Universitat oberta de Catalunya (UOC)
Gener 2013

Ciclo de validación de una aplicación informática

Memoria del proyecto
de Ing. Técnica Teleco. Esp. Telemática
Autor: Raúl Villegas Beltrán
Tutor: Ricard Burriel

ÍNDICE

Introducción

La obtención de un *software* con calidad implica la utilización de metodologías o procedimientos estándares para el análisis, diseño , programación y prueba del *software* que permitan uniformizar la filosofía de trabajo, con el fin de lograr una mayor confiabilidad, "mantenibilidad" y facilidad de prueba, a la vez que eleven la productividad, tanto para la labor de desarrollo como para el control de la calidad del *software*.

El proyecto se desarrolla en colaboración con la empresa *Pharma S.L.*, empresa que se dedica, entre otras tareas, al desarrollo de software hospitalario, desde el *software* utilizado para gestionar las peticiones e históricos de los pacientes, hasta los drivers necesarios para las conexiones con los aparatos utilizados.

1.1 ¿Porque es necesario el testing?

Los sistemas de *software* están creciendo como parte de nuestra vida diaria. Existen aplicaciones para todo tipo de negocios.

El testing es necesario porque si el desarrollador introduce un error, esto hace que se introduzca un defecto en el programa y este defecto causa un fallo en el *software*.

Mucha gente ha tenido experiencias con *software* que no trabajan de la manera esperada. El *software* que no funciona correctamente puede producir muchos problemas, incluyendo la pérdida de dinero, tiempo o reputación del negocio, e incluso puede ser la causa de daño o muerte.

1.2 ¿Qué es el testing?

Las definiciones del testing son variadas y están cambiando con el tiempo. De todas maneras hay definiciones que cubren todos los aspectos de qué es el testing.

Algunas definiciones incluyen:

> ".. ejecutar un programa con la intención de certificar su calidad".
>
> [1] *Mills*, 1970
>
> "..ejecutar un programa o sistema con la intención de encontrar errores"
>
> [1] *Myers*, 1979
>
> "..establecer confianza que un programa o sistema hace lo que se supone que debe hacer"
>
> [1] *Hetzel*, 1973

Una de las más completas probablemente es:

"el proceso de ejercitar el *software* o un sistema para detectar errores, para verificar que cumple funcional y no funcionalmente los requerimientos y para explorar y entender el estado de los beneficios y riesgos asociados con su liberación." [2]

Una percepción común del testing es que sólo consiste en ejecutar los *tests*, en otras palabras ejecutar el *software*. Este es una parte del testing, pero no lo es todo.

Algunas actividades de prueba tienen lugar antes y después de la ejecución de los *tests* como:

- Planificación
- Control
- Elección de las condiciones del test
- Diseño de los casos de prueba
- Comprobación de los resultados
- Evaluación del criterio de finalización
- Presentación de informes en el proceso de pruebas del sistema bajo *test*
- Finalización o clausura (por ejemplo después de una fase de *test* que ha sido completada)

El testing también incluye la revisión de documentos (incluyendo el código fuente) y otro análisis estático.

El testing dinámico y estático se puede como un mismo significado para archivar objetivos similares y de esta manera proporcionarán información para ayudar a mejorar el sistema para ser testeado y desarrollado y para realizar el proceso del testing.

Los diferentes objetivos del testing pueden ser definidos como:

- Búsqueda de defectos
- Prevención de defectos
- Conseguir confianza sobre el nivel de calidad
- Proporcionar información sobre el sistema

El proceso de razonamiento del diseño de los tests al inicio del ciclo de vida (verificando el *test* basado en el diseño de *test*) puede ayudar a prevenir defectos desde el inicio introduciéndolos en el código. La revisión de los documentos (por ejemplo los requerimientos) también ayuda a prevenir defectos que pueden aparecer en el código.

Hay diferentes puntos de vista en el testing que pueden estar basados en diferentes objetivos. Por ejemplo, en el desarrollo del testing (como un componente, integración y sistema de testing) el objetivo principal será encontrar la causa de la gran mayoría de fallos como sea posible, de tal manera que los

defectos en el *software* son identificados y pueden ser arreglados. Durante la aceptación del testing el objetivo principal debe ser confirmar que el sistema trabaja como se espera y cumple los requisitos definidos en los requerimientos.

El mantenimiento del testing a veces incluye un testing para comprobar que no hay nuevos errores introducidos durante el desarrollo e implementación de la corrección de los errores y de los cambios.

Hay que diferenciar dos términos muy importantes, debugar y testear, no son la misma cosa:

- El testing puede mostrar fallos que son causados por los defectos
- Debugar es una actividad de desarrollo que identifica la causa del defecto, repara el código y verifica que el defecto ha sido corregido correctamente.

1.3 Metodologías de testing

Tradicionalmente el testing de software se ha dividido en dos estrategias básicas que se supone son de aplicación universal.

- **Testing estructural o de caja blanca.** Testear un *software* siguiendo esta estrategia implica que se tiene en cuenta la estructura del código fuente del programa para seleccionar casos de prueba, es decir, el testing está guiado fundamentalmente por la existencia de sentencias tipo *if*, *case*, *while*, etc. En muchas ocasiones se pone tanto énfasis en la estructura del código que se ignora la especificación del programa, convirtiendo el testing en una tarea un tanto desprolija e inconsistente.

  Como los casos de prueba se calculan de acuerdo a la estructura del código, no es posible generarlos sino hasta que el programa ha sido terminado. Peor aun, si debido a errores o cambios en las estructuras de datos o algoritmos, aun sin que haya cambiado la especificación, puede ser necesario volver a calcular todos los casos.

  Sin embargo es interesante remarcar la racionalidad detrás de esta estrategia: no se puede encontrar un error si no se ejecuta la línea de código donde se encuentra ese error, aunque ejecutar una línea de código con algunos casos de prueba no garantiza encontrar un posible error.

  Por consiguiente es importante seleccionar casos de prueba que ejecuten al menos una vez todas las líneas del programa, lo cual se logra analizando la estructura del código.

  Se dice que el testing estructural prueba lo que el programa hace y no lo que se supone que debe hacer.

- **Testing basado en modelos o de caja negra**. Testear una pieza de *software* como una caja negra significa ejecutar el *software* sin considerar ningún detalle sobre cómo fue implementado. Esta estrategia se basa en

seleccionar los casos de prueba analizando la especificación o modelo del programa, en lugar de su implementación.

Algunos autores consideran que el testing basado en modelos (MBT) es la automatización del testing de caja negra. Para lograr esta automatización el MBT requiere que los modelos sean formales dado que ésta es la única forma que permite realizar múltiples análisis mecánicos sobre el texto de los modelos. Por el contrario, el testing de caja negra tradicional calcula los casos de prueba partiendo del documento de requerimientos.

Como en el MBT los casos de prueba se calculan partiendo del modelo, es posible comenzar a testear casi desde el comienzo del proyecto, al menos mucho antes de que se haya terminado de programar la primera unidad. Por otra parte, pero por la misma razón, los casos de prueba calculados con técnicas de MBT son mucho más resistentes a los cambios en la implementación que aquellos calculados con técnicas de testing estructural. Más aun, el MBT es, efectivamente, automatizable en gran medida como demostraremos más adelante al utilizar una herramienta para tal fin. La gran desventaja del MBT radica en su misma definición: se requiere de un modelo formal del *software*, cosa que solo un porcentaje ínfimo de la industria es capaz de realizar.

Se dice que el testing basado en modelos, prueba lo que el programa se supone que debe hacer, y no lo que el programa hace.

Es muy importante remarcar que estas estrategias no son opuestas sino complementarias.

1.4 Objetivos del proyecto

El proyecto se realiza conjuntamente con la empresa *Pharma S.L*. En el proyecto participan un total de 3 personas validando, 1 lider de grupo (Team Leader) y 2 ingenieros de validación (Validation Engineer).

El software a validar facilita la gestión de pacientes ambulatorios en tratamiento anticoagulante oral (TAO) ofreciendo soporte informático a las tareas específicas del personal de los centros a los que acuden los pacientes.
El TAONet se utiliza en la prevención y tratamiento de las enfermedades tromboembólicas.

Los objetivos del proyectos es explicar desde un entorno profesional y real, el ciclo de validación de un producto informático. La idea principal es validar una versión integra de un producto que ya esta en el mercado. Los objetivos principales serían:

- Creación de Test Case (TC's o casos de pruebas) a través de las nuevas implementaciones propuestas por el equipo de desarrollo.
- Creación y actualización de TC's con los errores a solucionar en esta versión.
- Creación de Test plan con las pruebas seleccionadas.
- Creación de Test report con el resultado de las pruebas ejecutadas.
- Instalación en diferentes escenarios de la aplicación. Se tendrán en cuentan

diferentes Sistemas operativos (Windows 7, W2003, Linux…), diferentes Bases de Datos como MySQL y Oracle y por último diferentes navegadores como Internet Explorer (en versión 7 y 8), Mozilla Firefox i Chrome.
- Registro de las incidencias surgidas a raíz de la validación.
- Generar toda la documentación necesaria tras la validación de la aplicación (métricas, informe de validación…).
- Si hubiese la posibilidad, incluso existe la opción de una validación externa en un cliente.

La validación de este *software* debe ser muy minuciosa y estricta ya que es un *software* orientado a sanidad, y no se pueden permitir fallos graves. Aplicando metodologías descritas, como la caja negra, se conseguirá un *software* fiable y robusto que cumpla los requerimientos definidos desde un inicio para cada módulo definido.

Durante el periodo de verificación se revisarán tanto funcionalidades del sistema como introducción de resultados, validación de las peticiones o impresión de los informes, pero también será necesario revisar las partes más técnicas como la instalación en diferentes entornos de trabajo, revisando diferentes navegadores y sistemas operativos.

1.5 Contenido de la memoria

La información de la memoria se divide de la siguiente manera:

- En el capítulo 2 se hace una presentación de los conceptos de validación, verificación y testing. Después de ello se explican detalladamente los ciclos de vida del *software,* argumentando que el ciclo de vida en V es el utilizado.
- En el capítulo 3 se detallan los objetivos del proyecto definiendo el escenario, el perfil del usuario, recursos, riesgos y organización. Se fija el modelo de ciclo de vida a seguir y la planificación del proyecto.
- En el capítulo 4 se encuentra el desarrollo del proyecto, presentando los antecedentes del *software* a tratar, las funcionalidades del programa y que tiempo de estrategia se va a seguir en la validación. Se detallan los recursos utilizados para realizar la validación y de que manera se aplican en nuestro caso. Finalmente se desarrolla la validación de programa.
- En el capítulo 5 se hace una reflexión de los objetivos conseguidos o no presentando mejoras para poder agilizar el proceso de validación. Para finalizar se adjunta una valoración personal sobre el desarrollo del proyecto.
- En la carpeta adjunta a la memoria se incluyen un conjunto de documentos a los que se hace referencia en el texto y con los que se ha trabajado durante el desarrollo del proyecto. Estos documentos son:
    - *Test Plan* del build 1, 2 y 3.
    - *Test Report* del build 1, 2 y 3.

- *Excel Resumen estado errores build 1 y 2.*
- SW Build Notes 1, 2 y 3.
- Un error (PI) introducidos en *Clear Quest*.
- Una petición de mejora (PCA) introducida en *Clear Quest.*
- Cinco *Test Case* ejecutados durante la validación de versión 3.1.
- Informe de *Validación externa*.
- Excel *Requerimientos funcionales* en la versión 3.1.
- Excel *Matriz de trazabilidad* en la versión 3.1

2- MÉTODOS DE VALIDACIÓN

2.1 V, V &T: Validación, Verificación y testing

A continuación se detallan términos claves y necesarios para poder entender qué significado tiene hacer una validación, verificación y testing del *software*.

2.1.1- Verificación

Confirmación para examinar y proveer sobre un objetivo evidente qué requerimientos específicos están bien realizados. [1]

La verificación sobre cualquier sistema informático es una tarea para determinar que el sistema está construido de acuerdo con sus especificaciones. La verificación pregunta la cuestión "¿está el sistema construido correctamente?"

Las cuestiones planteadas durante la verificación incluyen:

- ¿El diseño refleja los requerimientos?
- ¿Están todos los temas incluidos en los requerimientos definidos en el diseño?
- ¿El diseño detallado refleja los objetivos del diseño?
- ¿El código refleja exactamente el diseño detallado?
- ¿Está el código correcto respecto con la sintaxis del lenguaje?

Cuando el programa se ha verificado, se asegura (en la medida de lo posible) que no hay "*bugs*" o errores técnicos.

2.1.2- Validación

Confirmación mediante exanimaciones y presentación de evidencias objetivas de que los requerimientos particulares de un uso que se hayan especificado se han cumplido. [1]

La validación es un proceso de determinación de que el sistema actual cumplirá el propósito para el que se destinó. De acuerdo con lo descrito, se asegura que el software refleja correctamente lo que el usuario necesita.

La validación da respuesta a las preguntas

- ¿Es un sistema correcto?
- ¿El conocimiento es correcto?

  ó

- ¿El programa hace el trabajo que se intentó hacer?

Así, la validación es la determinación de que el sistema completo cumple las funciones de los requerimientos especificados y que sea utilizable para los fines propuestos.

El alcance de las especificaciones es raramente preciso, y prácticamente imposible para un sistema de *test* por debajo de todos los eventos excepcionales posibles.

Aun así, es imposible tener absoluta garantía que un programa satisface las especificaciones, solo se puede obtener un grado de confianza para que un programa sea válido.

2.1.3- Testing

Proceso de ejecución del *software* que verifica los requerimientos especificados y la detección de errores.[6]

El testing es una actividad desarrollada para evaluar la calidad del producto, y para mejorarlo al identificar defectos y problemas. El testing de *software* consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados, a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento esperado.

Es una técnica dinámica en el sentido de que el programa se verifica poniéndolo en ejecución de la forma más parecida posible a como se ejecutará cuando esté en producción. Esto se contrapone a las técnicas estáticas las cuales se basan en analizar el código fuente. El programa se prueba ejecutando sólo unos pocos casos de prueba dado que por lo general es física, económica o técnicamente imposible ejecutarlo para todos los valores de entrada posibles. Si uno de los casos de prueba detecta un error el programa es incorrecto, pero si ninguno de los casos de prueba seleccionados encuentra un error no podemos decir que el programa es correcto (perfecto). Esos casos de prueba son elegidos siguiendo alguna regla o criterio de selección. Se determina si un caso de prueba ha detectado un error o no comparando la salida producida con la salida esperada para la entrada correspondiente. La salida esperada deberá estar documentada en la especificación del programa.

Las limitaciones antes mencionadas no impiden que el testing se base en técnicas consistentes, sistemáticas y rigurosas (e incluso, como veremos más adelante, formales). Sin embargo, en la práctica industrial, como ocurre con otras áreas de Ingeniería de *Software*, usualmente se considera sólo una parte mínima de dichas técnicas tornando a una actividad razonablemente eficaz y eficiente en algo útil y de escaso impacto.

Aunque en la industria de *software* el testing es la técnica predominante, en aquellos casos minoritarios en que se realiza una actividad seria de V&V (validación&verificación), en el ambiente académico se estudian muchas otras técnicas. Veamos algunos conceptos básicos de testing. Testear un programa significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing se estructura en casos de prueba o casos de test; los casos de prueba se reúnen en conjuntos de prueba. Desde el punto de vista del testing se ve a un programa (o subrutina) como una función que va del producto cartesiano de sus entradas al producto cartesiano de sus salidas. Es decir:

$$P : ID \rightarrow OD$$

donde ID se llama dominio de entrada del programa y OD es el dominio de salida. Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyos campos identifican a cada una de las variables de entrada o salida del programa, es decir:

$$ID = [x1 : X1, ... , xn : Xn]$$

$$OD = [y1 : Y1, ... ,ym : Ym]$$

De esta forma, un caso de prueba es un elemento, x, del dominio de entrada (es decir x 2 ID) y testear P con x es simplemente calcular P(x). En el mismo sentido, un conjunto de prueba, por ejemplo T, es un conjunto de casos de prueba definido por extensión y testear P con T es calcular P(x) para cada x 2 T. Es muy importante notar que x es un valor constante, no una variable; es decir, si por ejemplo ID = [num : N] entonces un caso de prueba es 5, otro puede ser 1000, etc., pero no n > 24.

También es importante remarcar que x1, ... , xn son las entradas con que se programó el programa (esto incluye archivos, parámetros recibidos, datos leídos desde el entorno, etc.) y no entradas abstractas o generales que no están representadas explícitamente en el código. De la misma forma, y1, ... , ym son las salidas explícitas o implícitas del programa (esto incluye salidas por cualquier dispositivo, parámetro o valor de retorno, e incluso errores tales como no terminación, *Segmentation fault*, etc.).

De acuerdo a la definición clásica de corrección, un programa es correcto si verifica su especificación. Entonces, al considerarse como técnica de verificación el testing, un programa es correcto si ninguno de los casos de prueba seleccionados detecta un error. Precisamente, la presencia de un error o defecto se demuestra cuando P(x) no satisface la especificación para algún x en ID. Un fallo es el síntoma manifiesto de la presencia de un error. Es decir que un error permanecerá oculto hasta que ocurra un fallo causada por aquel. Por ejemplo, si la condición de una sentencia if es x > 0 cuando deberá haber sido x > 1, entonces hay un error en el programa, que se manifestará (falla) cuando se testee el programa con x igual a 1 -si por fortuna este caso de prueba es seleccionado- porque en tal caso aparecerá cierto mensaje en la pantalla que para ese valor de x no deberá haber aparecido. En este sentido el testing trata de incrementar la probabilidad de que los errores en un programa causen fallos, al seleccionar casos de prueba apropiados. Finalmente, una falta es un estado intermedio incorrecto al cual se llega durante la ejecución de un programa. Por ejemplo, se invoca a una subrutina cuando se deberá haber invocado a otra, o se asigna un valor a una variable cuando se deberá haber asignado otro.

2.2 Modelos del ciclo de vida

Las principales diferencias entre distintos modelos de ciclo de vida están dividas en tres grandes versiones [3]:

- **El alcance del ciclo de vida,** que depende de hasta dónde deseamos llegar con el proyecto: sólo si es viable el desarrollo de un producto, el desarrollo completo o el desarrollo completo más actualizaciones y el mantenimiento.
- **La calidad y cantidad de las etapas** en las que dividiremos el ciclo de vida; según el ciclo de vida que adoptamos, y el proyecto para el cual lo adaptamos.
- **La estructura y la sucesión de las etapas,** si hay realimentación entre ellas, y si tenemos libertad de repetirlas (iterar).

En los distintos modelos de ciclo de vida mencionaré el riesgo que suponemos aceptar al elegirlo. Cuando hablamos de riesgo, nos referimos a la probabilidad que tenemos de volver a retomar una de las etapas anteriores, perdiendo tiempo, dinero y esfuerzo.

2.2.1- Ciclo de vida lineal

Es el más sencillo de todos los modelos. Consiste (Figura 2.1) en descomponer la actividad global del proyecto en etapas separadas que son realizadas de manera lineal, es decir, cada etapa se realiza de una sola vez, a continuación de la etapa anterior y antes de la etapa siguiente. Con un ciclo de vida lineal es muy fácil dividir las tareas y prever los tiempos (sumando linealmente los de cada etapa).

Las actividades de cada una de las etapas mencionadas deben ser independientes entre sí, es decidir, que es condición primordial que no haya retroalimentación entre ellas, aunque sí pueden admitirse criterios supuestos de realimentación correctiva. Desde el punto de vista de la gestión, requiere también que se conozca desde el primer momento, con excesiva rigidez, lo que va a ocurrir en cada una de las distintas etapas antes de comenzarla. Este último minimiza, también, las posibilidades de errores durante la codificación y reduce al mínimo la necesidad de requerir información del cliente o del usuario.

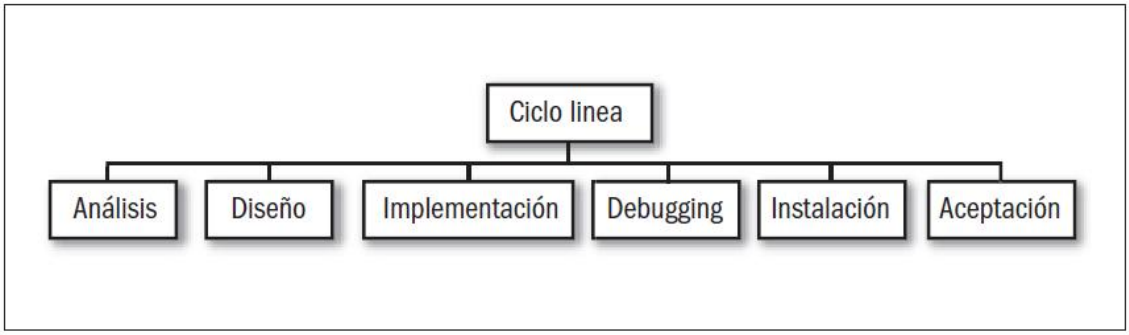


Figura 2.1: Ciclo de vida lineal

Se destaca como ventaja la sencillez de su gestión y administración tanto económica como temporal, ya que se acomoda perfectamente a proyectos internos de una empresa para programas muy pequeños de ABM (sistemas que realizan Altas, Bajas y Modificaciones sobre un conjunto de datos). Tiene como desventaja que no es apto para Desarrollos que superen mínimamente requerimientos de retroalimentación en etapas, es decir, es muy costoso retomar una etapa anterior al detectar algún error.

Es válido tomar este ciclo de vida cuando algún sector pequeño de una empresa necesita llevar un registro de datos acumulativos, sin necesidad de realizar procesos sobre ellos más que una consulta simple. Es decir, una aplicación que se dedique exclusivamente a almacenar datos, sea una base de datos a un archivo plano. Debido a que la realización de las etapas es muy simple y el código muy sencillo.

2.2.2- Ciclo de vida en cascada puro

Este modelo de ciclo de vida fue propuesto por *Winston Royce* en el año 1970. Es un ciclo de vida que admite iteración, contradictoriamente a la creencia de que es un ciclo de vida secuencial como el lineal (Figura 2.2). Después de realizar cada etapa se realiza una o varias revisiones para comprobar si se puede pasar a la siguiente. Es un modelo rígido, poco flexible, y con muchas restricciones. Aunque fue uno de los primeros y sirvió de base para el resto de modelos de ciclo de vida.

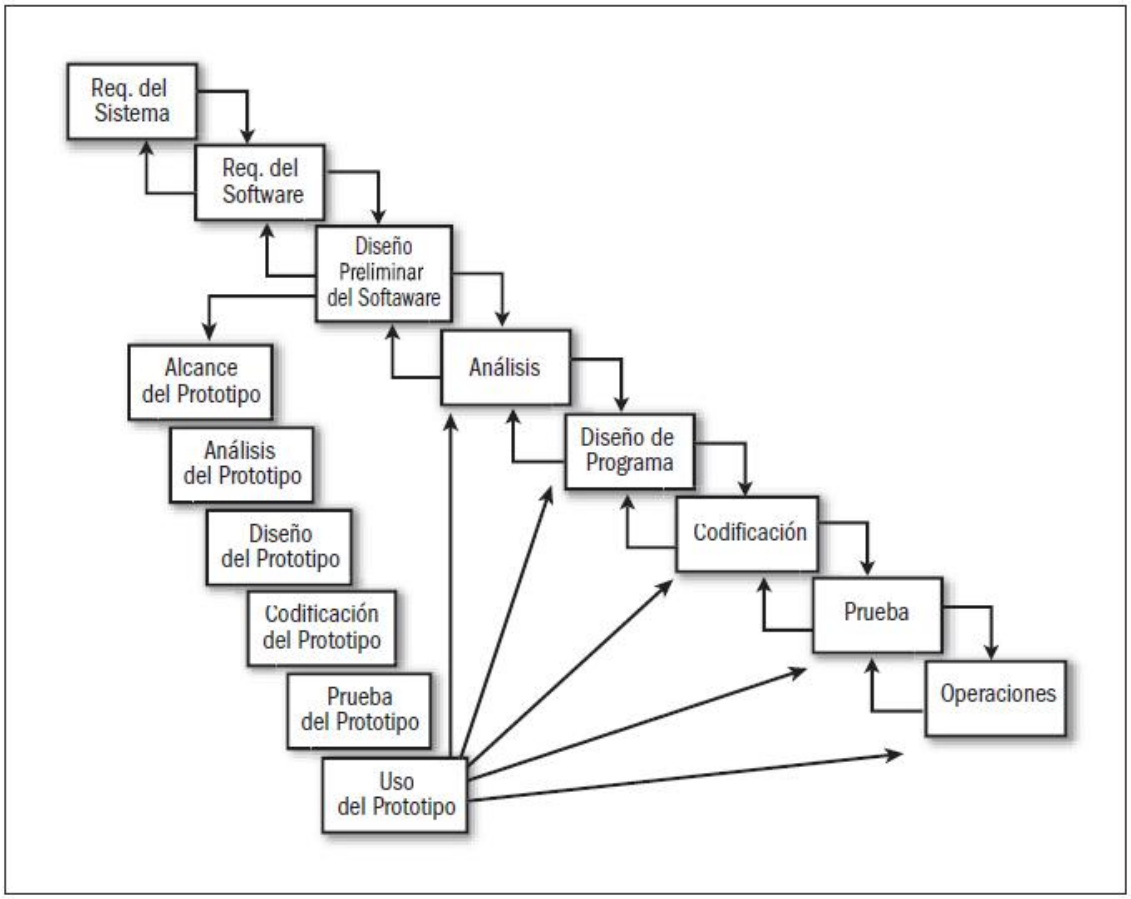


Figura 2.2: Ciclo de vida en cascada puro

Una de sus ventajas, además de su planificación sencilla, es la de proveer un producto con un levado grado de calidad sin necesidad de un personal altamente cualificado. Se pueden considerar como inconvenientes: la necesidad de contar con todos los requerimientos (o la mayoría) al comienzo del proyecto y si se han cometido errores y no se detectan en la etapa inmediata siguiente, es costoso y difícil volver atrás para realizar la corrección posterior.

Además, los resultados no los veremos hasta que no estemos en las etapas finales del ciclo, por lo que, cualquier error detectado nos trae un retraso y aumentará el costo del desarrollo en función del tiempo que insume la corrección de éstos.

Es un ciclo adecuado para los proyectos en los que se dispone de todos los requerimientos al comienzo, para el desarrollo de un producto con funcionalidades

conocidas o para proyectos, que aun siendo complejos, se entienden perfectamente desde el principio.

Se evidencia que es un modelo puramente teórico, ya que el usuario rara vez mantiene los requerimientos iniciales y existen muchas posibilidades de que debamos retomar alguna etapa anterior.

2.2.3- Ciclo de vida en V

Este ciclo fue diseñado por *Alan Davis* (Figura 2.3) y contiene las mismas etapas que el ciclo de vida en cascada puro. A diferencia de aquél, a éste se lo agregaron dos subetapas de retroalimentación entre las etapas de análisis y mantenimiento y entre las de diseño y *debugging*.

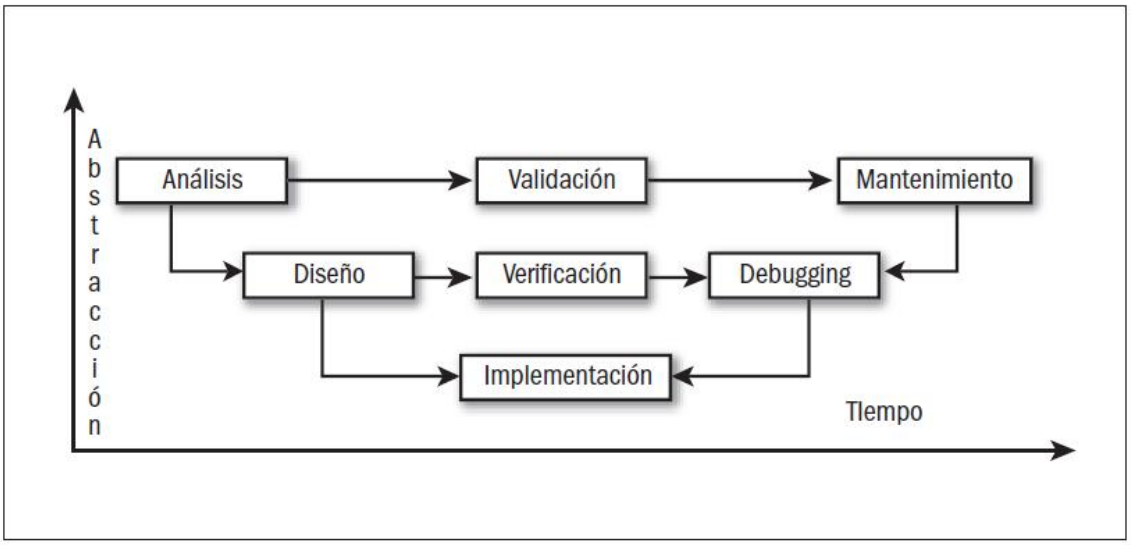


Figura 2.3: Ciclo vida en V

Las ventajas y las desventajas de este modelo son las mismas del ciclo anterior, con el agregado de los controles cruzados entre las etapas para lograr mayor corrección.

Este modelo de ciclo de vida refleja la validación & verificación durante el proceso de desarrollo del *software*.

2.2.4- Ciclo de vida tipo *Sashimi*

Este ciclo de vida es parecido al ciclo de vida en cascada puro, con la diferencia de que en el ciclo de vida en cascada no se pueden solapar las etapas, y en este sí. Esto suele aumentar su eficiencia ya que la retroalimentación entre etapas se encuentra implícitamente en el modelo (Figura 2.4).

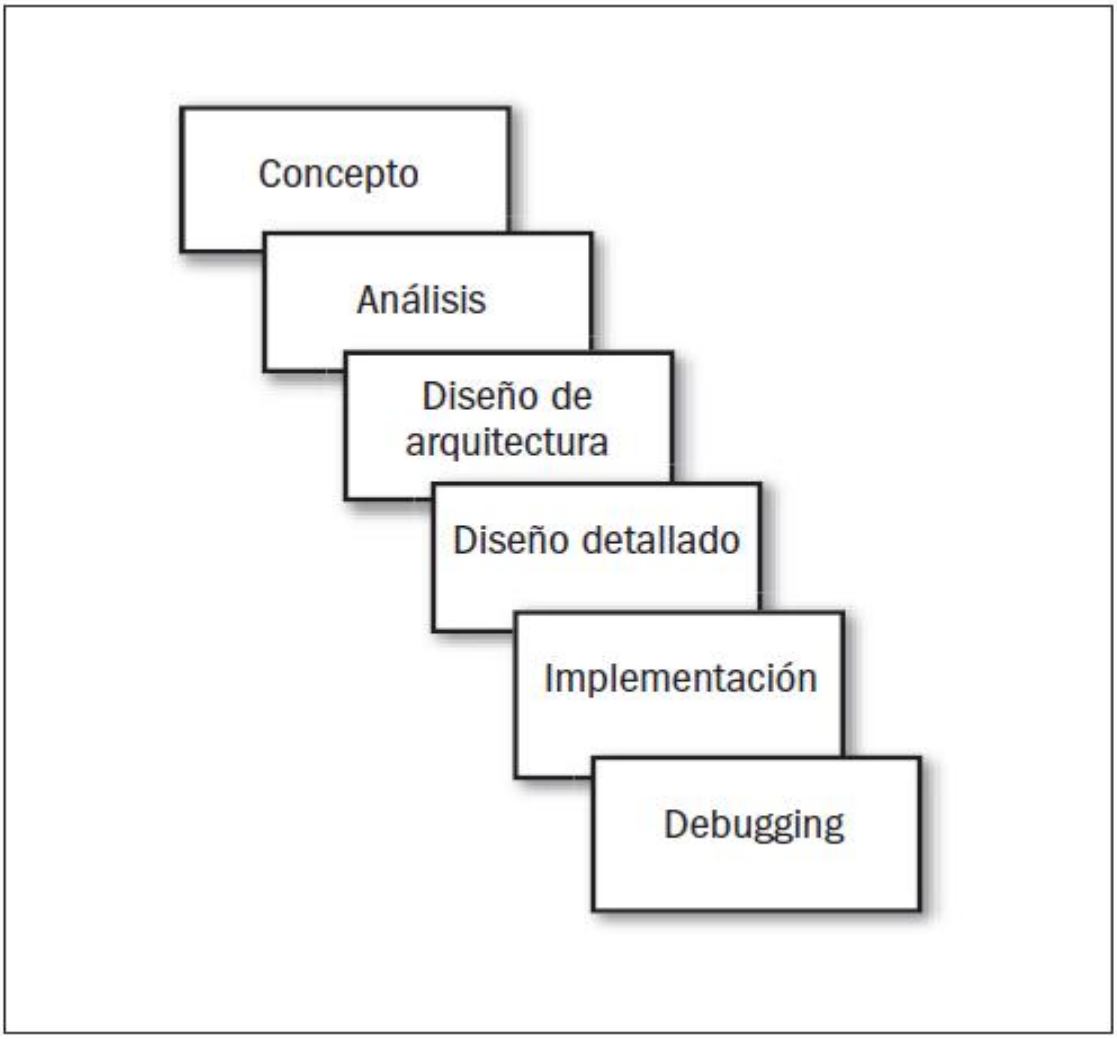


Figura 2.4: Ciclo vida tipo Sashimi

Se hace notar como ventajas la ganancia de calidad en la que respeta al producto final, la falta de necesidad de una documentación detallada (el ahorro proviene por el solapado de etapas). Sus ventajas también se refieren al solapamiento de las etapas: es muy difícil gestionar el comienzo y fin de cada etapa y los problemas de comunicación, si aparecen, generan inconsistencias en el proyecto.

Cuando necesitemos realizar una aplicación que compartirá los recursos (CPU, memoria o espacio de almacenamiento) con otras aplicaciones en un ambiente productivo, este modelo de ciclo de vida es una opción muy válida. El solapamiento de una de sus etapas nos permite en la práctica jugar un poco con el modelo de tres capas ahorrando recursos.

2.2.5- Ciclo de vida en casada con subproyecto

Sigue el modelo de ciclo de vida en cascada como se ve representado en la figura 2.5. Cada una de las cascadas se dividen en subetapas independientes que se pueden desarrollar en paralelo.

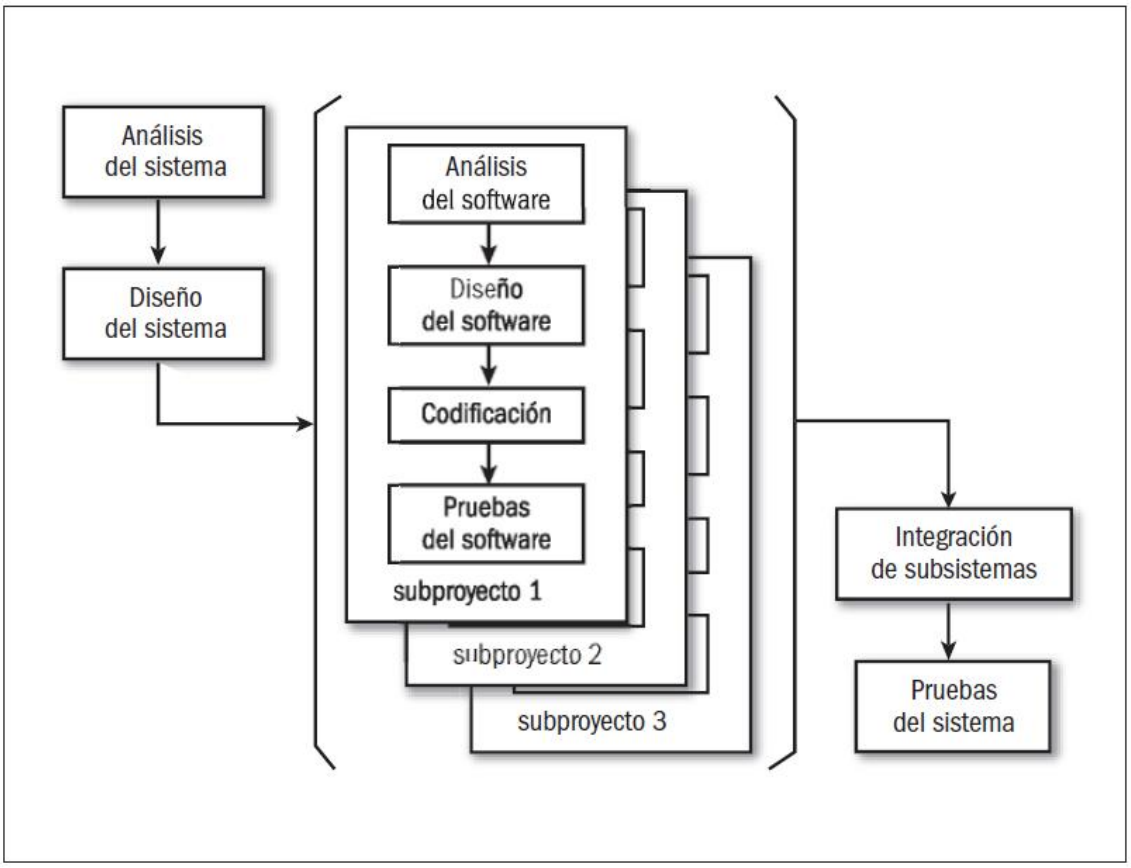


Figura 2.5: Ciclo de vida en cascada con subproyectos

La ventaja es que se puede tener más gente trabajando al mismo tiempo, pero la desventaja es que pueden surgir dependencias entre las distintas subetapas que detengan el proyecto temporalmente si no es gestionado de manea correcta.

Podemos utilizar este modelo para administrar cualquier proyecto mencionado en los modelos anteriores. Pero cuidando de administrar muy bien los tiempos.

2.2.6- Ciclo de vida interactivo

También derivado del ciclo de vida en cascada puro, este modelo busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malos entendidos durante la etapa de solicitud de requerimientos (Figura 2.6).

Es la interacción de varios ciclos de vida en cascada. Al final de cada interacción se le entrega al cliente una versión mejorada o con mayores funcionalidades del proyecto. El cliente es quien luego de cada iteración, evalúa el producto y lo corrige o propone mejoras. Estas interacciones se repetirán hasta obtener un producto que satisfaga al cliente.

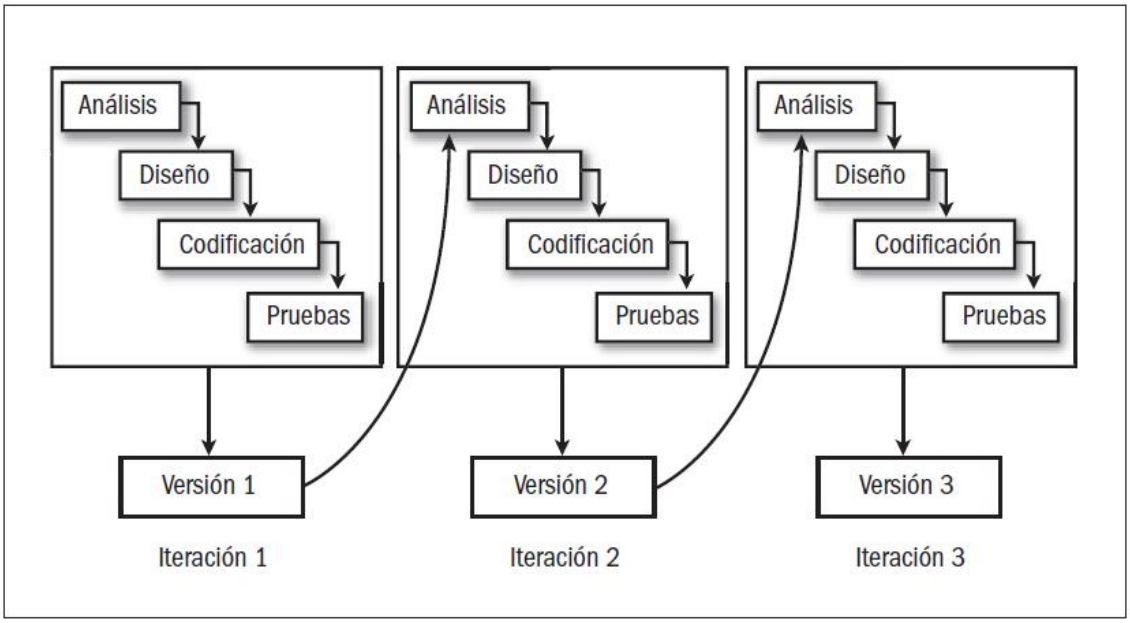


Figura 2.6: Ciclo vida iterativo

Se suele utilizar en proyectos en los que los requerimientos no están claros de parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente.

Podemos adoptar el modelo mencionado en aplicaciones medianas a grandes, en las que el usuario o el cliente final no necesitan todas las funcionalidades desde el principio del proyecto. Quizás una empresa que debe migrar sus aplicaciones hacia otra arquitectura y desea hacerlo paulatinamente, es un candidato ideal para este tipo de modelo de ciclo de vida.

2.2.7- Ciclo de vida por prototipos

El uso de programas prototipo no es exclusivo del ciclo de vida iterativo. En la práctica los prototipos se utilizan para validar los requerimientos de los usuarios en cualquier ciclo de vida (Figura 2.7).

Si no conoce exactamente como desarrollar un determinado producto o cuáles son las especificaciones de forma precisa, suele recurrirse a definir especificaciones iniciales para hacer un prototipo, o sea, un producto parcial y provisional. En este modelo, el objetivo es lograr un producto intermedio, antes de realizar el producto final, para conocer mediante el prototipo cómo responderán las funcionalidades previstas para el producto final.

Antes de adoptar este modelo de ciclo debemos evaluar si el esfuerzo por crear un prototipo vale realmente la pena adoptarlo.

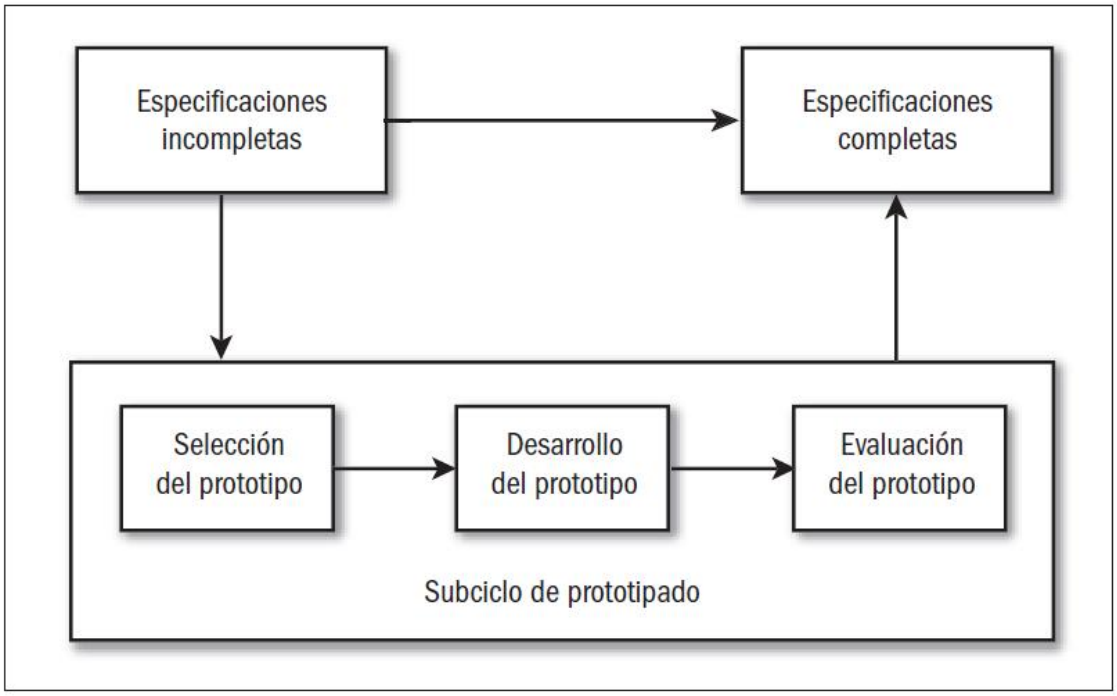


Figura 2.7: Ciclo de vida por prototipos

Se utilizan mayoritariamente en desarrollos de productos con innovaciones importantes, o en el uso de tecnologías nuevas o poco probadas, en las que la incertidumbre sobre los resultados a obtener, o la ignorancia sobre el comportamiento, impiden iniciar un proyecto secuencial.

La ventaja de este ciclo se basa en que es el único apto para desarrollos en los que no se conoce a priori sus especificaciones o la tecnología a utilizar. Como contrapartida, por este desconocimiento, tiene la desventaja de ser altamente costoso y difícil para la administración temporal.

2.2.7- Ciclo de vida evolutivo

Este modelo acepta que los requerimientos del usuario pueden cambiar en cualquier momento como se ve en el diagrama de la Figura 2.8.

La practica nos demuestra que obtener todos los requerimientos al comienzo del proyecto es extremadamente difícil, no sólo por la dificultad del usuario de transmitir su idea, sino porque estos requerimientos a cumplir. El modelo de ciclo de vida evolutivo afronta este problema mediante una iteración de ciclos requerimientos-desarrollo-evaluación.

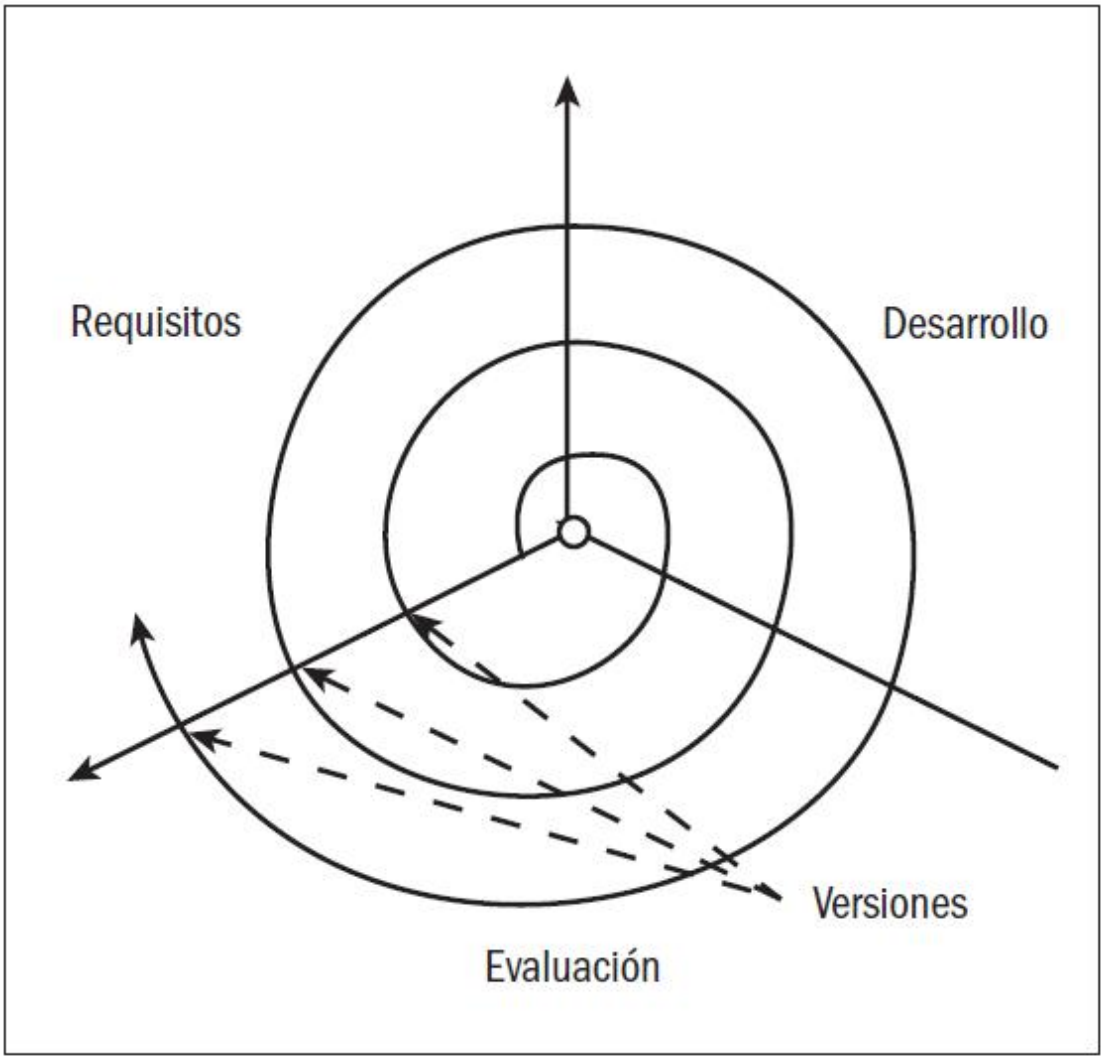


Figura 2.8: Ciclo de vida evolutivo

Resulta ser un modelo muy útil cuando desconocemos la mayoría de los requerimientos iniciales, o estos no están completos.

Tomemos como ejemplo un sistema centralizado de stock-ventas-facturación, en el cual hay muchas áreas que utilizarán la aplicación. Tenemos dos complicaciones: la primera, los usuarios no conocen la informática, la segunda, no es unos, sino varios de los sectores que nos puedan influir en los requerimientos del otro. Se hace necesario, lograr que la aplicación evoluciones hasta lograr las satisfacciones de todos los sectores involucrados.

2.2.8- Ciclo de vida incremental

Este modelo de ciclo de vida se basa en la filosofía de construir incrementando las funcionalidades del programa tal y como se ve en la Figura 2.9.

Se realiza construyendo por módulos que cumplen las diferentes funcionalidades del sistema. Esto permite ir aumentando gradualmente las capacidades del *software*. Este ciclo de vida facilita la tarea del desarrollo permitiendo a cada

miembro del equipo desarrollar un modulo particular en el caso de que el proyecto sea realizado por un equipo de programadores.

Es una repetición del ciclo de vida en casaca, aplicándose este ciclo en cada funcionalidad del programa a construir. Al final de cada ciclo le entregaremos una versión al cliente que contiene una nueva funcionalidad. Este ciclo de vida nos permite realizar una entrega al cliente antes de terminar el proyecto.

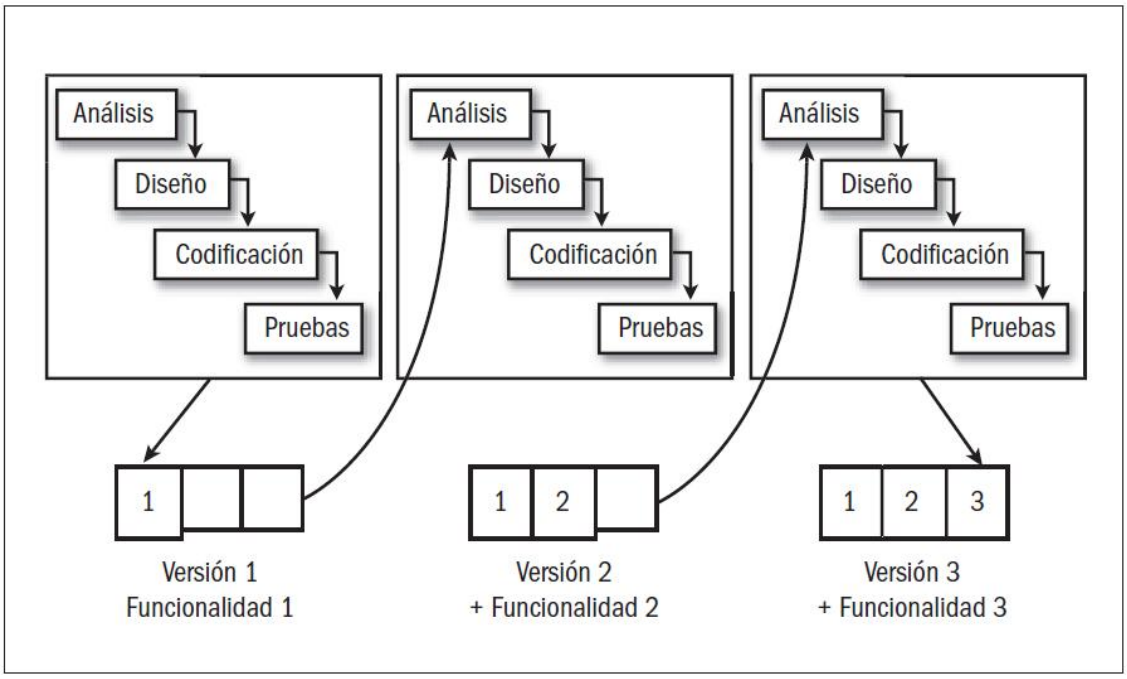


Figura 2.9: Ciclo vida incremental

El modelo de ciclo de vida incremental nos genera algunos beneficios tales como los que se describen a continuación:

- Construir un sistema pequeño siempre es menos riesgoso que construir un sistema grande.
- Como desarrollamos independientemente las funcionalidades, es más fácil revelar los requerimientos del usuario.
- Si se detecta un error grave, sólo desechamos la última iteración.
- No es necesario disponer de los requerimientos de todas las funcionalidades en el comienzo del proyecto y además facilita la labor del desarrollo con la conocida filosofía de *divide & conqueror*.

Este modelo de ciclo de vida no está pensando para cierto tipo de aplicaciones, sino que está orientado a cierto tipo de usuario o cliente. Podremos utilizar este modelo de ciclo de vida para casi cualquier proyecto, pero será verdaderamente útil cuando el usuario necesite entregas rápidas, aunque sean parciales.

2.2.9- Ciclo de vida en espiral

Este ciclo puede considerarse una variación del modelo con prototipo, fue diseñado por *Boehm* en el año 1988. El modelo se basa en una serie de ciclos repetitivos para ir ganando madurez en el producto final, se puede observar su representación en la figura 2.10. Toma los beneficios de los ciclos de vida incremental y por prototipos, pero se tiene más en cuenta el concepto de riesgo que aparece debido a la incertidumbre e ignorancias de los requerimientos proporcionados al principio del proyecto o que surgirán durante el desarrollo. A medida que el ciclo se cumple (el avance del espiral), se van obteniendo prototipos sucesivos que van ganando la satisfacción del cliente o usuarios. A menudo, la fuerte de incertidumbres es el propio cliente o usuario, que en la mayoría de las oportunidades no sabe con perfección todas las funcionalidades que debe tener el producto.

En este modelo hay cuatro actividades que envuelven a las etapas: planificación, análisis de riesgo, implementación y evaluación.

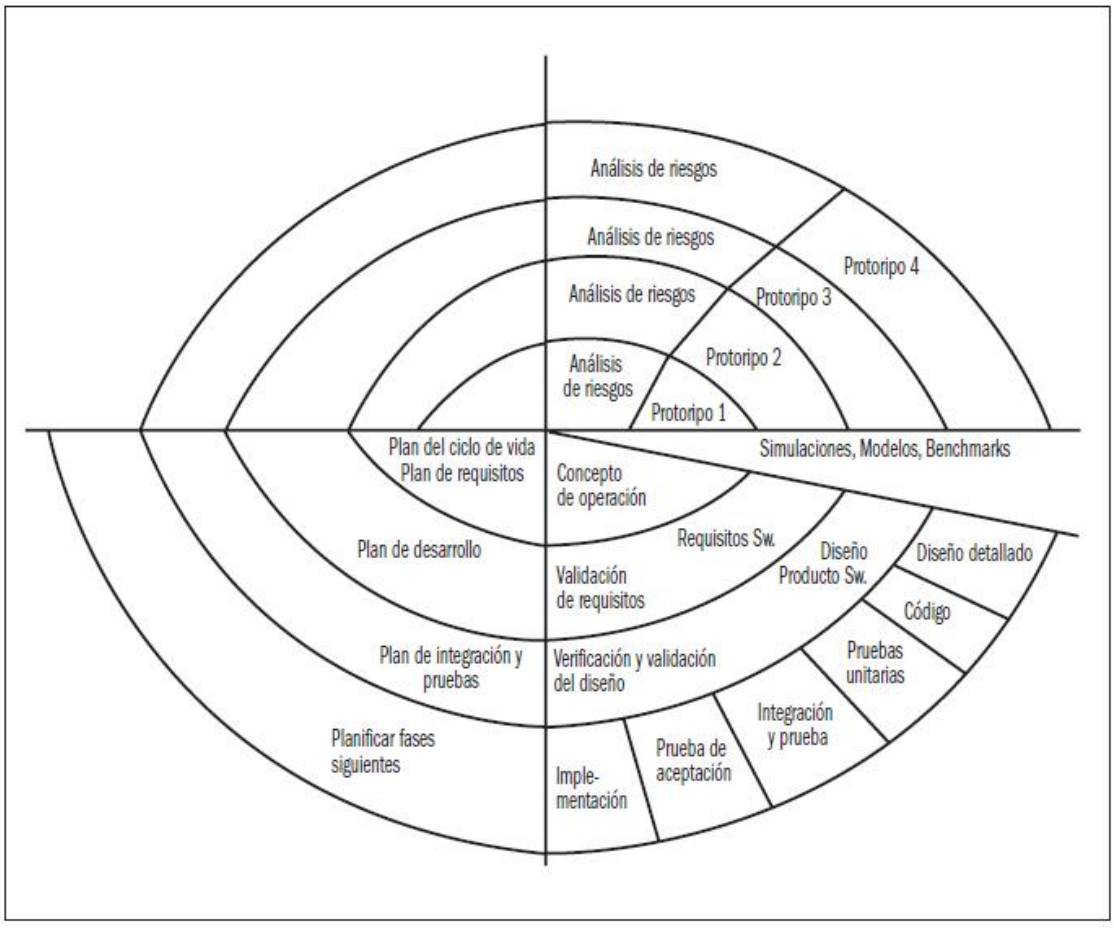


Figura 2.10: Ciclo de vida en espiral

La ventaja más notoria de este modelo de desarrollo de *software* es que puede comenzarse el proyecto con un alto grado de incertidumbre, se entiende también como ventaja el bajo riesgo de retraso en caso de detección de errores, ya que se puede solucionar en la próxima rama del espiral.

Algunas de las desventajas son: el costo temporal que suma cada vuelta del espira, la dificultad para evaluar los riesgos y la necesidad de la presencia o la comunicación continua con el cliente o usuario.

Se observa que es un modelo adecuado para grandes proyectos internos de una empresa, en donde no es posible contar con todos los requerimientos desde el comienzo y el usuario está en nuestro mismo ambiente laboral.

2.2.10- Ciclo de vida orientado a objetos

Esta técnica fue presentada en la década de los 90, tal vez, como una de las mejores metodologías a seguir para la creación de los productos *software*.

Al igual que la filosofía del paradigma de la programación orientada a objetos, en esta metodología cada funcionalidad, o requerimiento solicitado por el usuario, es considerado un objeto. Los objetos están representados por un conjunto de propiedades, a los cuales denominamos atributos, por otra parte, al comportamiento que tendrán estos objetos los denominamos métodos (Figura 2.11).

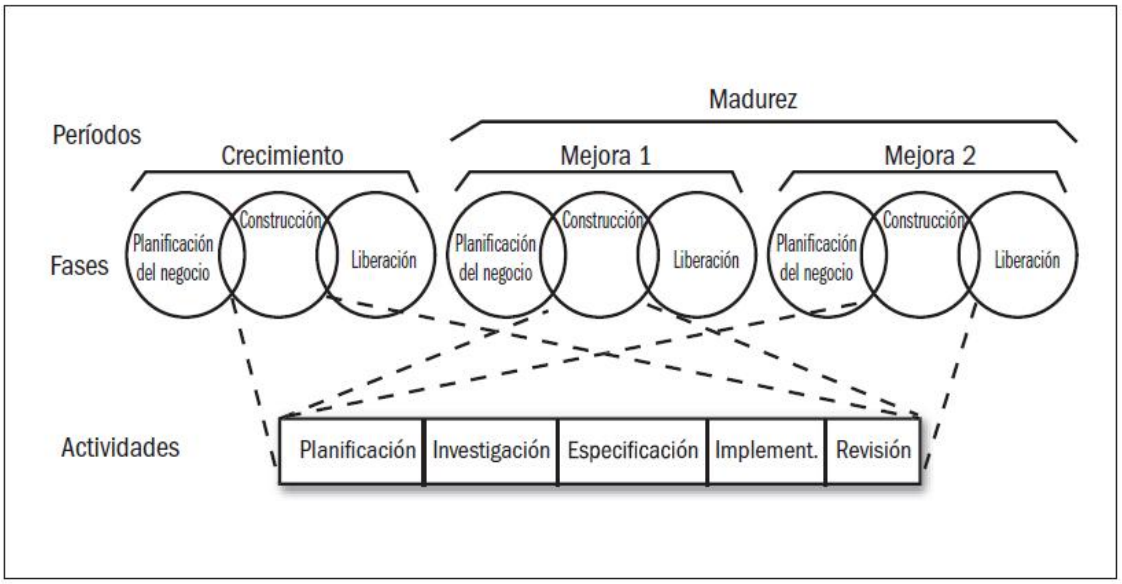


Figura 2.11: Ciclo de vida orientada a objetos

La característica principal de este modelo es la abstracción de los requerimientos de usuario, por lo que este modelo es mucho más flexible que los restantes, que son rígidos en requerimientos y definición, soportando mejor la incertidumbre que los anteriores, aunque sin garantizar la ausencia de riesgos. La abstracción es lo que nos permite analizar y desarrollar las características esenciales de un objeto (requerimiento), despreocupándonos de los menos relevantes.

En este modelo se utilizan las llamadas fichas CRC (clase-responsabilidades-colaboración) como herramienta para obtener la abstracción y mecanismos clave de un sistema analizado los requerimientos del usuario. Estas fichas nos ayudarán a confeccionar los denominados casos de uso.

2.3 Conclusión

Después de presentar los diferentes modelos de ciclo de vida utilizados nos preguntamos: ¿Qué modelo de ciclo de vida elegir?**.** Debemos elegir el modelo que mejor se adapte al proyecto que desarrollaremos. Se debe analizar, para poder guiarnos en nuestra elección, la complejidad del problema, el tiempo que disponemos para hacer la entrega final, o si el usuario o cliente desea entregas parciales, la comunicación que existe entre el equipo de desarrollo y el usuario y por último, qué certeza (o incertidumbre) tenemos de que los requerimientos dados por el usuario son correctos y completos.

El ciclo de vida en el desarrollo de los productos de *Pharma S.L* sigue el llamado Modelo en V (*V-Model*) que se muestra en la Figura 2.12**.**

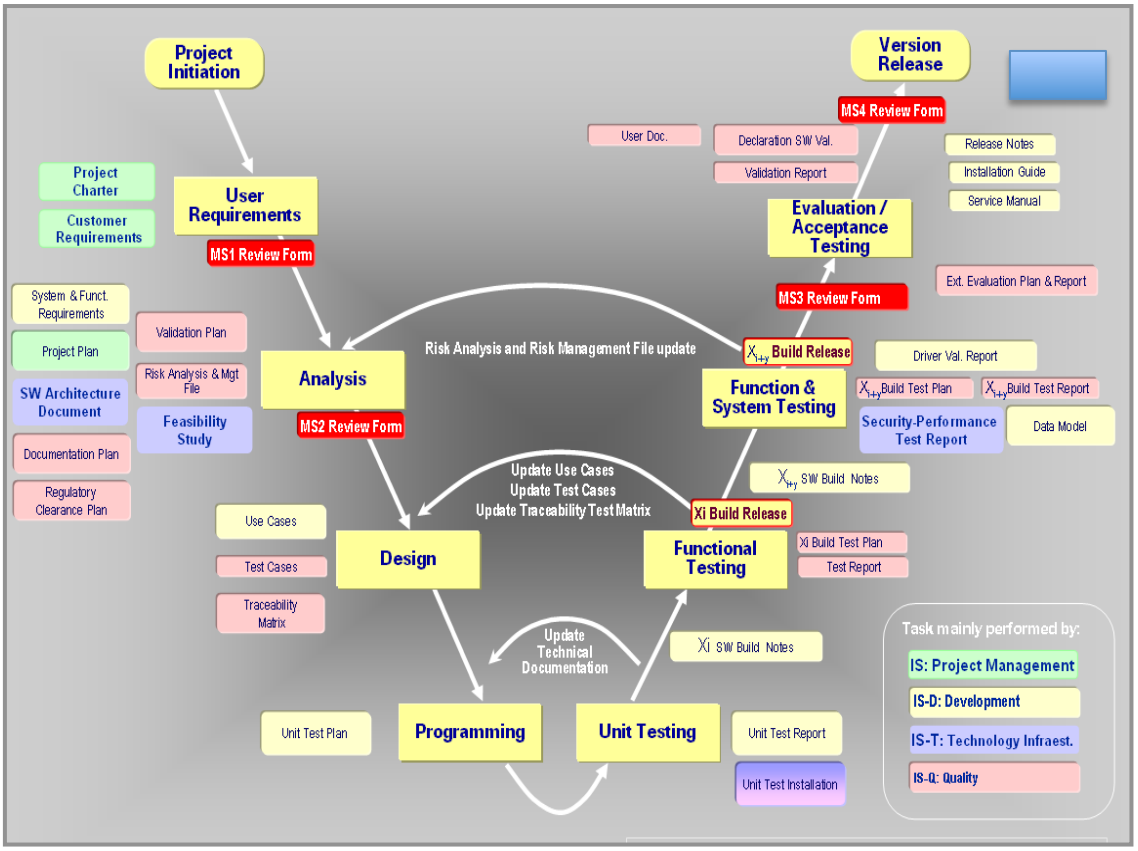


Figura 2.12: Modelo en V de desarrollo de Software

Este modelo se caracteriza porque cada fase de desarrollo lleva aparejada en paralelo una fase de verificación.

3- ESTUDIO DE VIABILIDAD

3.1 Introducción

El proyecto que se presenta consistirá en realizar un estudio de verificación y validación de un *software* orientado a sanidad, consiguiendo un certificado de calidad que garantice un producto fiable y seguro.

La ventaja de realizar una validación de un *software* será: obtener un *software* de calidad con el cual se evita que ocasione fallos que puedan causar riesgos con consecuencias graves para los pacientes.

El inconveniente principal de realizar la validación y verificación de un *software* es el tiempo necesario para llevarlo a cabo. Si un sistema necesita pasar por un control de calidad el tiempo de lanzamiento al mercado es más tardío. Debido a que al equipo de calidad se le da un periodo limitado para realizar la validación del sistema, eso hace que la mayoría de veces no sea posible cubrir todos los casos de prueba ni cumplir todos los plazos de entrega.

3.2 Objetivo

3.2.1- Descripción de la situación a tratar

Actualmente existen sistemas de información para centros de atención primaria (CAP) y laboratorios que consigan tratar toda la información mediante un entorno *web*. Se pretende obtener un *software* seguro y de calidad con mejores prestaciones, consiguiendo una interfaz igual o más rápida que para aplicaciones instaladas en cada máquina de usuario.

3.2.2- Perfil de usuario

La aplicación no tiene ningún tipo de límite en cuanto a usuarios, tanto puede ser utilizado por una secretario/a o un técnico/a de laboratorio o un médico o enfermero/a... todos ellos con un soporte informático y conexión a *internet*.

3.2.3- Objetivos

El objetivo principal es hacer que el *software* sea robusto, fiable y que cumpla todos los requerimientos definidos para la aplicación. Esto hace que la validación deba ser exhaustiva. Se deberán validar los siguientes puntos:

- Instalación de la aplicación en diferentes servidores.
- Gestión de datos del CAP/laboratorio.
- Envío y recepción de datos de periféricos.
- Interfaz con el usuario.
- Rendimiento.

3.3 Sistema a realizar

3.3.1- Descripción

Se deberá realizar un plan de *test* sobre los módulos a validar, con pruebas de *test* sobre cada módulo. Los errores encontrados se introducirán en una base de datos mediante una interfaz para que se puedan solucionar según su gravedad para el producto y para el paciente. La validación de la versión se dividirá en *builds*, en cada *build* se decidirán los errores y las nuevas implementaciones que se deben arreglar para que el sistema funcione correctamente. Finalmente se realizará una validación externa en un centro hospitalario con la última *build* presentada. Las *builds* están realizadas sobre una misma versión de la aplicación.

3.3.2- Recursos

Se utilizarán los siguientes recursos *software*:

- **Rational Clear Quest**:

Base de datos de defectos y cambios.

- **HP Quality Center**

Repositorio de test cases (TC's). Esta aplicación permite crear y almacenar los TC's necesarios para la validación de la aplicación. Además permite crear la matriz de trazabilidad.

- **Virtual Box**

Con el objetivo de disponer de diferentes escenarios de test, y de facilitar el intercambio entre ellos, se utiliza Virtual Box de Oracle. Esta herramienta permite simular varios PC's virtuales en un PC físico. De esta manera se pueden recrear los diferentes escenarios de test en que se valida *TAONet*, diferenciándose entre ellos entre otras cosas en el Sistema Operativo o en el idioma de la Interfaz de Usuario. Como cada instancia se basa en una máquina virtual, es posible recrear gran variedad de entornos de test sin necesidad de usar una gran cantidad de recursos "hardware".

- **VMWare**

Con el objetivo de disponer de diferentes escenarios de test, y de facilitar el intercambio entre ellos, se utiliza además VMWare. Esta herramienta permite simular varios PC's virtuales en un PC físico. De esta manera se pueden recrear los diferentes escenarios de test en que se valida *TAONet*,, diferenciándose entre ellos entre otras cosas en el Sistema Operativo o en el idioma de la Interfaz de Usuario. Como cada instancia se basa en una máquina virtual, es posible recrear gran variedad de entornos de test sin necesidad de usar una gran cantidad de recursos "hardware".

Recursos hardware para las pruebas:

- Coagulómetro Coaguchek® S / XS Plus Lectores ópticos.

3.3.3- Evaluación de riesgos

- Evaluación de riesgos sobre el paciente:

Cumplir con las necesidades básicas de los requerimientos para conseguir una información fiable y que no pueda ser fraudulenta.

- Incompatibilidad entre los navegadores de *internet*:

En la actualidad los navegadores más utilizados son el IE y *Mozilla Firefox*. Al realizar el proyecto se debe tener en cuenta ya que cada navegador tiene particularidades y se deberán implementar diferentes opciones según en qué plataforma se trabaje.

- Problemas de conexión:

Este problema debería de ser el más importante a tratar ya que el trabajo de la información se realiza en un servidor a que debemos acceder mediante una conexión a *internet* segura y solvente.

Este problema afecta a todo nivel ya que el traspaso de información se hace desde un servidor remoto a un cliente y viceversa. También hay que tener en cuenta el traspaso de información sobre los elementos periféricos al servidor y viceversa.

- **Actualización o inserción del nuevo SW:**

Este problema no debe ser algo muy complicado a tratar, puesto que se procederá a hacer el cambio con el paso de datos sobre el nuevo sistema. Puede que este proceso no sea rápido pero no debe ser complicado.

- Seguridad:

Encriptación del código de desarrollo para evitar posibles modificaciones que afecten a la funcionalidad de la aplicación y por lo tanto afecten a la calidad del producto.

Se deben controlar las bases de datos donde se almacene toda la información para que no haya posibles intromisiones en ellas y se viole la confidencialidad del paciente.

3.3.4- Organización del proyecto

- Definición de las etapas y metodología:

Obtener un listado de los problemas, deficiencias y mejoras sobre el sistema instalado actualmente.

1. Requisitos o requerimientos del sistema (funcionales y de usabilidad).
2. Organización de responsabilidades.
3. Planificación de las etapas de desarrollo del proyecto.
4. Realización del proyecto

- Definición de alternativas:

Al realizar el proyecto se pueden encontrar más defectos de los esperados por lo tanto se decide realizar más *builds* del sistema con lo que comportará un aumento del tiempo estimado. Esto comportará que se puedan utilizar diferentes métodos de validación, ya que el tiempo de verificación de la *build* se reducirá.

Características importantes a tener en cuenta:

1. Actualización del *software*: se deberían contemplar métodos que lo soporten.
2. Actualización del entorno *web* y máquina utilizada por el usuario: se deben incluir en la verificación las diferentes versiones que aparecen de los navegadores, *frameworks*, java y cualquier programa que necesite nuestro *software* que sea ejecutado desde cliente.

El código, documentación y demás herramientas necesarias son de uso interno hasta que no se libere la versión. Cualquier filtración sería un problema de cara a la competencia.

3.4 Modelo de desarrollo

En el ciclo de desarrollo (*development life cycle*) de un producto, se pueden seguir diversos modelos:

- Modelo en V
- Modelo en espiral
- Prototipo incremental
- Desarrollo de aplicación rápida

Todos ellos tienen en común que son útiles para definir, construir y testear el sistema.

En el desarrollo de un *software* en entorno regulado lo más habitual es seguir el modelo en V representado en la Figura 3.1

Este modelo refleja la correspondencia que existe entre las distintas fases de desarrollo y las de validación, así como las distintas actividades de validación y verificación que se llevan a cabo durante el mismo.

La razón de esta correspondencia es porque deben realizarse pruebas de toda la documentación y entregables generados durante la creación del producto final: las especificaciones de los requisitos, los casos de uso (UC), los diagramas de diversos tipos, el código fuente y las distintas partes de la aplicación.

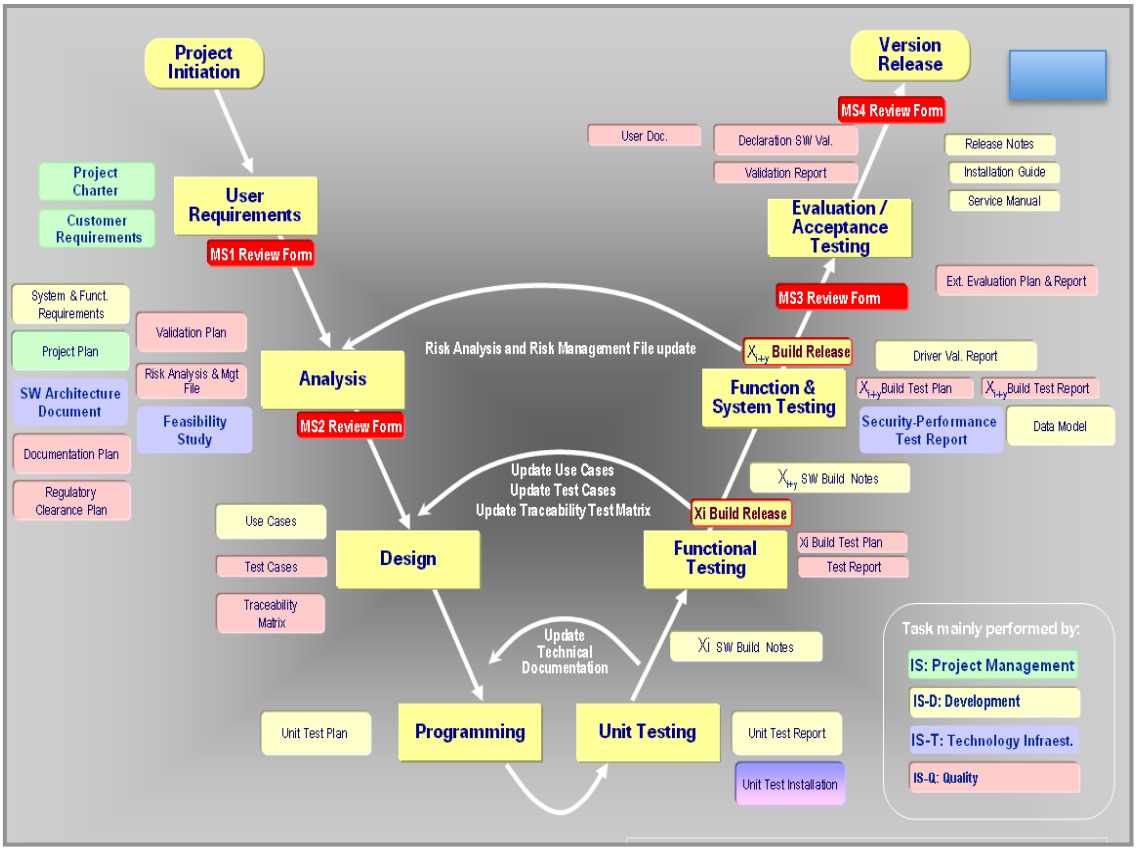


Figura 3.1: Modelo en V de desarrollo de Software

En las distintas etapas de desarrollo del sistema o proceso de verificación se testean cada una de las partes finalizadas: los requerimientos del sistema, las especificaciones de más alto nivel (funcionales), las de más bajo nivel (de diseño) y, finalmente, el propio código del *software*.

De esta manera, la correcta generación de cada una de las partes que componen el producto hará que el *software* final también sea correcto.

Durante la validación, se comprueba el cumplimiento de los objetivos de los requisitos y del producto final realizando idealmente los siguientes tipos de pruebas, que de menor a mayor complejidad, son:

- Componente de test
- Prueba de integración a pequeña escala
- Test funcional
- Test no funcional
    - Pruebas de carga, de rendimiento y estrés
    - seguridad
    - almacenaje y volumen
    - usabilidad
    - instalación
    - documentación

- Pruebas de integración a gran escala
- Test de aceptación

Estos términos se explican con más detalle en el glosario adjunto al final del proyecto.

3.5 Planificación del proyecto

Para el desarrollo del proyecto se han planteado inicialmente las siguientes actividades:

No.	Descripción actividad	Duración (en horas)
1	Estudio viabilidad	80
2	Realizar organización de *builds* a validar	40
3	Diseño de las pruebas según requerimientos	320
4	Diseño del plan de pruebas *build* 1	40
5	Realizar la validación *build* 1	104
6	Informe de pruebas *build* 1	8
7	Diseño del plan de pruebas *build* 2	40
8	Diseño de pruebas para *build* 2	40
9	Realizar la validación *build* 2	72
10	Informe de pruebas *build* 2	8
11	Diseño del plan de pruebas *build* 3	40
12	Diseño de pruebas para *build* 3	40
13	Realizar la validación *build* 3	56
14	Informe de pruebas *build* 3	8
15	Evaluación externa	24
16	Informe de validación final de versión	8
	Total de horas	928

Tabla 3.1: Planificación del proyecto

Se iniciará el proyecto el 15 de septiembre 2012 y finalizará el 31 de diciembre del 2012. En el proyecto lo realizarán 3 personas, un Team Leader y dos ingenieros de validación. Destacar que para el proyecto he realizado todas las tareas del proyecto, las de Team Leader y las de los ingenieros de validación.

3.6 Conclusiones

Con la validación y verificación se conseguirá un *software* rígido, fiable y de calidad de tal manera que sea atractivo para el mercado y que no comporte riesgos para el paciente.

Se reducirán los posibles errores en el cliente con lo que se eliminarán los riesgos de negocio y de paciente.

Considerando las ventajas y desventajas y habiendo hecho un estudio de los riesgos del producto y del paciente esta aplicación se considera viable.

4- DESARROLLO DE LA VALIDACIÓN DE TAONet 3.1

4.1 Antecedentes a TAONet 3.1

Actualmente TAONet 3.1 es la primera versión de actualización de software de TAOnet 3.0.

Tanto TAONet 3.0 como su versión actualizada TAONet 3.1 son un sistema de información de laboratorio o LIS (*Laboratory Informations System*) consistente en un programa *software* instalado en un ordenador y a cuyos datos se puede acceder desde otros ordenadores mediante un navegador *web* como *Mozilla Firefox* o *Microsoft Internet Explorer*. Este *software* está destinado a la recepción y gestión de resultados de *tests* procedentes de instrumentos de medición conectados al sistema, de manera que se pueda obtener y procesar información relativa al estado de salud de los pacientes, permitiendo detectar en base al procesado de estos resultados posibles anomalías, facilitar el diagnóstico y supervisar la administración de las medidas terapéuticas consignadas por los facultativos.

4.2 Funcionalidades de TAONet 3.1

TAONet 3.1 es un software que facilita la gestión de pacientes ambulatorios en tratamiento anticoagulante oral (TAO) ofreciendo soporte informático a las tareas específicas del personal de los centros a los que acuden los pacientes.
El TAO se utiliza en la prevención y tratamiento de las enfermedades tromboembólicas.

TAONet 3.1 ha sido desarrollado utilizando tecnología Web, lo que permite una integración total entres los Centros de Atención Primaria (CAP) y los centros de Atención Especializada (hospitales, laboratorios,…), proporcionando acceso a la información del paciente de forma flexible, escalable, fiable y multi-centro.

TAONet 3.1 Permite el manejo y coordinación de las diferentes tareas de los centros médicos / hospitales donde el paciente acude para el seguimiento del tratamiento de la enfermedad tromboembólica.

- Permite recibir los resultados de la determinación de INR* del paciente realizados por el coagulómetro Coaguchek® S / XS Plus desde dicho dispositivo.
- Asiste en la prescripción facultativa de pacientes bajo tratamiento TAO mediante el cálculo de la distribución de la dosis de anticoagulante en función de los resultados de la determinación de INR* del paciente.
- Genera un informe de la toma diaria de anticoagulantes orales para facilitar el seguimiento en domicilio del tratamiento por el paciente hasta la próxima visita en el centro médico.
- Almacena información relacionada con el paciente, relativa al diagnóstico, resultados de analíticas de laboratorio, demográficos.
- Gestiona la agenda del paciente, incluyendo las visitas al hospital / centro médico

- Dispone de herramientas de configuración de accesos para los diferentes tipos de usuarios: hematólogos, médicos de atención primaria, personal de enfermería.

4.3 Estrategia a seguir

Para el desarrollo de TAONet 3.1 se ha decidido aplicar el modelo de ciclo de vida del *software* en V considerando este módulo particular como un producto nuevo y por tanto sujeto a los pertinentes *Milestones*. Estos *Milestones* afectarán a todo el proyecto TAONet 3.1.

Las actividades llevadas a cabo por el equipo de validación serán las siguientes:

- *Test* Funcional: consiste en ejecutar las pruebas descritas (*test cases*).
- *Smoke Tests*: batería de pruebas básicas realizadas antes de iniciar la *build*.
- *Test* de componentes *Zen* imbuidos en las pantallas: pruebas de *framework*, *grids*, botones, en general pruebas de interfaz.
- Cobertura de *test.*
- *Builds* internos.
- Estrategia de *test* basada en riesgo: se desarrollará una prioridad de ejecución según el riesgo.
- Control de versiones de *Test Cases*: cada vez que se modifique un *Test Case* será necesario modificar su versión, imprimir la modificación y firmarlo.
- Evidencia objetiva: para los *Test Cases High* será necesario hacer "pantallazos" sobre cada punto de verificación de *Test Case*.
- Estrategia de regresión: la regresión deberá hacerse acorde con lo implementado teniendo en cuenta los elementos a los que puede afectar.

4.4 Validación de TAONet 3.1

Para realizar la validación de esta versión de TAONet se ha acordado que serán necesarios 3 *builds* para la versión 3.1 a través de los cuales se irán validando los diferentes módulos de la aplicación. Los módulos a validar son los siguientes:

- Módulo de paciente

  Este módulo incluye la funcionalidad correspondiente a la gestión de los pacientes y a la aplicación del tratamiento TAO de los mismos. Incluye la gestión y almacenamiento de datos de paciente (demográficos, diagnósticos, resultados de analíticas de laboratorio, tareas, visitas del paciente al hospital, cálculo y distribución de la dosis de anticoagulante oral y gestión de resultados de INR recibidos del coagulómetro Coaguchek® S / XS Plus)

- Estadísticas

  Permite crear y visualizar diferentes tipos de estadísticas (gráficos y tablas) y filtrar por diferentes criterios (centros de salud, médicos, sexo, fecha, diagnósticos, etc.).

- Agenda

  Permite realizar tareas de gestión de la agenda en la que se incluye el listado de pacientes y las correspondientes citas planificadas en el hospital / centro médico dentro de una franja temporal.

- Administración

  Permite tareas relacionadas con los usuarios y los elementos que les permiten trabajar con TAONet: (alta/baja/modificación de usuarios, gestión de perfiles de usuarios, administración de centros médicos, configuración de listados de Diagnósticos, notas clínicas, medicamentos, distribución de las dosis, formularios y agenda). El acceso a esta parte de la aplicación depende del perfil del usuario que ha entrado en TAONet.

La validación se inicia el día 7 de noviembre cuando el departamento de desarrollo libera el primer *build* el día 7 de noviembre de 2021. Cada *build* interno queda reflejado en el *test plan* de cada *build*.

Al realizar la validación se demuestra que los requerimientos funcionales y de sistema se han implementado en el *software* de forma correcta, completa, exacta y consistente. Para que la versión final no contenga errores, sobre todo con riesgo alto para el producto y/o para el negocio, se revisan todos los módulos para que los clientes tengan una versión fiable.

El flujo a seguir se detalla en la Figura 4.2 :

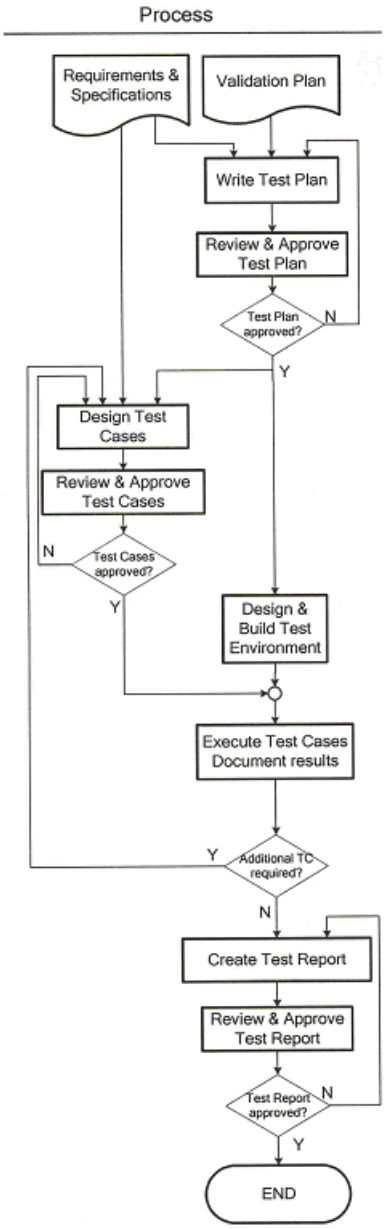


Figura 4.2: Estrategia a seguir

4.4.1- Validación del departamento de desarrollo

Las pruebas realizadas por el departamento de desarrollo tienen una gran importancia, el nombre que reciben son Pruebas unitarias (*Unit test*). El objetivo de estas pruebas es verificar que cada error corregido o nueva implementación, de forma individual, funciona de la manera descrita en el error o en el cambio. De esta manera se da a entender que el departamento de desarrollo es quien se encargará tanto de diseñar como de ejecutar estas Pruebas unitarias.

Los *Unit test* ejecutados por el departamento de desarrollo siguen un procedimiento especificado y desarrollado internamente.

La ejecución y diseño de las Pruebas unitarias quedan definidos en el apartado del *Unit test* de la herramienta utilizada *Clear Quest*, quedando reflejados los pasos seguidos para comprobar el error o cambio a nivel unitario para cada error.

Tal como se ve en la Figura 4.3 el desarrollador ha detallado su implementación indicando qué elementos del código ha modificado e indicando también la versión del componente o clase modificada. Entonces la persona que tiene asignado hacer el *Unit test* recibe el error y con la descripción dada del error más la implementación que indica el desarrollador se encarga de revisar que la solución del error esté bien implementada.

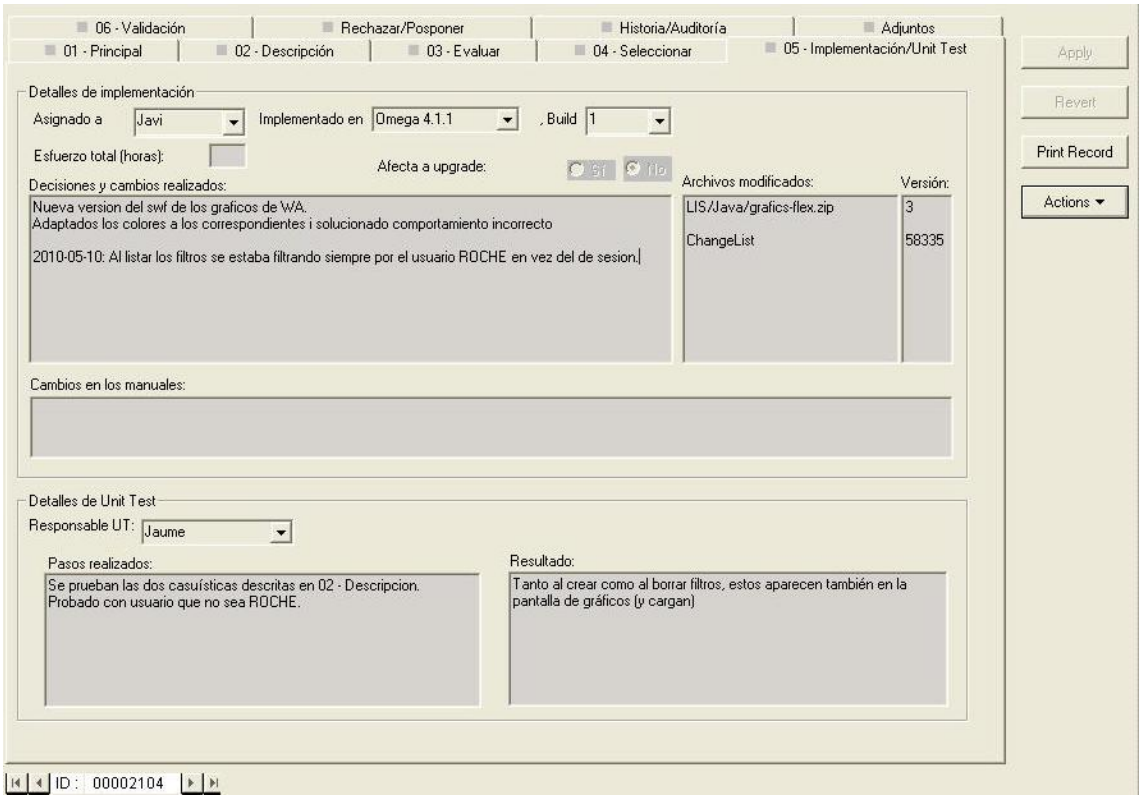


Figura 4.3: Unit Test en Clear Quest

Otra función realizada por parte del equipo de desarrollo son las revisiones de código, que consisten en:

- Detectar errores antes de que éstos sean integrados en el *build*, permitiendo solventarlos de forma instantánea ahorrando los costes de tiempo, dinero y recursos que supondría la detección del error en fases posteriores.
- Verificar que los estándares de programación se aplican de forma adecuada y consistente, consiguiendo un estilo de código común.
- Verificar que el código se genera de acuerdo a las especificaciones de diseño.
- Verificar que el código puede ser mantenido de forma eficaz y efectiva a lo largo de todo el ciclo de vida del producto.

- Verificar que los cambios se ajustan al ámbito acordado.
- Verificar que la codificación respeta el desarrollo de una arquitectura modular y escalable; así como la reutilización de código que evite el uso de implementaciones diferentes para una misma funcionalidad.
- Detectar y solucionar incongruencias o discrepancias en los comportamientos de una misma funcionalidad realizada por o para módulos diferentes.
- Formar a los programadores "junior", resolviendo de forma práctica sus dudas. Compartir conocimiento entre los componentes del equipo, de forma que se transfiera de los más expertos a los menos expertos.
- Detectar implementaciones de código no "refinadas" en el acceso a la base de datos que puedan provocar problemas de rendimiento en las aplicaciones.

4.4.2- Estrategia a seguir por el departamento de calidad

Las pruebas realizadas por el departamento de calidad tienen el nombre de Pruebas de Sistema. Éstas consisten en la verificación de la correcta integración de los diferentes componentes y módulos de que consta la aplicación. Se evalúa la aplicación entendida como un todo, en la misma manera y presentación en que será adquirida por los clientes. Con las pruebas de sistema la aplicación se verifica como un sistema completo, utilizando principalmente para ello la Interfaz de Usuario.

Se verifican todos los cambios y los errores solucionados que se describen en un documento llamado PCO (*Product Change Order*). La verificación se extiende a aquellas áreas o funcionalidades próximas o relacionadas con la funcionalidad afectada por el cambio o error. Aquellas partes o módulos no afectados por cambios o corrección de errores son validados y verificados también (test de regresión) a fin de comprobar que su funcionamiento permanece inalterado y no ha sido afectado por efectos colaterales no deseados.

El equipo diseña *Test Cases* (Casos de pruebas) donde el tester describirá los pasos que serán ejecutados sobre la aplicación a fin de comprobar que su implementación sea correcta. El diseño de estos *Test Cases* se detalla en el apartado 4.4.2.2 Diseño de los *Test Cases***.**

Las pruebas se priorizan en función del riesgo (para el paciente o usuario y para el negocio) asociado a cada cambio o error. Este riesgo viene dado por el requerimiento al que está asociada la prueba o al riesgo que el error o cambio tiene asignado en el *Clear Quest*. La evaluación de estos errores va a cargo del departamento de desarrollo.

La información de cada estrategia planeada para cada *build* de la aplicación a testear, se realiza en un *Test Plan* diferente para cada *build*. En cada uno se detallan tanto los escenarios donde se realizará la validación, el número de pruebas a realizar y los errores y cambios a validar. Se incluyen estadísticas tanto de errores validados por tester como sobre qué pruebas se han realizado con éxito y cuáles no. En el apartado 4.4.2.1 se detalla la información del *Test Plan*.

Los errores y cambios implementados planeados para esta versión se detallan en los documentos:

- PCO (*Product Change Order*): se definen los errores y cambios planeados para TAONet 3.1.
- PLN (*Product Launch Notification*): se definen los errores y cambios finalmente implementados para TAONet 3.1.

Los errores y cambios de los que se habla han sido encontrados en la versión anterior de la aplicación TAONet 3.0.

4.4.2.1 Test Plan- & Test Report

En el *test plan*, Figura 4.4, 4.5 y 4.6, se organizan las pruebas a realizar en cada *build*. En cada uno de estos planes aparece la siguiente información:

- Escenarios y entorno de *test* empleados, en lo referente a sistemas operativos, navegadores, idiomas y configuración regional.
- Propósitos y objetivos de la ronda de *test*, consignando los módulos a verificar.
- Estrategia de *test*, si difiriese de la consignada en este documento.
- Selección de *Test Cases*, diferenciados por módulos.
- Selección de *Test Cases* utilizados como regresión.
- Selección de errores que se van a validar.
- Versión del *Test Case.*
- Prioridad del *Test Case.*
- Tester asignado a la ejecución del *Test Case.*

Plan de Pruebas Funcional	
Ronda de Test	
Ronda	1
Propósitos y Objetivos de la Ronda de Test	
El propósito de esta ronda es verificar y validar todos los errores y/o cambios definidos en el Product Change Order (PCO). El objetivo de esta ronda es detectar cualquier nuevo error que pudiera suponer un alto riesgo desde el punto de vista del paciente o del negocio con tal de que sea solventado para la ronda siguiente. Asimismo, se plantean TC's a modo de regresión para ver que no se ven afectadas las diferentes funcionalidades del programa	
Build	
Nombre de la build:	TAONet 3.1 build 1
Documentos relacionados	
Plan de validación	TAONet - Plan de Validación.doc
Estimación de fechas	
Fecha de inicio estimada:	07-nov-2012
Fecha final estimada:	10-dic-2012
Esfuerzo estimado (días):	24
Estrategia de Test	
Consultar el documento TAONet - Plan de Validación.doc	
Recursos, Roles & Responsabilidades	
Empleado 3: Validation Team Leader	
Raúl Villegas: Software Validation Engineer	

Figura 4.4: Plan de pruebas del Test plan1

Nombre del Escenario	Topología	Escenarios Virtuales	SO	Navegador	Lenguaje de la Aplicación	Config. Del Cliente	Tipo de Instalación	Otros	Notas
Escenario 1	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: Oracle 10g	Microsoft Windows XP Professional Version 2002, Service Pack 2 Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain) Microsoft Windows XP Professional Version 2002, Service Pack 2 Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain)	IE7.0	Castellano	Ninguna	Actualización	Microsoft Virtual PC 2007	
Escenario 2	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: MySQL 5.1	Debian GNU/LINUX 5.0 Debian GNU/LINUX 5.0	Firefox 3.5	Inglés	Ninguna	Actualización	VMWare	
Escenario 3	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: Oracle 10g	Microsoft Windows XP Professional Version 2002, Service Pack 2 Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain) Microsoft Windows XP Professional Version 2002, Service Pack 2 Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain)	IE6.0	Castellano	Ninguna	Actualización	Microsoft Virtual PC 2007	
Escenario 4	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: MySQL 5.1	Microsoft Windows 7 Professional Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain) Microsoft Windows 7 Professional Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain)	IE8.0	Castellano	Ninguna	Actualización	Microsoft Virtual PC 2007	
Escenario 5	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: MySQL 5.1	Microsoft Windows 2008 Professional Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain) Microsoft Windows 2008 Professional Lenguaje: Spanish (Spain) Opciones Regionales : Spanish (Spain)	IE8.0	Castellano	Ninguna	Nueva	Microsoft Virtual PC 2007	
Escenario 6	Independiente	Servidor de aplicaciones: Apache TomCat v1.6 Servidor de base de datos: SQL server	Professional Lenguaje: Spanish (Spain) Professional Lenguaje: Spanish (Spain)	IE8.0	Castellano	Ninguna	Nueva	Microsoft Virtual PC 2007	

Entorno de Test
HPQC
ClearQuest

Métricas	
Número total de TC's seleccionados	373

Área del gráfico

Figura 4.5: Plan de pruebas del Test plan2

Matriz de Ejecución - Ronda 1

Escenarios	TC's		Suma	Resultados	%	Testers	TC's ejecutados
1	44		0	OK	#¡DIV/0!	Team Leader	0
2	69		0	NOK	#¡DIV/0!	Raúl	0
3	83		0	N/A	#¡DIV/0!		
4	77						
5	57						
6	43						
Total	373		0			% ejecutado	0,00%

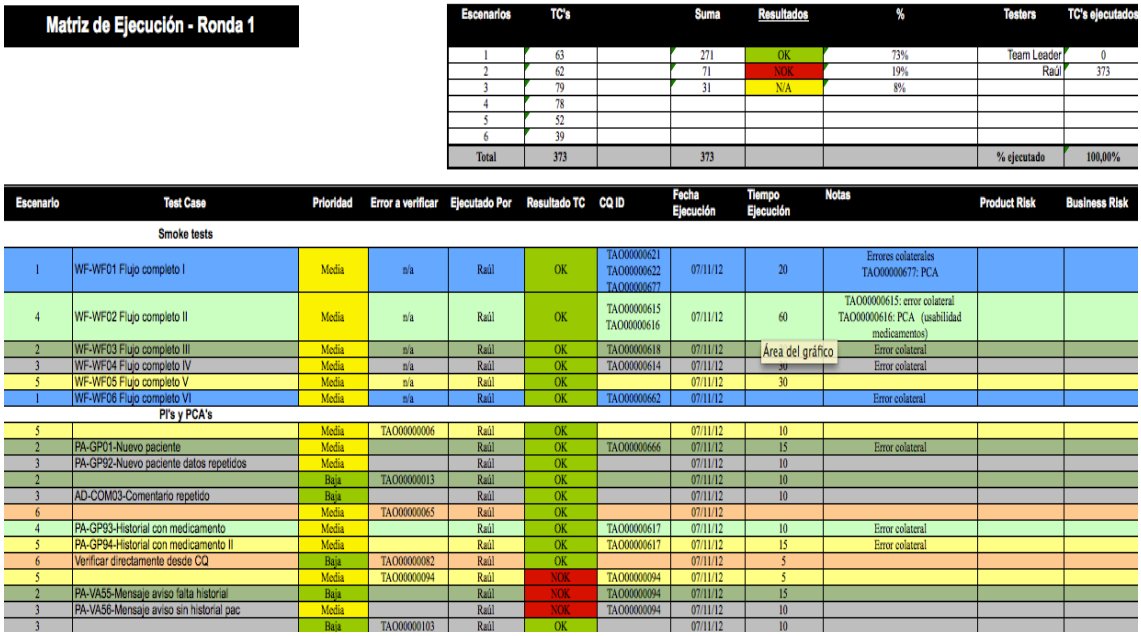
Escenario	Test Case	Prioridad	Error a verificar	Ejecutado Por	Resultado TC	CQ ID	Fecha Ejecución	Tiempo Ejecución	Notas	Product Risk	Business Risk	Implementation Risk
	Smoke tests											
1	WF-WF01 Flujo completo I	Media	n/a									
4	WF-WF02 Flujo completo II	Media	n/a									
2	WF-WF03 Flujo completo III	Media	n/a									
3	WF-WF04 Flujo completo IV	Media	n/a									
5	WF-WF05 Flujo completo V	Media	n/a									
6	WF-WF06 Flujo completo VI	Media	n/a									
	PI's y PCA's											
5		Media	TAO00000006									
2	PA-GP01-Nuevo paciente	Media										
3	PA-GP92-Nuevo paciente datos repetidos	Media										
2		Baja	TAO00000013									
3	AD-COM03-Comentario repetido	Baja										
6		Media	TAO00000065									
4	PA-GP93-Historial con medicamento	Media										
5	PA-GP94-Historial con medicamento II	Media										
6	Verificar directamente desde CQ	Baja	TAO00000082									

Área del gráfico

Figura 4.6: Matriz de ejecución del Test Plan

El documento sobre el cual se trabaja en cada *build* es el *test report*, el que es el mismo documento que el *test plan* pero en el cual se informa en la matriz de ejecución, Figura 4.7, de los resultados obtenidos al realizar la ejecución de los *test cases*. Para cada test case se informará de:

- Resultado del *Test Case* (OK o NOK)
- Versión del *Test Case*
- Identificador *Clear Quest* de los errores encontrados
- Tester que ejecutó el *Test Case*
- Fecha de ejecución
- Tiempo de ejecución
- Notas



Matriz de Ejecución - Ronda 1

Escenarios	TC's		Suma	Resultados	%	Testers	TC's ejecutados
1	63		271	OK	73%	Team Leader	0
2	62		71	NOK	19%	Raúl	373
3	79		31	N/A	8%		
4	78						
5	52						
6	39						
Total	373		373			% ejecutado	100,00%

Escenario	Test Case	Prioridad	Error a verificar	Ejecutado Por	Resultado TC	CQ ID	Fecha Ejecución	Tiempo Ejecución	Notas	Product Risk	Business Risk
	Smoke tests										
1	WF-WF01 Flujo completo I	Media	n/a	Raúl	OK	TAO00000621 TAO00000622 TAO00000677	07/11/12	20	Errores colaterales TAO00000677: PCA		
4	WF-WF02 Flujo completo II	Media	n/a	Raúl	OK	TAO00000615 TAO00000616	07/11/12	60	TAO00000615: error colateral TAO00000616: PCA (usabilidad medicamentos)		
2	WF-WF03 Flujo completo III	Media	n/a	Raúl	OK	TAO00000618	07/11/12		Error colateral		
3	WF-WF04 Flujo completo IV	Media	n/a	Raúl	OK	TAO00000614	07/11/12		Error colateral		
5	WF-WF05 Flujo completo V	Media	n/a	Raúl	OK		07/11/12	30			
1	WF-WF06 Flujo completo VI	Media	n/a	Raúl	OK	TAO00000662	07/11/12		Error colateral		
	PI's y PCA's										
5		Media	TAO00000006	Raúl	OK		07/11/12	10			
2	PA-GP01-Nuevo paciente	Media		Raúl	OK	TAO00000666	07/11/12	15	Error colateral		
3	PA-GP92-Nuevo paciente datos repetidos	Media		Raúl	OK		07/11/12	10			
2		Baja	TAO00000013	Raúl	OK		07/11/12	10			
3	AD-COM03-Comentario repetido	Baja		Raúl	OK		07/11/12	10			
6		Media	TAO00000065	Raúl	OK		07/11/12				
4	PA-GP93-Historial con medicamento	Media		Raúl	OK	TAO00000617	07/11/12	10	Error colateral		
5	PA-GP94-Historial con medicamento II	Media		Raúl	OK	TAO00000617	07/11/12	15	Error colateral		
6	Verificar directamente desde CQ	Baja	TAO00000082	Raúl	OK		07/11/12	5			
5		Media	TAO00000094	Raúl	NOK	TAO00000094	07/11/12	5			
2	PA-VA55-Mensaje aviso falta historial	Baja		Raúl	NOK	TAO00000094	07/11/12	15			
3	PA-VA56-Mensaje aviso sin historial pac	Media		Raúl	NOK	TAO00000094	07/11/12	10			
3		Baja	TAO00000103	Raúl	OK		07/11/12	10			

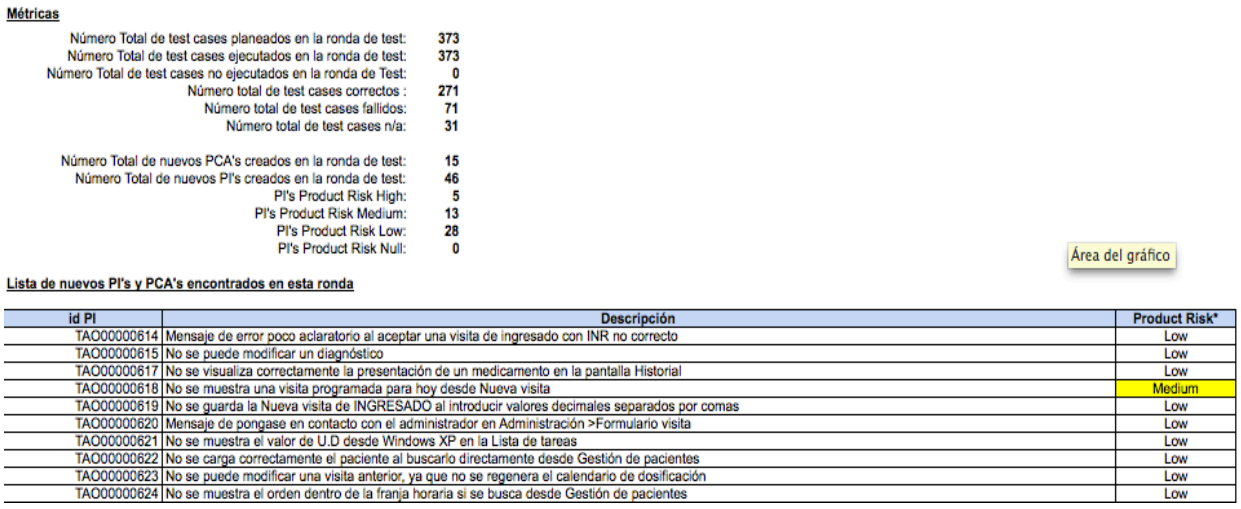
Figura 4.7: Matriz ejecución Test Report

Además, se deberán incluir las siguientes métricas:

- Número total de *test cases* planeados en la ronda de *Test*
- Número total de *test cases* ejecutados en la ronda de *Test*
- Número total de *test cases* no ejecutados en la ronda de *Test*
- Número total de *test cases* correctos
- Número total de *test cases* fallidos
- Número total de errores encontrados
- Número total de errores encontrados con Riesgo de Producto = *High*
- % *Bugs* con Riesgo de Producto = *High*
- % *Bugs* por módulo

- Eficiencia de corrección de errores (métrica definida más adelante en la sección de métricas)
- Eficiencia de los *test cases* (métrica definida más adelante en la sección de métricas)
- Tiempo de ejecución (tiempo total de ejecución de los *test cases*)
- Tiempo de ejecución/persona.

También se justificará debidamente cualquier cambio o modificación que haya podido acaecer en el *Test Plan* correspondiente a cada *build* (Figura 4.8).



Métricas

Número Total de test cases planeados en la ronda de test:	373
Número Total de test cases ejecutados en la ronda de test:	373
Número Total de test cases no ejecutados en la ronda de Test:	0
Número total de test cases correctos :	271
Número total de test cases fallidos:	71
Número total de test cases n/a:	31
Número Total de nuevos PCA's creados en la ronda de test:	15
Número Total de nuevos PI's creados en la ronda de test:	46
PI's Product Risk High:	5
PI's Product Risk Medium:	13
PI's Product Risk Low:	28
PI's Product Risk Null:	0

Lista de nuevos PI's y PCA's encontrados en esta ronda

id PI	Descripción	Product Risk*
TAO00000614	Mensaje de error poco aclaratorio al aceptar una visita de ingresado con INR no correcto	Low
TAO00000615	No se puede modificar un diagnóstico	Low
TAO00000617	No se visualiza correctamente la presentación de un medicamento en la pantalla Historial	Low
TAO00000618	No se muestra una visita programada para hoy desde Nueva visita	Medium
TAO00000619	No se guarda la Nueva visita de INGRESADO al introducir valores decimales separados por comas	Low
TAO00000620	Mensaje de pongase en contacto con el administrador en Administración >Formulario visita	Low
TAO00000621	No se muestra el valor de U.D desde Windows XP en la Lista de tareas	Low
TAO00000622	No se carga correctamente el paciente al buscarlo directamente desde Gestión de pacientes	Low
TAO00000623	No se puede modificar una visita anterior, ya que no se regenera el calendario de dosificación	Low
TAO00000624	No se muestra el orden dentro de la franja horaria si se busca desde Gestión de pacientes	Low

Figura 4.8: Métricas del Test Report

Todos los errores y anomalías detectados a través de la ejecución de *Test Cases* se reportarán usando la herramienta *Clear Quest* de *Rational*. El proceso a seguir para introducir un error de forma correcta en *Clear Quest* se detalla en el apartado 4.4.2.4

4.4.2.2 Diseño de Test Cases

El diseño de *Test Cases* es una actividad consistente en desarrollar el número de casos de prueba suficientes y necesarios para asegurar que una determinada área, módulo o componente del *software* se comporta de la forma correcta y esperada y que se atiene a los requisitos y especificaciones.

La gestión de los *Test Cases* se realiza utilizando el *software HPQC (HP Quality center)* (Figura 4.9). Este software facilita la implementación y almacenaje de los *test cases* con sus respectivos atributos, pasos de ejecución, notas y puntos de verificación.

Para cada producto *software* se creará un proyecto nuevo en *HPQC*. Dentro de este proyecto se creará una estructura de carpetas para cada módulo de la aplicación, dentro de las cuales estarán los *Test Cases* que prueban y verifican dicho módulo.

Figura 4.9: *Test plan* en *Test Manager* de *Rational*

Los *Test Cases* pueden y deben empezar a escribirse a medida que se aprueban los requerimientos y los *use cases* (Figura 4.10).

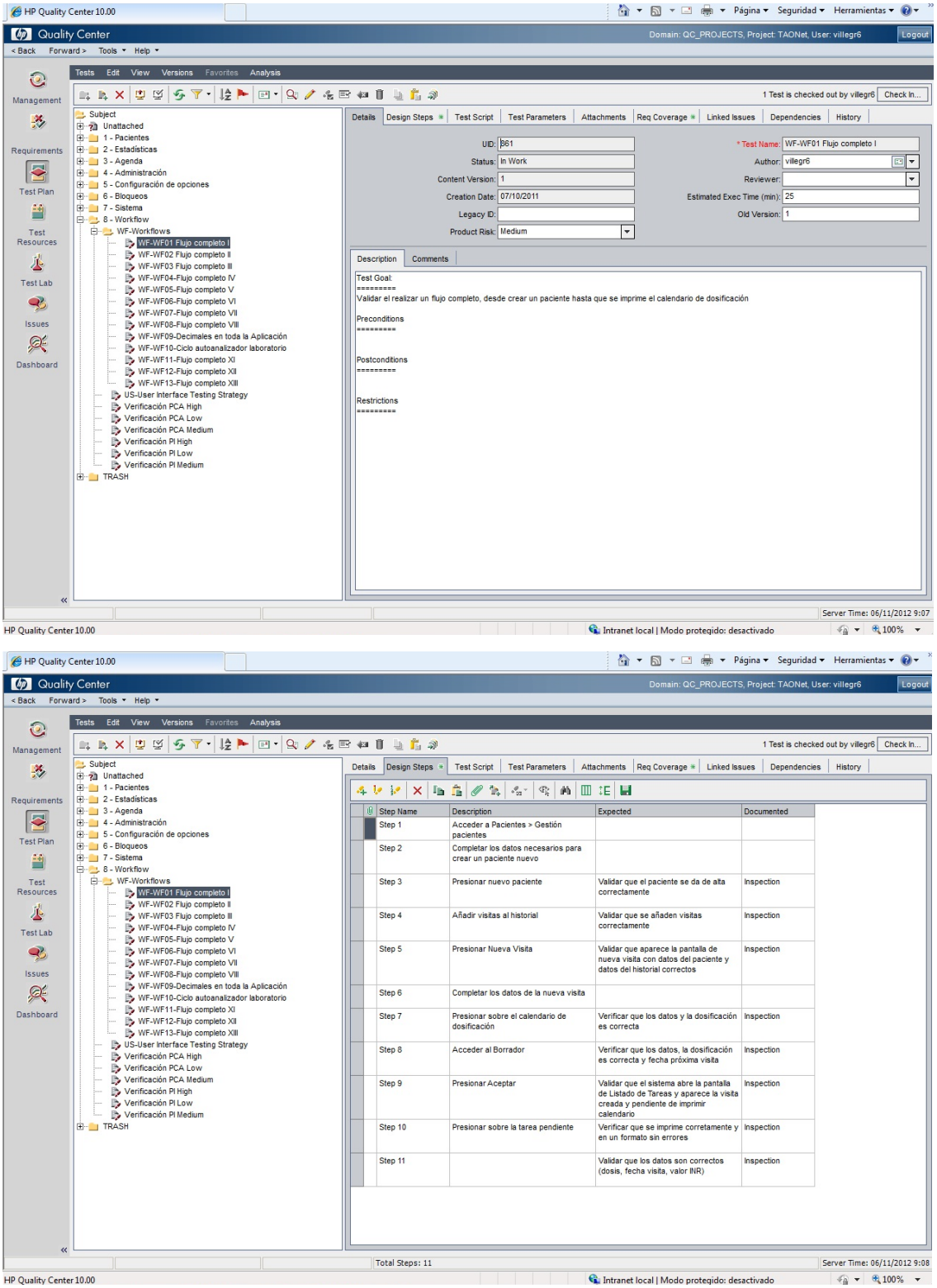


Figura 4.10: *Test Case* de un *Test Plan*

En nuestro caso los *Test Cases* que diseñamos, no sólo cubren el comportamiento esperado del requerimiento cuando el usuario utiliza la aplicación de forma correcta (pruebas positivas), sino que también se somete el *software* a pruebas que comprueban que la aplicación es robusta cuando el usuario usa la aplicación de un modo erróneo (pruebas negativas).

4.4.2.3 Ejecución de Test Cases

La ejecución de los *Test Cases* se desarrollará de la manera establecida en cada *test plan* particular.

Como norma general, antes de la liberar una versión, todos los *Test Cases* funcionales deben haber sido ejecutados al menos una vez, consiguiendo por tanto un 100% de cobertura funcional.

La ejecución de los *Test Cases* de los diferentes módulos se realiza en paralelo al desarrollo de las mismas, mediante rondas de validación informal. Es una manera eficaz de agilizar el proceso de validación formal ya que realizando rondas informales se evita que errores muy evidentes hagan que la versión se retrase. De esta manera se conseguirá una más rápida estabilización del *software* y librarlo de los errores más graves antes de empezar el proceso de validación formal. Durante las rondas informales de validación no será necesario utilizar el *Clear Quest*, los errores detectados se registrarán en un *Excel* compartido con el departamento de desarrollo para realizar el seguimiento y corrección del error. Este archivo se mantiene diariamente por el departamento de desarrollo.

En cada ronda de *test*, además de los módulos y peticiones de cambios a verificar, se deberán también consignar y verificar aquellos errores que se haya decidido solucionar. Estos errores deben aparecer claramente diferenciados en el *Test Plan* correspondiente a aquella ronda para facilitar su seguimiento.

Los errores pueden llevar asociados uno o varios *Test Cases* dependiendo del riesgo que tenga el error. En este caso, se deben ejecutar los *Test Cases* y comprobar a través de ello que el error ya no es reproducible, haciendo con esto una pequeña regresión respecto al error. Estos errores tienen asociados test cases debido a que es posible que al solucionar el error se vean afectados elementos relacionados o no y se produzcan nuevos errores directos o colaterales. Los errores que no tienen *test cases* relacionados se validarán con la descripción dada en el *Clear Quest* y haciendo un poco de *workaround* respecto al error.

Las peticiones de cambio se validarán a través de la ejecución de *Test Cases* especialmente diseñados para verificar la nueva funcionalidad. En las versiones de actualización, la verificación de la corrección de errores y la verificación de la implementación de las peticiones de cambio tendrá prioridad sobre los test de regresión. Se empezará la ejecución de los *test cases* empezando por aquellas áreas más críticas o que implementan los cambios/correcciones más importantes atendiendo al riesgo de producto y negocio.

Cada vez que el ***departamento de desarrollo*** libere un *build*, éste tiene que pasar un test de instalación llevado a cabo por el ***departamento de infraestructuras.*** Como parte de ese test de instalación el ***departamento de calidad***, ejecutará unos *Smoke Tests*, es decir, un conjunto de pruebas orientadas a verificar que la funcionalidad más básica de un módulo o programa funciona de forma correcta y según lo esperado. De esta manera se detectaría cualquier anomalía grave de forma inmediata sin esperar a empezar la ejecución formal de la validación del *build*.

Si el *build* no logra pasar con éxito las pruebas de instalación y/o *Smoke Test*, será devuelto al ***departamento de desarrollo*** para proceder a su corrección. Si por el contrario las pruebas son exitosas, se procederá a iniciar la validación funcional del mismo.

Aquellos módulos o funcionalidades de alto riesgo de producto deben también ser tenidos en cuenta en cada ronda de validación para ser verificados por test cases de regresión, y de forma muy especial en el *build* que se considere *Release Candidate*.

4.4.2.4 Clear Quest *(CQ)*

Existen diversos campos a rellenar en CQ a la hora de introducir un defecto o error en el mismo. Dichos campos aparecen en rojo de manera que el usuario detecte rápidamente cuáles son. En la primera pestaña (figura 4.11) es necesario indicar en qué versión encontramos el error, el módulo, componente y por último es necesario indicar el impacto en testing que tiene el error. Este impacto viene dado por la siguiente clasificación (Tabla 4.1):

Impacto en el proceso de testing	Valor
El error provoca que la ronda de test no se pueda ejecutar, por lo menos para aquellos *test cases* más importantes. El error por consiguiente impactaría de forma severa en el cliente, sin que exista un modo alternativo de evitarlo.	*High*
El error provoca que algunos de los *test cases* de un módulo no se puedan ejecutar. El error impactaría en el cliente, pero existe un modo alternativo de operación que permitiría evitar el error.	*Medium*
El error no tiene impacto significativo en el proceso de *test*. Implicaría un inconveniente menor para el cliente, aunque podría hacer uso normal del aplicativo.	*Low*

Tabla 4.1: Asignación de Impacto en el proceso de testing

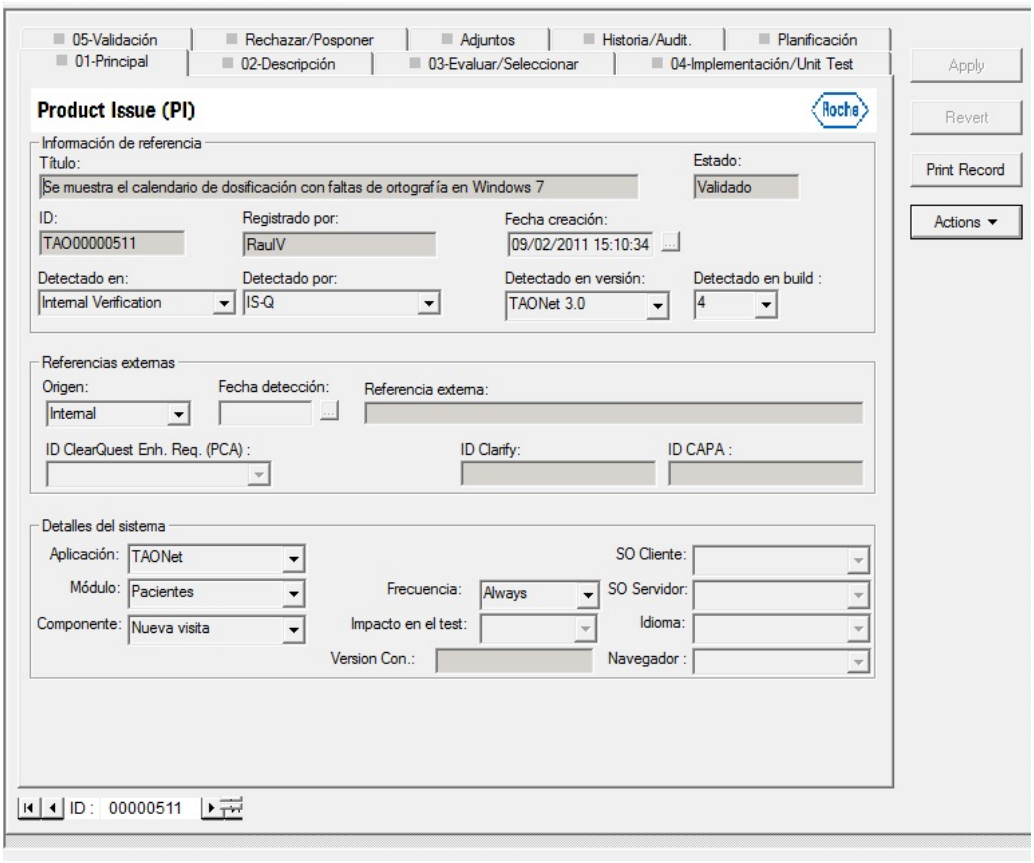


Figura 4.11: Introducción de un error en Clear Quest

En la siguiente pestaña, la pestaña 2, (Figura 4.11) se deberán rellenar los campos de *short description* y *description,* indicando el error concreto. El campo *short description*, debe contener la palabra clave para que en un vistazo el resto del equipo entienda el error. En el apartado de notas se puede introducir información adicional. Existe una pestaña para poder añadir archivos adjuntos si se desea.

Cada uno de los errores, una vez introducidos, se evaluará y asignará a una versión siguiendo el ciclo de vida de desarrollo descrito.

4.4.3- Desarrollo de la validación

4.4.3.1 Evaluación interna

Tal como se ha comentado TAONet 3.1 es una actualización de TAONet 3.0. Para la nueva versión de TAONet se plantearon efectuar incialemente dos Builds internas. Debido al elevado número de cambios efectuados en la versión, nuevas opciones en los accesos a pantallas y también a los muchos errores detectados durante la validación interna, finalmente ha sido necesario realizar tres Builds. Cada uno de ellos se ha validado en un periodo oscilante entre 7 y 14 días. Mencionar que el esfuerzo en día ha sido menor gracias a que se incorporó un nuevo tester al equipo y la ejecución de test case por parte de Team Leader.

Anterior a la entrega de la Build 1, se hizo una pequeña validación informal durante unos diez días. Dicha validación no sigue ningún orden en concreto ni registro

formal (si un documento Excel de control personal con los errores detectados). De esta manera, y mientras el departamento de desarrollo acaba de completar la Build 1, se va informando a los programadores de errores a solucionar antes de la entrega oficial de Build1.

No fue necesario realizar un *smoke test* de la versión, ni ejecutar test cases ya que era una validación informal.

***Build* 1**

Primera entrega oficial de Build 1, donde se intentar introducir todos los cambios realizados en esta versión y corregir el máximo de errores. De forma extraordinaria se ejecuta todos los TC's. Habitualmente, cuando se detectan un gran número de errores y hay nuevos cambios por implementar, se finalizada la validación sin llegar al 100%, dejando los TC's no ejectutados para la próxima Build. Además en esta caso se dejan errores pendientes por la conexión entre la aplicación y la hardware externo Cougucheck.

Build 2

Durante la segunta Build se validaron errores no corregidos y rehusados en la build 1. Además se validaron nuevos ítems que no habían entrado en la primera entrega, por no estar corregidos en la Build 1. Como se puede observar en la Test Report de la Build 2, se ejecutaron un total de 332 items e 392, lo que hace un 75% del total. Los motivos de no completar el 100% el test plan vienen provocados por la imposibilidad de validar la aplicación con la conexión a Couguchek y la gravedad de algunos errores. Por ese motivo se entregó la Build 3 sin estar completamente ejecutada la Build 2.

Build 3

Para terminar, y teniendo en cuenta que el *build* 3 era el *Release Candidate*, se validaron los ítems corregidos en este *build* y se ejecutaron como regresión todos los test cases de prioridad alta, además de realizar una regresión de los módulos más críticos de la aplicación. Cabe destacar que en cada build se ha ejecutado los smoke test, donde la validación de las funciones principales de la aplicación quedan cubiertas

De este modo se consigue un *build* final estable aunque no exento de errores.

En este *build* se ejecutaron el 100% de los ítems planeados.

Métricas sobre la versión

Se presenta una tabla con las métricas más importantes de cada uno de los *builds* de esta versión. No hay métricas sobre la validación informal ya que tal y como se ha comentado anteriormente, se realiza de forma no-oficial.

Cabe destacar que las cifras mostradas en la tabla pueden diferir de las que se muestran en los *test report* de cada *build* debido a errores registrados posteriormente al cierre de la edición de cada *build*.

	***Build*1**	***Build*2**	***Build* 3**
Errores corregidos y validados	92	75	25
Errores reabiertos	12	15	2
Nuevos errores detectados	46	25	10
Nuevos errores con riesgo de producto *High*	5	1	1

Tabla 4.2: Métricas sobre la validación

De *build* a *build* no se arreglan todos los errores detectados en los *builds* anteriores. Se priorizan y si no tienen un gran riesgo o son muy complicados de arreglar, se planifican para versiones o *builds* de la misma versión.

Se finalizó la *build* 3 con errores y quizá se caiga en el error de pensar que la versión sale con errores, y es así, sale con errores porque el *software* no es perfecto porque las personas no somos perfectas, pero no sale con errores graves para el paciente ni para el negocio. La gran mayoría de errores son errores que el usuario quizá nunca encuentre porque para poder reproducirlos se deben dar muchos factores que en un hospital/laboratorio es difícil que ocurran. De todas maneras todos los errores que restan por arreglar se planifican para una próxima versión y serán arreglados en el futuro.

*****Nota:*** *Todos los documentos, Test plan, Test report, Requerimientos, matriz de trazabilidad… se encuentran en la carpeta TAONet (adjunta a la memoria).*

4.4.3.2 Evaluación externa

4.4.3.2.1 Plan de evaluación externa

En los productos nuevos, tras la firma del *Milestone* 3 (MS3) al finalizar la fase de DO (*Design Output*), y por tanto con el desarrollo y la validación interna acabados, se debe efectuar una evaluación formal del producto en un laboratorio u hospital.

Para ello, el centro deberá disponer de un entorno de *test*, duplicando el que usan en sus tareas diarias. El *software* se instalará en este entorno y por tanto podrá ser alimentado con datos reales provenientes del centro, permitiendo una evaluación similar a la que el producto tendría si se dejase en producción; sin que ello incurra en el riesgo de que los datos puedan verse perturbados por errores potenciales que podrían significar un impacto en la seguridad.

El objetivo de estas pruebas es múltiple:

a) Detección de errores que son difíciles de reproducir internamente y que son debidos a usos particulares u opciones de configuración utilizadas por los clientes y que no hayan sido tenidas en cuenta en los *test* funcionales.

b) Detectar cambios o mejoras que interesan a los clientes y que todavía pueden introducirse en la aplicación antes del lanzamiento final.

c) Evaluar el rendimiento de la aplicación en un entorno real.

La evaluación externa deberá ser realizada por técnicos de *Pharma S.L*, con el soporte de personal de los departamentos de Desarrollo, Calidad e Infraestructuras. Idealmente, el cliente o un representante del mismo deberían participar en el proceso, a fin de captar la impresión del usuario final.

La evaluación externa también es requerida para las versiones de actualización de un producto.

Para realizar la evaluación externa se desarrollará un Plan de Evaluación Externa, creado en base a la plantilla Plan de Evaluación Externa. Este plan deberá constar de dos tipos de pruebas:

- **Evaluación guiada:** Selección de *Test Cases* desarrollados por el equipo de Calidad como parte de la validación funcional, que serán ejecutados esta vez en un entorno real. Se procurará hacer una selección tal que todos los módulos de la aplicación, o por lo menos los más importantes desde el punto de vista del usuario, sean evaluados.
- **Evaluación no guiada:** Pruebas que, sin formar parte de ningún *test case*, se puedan llevar a cabo en el centro, siguiendo el flujo específico de trabajo del centro en cuestión.

El resultado de la evaluación externa se registra en el documento al efecto, creado en base a la plantilla Informe de Evaluación Externa*.* En esta documento se reflejan los problemas que se haya tenido al ejecutar los *test cases* y otros problema relacionado con los escenarios o cualquier imprevisto. Hay que tener en cuenta que las pruebas se realizan en paralelo al trabajo del laboratorio y que para nada se puede intervenir en el trabajo diario del laboratorio, siempre se deberán de realizar las pruebas acorde con lo que el centro permita y pueda ofrecernos.

4.4.3.2.1 Pruebas de Evaluación Externa

Se ha llevado a cabo una única validación externa en Hospital El Bierzo (Ponferrada).

Se ha elegido este centro que actualmente trabaja con la versión de TAONet 3.0 y actualizará la aplicación a la 3.1. Además este centro nos proporciona la posibilidad de poder testear un Autoanalizador Siemens BCS y una integración con una gestión de agendas externa a TAONet. De esta manera se prueba en un entorno que no se ha probado anteriormente ya que con los escenarios que se disponemos, es prácticamente imposible simular la este entorno en concreto. La validación consistirá en "trabajar" en un entorno real, con un número de peticiones elevado y con una conectividad a varios dispositivos de laboratorio.

Las conclusiones que se pueden sacar sobre la evaluación externa son:

- Se realizaron más pruebas de las estipuladas cosa que fue beneficioso para la puesta en marcha.
- Los usuarios valoraron muy positivamente el cambio realizado, alegando que era más rápido que el sistema antiguo sobre todo con la conexión con los aparatos y mucho más completo que la aplicación anterior.

- Los usuarios nos hicieron llegar quejas pero todas ellas eran referentes a la falta de configuración.
- La valoración de los usuarios fue un 9 sobre 10, muy buena puntuación para una puesta en marcha.

4.5 Liberación de versión

Como se ha explicado al inicio se planificaron 3 *builds*. Esto ha hecho que la validación de la versión finalmente se cerrara el día 31 Diciembre. La fecha de inicio de la validación se retraso 15 días por motivos ajenos al departamento de calidad.

Este pequeño retraso aun que pueda parecer grave es algo bastante normal, porque aun que se planifiquen las cosas siempre hay imprevistos, nuevas implementaciones y errores importantes a solucionar. Finalmente la versión se liberará el día 2 de enero de 2013.

5- CONCLUSIONES

5.1 Objetivos conseguidos y no conseguidos

El objetivo principal era conseguir un sistema de información para laboratorios mediante un entorno *web* seguro y fiable en las fechas propuestas. Se ha conseguido lanzar al mercado el producto deseado, es cierto que el plazo se alargado un poco más de lo necesario debido al retraso en la entrega de la Build 1 por parte del equipo de desarrollo. Pero de esta manera se ha consigo validar un gran número de nuevas implementaciones y corregir nuevos errores.

Sin tener en cuenta el pequeños retraso del principio, los posteriores plazos se han ido cumpliendo de forma exacta.

5.2 Mejoras propuestas

Con el proyecto presentando queda reflejado la importancia que toma realizar una validación sobre los productos de *software*. Hay que tener en cuenta que cualquier validación tampoco sirve ya que, como vemos en el proyecto, los desarrolladores pasan un *Unit Test* y aún así hay errores de conjunto de código que no se detectan. Es cierto que al lanzar la versión se han dejado errores por corregir, pero ninguno de ellos es grave para el paciente ni para el negocio. Estos errores se corregirán en versiones futuras del programa.

Durante el trascurso del proyecto ha habido pocos problemas, pero probablemente el de mayor importancia ha sido el retraso en la entrega de la Build 1. Desde mi punto de vista, intentaría mejorar la planificación y los plazos de entrega. Si el equipo de desarrollo se retrasa en una entrega, afecta directamente al trabajo de los tester, que tenemos que realizar la validación en menor tiempo y este morivo conlleva la posibilidad de no detectar errores importantes por las "prisas" y la imposibilidad de retrasar fechas.

Una mejora, relacionada con la anterior, muy importante sería realizar una planificación que contase con los retrasos causados por las nuevas implementaciones (que harán que se introduzcan nuevos errores en el código). Los nuevos errores hacen que se deban tocar muchas partes del código que harán que de nuevo se puedan introducir nuevos errores. Otra mejora sería dejar un tiempo de margen en el lanzamiento del producto ya que los validadores intentamos agilizar el tiempo de lanzamiento pero si encontramos errores importantes debemos asegurar que se solucionen por muy retrasado que vaya el proyecto. La última mejora propuesta sería automatizar pruebas. Consiste en programar pruebas que se ejecuten automáticamente y que finalmente nos muestre un resultado. Este tipo de pruebas ahorrarían tiempo ya que se ejecutarían de forma automática mientras se realiza la validación manual y serían pruebas que no las deberían de ejecutar los testers.

5.3 Valoración personal

Personalmente he alcanzado mi objetivo, que era aprender el proceso de lanzamiento de un proyecto de gran envergadura desde su inicio hasta su fin. Durante mi corta experiencia en el sector, únicamente había visto la parte de la validación que me afectaba directamente, pero con la realización del proyecto he podido entender todo el proceso desde el inicio hasta la salida al mercado. Al tratarse de un producto de ámbito hospitalario, hace que la validación sea más detallada y minuciosa ya que el cliente final son personas con problemas de la salud.

Además he entendido la necesidad de generar tanta documentación para realizar cualquier cambio. Es verdad que sigo pensando que tanto protocolos, documentos, firmas, etc causan que el proceso sea más lento pero a la vez con un control enorme por el tipo de producto que es.

En el ámbito más personal del proyecto, me ha gustado mucho al interacción con el departamento de desarrollo. Normalmente hay momentos de tensión, ya que para ellos somos “”los malos” que buscan sus errores en el código, pero la relación y la organización con ellos ha sido excelente.

Hay un apartado que no he podido ampliar ni estudiar como tenía pensado. Es el apartado de Automatización de las pruebas. La intención era realizar un estudio de una posible automatización de algunas de pruebas, pero ha sido imposible realizar ya que finalmente la validación se ha ajustado demasiado al tiempo de entrega.

Aclarar que por motivos ajenos a mi, no he podido utilizar nombres, documentos, plantillas oficiales de la empresa donde he realizado el proyecto. Inicialmente se acepto la propuesta de colaboración con la universidad, pero finalmente se rechazó. Por ese motivo aparecen nombres inventados aunque los datos que se muestran son reales de una validación actual.

GLOSARIO DE TÉRMINOS CLAVE

Término	Definición
Bug	Error del programa de *software*. Comportamiento no deseado o no ajustado a las especificaciones.
Build	Cada una de las versiones internas de *software* que se entregan al departamento de calidad para que este proceda a su verificación.
Cambio / Petición de mejora (*Product Change*).	Nueva funcionalidad añadida al producto que no fue incluida en las primeras especificaciones y diseños del producto.
Cobertura	Métrica usada para cuantificar la cantidad de pruebas que se realizan sobre un componente *software*. Mide cual es la completitud del testing cuantificándolo como el porcentaje de requerimientos o módulos de una aplicación que son verificados con respecto al total de los mismos.
Debugar	Actividad de desarrollo que identifica la causa del defecto, repara el código y verifica que el defecto ha sido corregido correctamente.
Especificaciones de los Requerimientos del *Software*	Documentación donde se recogen los requerimientos esenciales (funciones, atributos, etc.) del *software* y su interoperabilidad con otros.
Host	Sistema informático externo a TAONet 3.1 y con el que intercambia datos de diversa índole.
LIS	*Laboratory Information System* o Sistema de Información de Laboratorio. Software de soporte a las actividades realizadas en un laboratorio clínico.
Modelo en V	Modelo de ciclo de vida de desarrollo de *software* orientado al testing, de manera que cada actividad de diseño y/o desarrollo lleva aparejada una actividad de verificación desde las fases más tempranas.
Operational Usage Test Case	*Test Cases* diseñados desde la óptica de un usuario final, de manera que intentan reproducir y ejecutar el flujo habitual de acciones y eventos que tienen lugar en un entorno de laboratorio. Verifican la integración en conjunto de varios módulos o funcionalidades de la aplicación.
PI	*Product Issue*. Un error o *bug* del programa.
Pruebas de Regresión	Selección de pruebas para una ronda de test que se ejecutan o reejecutan para verificar que las modificaciones hechas en algunos componentes no han causado efectos no deseados en otros.
Release candidate	*Build* que por su estabilidad y calidad se prevé que será el definitivo y que podrá ser liberado tras su validación, sin necesidad de generar otra versión.
Riesgo de negocio	Riesgo asignado a cada requerimiento, error o petición de cambio dependiendo de su importancia en términos de marketing, costes, beneficios y satisfacción del cliente.

Término	Definición
Riesgo de producto	Riesgo asignado a cada requerimiento, error o petición de cambio en función de su peligro potencial para la salud del paciente o usuario.
Smoke Test	Conjunto de pruebas orientadas a verificar que la funcionalidad más básica de un módulo o programa se comporta de forma correcta y según lo esperado.
Test Case / Caso de Prueba	Conjunto de pasos y puntos de verificación que se ejecutan de forma secuencial sobre el *software* para verificar que se cumplen los requerimientos y que la aplicación funciona de la forma esperada. Se comprueba con ellos el comportamiento de la aplicación tomando como base el comportamiento esperado descrito en los requisitos.
Test de instalación	Conjunto de pruebas realizadas para verificar que el sistema se puede instalar, actualizar, reinstalar y desinstalar de forma correcta.
Test de regresión	Re ejecución de pruebas que ya habían sido ejecutadas en un módulo con el fin de verificar que las modificaciones desarrolladas en otro módulo o componente no hayan causado efectos no deseados en aquel.
Test de rendimiento	Pruebas orientadas a evaluar los requerimientos de rendimiento de un sistema, entendidos como los tiempos de respuesta de la aplicación ante los eventos y acciones del usuario.
Test de seguridad	Pruebas orientadas a verificar los requerimientos de seguridad del sistema, centrados en impedir el uso malintencionado del *software* por usuarios no autorizados.
Test functional	Test orientado a la verificación de los requerimientos funcionales de una aplicación, utilizándola y usándola desde un punto de vista similar al que realizaría un usuario.
Test plan	Documento que describe el proceso de testing para una *release* en particular. En él se especifican y detallan las actividades de validación que se llevarán a cabo en la *release* en cuestión. En definitiva, es una ampliación del *Validation Plan*.
Test Report / Informe de pruebas	Documento que describe los resultados de la ejecución de las pruebas sobre el *software*.
Test Unitario	Conjunto de pruebas desarrollado y ejecutado por el departamento de desarrollo con el objetivo de validar el correcto funcionamiento de una pieza de código. En el caso de TAONet 3.0 el test unitario se ha realizado sobre cada una de las pantallas, considerándose estas como la porción de código más pequeña a partir de la cual se realiza actividad de validación.
Testing	Conjunto de pruebas que pueden mostrar fallos que son causados por los errores

Término	Definición
Testing estructural o de caja blanca	Técnica para realizar pruebas sobre la estructura interna del sistema (código). Es necesario un conocimiento explícito del código y la arquitectura del mismo. Esta técnica es aplicada por los desarrolladores.
Testing basado en modelos o de caja negra	Técnica de testing que se basa en el análisis de la especificación del componente o *software* bajo testeo sin hacer referencia a su funcionamiento interno. Es equivalente a Testing funcional.
Validación	El proceso de determinar el grado de corrección de los productos *software* con respecto a las necesidades del usuario. Se respondería a la pregunta: "¿Hemos desarrollado el sistema correcto?"
Verificación	El proceso de evaluar un sistema para determinar si se ajusta a los requerimientos y especificaciones que sobre él se han hecho antes de comenzar su diseño. Se respondería a la pregunta: "¿Hemos desarrollado el sistema correctamente?"

REFERENCIAS BIBLIOGRÁFICAS

[1] Curso sobre testing, *ISEB Course* www.softwaretesters.co.uk,

[2] Glosario de estándares del testing, *The BCS SIGIST Standard glossary of Testing Terms* (*British Standard* BS 7925-1).

[3] Documentación sobre testing
http://www.ia.uned.es/ia/asignaturas/adms/GuiaDidADMS/node10.html

[5] Documentación sobre procedimientos de *Pharma S.L* y procedimientos de los estándares seguidos:

Organismo	Link	Descripción
ANSI	http://www.ansi.org	ANSI es una organización privada sin fines de lucro, que permite la estandarización de productos, servicios, procesos, sistemas y personal en Estados Unidos. Además, ANSI se coordina con estándares internacionales para asegurar que los productos estadounidenses puedan ser usados a nivel mundial.
FDA	http://www.fda.gov	FDA es una agencia de *US Department of Health and Human Services* responsable de la protección de la sanidad pública para asegurar la seguridad, eficacia, seguridad humana y drogas veterinarias, productos biológicos, dispositivos médicos, cosméticos y productos que emiten radiación, y para regular la manufactura, *marketing* y distribución de productos de tabaco.
IEEE	http://www.ieee.org	IEEE es la más grande asociación de profesionales del mundo dedicada al avance de la innovación tecnológica y excelencia en beneficio de la humanidad. IEEE y sus miembros inspiran una gran comunidad global a través de las publicaciones más citadas del IEEE, conferencias, estándares de tecnología, profesionales y actividades educativas.
ISO	http://www.iso.org	ISO (Organización Internacional para los Estándares) es el mayor desarrollador mundial y editor de Normas Internacionales. ISO es una red de los institutos de normas nacionales de 163 países, un miembro por país, con una Secretaría Central en Ginebra, Suiza, que coordina el sistema.
ISPE	http://www.ispe.org	ISPE, la Sociedad Internacional de la Ingeniería Farmacéutica, es la asociación más grande del mundo sin fines de lucro dedicada a educar y promover a los profesionales farmacéuticos de fabricación y su industria. Fundada en 1980, ISPE hoy sirve de 25.000

		miembros en 90 países.
PDA	http://www.pda.org/	La Asociación parental de drogas (PDA) es el proveedor líder mundial de la ciencia, la tecnología y la información reglamentaria y la educación para la comunidad farmacéutica y biofarmacéutica. Fundada en 1946 como una organización sin fines de lucro, el PDA se ha comprometido a desarrollar información sólida, práctica, técnicas y recursos para promover la ciencia y la regulación a través de la experiencia de sus más de 95,00 miembros en todo el mundo.

[6] Definiciones básicas : www.fceia.unr.edu.ar/ingsoft/**testing**-intro-a.pdf, Autor: Maximiliano Cristiá.

[7] Conceptos sobre los ciclos de vida: http://www.calidadysoftware.com/testing/introduccion_software_testing.php , Autor: Ing. Alexander Oré B.

ÍNDICE DE FIGURAS Y TABLAS