaBeans y Contexts and Dependency Injection
- Ejecución de procedimientos batch
- Capa de seguridad

Herramientas de testing: Para las pruebas de unidad e integración son necesarias herramientas que permitan automatizar las pruebas de unidad, realizar mocks de métodos y clases, y correr los test en un entorno similar al del servidor de aplicaciones. Para las pruebas de unidad se utiliza JUnit, para los mocks Mockito, y Arquillian para ejecutar los contenedores donde correrán las pruebas, simulando ser el servidor de aplicaciones.

Selenium WebDriver: Esta herramienta se utilizó con el propósito de automatizar casos de prueba funcionales del sistema y automatización de testing de Interfaz Gráfica web del sistema.

TestNG: Esta herramienta se utilizó para facilitar la configuración de ejecución de los tests automatizados y para tener un informe simple pero de calidad de los resultados de las pruebas funcionales ejecutadas en forma automática. TestNG nos permitía conocer el estado de los diferentes casos de prueba corridos en forma automática y la causa de la falla en caso que alguno no pasase la prueba.

Selenide: Es un framework basado en Selenium WebDriver que envuelve (*wrappea*) los WebElements en SelenideElements proveyendo una mayor flexibilidad y potencia en la automatización de tests automáticos. Además, Selenide simplifica la notación y utilización de los elementos y *locators* en los test automáticos, de forma que el código de los test automáticos es más claro y mantenible.

Glassfish (Payara): Glassfish es un servidor de aplicaciones libre desarrollado por Oracle, que implementa las tecnologías definidas en la plataforma Java EE y permite ejecutar aplicaciones que siguen esta especificación. Hoy es mantenido por otro grupo de desarrolladores, que lo hacen llamar Payara Server.

PrimeFaces 5.3: es un framework que funciona sobre la tecnología web de Java EE, más precisamente sobre JavaServer Faces (JSF). Nos proporciona una serie de componentes web reutilizables, como ser barras de menú, galería de imágenes, selectores de fecha, etc., y lo

más importante, refuerza la tecnología JSF para agregar funcionalidades AJAX y hacerlas más usables.

Herramientas Web: Fueron utilizadas, si bien en menor medida, las siguientes tecnologías: *HTML5* y *JavaScript*. El poco uso de las mismas fue debido a que el Framework web utilizado resuelve la mayoría de las cuestiones. Por otro lado, se hizo un uso más amplio de *CSS3* para obtener una interfaz amigable y un diseño responsivo.

3. Capítulo 3: Desarrollo

3.1. Descripción general del sistema

3.1.1. Arquitectura

El sistema consiste en una aplicación web a la cual los usuarios pueden acceder a través de un navegador web. Las aplicaciones web están generalmente estructuradas en capas para poder manejar su complejidad y hacer una separación de responsabilidades. Además esto brinda la posibilidad de distribuir el sistema físicamente en distintos nodos (de ser necesario) contribuyendo a la escalabilidad del mismo. A continuación se presenta un diagrama de alto nivel de nuestro sistema:

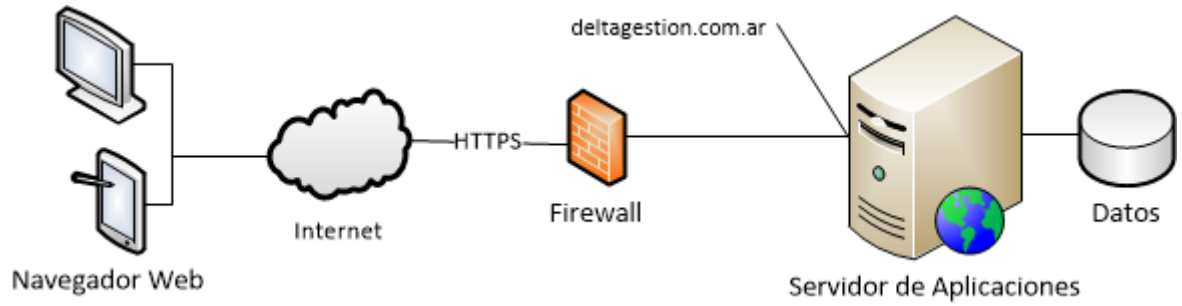

Ilustración 1.4: Los navegadores web se encuentran en móviles y computadoras, y nos permiten interactuar con el sistema. La encriptación de la comunicación y el firewall proveen la seguridad necesaria en el ambiente inseguro que es Internet, mientras que el servidor y los datos soportan el funcionamiento de la aplicación.

El sistema es utilizado por usuarios que, mediante un navegador web, envían solicitudes al servidor. El servidor recibe estas solicitudes, las procesa a través de sus capas, y devuelve la respuesta correspondiente. Si la solicitud o petición es aceptada (ya que, por ejemplo, podría ser rechazada por cuestiones de seguridad), el servidor comienza el procesamiento haciendo las llamadas necesarias a la capa web. La capa web se encarga recibir solicitudes *http* y generar la página (o recurso) que se requiere. Para generar la respuesta, es necesario procesar los datos ingresados por el usuario y datos almacenados, según una cierta lógica de negocio. La capa de negocio provee este servicio. Como se mencionó, es necesario acceder a los datos almacenados, aquí es donde la capa de datos entra en juego para hacer consultas y modificaciones en la base de datos. A partir de este momento las capas comienzan a responder en sentido inverso, hasta que finalmente la capa web genera la respuesta requerida, renderizando en la mayoría de los casos, una página web. En la sección 3.4.2 Diseño de Arquitectura justificamos y ampliamos los aspectos arquitectónicos.

3.1.2. Funcionalidades

3.1.2.1. Características generales

- Inicio de sesión de usuarios. Los recursos o páginas están protegidos. Esto quiere decir que si el usuario no está autenticado no puede acceder a ellas. Al intentar acceder a un recurso protegido, el sistema solicitará el ingreso de usuario y contraseña y, en caso de ser correctos, redirigirá hacia la página a la cual se quería acceder.
- Alertas y notificaciones. El sistema muestra un mensaje emergente de carácter informativo ante eventos claves en la gestión diaria del consultorio, como por ejemplo, fechas de entrega de planillas por vencer, órdenes médicas adeudadas, tratamientos por finalizarse, topes de sesiones en el año para un paciente, entre otros.
- Seguridad de los datos. Se procura que el servicio de *hosting* del sistema esté entre los más renombrados servicios de *cloud computing*, los cuales cuentan con centros de datos de alta seguridad. Los servicios de *cloud computing* implementan las mejores prácticas de seguridad, dan la posibilidad de replicar físicamente y encriptar los datos, garantizan la no divulgación del contenido, entre otras sólidas características.
- Privacidad de la información. Los datos que gestiona el sistema son datos sensibles, que pueden llegar a transmitirse por redes inseguras. Existe la posibilidad de que las comunicaciones sean *escuchadas* y se capturen los datos. Para mitigar este riesgo el sistema encripta el tráfico web.
- Diseño web adaptado para dispositivos móviles y tablets. El diseño web *responsive* o adaptativo es una técnica de diseño web que busca la correcta visualización de una misma página en distintos dispositivos, desde ordenadores de escritorio a tablets y móviles. Se trata de redimensionar y colocar los elementos de la web de forma que se adapten al ancho de cada dispositivo permitiendo una correcta visualización y una mejor experiencia de usuario.
- Gráficos estadísticos: El sistema presenta una sección en la cual se pueden visualizar datos estadísticos de interés para la toma de decisiones, como por ejemplo, cuáles son los meses en los que concurre mayor cantidad de pacientes, qué obras sociales son las que tienen mayor cantidad de asociados en el consultorio, etc.

3.1.2.2. Agenda

La Agenda es una de las funcionalidades más importantes, siendo ésta el punto de partida del sistema. El usuario puede agregar sesiones a la agenda para un día y horario específico picando en la región deseada, o bien arrastrar una sesión existente para re-programarla. Además el usuario puede seleccionar y visualizar los detalles de una sesión,

navegar hacia el tratamiento del que forma parte, agregar días feriados y remarcarlos, y ver la distribución de sesiones del día, la semana o el mes. Entre otras características a mencionar, las sesiones de distintas especialidades (tipos de tratamiento) son distinguidas con colores, y los posibles estados de una sesión (pendiente, transcurrida, etc.) se identifican con íconos.

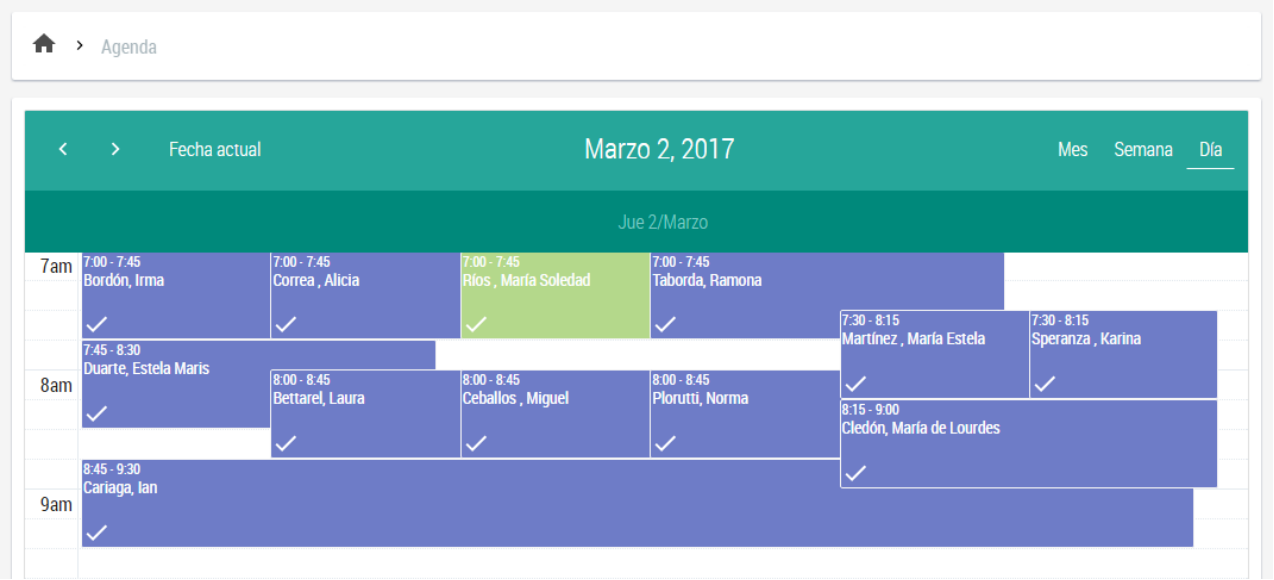

Ilustración 1.5: Vista diaria de la agenda.

3.1.2.3. Pacientes

Para gestionar los pacientes del consultorio, el sistema permite agregar, eliminar y editar datos de los mismos. La cantidad de pacientes que asisten diariamente y cuyos datos quedan guardados en el sistema, genera la necesidad de contar con un mecanismo de búsqueda. El sistema aporta esta característica permitiendo encontrar un paciente de forma rápida y, de esta manera, agilizar las tareas del usuario. Para facilitar la comunicación con los pacientes, los datos de los mismos son sincronizados con los contactos de la cuenta de GMail del cliente.

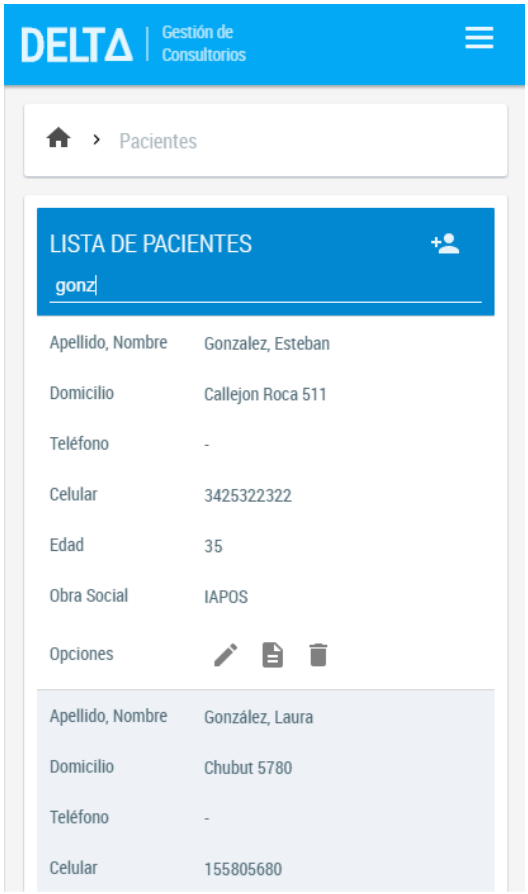

Ilustración 1.6: Búsqueda de pacientes desde un dispositivo móvil.

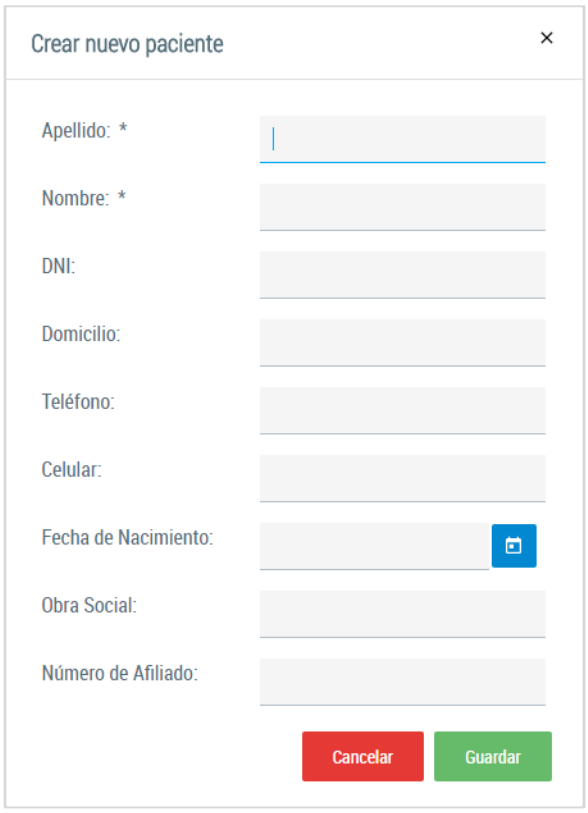

Ilustración 1.7: Formulario de creación de un nuevo paciente.

3.1.2.4. Tratamientos

La historia clínica de cada paciente está representada en el sistema mediante una serie de tratamientos. El usuario puede cargar los tratamientos que un paciente va realizando en el consultorio, especificando el diagnóstico, qué cantidad de sesiones tendrá, el tipo de tratamiento realizado, entre otros datos.

El listado de tratamientos de cada paciente nos permite tener una rápida noción de su historia kinésica e identificar qué tratamiento se encuentra realizando actualmente. Además nos brinda información acerca de cuántas sesiones lleva realizadas para el tratamiento en curso, nos permite identificar rápidamente el diagnóstico, saber si se trata de un tratamiento particular o no, entre otras cuestiones que permiten al kinesiólogo realizar el seguimiento del paciente y sus tratamientos.

Asociado a la gestión de tratamientos, el sistema soporta otras funcionalidades, como la impresión de la planilla de consentimiento informado que debe firmar el paciente al iniciar un tratamiento y la carga de imágenes referentes a estudios médicos.

3.1.2.5. Sesiones

La gestión de sesiones relaciona la agenda con los pacientes y sus tratamientos. El sistema soporta el alta, baja y modificación de sesiones, tanto desde la agenda como desde la vista de tratamientos de un paciente.

Cuando creamos una sesión desde la agenda, el sistema facilita la selección del paciente mediante campos con autocompletado y configura la fecha y hora dependiendo de qué rango horario y fecha se haya seleccionado en la agenda. Puede ocurrir que al momento de dar el alta de una sesión el paciente no exista previamente en el sistema. Esto ocurre por lo general cuando se dan turnos por teléfono a pacientes nuevos. En tal caso el sistema muestra en la misma pantalla un *popup* para la carga rápida del paciente, solicitando los datos mínimos necesarios para dar de alta el mismo.

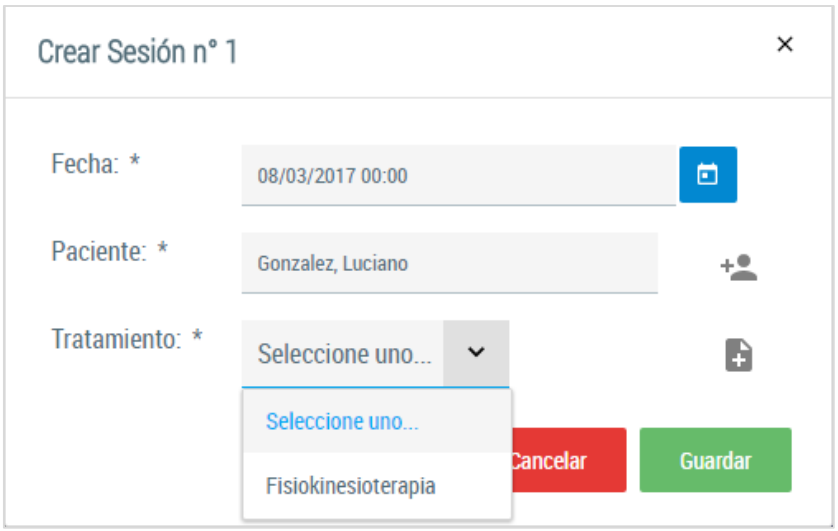

Ilustración 1.8: Formulario de creación de una nueva sesión. Los íconos a la derecha de los campos Paciente y Tratamiento nos permiten dar de alta nuevos pacientes y tratamientos desde la misma pantalla de alta de sesión.

Por lo general, los pacientes prefieren asistir en horarios y días regulares, es decir, repetir días y horarios entre semanas. En estos casos la carga de sesiones se torna engorrosa ya que el usuario debe dar de alta una sesión a la vez para el mismo paciente y tratamiento hasta completar el total requerido de sesiones. En el peor de los casos, se da de alta una sola sesión, postergando la carga de las restantes sesiones. Esto implica pérdida de tiempo extra asociado a tener que comunicarse con el paciente para coordinar nuevos turnos. La carga masiva de sesiones, en una única vez, permite dar de alta la cantidad deseada de sesiones repitiéndolas en el tiempo para los días y horarios configurados.

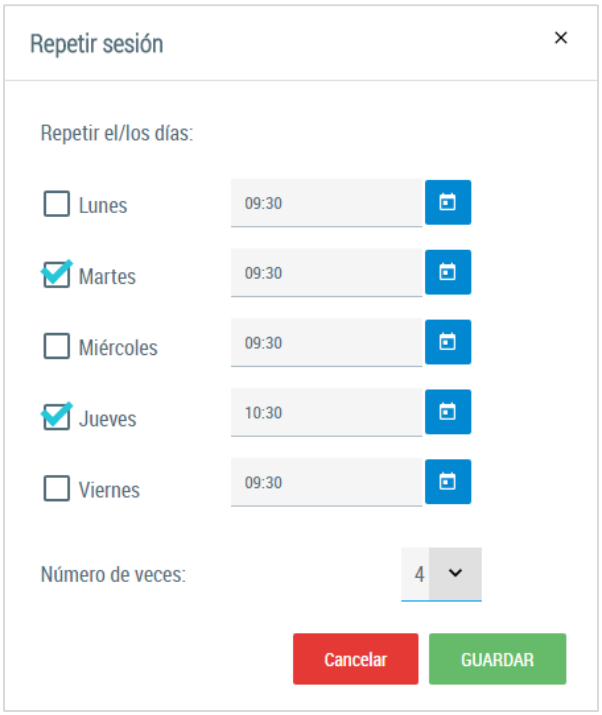

Ilustración 1.9: Carga masiva de sesiones mediante la posibilidad de repetir una sesión en ciertos días y horarios

Cada día tiene un conjunto de sesiones. Al final de cada día el sistema asume que las sesiones han transcurrido y actualiza el estado de las mismas. Esto permite el seguimiento automático del estado de los tratamientos. En cada instante el avance del tratamiento está representado por la cantidad de sesiones que transcurrieron del mismo con respecto al total. Cuando el tratamiento llega al final el sistema lo marca como finalizado. También existe la posibilidad de finalizar el tratamiento manualmente.

En los casos en que el paciente no asista a una sesión ésta puede configurarse desmarcando la opción *Cuenta*. Luego estas sesiones serán ignoradas a la hora de hacer el seguimiento del tratamiento y el cálculo de órdenes médicas necesarias.

Otra funcionalidad asociada a la gestión de sesiones es la posibilidad que ofrece el sistema de enviar recordatorios a los pacientes sobre sus turnos. Se hace mediante WhatsApp a aquellos pacientes que tengan configurado su número de celular y posean este servicio de mensajería.

3.1.2.6. Obras Sociales

En el sistema se encuentran cargadas todas las obras sociales con las cuales trabajan los kinesiólogos adheridos al Colegio de Kinesiólogos de la Provincia de Santa Fe. Es necesario que el sistema gestione algunos aspectos relacionados con las obras sociales.

Desde el apartado de configuración el usuario puede mantener actualizada la relación Obra Social - Tipo de Tratamiento. Para cada caso se puede configurar el código de prestación y si requiere de autorización o no. Es importante proporcionarle al usuario esta configuración debido a la gran cantidad de combinaciones que pueden surgir entre Obras Sociales y Tipos de Tratamientos; cada caso es particular. De esta manera, se van gestionando bajo demanda las necesidades del usuario.

3.1.2.7. Órdenes médicas

El objetivo principal de la gestión de las órdenes médicas por parte del sistema es que el usuario pueda generar fácilmente la planilla que debe presentar regularmente al Círculo de Kinesiólogos. Es por esto que se brinda la posibilidad de cargar órdenes médicas, configurando la cantidad de sesiones que cubrirán. La carga de órdenes se hace sobre un determinado tratamiento de un paciente y la cantidad de sesiones que cubre no puede sobrepasar el número de sesiones configuradas para el tratamiento.

El sistema permite llevar un control de la cantidad de sesiones cubiertas por las órdenes médicas. Lo mencionado anteriormente se puede explicar mejor con el siguiente ejemplo: Si un tratamiento tiene cargadas 10 sesiones, el sistema va a controlar que el mismo también tenga cargadas órdenes que cubran esas sesiones (por ejemplo, que tenga cargada 1 orden de 10 sesiones ó 2 órdenes de 5 sesiones cada una). El control se realiza validando que al finalizar un tratamiento, el mismo tenga órdenes que cubran las sesiones cargadas.

En caso de que el tratamiento no tenga órdenes cargadas que cubran la cantidad de sesiones del mismo, el sistema emitirá una notificación indicando que el paciente debe órdenes. Esto ayuda al profesional a hacer un seguimiento de los pacientes que le deben órdenes para exigirles las mismas.

Como se ve en la planilla de ejemplo, las órdenes médicas tienen asociado un código de autorización. Una vez que el usuario cuenta con dicho código, lo carga en el sistema mediante la opción de *Autorizar orden*.

Obra Social	Cód. de Autorización	N° de Afiliado	Código	Cantidad
1 IAPOS - ACCIDENTE DE TRABAJO [illegible]	092585	26093531	250101 - 250102	10
2 IAPOS [illegible]			25.01.01-25.01.02	5
3 OSDE [illegible]	34563231	62428673-7-02	250181	10
4 AMUR [illegible]	AU-014252	60254/01	250101-250102	10
5 IAPOS [illegible]	A04-O23-G54 (15/3)	5598335	25.01.01-25.01.02	5

Ilustración 1.10: Ejemplo de planilla de órdenes médicas emitida por el sistema, conforme a los requerimientos del Círculo de Kinesiólogos.

A la hora de generar la planilla, el usuario puede requerir agrupar y ordenar los datos (órdenes médicas) de diferentes maneras. Por ejemplo, ordenar las órdenes médicas por obra social, o filtrar por una determinada obra social y tipo de tratamiento. Luego que el usuario configura el listado de órdenes médicas como desea, puede realizar la impresión del mismo. Una vez impresa la o las planillas para un determinado período, el sistema considera que las órdenes incluidas en dichas planillas están presentadas al Círculo de kinesiólogos. Esto se logra modificando el estado de las órdenes médicas, para que en el período siguiente, al momento de generar la nueva planilla, se listen por defecto todas aquellas órdenes que no están presentadas aún.

La pantalla desde la cual se puede generar la planilla también cumple la función de búsqueda y seguimiento de las órdenes médicas. Por ejemplo, permite filtrar las órdenes por fecha y visualizar sólo aquellas que no fueron autorizadas y son de una determinada obra social.

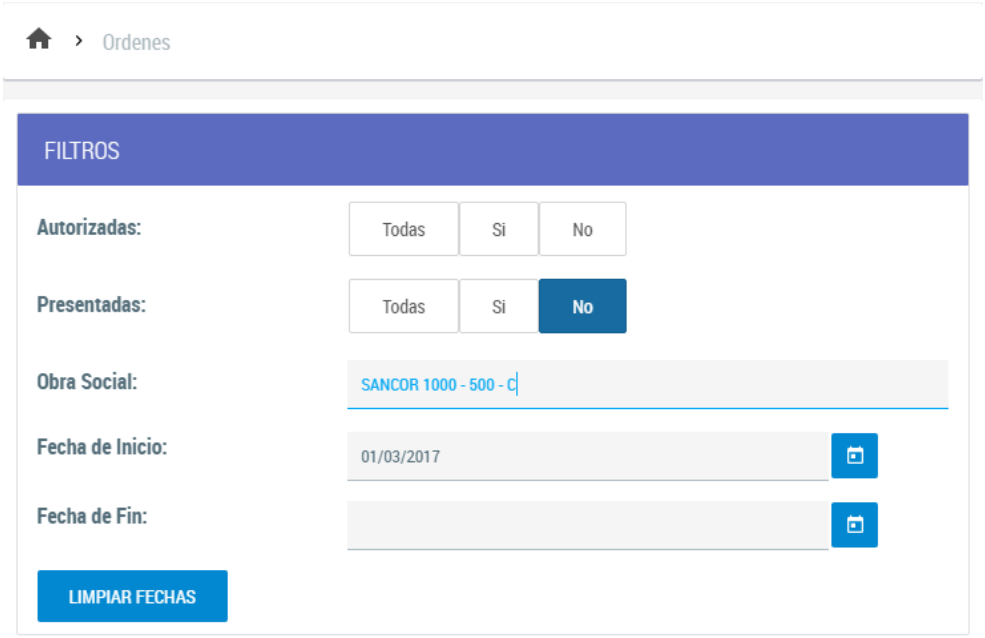

Ilustración 1.11: Sección de filtros dentro de la pantalla de órdenes médicas. Los componentes están adaptados a la resolución de una tablet.

3.1.2.8. Caja

Permite cargar ingresos y egresos de efectivo seleccionando el concepto asociado. Los conceptos se pueden ir dando de alta a medida que el usuario los necesite. Por ejemplo cuando un paciente paga un estampillado, se carga un ingreso con el concepto “Estampilla”.

Los movimientos pueden listarse filtrando por fecha y concepto.

3.2. Elección de las tecnologías y herramientas

Una de las primeras y más determinantes decisiones tuvo que ver con qué tipo de aplicación íbamos a hacer. ¿Una aplicación standalone, móvil, o un sistema web? Cada opción tiene ventajas y desventajas, que fueron analizadas teniendo en cuenta las características del proyecto.

Una característica importante es que se trata de un sistema de gestión. Por lo general, este tipo de sistemas requiere de mucha entrada de datos, visualización de reportes y tablas, generación e impresión de informes, entre otras funcionalidades. Una aplicación móvil quizás no de soporte a estos requerimientos de la mejora manera.

Por otro lado buscamos innovar en el dominio que resuelve el sistema, y queríamos incorporar otros atributos de calidad. Un sistema web nos pareció apropiado para lo que buscábamos, mientras que un sistema de escritorio o standalone no en la misma medida. Los sistemas web tienen ciertas características que hicieron que nos inclinemos por su elección:

- Las aplicaciones web son típicamente descritas como aplicaciones multiplataforma, ya que pueden accederse desde un navegador que corra sobre cualquier sistema operativo. Contando con esta ventaja, si sumamos a la aplicación la capacidad de adaptarse a la pantalla de los distintos dispositivos (diseño responsive), la misma puede accederse desde un smartphone, una tablet o una PC.
- Son fáciles de mantener actualizados: Existe solo una versión de la aplicación web en el servidor, por lo que no hay que distribuirla entre otros ordenadores si los hay. El proceso de actualización es rápido y limpio. Las aplicaciones basadas en web no requieren que el usuario se preocupe por obtener la última versión, ni interfieren en su trabajo diario para descargar, instalar y configurar últimas versiones.
- Es posible instalarlos y que funcionen sobre algún servicio de *cloud computing*, eliminando la necesidad de que el cliente tenga que mantener equipos físicos, con todo lo que esto implica (seguridad, disponibilidad, conectividad, etc.).
- Facilitan el trabajo colaborativo y a distancia: Las aplicaciones web pueden ser usadas por varios usuarios al mismo tiempo. Al estar toda la información centralizada no es necesario compartir pantallas o enviar emails con documentos adjuntos. Además son accesibles desde cualquier lugar. Por ejemplo, en nuestro caso la secretaria puede agregar las órdenes por un lado y por otro lado el profesional puede usar estas órdenes para imprimir el reporte que debe presentar en el Colegio de Kinesiólogos.
- Datos más seguros: El usuario no deberá preocuparse de posibles rupturas del disco duro ni de los virus que pueden hacerle perder toda la información. Los proveedores de hosting donde se almacenan las aplicaciones usan “granjas de servidores”, con altísimas medidas de seguridad, donde guardan los datos de forma redundante y con amplios servicios de backups.

Una vez escogido el tipo de sistema a realizar se comenzó a pensar en la tecnología a utilizar. Los sistemas web involucran muchos conocimientos y conceptos que en su mayoría resultaban nuevos para el equipo. Teníamos en común conocimientos base sobre Java, en su versión SE (Standard Edition), por lo que fue el primer camino de búsqueda.

Comenzamos explorando qué opciones ofrecía Java en el terreno de las aplicaciones empresariales y web. Nos encontramos con un conjunto de módulos (API) especificado y estandarizado por la comunidad Java que conforman Java EE en su versión 7, lanzada en 2013.

Las versiones anteriores (Java EE 6 y 5) se perciben como complejas a la hora de desarrollar, y algunas funcionalidades (como un framework web, módulo de persistencia, o inyección de dependencia) no están dentro de su especificación. Muchos frameworks de terceros (Spring, Struts, Vaadin) cubren estas necesidades, y cada producto tiene sus ventajas, desventajas, múltiples versiones y problemas de compatibilidad. Java EE 7 viene a solucionar esto actualizando muchos de sus módulos e incorporando otros, para estar a la altura de los frameworks más utilizados del mercado.

Java EE está compuesto por diversos módulos. A continuación describimos aquellos que usamos:

JSF (JavaServer Faces): Forma parte del estándar de Java EE y consiste en la especificación para desarrollar interfaces de usuario en aplicaciones web. Es un framework web basado en componentes. Junto a esta tecnología usamos la librería de componentes para JSF que provee PrimeFaces.

CDI (Contexts and Dependency Injection): Permite usar *inyección de dependencia* sin que el desarrollador tenga que implementar por su cuenta los métodos y constructores necesarios para establecer las dependencias entre objetos. Un framework que provee inyección de dependencias simplifica la inicialización de las clases con los objetos correctos.

EJB (Enterprise JavaBeans): Permite desarrollar componentes de software (Enterprise JavaBeans) que encapsulan la lógica del negocio permitiendo modularizar el sistema. Los EJB son componentes del lado del servidor, y su especificación pretende cubrir las siguientes responsabilidades:

- Procesamiento transaccional
- Scheduling de trabajos
- Control de concurrencia
- Implementación de web services
- Seguridad (autenticación y autorización)
- Integración con JPA

JPA (Java Persistence API): Es la API de persistencia desarrollada para la plataforma Java EE. El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de

mapeo objeto-relacional) y permitir usar objetos regulares (conocidos como POJOs). El mapeo objeto-relacional (es decir, la relación entre entidades Java y tablas de la base de datos, queries con nombre, etc) se realiza mediante anotaciones en las propias clases de entidad.

Batch Processing: Dentro de Java EE está definido un modelo de programación para aplicaciones batch y un entorno de ejecución (incluido en el servidor de aplicaciones) para administrar, ejecutar y programar la ejecución de trabajos batch. Los procesamientos batch son por lo general orientados a manipular grandes cantidades de información, son de larga duración y pueden requerir ejecutarse periódicamente.

Seguridad: No hay un único aspecto de seguridad a resolver, por lo tanto se emplearon varios mecanismos. Para la seguridad de la aplicación, Java EE nos proporciona un mecanismo robusto para autenticar usuarios y autorizar acceso a las funciones de la aplicación; por otro lado a nivel de transporte el mecanismo que nos provee Java depende y hace uso del HTTPS.

En la imagen siguiente pueden verse las tecnologías de Java EE 7 organizadas en capas. Es sólo una representación lógica, pudiendo luego usar cada tecnología en la capa de nuestra arquitectura donde la necesitemos.

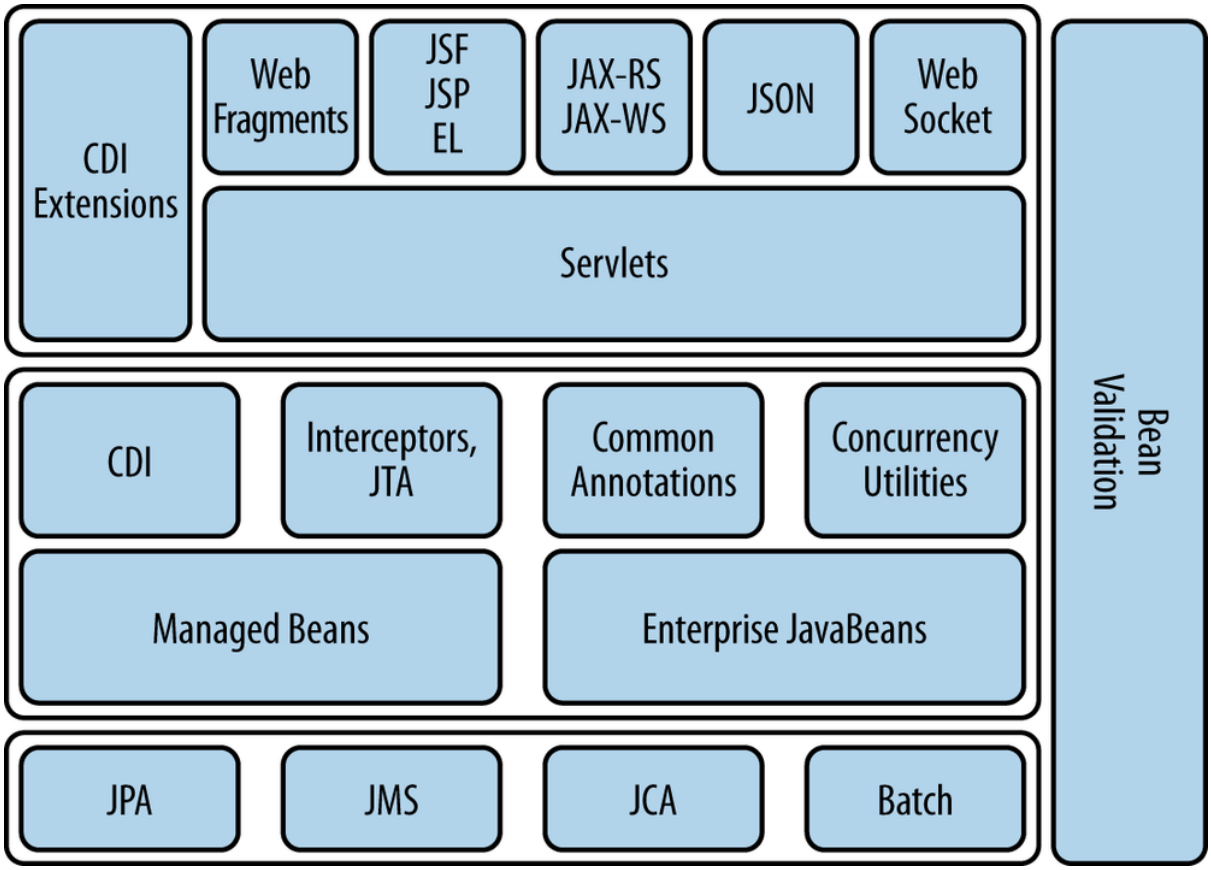

Ilustración 1.12: Las tecnologías de Java EE 7 pueden organizarse en capas. La capa superior contiene tecnologías web, la capa intermedia tecnologías orientadas a servicios y lógica de negocio, y finalmente encontramos la capa orientada a la gestión de datos.

Más adelante en el informe vemos cómo las tecnologías se relacionan con el diseño arquitectónico.

3.3. Obtención de requerimientos

Sommerville [6] habla de un proceso estructurado para el manejo de requerimientos, formado por tres grandes grupos de actividades:

1. Adquisición y análisis de requerimientos: “En esta actividad los ingenieros de software trabajan con clientes y usuarios finales del sistema para descubrir el dominio de aplicación, qué servicios debe proporcionar el sistema, el desempeño requerido de éste, las restricciones de hardware, etc”. Dentro de esta actividad se encuentran las tareas de descubrimiento de requerimientos, clasificación y priorización de los mismos.
2. Especificación de requerimientos: “La especificación de requerimientos es el proceso de escribir, en un documento de requerimientos, los requerimientos del usuario y del sistema. De manera ideal los requerimientos deben ser claros, sin ambigüedades, fáciles de entender, completos y consistentes”.

3. Validación de requerimientos: “La validación de requerimientos es el proceso de verificar que los requerimientos definan realmente el sistema que quiere el cliente. Durante el proceso de validación de requerimientos tienen que realizarse diferentes tipos de comprobaciones sobre los requerimientos contenidos en el documento de requerimientos.”

Estas tres grandes actividades contienen diferentes tipos de tareas a realizar en cada fase de la metodología. Decidimos que algunas de ellas nos servirían de guía para obtener los requerimientos del sistema, por lo cual tomamos los principios mencionados por Sommerville [6] y los pusimos en práctica. A continuación definiremos lineamientos generales que tomamos respecto a los requerimientos del sistema para luego indicar qué tareas realizamos en las fases de Elaboración y Construcción del sistema.

3.3.1. Lineamientos generales de Adquisición de Requerimientos

3.3.1.1. Uso de entrevistas

Para el proceso de adquisición de requerimientos decidimos que la mejor estrategia para realizarlo era mediante entrevistas, ya que la misma permite recabar una cantidad mayor de información respecto a lo que permite un cuestionario. Si bien esto podía ser contraproducente –el tener mucha información significaba analizar una cantidad mayor de datos–, nos permitía entrar en contacto con el vocabulario y el dominio específico, al mismo tiempo que le permitía al cliente expresarse y sentirse cómodo.

En las primeras entrevistas que tuvimos, predominaron las preguntas abiertas, ya que no teníamos mucho conocimiento del dominio específico ni de cómo se llevaban adelante las actividades cotidianas en el consultorio. Conforme fue pasando el tiempo y el análisis avanzaba, las preguntas se fueron volviendo preguntas más cerradas o puntuales, de forma de refinar los requerimientos que íbamos recabando.

No obstante, antes de cada entrevista nos reuníamos entre todos y llevábamos a cabo una serie de tareas, que Kendall & Kendall [8] describen como actividades de preparación para una entrevista:

1. “Establecer objetivos de la entrevista”.

Si bien el fin principal de toda entrevista es la obtención de información, nos preguntábamos ¿qué es lo que queremos lograr con esta entrevista? En la primera entrevista, por ejemplo, el objetivo era aprender del dominio; ¿qué hacía el usuario?, ¿cómo lo hacía?,

¿en qué consistía un día habitual en el trabajo?. En una segunda instancia, la entrevista apuntaba a cosas más específicas, como tipos de tratamientos, obras sociales, turnos, documentos manejados. Según Kendall & Kendall [8] existen algunos puntos que suelen tocarse en las entrevistas, y a los cuales apuntamos en las diferentes sesiones con el usuario: "fuentes de información, formatos de información, frecuencia en la toma de decisiones, cualidades de la información y estilo de la toma de decisiones".

2. "Decidir a quién entrevistar"

Dada las características del consultorio, decidimos entrevistar a uno de los dos profesionales que trabajan en él. Como ambos realizan tareas similares, decidimos entrevistar a uno de ellos en forma sistemática mientras que al otro le realizábamos consultas específicas cuando necesitábamos validar un requerimiento que nos generaba dudas. De esta forma acordamos con uno de los profesionales la responsabilidad de realizar las entrevistas para la obtención y refinamiento de los requerimientos.

3. "Preparar al entrevistado"

Esto hace referencia a comunicarse por anticipado con la persona que va a ser entrevistada para que ésta se haga un tiempo para la entrevista y se mentalice acerca de la misma. En nuestro caso las entrevistas eran acordadas con, a lo sumo, una semana de anticipación, de forma que pudiésemos encontrar un horario que al entrevistado le fuese cómodo. Además, en caso de necesitar cierta información le anticipábamos de ellos para que, en lo posible, para el día de la entrevista contase con esa información. Por ejemplo, en las reuniones donde necesitábamos revisar los documentos (planillas, informes) que el cliente utilizaba.

4. "Decidir el tipo de preguntas y la estructura"

A esto nos referíamos cuando hablábamos de utilizar preguntas abiertas en las primeras reuniones e ir virando a preguntas más cerradas conforme avanzaban las reuniones con el entrevistado. Respecto a la estructura utilizada Kendall & Kendall [8] proponen tres enfoques diferentes para organizar las preguntas dentro de una entrevista: pirámide (se comienza por preguntas cerradas y luego se avanza hacia preguntas más abierta), embudo (se comienza por preguntas abiertas y luego se avanza hacia preguntas más cerradas) o diamante (una concatenación de pirámide y embudo). En nuestro caso, decidimos no restringirnos a un formato específico: podría decirse que las entrevistas comenzaban con un formato de embudo, pero luego de llegar a cuestiones específicas volvíamos a ir hacia preguntas más

generales y luego específicas otra vez. Priorizamos un orden lógico en las preguntas a realizar que la especificidad de las mismas.

3.3.1.2. Escenarios de uso

En general las entrevistas realizadas eran utilizadas para discutir puntos de diversa índole. Sin embargo, siempre que era posible tratábamos de utilizar escenarios (casos) de uso para obtener requerimientos. Esta metodología nos resultó sumamente útil, ya que nos permitía encontrar requerimientos que el cliente daba por sentado, dado que lidiaba con ellos en forma diaria. Además nos permitía encontrar distintos casos de excepción. Ejemplos de estas preguntas:

- ¿Qué sucede si un paciente entrega una orden médica que sobrepasa la cantidad de sesiones del tratamiento?
- ¿Cómo registra un tratamiento si el paciente aún no está registrado en la lista de pacientes de la clínica?
- ¿Cómo registraría movimiento de dinero de la venta de tapping si el pago no se realiza en el mismo momento en que se le entrega dicho elemento al paciente?

Además de los beneficios que tenía para nosotros, utilizar escenarios o casos de uso era también beneficioso para el cliente, ya que utilizábamos un vocabulario común y con el cual estaba familiarizado.

3.3.1.3. Cuándo pactar reuniones con el cliente

Otro de los puntos que decidimos acordar desde el principio fue con qué periodicidad o ante qué circunstancias debíamos acordar una reunión con el cliente. Si bien por parte del cliente no teníamos oposición o rechazo a las reuniones, decidimos que era mejor restringir la cantidad de encuentros al mínimo posible.

Por lo tanto, la idea inicial era pactar una entrevista cuando estuviésemos próximos a terminar de realizar un análisis completo de los datos que teníamos. Esto implicaba una revisión de la información recopilada en la entrevista anterior, especificación de requerimientos, elaboración de casos de usos, elaboración de *mockups* (o prototipos de interfaz de ser posible) y elaboración de casos de pruebas. Luego de realizar todas estas actividades, cualquier duda que nos surgiera quedaba registrada en un documento de notas con preguntas para hacerle al cliente. Luego de revisar dichas preguntas y darle cierta organización y refinamiento, este documento servía de cuestionario para la siguiente entrevista.

Si durante el proceso de análisis encontrábamos algún problema que necesitábamos resolver de inmediato (por ejemplo, un requerimiento clave que era decisivo para el diseño de una funcionalidad), lo que hacíamos era contactar al cliente por email. Esto le brindaba al cliente la comodidad de contestar cuando tuviese tiempo y a nosotros la comodidad de obtener las respuestas que requeríamos sin la necesidad de invertir tiempo en otra entrevista.

3.3.2. Lineamientos generales de Especificación de Requerimientos

3.3.2.1. El formato de los requerimientos

Dado que los requerimientos pueden ser escritos de diferentes formas por diferentes personas, lo primero que hicimos antes de comenzar a especificar formalmente los requisitos funcionales del sistema fue ponernos de acuerdo en la forma en que íbamos a escribirlos.
Para esta actividad decidimos seguir los lineamientos que proponen Arlow y Neustadt [7], y utilizar un formato como el que sigue:

<id> El <sistema/actor> debe(ría) <hacer algo>

De esta forma sencilla acordamos un formato estándar para los requerimientos del sistema a ser escritos. Las partes del enunciado de un requerimiento fueron:

- "Id": es un código único de identificación del requerimiento.
- "Sistema/Actor": hace referencia al sistema, parte del sistema o actor al cuál refiere el requerimiento. Si bien normalmente hablamos del "sistema", en ocasiones queremos que ciertas partes del sistema tengan alguna característica funcional particular. Por ejemplo: "La agenda de turnos debe poder verse en escala temporal diaria o semanal".
- "Hacer algo": esta es la parte del enunciado donde hablamos de la característica que debe tener el sistema/actor; normalmente contendrán verbos como "tener", "haber", "mostrar" o "permitir". Además de la acción en sí, en esta parte del requerimiento se incluyen detalles adicionales; por ejemplo, un requerimiento "El sistema debe permitir asignar distintas duraciones a los turnos de acuerdo al tipo de tratamiento" no solo indica que el sistema debe permitir asignar distintas duraciones a los turnos, sino que también el criterio a seguir para hacerlo.

3.3.2.2. Prioridad de los requerimientos

Hasta aquí teníamos una forma común de escribir los requisitos del sistema. Sin embargo, una vez escritos los mismos, necesitábamos alguna información adicional que nos diera una idea de la urgencia o importancia de cada uno de ellos.

Para contar con este recurso, decidimos utilizar lo que Arlow y Neustadt [7] mencionan como “Atributos de Requisitos”. “Todo requisito puede tener un conjunto de atributos que captura información adicional (metadatos) sobre el requisito. […] Quizás el atributo de requisito más importante es la prioridad. El valor de este atributo expresa la prioridad del requisito relativa a todos los otros requisitos”. Además, propone cuatro niveles de prioridad:

- “Debe” para requisitos obligatorios que son fundamentales para el sistema.
- “Debería” para requisitos importantes que pueden omitirse.
- “Podría” para requisitos que son opcionales.
- “Quiere” para requisitos que pueden esperar para versiones posteriores del sistema.

En este punto decidimos tomar esa clasificación como punto de partida, pero la simplificamos removiendo categorías y cambiando la semántica de las categorías que quedaron. Para la semántica de las categorías remanentes nos pusimos de acuerdo con el cliente, de forma tal de estar hablando en un terreno común. La simplificación fue la siguiente:

- “Debe” para requisitos obligatorios fundamentales para el sistema o requisitos no necesariamente esenciales, pero que deben incluirse sí o sí.
- “Debería” para requisitos opcionales para el sistema o requisitos que sería bueno incorporar pero que no son obligatorios.

De esta forma contamos con una rápida separación de los requisitos obligatorios de los requisitos opcionales.

3.3.2.3. Organización de requerimientos

El último lineamiento que seguimos a la hora de especificar requerimientos funcionales fue el de organizarlos de alguna manera conveniente de forma tal que:

- A simple vista pudiésemos saber a qué se referían distintos subconjuntos de la lista de requerimientos.
- Fuese más fácil manejarlos. Por ejemplo, ver si un requerimiento ya existe, agregar y borrar requisitos fuese más fácil, etc.

Por dichos motivos lo que decidimos fue agrupar los requerimientos de acuerdo a las partes del sistema a las cuales hacían referencia. De esta forma, si quisiéramos agregar un

requisito respecto al manejo de pacientes del sistema, bastaría con revisar el conjunto de requerimientos que hacía referencia a dicha funcionalidad.

Además de agrupar los requerimientos, para requerimientos largos o con muchas condiciones, decidimos separarlos en varios requerimientos. Sin embargo, como podía que varios requerimientos respondieran a un mismo propósito, decidimos enunciar un requerimiento corto y, como sub-requerimientos (en un nivel inferior) los requerimientos más específicos que de éste se desprendían. Así pues, los identificadores de los requerimientos venían dados por el formato *"#"* para requerimientos de un nivel y *"#.#"* para requerimientos de segundo nivel (éste fue el nivel máximo de requisitos alcanzado).

3.3.3. Lineamientos generales de Validación de Requerimientos

Sommerville [6] enuncia la validación de requerimientos como "el proceso de verificar que los requerimientos definan el sistema que en verdad quiere el cliente". Para lograr este acometido propone tres técnicas que nosotros utilizamos como proceso de validación de requerimientos en las fases de Elaboración y de Construcción del sistema.

3.3.3.1. Revisiones de requerimientos

Sommerville [6] describe esta actividad como el "proceso de revisar los requerimientos sistemáticamente usando un equipo de revisores que verifican errores e inconsistencias". En nuestro caso incorporamos al cliente como revisor de la lista de requerimientos, de forma que la revisión además de validar los requisitos funcionales nos permitía llegar a un acuerdo con el cliente acerca de qué era necesario y qué no para el sistema de información.

En este punto resultó muy útil haber agrupado los requerimientos en partes del sistema, dado que permitió separar la revisión de los requisitos en revisiones de las sub-listas de requerimientos para cada sección del sistema.
En esencia el proceso de revisión seguido era algo similar a lo que se enuncia a continuación:
Al nivel de listas de requerimientos:

1. Se hace una revisión de sus requerimientos.
2. Si la lista necesita ser actualizada se actualiza como sigue:
    a. Se borran los requerimientos que no deben ser incluidos.
    b. Se modifican los requerimientos que deben modificarse.
    c. Se especifican los requerimientos en sub-requerimientos según corresponda.
    d. Se agregan nuevos requerimientos.

e. Se vuelve al paso 1.

3. De lo contrario, se termina el proceso de revisión.

Para cada lista de requerimientos:

1. Si hay requerimientos por analizar, se toma el siguiente de la lista y se analiza cómo sigue (cualquier cambio se anota a un costado para realizarlo luego de terminar con todos):
    a. Analizamos si el cliente lo considera relevante:
        i. Si es de suma importancia contar con el mismo, se lo marca como un requerimiento del tipo "Debe" en caso de no estarlo.
        ii. Si es una funcionalidad importante de tener pero no vital, se lo marca como un requerimiento del tipo "Debería" en caso de no estarlo.
        iii. Si es una funcionalidad que sería interesante tener si hay tiempo, o una menor, se lo marca para ser borrado.
    b. Analizamos si el requerimiento puede ser completado:
        i. Si hay algún detalle importante que se omitió, se lo marca al margen.
        ii. Si el requerimiento lo necesitase, se marca para el requerimiento para división, enunciando los sub-requerimientos en los cuales se divide.
    c. Analizamos si existen casos opuestos al requerimiento:
        i. Si existen preguntamos al cliente que espera del sistema ante cada situación.
        ii. Preguntamos cuán importante es manejar cada una de esas situaciones.
        iii. Si corresponde, se marcan al margen requerimientos a agregar.
2. Si no hay requerimientos por analizar, revisamos la lista una vez más para detectar requerimientos que han sido omitidos:
    a. Si existen, se los enuncia y se vuelve al paso 1 tomando estos requerimientos como requerimientos por revisar.
3. Si ya no hay requerimientos por analizar ni por agregar, se toman todos los cambios anotados y se los efectúa sobre la lista de requerimientos del documento.

3.3.3.2. Creación de prototipos

Sommerville [6] enuncia a un prototipo como "un modelo ejecutable del sistema que los usuarios finales y el cliente pueden utilizar para constatar que el sistema cubre sus necesidades reales". Por otro lado, al considerar la usabilidad como un atributo de calidad requerido, el prototipado de la interfaz gráfica es un buen complemento a las tácticas arquitectónicas para alcanzar dicho atributo.

En la fase de Elaboración desarrollamos un diseño de interfaz gráfica navegable con datos *mockeados* de forma de permitirle al cliente indagar en cómo sería la interfaz del sistema, las acciones que podría realizar y la dependencia entre las pantallas. Pueden encontrarse capturas de pantalla de este prototipo en el Documento de Especificación de Requerimientos (Anexo E). Como feedback importante del usuario teníamos dos grandes grupos de cuestiones:

- Cuestiones de Interfaz Gráfica de Usuario: cambios referentes al diseño de la interfaz, pero que no cambiaban la funcionalidad del sistema.
- Cuestiones de Requerimientos Funcionales: cambios referentes a cosas que el sistema no permitía hacer, cosas que estaban limitadas con la UI expuesta, cuestiones que estaban de más, etc.

Ambas cuestiones eran registradas para un posterior análisis/rediseño en la siguiente iteración de Elaboración, o al final de esta fase.

3.3.3.3. Generación de casos de prueba

Sommerville [6] sostiene que, como un requerimiento debe ser comprobable, diseñar casos de prueba para un requerimiento indirectamente puede revelarnos cuán difícil será implementarlo. “Si una prueba es difícil o imposible de diseñar, esto generalmente significa que los requerimientos serán difíciles de implementar, por lo que deberían reconsiderarse”. Si bien no estamos del todo de acuerdo con esta afirmación, sí creemos que los casos de prueba permiten revelar problemas o huecos en los requerimientos especificados.

Por este motivo, decidimos elaborar casos de prueba como parte del proceso de pruebas, altamente relacionado con el proceso de análisis. Para detalles de este proceso puede consultar la sección Proceso de Pruebas. Respecto al aspecto del análisis de requerimientos asociados a esta actividad, destacamos que los casos de prueba fueron elaborados a partir de los casos de uso del sistema, por lo que el primer nivel de error se detecta allí. Cuando se detectaba que un caso de uso era incorrecto o faltaba, se lo agregaba como caso de uso y se listaban los requerimientos asociados a este. Al mismo tiempo, ambas cosas se marcaban para revisión con el cliente en la siguiente reunión.

De esta forma, al especificar casos de prueba pudimos detectar algunos requerimientos faltantes o incompletos.

3.3.4. Requerimientos en la fase de Elaboración

En los puntos anteriores vimos los lineamientos de cada actividad comprendida en el proceso de adquisición, especificación y validación de requerimientos. En esta sección hablaremos de cómo están interrelacionadas estas actividades en la fase de Elaboración del sistema.

En la fase de Inicio llevamos a cabo las primeras entrevistas con el cliente, que nos brindaron una base para realizar un análisis inicial del dominio y de potenciales requerimientos, que se plasmaron en un documento con información acerca del dominio. A partir de la información obtenida en esta fase, en la fase de elaboración se creó un cuestionario orientado a obtener más detalles de dominio y de los requerimientos funcionales del sistema, así como también diversos escenarios para que el cliente pudiera explayarse en cuanto a las actividades llevadas a cabo al atender un paciente, revisar movimientos de caja, registrar datos en la historia clínica del paciente, etc. El siguiente paso luego de la elaboración del cuestionario y los escenarios, fue reunirse con el cliente para concretar la entrevista. Si bien tratamos de seguir un flujo de entrevista en base al plan de la misma, también aprovechamos para realizar preguntas adicionales que se nos ocurrían mientras llevábamos a cabo la entrevista. Luego de la entrevista pasamos al análisis de los diferentes escenarios, a partir de los cuales pudimos realizar algunas preguntas adicionales y plantear algunos escenarios adicionales para analizar.

Una vez terminada la entrevista y la revisión de escenarios, luego de tomar nota de todo lo hablado, pactábamos entre nosotros una reunión para analizar toda la información obtenida. Decidimos realizar la reunión dentro de los dos días siguientes a la entrevista, para asegurarnos de tener frescas las respuestas dadas por el cliente. En esta reunión interna del equipo, procedíamos a descomponer las respuestas del cliente en requerimientos del sistema, separándolos en tres categorías principales:

- Requerimientos claramente definidos; cuestiones que el cliente dejó en claro y que, en principio, no prestaban lugar a confusión.
- Requerimientos en oposición entre sí; cuestiones que en algunos momentos de la entrevista (o escenarios) parecían de una manera, pero en otros momentos (u otros escenarios) estaban abordadas de manera opuesta.
- Posibles requerimientos faltantes; requerimientos incompletos que presentíamos requerían mayor información para ser definidos.

Si bien las tres cuestiones serían abordadas en la iteración de Elaboración en que nos encontrábamos o en futuras entrevistas con el cliente, decidimos hacer una separación de las mismas. Los requerimientos aparentemente bien definidos pasaban a la especificación de requerimientos, y luego seguían el camino del análisis al elaborarse los casos de uso y los casos de prueba. Por otro lado, los requerimientos en conflicto y los posibles requerimientos faltantes pasaban a un documento de consultas a hacer al cliente, y solo serían especificados una vez que se hubiesen convertido en requerimientos bien definidos.

Como siguiente paso en el análisis de requerimientos, se pasaba a especificar casos de uso a partir de los requerimientos bien definidos. De la especificación de casos (Anexo E) de uso surgían nuevas consultas o escenarios que se registraban junto a las consultas ya existentes, para tratarlas con el cliente en la siguiente reunión. Una vez terminada la especificación de los casos de uso procedimos a especificar casos de pruebas. En este punto podían surgir algunas consultas adicionales que se registraban junto con el resto.

A partir del listado de requerimientos, especificación de casos de uso y especificación de casos de prueba, se realizaba (y en posteriores iteraciones, se mejoraba) un diseño de interfaz de usuario. En la primera iteración de elaboración el diseño se realizó mediante *mockups* o bocetos, mientras que en la siguiente iteración pasaron a ser prototipos ejecutables de interfaz. Mencionamos esto en el proceso de análisis de requerimientos, ya que la elaboración de *mockups* nos permitía analizar cuán completos eran los requerimientos que teníamos definidos y así decidir si necesitábamos realizar alguna pregunta adicional en la siguiente entrevista con el cliente.

Una vez terminadas las actividades anteriores, revisábamos la lista de consultas y escenarios pendientes para la siguiente reunión con el cliente y le dábamos un formato que nos permitía elaborar el cuestionario para la siguiente entrevista. De esta manera, el proceso se repetía una vez más, permitiendo reducir la lista de consultas y completando la lista de requerimientos bien definidos. La única diferencia entre las siguientes reuniones con el cliente y la primera reunión formal descrita en esta sección, es que antes de llevar a cabo la entrevista propiamente dicha, realizamos una revisión de los requerimientos (aparentemente) bien definidos y le mostrábamos al cliente los spikes (o prototipos ejecutables) de interfaz. Luego de estas dos actividades, procedíamos a realizar la entrevista y el análisis posterior de requerimientos de la manera ya descrita.

3.3.5. Requerimientos en la fase de Construcción

Luego de la fase de Elaboración la necesidad de reuniones con el cliente para realizar análisis se redujo notablemente. El refinamiento de la lista de requerimientos se realizaba mediante consultas por email en manera general; sólo recurríamos a reuniones si se trataban de asuntos críticos o cuestiones de dominio que podían afectar de manera significativa al sistema en construcción.

En la etapa de construcción cada iteración seguía una estructura genérica como se describe a continuación:

- Refinamiento de casos de prueba antes de la codificación.
- Codificación.
- Ejecución de las pruebas.
- Revisión de la iteración, análisis y rediseño.

El primero de los puntos mencionados que afectaba los requerimientos del sistema era el refinamiento de casos de prueba. Dado que luego del análisis y diseño de la fase de Elaboración el modelo de sistema había evolucionado y se había perfeccionado, era necesario revisar los casos de prueba para ver si era necesario plantear nuevos casos o casos con mayor detalles que se adaptasen al modelo más completo del sistema. En este punto, podía que surgieran nuevas preguntas para el cliente, aunque en general eran dudas menores que registrábamos en un documento y consultábamos por email al cliente.

3.3.6. Artefactos de Análisis

Durante el análisis y la captura de requisitos se generaron distintos artefactos destinados a documentar, comunicar y ayudar a la obtención de los requerimientos del sistema. El Documento de Especificación de Requerimientos (Anexo E) contiene la descripción del comportamiento esperado del sistema y artefactos tales como el diagrama de casos de uso, la especificación de los mismos y capturas de pantalla del prototipo de interfaces de usuario. Sobre este documento se asentaron y organizaron los requerimientos, para luego someterlo a revisiones y actualizaciones.
A continuación podemos ver el diagrama de casos de uso:

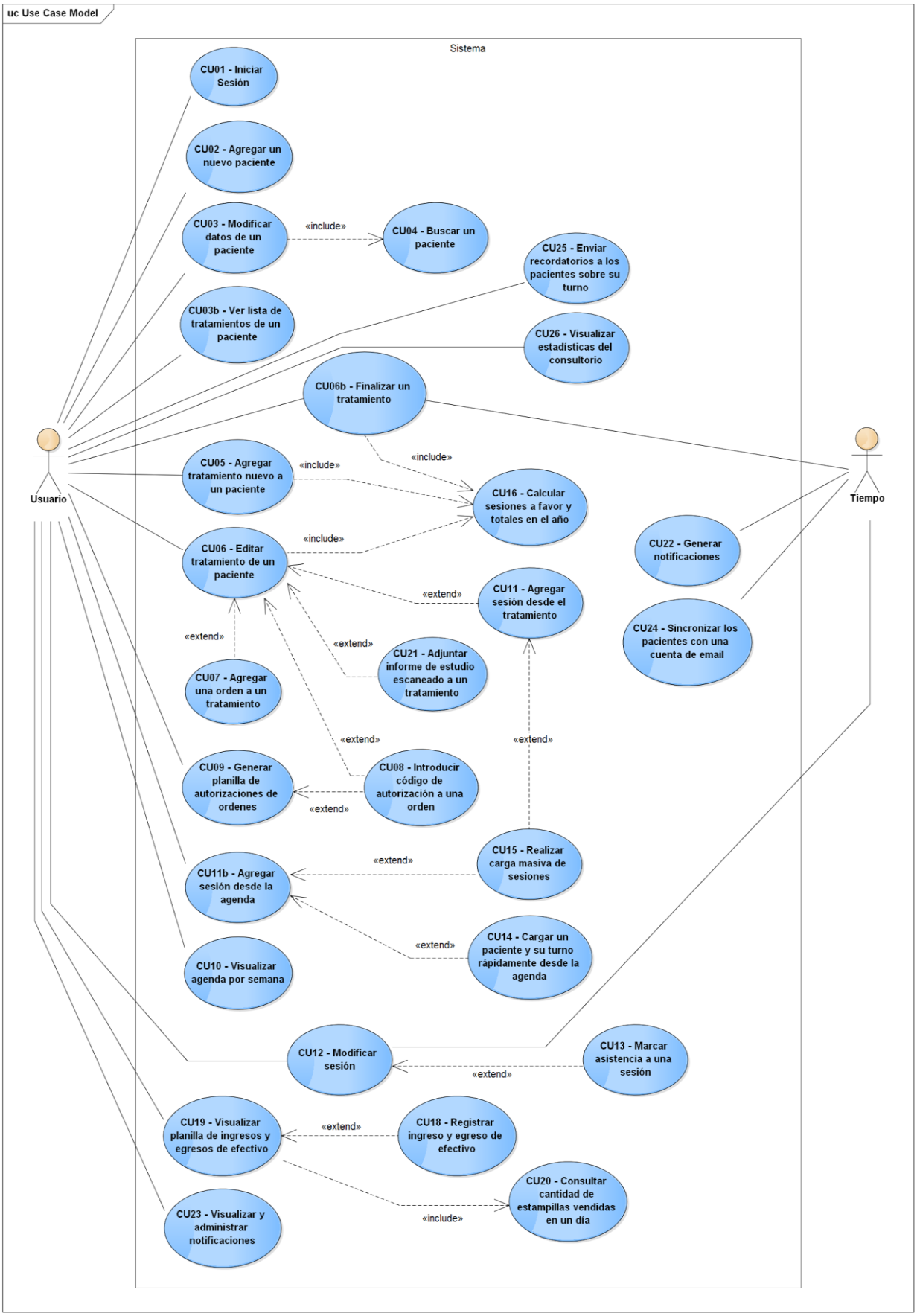

Ilustración 1.13: Diagrama de casos de uso del sistema

Como parte del análisis generamos el modelo de clases de análisis:

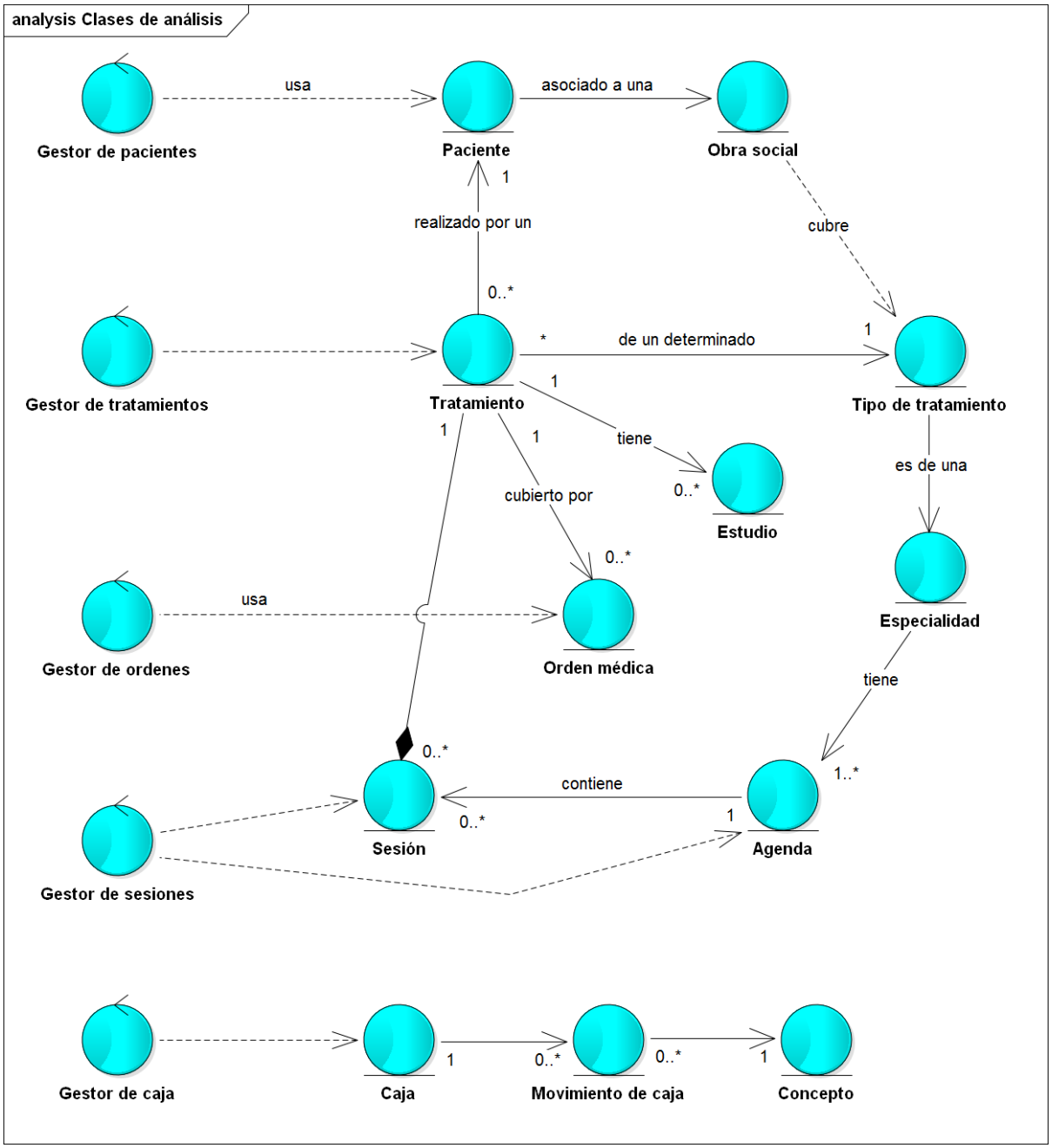

Ilustración 1.14: Diagrama de clases de análisis del sistema.

3.4. Diseño

Las actividades de diseño son el centro de atención sobre el final de la fase de Elaboración y el comienzo de las iteraciones de Construcción. Esto contribuye a crear una arquitectura estable y sólida y una base del modelo de implementación. Más tarde, durante la fase de Construcción, cuando la arquitectura es estable y los requisitos están bien entendidos, el centro de atención se desplaza a la implementación [1].

Cada uno de los elementos del análisis proporciona información necesaria para crear los modelos de diseño necesarios para la especificación completa del diseño. En el proyecto especificamos el diseño en sus distintos niveles mediante modelos que fueron agrupados en: diseño de datos y componentes, diseño de arquitectura, y diseño de interfaces de usuario.

3.4.1. Diseño de datos y componentes

Los modelos de diseño de datos y componentes describen cuáles serán y de qué forma van a interactuar las entidades y componentes para satisfacer los requerimientos funcionales. En otras palabras, se modelan las clases de diseño que van a permitir implementar (en código) las especificaciones de los casos de uso. Cualquier instancia de diseño es el paso previo a la implementación. Por eso se busca que las clases de diseño tengan una correspondencia directa con los componentes de la implementación, buscando expresar los modelos en el lenguaje de programación que se usará (tipos de datos, nomenclatura, modificadores de acceso, estereotipos).

Las entradas para este proceso consisten fundamentalmente en el diseño de clases de análisis y la especificación de los casos de uso. Modelar las clases de diseño implica crear clases que soporten los requerimientos funcionales; para esto hay que definir sus: operaciones, atributos, relaciones (agregaciones, generalizaciones), estados (si corresponde) y dependencia respecto al diseño de arquitectura (relación con los patrones de diseño). La definición de estas características dependerá del tipo de clase de diseño con el que estemos tratando. En general se distinguen tres tipos de clases de diseño:

- Entidades: Representan la información persistente, mayormente asociada al dominio. No poseen comportamiento. La relación entre las entidades en parte ya está plasmada en el modelo de clases de análisis.
- Clases de control: En este tipo de clases incluimos los componentes controladores y gestores. Estos últimos contienen la lógica de negocio. En la tarea de diseñar estos componentes nos encontramos con dependencias arquitectónicas.
- Interfaces: Se incluyen en el modelo las interfaces propias y las que establecen dependencias con el framework u otras librerías.

Diseñamos con la idea de que el modelo del dominio sea extensible en un futuro hacia otras especialidades médicas. Se planteó el hecho de que pueda haber más de una especialidad, y que cada especialidad soporte la gestión de más de un profesional con su respectiva agenda. Por otro lado, se separó la gestión de los pacientes de la gestión de los

tratamientos y sesiones, haciendo que la gestión de pacientes no dependa de los tratamientos, sino que la visibilidad sea inversa. Con esto se logra mantener la misma base de pacientes, que pueden estar relacionados con la gestión de distintas especialidades.

Durante el modelado, al momento de identificar las entidades del dominio, consideramos importante introducir el concepto de *Tratamientos*. Si bien éste es un concepto que prácticamente no se utiliza en la gestión diaria del consultorio, el mismo es clave en el modelado, ya que nos permite estructurar el modelo conectando el paciente con las sesiones y órdenes.

La siguiente imagen contiene el diagrama de clases detallado. En él pueden encontrarse, de izquierda a derecha, los controladores que interactúan con la vista y el usuario, las clases que contienen la lógica del negocio y las entidades que representan o abstraen los componentes del dominio. Cada clase tiene asociado un estereotipo que indica las características generales que tendrá esa clase al momento de la implementación. Por ejemplo, una Entity JPA será una clase sin comportamiento, o los gestores EJB no tendrán estado.

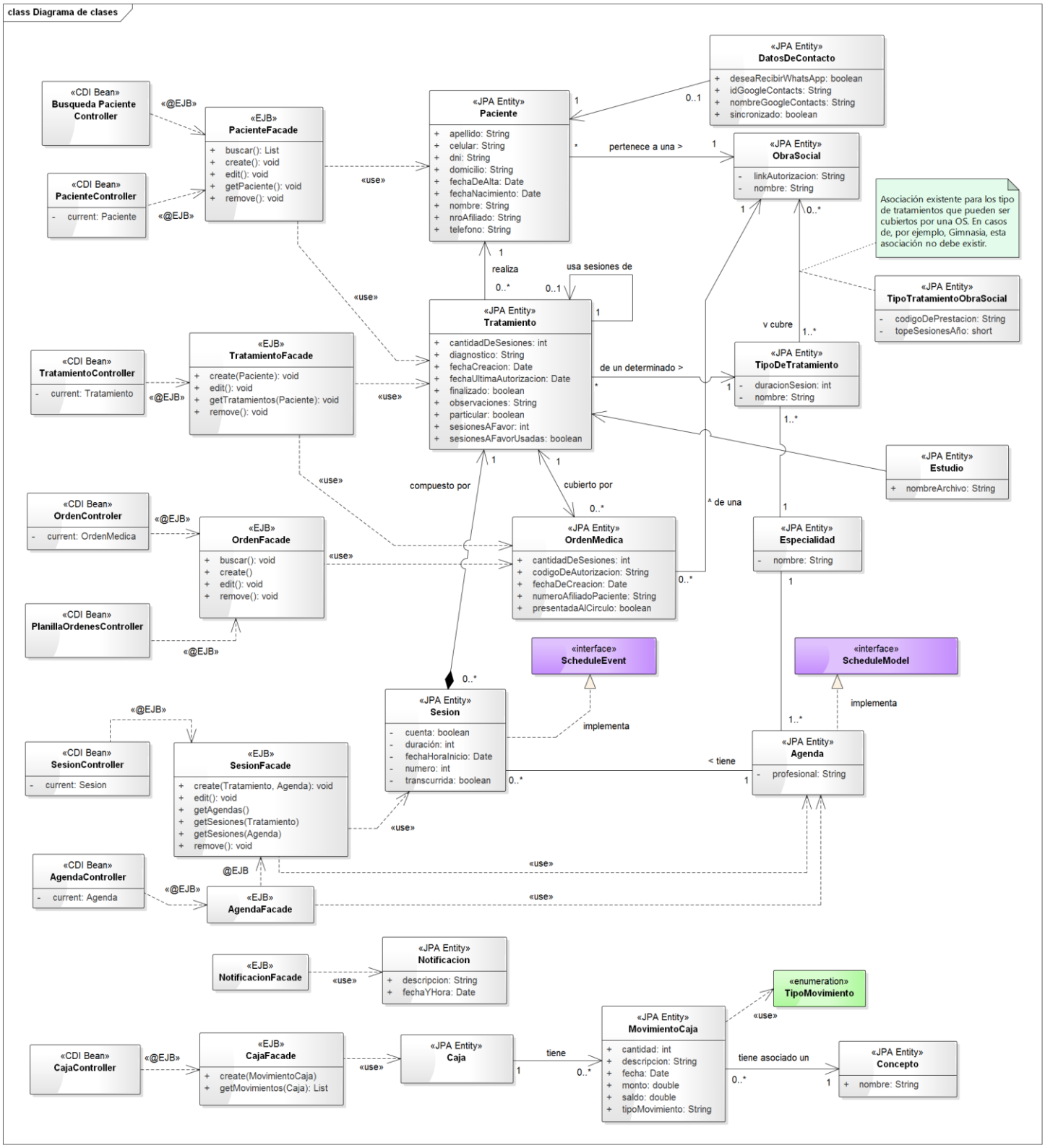

Ilustración 1.15: Diagrama de clases detallado del sistema.

3.4.2. Diseño de Arquitectura

Los requerimientos funcionales de un sistema pueden lograrse mediante el uso de cualquier estructura. De hecho, si la funcionalidad fuera el único requisito, el sistema podría existir como un único módulo monolítico sin estructura interna. En lugar de ello, se descompone en módulos para que sea comprensible y para contribuir a una variedad de otros fines. De esta manera, la funcionalidad es en gran medida independiente de la estructura. El interés en la funcionalidad está en cómo ésta interactúa y limita otras cualidades, como los atributos de calidad o requerimientos no funcionales [2].

3.4.2.1. Arquitectura y calidad

La arquitectura es crítica para alcanzar muchas cualidades de interés en un sistema, y esas cualidades deben ser diseñadas y evaluadas a nivel arquitectónico [2].

El diseño es el punto en el que se introduce calidad en la ingeniería de software. Sin diseño se corre el riesgo de obtener un sistema inestable, que falle cuando se hagan cambios pequeños, o uno que sea difícil de someter a prueba, o en el que no sea posible evaluar la calidad hasta que sea demasiado tarde en el proceso de software, cuando no queda mucho tiempo y ya se ha gastado mucho dinero [9].

Si es que un sistema será capaz o no de exhibir los atributos de calidad deseados (o requeridos) está sustancialmente determinado por su arquitectura. Performance, modificabilidad, seguridad, escalabilidad, son algunos ejemplos, y las estrategias para estos y otros atributos de calidad son totalmente arquitectónicas. Sin embargo, es importante entender que la arquitectura por sí sola no puede garantizar la funcionalidad o la calidad. Diseños de bajo nivel malos, o decisiones de implementación siempre pueden degradar un diseño arquitectónico adecuado. Las decisiones que se toman durante todo el ciclo de vida, desde el diseño de alto nivel hasta la codificación e implementación, afectan a la calidad del sistema. Para asegurar calidad, una buena arquitectura es necesaria, pero no suficiente [2].

El diseño de la arquitectura lo planteamos desde esta perspectiva, como una actividad necesaria para alcanzar muchos de los atributos de calidad requeridos. Los mismos autores de este enfoque nos ayudan a encontrar la forma de expresar las cualidades que queremos que nuestra arquitectura proporcione al sistema. Para esto se define una serie de escenarios de atributos de calidad, que son incluidos dentro del Documento de Arquitectura del Sistema (Anexo C).

Los escenarios de atributos de calidad son la forma en que se describen los atributos de calidad. Cada escenario es un requerimiento para un atributo de calidad específico. En breve, un escenario de calidad consiste en seis partes:

- Fuente del estímulo: Alguna entidad (humano, sistema o cualquier otro actor)
- Estímulo: Es una situación que debe ser considerada y atendida cuando ésta se produce en el sistema. Por ejemplo: *Un usuario intenta acceder a una funcionalidad del sistema a la que no está autorizado*.
- Entorno: Al momento del estímulo el sistema se encuentra en un determinado estado. El sistema puede estar sobrecargado, o trabajando normalmente, entre otros estados.
- Artefacto: Algún artefacto es estimulado. Puede ser el sistema entero o alguna pieza.
- Respuesta: La actividad que ocurre luego del arribo del estímulo.
- Medida de la respuesta: La respuesta debe ser medida de alguna forma para que el requerimiento pueda ser testeado

Los escenarios especifican los requerimientos no funcionales, y definen una forma de medirlos. Pero, ¿qué es lo que le da a un diseño el atributo de portabilidad, de performance o integrabilidad? El hecho de alcanzar esas cualidades recae en decisiones de diseño fundamentales. Las decisiones de diseño las llamamos *tácticas*. Una táctica es una decisión de diseño que influye en el control de la respuesta de un atributo de calidad. Dado un atributo de calidad, las tácticas intentan controlar la respuesta del sistema ante los estímulos [2].

A modo de ejemplo presentamos el escenario 1 de seguridad. El total de los escenarios se encuentran en el Documento de Arquitectura (Anexo C).

Atributo de calidad: SEGURIDAD	
Parte del Escenario	**Valor Seleccionado**
Fuente	Individuo o dispositivo externo al sistema
Estímulos	Intento de analizar el tráfico de red generado por el sistema
Artefacto	Mecanismo de comunicación entre el navegador y el servidor.
Entorno	Funcionamiento normal. El sistema opera en una red LAN que no es de confianza.
Respuesta	La comunicación es ilegible para el individuo/dispositivo. No puede llevarse a cabo el análisis de tráfico.

Medida de la respuesta	La respuesta mencionada se da en el 99,99% del total de ocurrencias del escenario.
Tácticas propuestas	Encriptar la comunicación cliente-servidor.

Tabla 1.1: Ejemplo de un escenario de calidad. Se trata de uno de los escenarios asociados al atributo Seguridad.

En las secciones siguientes nos limitamos a explicar los atributos de calidad que forman parte de los requerimientos de nuestro proyecto, junto con las tácticas asociadas a cada uno, dejando para el documento de arquitectura el detalle de cada escenario.

3.4.2.2. Seguridad

Uno de los aspectos de seguridad dentro de los requerimientos no funcionales es la autenticación y la autorización. En el diseño se definieron los roles que existirán y a qué funcionalidades podrá acceder cada uno.

Nuestro sistema prevé la existencia de 2 roles. Uno para los usuarios con acceso total a las funcionalidades del sistema, y otro para usuarios con un perfil de secretaria, con acceso a una menor cantidad de funcionalidades. Esto da lugar a un rol *administrador* y un rol *secretaria*.

Como mecanismo de autenticación se decidió utilizar la autenticación por formulario, donde se solicita al usuario un nombre de usuario y una contraseña. Ante el intento de acceder a una página protegida, el sistema redirige al usuario al formulario de login. En caso de que las credenciales sean las correctas, el sistema muestra finalmente la página solicitada.

Para acceder a cualquier apartado del sistema es necesario estar autenticado en el sistema. Es decir, que no hay páginas de acceso público. El rol de administrador, como ya se mencionó, tiene acceso a todas las funcionalidades, mientras que el rol de secretaria tiene acceso a todo menos las funcionalidades de caja e impresión de planilla de órdenes.

Otro escenario relacionado a la seguridad tiene que ver con dos requerimientos:

- Proteger la privacidad o confidencialidad de los datos intercambiados entre el servidor y los clientes.
- Garantizar la integridad de dichos datos; esto hace referencia a asegurar que los datos no fueran modificados en el camino.

La táctica seleccionada tiene que ver con utilizar protocolos de comunicación que cifren el contenido para hacerlo ilegible a terceros y que hagan chequeos de integridad.

Los mecanismos que cifran la información están basados en los estudios de la criptografía. La criptografía es la ciencia que se encarga de estudiar las distintas técnicas empleadas para transformar (encriptar o cifrar) la información y hacerla irreconocible a todos aquellos usuarios no autorizados de un sistema de información basado en TICs, de modo que sólo los legítimos *propietarios* puedan recuperar (desencriptar o descifrar) la información original. Las técnicas de criptografía se basan en algoritmos de encriptación que realizan determinadas transformaciones sobre el *texto* original (reconocido como texto claro) para obtener un *texto* modificado (conocido como texto cifrado o criptograma). Mediante el procedimiento inverso, utilizando algoritmos de desencriptación, puede recuperarse el *texto* original.

El funcionamiento de los algoritmos de encriptación y desencriptación dependen de *claves*. Aunque los algoritmos sean públicos y conocidos por todos, si no se dispone de la clave correcta, resulta imposible realizar el proceso de desencriptación. Por ello, la robustez del sistema criptográfico se basa en la clave utilizada.
Pueden clasificarse los sistemas criptográficos en simétricos, asimétricos e híbridos, dependiendo de la naturaleza de la clave utilizada.

Con respecto al control de integridad de los datos transmitidos se utilizan algoritmos de compendio o *hashing*. Estos algoritmos se caracterizan por reducir el mensaje original a una secuencia de bits que identifica dicho mensaje, y que se denomina *huella digital*. La generación de la huella digital se basa en la realización de una serie de operaciones matemáticas sobre el mensaje original, utilizando una función de dispersión unidireccional (es decir, no puede reconstruirse el mensaje a partir de su compendio). Para considerarse un mecanismo que asegure integridad deben cumplirse un conjunto de propiedades:

- Conociendo la huella digital, no se obtiene ninguna información sobre el mensaje original.
- No es factible encontrar dos mensajes originales que generen la misma huella digital. La probabilidad de colisión (es decir, la obtención de una misma secuencia de bits a partir de dos mensajes distintos) es muy remota, prácticamente nula.
- Cualquier cambio en el mensaje de entrada debe modificar, en promedio, la mitad de los bits que se generan a la salida del algoritmo, es decir, un pequeño cambio en el mensaje cambia totalmente su huella digital.

Dentro del apartado de desarrollo (codificación), en la sección Seguridad se detalla la implementación de estas tácticas.

3.4.2.3. Modificabilidad

El ***escenario 1*** de modificabilidad plantea la capacidad de modificar los textos contenidos en las interfaces de usuarios o aquellos textos o datos contenidos, por ejemplo, en reportes. Una buena táctica para favorecer este atributo de calidad es obtener los textos desde un lugar centralizado en vez de incrustarlos (hard code) directamente en el código fuente.
La fuente de estos textos debe servir tanto para los textos de las interfaces de usuario como para los textos de los componentes del modelo de negocio que emitan mensajes dirigidos al usuario o a los desarrolladores. Es decir, debe poder accederse desde distintas capas.

El ***escenario 2*** está relacionado con el grado de facilidad de implementar un cambio en el sistema, tanto en la lógica de negocio, como en las interfaces de usuario. Para favorecer este atributo se propone separar las responsabilidades del sistema en capas o niveles. Se hizo un diseño de capas siguiendo las convenciones sobre separación más conocidas, ya que son lineamientos que cumplen con su objetivo para la gran mayoría de los proyectos de software web. Las capas son las siguientes:

- Capa de Presentación: Se encarga de presentar la información al usuario y de capturar la interacción de este con el sistema. Se comunica con la capa de negocio abstrayéndose de la lógica implicada en el mismo.
- Capa de Negocio: Contiene los modelos y la lógica del negocio, dando servicio a la capa de Presentación.
- Capa de Datos: Interactúa directamente con el mecanismo de persistencia para manipular los datos implicados en la lógica de negocio.

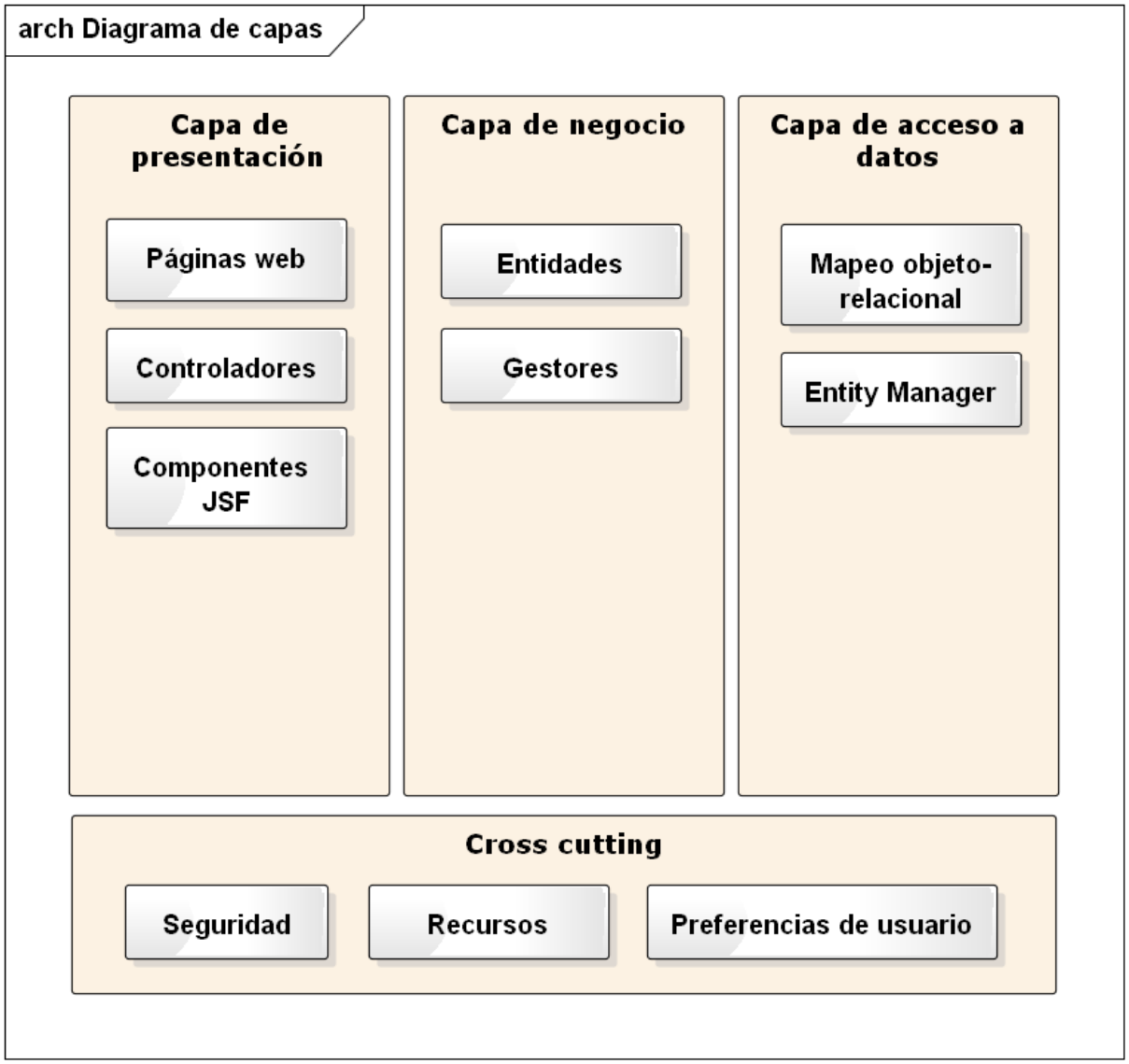

Ilustración 1.16: Diagrama de capas del sistema. La separación en capas de los componentes del sistema contribuye a la facilidad de realizar modificaciones y correcciones sobre el mismo.

Por otro lado, dentro de la capa de negocio, ayuda a la modificabilidad el hecho de modularizar su estructura.

La modularización se hizo buscando favorecer la modificabilidad, según las tácticas propuestas por Kruchten [3].

- Mantener la coherencia semántica: La coherencia semántica refiere a las relaciones entre las responsabilidades que existen dentro de un módulo. El objetivo es asegurar que todas esas responsabilidades trabajen juntas sin depender excesivamente de otros módulos. El logro de este objetivo proviene de elegir responsabilidades que tengan coherencia semántica. (...) Generar servicios comunes que sirvan a módulos especializados se suele considerar como una práctica de reutilización. Esto es

correcto, pero abstraer servicios comunes también soporta modificabilidad. Las modificaciones deberán hacerse sólo una vez en dichos servicios comunes en lugar tener que hacerlas en cada módulo donde se utilizan.

- Anticipar los cambios esperados: (...) La pregunta básica es "¿Para los cambios esperados, la descomposición actualmente propuesta reduce la cantidad de módulos afectados para lograr dicho cambio?" Otra pregunta es "¿Aquellos cambios sustancialmente diferentes afectan a los mismos módulos?" (...) La táctica de anticipar los cambios esperados no se refiere a la coherencia de las responsabilidades de un módulo, sino más bien a minimizar los efectos de los cambios.
- Generalizar el módulo: Hacer un módulo más general permite resolver un rango más amplio de funciones basados en su interfaz. (...) Cuanto más general es un módulo, más probable es que los cambios solicitados se pueden hacer ajustando la interfaz o inputs en lugar de modificar el módulo por dentro.

Teniendo en cuenta lo mencionado anteriormente, se diseñó la capa de negocio identificando los siguientes módulos:

- Módulo de pacientes: Se encarga de la gestión de pacientes, su alta, baja, modificación y mantenimiento de datos relacionados. Incluye la gestión de las obras sociales. Es un módulo desde donde básicamente son consumidos datos, no dependiendo directamente de otro módulo.
- Módulo de tratamientos y sesiones: Gestiona los tratamientos, sesiones y agendas en conjunto. Encapsula las funcionalidades del dominio relacionadas con el seguimiento de los tratamientos que realizan los pacientes, incluyendo su historia clínica o kinésica, el profesional que lo atiende y la gestión de turnos. Depende en parte del módulo de pacientes.
- Módulo de órdenes médicas: Por un lado se encarga del alta, baja y modificación de órdenes médicas, y por otro, abarca las funcionalidades la generación de reportes.
- Módulo de caja: Provee las funcionalidades relacionadas con los ingresos y egresos de efectivo.
- Módulo notificaciones

El ***escenario 3*** se enfoca en la modificabilidad dentro de la capa de presentación, tanto del aspecto visual como del comportamiento de los componentes gráficos (visualización de errores de validación, paginación de las tablas, visualización de la información, etcétera).

El problema en concreto sería: ¿Cómo modularizar la funcionalidad de la interfaz de usuario de una aplicación web para que se puedan modificar fácilmente las partes individuales?

Hay diversos aspectos que dan lugar a pensar en un patrón como MVC:

- La lógica de la interfaz usuario tiende a cambiar con mayor frecuencia que la lógica de negocio, especialmente en aplicaciones web. Por ejemplo, se pueden agregar nuevas páginas o se pueden mezclar los diseños de páginas existentes. Si el código de la presentación y la lógica de negocio se combinan en un único objeto, debe modificarse un objeto de la lógica de negocio cada vez que cambie la interfaz de usuario. Esto es probable que introduzca errores y requiera la ejecución de todos los test de la lógica de negocio ante cualquier cambio mínimo en interfaz de usuario.

- En algunos casos, la aplicación muestra los mismos datos de diferentes maneras. Por ejemplo, puede darse el caso de que para un determinado tipo de usuario se le ofrezca una vista de hoja de cálculo de los datos, mientras que para otro se prefiera un gráfico circular. Es necesario que si un usuario cambia los datos en una vista, el sistema actualice automáticamente todas las demás vistas de los datos.

- Crear casos de prueba automatizados para las interfaces de usuario suele ser más difícil y demanda más tiempo que hacerlo para la lógica de negocio. Por lo tanto, reducir la cantidad de código que está directamente relacionado a la interfaz de usuario mejora la facilidad de prueba de la aplicación.

El patrón Model-View-Controller (MVC) separa el modelado del dominio, la presentación y las acciones originadas por el usuario en tres clases:

- Modelo. El modelo gestiona el comportamiento y los datos del dominio de la aplicación, responde a las solicitudes de información (normalmente desde la vista) y responde a las instrucciones para cambiar de estado (normalmente desde el controlador) [4].
- Vista. La vista gestiona la visualización de la información.
- Controlador. La definición de controlador depende del tipo de sistemas del que estemos hablando y del nivel de abstracción desde donde lo miremos. En nuestro caso es un sistema web y podemos mirarlo desde el punto de vista de las solicitudes web. Así el controlador puede definirse como aquel componente MVC que recibe y

gestiona las solicitudes http, que incluye los comandos y datos ingresados por el usuario, para luego dirigir esas solicitudes al lugar correcto y seleccionar la vista a mostrar.

La siguiente figura representa la relación estructural entre los tres componentes.

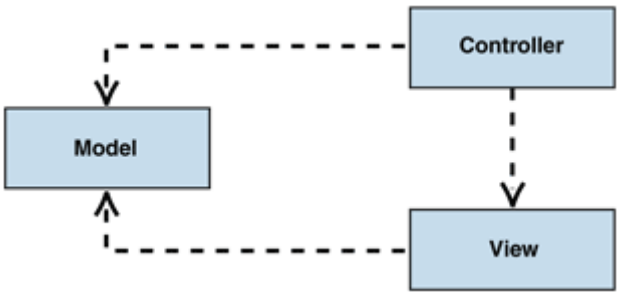

Ilustración 1.17: Componentes involucrados en el patrón MVC. Notar que la vista y el controlador dependen del modelo, mientras que el modelo no depende de ninguno de ellos.

3.4.2.4. Usabilidad

Al referirnos a este atributo, lo que se busca es que el sistema mantenga sus características de facilidad de uso independientemente del dispositivo desde donde se lo acceda. Para lograr este objetivo, se emplean tácticas transversales al diseño de las interfaces de usuario, como ser grillas *responsive* y media *queries*. Estas características se explican en detalle en la Implementación del diseño responsive.

En necesario también que sea posible manipular la presentación (interfaces de usuario) independientemente de los demás componentes del sistema. El patrón Modelo-Vista-Controlador (MVC) explicado en la sección Patrón MVC cubre esta necesidad.

3.4.3. Documentación de la arquitectura

Hemos visto muchos libros y artículos donde un diagrama intenta capturar la esencia de la arquitectura de un sistema. Pero mirando cuidadosamente el conjunto de cajas y flechas que se muestran en estos diagramas, queda claro que sus autores han lidiado mucho para representar en un diagrama más de lo que realmente puede expresar. ¿Las cajas representan programas en ejecución? ¿O trozos de código fuente? ¿O computadoras físicas? ¿O simplemente agrupaciones lógicas de funcionalidad? ¿Las flechas representan dependencias de compilación? ¿O flujos de control? ¿O flujos de datos? Normalmente es un poco de todo. ¿La arquitectura necesita un único estilo arquitectónico? A veces la arquitectura sufre un exceso de énfasis en un aspecto determinado del desarrollo de software: la ingeniería de datos, o la eficiencia en tiempo de ejecución o la estrategia de desarrollo y organización del

equipo (...). Como remedio, proponemos organizar la descripción de una arquitectura de software utilizando varias *vistas* concurrentes, cada una de las cuales aborda un conjunto específico de asuntos o cuestiones [3].

La arquitectura del software incluye abstracciones, descomposiciones, relaciones de composición y estilos arquitectónicos. Para describir la arquitectura del software usamos un modelo compuesto por múltiples vistas o perspectivas. Para manejar grandes y complejas arquitecturas, el modelo que proponemos se compone de 5 vistas:

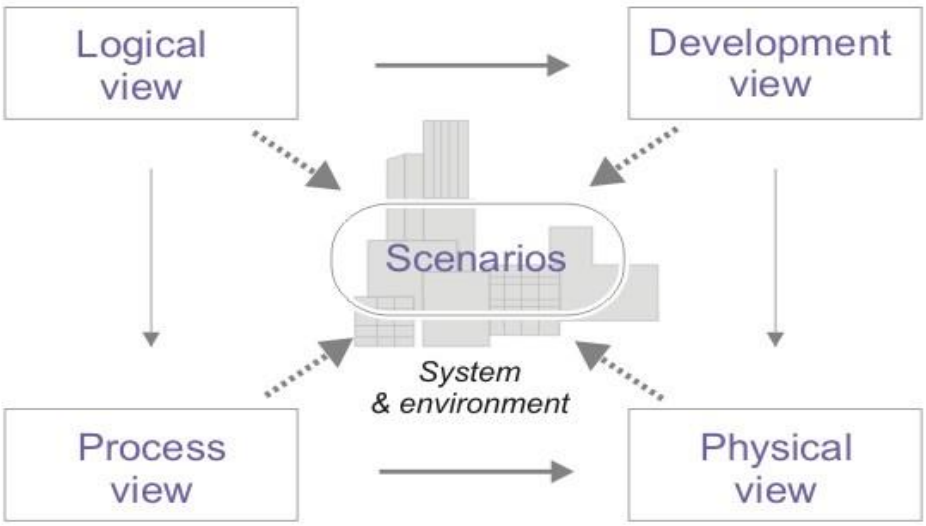

Ilustración 1.18: Modelo de vistas 4+1 propuesto por Philippe Kruchten [3]. Su objetivo es documentar la arquitectura del software de una forma completa y ordenada.

- Vista lógica: Está enfocada en describir la estructura y funcionalidad del sistema. Los diagramas UML que se utilizan para representar esta vista son los Diagrama de Clase, Diagrama de Comunicación, Diagrama de Secuencia
- Vista de desarrollo (o de implementación): Ilustra el sistema desde la perspectiva del programador y está enfocado en la administración de los artefactos de software. Utiliza el Diagrama de Componentes UML, o el de paquetes, para describir los componentes de sistema.
- Vista de proceso: Trata los aspectos dinámicos del sistema, explica los procesos de sistema y cómo se comunican. Se enfoca en el comportamiento del sistema en tiempo de ejecución. La vista considera aspectos de concurrencia, distribución, rendimiento, escalabilidad, etc. En UML se utiliza el Diagrama de Actividad para representar esta vista

- Vista física (o de despliegue): Describe el mapeo del software sobre los componentes de hardware y refleja el aspecto de distribución. Está relacionada con la topología de componentes de software en la capa física, así como las conexiones físicas entre estos componentes.
- Escenarios: Son en sí mismos una abstracción de los requerimientos más importantes. Sirve para descubrir los elementos arquitectónicos durante el diseño de la arquitectura y como un elemento de validación después que el diseño está completo. Su diseño se expresa generalmente mediante Diagramas de Casos de Uso [5].

3.4.4. Diseño de interfaces de usuario

Durante el proceso de análisis se atendió el diseño preliminar de las distintas pantallas del sistema, dando como resultado un prototipo básico navegable de las interfaces de usuario (ver 3.3.3.2. Creación de prototipos). Partiendo de esta base, sabiendo los componentes que necesitábamos en la interfaz de usuario, se comenzó a pensar en un diseño que sea consistente entre todas las pantallas, pero que además, resulte familiar y amigable de usar. Para esto, se investigó y se llegó a la conclusión de que los usuarios pasan la mayoría de su tiempo interactuando con aplicaciones que siguen una misma línea de diseño, por lo tanto ya están familiarizados y decidimos sacar provecho de esta situación. Material Design es una de estas normativas de diseño creada y diseñada por Google y lo que busca es una experiencia de usuario unificada a través de todos sus productos en cualquier plataforma[1].

Como se mencionó anteriormente, basamos nuestras interfaces de usuario en Material Design, lo que nos dió como resultado pantallas más limpias y minimalistas, ayudando al usuario a interactuar de manera más directa con el sistema.

Un aspecto clave que tuvimos que considerar en el diseño fue un pedido que nos hizo el cliente desde un principio, poder visualizar el contenido desde un dispositivo móvil o tablet. Si bien, al tratarse de un sistema web, el contenido se puede visualizar desde cualquier dispositivo, hay que adaptarlo para que la información sea presentada de manera correcta. El ***Diseño Web Responsive*** es considerado una de las mejores prácticas hoy en día para la correcta visualización de una misma página en distintos dispositivos.

[1] En el sitio material.io puede encontrarse más información acerca de Material Design, así como también los lineamientos o guías de dicho sistema de diseño.

Ilustración 1.19: En el mercado existen diversos dispositivos capaces de acceder a contenido web, con diversos tamaños de pantalla y resoluciones. Con un diseño responsive logramos adaptar los elementos de la vista a este escenario.

Realizar un diseño de interfaces de usuario responsive consiste en redimensionar y colocar los elementos de la web de forma que se adapten al ancho de cada dispositivo permitiendo una correcta visualización y una mejor experiencia de usuario.

3.5. Desarrollo del sistema

3.5.1. Desarrollo de aspectos transversales

3.5.1.1. Seguridad

Autenticación y autorización

La tecnología empleada nos permite abstraer el desarrollo de la aplicación de la gestión de los usuarios; sólo hay que tener en cuenta los roles que puedan existir. Es decir, Java EE nos permite concentrarnos durante el desarrollo en los roles que habrá, configurar los recursos (generalmente páginas web) a los que puede acceder cada rol, y luego, ya en una fase de instalación, cargar los usuarios mediante el servidor de aplicaciones Glassfish.

Los roles definidos en la etapa de diseño son configurados en la aplicación mediante el Descriptor de Despliegue *web.xml*, uno de los archivos de configuración de la aplicación.

```
… <security-role>
      <description>Usuario con todos los permisos</description>
      <role-name>admin</role-name>
</security-role>
<security-role>
      <description>Usuarios con permisos restringidos</description>
      <role-name>secretaria</role-name>
</security-role> …
```

Ilustración 1.20: Un rol se encuentra definido por su nombre; la descripción es un dato adicional. De esta forma establecemos qué roles existirán en el sistema, para luego asociarles permisos o restricciones.

En el mismo archivo, se configuran los permisos, definiendo qué URL puede acceder cada rol.

Para conectar la gestión de usuarios (llevada a cabo por el servidor de aplicaciones) con los roles de la aplicación, sólo queda mapear los usuarios o grupos de usuarios con los roles de la aplicación. Es decir, queda determinar qué rol tendrá cada usuario. Si el nombre del grupo de usuarios es el mismo que el rol, el mapeo es realizado automáticamente por el servidor de aplicaciones.

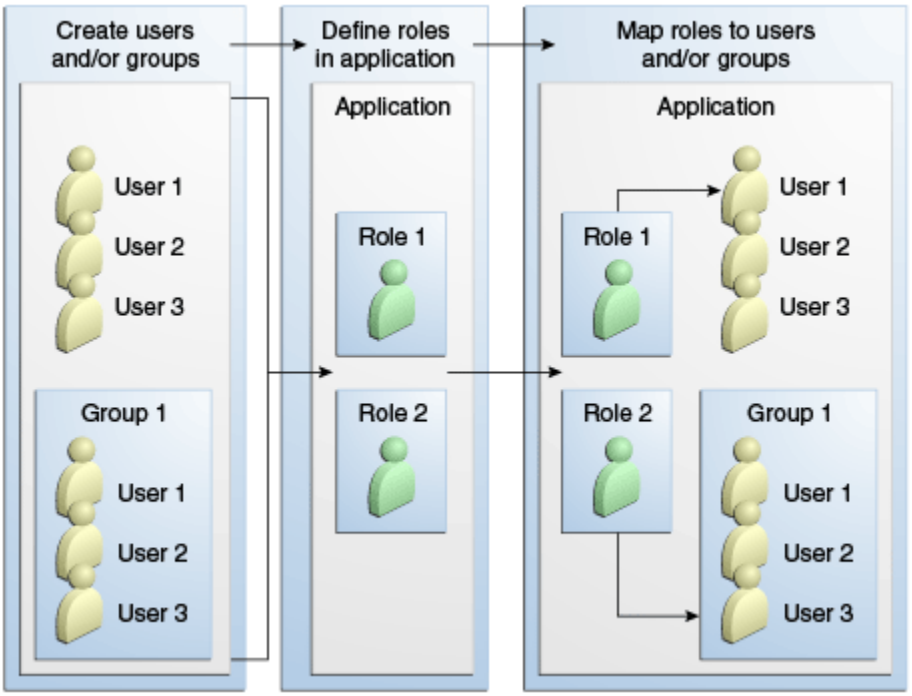

Ilustración 1.21: Relación entre los usuarios y los roles. La imagen muestra de izquierda a derecha: los usuarios del servidor, los roles de la aplicación y el posterior mapeo. Si se cambiase el nombre del Group 1 *de modo que coincida con el de algún rol, por ejemplo* Role 2*, el mapeo sería automático.*

Cifrado de la información transmitida

En la sección de diseño de arquitectura, donde se abordan los atributos de calidad, se describen los requerimientos existentes de confidencialidad e integridad de los datos transmitidos por el sistema. Además se explica la táctica empleada para resolver dichos requerimientos y se introducen los conceptos teóricos relacionados. En esta sección se tratan los aspectos prácticos de la implementación de la táctica de cifrado y chequeos de integridad.

Los sistemas web intercambian contenido con sus clientes, los navegadores, haciendo uso del protocolo HTTP (Hypertext Transfer Protocol). HTTP por sí sólo no aporta las características de seguridad requeridas, pero existe otro protocolo que le adiciona dichas cualidades. Se trata de HTTPS (HTTP Secure). HTTPS es ampliamente usado en el mercado y la tecnología con la que trabajamos lo soporta de forma nativa; es decir, sólo hay que configurar apropiadamente el servidor de aplicaciones. Antes de detallar los elementos necesarios para configurar HTTPS en nuestro sistema, explicamos brevemente en qué consiste el protocolo.

HTTPS es también llamado HTTP over TLS o HTTP over SSL, y se define como una comunicación HTTP dentro de una conexión cifrada por el protocolo TLS (o su antecesor SSL). La siguiente imagen muestra la capa de seguridad que se añade al modelo de capas que da lugar a HTTPS.

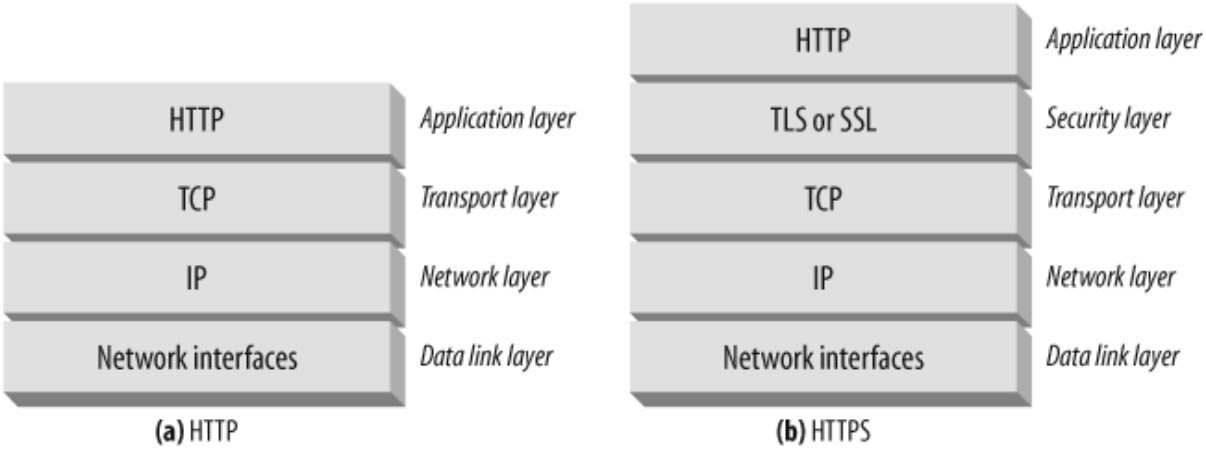

Ilustración 1.22: Gráfico de las capas del modelo TCP/IP, con el protocolo HTTP ocupando la capa de aplicación. En un esquema que emplee HTTPS puede decirse que se adiciona una capa de seguridad, donde comúnmente se emplea el protocolo TLS.

TLS (Transport Layer Security) es un protocolo criptográfico que otorga privacidad a la conexión cifrando los datos transmitidos mediante criptografía simétrica, posibilita autenticar la identidad de las partes (cliente y servidor) empleando criptografía asimétrica y asegura la integridad de los mensajes. Para funcionar, el cliente debe indicar al servidor que desea establecer la conexión usando este protocolo. En HTTPS esto se hace indicando un puerto específico, comúnmente el 443. En el inicio de la comunicación se emplea la criptografía asimétrica para establecer la configuración del cifrado, y luego el resto de la comunicación es cifrada usando criptografía simétrica. Para la criptografía asimétrica o de clave pública es necesario contar con certificados digitales que permitan autenticar al servidor (deltagestion.com.ar en nuestro caso) y que contenga nuestra clave pública. Los certificados son emitidos por autoridades de certificación (CA).

Sabiendo esto, para poner en funcionamiento HTTPS debíamos adquirir un certificado digital que autentique nuestro dominio y que permita habilitar en el servidor de aplicaciones la comunicación HTTP segura. Una vez obtenido el certificado digital hay que importarlo o almacenarlo en lo que Java EE denomina *keystore*. Un keystore es una base de datos de claves privadas y sus certificados asociados, que a su vez autentican las claves públicas correspondientes. Teniendo los certificados y claves necesarias en el keystore debemos configurar el servidor de aplicaciones para que funcione en el puerto 443 y emplee el protocolo TLS. Esto se hace configurando los http-listeners de Glassfish:

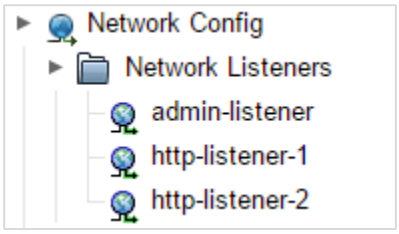

Ilustración 1.23: Listado de network listeners existentes por defecto en Glassfish. Un listener esta definido por una dirección IP y un puerto. El admin-listener es usado por la consola de administración web del servidor, mientras que los demás están a disposición del usuario.

Cada http listener o network listener es un socket que tiene una dirección IP y un puerto. Cada listener debe tener una combinación única de puerto y dirección IP. También es posible indicar un puerto determinado y que escuche en cualquier dirección IP. La imagen siguiente muestra la configuración para un listener que trabaje con HTTPS:

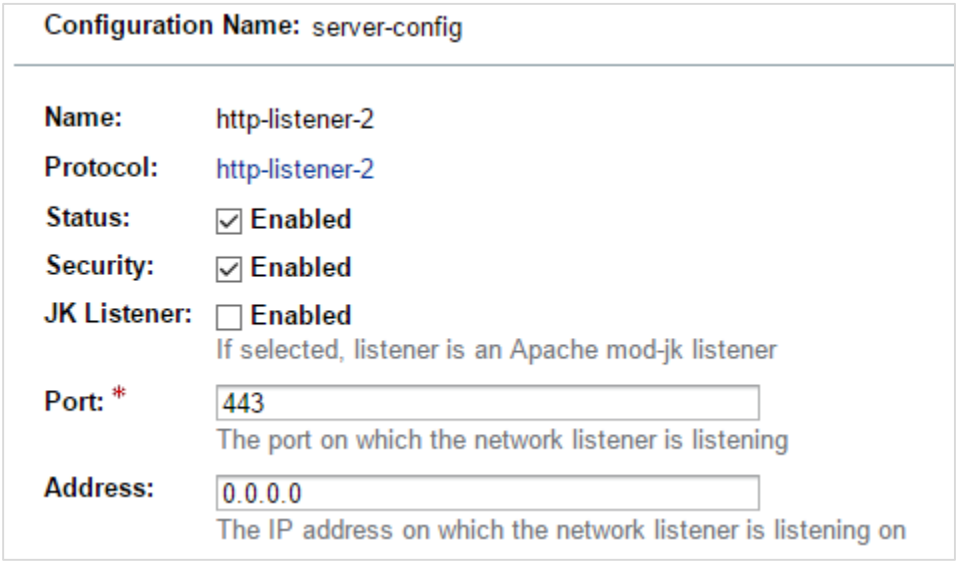

Ilustración 1.24: Puede notarse que se habilita la seguridad y se configura el puerto comúnmente usado para HTTPS, además de indicar 0.0.0.0 como dirección IP a fin de que escuche en cualquiera de ellas.

Luego hay que configurar en la sección de Seguridad de cada listener qué certificados y claves usar. Esto se logra, como muestra la imagen siguiente, indicando el NickName, alias o nombre con el cual identificamos nuestra clave privada y certificado digital en el keystore mencionado anteriormente.

Ilustración 1.25: Configuración de seguridad del listener HTTPS. Se habilita el protocolo TLS y se indica el alias o identificador que indica qué certificado digital y claves deben usarse.

Una vez configurado esto, el servidor responderá a solicitudes HTTPS y utilizará el protocolo TLS para establecer una conexión segura. Los clientes (navegadores) resaltan este hecho indicando el carácter seguro de la conexión y dando la posibilidad de visualizar el certificado digital que autentica el sitio.

Ilustración 1.26: Los navegadores web nos permiten visualizar el Certificado digital entregado por el sitio web que estamos accediendo. En la imagen puede verse el certificado que entrega nuestro sistema.

3.5.1.2. Implementación del diseño responsive

Para la creación de páginas Responsive se utilizaron algunos conceptos que plantea el framework Bootstrap junto con componentes provistos por PrimeFaces. A continuación se van a introducir brevemente algunos de estos conceptos:

- **Sistema de Rejilla**: El sistema de rejilla está pensado para ayudarnos en la disposición de los contenidos de nuestra web y su adaptación a los diferentes tamaños

de pantalla de forma automática. Para ello tenemos que poner el contenido dentro de celdas o columnas que irán dentro de filas, es decir, las filas se utilizan para agrupar horizontalmente a varias columnas. El contenido siempre se coloca dentro de las columnas, ya que las filas sólo deberían contener como hijos elementos de tipo columna. Las columnas de la rejilla definen su anchura especificando cuántas de las 12 columnas de la fila ocupan. Si el tamaño total de las columnas de una fila excede de 12, el tamaño sobrante se colocará en la siguiente fila.

- El sistema de rejilla de Bootstrap establece 4 tamaños diferentes de pantalla, con los correspondientes tres puntos de corte (tamaño de pantalla o viewport) y las ***clases CSS*** que nos permiten controlarlo:
    - Pantallas grandes (LG) a partir de 1200 pixeles
    - Pantallas medias (MD) a partir de 992 píxeles
    - Pantallas tipo tablet (SM) a partir de 768 pixeles
    - Pantallas móviles (XS), por debajo de esta última cifra.

  Básicamente, lo que hace Bootstrap es definir un modelo de divisor horizontal (una fila) divisible en 12 columnas, pero con la ventaja de que podemos decir cuántas hay en cada resolución de pantalla. Esto es más fácil de entender con un ejemplo: pensemos en una serie de bloques (imágenes, elementos html, etc.) que cuando estamos en pantalla LG (más de 1200 px) queremos que estén todos en fila. Pero que a medida que se vaya estrechando la pantalla, en vez de desaparecer de la vista, vayan saltando de línea: 3 por fila en resolución MD (de más de 992 px), 2 por fila en resoluciones SM (de más de 768 px), y uno por fila en móviles (pantalla XS). Para eso, tenemos que definir grupos de columnas en donde introducir los elementos. En la siguiente imagen vemos una representación del ejemplo:

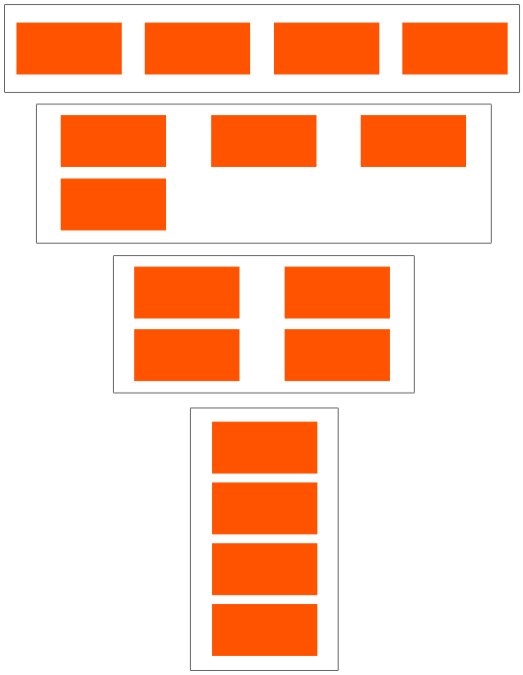

Ilustración 1.27: El primer caso es una pantalla LG, donde queremos 4 bloques por fila. Teniendo 12 columnas necesitamos 4 grupos de 3 columnas. Para una resolución MD queremos 3 bloques por fila, entonces agrupamos de a 4 columnas. Para resoluciones SM agrupamos de a 6 columnas obteniendo 2 bloques por fila, y para XS sólo habrá un bloque que abarque las 12 columnas.

- **Media queries**: En la mayoría de los casos gracias a todas las clases que provee Bootstrap nos será suficiente para componer nuestra web. Sin embargo, en algunas situaciones es posible que queramos modificar dicho comportamiento, por ejemplo para aplicar determinados estilos CSS (como colores, alineaciones, etc.) que cambien según el tamaño de pantalla. En estos casos será necesario que creemos nuestra propia media query para aplicar los estilos deseados. En este caso, los estilos que estén dentro de esta media query se aplicarán sólo a partir del tamaño en píxeles indicado.

Haciendo uso de estos conceptos, se diseñó prácticamente desde cero el layout (template) general del sistema. Esto fue así debido a que queríamos un diseño particular y adaptado a los lineamientos planteados por Material Design. Para esto se definieron clases y media queries propias. Una vez definido el layout, el contenido de cada página web fue adaptado a un diseño responsive. Además, como mencionamos anteriormente, se incorporaron muchas otras utilidades y complementos provistos por PrimeFaces como Tablas de Datos, Calendarios, etc., para simplificar el desarrollo de una web responsive.

Cuando se diseña un sitio web responsivo se tiene que considerar una enorme cantidad de dispositivos donde el contenido va a ser presentado. Utilizamos la filosofía

Mobile First para organizar el código CSS que permitirá adaptar el contenido a los diferentes tamaños de pantalla.

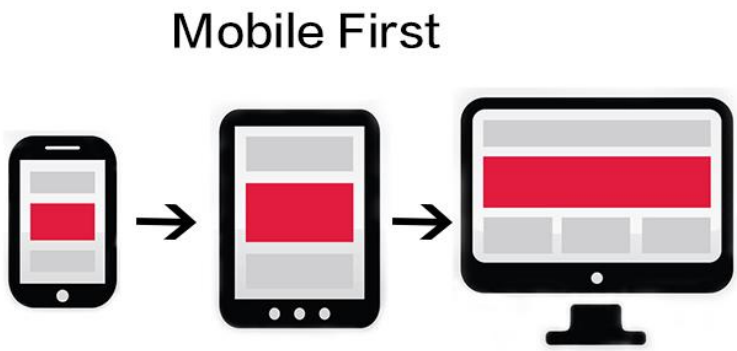

Ilustración 1.28: Filosofía Mobile First. Se diseñan las interfaces de usuario comenzando por las pantallas pequeñas. Luego se reordenan los elementos de la vista para llenar pantallas de mayor tamaño.

Mobile First plantea comenzar el diseño de las interfaces de usuario por la pantalla más pequeña e ir adaptándolo posteriormente a las más grandes. De esta forma lo que conseguimos es centrarnos en lo más importante, aquello imprescindible para la web de acuerdo a los objetivos que tenga, que se verá reflejado en la web mobile. Y así, al diseñar las webs del resto de dispositivos, no habrá más que aumentar el tamaño de los elementos e incorporar otros, si es necesario, conforme vaya creciendo el tamaño de la pantalla con la que se esté trabajando. Esta filosofía la plasmamos luego en el código.

3.5.1.3. Patrón MVC

La implementación hecha del patrón MVC puede explicarse mejor si describimos un caso sencillo, donde nombramos los componentes concretos que usamos y sus relaciones. Vamos a usar un caso relacionado con el módulo de pacientes. Los elementos implicados en este ejemplo son:

- Paciente.java (modelo): entidad sin comportamiento que representa a los pacientes y contiene el mapeo con el modelo relacional.
- PacienteFacade.java (modelo): también denomiado servicio, o gestor, posee comportamiento relacionado a la lógica de gestión de pacientes.
- PacienteController.java (controlador): se ubica entre el modelo y la vista, adapta la información del modelo para mostrarla en la vista, mantiene el estado de las vistas, recibe y ejecuta la lógica de las llamadas AJAX y redirecciones, se comunica con el modelo, muestra errores de validación, permiten acceder a los parámetros de las solicitudes http, entre otras cosas.

- PacienteList.xhtml (vista): interfaz gráfica encargada de visualizar y permitirle al usuario interactuar con la información relacionada a los pacientes.

El caso comienza cuando, a través de una URL, se solicita la lista de pacientes. El framework se encarga de interpretar la URL y sus parámetros. A partir de esto sabe que vista procesar y mostrar. Para procesar el contenido de la vista, es necesaria una instancia del PacienteController. El controlador en este caso se encarga de consultar al modelo por la lista de pacientes para que la vista pueda mostrarlos. Para esto, el controller usa una instancia del PacienteFacade, que se encarga de acceder a la base de datos. Una vez que la base de datos responde, el PacienteFacade devuelve una lista de entidades Paciente al PacienteController. En este momento el framework está en condiciones de finalizar el procesamiento (o renderizado) de la Vista para producir el resultado final, que es la tabla con la lista de pacientes. La tecnología soporta el binding entre la lista mostrada al usuario y la lista de entidades Paciente que representa el modelo.

Ciclo de vida de los controladores

Diseñamos los componentes del modelo (facades, gestores o servicios) de forma que no posean *estado,* es decir, que no recuerden, por ejemplo, el paciente que estamos editando. El modelo simplemente se limita a recibir solicitudes y responder. La responsabilidad de mantener el estado la trasladamos a los controladores.

Los controladores se mantienen *vivos* (junto con sus propiedades, como el paciente actualmente editado) mientras no cambiemos de URL. Esto se logra definiendo el alcance o *scope* de cada controlador. Definir un *scope* implica definir cuánto vivirá un controlador. La mayoría fueron configurados con un scope llamado *ViewScoped,* que tiene el comportamiento anteriormente descrito: el controlador no se descarta mientras que no se cambie la URL. Otros scopes que utilizamos son el RequestScoped, SessionScoped y ApplicationScoped. A continuación definimos brevemente cada uno:

- RequestScoped: el controlador vive tanto como un ciclo de solicitud y respuesta HTTP. Las solicitudes AJAX cuentan también como una única solicitud-respuesta. Es decir, el controlador se crea para atender una solicitud y luego es descartado.
- SessionScoped: el controlador vive tanto como lo haga la sesión HTTP que está asociada a la sesión del usuario, que comienza cuando el mismo se loguea y finaliza cuando cierra su sesión o transcurre un determinado tiempo. Lo usamos para controladores que deben mantener estado relacionado a los usuarios.

- ApplicationScoped: el controlador vive mientras la aplicación web se ejecute. Lo empleamos para mantener el contenido de listas desplegables y contenido transversal a todos los usuarios. Por ejemplo, las obras sociales, o tipo de tratamientos.

Por lo general cada vista tiene asociada uno o más controladores. Usar controladores ViewScoped permite que el usuario interactúe en pestañas diferentes del navegador con la misma vista (edición de tratamiento por ejemplo) y para cada pestaña se emplea un conjunto de controladores independiente. Esto no ocurriría si hubiésemos empleado controladores SessionScoped, habría conflictos entre los datos mostrados en las distintas pestañas.

3.5.1.4. Procesos batch

Java EE 7 cuenta con una API destinada a la ejecución de procesos batch. El servidor de aplicaciones implementa esta API y nos permite definir procesos o trabajos de este tipo, para que luego sean ejecutados a demanda o periódicamente en un determinado horario. Esto último es lo que necesitamos: ejecutar uno o más procesos batch periódicamente. Para esto es necesario definir en un archivo xml especial las clases que van a llevar a cabo el procesamiento, y por otro lado, es necesario crear un EJB que llame al proceso batch en un determinado horario. Esto último se logra anotando el método del EJB que desencadena el procesamiento con la *annotation* @Schedule.

Esta API fue utilizada para la *Marcación de sesiones como Transcurridas*. Al finalizar un día laboral, entre el rango horario de las 9 de la noche y las 6 de la mañana (horario durante el cual no se hace uso del sistema), se corre un proceso batch el cual se encarga configurar el estado “Transcurrida” de las sesiones del día que pasó, cambiándolo de *false* a *true*.

Más adelante, con el surgimiento de nuevos requerimientos, se incorporó a esta arquitectura ya existente otros procesos para correr de forma programada. Hacemos referencia a la generación de notificaciones para el usuario.

3.5.1.5. Páginas bookmarkables

El framework web que se utiliza es del tipo *basado en componentes*. Este tipo de frameworks se encarga de procesar él mismo las solicitudes y respuestas http en lugar de dejar esta tarea al desarrollador. Esto nos ahorra el trabajo de escribir el código necesario para gestionar dichos mensajes http, pero por otro lado genera que los parámetros de las solicitudes se oculten en el cuerpo de las mismas, (ya que por defecto se realizan solicitudes POST), causando que una URL no sirva para acceder a un contenido determinado.

El funcionamiento por defecto podemos verlo con un ejemplo. Si estamos en la agenda y solicitamos la información de una sesión, se realiza la siguiente solicitud:

```
URL:http://deltagestion.com.ar/deltagestion/faces/protected/agenda/Agenda.xhtml
Request Method: POST
```

Ilustración 1.29: Por defecto, las solicitudes HTTP se realizan mediante el método POST y sin necesidad de añadir parámetros en la URL.

El cuerpo de la solicitud contiene, entre otros datos, los siguientes:

...	
javax.faces.partial.event	eventSelect
AgendaForm:schedule_selectedEventId	**1912**
AgendaForm	AgendaForm
...	

Ilustración 1.30: Contenido de una solicitud POST para mostrar los datos de una determinada entidad. El dato resaltado nos permite acceder al recurso o entidad que se quiere mostrar.

En la imagen anterior puede verse que el dato necesario para acceder a la entidad que deseamos está embebido en el cuerpo de la solicitud, de modo que si refrescamos la página solicitando la misma URL pero sin ningún dato adicional, se nos mostrará la agenda pero no los detalles de la sesión `id=1912`.

Lo descrito anteriormente, hace que se pierda un atributo deseable en cualquier sitio web: la capacidad de las páginas de ser bookmarkables. A partir de este inconveniente, hicimos las modificaciones necesarias para corregirlo haciendo que los parámetros de las solicitudes se encuentren en la URL, algo que resulta intuitivo pero la tecnología no lo implementa por defecto de esta manera. Es decir, en las secciones que deseamos obtener páginas bookmarkeables empleamos el método GET para solicitarlas. Algunos ejemplos:

```
http://deltagestion.com.ar/.../tratamiento/List.xhtml?paciente=116
http://deltagestion.com.ar/.../editTratamiento/EditTratamiento.xhtml?tratamiento=8
```

Ilustración 1.31: Ejemplos de solicitudes HTTP GET que proveen a las páginas de la capacidad de ser bookmarkeables. En estos ejemplos, a diferencia de los anteriores, si refrescamos la página accederemos a los datos de la entidad indicada en los parámetros de la URL.

3.5.1.6. Tecnologías usadas en las capas

Cada una de las capas planteadas en el diseño de la arquitectura se distingue por tener una responsabilidad determinada. La misma separación se da en la tecnología usada. La plataforma Java EE se compone de diversas API o módulos que apuntan a solucionar una problemática concreta. El mapeo entre las capas y la tecnología es el siguiente:

- Capa de presentación: En esta capa se encuentra implementado el patrón MVC. Las vistas están implementadas con JSF y PrimeFaces. Los controllers se inyectan en las

vistas usando CDI, mientras que el modelo se encuentra en la capa de negocio. JSF, a su vez, al tratarse de un framework de UI web, depende de tecnologías como CSS, el lenguaje JavaScript y HTML.

- Capa de negocio: La capa de negocio está pensada para que preste servicios a la capa de presentación o a cualquier otro cliente, y no se preocupe de mantener estados o conversaciones. Los EJB están pensados para esto, además son transaccionales y pueden inyectarse también usando CDI, en controllers u otros EJB. Para los procesos batch se emplea la API de Batch Processing. Para las validaciones al momento de crear o modificar entidades, se usan funcionalidades de JPA. JPA también es usado en la capa de acceso a datos.
- Capa de acceso a datos: La capa de acceso a datos principalmente hace uso de JPA, que se encarga del mapeo entre objetos y tablas relacionales, de gestionar la carga de entidades, del uso de cache, y, en conjunto con la tecnología EJB, nos brinda transaccionalidad a alto nivel (a nivel de las llamadas a métodos).
- Aspectos transversales: Para la seguridad se emplea otra de las API incluidas en Java EE, la Security API, que permite dar seguridad tanto al momento de acceder a las páginas (recursos) como cuando se intentan llamar métodos del dominio.

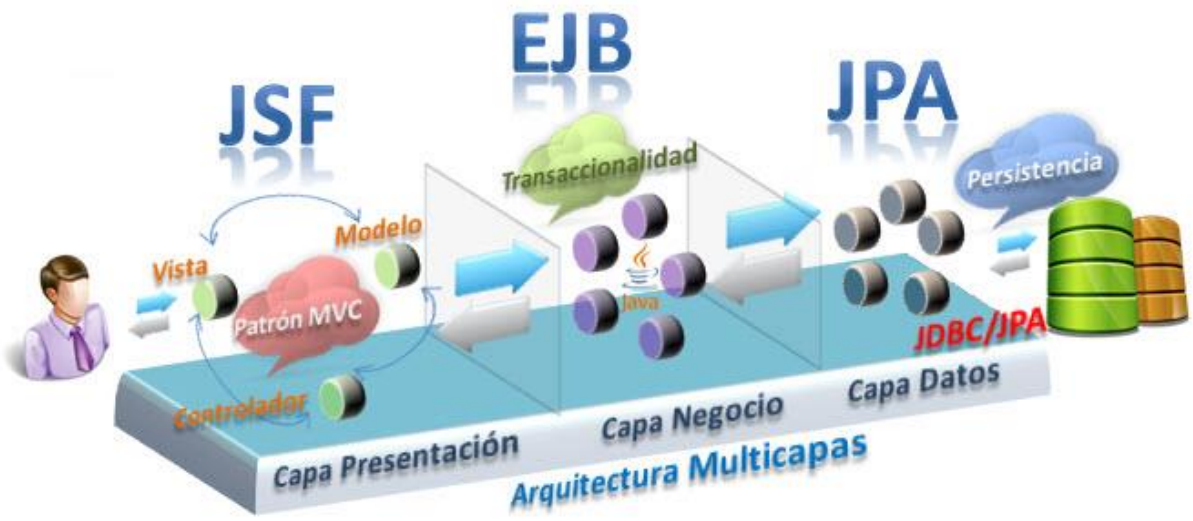

Ilustración 1.32: Java EE nos provee de tecnologías para todas las capas de nuestra arquitectura. En la capa de presentación la más importante es JavaServer Faces, mientras que Enterise Java Beans lo es para la capa de negocio y Java Persistence API para la de datos.

3.5.2. Desarrollo de los módulos

Una vez que teníamos desarrolladas cuestiones transversales, como la seguridad y el patrón MVC, comenzamos por desarrollar el módulo de pacientes. Este planteo nos pareció el más adecuado ya que los demás módulos dependen del de pacientes. Haber comenzado por

aquí nos permitió hacer los primeros ABM, que los tomamos como un caso base, aprovechando que no existe mucha complejidad técnica ni de negocio. Implementar esta funcionalidad implica desarrollar desde los formularios de la interfaz de usuario y sus validaciones, hasta la persistencia en la base de datos, sin complicaciones extras, permitiéndonos asimilar el camino completo de los datos y la interacción básica entre los componentes.

Desarrollar las funcionalidades del ABM de pacientes nos llevó más tiempo del que uno podría estimar para este tipo de de trabajo. El motivo fue que debimos entender los aspectos de la tecnología que nos quedaron sin cubrir en las iteraciones anteriores. Más precisamente aprendimos a cómo y dónde hacer las validaciones de los datos ingresados por el usuario; cómo mostrar los mensajes de error; cómo trabaja JPA respecto a modificaciones y eliminaciones de entidades; a utilizar los componentes de PrimeFaces teniendo un primer contacto con tablas de datos, formularios, botones, menús, etc.; a manejar las capacidades AJAX de JSF y PrimeFaces entendiendo su flujo de información; comprender el ciclo de vida de páginas y los controladores.

Luego, según lo planificado, continuamos con el módulo de tratamientos. Desarrollamos la página donde se listan los tratamientos de cada paciente, y los formularios de alta y modificación. Los tratamientos están estrechamente relacionados con las sesiones, la agenda y las órdenes médicas. Los siguientes pasos fueron explorar las facilidades que la tecnología nos daba para visualizar y gestionar las agendas y sesiones.

PrimeFaces cuenta con más de 100 componentes que los desarrolladores pueden reutilizar. Hay desde componentes sencillos como una barra de menú, hasta componentes más complejos como la agenda. Cada componente complejo tiene asociado por lo general un modelo que soporta sus funcionalidades. En el caso de la agenda o schedule, que es el componente que necesitábamos, son dos las interfaces más importantes de su modelo: la interfaz ScheduleModel y la interfaz ScheduleEvent. ScheduleEvent representa un evento en el tiempo, con una fecha y hora de inicio y de fin, un título, una descripción, entre otros atributos. ScheduleModel representa un conjunto de ScheduleEvent, y soporta las operaciones de agregar, modificar, eliminar y obtener la totalidad de eventos.

Para interactuar con el modelo de PrimeFaces es necesario implementar las interfaces mencionadas. Para que nuestro modelo de negocio conviva con el modelo que nos provee

PrimeFaces, es decir, para poder manejarnos con nuestras clases Agenda y Sesión, éstas implementan las interfaces ScheduleModel y ScheduleEvent respectivamente.

Una vez realizado lo anterior se creó la página para la vista de agenda, conteniendo el componente o widget de PrimeFaces y las funcionalidades de creación y modificación de sesiones.

Se continuó desarrollando la funcionalidad de editar un tratamiento de un paciente. Una vez que sabíamos cómo implementar las sesiones y la agenda, agregamos a la edición de tratamientos la capacidad de agregar, modificar y quitar sesiones, con todas las validaciones que esto implica.

Como se explicó en las secciones referidas al dominio del sistema, los tratamientos poseen sesiones que en caso de realizarse por medio de una obra social deben ser cubiertas por órdenes médicas. Las órdenes médicas son cargadas dentro de la edición de cada tratamiento. Por lo tanto las siguientes funcionalidades desarrolladas fueron las de creación y autorización de órdenes médicas.

Luego de contar con los mecanismos de alta y autorización de órdenes médicas se agregaron las características de búsqueda, filtrado y posterior conformación de la planilla de órdenes médicas.

El módulo de caja fue dejado para lo último debido a su baja complejidad y a que se encuentra aislado de los demás módulos.

3.6. Pruebas

3.6.1. Introducción al proceso de pruebas utilizado

Dado que la metodología de desarrollo que adoptamos está guiada por la estructura del Proceso Unificado de desarrollo, las tareas de prueba de software se han dividido a través de las diferentes fases del mismo. En cada fase en que las pruebas estuvieron involucradas, el tipo y carga de las mismas fue diferente y apuntó a un propósito específico.

- Fase de Elaboración: en esta fase el proceso de prueba apuntó a la obtención y refinamiento de requerimientos y casos de uso. La metodología en este punto se basó en la construcción de una planilla de casos de pruebas iniciales.
- Fase de Construcción: en esta fase, el proceso de pruebas se vio dirigido por tres sub-objetivos principales:

- El primero de ellos fue refinar los casos de prueba generados inicialmente, de forma tal de completar los requerimientos y casos de prueba para las funcionalidades a ser desarrolladas.
- Ejecución de las pruebas en forma manual, registración de casos de excepción no contemplados en las pruebas y actualización de casos de prueba.
- Automatización de casos de prueba principales, así como también el desarrollo de pruebas automatizadas de integración de sistema.

3.6.2. Introducción a los tipos de pruebas utilizadas

Existen diferentes tipos de pruebas que pueden ejecutarse sobre el software. Para el proceso de prueba llevado a cabo utilizamos el Modelo en V como guía.

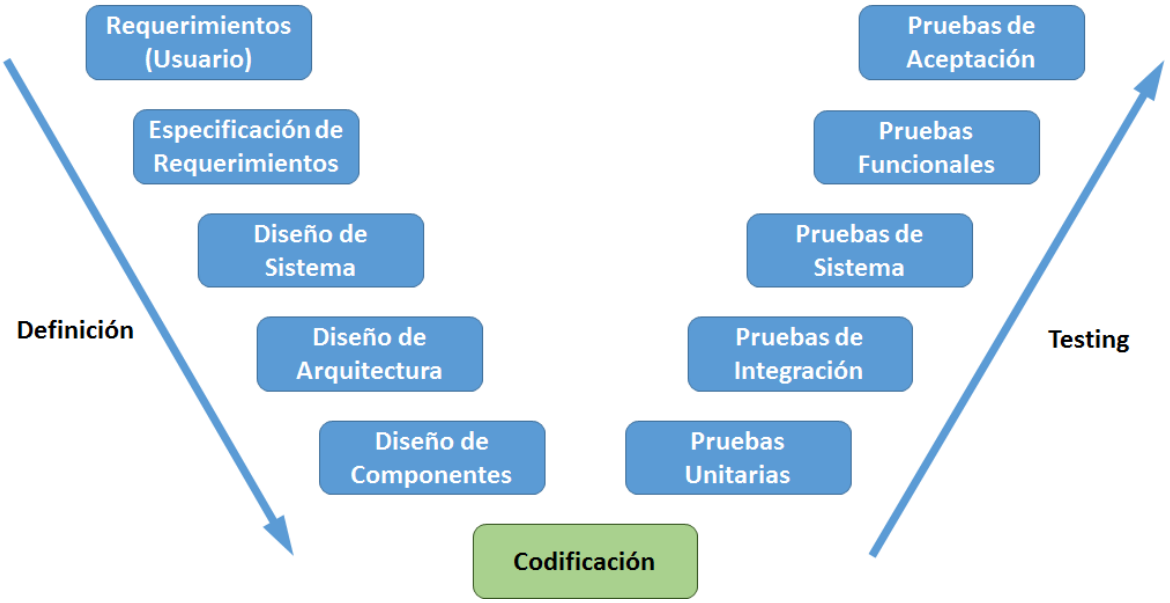

Ilustración 1.33: Para definir nuestro proceso de pruebas se tomaron elementos del Modelo de desarrollo en V.

"A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución. Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas (acciones para asegurar la calidad) que validan cada uno de los modelos creados cuando el equipo fue hacia abajo por el lado izquierdo."[9]

Si bien tomamos el modelo en V como referencia, adaptamos el mismo a las necesidades y características de nuestro proceso de desarrollo:

- Niveles: No todos los niveles de pruebas del proceso fueron implementados. Se hizo mayor hincapié en las pruebas funcionales del sistema, y se desarrollaron pruebas de integración y pruebas unitarias.
- Cobertura: En ninguno de los niveles se implementaron pruebas de todo el sistema. La mayor cobertura la tuvieron las pruebas funcionales del sistema. Luego podríamos decir que le siguen las pruebas de integración y por último las unitarias. Establecimos este nivel de importancia de los distintos niveles de pruebas fundamentalmente porque consideramos que las pruebas de unidad e integración, en el marco de la metodología empleada, deberían servirnos para encontrar errores y detectar regresiones. En la sección Elaboración de casos de prueba de bajo nivel ampliamos esta idea.

3.6.3. Proceso de Pruebas

3.6.3.1. Pruebas en la etapa de Elaboración del sistema

3.6.3.1.1. Proceso de elaboración de casos de prueba

En la fase de Elaboración del sistema el objetivo principal fue completar la especificación de requerimientos y elaborar un diseño detallado de la solución. A estos propósitos, se apuntó a elaborar desde la perspectiva de pruebas una planilla con pruebas del sistema. Las mismas apuntaron específicamente a pruebas de aceptación del sistema, es decir, pruebas dirigidas a validar requerimientos funcionales del sistema. “La prueba de aceptación es el proceso de comparar el programa con sus requerimientos iniciales y con las necesidades actuales del usuario final” [12]. En este punto, nos situamos en el nivel más alto de pruebas en cuanto a lo propuesto por el modelo en V. Sin embargo, como sostiene Myers [12] “es un tipo de test poco usual que es ejecutado generalmente por el cliente o usuario final y normalmente no se considera responsabilidad de la organización de desarrollo”. Para validar el sistema desde el punto de vista del usuario final y asegurarnos que cumplimos con los requerimientos funcionales del sistema, Myers [12] propone un tipo de prueba al que llama prueba funcional del sistema. Este tipo de prueba consiste en elaborar un conjunto de casos de prueba derivados de la especificación del comportamiento esperado del sistema; en nuestro caso esto se trata de los requerimientos funcionales y los casos de uso.

El proceso de elaboración de los casos de pruebas en esta etapa se ligó mucho a los casos de usos identificados para el sistema. Una estructura general del procedimiento utilizado puede enunciarse como sigue:

1. Seleccionar un caso de uso.

2. Elaborar un caso de prueba para el flujo principal del caso de uso.
3. Elaborar un caso de prueba para cada flujo alternativo del caso de uso.
4. Preguntarse: ¿Existe alguna excepción al flujo principal que no hayamos considerado?
    a. Si existe, tomamos nota de la misma para un posterior análisis.
5. Preguntarse: ¿Existe algún caso negativo o de falla para alguno de los casos positivos enunciados, que no hayamos considerado aún?
    a. Si existe, tomamos nota del mismo para un posterior análisis.
6. Preguntarse: ¿Existe algún caso de éxito que no hayamos considerado aún?
    a. Si existe, tomamos nota del mismo para un posterior análisis.
7. Revisamos los casos adicionales detectados para el caso de uso en los puntos 4, 5 y 6, y nos preguntamos para cada uno de ellos.
    a. ¿Conocemos el resultado esperado para este caso?
        i. Si lo conocemos, elaboramos un caso de prueba para el nuevo caso encontrado.
        ii. Sino, anotamos el caso como punto a revisar en la próxima reunión con el cliente.

Además, nos planteamos algunos lineamientos de alto nivel a la hora de elaborar los casos de prueba (puntos 2, 3 y 7.a.i):

- De ser posible, utilizar técnicas de clases de equivalencia y valores límites; esto lleva a que un caso de prueba se separe en varios casos con el mismo flujo pero utilizando diferentes entradas.
- En lo posible, cada test debía ser autónomo; esto quiere decir que cada caso de prueba tenía un objetivo específico, y su ejecución no dependía de la ejecución de otros casos de prueba.

De esta manera, al concluir el proceso antes descrito, contamos con dos documentos bien definidos:

- Planillas de casos de prueba, donde cada caso de prueba proviene directamente de un caso de uso.
- Documento con puntos a discutir con el cliente, potenciales nuevos requerimientos a incorporar al sistema.

Puede verse claramente cómo a través de la especificación de casos de prueba fueron surgiendo ciertos casos que al ser discutidos con el cliente terminaron por transformarse en nuevos requerimientos a ser incorporados a los casos de usos y los casos de prueba.

3.6.3.1.2. Formato de los casos de prueba documentados

En un principio, todos los casos de pruebas formaban parte de una única lista, donde el formato de cada uno consistía de:

1. Número: un valor que identifica el caso de prueba unívocamente.
2. Objetivo de la prueba: una oración que a simple vista deja en claro qué es lo el caso pretende probar.
3. Ejecución: una secuencia de pasos de alto nivel que indica de qué manera se va desarrollando el flujo del caso de prueba.
4. Resultado esperado: una serie de resultados que se espera sean ciertos luego de ejecutarse la prueba.

El siguiente es un caso de prueba especificado utilizando dicho formato:

Numero	34
Objetivo	Eliminar un tratamiento cuando es el único en la lista
Ejecución	1. Busque el paciente que desea manualmente o a través de la búsqueda de pacientes; el mismo debe poseer un único tratamiento. 2. Posiciónese sobre el registro de la tabla que contiene el paciente deseado, y seleccione la opción Tratamientos. 3. Seleccionar la opción “Eliminar” para el tratamiento mostrado.
Resultado Esperado	- El tratamiento es eliminado. - La tabla desaparece. - Se muestra el mensaje: "El paciente no posee tratamientos cargados"

Tabla 1.2: Formato con el cuál inicialmente se documentaron los casos de prueba.

A medida que aumentaba la cantidad de casos de prueba nos dimos cuenta que necesitábamos agrupar los casos de prueba de alguna forma más manejable. Por este motivo, la lista se seccionó en varias sub-listas de casos de prueba agrupados funcionalmente. De esta forma, para cada módulo del sistema contábamos con los casos de prueba relacionados a dicho módulo agrupados bajo una misma lista. Dado dicho agrupamiento, optamos por re-enumerar los casos de prueba dándole a cada uno un identificador del tipo “AA##” donde:

- AA es una letra o secuencia de letras mayúsculas que identifican el módulo del sistema en el cual se encuentra inmerso el caso de prueba.
- ## es el número del caso de prueba dentro de dicho módulo.

De esta forma, un caso de prueba queda unívocamente identificado no sólo dentro del conjunto de todos los casos de prueba del sistema, sino que además teníamos una forma rápida de identificar el caso de prueba dentro del módulo del sistema al cual se refería.

Al mismo tiempo, a medida que el número de casos de prueba que se desprendían de un mismo caso de uso crecía, surgió la necesidad de agregar un nuevo campo a los casos de uso: el campo "precondiciones". Este campo nos brindó la posibilidad de especificar el contexto en el cual un caso de prueba (así como sucedía con los casos de uso) comenzaba a ejecutarse. Así obtuvimos para cada caso de prueba una especificación no ambigua que nos permitía identificarlo claramente y ejecutarlo en una etapa posterior del desarrollo del sistema.

Con estos cambios, el caso de prueba antes expuesto queda como sigue:

Identificador	T6
Objetivo	Eliminar un tratamiento cuando es el único en la lista
Precondiciones	- Existe un paciente que posee un solo tratamiento creado. - El usuario está ubicado en la pantalla de lista de tratamientos de dicho paciente.
Ejecución	1. Seleccionar la opción "Eliminar" para el tratamiento mostrado.
Resultado esperado	- El tratamiento es eliminado. - La tabla desaparece. - Se muestra el mensaje: "El paciente no posee tratamientos cargados"

Tabla 1.3: Formato con el cuál se documentaron los casos de prueba.

3.6.3.2. Pruebas en la fase de Construcción del sistema

Llegada la fase de Construcción, ya contábamos con una especificación de casos de uso casi completa, un diseño detallado del sistema y sus componentes, un diseño de interfaz de usuario detallado y una estructura arquitectónica base para el sistema. El siguiente paso era comenzar con la codificación del sistema.

En esta fase, el proceso de pruebas se segmentó en dos grandes aspectos que envolvieron la codificación del módulo de sistema bajo desarrollo en cada iteración de la fase de Construcción:

- Proceso anterior a la codificación del módulo: refinamiento de casos de prueba y elaboración de casos de prueba de bajo nivel para el módulo en cuestión.
- Proceso posterior a la codificación del módulo: codificación de pruebas de bajo nivel para el módulo, ejecución manual de pruebas funcionales y automatización de las mismas.

A continuación se detallan las actividades comprendidas en ambos procesos.

3.6.3.2.1. Refinamiento de casos de prueba

Llegado el momento de construir un módulo o funcionalidad del sistema, nos encontramos que los casos de prueba de alto nivel planteados en la fase de elaboración podían refinarse y adaptarse a lo que habíamos especificado como el diseño de interfaz de usuario.

Para el refinamiento de los casos de prueba, analizamos los casos de prueba existentes de la siguiente manera:

1. Analizamos las precondiciones y nos preguntamos: ¿existe alguna precondición que falte enunciar en términos de interfaz de usuario y el sistema? De no cumplirse la misma: ¿estamos ante un caso de prueba diferente?
2. Analizamos el flujo de ejecución del caso de prueba y nos preguntamos: ¿existe algún paso en término de interfaz de usuario que esté faltando en el flujo? ¿Algún paso debe cambiarse o suprimirse debido al flujo del sistema?
3. Analizamos el resultado esperado y nos preguntamos: ¿existe algún resultado esperado que debamos agregar en términos de interfaz? ¿Existe algún resultado esperado que debe dividirse en varios resultados esperados en términos de interfaz?

En cada punto se modifica el aspecto del caso de prueba que corresponda de acuerdo a las respuestas a dichas preguntas.

Luego de refinar cada caso de prueba, se validaron dos cosas para asegurarnos que el caso de prueba seguía teniendo la esencia que tenía antes de ser refinado:

- El caso de prueba aún respondía al mismo comportamiento funcional.
- El caso de prueba contenía suficiente detalle como para que alguien que no está familiarizado con la funcionalidad pueda ejecutarlo.

Si ambos puntos eran ciertos, entonces el refinamiento del caso estaba completo. De lo contrario, se modificaba el mismo hasta que lo fuera.

Además de refinar el caso de prueba, se anotaban nuevos casos de prueba que surgían de refinar el caso de prueba con detalles de interfaz. Por ejemplo, un caso ligado a la interfaz que se repitió para varias interfaces fue cancelar la carga de un formulario y que al abrirlo nuevamente el formulario estuviese en su estado inicial. Estos nuevos casos se agregaban a la lista de casos de prueba.

Además de refinar casos existentes y agregar nuevos casos, también decidimos reordenar los casos de prueba de forma que tuviesen cierta correspondencia con la utilización

del sistema. Una estructura común adoptada, por ejemplo, para los casos de pruebas de formularios fue la siguiente:

1. Casos de validación de campos obligatorios.
2. Casos de validación de formatos de datos de los campos.
3. Casos de validación lógica del formulario (por ejemplo, que no se pudiera crear una sesión para un tratamiento si éste estaba completo).
4. Casos de cancelación de carga del formulario.
5. Casos de carga exitosa de un formulario.
6. Casos de validación de campos obligatorios al editar el formulario.
7. Casos de validación de formatos de datos de los campos al editar el formulario.
8. Casos de validación lógica del formulario al editarlo.
9. Casos de cancelación de edición del formulario.
10. Caso de edición exitosa del formulario.

Del refinamiento de los casos de prueba surgieron nuevas cuestiones a ser revisadas con el cliente. Por ejemplo, al plantear casos de creación, edición y eliminación de tratamiento en el sistema se nos plantearon preguntas como:

- ¿Debería crearse el tratamiento sólo ante un set específico de condiciones?
- ¿Qué sucede si editamos un tratamiento cambiando el tipo de tratamiento?
- ¿Deberíamos permitir que un tratamiento sea borrado por completo del sistema si posee órdenes asignadas o sesiones programadas?

Una vez registrados los nuevos casos a discutir con el cliente, dimos por concluida la etapa de refinamiento de casos de prueba.

3.6.3.2.2. Elaboración de casos de prueba de bajo nivel

El Modelo en V ubica por debajo de las pruebas de Aceptación y Funcionales las pruebas de Sistema. Myers [12] subdivide las pruebas de Sistema en varias categorías, algunas de ellas se describen a continuación:

- Pruebas de volumen: el sistema es sometido a trabajar con grandes volúmenes de datos.
- Pruebas de esfuerzo: se prueba el sistema para ver si éste soporta una carga grande de entradas en un período corto de tiempo.
- Pruebas de usabilidad: se prueba el sistema para encontrar problemas que surgen del uso en sí del sistema.

- Pruebas de seguridad: el propósito de este tipo de pruebas es encontrar casos de prueba que violen la seguridad del sistema.
- Pruebas de *performance*: se prueba el sistema para ver si éste cumple con los parámetros de desempeño esperados.
- Pruebas de configuración, pruebas de confiabilidad, pruebas de instalación…

Ninguno de los atributos de calidad externa requeridos (ver sección Arquitectura y calidad) justificaba incorporar pruebas para dichos requerimientos: respecto a seguridad empleamos mecanismos bien conocidos (y probados), y con respecto a pruebas de usabilidad serían llevadas a cabo por el usuario final del sistema a través de las pruebas de aceptación.

Por debajo de las pruebas de sistema se ubican las **pruebas de integración y las pruebas unitarias**. Las pruebas de integración apuntan a encontrar problemas en las interfaces de los componentes de un sistema cuando estos interactúan, mientras que las pruebas unitarias apuntan a probar un elemento del sistema aislado del resto (por ejemplo una clase o un método particular).

Según introducimos en la sección Introducción a los tipos de pruebas utilizadas, ponderamos las pruebas de integración y de unidad según la utilidad que consideramos que tienen a la hora de encontrar errores y detectar regresiones. Es decir, que la importancia que le dimos a los distintos niveles de pruebas, fundamentalmente en cobertura y esfuerzo dedicado, se condice con el rol que cumplen para detectar fallas. Así, creemos que es más útil un test de integración que uno unitario en este sentido. Sin embargo, no descartamos los test de unidad ya que nos sirven para detectar errores ante refactorizaciones. Seguramente en un proceso de desarrollo guiado por pruebas (TDD) las pruebas unitarias tomen un rol central, pero no es nuestro caso.

3.6.3.2.3. Codificación y ejecución de pruebas Unitarias y de Integración

Para la codificación y primera ejecución de las pruebas unitarias y de integración se siguió un enfoque sencillo del tipo bottom-up:

1. Se codificaron las pruebas unitarias de las clases seleccionadas.
2. Para cada clase o módulo a probar:
    a. Se corrieron las pruebas codificadas.
    b. Si el elemento bajo prueba arrojó errores, se corrigieron los errores y se volvió al paso 2.a.

3. Una vez que todas las pruebas unitarias pasaron, se codificaron las pruebas de integración; algunas de ellas utilizaban los elementos probados en el paso 2, mientras que otras no.
4. Una vez codificadas las pruebas de integración, para cada una:
    a. Se ejecutó la prueba de integración.
    b. Si se encontró algún error de integración debido a algún error en uno de los elementos que interactúan:
        i. Se generó la prueba unitaria necesaria para el elemento en cuestión.
        ii. Se arregló el problema en ese elemento.
        iii. Se corrió la prueba unitaria para validar que el problema esté solucionado.
        iv. Si lo estaba, se volvió al paso 4.a con el test de integración que generó el problema.
    c. Si se encontró algún error en la interacción de los elementos:
        i. Se analizó la interfaz de qué elemento modificar.
        ii. Se modificó la interfaz del elemento.
        iii. Se modificaron (o crearon) las pruebas para la nueva interfaz del elemento.
        iv. Se corrieron las pruebas unitarias del elemento para validar que el mismo funciona correctamente.
        v. Se volvió al paso 4.a con el test de integración que generó el problema.

Este procedimiento es llevado a cabo ante cada nueva prueba codificada. Esto genera que iteración tras iteración se aporten nuevos test a la base de pruebas del sistema. Queda por definir *cuándo* y *cómo* deben ejecutarse las pruebas de integración y de unidad.
Con respecto al *cómo*, decidimos que todas las pruebas de integración y de unidad sean codificadas para ejecutarse de forma automática, es decir, que no requieran intervención del desarrollador.

Con respecto al *cuándo,* definimos criterios para decidir cuándo ejecutar cada tipo de prueba, y, por otro lado, para la parte técnica nos apoyamos en la herramienta de gestión y construcción (build) de proyectos Java que utilizamos: Maven. El criterio tiene que ver con tratar de intervenir lo menos posible en el proceso de desarrollo al mismo tiempo que detectar las regresiones y errores lo antes posible. Naturalmente las pruebas de integración demandan más tiempo de ejecución, ya que previamente se debe inicializar y configurar el entorno

donde se ejecutarán, además de recordar el hecho de que existe un volumen mayor de estas pruebas con respecto a las pruebas de unidad, tal como se mencionó en la sección anterior. Esto nos llevó a determinar que:

- Las pruebas de unidad sean ejecutadas luego de cada compilación.
- Las pruebas de integración sean ejecutadas luego cada contribución importante al proyecto que implique cambios en diversas partes del sistema y que se considere que pueda tener efectos secundarios.

Volviendo a la parte técnica, Maven divide el proceso de compilación o construcción (build life cycle) en una serie de fases. La configuración por defecto (y la que utilizamos) propone las siguientes fases:

- validate
- compile
- test
- package
- integration-test
- verify
- install
- deploy

Las fases que nos interesan para el testing son *test* e *integration-test*. Lo interesante es que desde un principio la herramienta prevé la separación de los distintos tipos de pruebas. Esto nos permite agrupar las pruebas y elegir cuándo ejecutar las de unidad y cuándo las de integración. La practicidad está en que las clases que contienen las pruebas deben ser situadas en el directorio de testing previsto por Maven, y según el nombre que tengan son ejecutadas en una fase de compilación o en otra. Es decir, existe una convención de nombres para distinguir las pruebas unitarias de las de integración. La convención es la siguiente:

- Las clases ubicadas en el directorio de pruebas de Maven cuyo nombre tenga como sufijo la palabra *Test* serán ejecutadas en la fase *test* de Maven.
- Las clases ubicadas en el directorio de pruebas de Maven cuyo nombre tenga como sufijo las letras *IT* serán ejecutadas en la fase *integration-test* de Maven.

Como resultado tenemos una serie de fases de compilación, de las cuáles la fase *test* con las pruebas unitarias es ejecutada siempre, y en los casos que se requiera ejecutar las pruebas de integración ejecutamos a demanda la fase *integration-test*.

3.6.3.2.4. Ejecución manual de las pruebas funcionales

En el punto anterior explicamos cómo comenzábamos a subir por el modelo en V ejecutando las pruebas preparadas. Por encima de las pruebas unitarias y de integración se encontraban las pruebas funcionales que habíamos preparado.

Para ejecutar estas pruebas decidimos en una primera instancia ejecutarlas de forma manual, para más adelante (en una segunda instancia) automatizar estas pruebas para no tener que ejecutarlas manualmente. La razón de esta decisión se debió a dos puntos principales:

- Codificar los tests para ser corridos automáticamente hubiera aumentado demasiado el tiempo de desarrollo del incremento de cada iteración, comparado al tiempo que llevaba correrlos manualmente en dichas iteraciones.
- El hecho que sean los test automáticos los que interactúan con el sistema nos hubiese quitado la posibilidad de que se nos ocurrieran nuevos casos de prueba. Por lo general, cuando es uno el que interactúa con el sistema, suele tener nuevas ideas o nuevos casos de cómo utilizar el sistema, lo cual derivan en nuevas pruebas o preguntas para el cliente del sistema.

De esta manera, el proceso de pruebas funcionales no requería gran cantidad de tiempo y al mismo tiempo nos permitía descubrir, de haberlos, nuevos escenarios o casos de uso del sistema.

Para ejecutar las pruebas de una funcionalidad, el primer paso que tomamos fue preseleccionar de la lista de casos de prueba del módulo en cuestión aquellas pruebas que aplicasen al estado actual de esa función. Podía suceder que, por ejemplo, del listado de casos de prueba para el módulo de tratamientos, una parte requiriera que otro módulo (como el módulo de órdenes médicas) estuviese desarrollado.

Una vez hecha la preselección, se ejecutaba cada caso de prueba de acuerdo a lo especificado. Una vez ejecutado el caso de prueba, comparábamos los resultados obtenidos con los resultados esperados, y cada discrepancia se registraba en un borrador para ser analizado en la etapa de *rework*. Si el problema era un problema menor y fácil de arreglar, se hacía el cambio en la etapa de *rework* correspondiente a la iteración en que nos encontrábamos. Por el contrario, si intuíamos que para corregir el problema necesitábamos invertir una cantidad mayor de esfuerzo, entonces cargábamos el problema en un sistema de ítems de forma de poder *trackearlo*, registrando cuál era el problema y los pasos para reproducirlo. El valor que usábamos para tomar esta decisión era muy relativo, pero

podríamos decir que si el esfuerzo para corregir el error se encontraba en el orden de los minutos, se arreglaba allí mismo; de lo contrario, si rondaba la hora o más, se cargaba el ítem correspondiente para su inspección en una iteración posterior.

3.6.3.2.5. Automatización de los casos de prueba

La única pregunta que nos quedaba por hacer es: ¿vale la pena invertir esfuerzo en automatizar las pruebas funcionales del sistema?

Llegamos a la conclusión que era necesario automatizar los test funcionales una vez que la iteración estuviese cerrada (todo excepto los tests automáticos) e incluso en paralelo al desarrollo de otras funcionalidades.

Consideramos que es interesante plantear por separado el análisis que hicimos para decidir automatizar pruebas funcionales y el proceso que seguimos para hacerlo. Ambas cuestiones se detallan en la siguiente sección.

3.6.4. Automatización de pruebas funcionales

3.6.4.1. ¿Por qué automatizamos las pruebas funcionales?

Como dijimos anteriormente, no solo nos planteamos el hecho de documentar las pruebas funcionales en una planilla, sino que decidimos que era imprescindible que las pruebas creadas fueran ejecutadas en forma automática.

Fewster y *Graham* [11] mencionan varias razones por las cuales es conveniente poseer pruebas automáticas en un proceso de desarrollo de software.

1. Correr tests en nuevas versiones del sistema.

   Esto es especialmente útil cuando el sistema sufre grandes cantidades de cambios, y con mucha frecuencia. “Dado que los tests ya existen y han sido automatizados para correr en una versión anterior de sistema, debería ser posible seleccionar e iniciar su ejecución con poco esfuerzo manual”. En nuestro caso, si bien los requerimientos funcionales estaban bastante bien definidos esperábamos que el sistema sufriera muchos cambios, dado que los mismos se iban dando de a incrementos pequeños. Desde este punto de vista, inclinarnos por automatizar las pruebas era una buena opción.

2. Se pueden correr los tests con mayor rapidez y con mayor frecuencia.

   Es una realidad que los test automáticos son mucho más rápidos que los test de ejecución manual, lo que permite ejecutar una mayor cantidad de tests y hacerlo con mayor frecuencia. En este sentido, el tener tests automatizados nos brindaba la

tranquilidad de que ante un cambio en una funcionalidad, podíamos correr los tests con facilidad y detectar si los cambios introducidos generaban problemas.

3. Ejecutar tests que serían difícil o imposible ejecutar manualmente.
   Un ejemplo de atributo que no es fácil verificar manualmente, es la interfaz gráfica de usuario. En nuestro sistema eran muy importantes las alertas de mensajes de error, por ejemplo. Automatizar los tests de UI nos permitía validar esto (entre otras cosas) con facilidad.

4. Mejor uso de los recursos.
   "El hecho de tener tests automáticos permite tener a testers disponibles para otras tareas". En nuestro caso, en que hacíamos de testers y desarrolladores al mismo tiempo, esta afirmación se volvía mucho más importante. Si correr un set de tests nos llevaba, por ejemplo, una hora, lo más probable es que sin tests automáticos no hubiésemos probado las funcionalidades con cada cambio para poder estar disponibles para otras tareas de desarrollo. Contando con tests automáticos, el tiempo que nos ahorramos al correr tests automáticos nos servía para dedicarnos a otras actividades.

5. Consistencia y repetición de tests.
   "Cuando se ejecutan en forma manual, los tests son propensos a errores y omisiones por parte del tester, en especial si el mismo se ve obligado a ejecutar el mismo set de tests muchas veces". Los tests automáticos nos asegurarían que todos los sets de corridas ejecutarían exactamente los mismos tests y de la misma manera.

6. Rápida salida al mercado.
   Como los test automáticos pueden correrse en forma más rápida, una vez escritos acortan la salida al mercado de cambios en el sistema o nuevas funcionalidades, dado que el tiempo de pruebas del sistema se reduce considerablemente. Este punto no era decisivo para nosotros, dado que en nuestro proceso la primera prueba que se ejecutaba sobre un módulo se hacía de forma manual. Sin embargo, era una ventaja para iteraciones futuras al módulo, ya que de tener que probar una funcionalidad ya existente al agregar un módulo relacionado a ella, nos permitiría probar que la vieja funcionalidad siguiera funcionando correctamente en menor tiempo.

7. Confianza.

"Si un set de tests automáticos pasa sin fallas, podemos confiar en que no habrá ninguna sorpresa desagradable cuando el sistema salga al mercado". Si bien este punto es muy discutible, es cierto que contar con tests automáticos reduce la posibilidad de que surjan problemas graves cuando el sistema está en producción.

Por supuesto, desarrollar tests automáticos tenía algunas desventajas también.

- Si las pruebas no iban a ser muy utilizadas, entonces el tiempo invertido en el desarrollo del test automático era tiempo perdido.
- Si cambiaba una funcionalidad, posiblemente sería necesario cambiar los tests automáticos que la probaban.

Dado que la mayoría de las ventajas se adaptaban a las características de nuestro proceso y proyecto, decidimos llevar a cabo la automatización de pruebas.

3.6.4.2. Introducción al proceso de automatización de pruebas funcionales

Lo primero que tuvimos que pensar luego de decidir llevar a cabo la automatización fue cuándo automatizar las pruebas. Notamos que automatizar las pruebas de un módulo dentro de la misma iteración en la que lo desarrollábamos hubiese demorado demasiado la iteración en base a lo originalmente planificado. De modo que para evitar esto, decidimos desarrollar las pruebas automatizadas para las funcionalidades de una determinada iteración en paralelo al desarrollo de la siguiente iteración. De esta manera, cuando un recurso se desocupaba antes en el desarrollo de una iteración, podía dedicarse a automatizar las pruebas concernientes a funcionalidades de la iteración (o iteraciones) anteriores.

Al igual que sucedió con las pruebas de integración y las pruebas de unidad, no todas las pruebas funcionales fueron automatizadas. Si bien la gran mayoría fue automatizada, muchas pruebas funcionales no pudieron automatizarse dada su complejidad o dado que el esfuerzo para automatizarlas era, a nuestro criterio, mucho mayor que ejecutar esas pruebas en forma manual. De esta forma, casi todas las pruebas funcionales acabaron por tener su test automatizado asociado, si bien alguna que otra prueba debía ejecutarse de forma manual.

3.6.4.2.1. Un proyecto de pruebas funcionales automatizadas

Aún no ahondaremos en los detalles de la implementación de un test automático. Pero si podemos, en un alto nivel, describir la estructura que tiene un proyecto de pruebas automatizadas y la razón por la cual decidimos implementar las pruebas como parte de un proyecto secundario.

Las razones por las cuales decidimos implementar las pruebas funcionales en un proyecto diferente se listan a continuación:

- La cantidad de clases necesarias para representar las pruebas funcionales de todo el sistema eran muchas y hubiesen sobrecargado demasiado el proyecto principal.
- El objetivo que perseguía el conjunto de clases y archivos de los tests funcionales era en cierto punto diferente al objetivo de los tests de integración y tests unitarios. Dicho de otra manera, dado que los test funcionales son pruebas de caja negra y representan casos de uso que en última instancia un usuario final del sistema podría ejecutar, decidimos que merecían estar separados del resto de las pruebas del proyecto.
- La administración y desarrollo de este tipo de pruebas se realizaría en paralelo al desarrollo del sistema en sí. Puesto de otra forma, dado que el avance en el desarrollo de los test automatizados iba a seguir un ritmo diferente al desarrollo del sistema, tener estas pruebas en un proyecto diferente nos permitía hacer esta separación y avanzar en forma diferente en cada proyecto.

La forma en que organizamos nuestro proyecto de pruebas es muy sencilla:

- Un paquete que contiene los WebDrivers. Los WebDrivers son programas que interactúan con el DOM de la página web, traduciendo las sentencias de programación contenidas en las clases de los tests para buscar elementos, escribir en campos de la página, obtener estilos CSS, clickear elementos, entre otras operaciones. En un proyecto grande, este paquete normalmente contendría una importante variedad de WebDrivers, ya que cada navegador particular tiene su propio driver de interacción. En nuestro caso, solo utilizamos ChromeDriver, dado que nuestras pruebas funcionales serán ejecutadas en ese navegador.
- Un paquete que contiene los archivos TestNG. Estos archivos representan un subset de tests funcionales de todo el sistema. En ellos se especifica qué test van a correrse, con sus respectivos parámetros. Por ejemplo, tenemos un archivo TestNG para las pruebas del módulo de Paciente, otro archivo para las pruebas del módulo de Tratamientos, etc.
- Un paquete que contiene las clases de los tests funcionales. Éstas son las clases que contiene el flujo lógico de las pruebas; es decir, los pasos en la ejecución de las pruebas, de la misma forma que un usuario las haría, pero traducidas a líneas de código. Por ejemplo, para un test de login, esta clase contendría cláusulas del tipo

"*completarNombreDeUsuario(nombreUsr)*", "*clickLoginButton()*" y "*validarUsuarioLoggeado()*".

- Un paquete que contiene las clases que brindan la base de la interacción con el DOM de las páginas web. En estas clases se codificarán las sentencias que permiten buscar un elemento en la página, las sentencias que permiten obtener atributos de estilo de los elementos, sentencias para escribir en elementos, para clickear elementos, entre otras.

De esta forma, sin darle demasiada complejidad, contamos con un proyecto bien organizado.

3.6.4.2.2. Actividades en el proceso de automatización

Hay tres aspectos que conciernen a un test automático, cada uno de ellos muy importante:

1. Desarrollo de tests automáticos: esta es la etapa que concierne al diseño y codificación de un test automático. En esta etapa decidimos el número de tests que habrá, como se agruparán en clases, la dependencia que existe entre ellos –---diseño de tests automatizados– y la lógica que ejecuta cada uno.
2. Ejecución de tests automáticos: una vez desarrollados los tests, necesitamos decidir cuándo o con qué frecuencia van a ejecutarse dichos tests. ¿Se ejecutarán con cada deploy? ¿Se ejecutarán en forma diaria? ¿Qué tests se correrán con cada deploy?¿Qué tests se correrán diariamente?¿Se ejecutarán automáticamente?
3. Reporte de ejecución de test automáticos: ésta es tal vez la parte más importante de trabajar con tests automáticos. Una vez que se ejecutan los tests, necesitamos saber cosas como ¿cuántos tests pasaron? ¿cuántos tests fallaron? ¿por qué fallaron los tests que fallaron?

3.6.4.3. Etapas del proceso de automatización de pruebas funcionales

3.6.4.3.1. Proceso de desarrollo de un test automatizado

El proceso seguido para el desarrollo de los tests automatizados puede describirse a grandes rasgos como sigue:

1. Definir una clase por cada módulo del sistema. Así, cada sublista del documento de tests funcionales del sistema estaría confinada a una clase del paquete de tests del proyecto de tests automatizados.

2. Revisar la lista de tests de cada módulo, señalando aquellos test que no serían automatizados. De esta forma podríamos identificar qué tests correr de forma manual al ejecutar los test funcionales de un módulo.
3. Definir para cada test una función dentro de la clase correspondiente. En este punto decidimos definir los nombres de las funciones utilizando la descripción de aquello que se estaba probando con el test, y no el identificador único que habíamos definido para el test en el documento de tests. Esta decisión se basó simplemente en el hecho que si los identificadores de los tests del documento de pruebas funcionales cambiaban, no sería necesario actualizar los nombres de las funciones en el código de los tests automatizados.
4. Definir para cada test los parámetros de entrada. En esencia, los parámetros para los tests funcionales se corresponden con las precondiciones de dicho test en el documento de pruebas funcionales. Si bien en este punto no preparamos los datos en sí (es decir, no cargamos los datos necesarios en el sistema en el entorno local de pruebas), nombramos los parámetros a ser utilizados en el test.
5. Definimos, de existir, la dependencia entre los tests funcionales. Dado que en la etapa de escritura de las pruebas funcionales nos tomamos el trabajo de que los tests fuesen autónomos entre sí, este tipo de dependencia casi no existió en los tests automatizados.
6. Definimos el setup del conjunto de tests. En el setup de las pruebas que definimos incluimos la configuración del tipo de WebDriver a utilizar, la carga de la url necesaria para los tests y, en los casos correspondientes, el login al sistema.
7. Definimos cualquier acción de limpieza posterior a los tests. En nuestro caso, no es necesaria ninguna limpieza.
8. El siguiente paso es, para cada test:
    a. Cargar en el entorno de pruebas casos positivos y casos negativos para los parámetros de prueba; es decir, casos que cumplan las precondiciones del test y casos que no las cumplan.
    b. Definimos, si no existe, una clase con el nombre *descripción_page*, que es una clase que guardará todos los elementos de interacción con el DOM.
    c. Codificar en la clase del test:
        i. Los pasos del flujo de ejecución del test funcional; esto lo hacemos haciendo de cuentas que los métodos de interacción con el DOM ya

están implementados (que puede o no existir, dependiendo de si los métodos fueron ya creados para otro test).

ii. Las validaciones a realizar; esto lo hacemos comparando con una cláusula Assert el retorno de los métodos de interacción con el DOM contra los valores de los elementos pasados como parámetros de la función que representa el test.

d. Codificar las clases y funciones de interacción con el DOM utilizadas en el test que no han sido creadas aún.

e. Ajustar parámetros de las llamadas y la definición de las funciones de interacción con el DOM, así como el tipo de retorno de las mismas con las variables utilizadas en el flujo de ejecución del test.

Siguiendo un proceso similar a ese desarrollamos los test automatizados de los diferentes módulos del sistema. Una vez codificadas las pruebas, antes que éstas puedan considerarse terminadas y confiables, realizamos dos validaciones:

- Validación de falla: para esta prueba utilizamos los casos negativos cargados en el punto 8.a del proceso antes descrito. Ante estos casos, se espera que los tests preparados fallen. En la ejecución de esta prueba arreglamos cualquier error de codificación que pueda existir en el test y además validamos que los tests fallen cuando deban fallar.
- Validación de caso exitoso: en esta prueba utilizamos los casos positivos cargados en el punto 8.a del proceso antes descrito. Se espera que con estos datos, los tests pasen.

Una vez que ambos tipos de prueba se hubieron corrido y resultaron según lo esperado, los tests estaban listos para correrse en forma automática. Se dejaron seteados como parámetros los correspondientes a un caso de éxito, de forma que si los tests fallaban sabríamos que es debido a un error introducido en el sistema.

Antes de pasar a la etapa de ejecución de los tests automáticos, queremos mencionar dos convenciones importantes que utilizamos en la codificación de las funciones de interacción con el DOM:

- Locators: en los tests automáticos se definieron atributos de clases que representan formas de encontrar elementos del DOM. Estos atributos se denominan *locators*. Para diferenciar de forma especial estos atributos del resto de los atributos de la clase, los nomenclamos como *elementoDom_locator*, donde "*elementoDom*" está compuesto como el tag del DOM que representa el elemento y una palabra representativa. Por

ejemplo: "*usernameInput_locator*" es un atributo que sirve para encontrar el Input del nombre de usuario en el DOM.

- PageObject: *pageObject* es un patrón de diseño de tests automatizados que consiste en confinar las funciones de interacción con el DOM en una clase de forma que esta lógica esté oculta al flujo de interacción del test. De esta forma, cualquier cambio en el DOM solo afectará a estas clases. A su vez, si una página se utiliza en muchos tests, la lógica para interactuar con ella está nucleada en una sola clase. Para implementar este patrón definimos una clase para cada página y popup complejo (por ejemplo, popups de carga de formularios), definiendo su nombre como *paginaDescription_page*. Por ejemplo: "*cargaTratamientoPage*" es la clase que nuclea todos los elementos del DOM del popup de carga de tratamiento y la interacción con ellos. A su vez, se definió una superclase *Page* que contiene lógica común a todas las páginas, como por ejemplo lógica de escritura en campos, click de botones, etc.

3.6.4.3.2. Reporte de ejecución de pruebas funcionales automatizadas

Por último, es de vital importancia tener un buen reporte de las corridas de los tests automatizados.

TestNG brinda las clases necesarias para que podamos, en caso que deseemos, crear nuestro propio reporte de corridas de tests. Existen herramientas que incluso permiten guardar la salida en archivos XML que luego se mapean a páginas HTML y brindan una interfaz web de corridas-resultados.

Sin embargo, dado el uso que iba a darse a los tests decidimos simplemente tomar la salida por defecto que genera TestNG por consola. Esta salida separa los resultados por Suite y por Tests (que a los propósitos del uso que nosotros le dimos, resultan ser lo mismo), indicando la cantidad de tests corridos y la cantidad de ellos que se saltearon y fallaron. Además, para los casos de prueba que fallan, TestNG indica (dependiendo el nivel de *verbose*) la causa de la falla. Todos estos datos pueden verse por consola pero, además, NetBeans muestra un pequeño gráfico representativo de cantidad de tests corridos y porcentaje de tests que pasaron y fallaron. A su vez, separa los resultados en diferentes categorías para un mejor análisis. Decidimos que esta salida del reporte de TestNG era suficiente para nuestro uso –que se restringía a correr los tests para descubrir fallas y arreglarlas (o reportarlas).

A continuación se muestra un ejemplo de la salida generada al correr los tests automáticos del módulo de paciente; la Suite de tests sólo contiene los tests automáticos de la clase PacienteTest en este caso:

```
PASSED: test_breadCrumb
[...]
PASSED: test_busquedaSinResultado("Ana Lia")
PASSED: test_busquedaPacienteCarenteDeAlgunosDatos("Pedroso, Gabriela")
PASSED: test_camposObligatoriosAlCrearPaciente
[...]
PASSED:       test_eliminacionExitosaPaciente("Pacientest;PTUno;11222333;Ricardo       Aldao
1152;;34245111222;25/05/1995;JERARQUICOS SALUD;14258796")
FAILED:     test_creacionCanceladaDePaciente("Pacientest;PTUno;11222333;Pedro     de     Vega
2314;4608833;34245789245;25/05/1995;OSDE;5468974626")
java.lang.AssertionError: expected [null] but found [[Ljava.lang.String;@f14a7d4]
 at org.testng.Assert.fail(Assert.java:94)
 at org.testng.Assert.failNotSame(Assert.java:492)
 at org.testng.Assert.assertNull(Assert.java:428)
 at org.testng.Assert.assertNull(Assert.java:417)
 at systemTests.tests.PacienteTest.test_creacionCanceladaDePaciente(PacienteTest.java:246)
[...]

FAILED:      test_edicionPacienteCancelada("Pacientest;PTUno;11222333;Pedro      de      Vega
2314;4608833;34245789245;25/05/1995;OSDE;5468974626",     "Pacientest;PTUno;11222333;Ricardo
Aldao 1152;;34245111222;25/05/1995;JERARQUICOS SALUD;14258796")
java.lang.AssertionError: expected [Pedro de Vega 2314] but found [Ricardo Aldao 1152]
 at org.testng.Assert.fail(Assert.java:94)
 at org.testng.Assert.failNotEquals(Assert.java:496)
 at org.testng.Assert.assertEquals(Assert.java:125)
 at org.testng.Assert.assertEquals(Assert.java:178)
 at org.testng.Assert.assertEquals(Assert.java:188)
 at systemTests.tests.PacienteTest.test_edicionPacienteCancelada(PacienteTest.java:397)
[...]

===============================================
    Usuario Test
    Tests run: 14, Failures: 2, Skips: 0
===============================================

===============================================
SeleniumTestProject
Total tests run: 14, Failures: 2, Skips: 0
===============================================

[TestNG] Time taken by org.testng.reporters.EmailableReporter2@7a46a697: 31 ms
[TestNG] Time taken by org.testng.reporters.XMLReporter@1c6b6478: 30 ms
[TestNG] Time taken by org.testng.reporters.jq.Main@299a06ac: 164 ms
[TestNG] Time taken by org.testng.reporters.JUnitReportReporter@3abfe836: 41 ms
[TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@1f89ab83: 114 ms
[TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 10 ms
```

Ilustración 1.34: Ejemplo de salida del test automático correspondiente a la clase PacienteTest.

3.7. Instalación y despliegue

La instalación y despliegue se ubica, en el marco de la metodología, dentro de la Fase de Transición. En la sección Planificación se explica en qué consiste y lo que nos llevó a transcurrir dicha fase.

Los sistemas web deben ser desplegados en equipos con buena conectividad (una conexión hogareña no cumple en general los requisitos), con un determinado nivel de seguridad y disponibilidad, y con ciertas características de hardware y software. Lo más costoso son las dos primeras características: la conexión a internet y asegurar cierto nivel de seguridad y disponibilidad, ya que respecto al hardware los requerimientos son bajos para la concurrencia esperada, y respecto al software, se utilizan herramientas gratuitas que corren sobre sistemas operativos gratuitos. Los servicios de cloud computing apuntan a resolver las dos primeras cuestiones, es por esto que elegimos desplegar el sistema en alguno de ellos.

La tecnología empleada, tanto el motor de base de datos como el servidor de aplicaciones, son tecnologías multiplataforma. Esto nos permitió hacer el desarrollo sobre el sistema operativo Windows, e instalarlo posteriormente en un sistema operativo basado en GNU/Linux.

Desplegar el sistema en un servicio de cloud computing implica el uso de herramientas para la gestión remota, saber usar la consola de administración del servicio de cloud, y, cómo se desplegó en un entorno GNU/Linux, manipular la tecnología (servidor de aplicaciones y motor de base de datos) mediante línea de comandos.

Se desplegó la aplicación en un único equipo de Amazon AWS sobre Red Hat Enterprise Linux 7 (ofrecido por AWS a partir de la unión hecha entre el proveedor de cloud computing y Red Hat), y se configuró lo necesario para gestionar el equipo y transferir archivos.

Se instaló el software necesario, los más importantes son Java, el servidor de aplicaciones Payara y el motor PostgreSQL, y se configuraron estos dos últimos para que funcionen como servicios del sistema operativo.
Luego se migró toda la configuración del servidor de aplicaciones desde el entorno de desarrollo hacia el equipo remoto, utilizando el mecanismo de backup y recuperación.

Se creó el esquema de la base de datos, se la conectó con el servidor de aplicaciones y por último se instaló la aplicación.

Luego de hacer cambios en el sistema, para ponerlos en producción, se sube al servidor el archivo .war producto de la compilación y se lo re-despliega en el servidor de aplicaciones.

3.8. Gestión del proyecto

Esta sección describe las variables que tuvimos en cuenta durante el desarrollo del proyecto y el impacto que tuvieron sobre la planificación del mismo.

Kent Beck [10], en su libro Extreme Programming Explained, explica que "existen cuatro variables en el desarrollo de software que pueden controlarse: costo, tiempo, calidad y alcance. La forma de manejar estas variables es hacer que las fuerzas externas (clientes, gerentes) elijan el valor de tres de ellas. Luego el equipo de desarrollo elije el valor de la cuarta. Algunos gerentes o clientes creen que pueden elegir el valor de las cuatro variables."

Con respecto a esta última oración, el autor afirma que cuando se intenta fijar el valor de las cuatro variables, el equipo de desarrollo termina comprometiéndose en tiempo, alcance (requerimientos funcionales) y costo (recursos), y lo primero que se termina desatendiendo es la calidad, "dado que nadie hace un buen trabajo bajo mucho estrés".

Según el autor, las cuatro variables están relacionadas entre sí:

"Tiempo: Prolongar el tiempo de la entrega puede mejorar la calidad e incrementar el alcance. Pero demasiado tiempo puede perjudicar al sistema ya que el feedback más valioso proviene del sistema en producción. Si le damos a un proyecto muy poco tiempo la calidad se verá perjudicada.

Alcance: Disminuir el alcance hace posible entregar mejor calidad. También permite entregar más temprano o a menor costo.

Costo: Es la variable más restringida.

Calidad: Es terrible como variable de control. Se pueden tener cortas ganancias de tiempo sacrificando deliberadamente la calidad, pero el costo humano, técnico y de negocio es enorme [...]. La calidad externa es la calidad vista por el cliente. La calidad interna es la calidad percibida por los desarrolladores. Sacrificar la calidad interna temporalmente para reducir el *time to market* esperando que la calidad externa no sufra demasiado puede ser

tentador para el corto plazo. Eventualmente, pueden ganarse semanas o meses. Luego, sin embargo, los problemas de calidad interna tienen sus consecuencias haciendo su software prohibitivamente caro de mantener, o incapaz de alcanzar un nivel competitivo de calidad externa."

Situándonos en nuestro caso, el costo es una variable que ni el cliente ni el equipo decidió gestionar, considerando el marco y particularidades del desarrollo del proyecto.

Sin embargo, durante la gestión de este proyecto decidimos prestar mayor atención a una variable adicional, la calidad del proyecto final de carrera. Esta variable la relacionamos con el alcance y el tiempo. Lo que decidimos hacer fue incrementar el alcance y mejorar la calidad externa. Esto indefectiblemente se tradujo en un mayor tiempo, teniendo en cuenta limitaciones en la disponibilidad de los recursos. Las siguientes subsecciones detallan los cambios hechos en la planificación y cómo fueron aconteciendo los hechos.

3.8.1. Planificación

En esta sección pretendemos repasar la planificación inicial y describir aquellos aspectos que impactaron en la misma. La tabla siguiente lista las fases, iteraciones y tareas que se planificaron en un principio.

Fase	Iteración	Objetivos
Inicio	I1	• Identificación de las funciones principales del sistema. • Esbozo de la arquitectura del sistema. • Planificación y estimación inicial del proyecto. • Identificación de los principales riesgos y su gestión. • Selección e inducción base en las tecnologías para realizar el trabajo.
Elaboración	E1	• Elicitación y refinamiento de los requerimientos asociados con la gestión de los pacientes y los tratamientos. Validación de requerimientos funcionales. Definición de Casos de Prueba. • Definición y especificación de la arquitectura

		candidata. ● Realización y documentación del diseño detallado. ● Identificación de nuevos riesgos y gestión de los existentes. ● Prueba de las tecnologías seleccionadas para realizar el trabajo.
	E2	● Elicitación y refinamiento de los requerimientos acerca de la gestión de sesiones y movimientos de caja. Validación de requerimientos funcionales. Definición de casos de prueba. ● Ampliación y refinamiento del diseño detallado. ● Integración de los nuevos modelos con los realizados en la iteración E1. ● Identificación de nuevos riesgos y gestión de los existentes. ● Comprensión del flujo de información a través de la arquitectura tecnológica.
Construcción	C1	● Codificación del módulo de pacientes, tratamientos y diagnósticos. ● Configuración y codificación de los aspectos de seguridad. ● Diseño y ejecución de pruebas de unidad y de sistema de dichos módulos. ● Configuración del flujo de información a través de la arquitectura tecnológica.
	C2	● Incorporación de la gestión de órdenes médicas al módulo de pacientes. ● Construcción de la base del módulo de sesiones y agenda, codificando las funcionalidades más sencillas. ● Diseño y ejecución de pruebas de unidad y de

		sistema de dicho módulo. ● Integración de la gestión de sesiones y la de órdenes con el módulo de tratamientos.
	C3	● Finalización del módulo de sesiones y agenda, incorporando las funcionalidades más complejas. ● Diseño y ejecución de pruebas de unidad y de sistema de dicho módulo. ● Integración de los nuevos componentes con los ya codificados.
	C4	● Codificación del módulo de reporte de órdenes médicas. ● Codificación del módulo de configuraciones del sistema. ● Diseño y ejecución de pruebas de unidad y de sistema de dichos módulos. ● Integración de los nuevos componentes con los ya codificados.

Tabla 1.4: En un principio planificamos un único ciclo de desarrollo, con una iteración de inicio, 2 de elaboración y 4 de construcción. A la fase de transición inicialmente no la pensamos como parte del proyecto.

Consideramos el no cumplimiento estricto de la planificación inicial (ver Anexo A) como un proceso normal y esperable de la ejecución del proyecto, la naturaleza del mismo y adecuado al proceso de aprendizaje que atravesamos. En cuestión, si bien se trabajó con el objetivo de cumplir al máximo la planificación, resulta inevitable el cambio en los requerimientos, en la prioridad de los mismos o en la precisión de las estimaciones. Estos hechos nos llevaron a:

- Replanificar las iteraciones 3 y 4 de construcción.
- Realizar la fase de transición.
- Incorporar un nuevo ciclo de desarrollo dentro del marco de la metodología.

Las siguientes secciones explican cada uno de estos puntos.

3.8.1.1. Replanificación

Durante el desarrollo de la tercera iteración de construcción surgieron tareas no planificadas que eran necesarias desde la calidad interna y externa del sistema. **Invertimos tiempo en hacer un diseño responsive, darle al sistema un aspecto gráfico agradable y favorecer su usabilidad**. Todas estas tareas no fueron dimensionadas correctamente al momento de planificar, incrementando la duración de las tareas ya planificadas y originando otras nuevas. Este tipo de contratiempos fue contemplado en el Plan de Riesgos; (ver Anexo B) por riesgo ID R1, cuya descripción es la siguiente:

Surgen actividades imprevistas y necesarias para la ejecución adecuada del proyecto y las mismas conllevan un tiempo no despreciable. La probabilidad de ocurrencia está amplificada por la falta de conocimiento de las tecnologías empleadas

Como acción correctiva se propuso:

En la iteración en curso se realizan las tareas necesarias (se reconsidera la prioridad de las tareas) manteniendo la duración planificada de la misma. Las nuevas tareas o tareas postergadas se planifican para la iteración siguiente, replanificando la misma en caso de ser necesario.

Estas tareas no planificadas nos llevaron a aplicar la acción correctiva prevista y replanificar las funcionalidades pensadas para las iteraciones 3 y 4 de construcción.

3.8.1.2. Fase de Transición

En un determinado momento, el calendario nos indicaba que habíamos alcanzado el fin del plan. Por otro lado el cliente necesitaba una primera versión y sólo faltaban funcionalidades que no eran indispensables; además sabíamos que teníamos una versión totalmente funcional y que era sumamente importante obtener feedback de producción lo antes posible. Estos factores nos llevaron a iniciar lo que el Proceso Unificado llama la Fase de Transición.

En el marco del Proceso Unificado, se describe a la Fase de Transición como aquella en la que el equipo “se centra en implantar el producto en su entorno de operación. La forma en que el proyecto lleva a cabo este objetivo varía con la naturaleza de la relación del producto con su mercado. Si un producto va a salir al mercado para muchos clientes, el equipo de proyecto distribuye una versión beta [...]. Si un producto va a distribuirse a un

único cliente [...], el equipo instala el producto en un solo sitio para llevar a cabo las pruebas de aceptación."

Al momento de iniciar esta fase, "el jefe de proyecto ha considerado que el sistema ofrece la confianza suficiente como para operar en el entorno del usuario, aunque no sea necesariamente perfecto. Por ejemplo, algunos problemas, riesgos y defectos que no se han puesto en evidencia durante las pruebas del sistema al final de la fase de construcción, pueden ponerse de manifiesto en el entorno del usuario."

El Proceso Unificado detalla "lo que se hace en la fase de transición:

- Preparar la versión de pruebas de aceptación a partir de la versión con capacidad operativa inicial producida durante la fase de construcción.
- Instalar (o preparar la instalación de) esta versión en los lugares elegidos, junto con las actividades relacionadas, como la migración de datos desde el sistema anterior.
- Actuar a partir de la información recogida en las instalaciones de pruebas.
- Adaptar el producto corregido a las circunstancias de los usuarios.
- Determinar cuándo se acaba el proyecto."

"La fase de transición no acaba cuando se completan todas las tareas y artefactos, sino cuando el cliente queda "satisfecho" (...) El jefe de proyecto concluirá que el cliente está satisfecho una vez que el sistema pase las pruebas de aceptación. Por supuesto, este punto depende de la interpretación de los requisitos detallados originalmente (...)."

3.8.1.3. Nuevo ciclo

Además de los defectos y errores, durante la fase de transición "el usuario puede descubrir con retraso la necesidad de determinadas características. Si son muy importantes y casan bien con el producto existente, el jefe de proyecto puede aceptar añadirlas. Sin embargo, los cambios deben ser lo suficientemente pequeños como para que puedan ser introducidos sin afectar seriamente el plan de proyecto. Si una característica propuesta afecta a la planificación, su necesidad debe ser aguda. En la mayoría de los casos, consideramos que es mejor añadirla a la lista de características y dejarla para el siguiente ciclo de desarrollo, es decir, para el desarrollo de la siguiente versión del sistema" [1].

Esto que nos sugiere el Proceso Unificado, sumado a los cambios de planificación, nos llevó a plantear un nuevo ciclo de desarrollo.

Repasando la estructura de la metodología, el Proceso Unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo concluye con una versión del producto para los clientes y consta de las cuatro fases: inicio, elaboración, construcción y transición. De esta forma, el hecho de haber realizado la fase de transición implicó finalizar el primer ciclo de la vida del sistema. Como sugiere la bibliografía, las características postergadas y los nuevos requerimientos solicitados fueron considerados para ser realizados en un nuevo ciclo.

3.8.2. Cronología del desarrollo

3.8.2.1. Primer ciclo

3.8.2.1.1. Fase de Elaboración

El análisis de requerimientos hecho en la primera iteración comenzó a partir de un documento escrito en conjunto con el cliente producto de las reuniones y entrevistas hechas durante la fase de inicio. Dicho documento describe aquellas actividades que se realizan en el consultorio y que se querían informatizar. Al comienzo de cada iteración de elaboración se hicieron mockups o bocetos de las pantallas, que originaron dudas y cuestiones por resolver. Se entrevistó al cliente hacia fines de cada iteración de elaboración. Además de las entrevistas, la comunicación con el cliente siempre estuvo; a través de mensajes, chat o email, obtuvimos respuestas en tiempo y forma.

Especificar los casos de uso nos llevó a refinar aún más los requerimientos, al mismo tiempo que hacíamos supuestos que luego debíamos validar.
En un principio no estaban claras todas las entidades involucradas en el dominio, ni tampoco sus relaciones. Debimos descubrir y modelar en esta fase las clases de análisis principales, para luego refinarlas en las tareas de diseño.

Como parte de la especificación de requerimientos se documentaron los atributos de calidad o requerimientos no funcionales en el documento de arquitectura. También se definieron los atributos de calidad internos y se los documentó de igual forma.

Durante el flujo de trabajo de diseño se hizo el diseño detallado de las clases, los mockups sirvieron de base para el diseño de las pantallas y se determinaron los módulos necesarios.

También se realizaron ensayos sobre la arquitectura a implementar y preparamos el entorno de desarrollo, acorde a las tareas planificadas con este fin.

3.8.2.1.2. Fase de Construcción

De **enero a marzo 2016** se realizó la primera iteración. Las tareas realizadas fueron las detalladas en el plan. La única disonancia entre lo realizado y lo planificado fueron los tiempos que se extendieron en tres semanas aproximadamente. Durante el turno de examen de febrero esperábamos (y así lo planificamos) dedicarnos sólo un poco menos a las tareas del proyecto. Lo que sucedió en la práctica es que el avance en el desarrollo durante los exámenes fue nulo, conllevando al atraso mencionado anteriormente en los tiempos.

En la segunda iteración el desfase temporal se mantuvo, además de verse influenciado nuevamente por el turno de examen de mayo. Además, durante la iteración se produjo una actualización de dos de las tecnologías con las que estábamos trabajando (PrimeFaces y Glassfish). La migración a las nuevas versiones llevó un tiempo que no habíamos contemplado en el plan. También en esta iteración ya teníamos varios componentes interactuando entre sí, lo que nos llevó a implementar las pruebas de integración y las pruebas funcionales. La organización y configuración de las pruebas, así como también la configuración del entorno de testing, incorporó al proyecto otros tiempos no planificados. Estos factores retrasaron el inicio de la tercera iteración en un mes. El tiempo total dedicado a esta iteración abarcó desde **marzo a mayo de 2016**.

En la tercera iteración, comenzamos desarrollando las funcionalidades del los CU 12 y 13, tal como indica el plan. El resto de las tareas planificadas de esta iteración fueron postergadas para la cuarta iteración de Construcción del primer ciclo (las más prioritarias) o para al segundo ciclo del Proceso Unificado. Esto se realizó acorde al plan de riesgos (ver sección 3.8.1.1 Replanificación). En su lugar se realizaron tareas destinadas a satisfacer atributos de calidad del sistema, también detalladas en la sección 3.8.1.1 Replanificación. Además, en esta iteración se comenzó con la automatización de test funcionales, específicamente aquellos referentes al login en el sistema, módulo de Pacientes y módulo de Tratamientos. La iteración 3 finalizó aproximadamente a mediados de **agosto**.

En la cuarta iteración de construcción se desarrollaron algunas funcionalidades que habían quedado pendientes de la iteración anterior (como son la carga masiva de sesiones y la carga rápida de pacientes desde la agenda) y las tareas planificadas para la cuarta iteración (como son el consentimiento informado y el módulo de caja). Teniendo este paquete de funcionalidades decidimos desplegar el sistema en un entorno similar al de producción para probar su funcionamiento y prepararnos para la fase de transición. Las tareas de desarrollo y

despliegue en testing se realizaron desde mediados de agosto hasta mediados de **septiembre**, momento en el cual se decidió comenzar la fase de transición. Ver sección 3.8.1.2 Fase de Transición.

3.8.2.1.3. Fase de Transición

Durante esta fase se cargaron en la base de datos los datos que se usarían en el entorno de producción, sobre todo las obras sociales y los tipos de tratamientos y se configuraron los usuarios que tendrán acceso al sistema.

En el transcurso de la fase los usuarios reportaron errores y cambios menores: nuevos filtros para la lista de órdenes, posibilidad de seleccionar la hora para cada día en la carga masiva de sesiones, formato del reporte de órdenes, obligatoriedad de los datos del paciente, etc. Llegó un punto en que mermaron las solicitudes, a mediados de **octubre**, cuando comenzamos a planificar el siguiente ciclo.

3.8.2.2. Segundo ciclo

El segundo ciclo abarcó el desarrollo de las tareas postergadas del primer ciclo, asociadas a los casos de uso que se listan a continuación:

- CU18: Registrar ingreso y egreso de efectivo
- CU19: Visualizar planilla de ingresos y egresos de efectivo
- CU21: Adjuntar informe de estudio escaneado a un tratamiento.

Además, se desarrollaron las siguientes nuevas funcionalidades requeridas:

- Generar y mostrar estadísticas a partir de los datos de los pacientes.
- Notificar al usuario respecto a la gestión de órdenes médicas.
- Sincronizar los pacientes con una cuenta de correo
- Enviar recordatorios a los pacientes sobre su turno

A continuación se presenta la planificación realizada.

Fase	Iteración	Tareas
Elaboración	E1	Especificar los nuevos CU requeridos. Rever los CU ya especificados. Actualizar documentación de requerimientos. Determinar las nuevas entidades y/o cambios del diseño de datos. Determinar los impactos en la arquitectura de los nuevos CU. Actualizar la documentación de diseño. Elaborar casos de prueba necesarios.

Construcción	C1	Codificar los casos de uso: ● CU18: Registrar ingreso y egreso de efectivo. ● CU21: Adjuntar informe de estudio escaneado a un tratamiento. ● CU24: Sincronizar los pacientes con una cuenta de email.
	C2	Codificar los casos de uso: ● CU26: Visualizar estadísticas del consultorio. ● CU22: Generar notificaciones ● CU23: Visualizar y administrar notificaciones ● CU25: Enviar recordatorios a los pacientes sobre su turno.
Transición	T1	Preparar la versión a ser instalada. Instalar las nuevas funcionalidades en producción. Resolver los problemas y cambios menores que surjan.

Tabla 1.5: Como segundo ciclo de desarrollo se propuso uno mas corto que el primero, con una iteración de elaboración, dos de construcción y una de transición.

Para este nuevo ciclo contábamos con la experiencia adquirida luego de planificar el primero de los ciclos. Estimamos una duración de 1 mes entre las fases de Elaboración y Construcción, sin contar la fase de Inicio que nos llevó la primera semana de noviembre. La fase de Transición es muy difícil de planificar, ya que pueden salir a la luz errores, o plantearse cambios, que a su vez hay que decidir si incorporarlos en la corriente versión o plantear una nueva.

3.8.2.2.1. Fase de Elaboración

En este segundo ciclo, la relación con el cliente estaba más afianzada y mucho más fluida. Esto nos llevó a conseguir una comunicación estrecha e informal. De las diversas reuniones o charlas surgieron propuestas, las cuales se hicieron formales en una entrevista con el cliente realizada luego del primer ciclo.

Esta fase comenzó a realizarse el **7 de noviembre** y se finalizó aproximadamente el día **20 de noviembre**. Durante la misma, con los nuevos requerimientos obtenidos de la entrevista, se procedió a actualizar documentación de requerimientos.

Además, se documentaron los nuevos casos de uso y se revisaron los anteriormente especificados para ver si las nuevas funcionalidades tenían impacto en los mismos. Al tratarse de características nuevas y desacopladas de las desarrolladas en el anterior ciclo, no hubo

mayores influencias de los nuevos casos de uso sobre los del ciclo pasado. En cuanto al diseño de datos sucedió una situación similar, solo se agregaron las nuevas entidades relacionadas a las nuevas funcionalidades.

También se realizaron ensayos sobre la arquitectura a implementar para desarrollar el *CU21: Adjuntar informe de estudio escaneado a un tratamiento* y el *CU24: Sincronizar los pacientes con una cuenta de email*.

3.8.2.2.2. Fase de Construcción

Esta fase fue encarada en dos iteraciones. La primera iteración comenzó inmediatamente después de la fase de Elaboración y tuvo una duración aproximada de una semana y media. Durante la misma se llevaron a cabo los casos de uso número 18, 21 y 24.

Un punto importante a destacar es que, al ser estas 3 funcionalidades completamente desacopladas entre sí, se pudo hacer una división de trabajo de las mismas y asignar cada una a un recurso del equipo de trabajo. De esta manera se fueron desarrollando en forma paralela, lo que aceleró mucho los tiempos.

Luego de la primera iteración se desarrollaron los casos de uso 22, 23, 25 y 26, con la misma metodología de trabajo de la iteración anterior. La segunda también tuvo una duración aproximada de una semana y media, siendo concluida el **8 de diciembre**.

3.8.2.2.3. Fase de Transición

La fase de Transición del segundo ciclo comenzó alrededor del **10 de diciembre**. Durante esta Transición hubo que resolver algunos problemas. Era necesario incorporar nuevos mecanismos para gestionar la configuración del sistema al momento de cambiar de entornos: entorno de desarrollo, testing y producción. Ciertos parámetros ahora dependían del entorno, como por ejemplo el directorio donde se guardan las imágenes en el sistema de archivos. Esto surgió como una necesidad al momento de preparar las versiones a instalar en producción.

También en esta fase aparecieron errores que escaparon a las pruebas y sugerencias de cambios menores.

3.8.3. Evaluación de la metodología

La sección 3.8.1 Planificación desarrolla la experiencia que tuvimos con respecto al uso de la metodología elegida y cómo usamos ésta para afrontar los hechos de la realidad. Consideramos que para adaptarnos mejor a los cambios, en lugar de haber planteado un único ciclo que abarque todas las funcionalidades, lo ideal hubiese sido plantear desde un principio

dos ciclos más cortos, por ende iteraciones más cortas, lo que da lugar a instalar versiones en intervalos menores.

4. Capítulo 4: Conclusiones

Contribuciones, ventajas y aportes de la solución

Como resultado del proyecto se obtuvo una solución informática que cubre las necesidades y problemas del cliente, quién percibe una serie de aportes y mejoras en su actividad fruto de la puesta en producción del sistema.

Alineado a la naturaleza de los proyectos de desarrollo de software, podemos reconocer que los requerimientos funcionales y no funcionales fueron cambiando en el transcurso del presente proyecto; algunos se quitaron, otros se redefinieron y surgieron nuevos. El balance final nos muestra un incremento en el alcance del proyecto y del producto, tanto en calidad como en características aportadas.

A través de las distintas funcionalidades implementadas, es preciso mencionar los siguientes aportes y beneficios que este proyecto consiguió aportar al consultorio de kinesiología:

- Proveer un sistema que centralice la gestión de información.
- Facilitar y optimizar la ejecución de las tareas diarias.
- Brindar más seguridad a la información almacenada y transmitida.
- Permitir procesar la información para obtener informes y estadísticas.
- Contribuir al incremento en el nivel de servicio prestado al paciente.
- Lograr un mayor control sobre la asistencia de los pacientes.
- Reducir tiempos de proceso en tareas de gestión habituales.
- Brindar la capacidad para gestionar aspectos relacionados al consultorio en cualquier momento y lugar.

Experiencias y dificultades de la utilización de los métodos y herramientas

Como se ha explicado antes en el desarrollo del presente informe, en el proyecto se presentaron distintos desafíos (cambios en los requerimientos, surgimiento de nuevas tareas, resolución de problemas diversos, contratiempos, etc). Esto nos condujo a tener que reorganizar tareas e incurrir en actividades del ciclo de vida del software que no habían sido planificadas. El *cómo hacerlo* se relaciona sin duda con la metodología empleada: el Proceso Unificado de desarrollo de software. La bibliografía sobre el Proceso Unificado arrojó luz

sobre estos temas, al mismo tiempo que nos permitió comprender y aprender con mayor profundidad sus conceptos e ideas.

Respecto a los resultados obtenidos de la ejecución del proyecto, podemos mencionar que logramos el objetivo propuesto inicialmente de desarrollar el sistema, pero además, dimos un paso más allá realizando la puesta en producción y comienzo del mantenimiento post-productivo, que se encuentra en vigencia al momento de desarrollar este informe final.

Si bien la planificación inicial no contemplaba la puesta en producción del sistema, el mismo avance del proyecto y cambio de prioridades en el cliente, nos condujo a la necesidad de instalar el sistema en producción y planificar un segundo ciclo de desarrollo. Esto nos demuestra que por mucho que nos esforcemos en controlar diversos factores para elaborar una planificación completa desde un principio, la inexperiencia en el área de la planificación y la dinámica del proyecto y del negocio, conducen a que su cumplimiento en tiempo y forma sea un aspecto muy difícil de lograr.

Finalmente, otro tema que destaca es la adquisición de experiencia y conocimientos técnicos durante la realización del proyecto. Una amplia variedad de proyectos en el mercado trabajan hoy en día con tecnologías web, y la realización del presente sistema nos ha brindado la posibilidad de dar los primeros pasos en el uso de tecnologías afines.

Posibles trabajos futuros

La solución de software resultante del trabajo en este proyecto puede analizarse también respecto a los aspectos técnico/funcionales bajo los cuales fue desarrollada y a la flexibilidad para adaptarse a otros clientes de la misma industria. Podemos decir que si bien fue desarrollado como un sistema a medida, las tareas a las que da soporte forman parte de procesos conocidos y empleados por cualquier profesional del dominio. Es importante mencionar que puede ser considerado un sistema de interés para cualquier kinesiólogo de la ciudad de Santa Fe. En este sentido, como posible trabajo futuro, se puede analizar la incorporación de algunas extensiones (especialmente en una plataforma mobile) y una reestructuración de determinados componentes que permitan lograr una solución más genérica, cuyo objetivo final sea la definición de un producto de software que atienda la mayor parte de los requerimientos de los profesionales de este ámbito en la ciudad de Santa Fe, para los cuales se reconoce una demanda a atender.

Referencias bibliográficas

[1] Jacobson et al.: El Proceso Unificado de Desarrollo de Software (1999)

[2] Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice (1997)

[3] Philippe Kruchten: Architectural Blueprints—The "4+1" View Model of Software Architecture. Rational Software Corp, IEEE (1995)

[4] Microsoft patterns & practices: Model-View-Controller. http://msdn.microsoft.com/en-us/library/ff649643.aspx

[5] Wikipedia: Modelo de vistas de arquitectura. https://es.wikipedia.org/wiki/Modelo_de_Vistas_de_Arquitectura_4%2B1

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

[7] Jim Arlow y Ila Neustadt: UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design. 2nd Edition. Pearson (2005)

[8] Kendall & Kendall. Análisis y Diseño de Sistemas. 6ta Edición. Pearson (2005)

[9] Roger S. Pressman: Ingeniería del software: Un enfoque práctico. 7ma Edición. McGraw-Hill Education (2009)

[10] Kent Beck and Cynthia Andres: Extreme Programming Explained: Embrace Change. 2nd Edition. Addison-Wesley (2004)

[11] Mark Fewster, Dorothy Graham: Software Test Automation. Addison-Wesley (1999)

[12] Glenford J. Myers: The art of software testing. 2nd Edition. Wiley (2004)