Gestión de Requisitos dirigida por Pruebas de Aceptación

Autor: Álvaro Muñoz Pérez
Director: Dr. Patricio Letelier Torres

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Ingeniería del Software y Sistemas de Información

Diciembre 2010

Tesis presentada para cumplir con los requisitos finales para la obtención del título de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información, de la Universidad Politécnica de Valencia, 2010.

Agradecimientos

Sobretodo querría dar las gracias a mis padres, por haberme subvencionado el máster y todos mis estudios, por la educación que me han dado y haberme apoyado y creído en mí siempre y en especial durante los últimos 2 años con la realización de la tesis.

Al Doctor Patricio Letelier, mi director de la tesis, por su dedicación, confianza y la paciencia que ha tenido conmigo y especialmente por la cantidad de conocimientos que me ha aportado.

A mi amigo Jaime Rodríguez, por haberme echado una mano en mis carencias de diseño.

A mis compañeros del DSIC, por toda la ayuda que me han prestado, su apoyo desinteresado y por los conocimientos que he adquirido de ellos. Mención especial a Juan Vicente Pérez, María Isabel Marante, Carlos del Fresno.

Dedicatoria

Dedico la elaboración de esta tesis a mis padres por haberme dado todo y a Lidia, por haber soportado que pasara tantas horas en la Universidad.

Resumen

Es bien sabido que el proceso de desarrollo de software necesita todavía alguna reforma para aumentar la tasa de éxitos en su producción. Históricamente se han tocado todas las actividades del desarrollo del software (Análisis, Diseño, Codificación) añadiendo nuevas técnicas y herramientas para mejorar éstas. Sin embargo, hay una actividad que desde los inicios del software no ha cambiado significativamente: La fase de especificación de requisitos.

En esta tesis se propone un nuevo enfoque llamado “Test-Driven Requirement Engineering”. Además, se diseña e implementa un módulo llamado: “Gestor de Requisitos”, el cual sirve de soporte para este nuevo enfoque. Este Gestor de Requisitos provee de la plataforma base para la gestión de todos los requisitos definidos en un proyecto de desarrollo software, según se indica en el enfoque del TDRE.

Sumario

Lista de figuras e ilustraciones

Lista de tablas

Capítulo 1. Introducción

1.1 Motivación y objetivos de la Tesis

(El siguiente capítulo ha sido extraído de [1]*)*

TDD (Test-Driven Development) [2] es un enfoque basado en la idea de que las pruebas deben dirigir la forma en que se desarrolla un producto software. Este enfoque mayormente se ha aplicado en el ámbito de la implementación, particularmente siguiendo el planteamiento "*no escribir código hasta disponer de las pruebas que debe satisfacer dicho código*". El TDD ha tenido un fuerte impulso con las metodologías ágiles, las cuales lo incorporan entre sus prácticas esenciales.

Por otra parte, la ingeniería de requisitos, especialmente en el ámbito de productos software industriales, sigue basándose en la especificación de requisitos usando lenguaje natural, con los ya conocidos inconvenientes relativos a ambigüedad, legibilidad, completitud, etc. Sin embargo, la necesidad de validación puede más que las ventajas que ofrece una especificación más formal y rigurosa de los requisitos. El cliente debe poder leer y comprender los requisitos para así poder dar su conformidad respecto de ellos.

Las técnicas más populares para especificación de requisitos se resumen en Casos de Uso (utilizados en metodologías tradicionales, como RUP [3]) e Historias de Usuario (fichas de XP [4] o representaciones similares en otras metodologías ágiles). Si bien los elementos Actor (o tipo de usuario) y requisitos (Caso de Uso o Historia de Usuario) pueden visualizarse y manipularse gráficamente o en fichas, la descripción de cada requisito asociado es en lenguaje natural.

Es indudable que el acuerdo/contrato con el cliente y la planificación del producto deben estar basados en requisitos incorporados en el producto. Los requisitos son el objetivo a conseguir, es lo que espera el cliente que el producto software satisfaga. En este sentido podemos afirmar que todo proyecto de desarrollo de software está dirigido por los requisitos (Requirements-Driven Development). Esto es obvio y se ve reflejado en todas las metodologías.

El modelo V [5] para pruebas del software establece diferentes niveles de pruebas, cada uno de ellos asociado a un nivel de abstracción del software. Desde requisitos hasta la implementación de unidades de código, y desde pruebas unitarias hasta pruebas de aceptación.

Esto se muestra en la siguiente Figura 1 (Página siguiente).

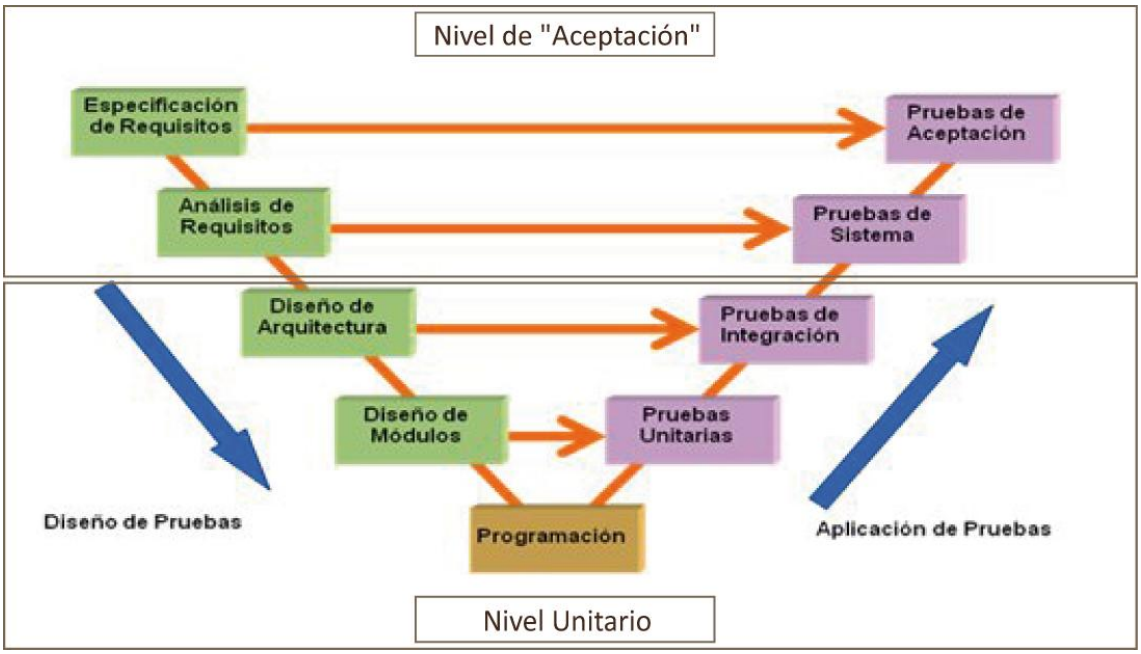


Figura 1. Modelo V para pruebas del software

La mayoría de las metodologías se basan en este modelo de pruebas, sin embargo, aquellas que apuestan decididamente por modelos de proceso iterativos e incrementales (RUP, entre las metodologías tradicionales, y todas las metodologías ágiles) utilizan de forma más explícita las pruebas de aceptación como criterio de éxito de cada iteración. Esto nos llevaría a clasificar estas metodologías como Test-Driven, en un contexto más de planificación y desarrollo global. Sin embargo, no existe una analogía entre el Test-Driven del ámbito de implementación ("*no escribir una línea de código antes de tener establecidas la pruebas unitarias*"), correspondiente a la parte inferior de la Figura 1, y lo que sería Test-Driven en el ámbito de la planificación, lo cual corresponde al recuadro superior de la misma. En el ámbito de los requisitos y la planificación el planteamiento sería "*no incorporar un requisito a una iteración (o al plan en general) sin haber antes establecido sus pruebas de aceptación".*

En las metodologías populares (tradicionales o ágiles), los conceptos Requisito y Prueba de Aceptación son tratados de forma independiente y abordados en diferentes momentos del desarrollo. Una vez establecidos los requisitos, se definen pruebas de aceptación para ellos. Es considerable la duplicidad de trabajo que ocasiona esta separación de artefactos y actividades asociadas a, por una parte, establecer Requisitos y por otra, Pruebas de Aceptación. Esto es debido a que muchos detalles de la especificación de requisitos provienen de la identificación de pruebas de aceptación específicas. Además, usualmente están involucrados dos roles también diferentes: Ingeniero de Requisitos (Analista o Analista Funcional) y Tester (más bien un tester especializado en pruebas de aceptación).

Nuestra propuesta de Test-Driven Requirements Engineering - TDRE (por referirnos a TDD en el ámbito de los requisitos y la planificación) unifica los artefactos, actividades y roles asociados a la especificación y validación, tanto de los Requisitos como de las Pruebas de Aceptación. El concepto de Requisitos se reduce a un contenedor de

Pruebas de Aceptación, y son estas últimas las que adquieren protagonismo como especificación de cada requisito. Una de las ocho "*buenas características*" que recomienda la IEEE 830 [6] para una especificación de requisitos se refiere a que cada requisito debe ser Verificable. En nuestra propuesta los requisitos son verificables pues están especificados por Pruebas de Aceptación.

Si bien, como especificación de requisitos, las Pruebas de Aceptación pueden sufrir los mismos inconvenientes que los requisitos tradicionales (al usar lenguaje natural para especificarlas), una Prueba de Aceptación puede acotarse y hacerse independiente, de forma que establezca una unidad manejable que determine de forma precisa una parte del comportamiento del sistema. Así, otras características recomendadas por el estándar IEEE 830 (Corrección, No ambigüedad, Completitud, Consistencia, Orden por importancia o estabilidad, Modificabilidad y Trazabilidad) también se ven favorecidas.

A continuación, definiremos el concepto de Prueba de Aceptación [7] y explicaremos cómo un requisito puede ser especificado mediante un conjunto de Pruebas de Aceptación.

Si bien los requisitos y las actividades asociadas han ido haciéndose espacio entre los aspectos esenciales de un proyecto de desarrollo de software, con las pruebas no ha ocurrido lo mismo y sólo equipos de desarrollo con cierta madurez las integran como pieza imprescindible en sus procesos de desarrollo. Existen dificultades para la introducción de una "*cultura*", disciplina y prácticas de pruebas en un equipo de desarrollo. Entre los obstáculos o malas estrategias de introducción podemos destacar: carencia de un proceso de desarrollo que integre las actividades de pruebas con el resto de actividades, sobrevaloración de la automatización de las pruebas como objetivo inmediato (o único), No "*Rentabilización*" del esfuerzo invertido en pruebas.
Nuestro enfoque sigue una estrategia de implantación de una "*cultura*" de pruebas a partir del aprovechamiento de las Pruebas de Aceptación ya desde su definición.

Podemos definir Pruebas de Aceptación (PA), con la siguiente afirmación: "*Una PA tiene como propósito demostrar al cliente el cumplimiento de un requisito del software*". Precisando un poco más, una PA:

- Describe un escenario (secuencia de pasos) de ejecución o uso del sistema desde la perspectiva del cliente.

- Puede estar asociada a un requisito funcional o también a un requisito no funcional.

- Un requisito tiene una o más PAs asociadas.

- Las PAs cubren desde escenarios típicos/frecuentes hasta los más excepcionales.

- Una PA puede tener infinitas instanciaciones (ejecuciones con valores concretos). El diseño de las instanciaciones y su aplicación es trabajo del tester.

Adicional a su propósito fundamental de especificación y validación de los requisitos, las PAs pueden rentabilizarse usándose para:

- Valorar adecuadamente el esfuerzo asociado a la incorporación de un requisito. Es más sencillo valorar el esfuerzo que requiere la satisfacción de cada PA.

- Negociar con el cliente el alcance del sistema en cada iteración de desarrollo. Las PAs introducen un nivel de granularidad útil para negociación de funcionalidad con el cliente. Un requisito no necesita implementarse "*todo o nada*", puede hacerse de forma incremental postergando la satisfacción de ciertas PAs.

- Guiar a los desarrolladores en la implementación ordenada del comportamiento asociado a un requisito. Si bien las PAs no son tan exhaustivas como las pruebas unitarias, sí que son parte de ellas (generan gran parte de la cobertura del código). Esto no significa necesariamente prescindir del resto de niveles de pruebas del Modelo V, sino que en un contexto de recursos limitados, las PAs deberían ser las pruebas esenciales o mínimas que deberían definirse y aplicarse al sistema.

- Identificar oportunidades de reutilización. El detalle y precisión proporcionado por las PAs permitiría identificar en el nivel de especificación de requisitos posibles solapes de comportamiento que den origen a oportunidades de reutilización de comportamiento (lógica de la aplicación).

Pero, popularmente, ¿cómo se especifican los requisitos?

- Narrativamente
- UML (Diagramas de Casos de Uso y otros diagramas)
- Plantillas o fichas
- Interfaces de usuario (bocetos)
- Combinación de los anteriores

Ejemplo: consideremos el requisito "*Retirar dinero*" en el contexto de un cajero automático. Consideremos una especificación narrativa como la siguiente: "*El cliente debe poder retirar dinero del cajero en cantidades seleccionables. Siempre recibe un comprobante, a menos que el cajero se quede sin papel. Cuando se trata de un cliente preferencial puede retirar más dinero del que tiene en su cuenta, pero se le debe advertir que se le cobrarán intereses. El cliente debería poder cancelar en cualquier momento antes de confirmar el retiro de dinero. Las cantidades deberían poder servirse*

con los billetes que en ese momento tenga el cajero y no se deberían aceptar otros montos. Sería conveniente avisar cuando el cajero esté realizando operaciones internas mostrando un mensaje. El dinero retirado de la cuenta debe poder comprobarse en los registros de movimientos de la cuenta…".

La Figura 2 ilustra algunas alternativas de especificación para este requisito (incluyendo la anterior descripción narrativa como opción por defecto). Los iconos reflejan la conveniencia de cada alternativa de especificación.

Figura 2. Alternativas populares para la especificación de requisitos

Elaborar un Diagrama de Secuencia para definir cada escenario de ejecución del requisito puede parecer interesante, sin embargo, en general no resulta apropiado por la gran cantidad de diagramas generados siendo que además, por tratarse de requisitos, deberían ser extremadamente sencillos. Resulta más interesante la identificación de los escenarios que su ilustración en un diagrama. La Descripción Narrativa no es descartable, al menos para dar una breve definición del requisito, especialmente centrándose en definir los conceptos involucrados (con la idea de un glosario o de un sencillo Modelo de Dominio).

Un modelo de Casos de Uso puede ser una buena opción, especialmente para organizar y visualizar los requisitos de un sistema nuevo. Sin embargo, un Modelo de Casos de Uso no es apropiado para ilustrar la arquitectura detallada de un producto software (desde la perspectiva de requisitos) en situaciones de mantenimiento a largo plazo. Un producto software de tamaño medio puede tener miles de requisitos o unidades funcionales (o no funcionales), lo cual no es adecuado representarlo en Diagramas de Casos de Uso. La visualización y gestión de gran cantidad de requisitos requiere más bien de mecanismos tipo árbol multinivel o grafo.

Los bocetos (visualizaciones muy preliminares) de la IU son siempre bienvenidos pues son una herramienta efectiva de comunicación y validación con el cliente, el cual debe visualizar el resultado final. En este contexto de requisitos no debe pretenderse realizar el diseño de las IUs sino más bien paneles con cierto ámbito de datos, sin profundizar en tipos de controles de interfaz o cuestiones de estética de formularios/páginas.

Las Plantillas son una de las alternativas de especificación más usadas para Casos de Uso. Las plantillas son elegantes y proporcionan una sensación de orden en la especificación. Sin embargo, en general resultan contraproducentes ya que tienden a dar un tratamiento uniforme en cuanto a nivel de detalle para todos los requisitos. En aquellos muy simples se tiende a incluir cosas obvias o irrelevantes sólo para poder cubrir todos los apartados de la plantilla. Cuando un requisito incluye varios (o muchos) escenarios, el intento por sintetizar todos los escenarios en una plantilla (que sólo ofrece pasos y excepciones) lleva normalmente a especificaciones enrevesadas.

Según lo anterior, nuestro enfoque TDRE apuesta por especificar los requisitos usando los siguientes elementos (ilustrados en la Figura 3).

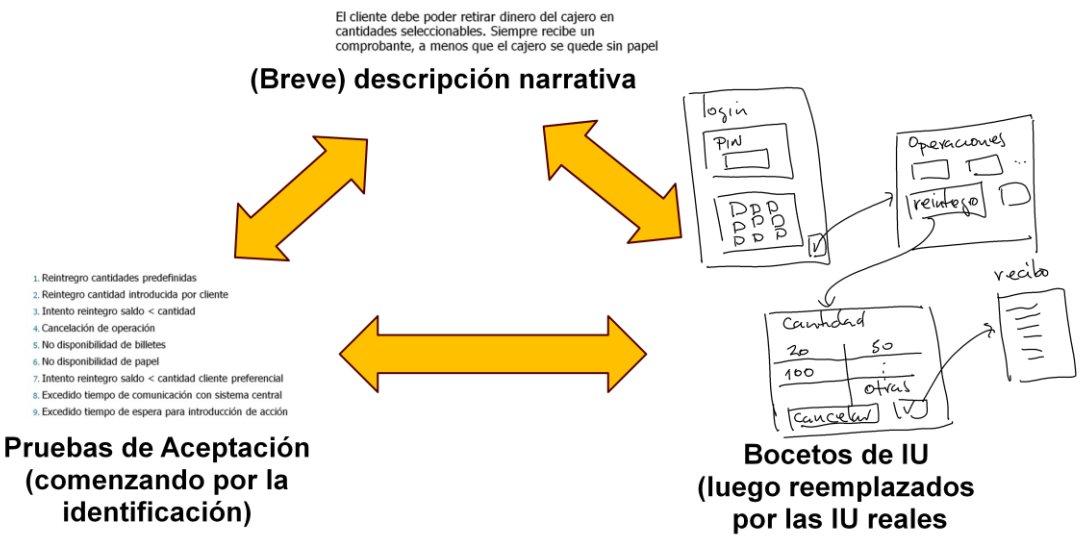


Figura 3. Trilogía para la especificación de requisitos

- Un grafo dirigido para la representación de la arquitectura de requisitos. Cada nodo es un requisito funcional o no funcional. Los arcos desde un nodo a otros nodos establecen una relación de nodo padre y nodo hijo, mediante la cual se hace una descomposición jerárquica de los requisitos. Los nodos de la parte más alta podrían, por ejemplo, seguir la clasificación de características y subcaracterísticas ofrecida por la ISO/IEC 9126 como punto de partida para la definición de los requisitos del sistema.

- PAs asociadas a cada nodo (requisito). Las PAs tendrán diferentes estados de definición de menor a mayor detalle: identificación (un nombre para la PA),

definición (establecimiento de condiciones, pasos de ejecución y resultado esperado), y validación con el cliente (Figura 4)

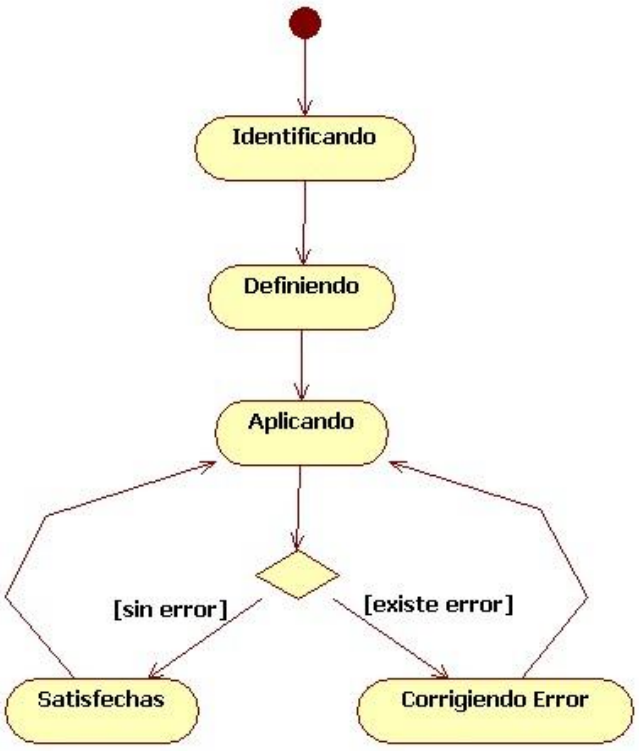


Figura 4. Diagrama Actividad de las PA

- Ilustraciones de las IUs para los nodos (requisitos) que tengan interacción con el usuario. Inicialmente se trata de bocetos de IUs, pero una vez implementados los requisitos los bocetos son reemplazados por las IU reales.

Dependiendo del requisito, podría ser útil utilizar otras formas de especificación, por ejemplo un Diagrama de Actividad si el comportamiento asociado al requisito es de carácter algorítmico o un Diagrama de Estados si el comportamiento incluye habilitación o deshabilitación de acciones de acuerdo con el estado del sistema. La premisa esencial es pragmatismo respecto de la especificación, con lo cual no se descarta el uso combinado de alternativas de especificación, pero el criterio debe ser el rentabilizar el esfuerzo en especificación y mantenimiento de dicha especificación.

En el ejemplo anterior, "*Retirar dinero*" sería un nodo de la arquitectura de requisitos. Sus pruebas (los nombres de las pruebas) podrían ser:

1. Reintegro usando cantidades predefinidas
2. Reintegro con cantidad introducida por cliente
3. Intento reintegro saldo < cantidad

4. Cancelación de operación
5. No disponibilidad de billetes
6. No disponibilidad de papel para recibo
7. Intento reintegro saldo < cantidad con cliente preferencial
8. Excedido tiempo de comunicación con sistema central
9. Excedido tiempo de espera para introducción de acción
10. …

Este trabajo de tesis se ha realizado en el contexto de un convenio universidad-empresa durante un cuatrimestre. La empresa es una PYME de desarrollo de software que tiene un ERP dirigido al sector socio-sanitario y cuenta con más de 40 empleados y más de 850 clientes. El autor de la tesis ha participado en la mejora de la metodología mediante la implementación de un nuevo módulo para la herramienta de apoyo a dicha metodología (TUNE-UP Process Tool). Tanto la metodología como la herramienta se están desarrollando y utilizando en la PYME desde hace 3 años.

La necesidad de crear este nuevo enfoque, surge ante la problemática de la empresa al verse incapaz de gestionar adecuadamente grandes cantidades de documentos de requisitos. Las metodologías Ágiles han criticado siempre estos documentos de requisitos, que simplemente quedan en desuso en la estantería. La realidad es que la mayoría de documentos de requisitos quedan desactualizados antes de que la tinta seque. Guardar los documentos de requisitos actualizados es una ardua tarea y raramente se completa de una manera satisfactoria. Debido a que la documentación estática queda anticuada rápidamente, puede dar lugar a malentendidos. Es por eso que Test Driven Requirements Engineering se refiere al uso de documentación de pruebas (pruebas de aceptación) bien escritas (y actualizadas) como especificaciones de requisitos.

El objetivo, entonces, es poder gestionar todos esos documentos de requisitos, a partir de los cuales surgieron alrededor de 400 formularios, 600 tablas y 600.000 líneas de código que comprendían los programas. Los documentos de requisitos se encontraban en formato de Word y eran editados constantemente usando las herramientas de revisión proporcionadas por dicho programa (comentarios, subrayados, tachado y un código de colores eran algunos de los recursos utilizados).

La organización era simple: cada producto estaba dividido en unas áreas y éstas a su vez por subáreas. Para cada versión había carpetas con las incidencias y éstas a su vez llenas de pruebas de aceptación afectadas por ellas (Figura 5). Una incidencia puede ser un nuevo requisito, una mejora o la corrección de un defecto, expresados principalmente en términos de pruebas de aceptación. Las correcciones, además, pueden tratarse como pruebas de aceptación que no queremos que se den en el producto.

Los documentos Word, además de contener las pruebas de aceptación, contenían también la información del contexto, esto es, la descripción y los bocetos que hicieran falta, todo en el mismo documento.

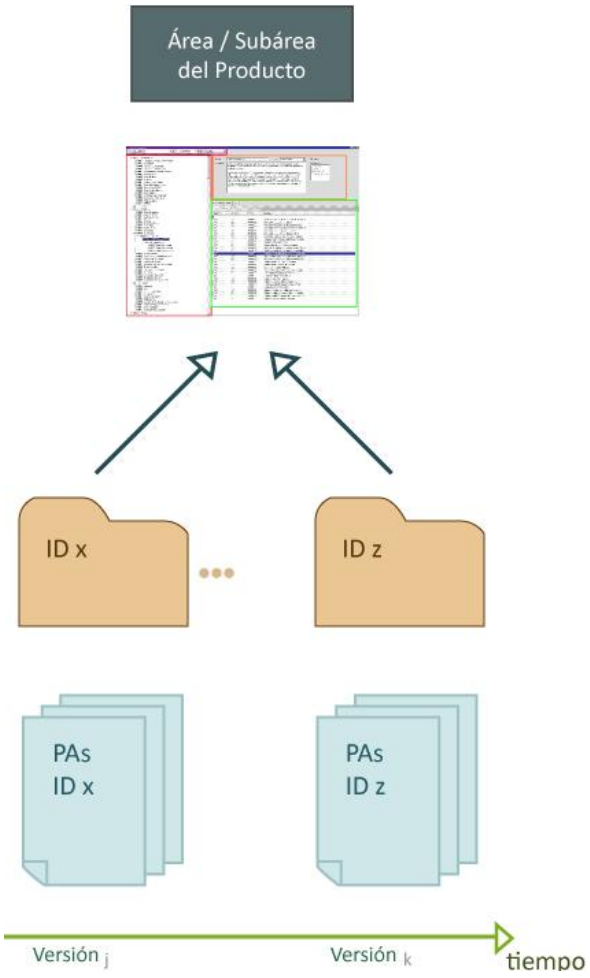


Figura 5. Organización de los requisitos por ficheros

El problema de que la organización fuera simple es que era poco flexible y al final un poco desordenado. Decimos al final porque en el momento en el que se está trabajando con una primera versión se tienen las pruebas archivadas por carpetas, cada carpeta correspondiente a una subárea y éstas a su vez dentro de otra área. Y en cada momento se sabe dónde está todo. El problema (y el desorden) viene cuando se va avanzando en el desarrollo, surgen otras versiones y en ese momento es muy complicado saber cuál es realmente la funcionalidad de cada área de la aplicación (tanto para el desarrollador como para el cliente), ya que al terminar la versión las pruebas de aceptación quedan en otra incidencia, a la que hay que volver si en el futuro queremos alguna información de ahí, y será difícil de encontrar. Si se quisiera por ejemplo saber qué hace un grid que tuviera la aplicación, tendríamos que ir buscando por las carpetas todas las pruebas de aceptación (que están en documentos de Word) que hacen referencia a ese grid. Era posible encontrar una prueba que dijera que tiene un comportamiento determinado y la siguiente prueba que dice que ya no lo tiene, con lo que la recolecta de información era una gran pérdida de tiempo que además podía llevar a confusiones.

El motivo de encontrarnos con esta situación es que una vez terminaba una incidencia, el documento Word quedaba asociado a la incidencia pero luego rápidamente ese

documento Word quedaba obsoleto cuando se producían nuevos cambios en esa parte de la aplicación.

Con esta forma de trabajo ellos iban sacando versiones; podían determinar precisamente cuáles son los cambios que realizaba una incidencia en el producto; pero no eran capaces de responder cuál es el comportamiento actual del producto, cuál era el comportamiento a fecha pasada o cuál sería el comportamiento futuro, porque el documento Word no daba más de sí en cuanto a esa información. No había una manipulación más allá del documento de Word y la organización de área/subárea no permitía saber qué parte de la aplicación ha sufrido o va a sufrir cambios, porque el mecanismo de área/subárea era muy pobre, un cambio solo podía afectar a un área/subárea, cuando en productos de gran envergadura es normal que pueda afectar a más de una parte del producto, por lo que se necesitaba mayor nivel de detalle. Además, cuando se definen nuevos cambios es siempre importante poder aprovechar la definición del comportamiento que ya está hecho en versiones pasadas.

Mientras en la empresa no había demasiados proyectos en marcha no se disponía de muchas pruebas y, por tanto, Word era una herramienta suficiente como soporte de especificación. Pero según iban creciendo los proyectos, y se alcanzaban las cifras antes comentadas, iba siendo más difícil gestionar la ingente cantidad de pruebas en documentos de texto y distribuidos por carpetas, y se hacía necesario un software, no sólo para el soporte de especificaciones, sino también para su total gestión.

Si en lugar de tenerlo así, organizamos los documentos como contenedores de pruebas de aceptación, en forma de nodos en un grafo. Al ver ese nodo veremos todas las pruebas de aceptación de las que constan más las propuestas de cambios, más las versiones anteriores, eso nos permite responder a las preguntas de qué funcionalidad se tenía, se tiene y se tendrá. Además todo queda de manera mucho más clara y ordenada.

El objetivo es incorporar el enfoque de Test-Driven Requirements Engineering en una herramienta de apoyo al proceso de desarrollo de software, para ello había que diseñar e implementar un módulo para gestionar los requisitos, así como integrar el módulo con otras herramientas para poder llevar a cabo todo el proceso de desarrollo y, por último, validar la efectividad del módulo.

Con esto lo que se pretende es:

- Mejorar la gestión de los requisitos en los programas, rompiendo con la estructura de áreas/subáreas y creando una nueva estructura en forma de grafo, más flexible y potente para organizar los requisitos (Figura 6). Para nosotros la estructura de requisitos corresponde con la estructura del producto, que es un grafo acíclico dirigido cuyos nodos son contenedores de comportamiento especificado por pruebas de aceptación. Un cambio en el comportamiento del producto viene dado por una incidencia (o unidad de trabajo), que va a impactar a uno o más requisitos de la estructura de

requisitos del producto, añadiendo, modificando o eliminando pruebas de aceptación.

- Establecer un mecanismo para poder definir el impacto de una incidencia a los nodos en términos de añadir, modificar y eliminar pruebas de aceptación. Es decir, establecer un mecanismo de gestión de pruebas de aceptación asociadas a las incidencias. De esta forma si hay una incidencia ya no existe la limitación de que afecte sólo a un requisito, sino que pueda afectar a más de uno a la vez.

- Evitar duplicaciones y poder reutilizar (pues algunos requisitos pueden servir para varios programas) con el consiguiente ahorro de tiempo.

- Aumentar drásticamente la facilidad de realizar modificaciones (mantenimiento) sobre las pruebas de aceptación, concentrando el trabajo de análisis y de programación en la implementación de "la diferencia de comportamiento" y dejando para los testers el verificar la satisfacción de las pruebas de regresión.

- Hacer más intuitiva la edición tanto para el que hace el cambio en la prueba como para el que tenga que leerla posteriormente.

- Mantener las pruebas de aceptación siempre en concordancia y armonía con los requisitos actuales, sin quedar la documentación desfasada o intocable por ser ya demasiado difícil o tedioso realizar más cambios sobre una prueba.

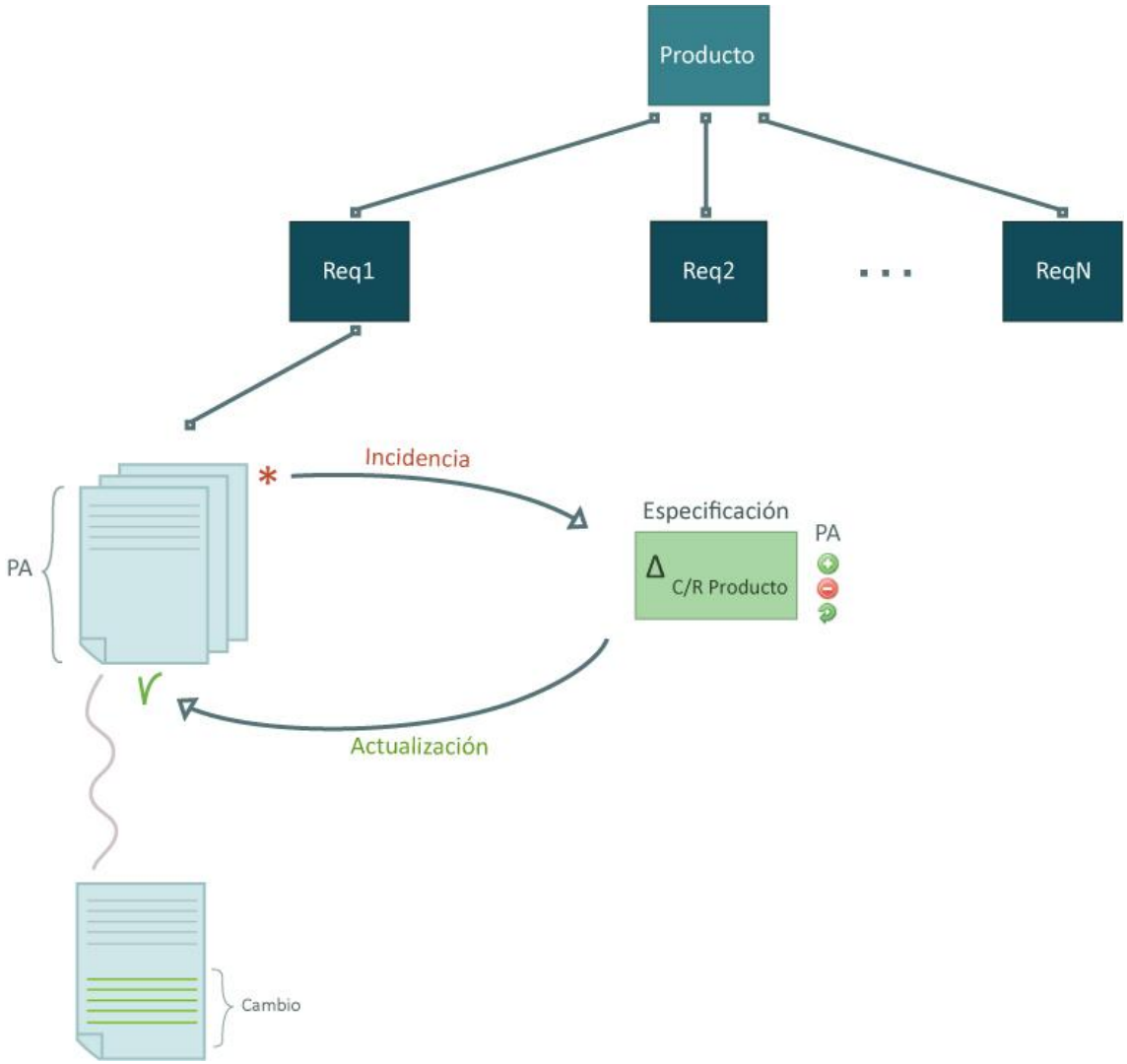


Figura 6. Enfoque TDRE

La figura 6 muestra nuestra versión simplificada de lo que sería el desarrollo Test-Driven en el ámbito de los requisitos y la planificación (TDRE), la cual va más allá de la programación ya que utilizamos la connotación de iteraciones, versiones, etc. En el paso de una versión a otra del producto, el cambio en el comportamiento viene dado por incidencias (o unidades de trabajo) a realizar durante una determinada versión.

1.2 Estructura de la Memoria

Los pasos para la creación de la tesis han quedado plasmados en los siguientes apartados de esta memoria:

• Capítulo 2: Estado del arte, en este capítulo introducimos la base teórica sobre la que se basa este trabajo, cuál es la situación actual en el desarrollo de software y las propuestas que hay de mejora. Usaremos estas mismas propuestas como fundamento para rebatirlas y dar una visión clara de cuáles son las aportaciones al conocimiento que realiza nuestra propuesta de Test Driven Requirements Engineering al estado del conocimiento actual.

• Capítulo 3: Gestión de requisitos dirigida por pruebas, en este capítulo se explica nuestra propuesta y cuál debería ser la forma en la que se deberían especificar los requisitos. Hablaremos también de la herramienta TUNE-UP y TUNE-UP Process Tool como introducción para entender el contexto en el que se enmarca el Gestor de Requisitos que hemos desarrollado.

• Capítulo 4: Módulo de requisitos, en este capítulo analizamos la creación del Gestor de Requisitos, su arquitectura, tecnología utilizada y sus requisitos, estos últimos hechos con el mismo módulo.

• Capítulo 5: Aquí explicamos cómo fue el proceso de implantación del módulo de Requisitos con la herramienta TUNE-UP: los problemas, desafíos y la valoración del proceso; así como las conclusiones que se extraen de ellos. También hablaremos de posibles ampliaciones que se pueden realizar en un futuro para dicho módulo.

• ANEXO A: Podremos encontrar las pruebas de aceptación que se hicieron, a las que se sometió el módulo obtenido, los cuales sirven para garantizar que se cumplen los objetivos planeados.

Capítulo 2. Estado del Arte

2.1 Introducción

(El siguiente subcapítulo está extraído y traducido de [8]*)*

Desde finales de los 90, cuando la revolución del software hizo que los proyectos se aceleraran, empresas de desarrollo se han esforzado por encontrar la manera de operar dentro de las estrictas exigencias impuestas por los presupuestos, plazos de entrega tempranos, y sin haber un compromiso de calidad. Tanto los departamentos de marketing, testing, facilidad de uso, y desarrollo trataban de hacer realidad una idea, se esperaba que los programadores funcionasen como un engranaje, más que como la rueda. Como resultado de ello, el enfoque de un determinado proyecto se dispersaba a través de grupos empresariales. La comprensión de la necesidad que había inspirado el proyecto se perdía en el impulso para hacerlo. La prioridad era producir un programa que funcionara, hacerlo rápido y dentro del plazo establecido. El cumplimiento de las necesidades del cliente o de su punto de vista no se tenía en consideración.

A finales de los 90 apareció Extreme Programming (XP). XP es una metodología de programación que sitúa a los desarrolladores en el centro del proceso. En realidad, los clientes están en el centro con los desarrolladores, con una estrecha relación de equipo que incorpora una intensa filosofía test-first. Test-First Development (TFD) o Test Driven Development (TDD), una práctica del núcleo de XP, se convirtió rápidamente en una práctica ampliamente adoptada.

Desde entonces, TDD se ha hecho cada vez más importante, no sólo donde se practica XP. La reputación de TDD se ha extendido hasta un grado que muchas empresas requieren una base de TDD como condición de trabajo. De esta forma TDD se ha hecho bastante popular, aunque oficialmente no se requiere.

Los pasos de los que consta el TDD se pueden ver en el Diagrama de Actividad UML de la Figura 7. El primer paso es añadir una prueba. Después se ejecutan los tests, normalmente el conjunto completo de pruebas, aunque en aras de aumentar la velocidad se puede decidir ejecutar sólo un subconjunto para asegurar que las nuevas pruebas no fallan. Entonces se actualiza el código funcional para hacer pasar las nuevas pruebas. El cuarto paso es ejecutar las pruebas de nuevo. Si fallan hay que actualizar el código funcional y repetir las pruebas. Una vez que las pruebas pasan, el siguiente paso es volver a empezar.

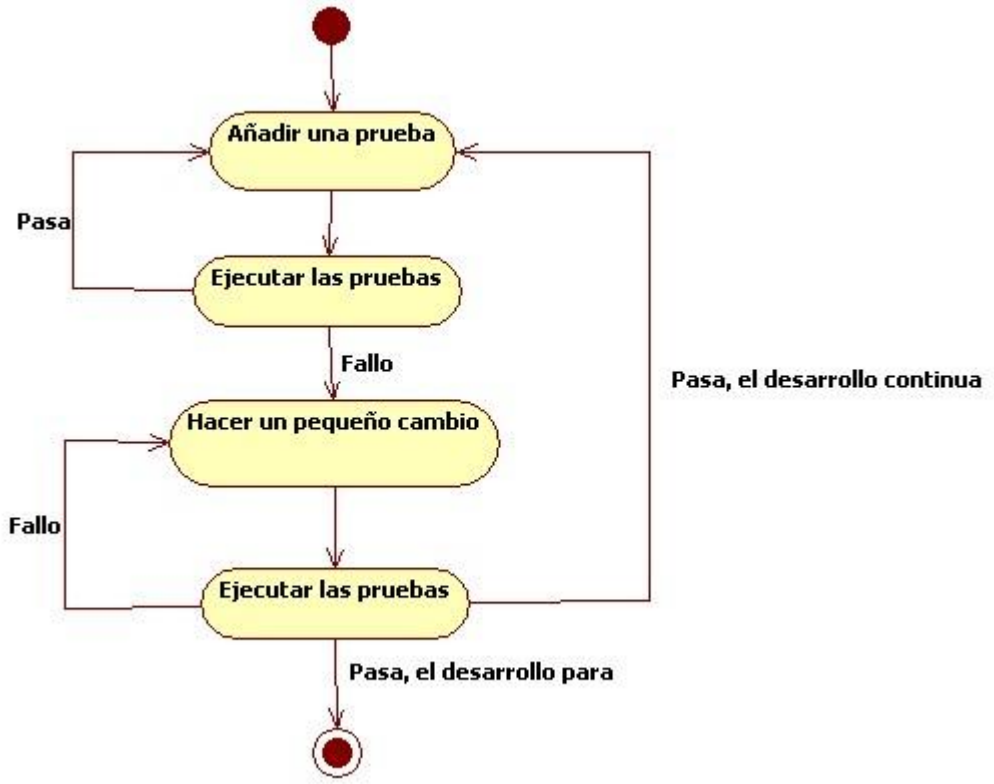


Figura 7. Pasos de test-first design (TFD).

Cuando se va a implementar una nueva característica, la primera pregunta que surge es si el diseño actual es el mejor diseño posible que permite implementar la nueva funcionalidad. Si es así, se procede a través de un enfoque TDD. Si no es así, se refactoriza localmente para cambiar el diseño de la parte afectada por la nueva función, lo que permite añadir características de la manera más fácil posible. Como resultado siempre se estará mejorando la calidad del diseño, haciendo más fácil trabajar en el futuro.

En vez de escribir código funcional primero y luego su código de prueba, si es que se llega a escribir, en lugar de eso se escribe el código de prueba antes del código funcional. Un programador usando un enfoque TDD se debe negar a escribir una nueva función hasta que primero haya una prueba que falla debido a que dicha función no está presente. De hecho, se niegan a añadir ni una sola línea de código hasta que exista una prueba para ello. Una vez que la prueba está en su lugar, a continuación, hacen el trabajo necesario para asegurar que el conjunto de pruebas pasa (su nuevo código puede hacer fallar varias de las anteriores pruebas, así como la nueva).

Uno de los aspectos destacables de TDD es que existe un unit-testing Framework disponible para el usuario. Los desarrolladores de software Ágiles normalmente usan la familia de herramientas libres de xUnit [9], como JUnit [10], aunque las herramientas comerciales también son opciones viables. La Figura 8 presenta un Diagrama de Actividad UML de cómo la gente trabaja típicamente con las herramientas de xUnit.

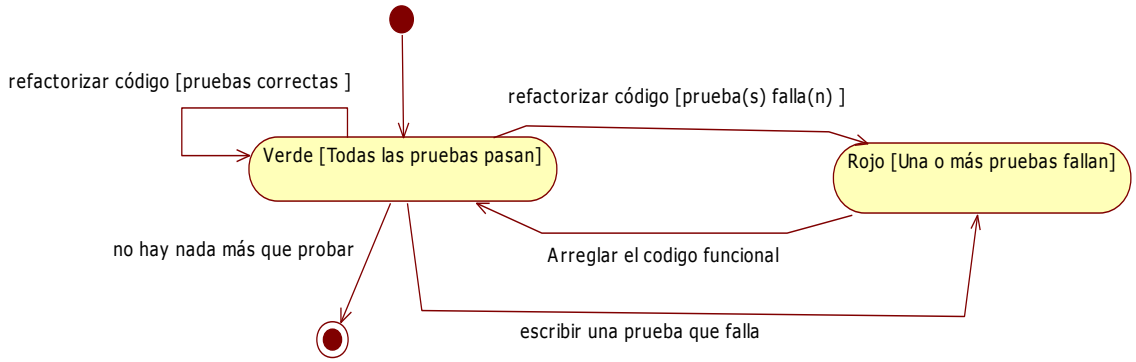


Figura 8. Testing a través del xUnit Framework.

Kent Beck, quién popularizó TDD en eXtreme Programming (XP) [11], define dos simples reglas para TDD: Primero, se debe escribir el código de la lógica de negocio sólo cuando una prueba automática ha fallado. Segundo, se debe eliminar cualquier duplicación que se encuentre. Beck explica cómo estas dos simples reglas generan un complejo comportamiento individual y colectivo:

- Un diseño orgánico, con el código ejecutable proviniendo de retroalimentación entre decisiones.

- El desarrollador escribe sus propias pruebas, evitando pérdidas de tiempo si tiene que esperar que sea un tercero quien lo haga.

- El entorno de desarrollo debe proveer una respuesta rápida a pequeños cambios (se necesita un compilador rápido y un conjunto de pruebas de regresión)

- El diseño debe estar altamente cohesionado, con componentes débilmente acoplados (un diseño altamente normalizado) para hacer las pruebas de manera más fácil (esto también hace que la evolución y el mantenimiento del sistema sea más fácil)

Para los desarrolladores, la implicación es que necesitan aprender cómo escribir unidades de prueba efectivas. En la experiencia, Beck aconseja:

- Ejecutar rápido (tienen instalaciones, tiempos de ejecución y fallos cortos).
- Ejecutar en aislamiento (se deberían poder reordenar).
- Usar datos que las hagan más fáciles de leer y entender.
- Usar datos reales (ej: copias de los datos de producción) cuando se necesiten.

- Representan un paso hacia el objetivo general.

TDD es principalmente una técnica de diseño con el efecto secundario de asegurar que el código fuente está rigurosamente probado en unidades. Sin embargo, hay más que probar que esto. Aún se necesita considerar otras técnicas de testing, como la "*Agile Acceptance Testing*" [12]. Gran parte de este testeo puede ser hecho en una fase temprana del proyecto si se decide así (y así debería de ser). De hecho, en XP las pruebas de aceptación para una historia de usuario son especificadas por el cliente del proyecto, incluso antes o en paralelo al código, dando al cliente la confianza de que el sistema cumple sus requisitos.

Con el testing tradicional una prueba satisfactoria encuentra uno o más defectos. Lo mismo pasa con TDD; cuando una prueba falla se ha hecho un progreso porque se sabe qué es necesario para resolver el problema. Más importante, se tiene una clara seguridad de éxito cuando la prueba no vuelve a fallar. TDD incrementa la confianza de que el sistema actual cumple los requisitos definidos, que el sistema actual funciona y por tanto, que se puede proceder con confianza.

Al igual que con las pruebas tradicionales, mientras mayor es el perfil de riesgo del sistema, más rigurosas tienen que ser las pruebas. Con ambos, pruebas tradicionales y TDD, no se está alcanzando la perfección, sino que son pruebas por la importancia del sistema. Parafraseando Modelado Ágil (AM) [13], se debe "*probar con un fin*", saber por qué se está probando y hasta qué nivel necesita ser probado. Un interesante efecto secundario de TDD es que logra una cobertura del 100% de pruebas - cada línea de código se prueba – lo cual las pruebas tradicionales no garantizan (aunque lo recomienda). Aunque TDD es una especificación técnica, un valioso efecto secundario es que el código resultado es significativamente mejor que con las técnicas tradicionales.

La mayoría de programadores no leen la documentación escrita para el sistema; en lugar de eso prefieren trabajar directamente con el código. Y no hay nada de malo en eso. Cuando intenta entender una clase u operación, la mayoría primero mirará un código de muestra que lo invoque. Las pruebas unitarias hacen exactamente esto – proveen de una especificación que funciona del código funcional- y como resultado las pruebas unitarias consiguen ser una porción significante del documento técnico. Similarmente, las pruebas de aceptación pueden formar una parte importante de la documentación de requisitos. Las pruebas de aceptación definen exactamente lo que el cliente espera del sistema, por tanto, ellas especifican los requisitos críticos. El conjunto de pruebas de regresión, particularmente con una aproximación test-first, se convierte en unas especificaciones ejecutables detalladas de manera eficaz.

¿Son las pruebas suficiente documentación? Posiblemente no, pero forman una parte importante. Por ejemplo, es posible que todavía se necesite al usuario, la visión general del sistema, operaciones y documentación de apoyo. También puede necesitarse documentación resumen de la visión general del proceso de negocio que el sistema soporta. Es posible observar con facilidad que estos dos tipos de pruebas

cubren la mayoría de la documentación necesaria para los desarrolladores y los clientes del negocio.

Los desarrolladores enumeran muchas ventajas por trabajar de esta manera rápida e incremental. Algunas de las más destacadas son [8]:

- Rápidos resultados: Los desarrolladores pueden ver el efecto de decisiones de diseño en minutos.

- Flexibilidad: Los cambios son fáciles, debido a la pequeña distancia entre pruebas.

- Catálogo automático de pruebas de regresión: Si lo que se desarrolló hace seis meses de repente falla en el código de hoy, se sabe inmediatamente.

- Código bueno y limpio que funciona

2.2 Enfoques

En TDD, los desarrolladores escriben pruebas para especificar qué debería hacer una unidad de código, entonces implementar esta unidad de código para satisfacer los requisitos. Estas pruebas se llaman pruebas unitarias, y se centran en pequeñas unidades de código. Los beneficios de esta aproximación son que se centra más en qué debería hacer el sistema y tener un objetivo claro definido para los desarrolladores. Las pruebas unitarias también mantienen el sistema unido durante los cambios, permitiendo hacerlo más flexible. Si un cambio no intencional hace fallar alguna funcionalidad existente, la prueba unitaria relevante fallará y alertará a los desarrolladores del problema.

La práctica de pruebas unitarias era tan útil que era lógico preguntarse si la misma práctica podría aplicarse a las reglas del negocio y dirigir todas las fases del proyecto en lugar de sólo las unidades de código. Las pruebas unitarias se centran en el código, así que están completamente en el dominio de los desarrolladores de software. Las reglas de negocio, por otro lado, no deberían estar definidas por los desarrolladores. Tienen que ser definidas por expertos del dominio y clientes. Pero los expertos del dominio del negocio rara vez entienden los lenguajes de programación, así que usar las mismas herramientas para dirigir la implementación de las reglas del negocio y las unidades de código normalmente falla. Los desarrolladores podrían escribir pruebas de reglas del negocio con herramientas de pruebas unitarias, pero estas pruebas sólo reflejarían lo que el desarrollador entiende y se verían afectados por problemas de comunicación con el cliente. Para conseguir las mejores especificaciones, el cliente y los equipos de desarrollo tienen que trabajar juntos. No hay forma de que una persona del negocio verifique que una prueba de desarrollo describe el objetivo final correctamente con herramientas de pruebas unitarias. Así que, en teoría, las ideas de

pruebas unitarias podrían ser aplicadas a las reglas del negocio también, pero en la práctica las pruebas hacen imposible la comunicación.

Para solucionar el problema, se necesitan mejores formas para especificar y automatizar pruebas para las reglas del negocio, que se puedan usar también para comunicarse con la gente del negocio. Dichas herramientas tendrían que centrarse en capturar la visión del cliente de qué debería hacer el sistema cuando esté acabado. Entonces podríamos aplicar las ideas de pruebas unitarias al código de desarrollo, asegurando las pruebas, y ejecutar estas pruebas para verificar que el código va en la dirección correcta, repitiendo el proceso hasta que todas las pruebas estén correctas.

Es a partir de este punto cuando empiezan a describirse los diferentes enfoques que han surgido con tal fin.

2.2.1 Test Driven Requirements

Sobre este enfoque sólo se ha encontrado un libro [14], y consultando la página web del autor, Amr Elssamadisy [15], parece ser que no ha habido ningún trabajo más al respecto. El libro data de Marzo del 2007.

Aunque el nombre pueda coincidir con el que le hemos dado a nuestro enfoque, el concepto que le da no es el mismo, tratándose éste de un conjunto de prácticas bastante parecido al de Story-Driven Development, presentado en la sección 2.2.6.

La clave de Test-Driven Requirements reside en tener un *cliente como parte del equipo,* que trabaja cerca de los desarrolladores para escribir Pruebas de Aceptación; que el cliente escriba sus propios requisitos como Pruebas de Aceptación en lugar del método previo que se usara. Haciendo esto se obtendrá un método de comunicación concreto y no ambiguo entre el cliente y los desarrolladores, aunque estén distribuidos y sean equipos multi-culturales. También tener Integración Continua [16] incluye no sólo pruebas desarrolladas automatizadas, sino también todas las Pruebas de Aceptación en cada compilación.

La misión del desarrollador es construir la parte del sistema que cumplirá las Pruebas de Aceptación y construir la infraestructura para que las pruebas se ejecuten correctamente. Una vez que las nuevas Pruebas de Aceptación pasen, el desarrollador ejecuta todas las pruebas desarrolladas automatizadas y todas las Pruebas de Aceptación para el sistema entero localmente, en caso de éxito se introduce el nuevo código en la fuente. Debido a que las Pruebas de Aceptación se ejecutan por Integración Continua, todos los requisitos hechos por el equipo entero durante todas las iteraciones estarán probados.

Estas prácticas, cuando se usan conjuntamente como se describe, conforman el núcleo de Test-Driven Requirements. Los requisitos se escriben como pruebas y el mismo bucle de realimentación que encontramos en Test-Driven Development se expande para incluir al equipo entero.

Como otros enfoques técnicos, Test-Driven Requirements depende de todas sus prácticas para ser llevado a cabo correctamente. Si cualquiera de estas tres prácticas tiene problemas afectará al núcleo del funcionamiento.

El problema más común es que las Pruebas de Aceptación se ejecuten lentas. Esto causa dos problemas:

- Los desarrolladores no ejecutaran todas las pruebas en cada compilación. Por tanto, la Integración Continua probablemente fallará.

- La elaboración de la Integración Continua será más lenta y las pruebas fallarán sin una indicación clara de quien debería arreglar las pruebas defectuosas.

Con el fin de hacer llegar las Pruebas de Aceptación a la Integración Continua, las pruebas deben ser suficientemente rápidas. Primero, el equipo debe comprometerse a las Pruebas de Aceptación como una práctica de desarrollo primera, en lugar de dejarla en segundo lugar. El objetivo principal es acelerar la ejecución de las Pruebas de Aceptación, para que puedan ser ejecutadas eficientemente por los desarrolladores en sus máquinas locales, antes de hacer la integración.

La herramienta más popular para Test-Driven Requirementes es FIT [17]. Básicamente es un entorno que ayuda a añadir pruebas como especificaciones. Fue creada originalmente por Robert C Martin, Micah Martin y Michael Feathers. Posteriormente Robert C Martin trabajó en otra herramienta llamada SLIM [18].

Hay una variación de Test-Driven Requirements usando Pruebas xUnit cuando el cliente es un técnico. Con un cliente técnico tanto las pruebas como el código pueden ser más apropiados y naturales que una solución con FIT. Esta técnica puede ser válida también si el cliente no escribe las pruebas pero le "*cuenta*" al desarrollador qué hacer.

2.2.2 Behaviour Driven Development

(Esta sección ha sido extraída de [19] *)*

Behaviour Driven Development (BDD) [20] es una práctica de desarrollo que surge de las metodologías de desarrollo ágiles, creada por Dan North [21] en el 2006 como respuesta a los problemas experimentados usando y enseñando TDD.

En su núcleo BDD es un refinamiento de TDD que mueve el énfasis de las pruebas a la especificación y es, en efecto, la combinación de las "*mejores prácticas*" de TDD con las características de otra práctica de desarrollo para intentar vencer los impedimentos técnicos y organizacionales de la adopción de TDD. Aunque no parece un gran cambio, el mover el énfasis de las pruebas a la especificación trae un importante número de beneficios que impactan en el desarrollo de la prueba y el código producido.

Como dice Robert C. Martin en [19]:

"El acto de escribir una prueba unitaria es más un acto de diseño que de verificación. También es más un acto de documentación que de verificación. El acto de escribir una prueba unitaria cierra un número considerable de ciclos de retroalimentación, el menor de los cuales es el que atañe a la verificación de la función."

Algunas características de BDD son las siguientes:

- Un lenguaje ubicuo.

Para muchos, el lenguaje usado para expresar un concepto tiene un impacto en la forma en que se piensa. "*Las abstracciones y conceptos que usamos para expresarnos en cualquier lenguaje forman el camino en que pensamos sobre el problema que estamos resolviendo*" [22]. Esta teoría está apoyada por la hipótesis del lingüístico Sapir-Whorf, la cual postula que hay "*una relación sistemática entre las categorías gramaticales de la lengua en la que habla una persona y cómo esta persona entiende el mundo y como se comporta*" [23]. Para el software, esto implica que el lenguaje que usamos para describir las construcciones de software tiene un impacto en cómo creamos estas construcciones; cambiando el lenguaje, afectamos al código que creamos.

BDD apoya el uso de lenguajes significativos de dos formas. Primero, se centra en "*obtener las palabras correctas*", motivando nombres apropiados para clases, métodos y variables. Segundo, saca conceptos del Domain Driven Development (DDD) [24] para cubrir el hueco entre los artefactos de negocio y los técnicos.

- Centrado en el diseño

"Un 'diseñador' en cascada parte de una comprensión del problema y crea algún tipo de modelo para una solución, la cual entonces pasa a quien lo implementará. Un desarrolador ágil hace exactamente lo mismo, pero el lenguaje que usa para el modelo es código fuente ejecutable en lugar de documentos o UML." Kerry Buckley [19]

Una de las consecuencias más importantes de practicar TDD es su influencia en el diseño del código resultante. El código test-driven tiene menos defectos que el código no dirigido por pruebas y normalmente es más cohesivo y está menos ligado que el no dirigido por pruebas. Sin embargo, el énfasis en las pruebas limita la eficacia de TDD en diseño dirigido. BDD reconoce que el diseño es uno de las aportaciones más importantes de las pruebas, produciendo un lenguaje y herramientas que soportan la creación de código bien diseñado.

- Centrado en el comportamiento

"Así que si no va sobre hacer pruebas, ¿cuál es el propósito? Entender lo que se está intentando hacer antes de empezar nada sin planificar ni saber completamente qué estás haciendo." Dave Astels [19]

Centrándose en el comportamiento, los frameworks de BDD rompen el tradicional mapeo 1 a 1 de las clases de pruebas unitarias para una clase de producción, BDD anima a usar un mapeo M a N, permitiendo tantas clases de especificación como sean necesarias para especificar el comportamiento requerido.

- BDD Frameworks

Los primeros frameworks de BDD se centraron en el nivel de compresión de aceptación. Algunos frameworks como JBehave [25] permiten que el comportamiento de una aplicación sea descrito en términos de historias y escenarios. Subsecuentemente, los frameworks como RSpec [26] (para Ruby) y NSpec [27] (para C#) se centran más en el nivel de pruebas del código, ofreciendo un equivalente a la característica de xUnit [9], para niveles más bajos de especificaciones. Ambas aproximaciones son legítimas y usan lenguaje ubicuo para describir el comportamiento deseado del sistema en el nivel apropiado.

Herramientas (lenguaje de implementación y nombre):

- BOO - Specter - http://specter.sourceforge.net/
- C - CSpec - http://github.com/arnaudbrejeon/cspec/
- C++ - CppSpec Spec-CPP - http://www.laughingpanda.org/projects/cppspec/
- C# .Net - NSpec - http://nspec.tigris.org/
- .Net - NBehave - http://nbehave.org/
- .Net - NSpecify (incompleto) - http://nspecify.sourceforge.net/
- Delphi - dSpec - http://dspec.sourceforge.net/
- Groovy - GSpec – http://codeforfun.wordpress.com/gspec/
- Java - JBehave – http://xircles.codehaus.org/projects/jbehave
- Java - JDave –http://www.jdave.org/
- Java - beanSpec –http://beanspec.sourceforge.net/
- Java - Instinct - http://code.google.com/p/instinct/
- Javascript - JSSpec - http://jania.pe.kr/aw/moin.cgi/JSSpec
- PHP - PHPSpec - http://code.google.com/p/phpspec/
- Python - Specipy - http://darcs.idyll.org/~t/projects/pinocchio/doc/
- Ruby - RSpec - http://rspec.info/
- Ruby - Shoulda - http://thoughtbot.com/community/
- Ruby - test-spec & bacon – http://rubyforge.org/projects/test-spec/
- Scala - Specs - http://code.google.com/p/specs/

2.2.3 Acceptance Test Driven Development

(*La siguiente sección ha sido extraída y traducida de las siguientes fuentes:* [12] *y* [28])

Acceptance Test-Driven Development (ATDD) fue ideado por Lasse Koskela en el 2007. ATDD nos da un mecanismo para usar Domain-Specific Languages (DSL's) [29] e involucrar directamente al cliente en el proceso de creación de software, asegurando que se cumplen los requisitos. Una propiedad importante de las pruebas de aceptación es que se utilice el idioma del dominio y del cliente, en lugar de jerga técnica que sólo entiende el programador. Este es el requisito fundamental para tener al cliente involucrado en la creación de pruebas de aceptación y ayuda enormemente con el trabajo de validación de las pruebas.

En una aproximación ATDD, las especificaciones e implementaciones de software se dirigen por ejemplos concretos. Describir casos de uso como pruebas fuerza al diseñador a definir las especificaciones del software en un detalle que no se podría conseguir con la capacidad de los documentos de especificación de requisitos. En el enfoque tradicional, el rol de las pruebas de aceptación es menos importante: se llevan a cabo posteriormente en todo caso.

El enfoque ATDD se basa en la idea de hacer pruebas de aceptación automáticas, incluso antes de la fase de programación, permitiendo al equipo probar inmediatamente las funciones objetivo. Las pruebas pueden ser definidas por los usuarios, los clientes o especialistas de especificaciones funcionales mientras que los programadores o testers llevan a cabo su automatización.

Cuando convertimos los requisitos en pruebas de aceptación automáticas, es mucho más fácil para los clientes ver que está recibiendo lo que pidió. También es más fácil para los desarrolladores, que tienen un objetivo bien conocido que cumplir. De esta manera, los desarrolladores están más centrados en la entrega de una unidad específica con la funcionalidad que el cliente necesita, más que (como ocurre a menudo) en pensar algunas "*nuevas características*" que en su opinión podrían ser útiles.

Aunque no se elimina la necesidad de hacer pruebas manuales, particularmente en lo que concierne a la interfaz de usuario, las pruebas automatizadas permiten a los testers centrarse en tareas donde la gente suple a las máquinas. La falta de pruebas de aceptación automáticas implica que los testers gastarán su tiempo en pruebas de regresión. Así mismo, los desarrolladores tendrán que esperar para la realimentación ya que las pruebas manuales pueden tomar varios días.

ATDD toma el enfoque business-driven [30] hasta el extremo. El trabajo comienza con el desarrollo de las funciones concretas en las que el cliente debe centrar sus recursos. Las soluciones técnicas y detalles son elegidos para servir directamente a las necesidades de la empresa.

Una propiedad común de las pruebas de aceptación es que no pueden ser implementadas (automatizadas) utilizando el mismo lenguaje de programación de sistema que se está probando. Si éste es el caso, depende de las tecnologías involucradas y la arquitectura en general del sistema bajo prueba. Por ejemplo, algunos lenguajes de programación son más fáciles para interoperar que otros. Del mismo modo, es fácil escribir pruebas de aceptación de una aplicación web a través del protocolo HTTP [31] con prácticamente cualquier idioma que queramos, pero a menudo es imposible ejecutar pruebas de aceptación para software embebido escritas en cualquier lenguaje que no sea el del propio sistema.

Test-driven development da al programador las herramientas para hacer evolucionar su software en pasos pequeños, siempre con la seguridad de que el programa funcione como se esperaba. En acceptance test-driven development, esta seguridad se gana, no en la corrección del nivel técnico, sino en el nivel de las características, "*¿hace el software lo que quiero que haga?*"

En otras palabras, aunque en TDD estamos definiendo primero el comportamiento específico, queremos que nuestro código base lo muestre y sólo entonces implementar dicho comportamiento. En ATDD primero definimos los valores de la funcionalidad que el cliente quiere del sistema como un conjunto, para mostrarlo y sólo después implementar dicho comportamiento, posiblemente usando TDD como vehículo.

En su forma más simple, el proceso de Acceptance Test-driven Development puede ser expresado como el simple ciclo ilustrado en la Figura 9.

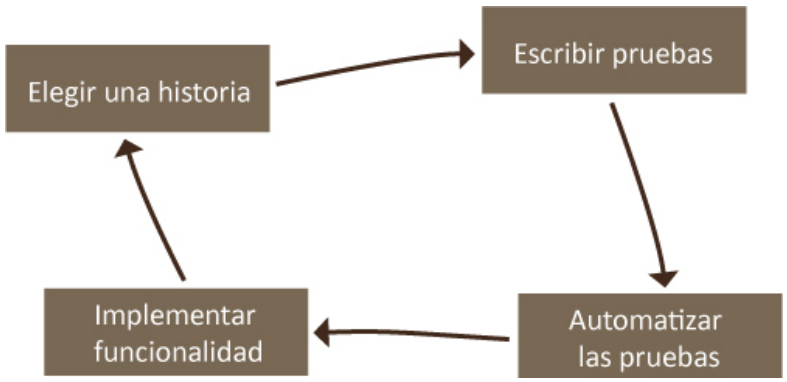


Figura 9. Ciclo del ATDD

Este ciclo continua a través de la iteración mientras tengamos más historias que implementar, empezando otra vez: seleccionando una historia de usuario, escribiendo entonces pruebas para la historia elegida, pasando estas pruebas a pruebas automatizables y ejecutables; y finalmente implementando la funcionalidad para hacer pasar las pruebas de aceptación.

ATDD funciona mejor cuando el equipo ha adoptado alguno de los muchos frameworks disponibles para automatización de pruebas, como FIT/Fitnesse, Concordian [32], Cucumber [33], etc.

2.2.4 Customer test-driven development

(*La siguiente sección ha sido extraída y traducida de* [34])

Test Driven Development tiene muchas ventajas. No sólo en el nivel de pruebas unitarias, sino también en los niveles funcionales, de sistema y de pruebas de aceptación, que se benefician de incrementar la comunicación y colaboración.

Test Driven Development (TDD) fuerza al programador a pensar en muchos aspectos de cada característica antes de implementarla. Como resultado da una red de pruebas seguras, asegurando que el código refactorizado mantiene su funcionalidad.

Customer test-driven development (CTDD) es ideado por Lisa Crispin en el año 2005, ante la idea de aprovechar esa red de pruebas que aparecen al utilizar TDD. Consiste, entonces, en dirigir proyectos con pruebas y ejemplos que ilustran los requisitos y las reglas de negocio.

Las pruebas de cliente (Customer Tests), básicamente, están entre las pruebas unitarias y de integración (o contrato), las cuales se hacen por y para los programadores, probando pequeñas unidades de código y su interacción. Se usa el término "*customer*" (cliente) en el sentido de XP [4], significando los propietarios del producto y la gente del lado del negocio que especifica las características a entregar. Las pruebas de cliente pueden incluir pruebas funcionales, de sistema, de extremo a extremo, de rendimiento, carga, estrés, seguridad y usabilidad, entre otros. Estas pruebas muestran al cliente si el código entregado cumple con sus expectativas.

Para saber con más facilidad lo que espera el cliente con el tiempo, se separan las características en pequeñas piezas más manejables, llamadas historias. Normalmente se escriben en pequeñas tarjetas.

Para cada historia, el cliente tiene que explicar, a través de pruebas y ejemplos, cómo sabrá que la historia está terminada. Como en TDD, antes de escribir ningún código, se escriben las pruebas, cuando pasan, demostramos que el código cumple los requisitos. Estas pruebas están idealmente hechas de forma que pueden ser usadas por una herramienta automatizada, pero también pueden ser pruebas de más alto nivel, o guías para una prueba exploratoria posterior.

¿Quiénes son estos clientes colaboradores? Son la gente experta en el dominio, que entienden las prioridades del negocio. Pueden estar en ventas o marketing, pueden ser gerentes de negocio o analistas, pueden ser usuarios finales, también pueden ser un manager de desarrolladores o apoyo del cliente. Posiblemente necesitaran ayuda para escribir pruebas eficientes que el programador pueda usar.

Los testers proveen esa ayuda, combinando su conocimiento de los requisitos técnicos del sistema con lo que el negocio necesita. Los testers hacen preguntas para ayudar a sacar los requisitos y detalles más ocultos.

Escribiendo pruebas de clientes en lugar de implementar las características, podemos sacar supuestos ocultos a la superficie. Las pruebas de cliente no reemplazan directamente la comunicación entre los programadores y los clientes, la mejoran y documentan los requisitos de historias. Las preguntas frecuentes con el cliente y el uso de ejemplos, permite obtener los mejores resultados posibles.

Pasándolas pronto a pruebas automatizadas, hace que se tengan que hablar entre sí cuando lo necesitan, antes y durante la implementación.

Es importante escribir las pruebas con un objetivo de equipo durante unas pocas iteraciones, hasta que sea un hábito.

Los equipos que usan TDD, especialmente aquellos que lo prueban por primera vez, pueden tender a hacer sólo las pruebas más obvias. Pueden quedar sin detectar requisitos no entendidos y defectos difíciles de encontrar. Escribiendo primero pruebas de cliente se provee una navegación para el proyecto. Las pruebas ayudan al equipo a identificar el camino a seguir. Cuando las pruebas pasan, sabemos que hemos llegado a nuestro objetivo.

2.2.5 Functional Test Driven Development

(*El siguiente texto ha sido extraído y traducido de* [35])

TDD tiene también una rama llamada Functional Test Driven Development (FTDD), creada por Dave Nicolette en el año 2005.

El propósito tradicional de las pruebas funcionales ha sido la evaluación de la calidad. Esta es un área de TDD que todavía necesita mucho mejores herramientas, pero su impacto e importancia para la calidad del software, es tan alta, que debería ser un punto a tener en cuenta para el futuro. Como en TDD, la idea de FTDD es que las pruebas resultantes puedan ser ejecutadas automáticamente.

Lo que se pretende con FTDD es que las pruebas funcionales escritas se comporten como requisitos y especificaciones para el software. Las pruebas de aceptación ya no se limitan a evaluar la calidad, su propósito ahora es guiar la calidad. Cuando las pruebas de aceptación sirven tanto para especificación del sistema como para pruebas de regresión automáticas, deben seguir siendo viables durante el tiempo de vida de la producción de código. Por todo esto, los casos de prueba deben ser fáciles de leer y escribir, tienen que ser más fáciles y seguros de mantener, y debe ser fácil encontrar alguna funcionalidad desde ellos. En cierta medida las pruebas también deben ser más correctas para no crear o enmascarar errores. Estas pruebas deberían perdurar más que una prueba unitaria.

El framework open source que intenta dirigir estas características es FitNesse. Se presenta a sí mismo como un Framework de pruebas de aceptación y parece operar

con pruebas funcionales y de integración. FitNesse contiene algunas definiciones funcionales y scripts que pueden ser usados con objetivos de pruebas, pero está todavía un poco lejos de un Functional Test Development Environment (FTDE).

2.2.6 Story Driven Development

(*Esta sección ha sido extraída y traducida de* [36])

La primera vez que se escuchó hablar de Story Driven Development fue por Preston Mack en el año 2004. La premisa básica es que antes de escribir ningún código, el equipo escribe una historia (o una clara idea de los requisitos) y se desarrolla esta historia produciendo un "storytest" ejecutable.

Las "*historias*" consisten en cómo el usuario interactúa con el sistema. Tiene el efecto de centrar las pruebas en el nivel de negocio, en lugar de en el nivel de programación; además lo hace más sencillo de leer y entender para el usuario.
Una historia típica (o característica) se describe con una o más situaciones. Cada situación describe un punto de inicio, una acción y un resultado.

STDD hace posible formalizar la especificación del cliente en contratos legibles y ejecutables que los programadores tienen que obedecer si quieren conseguir un sistema en funcionamiento. STDD reúne a los clientes, desarrolladores y testers antes de que ningún código se haya escrito. Colaboran para identificar una parte específica de funcionalidad, o "*historia*" (story) con la que trabajar. Los clientes y testers especifican los criterios para validar que la historia funciona y crean un documento ejecutable al que puede acceder cualquiera del equipo. Hay un contrato específico para la aceptación, por tanto las discrepancias son resueltas rápidamente. Hay un contexto claro para los clientes y desarrolladores para tener una conversación y eliminar malentendidos. El riesgo de construir el sistema equivocado es mucho menor.

Para poder estimar una historia, los programadores tienen que entender qué quiere el experto del negocio. Esto lo aprenden con conversaciones cara a cara. El experto del negocio describe la característica, explica por qué es útil y da ejemplos, tal vez haciendo dibujos en una pizarra. Los programadores hacen preguntas, separan la característica en trozos (tareas más pequeñas), hacen más preguntas sobre estas tareas, y a veces proponen variaciones en la característica que pueda hacerla más simple de implementar. Los ejemplos son particularmente útiles porque ponen la conversación en un terreno en concreto, y hace que pequeños pero importantes detalles no se pierdan en discusiones abstractas e imprecisas generalidades.

Una vez que la iteración está planeada, es el trabajo de los programadores producir características que satisfagan al experto del negocio y es el trabajo del experto ayudarles, haciendo posible mantener otras conversaciones.

Estas "storytests" no tienen por qué reemplazar completamente a las pruebas convencionales. Están diseñadas para construir un producto de calidad. Todavía se

necesita a alguien más para crear pruebas con la finalidad de detectar dónde puede fallar y evitar que el sistema tenga errores.

Hacer que una tabla de “storytests” pase requiere dos pasos:

El primero es traducir la tabla en un programa ejecutable. Normalmente no dirigen la interfaz gráfica. En lugar de eso va por detrás de la GUI y hace llamadas a la lógica de negocio del programa de la misma forma que la GUI lo haría. Esto hace que las pruebas sean más fáciles de mantener, y también ayuda a alinear el lenguaje del programa con el lenguaje de negocio. Este estilo de programación se llama “*domain-driven design*” [24].

El segundo paso es hacer pasar las pruebas. En algunos casos es sencillo, el programador ejecuta la tabla, ve si alguna fila falla, en tal caso cambia el código, ejecuta la tabla otra vez y ve que la fila ahora pasa. Este proceso continua hasta que todas las pruebas de la historia pasan. Hay que darse cuenta que estas pruebas pueden cambiar con el tiempo antes del final. Cualquiera de los expertos del negocio, testers o programadores pueden añadir o cambiar una prueba si ayuda a completar una historia con más seguridad, más rápido o sin muchas complicaciones. Sin embargo, el experto es el árbitro final, que dirá si una prueba es un ejemplo correcto de característica. Cuando todas las pruebas pasan, la historia está hecha y se puede empezar una nueva.

Así pues, a grandes rasgos los pasos para realizar STDD son:

1. Definir las Historias.
2. Definir Pruebas de Historias (“storytests”).
3. Integración de Pruebas de Historias.
4. Programar las Pruebas de Historias.

Una vez que los desarrolladores tienen una prueba de historia, empezarán a escribir código de prueba para mostrar que la prueba de historias falla. Si el equipo usa FIT [17], esto implica definir la subclase apropiada de FIT y hacer la producción de código para hacer pasar la prueba de historia. Durante este tiempo, es propio y óptimo hacer unit test-driven development en conjunto con STDD. Esto ayuda a hacer evolucionar la producción de código que tiene suficiente cobertura de pruebas. A lo largo de este proceso, los programadores integran código, pruebas unitarias y pruebas de historias conforme van pasando.

La herramienta más destacada de STDD es Rails [37].

2.2.7 Comparación / Conclusiones

En cuanto a las diferencias entre ATDD, TDR, FTDD, BDD, STDD y CTDD; por lo anteriormente expuesto podemos advertir que son muy similares. Todas tienen unas características esenciales en común:

- Usan la idea de prueba para obtener detalles de las explicaciones de los "*stakeholders*". Discutiendo las ideas de las pruebas proactivamente –como condiciones límite, configuraciones o diferentes secuencias de las acciones del usuario, etc.- podemos llegar a un entendimiento compartido de las expectativas reales del "*stakeholder*" del negocio para el sistema, en lugar de tener, posteriormente en el proceso, un archivo de pruebas con gran cantidad de errores.

- Transforman los criterios de aceptación en pruebas automatizables expresadas en lenguaje natural en lugar de en lenguajes de programación. Esto nos permite separar completamente la base de las expectativas de cualquier detalle o dependencia técnica; es decir, ayudan a escribir las pruebas con un enfoque más funcional y, por tanto, ayudan a centrarse en las necesidades del cliente.

- Escriben "*fixtures*"[1] o bibliotecas para conectar las palabras claves en las pruebas al software en desarrollo durante la implementación. Hablamos de conectar, no de traducir. Haciendo esto como parte del esfuerzo de implementación. Por lo tanto no necesitamos actualizar la automatización una vez que el código esté escrito.

En todo caso una diferencia que podríamos notar entre BDD y los demás enfoques, es que las herramientas del primero están pensadas más para el uso del desarrollador, mientras que en el resto están más enfocadas al analista del negocio: BDD ofrece más facilidades para especificar el comportamiento del sistema mediante el uso de plantillas y criterios de éxito. Pese a eso, BDD es sólo una variante, los principios básicos son los mismos y las herramientas de BDD son sólo otra forma de automatizar pruebas.

Ninguno de los enfoques estudiados está definido en el marco de una metodología de desarrollo. Cuando esto se hiciera debería resolverse cómo hacer factible que todo el proceso de desarrollo gire en torno a la interacción entre el cliente y el desarrollador, para que este último, prueba a prueba, vaya implementando inmediatamente el comportamiento asociado a la prueba. Esto es difícil de llevar a la práctica cuando

[1] Un "fixture" es un conjunto de datos conocidos (o comandos para preparar los datos) que provee el entorno para un conjunto de prueba. Los "fixtures" funcionan bien cuando se tiene una gran cantidad de pruebas que trabajan con datos similares, reduciendo la complejidad del entorno de pruebas.

intervienen varios clientes y/o desarrolladores y cuando por envergadura del sistema se requiere mayor especialización en cuanto a roles. Por otra parte, normalmente para la automatización de una prueba de aceptación se necesita que la funcionalidad que se quiere probar se haya implementado antes. Nuestro enfoque TDRE no obliga a instanciar las pruebas cuando se definen los requisitos ya que deja en manos de los testers el determinar las mejores combinaciones de datos y el decidir si es conveniente (y posible respecto a los recurso de testeo) la automatización de la prueba dentro de la misma iteración en la cual se incorpora el cambio asociado a la prueba.

2.2.8 Herramientas

A partir de ahora usaremos FTDD para hacer referencia a cualquiera de los enfoques ATDD, STDD, CTDD, ya que como hemos mencionado son lo suficiente similares como para tratarlos por igual.

Además de disciplina y habilidad para llevar FTDD a cabo, el apoyo de herramientas también es esencial para alcanzar el éxito a largo plazo. La mayoría de proyectos no tienen el tiempo ni el dinero para incluir la creación de herramientas en el presupuesto, así que comprometen sus estándares dejando que la herramienta que tienen dicte su enfoque y limite su solución.

Para que FTDD y otros enfoques tengan éxito a largo plazo, necesitamos una nueva generación de herramientas de pruebas de aceptación, que ayudarán a los equipos a darse cuenta de su verdadero potencial y efectividad.

Buscando en Internet se encuentran un número y variedad significativa de herramientas para pruebas de aceptación automatizadas.

Pero, ¿cómo de bien apoyan el proceso de TDD?

- Las herramientas de pruebas de grabado/reproducción preceden a FTDD y fueron construidas para soportar pruebas automáticas después del desarrollo. Scripts de pruebas, generados por la herramienta mientras grababa un conjunto de acciones de usuario, tienden a ser difíciles de leer y mantener.

- La introducción de TDD lanzó una nueva variedad de herramientas de pruebas, conocidas colectivamente como xUnit [9] (por ejemplo, SUnit [38] para Smalltalk, JUnit [10] para Java y NUnit [39] para .Net) [Tabla 1]. Estos frameworks soportan pruebas escritas por los desarrolladores en un lenguaje de programación existente, usando entornos de desarrollo integrados (IDEs) también existentes. Con el tiempo los desarrolladores escribieron otras

extensiones (como HttpUnit [40] y JwebUnit [41]) para soportar las pruebas de interfaces basadas en la Web. Aunque estas herramientas son idóneas para pruebas unitarias, carecen de la legibilidad (no se debe de necesitar un técnico experto en la materia para poder validar la corrección y completitud de las pruebas de aceptación) y la localidad (todas las pruebas de aceptación relevantes deben de encontrarse y actualizarse antes de que el código lo haga) requerida para las prueba de aceptación. Además, aunque existe una herramienta para el TDD en casi todos los lenguajes de programación imaginables, estas herramientas se centran sólo en el apoyo a la creación de pruebas automáticas. El propósito de dichas pruebas es validar que una unidad o módulo del código funcione correctamente.

C++	cppUnit
.Net	csUnit
C	CUnit
Borland Delphi	DUnitDelphi
JUnit (extensión para proyectos de base de datos)	DBUnit
Java	JUnit
.Net librería para proyectos de base de datos	NDbUnit
Oracle Unit Tester	OUnit
PHP	PHPUnit
Python	PyUnit
.Net	NUnit
Ruby	Test::Unit (Incluido con Ruby en Rails)
Visual Basic	VBUnit

Tabla 1. Algunos ejemplos de herramientas para TDD en diferentes lenguajes de programación

- Las herramientas de frameworks para Pruebas Integradas, o Fit [17], revolucionaron FTDD ya que proveen de pruebas de aceptación tabulares, expresivas y legibles y de presentación de informes detallados, visuales y sensibles al contexto. Fit tiene gran flexibilidad, permite separar las pruebas del código que las ejecuta (fixtures). FitLibrary enriquece aun más la especificación del dominio de negocio y el workflow. El Framework FitNesse [42] lleva las pruebas de aceptación un paso más cerca de los expertos en la materia, ya que permite ejecutar las pruebas de Fit en navegadores estándar. Aunque es un progreso, estas herramientas son difíciles de escribir y mantener porque no existen entornos de desarrollo de pruebas de aceptación para especificaciones tabulares. Veremos un ejemplo de FIT en 2.2.8.1 FIT/FitNesse.

- Muchas otras herramientas introducen diferentes capacidades y lenguajes de pruebas (por ejemplo, en WebTests [43] las pruebas se especifican en XML y en Watir [44] se especifican en Ruby), pero no avanzan en el estado del arte mucho más allá de lo que lo hacen los frameworks de xUnit.

- Behaviour-driven Development ha sido un paso adelante significativo, cambiando el enfoque a una especificación orientada al comportamiento del negocio, en lugar de a las pruebas. El énfasis está en establecer un dominio de lenguaje específico (DSL) [29] no ambiguo, resultando en especificaciones declarativas. Mientras herramientas de BDD como jbehave [25] y rspec [26] llevan el estado del arte en otra dirección, debemos continuar avanzando y añadir más potencia a este tipo de herramientas.

Todas estas herramientas resultan útiles cuando es la misma persona la que se encarga de definir las pruebas con el cliente, programarlas y automatizarlas, ya que al automatizar una prueba, además de conocer el domino del problema, se requiere tener conocimiento de detalles internos de cómo está implementado el producto. Esto es un obstáculo en un contexto industria y colaborativo, donde el trabajo está especializado en diferentes roles, y no todos tienen por qué conocer el código del producto. En TDRE las PAs actúan como elemento integrador del trabajo de los diferentes roles: el analista identifica y define las PAs, el programador debe implementar comportamiento para satisfacer las PAs y el testar debe preparar datos y ejecutar las PAs para validar el comportamiento del producto.

Además, ni estos enfoques ni sus herramientas abordan la problemática de gestionar la gran cantidad de pruebas que se generan. Es esencial disponer de mecanismos para manipular de forma ágil las PAs, particularmente en un contexto de mantenimiento continuo de un producto. En TDRE se ofrecen mecanismos que abordan esta necesidad.

2.2.8.1 FIT/FitNesse

Como hemos constatado que los enfoques anteriores son muy similares y dado que varios de ellos promueven FitNesse como herramienta para su desarrollo, dedicaremos esta sección a entrar algo más en profundidad en FIT o FitNesse viendo un ejemplo de cómo funciona.

Con FIT los casos de prueba consisten en pasos que se presentan en formato tabular. Los desarrolladores tienen que implementar código para las pruebas en cada tipo de paso.

Una vez hemos descargado FitNesse[2], lo ejecutamos para tener el servidor en marcha usando el comando de java (es necesario tener instalada previamente la JVM 1.6[3] o posterior):

```
java -jar fitnesse.jar -p 8080
```

Hecho esto, si abrimos cualquier navegador web y vamos a la dirección http://localhost:8080/ estaremos en la página inicial de FitNesse (Figura 10).

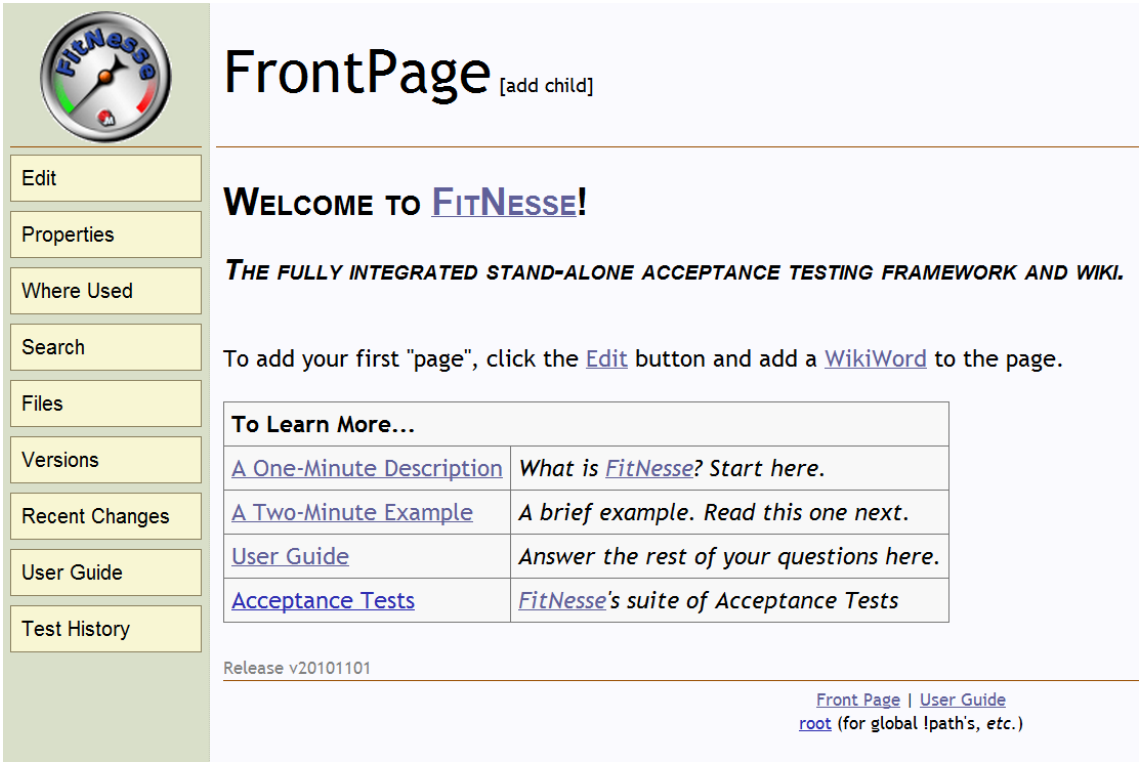


Figura 10. Páginia inicial de FitNesse

Par hacer nuestra primera página de prueba, en el menú de la izquierda seleccionaremos “Edit” y sobre la línea “!note Release v20101101” (puede cambiar en función de la versión que hayamos descargado) escribimos el nombre de nuestra prueba, precedido del símbolo mayor que (>) y siguiendo un código consistente en el nombre que tiene que estar escrito con al menos dos mayúsculas y no pueden ser consecutivas. Por ejemplo:
>DigitalVideoRecorder

[2] Podemos hacerlo de la siguiente dirección web:
http://fitnesse.org/FrontPage.FitNesseDevelopment.DownLoad

[3] Se puede obtener en https://www.java.com/

Al darle a “Save” veremos que en la página inicial ha aparecido lo siguiente:

Figura 11. Crear página en FitNesse

Si pulsamos sobre el signo de interrogación (?) se creará automáticamente una nueva página en blanco, a la que podemos referenciar en cualquier momento desde cualquier página escribiendo sólo su nombre, en nuestro caso “DigitalVideoRecorder”.

Si borramos el contenido que aparece en la nueva página y escribimos lo siguiente:

```
!define TEST_SYSTEM {slim}
!define COLLAPSE_SETUP {true}
!define COLLAPSE_TEARDOWN {true}

!|Crear programa                                                    |
|Nombre |Canal|DiaSemana|Hora     |DuracionEnMinutos|id?  |
|House  |4    |Lunes    |19:00    |60               |$ID=|
```

Veremos esto:

variable defined: TEST_SYSTEM=slim

variable defined: COLLAPSE_SETUP=true
variable defined: COLLAPSE_TEARDOWN=true

Crear programa					
Nombre	Canal	DiaSemana	Hora	DuracionEnMinutos	id?
House	4	Lunes	19:00	60	$ID=

Figura 12. Crear prueba en FitNesse

No es necesario que las barras (|) estén en línea, es sólo para que quede más claro.
Lo que hemos introducido es una tabla de decisión o tabla de verdad.

El comando “!define TEST_SYSTEM {slim} “ hace que se active el kernel de Slim (Simple List Invocation Method) para hacer la prueba. Si se desea se puede especificar “fit” entre corchetes. La ventaja que tiene Slim es que en lugar de ejecutar todo el procesamiento del HTML, comparaciones, etc, Slim mantiene todo el comportamiento

en FitNesse. Usar SUT (System Under Test) le permite a Slim invocar el método directamente en el SUT evitando tener que crear un método en un fixture.

Todavía no tenemos hecha nuestra prueba. Para concluir pulsamos en “Properties” del menú de la izquierda, y a continuación marcamos bajo tipo de página “Test” y pulsamos “Save” de nuevo. Si creamos las páginas con el nombre comenzando o terminando en “Test”, FitNesse creará automáticamente la página como tipo prueba.

Ahora en el menú de la izquierda debe haber aparecido la opción “Test”.

Sólo falta incluir la lógica. Elegiremos para eso el lenguaje de programación C#, por ser el mismo lenguaje que hemos empleado para desarrollar nuestro módulo; no obstante también se podría desarrollar con Java y algún entorno como Eclipse.
Abrimos el Visual Studio y creamos un proyecto de C# vacío. Llamaremos al namespace DigitalVideoRecorder y renombramos la clase Class1.cs que aparece por defecto a CrearPrograma.cs. Borramos todas las sentencias “using” que Visual ha creado automáticamente en la clase, pegamos el siguiente código y compilamos.

```
using System;
namespace DigitalVideoRecorder
{
  public class CreatePrograms
  {
    private string _name;
    private int _channel;
    public void setName(String name){
      _name = name;
    }

    public void setChannel(int channel){
      _channel = channel;
    }

    public void setDayOfWeek(String dayOfWeek){
    }

    public void setTimeOfDay(String timeOfDay){
    }

    public void setDurationInMinutes(int durationInMinutes){
    }

    public string id(){
      return String.Format("[{0}:{1}]", _name, _channel);
    }
  }
}
```

Volvemos a editar la página de prueba y añadimos las siguientes líneas al comienzo de la página:

```
!define TEST_SYSTEM {slim}
!define COMMAND_PATTERN {%m -r
fitSharp.Slim.Service.Runner,c:\tools\nslim\fitsharp.dll %p}
!define TEST_RUNNER {c:\tools\nslim\Runner.exe}
```

Sobre la tabla que teníamos importamos el namespace añadiendo la siguiente línea:

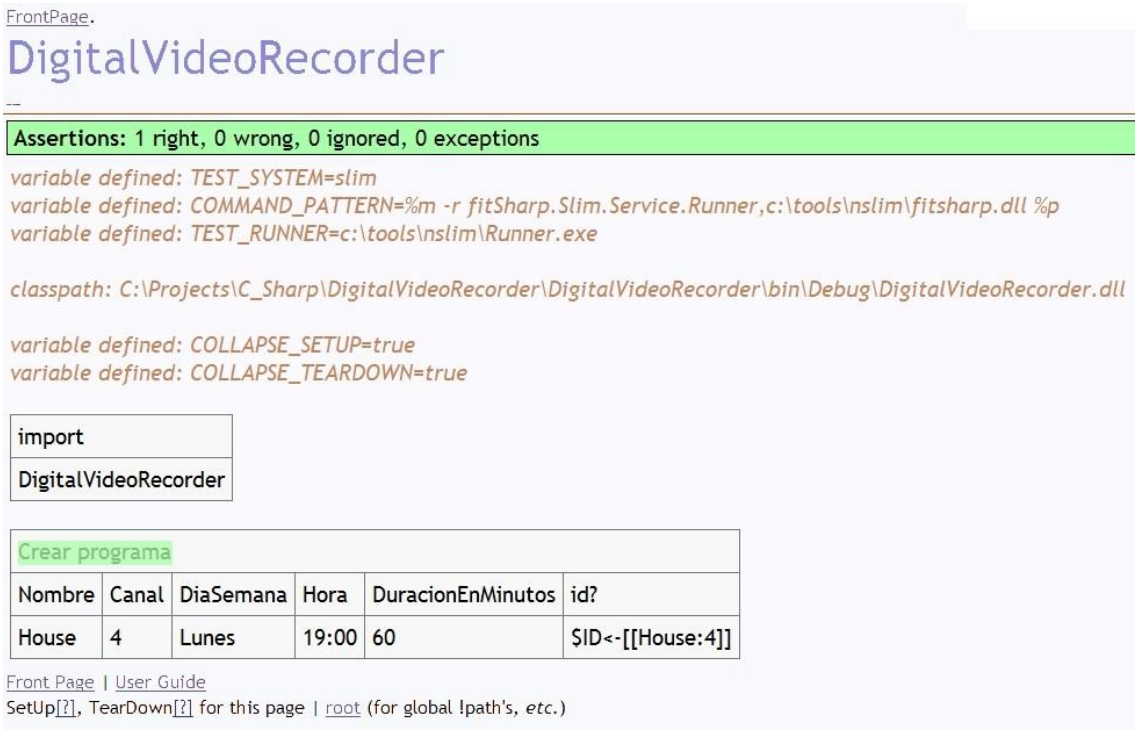
```
!|import|
|DigitalVideoRecorder|
```

Y finalmente añadimos el path donde está la dll de nuestro proyecto. Generalmente lo encontramos dentro de la carpeta \bin\Debug donde creamos el proyecto.

En conclusión, la página debería quedar como esto (exceptuando la ruta donde se enuentra el dll del proyecto):

```
!define TEST_SYSTEM {slim}
!define COMMAND_PATTERN {%m -r
fitSharp.Slim.Service.Runner,c:\tools\nslim\fitsharp.dll %p}
!define TEST_RUNNER {c:\tools\nslim\Runner.exe}

!path
C:\Projects\C_Sharp\DigitalVideoRecorder\DigitalVideoRecorder\bin\Debug\DigitalVide
oRecorder.dll

!define COLLAPSE_SETUP {true}
!define COLLAPSE_TEARDOWN {true}

!|import|
|DigitalVideoRecorder|

!|Crear programa                          |
|Nombre |Canal|DiaSemana|Hora|DuracionEnMinutos|id? |
|House|4     |Lunes   |19:00   |60            |$ID=|
```

Si guardamos los cambios y le damos a “Test”, veremos que la prueba pasa satisfactoriamente (Figura 13).


FrontPage.

DigitalVideoRecorder

--

Assertions: 1 right, 0 wrong, 0 ignored, 0 exceptions

variable defined: TEST_SYSTEM=slim
variable defined: COMMAND_PATTERN=%m -r fitSharp.Slim.Service.Runner,c:\tools\nslim\fitsharp.dll %p
variable defined: TEST_RUNNER=c:\tools\nslim\Runner.exe

classpath: C:\Projects\C_Sharp\DigitalVideoRecorder\DigitalVideoRecorder\bin\Debug\DigitalVideoRecorder.dll

variable defined: COLLAPSE_SETUP=true
variable defined: COLLAPSE_TEARDOWN=true

import
DigitalVideoRecorder

Crear programa					
Nombre	Canal	DiaSemana	Hora	DuracionEnMinutos	id?
House	4	Lunes	19:00	60	$ID<-[[House:4]]

Front Page | User Guide
SetUp[?], TearDown[?] for this page | root (for global !path's, *etc.*)

Figura 13. Test con FitNesse

Como hemos visto, FitNesse sirve para definir Pruebas de Aceptación, las cuales, para FIT, son páginas web que contienen tablas simples de entradas y salidas esperadas. FitNesse te permite ejecutar estas pruebas y comprobar si los resultados son correctos.

De esta manera, FitNesse evita el uso del Microsoft Word para las Pruebas de Aceptación, haciendo que éstas sean creadas, editadas y ejecutadas en entornos al estilo Wiki[4] como hemos podido ver. Aunque el hecho de evitar las páginas del Word para obtener mayor expresividad también era uno de nuestros objetivos, FitNesse carece de todas las demás funcionalidades de nuestro Gestor de Requisitos. No siendo así capaz de indicar incidencias, comentarios, ni tiene relación alguna con los requisitos, ni forma clara de organizar o gestionar los mismos.

De alguna manera tal vez se podrían tratar como requisitos a las SubWikis o a los “suits of tests”, usándolos como paquetes de pruebas de aceptación, aunque su objetivo inicial no es ese, sino el de ejecutar todos los tests incluidos en el “test suit” a la vez. Aun así, tendría una estructura jerárquica, en lugar de la estructura en forma de grafo más potente de la que provee nuestro gestor de requisitos. Y tratando como podríamos tratar con miles de pruebas, diferentes versiones… sería inviable usar el formato de la wiki, pues al no tener un árbol donde ver claramente toda la jerarquía nos perderíamos entre las páginas.

[4] Wiki es un estilo de servidor web que permite a los usuarios crear y editar libremente contenido web usando un navegador web. Wiki soporta hiperenlaces y tiene una sintaxis simple de texto para crear rápidamente nuevas páginas y enlaces entre páginas internas.

Realmente no hay manera de comparar nuestra propuesta de módulo con FitNesse, pues están destinados a diferentes usos. Como bien puntúan en la página oficial de FitNesse, ésta se trata de una herramienta de pruebas de software, una aplicación para probar la capa de negocio de los programas; mientras que nuestro módulo se basa principalmente en la gestión de requisitos software.

FitNesse provee de métodos para determinar automáticamente si una aplicación funciona correctamente, pero por ejemplo, no entra en temas de interface, cosa que nuestro Gestor de Requisitos sí hace.

Además, FitNesse tiene todas las desventajas de una aplicación web frente a una de escritorio, es muy limitada en cuanto a lo que pueden hacer, y no muy intuitiva. Tiene variedad de opciones, pero no son visibles. Consisten en pasar determinados parámetros por la URL, aunque es necesario conocer esos comandos de antemano.

Inicialmente parece sencillo, pero según se profundiza en FitNessese se observa que aparecen muchas variables a tener en cuenta: como estilos de tablas, tipos de fixture, la depuración de las fixture, la selección del ClassPath… Es por ello que el tiempo de aprendizaje inicial de FitNesse es grande comparado con nuestro módulo. Y no habría forma de que los requisitos fueran escritos por el cliente.

2.2.8.2 Rational RequisitePro

Antes de comenzar con el desarrollo del Gestor de Requisitos, se planteó si era viable usar alguna herramienta ya existente para aplicar el enfoque propuesto de Test Drive Requirement Engineering y así ahorrar trabajo.

La herramienta escogida fue RequisitePro [45], los motivos son claros: es una de las herramientas de requisitos tradicionales (como bien indica su nombre) y es una de las destacadas por la encuesta de herramientas de gestión realizada por INCOSE (International Council on System Engineering) [46].

Según la página oficial[5] de RequisitePro:

“Rational RequisitePro ayuda al equipo a gestionar sus requisitos, escribir buenos casos de uso, mejorar la trazabilidad, fortalecer la colaboración, reducir la duplicidad de trabajo e incrementar la calidad”.

Dicho esto, para tratar de usar TDRE con RequistePro primero creamos un nuevo proyecto vacío.

En el RequisitePro no hay otro concepto que el de “requisito”, así que tendremos que definir nuestros conceptos bajo ese nombre: En la barra de herramientas vamos a “Requirement” y seleccionamos “Properties”. Esto nos abrirá las propiedades del

[5] http://www-01.ibm.com/software/awdtools/reqpro/

proyecto recientemente creado. Le damos a “Add” y añadimos un “requisito” llamado “Nodo”, otro llamado “Prueba de Aceptación”, otro llamado “Interfaz de Usuario” y uno llamado “Prueba de Sistema”. Debería quedar como en la Figura 14.

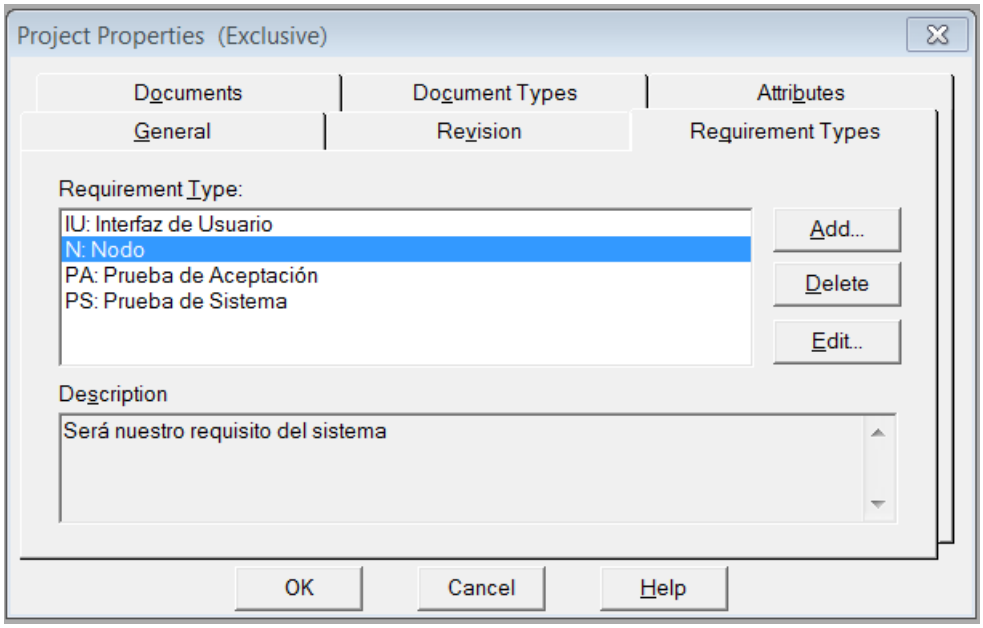


Figura 14. Creación de tipo de requisitos en RequisitePro

Bien, ahora vamos a definir los atributos de éstos. Vamos a la pestaña “Attributes”, y borramos todos los que aparecen por defecto para cada uno de los requisitos que acabamos de crear.

Para el requisito “Nodo” añadimos los atributos “Tipo”, que será de tipo “List (Single Value)”, y contendrá las cadenas: “Requisito Funcional”, “Requisito No Funcional” e “Interfaz de Usuario” (ver Figura 15); y el atributo “Activo”, cuyos valores serán “Si” y “No”.

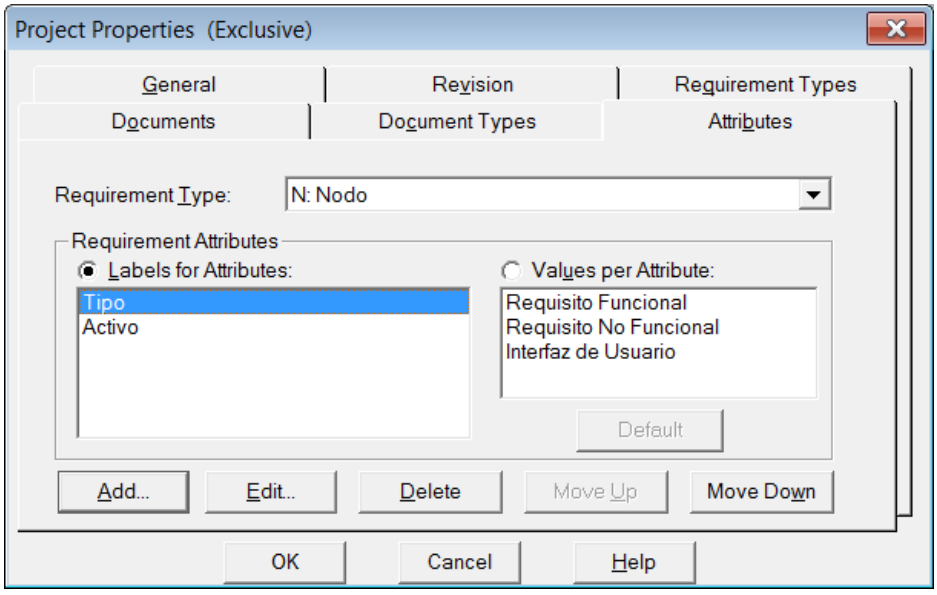


Figura 15. Asignando atributos en RequisitePro

En el requisito “Prueba de Aceptación” añadiremos el atributo “nroOrden”, de tipo Integer.

Vamos ahora a la pestaña “Document Types”. En nuestro caso tendremos 2 tipos de documentos, los que describan las PAs y los que contengan las capturas de las IU, así que procedemos a crear nuevos tipos de documentos asociados tanto a los requisitos antes creados de Prueba de Aceptación como a los de Interfaz de Usuario.

Quedaría como en la Figura 16:

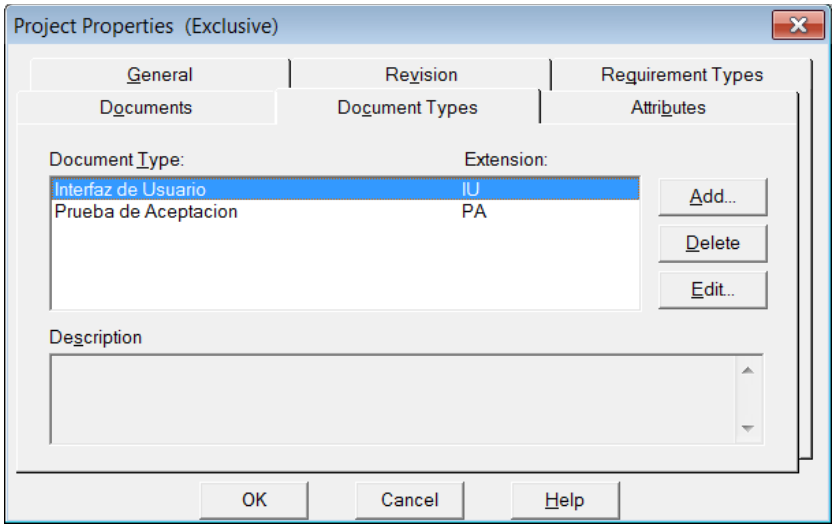


Figura 16. Creando tipos de documentos en RequisitePro

Pulsamos OK y con esto ya podemos empezar a construir nuestra jerarquía de requisitos; decimos jerarquía porque en RequisitePro no hay manera de hacer un grafo, no permite la recursividad ni la duplicidad de nodos.

En aras de organizar el proyecto, haremos clic derecho sobre la raíz del árbol y seleccionaremos “Package”, crearemos 4 paquetes, llamados: “Documentos”, “Nodos”, “PAs” e “IUs”.

Botón derecho sobre la carpeta “Nodos”, elegimos “New” y luego “Requierement”. De la ventana emergente, elegimos tipo “Nodo”, ponemos un nombre a nuestro nodo y le damos una descripción. A continuación vamos a la pestaña “Attributes” y especificamos el tipo y si está activo o no.

Creamos de esta forma todos los nodos que hagan falta.

En caso de tener los permisos suficientes se pueden mover los nodos por el árbol. Para eso hacemos clic derecho sobre un nodo, elegimos “Change Parent” y de la lista desplegable el nuevo nodo del que queremos que cuelgue (Figura 17).

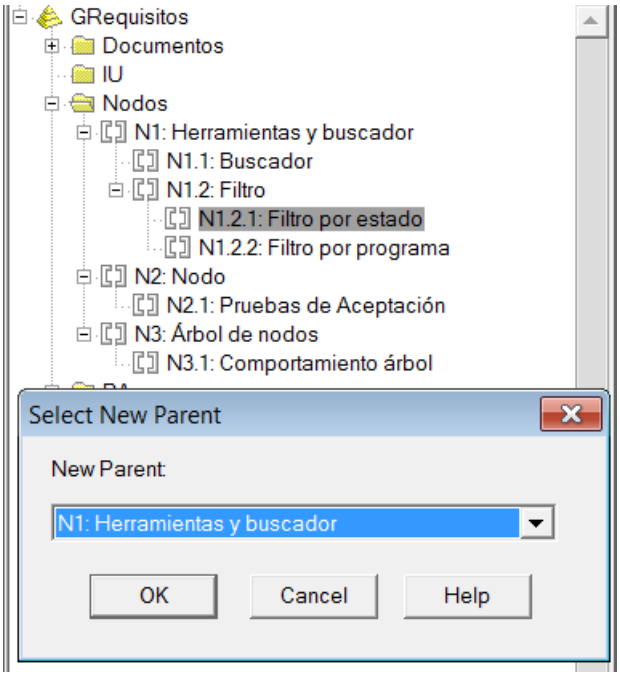


Figura 17. Mover nodos en RequisitePro

Ahora vamos a añadir unas pruebas de aceptación. Sobre la carpeta "Documentos" que hemos creado previamente hacemos botón derecho y seleccionamos "New"/ "Document". Le damos un nombre y una descripción que queramos a la prueba y en "Document Type" elegimos "Prueba de Aceptación". Pulsamos OK y se nos abrirá el Microsoft Word en la edición que tengamos instalada[6] con una barra de herramientas extra llamada "Complementos". Ahí están todas las posibilidades que ofrece RequisitePro para trabajar con documentos Word. De este menú elegiremos "New Requirement", en tipo elegimos "Prueba de Aceptación", le damos un nombre, una descripción, elegimos un número de orden si hiciera falta en los atributos y le damos a "OK". Guardamos el documento. Si volvemos ahora al RequisitePro veremos que se nos ha creado un nuevo requisito marcado como PA y el nombre que le dimos. Para mantener el orden moveremos este nuevo requisito a la carpeta PAs.

Existen 2 maneras de organizar las PAs. Una, la que hemos explicado en este último párrafo, consiste en tener una PA por cada documento de Word. La otra posibilidad es, al crear el documento, asociarlo con el tipo "Nodo", en lugar del tipo "PA", de esta manera los documentos equivaldrán a los Nodos, y dentro de cada documento pondremos todas las PAs de ese Nodo.

Una vez se ha aprendido a crear PAs, veremos como asociarlas a cada Nodo:
Hacemos doble clic sobre el Nodo en cuestión, eso abrirá sus propiedades. Nos movemos a la pestaña "Traceability" y pulsamos "Add" al lado de "To". Del primer desplegable elegimos "Prueba de Aceptación", y de la lista elegimos las pruebas que formarán parte de este nodo, para eso, podemos usar la tecla "shift" para seleccionar

[6] Versiones anteriores de RequistePro no operan con las últimas versiones de Microsoft Word, perdiendo todas las ventajas que puedan ofrecer. Si queremos poder operar sobre Microsoft Word 2007 o 2010 hay que adquirir una licencia de Rational RequisitePro 7.1.2 o posterior.

varias PAs a la vez, o bien utilizar la herramienta de búsqueda para en el caso que tuviéramos una lista muy grande de PAs encontrarlo más rápido (Figura 18).

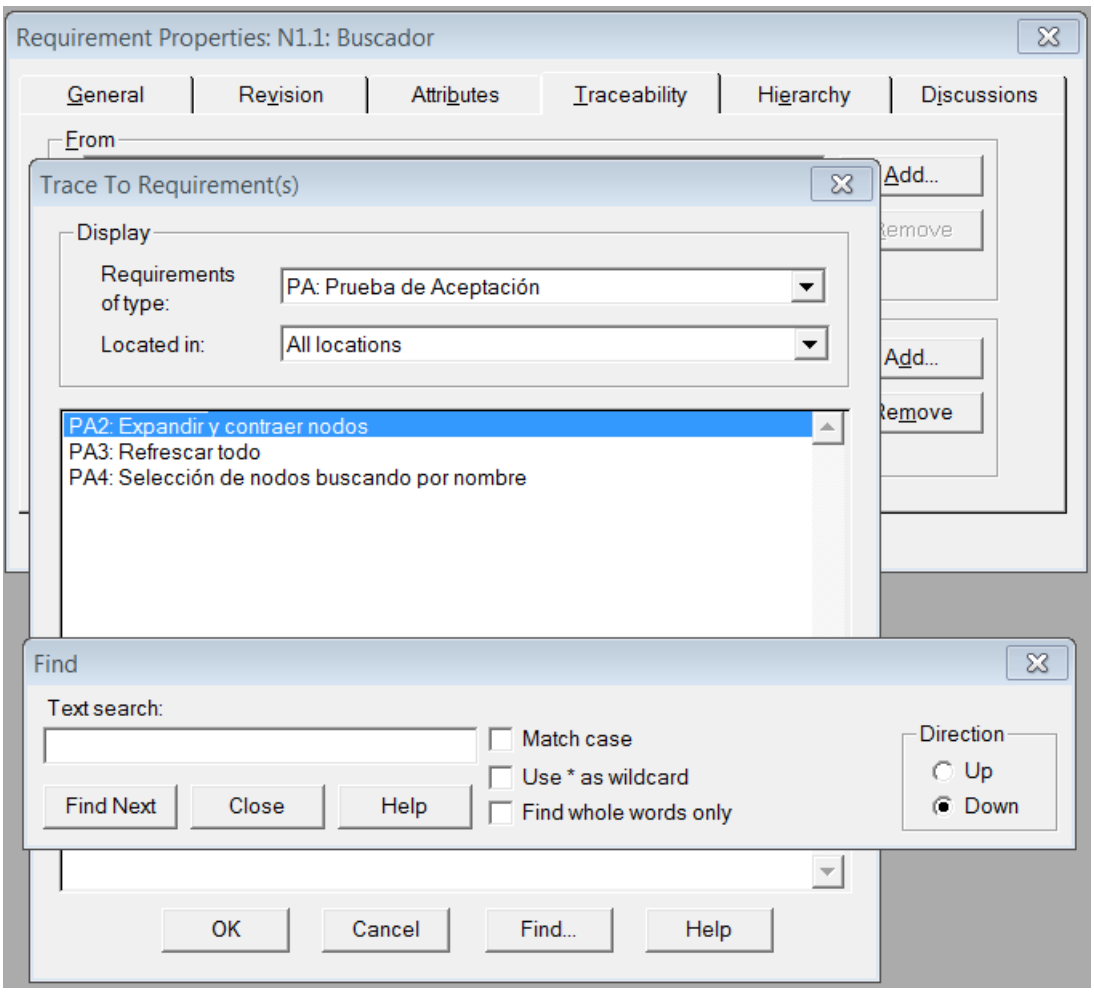


Figura 18. Asignar PA a Nodos en RequisitePro

Pulsamos OK y volvemos a la ventana anterior.

Cabe hacer notar que si vamos a la pestaña “Hierarchy” podemos cambiar la jerarquía del nodo en el que estamos, cambiando su padre y/o sus hijos.

Pulsamos OK otra vez y se guardan los cambios.

Si nos fijamos en el árbol de la izquierda, la relación que acabamos de hacer entre PAs y Nodo no queda reflejada. Para eso, hacemos botón derecho sobre el nombre del proyecto y seleccionamos “New / View”. Como nombre ponemos “Nodo – PA”, en “View Type” elegimos “Traceabilty Tree (Traced out of)” y en “Row Requirement Type” elegimos Nodo (Figura 19).

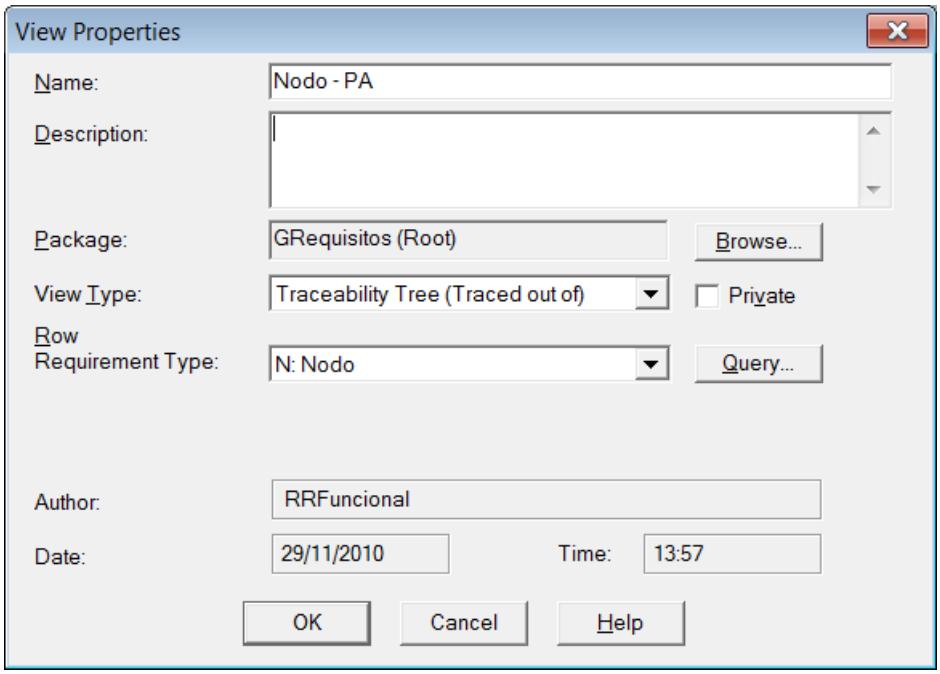


Figura 19. Trazabilidad en RequistePro

Pulsamos OK y vemos que aparece una nueva opción en el árbol. Si hacemos doble clic sobre el nuevo nodo "Nodo – PA" podemos ver que se nos abre a la derecha la jerarquía, esta vez sí, con la relación entre los nodos y las PAs (Figura 20).

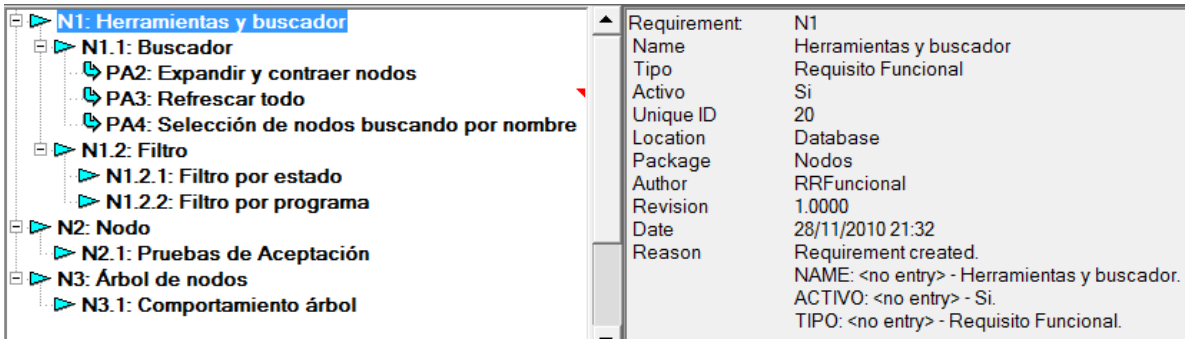


Figura 20. Árbol de nodos en RequisitePro

Si seleccionamos cualquiera de los Nodos o PA, a la derecha veremos sus detalles, y al seleccionar una PA en la parte inferior, veremos su descripción. Si hacemos doble clic sobre una PA en esta misma lista, se abrirá el documento donde está almacenada la PA para editarlo.

Otra de las funciones que debería tener un sistema que soportara TDRE es el de los comentarios. Es importante que los usuarios puedan comunicarse entre sí mediante algún tipo de mensaje que el analista pueda recoger, analizar y tomar una decisión al respecto. Para eso RequisitePro tiene la posibilidad de iniciar "discusiones":

Elegimos el requisito que queremos comentar (ya sea nodo o prueba de aceptación) , hacemos clic con el botón derecho del ratón y del menú desplegable elegimos

"Discussion". Se abrirá una ventana como la mostrada en la Figura 21 donde podemos crear un nuevo comentario, el cual se indicará con el asunto, la persona que lo escribió y la fecha y hora. Además existe la posibilidad de contestar los mensajes.

Como vemos en ésta misma figura, en la parte trasera podemos ver el árbol de Nodos y que la "PA3: Refrescar todo" tiene una marca roja a la derecha. Esa marca indica que tiene un comentario asociado y al hacer clic se mostrará.

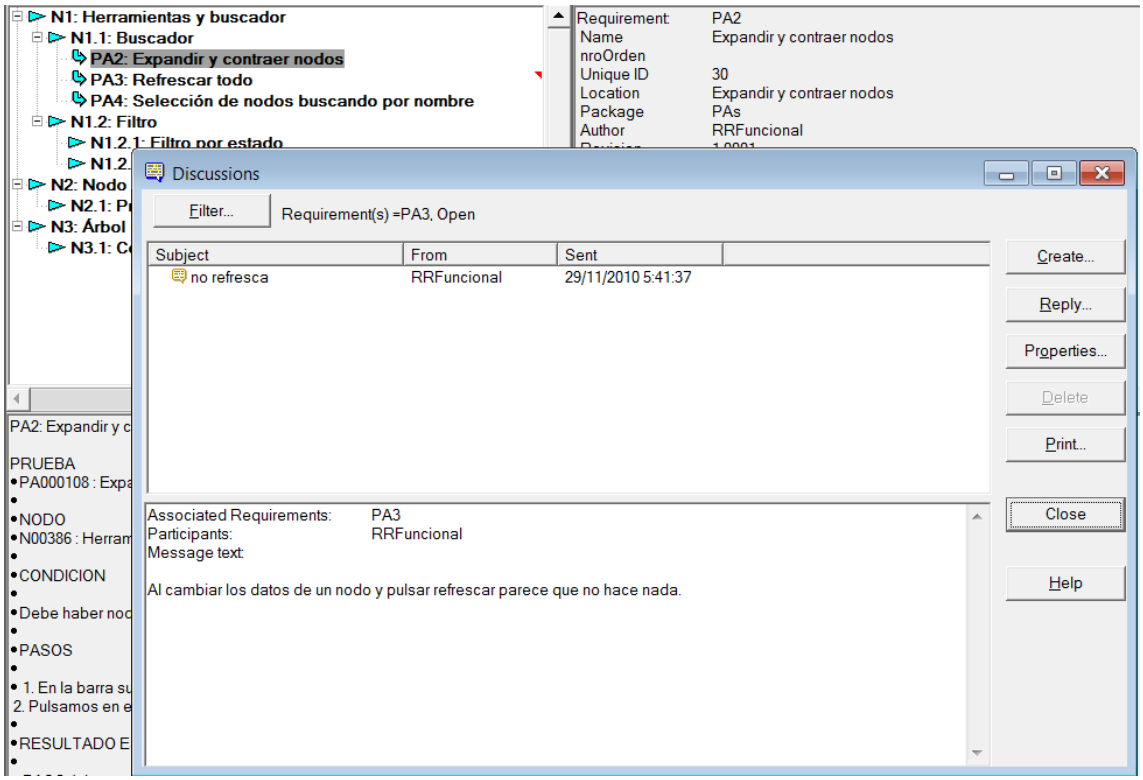


Figura 21. Comentarios en RequisitePro

Pero todos los usuarios no deberían poder realizar las mismas acciones. Sería caótico si cualquiera pudiera conectarse y borrar y crear nodos según quisiera. Para manejar esto tenemos los permisos.

Haremos clic con el botón derecho sobre el nombre del proyecto, y del menú desplegable elegiremos "Project Security". Es aquí donde podemos crear tantos tipos de usuario como queramos, con sus respectivos permisos. En nuestro caso, crearemos un usuario RRFuncional en el grupo "Administrators". Este usuario tiene permisos para todo por defecto.

A continuación creamos los grupos de usuario "Analistas" y "Otros", dentro de estos grupos basta que escribamos el nombre de usuario de cada uno de los que usarán la aplicación, junto con una contraseña.

Si seleccionamos en uno de los grupos creados, "Analistas", por ejemplo, y pinchamos en "Edit", podemos privarle de algunos permisos que supuestamente no debería tener; como la posibilidad de mover nodos en el árbol. Para eso desmarcaremos la

casilla de “Can manage Project structure”. Tampoco debe poder modificar datos de un nodo, así como añadir o borrar nodos, por lo que en la categoría de tipos de requisitos, elegimos “Nodo”, pulsamos “Edit” y desmarcamos “Update”. Con eso quitamos de golpe los permisos para hacer cualquier cambio en los nodos. En caso de requerir ser más precisos (por ejemplo, sólo impedir cambiar el estado del nodo), en la lista de atributos elegiríamos “Activo”, pulsaríamos “Edit Permissions” y desactivaríamos “Update”.

Para no extendernos más no explicaremos cómo deberían cambiarse los permisos de todos los usuarios, pero es bastante intuitivo, teniendo en cuenta lo anterior y la Tabla 2. Permisos por roles según contexto de la página 68.

Pese a que, como vemos, RequisitePro es bastante adaptable, cuando nos dispusimos a usarlo a fondo nos dimos cuenta que no tenía toda la capacidad que esperábamos. Entre otras cosas, al no manejar el concepto de Prueba de Aceptación, debido a que la granularidad en la que trabaja es a nivel de requisitos (los cuales no ofrecen flexibilidad con respecto a cambios en el producto software), no permite seguir el ciclo de vida de una PA ni facilita su visualización. Para eso requiere del grafo dirigido que hemos visto en la Figura 20.

Por si fuera poco, RequistePro está limitado a una jerarquía de 25 niveles de requisitos, no siendo recomendable, según el FAQ [47], más de 4 niveles por motivos de usabilidad. Esto nos podría suponer una gran limitación en cuanto a nuestra gestión de los requisitos.

Tampoco existe la idea de cambios en un producto ni de versiones. La única posibilidad sería crear paquetes dentro de cada proyecto para cada versión, con lo que nos encontraríamos en la misma situación que queríamos evitar en el contexto de nuestra empresa.

Otra de las razones que nos hizo reconsiderar elegir RequistePro como herramienta para aplicar TDRE fue que queríamos incorporarlo como un módulo en TUNE-UP (ver 3.1 Introducción a TUNE-UP y a TUNE-UP Process Tool). Para ello había que analizar en profundidad la API del RequisitePro lo cual, además de suponer un sobreesfuerzo, tenía algunos inconvenientes importantes al respecto [47]:

- La API de RequisitePro se puede usar para crear programas en C++, Java o VB para interactuar con los datos de RequisitePro. En nuestro caso, TUNE-UP estaba desarrollado con C#.
- No da la posibilidad de crear proyectos ¿Cómo podríamos entonces lanzar un nuevo proyecto desde TUNE-UP?
- No da la posibilidad de crear ni modificar documentos de RequisitePro.
- No permite llevar los documentos offline y guardarlos.
- No permite cambiar el número de versión de un requisito, eso agravaría el problema al tratar múltiples iteraciones y versiones.

- API no permitía modificar las propiedades de las vistas (que usamos como árboles para los nodos): como el ancho de las columnas, los atributos que se muestran o elegir entre mostrar trazabilidad directa o indirecta.

Además de ésas tenía más limitaciones, que en un principio no eran importantes, pero que en un futuro en que se pidieran mejoras/ampliaciones de la aplicación podrían suponer un problema.

Para más información acerca de cómo se puede realizar la integración con RequisitePro consultar [48].

Es por esto, y por lo anteriormente expuesto, que se echa en falta una herramienta que esté más enfocada en las pruebas, añadiendo una característica esencial que falta en el resto de herramientas, e incrementando de esta forma su potencial de uso. Por esa razón decidimos crear nuestro propio módulo que se integrara perfectamente con TUNE-UP.

En el siguiente capítulo introduciremos nuestra propuesta: un Gestor de Requisitos basado en pruebas de aceptación.

Capítulo 3. Gestión de Requisitos dirigida por pruebas

Dado que los requisitos, por muy claramente expresados que estén, pueden tener defectos, ¿cómo luchamos contra este problema antes del desarrollo en lugar de descubrirlo después? ¿Cómo podemos asegurar que los requisitos, independientemente de su forma y de si se elaboran gradualmente antes de cada iteración, son completos y correctos? Donald Gause y Gerald Weinberg escribieron en "*Exploring Requirements*" [49] que la forma más eficaz para comprobar los requisitos es usar casos de prueba muy parecidos a los usados para probar un sistema completo.

En esencia la idea es decidir cómo debería ser probado el sistema como una forma de comprobar si los requisitos nos dan suficiente información para construir el sistema.
En el tiempo de vida de un proyecto software, los requisitos están al principio mientras que las pruebas están tradicionalmente al final. Todas las actividades entre los dos hacen difícil ver la sutil relación entre estos conceptos. Ambos, realmente, hablan sobre lo mismo: cómo será usado el sistema una vez que esté desarrollado. De hecho, dado que las pruebas son muy precisas, ofrecen mucha menos posibilidad de malentendidos que los requisitos abstractos. Esto hace que las pruebas sean teóricamente una opción para reemplazar totalmente los requisitos. Robert C. Martin y Grigori Melnik además consideran que son lo mismo en su hipótesis de equivalencia [49]:

"A medida que aumenta la formalidad, las pruebas y requisitos se hacen indistinguibles entre sí. En el límite, pruebas y requisitos son equivalentes."

Habitualmente esta equivalencia no se reconoce debido a que los Requisitos y las Pruebas están establecidos como diferentes disciplinas.

Para identificar requisitos, los analistas del negocio normalmente trabajan junto a un determinado número de ejemplos realistas con el cliente.

Los requisitos y especificaciones abstractas dejan mucho espacio para la ambigüedad y los malentendidos. Para verificar o rechazar ideas sobre los requisitos, los desarrolladores normalmente recurren a los ejemplos e intentan poner las cosas en una perspectiva más concreta cuando se habla de casos excepcionales con los expertos del negocio o clientes. Los ejemplos reales y concretos nos proveen de una forma mucho más clara de explicar cómo funcionan realmente las cosas que los requisitos. Los ejemplos capturan un flujo de trabajo muy concreto, con entradas y salidas claramente definidas.

Ejemplos, requisitos y pruebas están esencialmente unidos y forman un bucle. La relación entre ellos se puede ver en la Figura 22.

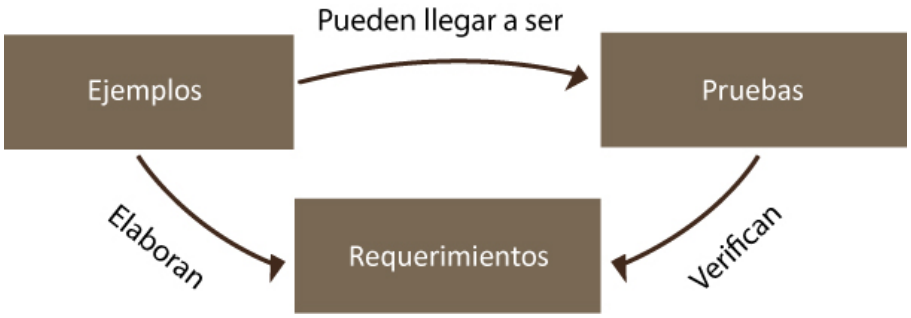


Figura 22. Relación entre pruebas, ejemplos y requisitos

Esta estrecha relación entre requisitos, pruebas y ejemplos indica que efectivamente tratan con conceptos relacionados.

Los requisitos son normalmente dirigidos por ejemplos, y los ejemplos acaban siendo pruebas. Con suficientes ejemplos, podemos construir una descripción completa del futuro sistema. Formalizando los ejemplos podemos obtener unos requisitos rigurosos para el sistema y un buen conjunto de pruebas. Si usamos los mismos ejemplos durante todo el proyecto, desde discusiones con el cliente y los expertos del dominio a las pruebas, entonces los desarrolladores o testers no tienen que usar sus propios ejemplos aislados.

Al final del proyecto todos los ejemplos deberían funcionar. Si idealmente tenemos un conjunto completo de ejemplos, y si todos los ejemplos funcionan, no hay nada más que debería ser desarrollado.

Tampoco aparecen requisitos abstractos y tenemos ejemplos para describir las excepciones. Cuando todos los ejemplos están implementados y el sistema funciona como lo describen, el trabajo está hecho.

Podríamos decir que los “ejemplos” de los que vengo hablando son al fin y al cabo nuestras Pruebas de Aceptación.

Una vez que el desarrollo está completado, empezarán a venir peticiones de cambios. Podemos usar el conjunto de pruebas de aceptación de fases previas del desarrollo como un documento relevante y fidedigno del sistema. Usamos las pruebas de aceptación para tratar las peticiones de cambios y descubrir rápidamente reglas y cambios en conflicto.

3.1 Introducción a TUNE-UP y a TUNE-UP Process Tool

(*Este subcapítulo ha sido extraído de* [50])

TUNE-UP es una metodología nacida en el trabajo día a día en una PYME de desarrollo de software y con una vocación de mejora continua del proceso. En tres años de aplicación y evolución, TUNE-UP ha conseguido la madurez suficiente para ser

presentada como una alternativa interesante, al menos para contextos de desarrollo de equipos pequeños. TUNE-UP incorpora elementos de metodologías ágiles y también del ámbito más tradicional. Una de las características clave de TUNE-UP es la planificación y seguimiento del proyecto centrada en la gestión de tiempos. TUNE-UP se inspira en la esencia de la propuesta de PSP (Personal Software Process), donde se destaca que la base del éxito radica en una disciplina de trabajo y productividad individual centrada en la gestión de los compromisos. TUNE-UP ayuda en cada momento del proyecto a responder a la pregunta: ¿conseguiré cumplir con los plazos de entrega de mis tareas? o ¿seremos capaces de cumplir con los plazos de entrega al cliente? Estas simples preguntas inquietan a cualquier gestor o participante de un proyecto. No contar con una respuesta acertada y sobre todo oportuna, implica en la mayoría de los casos graves complicaciones en el proyecto.

TUNE-UP es una metodología que incorpora aspectos ágiles y tradicionales con un sentido marcadamente pragmático. TUNE-UP se caracteriza fundamentalmente por combinar los siguientes elementos:

- **Modelo iterativo e incremental** para el desarrollo y mantenimiento del software. El trabajo se divide en unidades de trabajo que son asignadas a versiones del producto. Las versiones son frecuentes y de corta duración, entre 3 y 6 semanas dependiendo del producto.

- **Workflows flexibles** para la coordinación del trabajo asociado a cada unidad de trabajo. Los productos, según sus características, tienen asociados un conjunto de workflows que son utilizados para realizar cada unidad de trabajo. Cada una de éstas sigue el flujo de actividades del workflow y en cualquier instante se encuentra en un determinado estado en las actividades de dicho workflow. El estado puede ser: Por llegar, Pendiente, Activa, Pausada, Finalizada u Omitida. Bajo ciertas condiciones se permite saltar hacia adelante o hacia atrás en el workflow, así como cambios de agentes asignados e incluso cambio de workflow. Por ejemplo, las frecuentes situaciones de re-trabajo en desarrollo de software ocasionadas por detección de defectos se abordan con saltos atrás no explícitos en el workflow. Por ejemplo, desde la actividad de *Aplicar Pruebas de Sistema* hacia la actividad *Diseño e Implementación*, o hacia la actividad *Análisis*. La Figura 23 ilustra un workflow mínimo para el desarrollo de una unidad de trabajo.

- **Proceso de desarrollo dirigido por las pruebas** (Test-Driven). La definición de una unidad de trabajo es básicamente la especificación de sus pruebas de aceptación. A partir de ahí, todo gira en torno a ellas: se estima el esfuerzo de diseñar, implementar, y aplicar las pruebas, se diseñan e implementan y luego se aplican para garantizar el éxito de la implementación.

- **Planificación y seguimiento continuo**. En todo momento debe estar actualizado el estado de las versiones, de las unidades de trabajo, y del trabajo asignado a los agentes.

- **Control de tiempos**. Los agentes registran el tiempo que dedican a la realización de las actividades, el cual se compara con los tiempos estimados en cada una de ellas, detectando oportunamente desviaciones significativas. Esto permite a los agentes gestionar más efectivamente su tiempo, mejorar sus estimaciones y ofrecer al Product Manager información actualizada del estado de la versión.

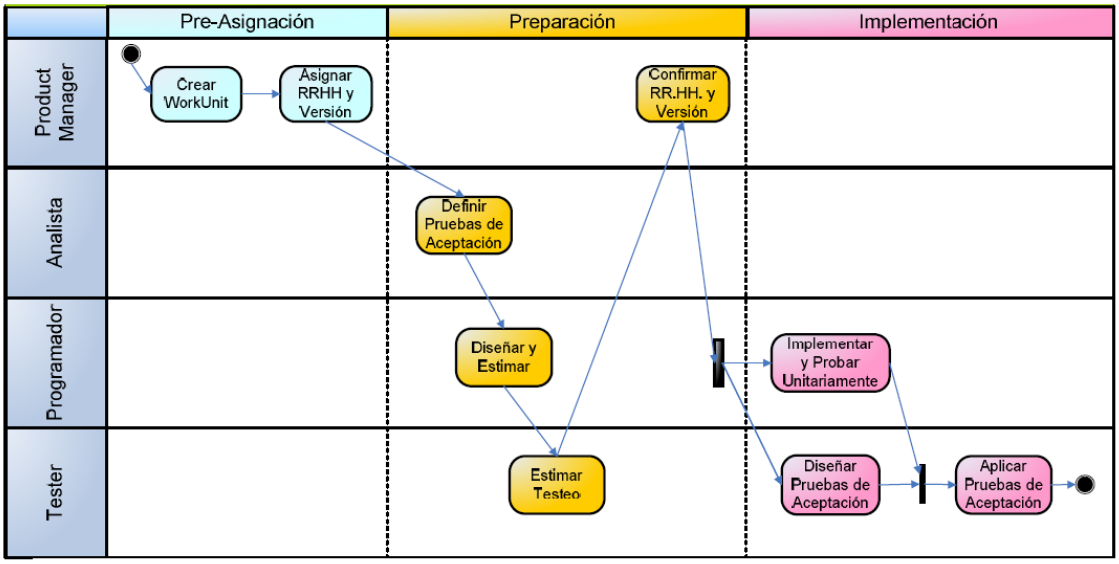


Figura 23. Workflow de desarrollo simple para unidades de trabajo.

Todas estas características están soportadas por TUNE-UP Process Tool, la cual está formada por tres módulos principales: Planificador Personal (gestión del trabajo individual de cada agente), Gestor de Unidades de Trabajo (soporte para la realización del trabajo de un agente en una unidad de trabajo) y Planificador de Versiones (planificación y seguimiento de las versiones de los productos).

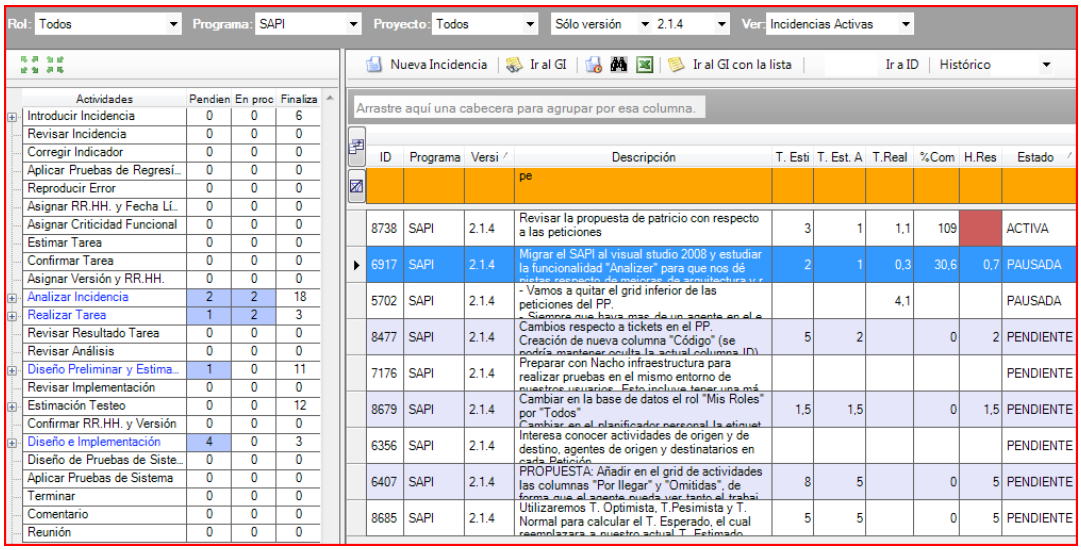


Figura 24. Planificador Personal

El **Planificador Personal** (PP) presenta la lista de unidades de trabajo que tiene asignadas un agente en una determinada actividad. El PP ofrece todos los recursos necesarios para que el agente consulte y decida qué unidad de trabajo debe ser atendida. En el PP se ofrece una variedad de facilidades de filtrado y ordenamiento, además de datos de tiempos restantes para que el agente pueda determinar las prioridades de su trabajo. La Figura 24 muestra el PP de un agente, en el cual se ha filtrado el trabajo asociado a un programa y versión determinada. En el grid de la izquierda se observan las contabilizaciones de unidades de trabajo en cada una de las actividades y sus correspondientes estados. Pulsando sobre una celda de la izquierda, el grid de la derecha muestra las respectivas unidades de trabajo. En el grid de la derecha se ofrecen varias alternativas de selección para acceder al Gestor de Unidades de Trabajo.

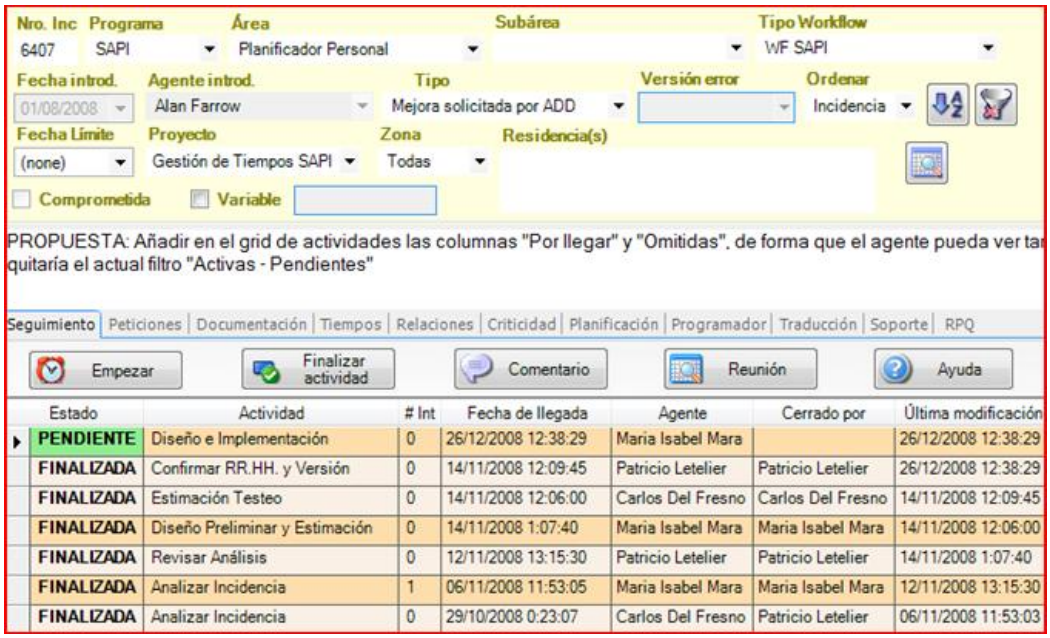


Figura 25. Gestor de Unidades de Trabajo - Pestaña de Seguimiento

El **Gestor de Unidades de Trabajo** (GUT) apoya al agente en el trabajo que debe realizar en una actividad asociada a una unidad de trabajo. Su funcionalidad se ha organizado en pestañas, dejando un panel superior permanente que muestra una ficha con los datos básicos de la unidad de trabajo. *La Pestaña Seguimiento* (Figura 25), es la pestaña por defecto, donde se puede consultar la historia de actividades que se han realizado sobre la unidad de trabajo los agentes. En esta pestaña además el agente puede realizar el registro de tiempos de su trabajo de una forma muy cómoda. El agente pulsa *Empezar* para comenzar su trabajo y registrar tiempo, *Pausar* para detener el registro de tiempo y *Finalizar* para terminar su trabajo en la actividad. La *Pestaña Peticiones* ofrece un mecanismo sencillo y efectivo de comunicación entre los agentes para comentar respecto de una unidad de trabajo.

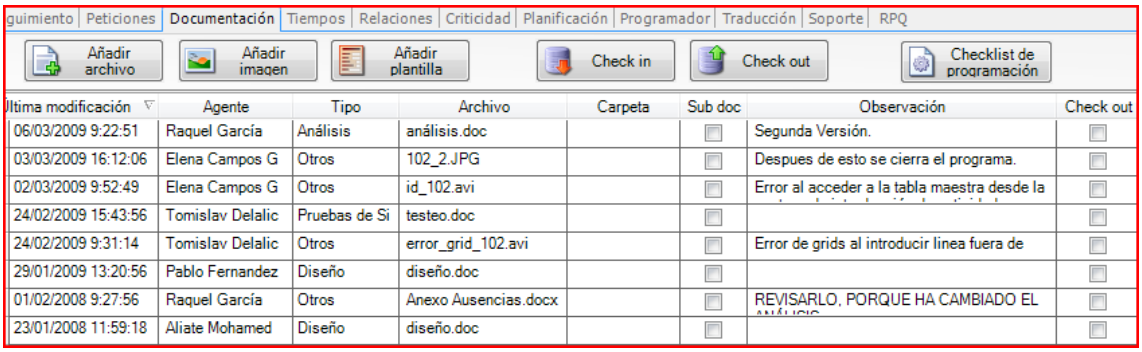


Figura 26. Pestaña de Documentación del GUT

La *Pestaña Documentación* (Figura 26) incluye una biblioteca de documentos (especificaciones y material complementario) compartida entre los agentes que trabajan con la unidad de trabajo. En ella se ofrecen plantillas y facilidades para llevar un control de versiones de los documentos. La Pestaña de Tiempos (Figura 27) permite introducir las estimaciones para cada actividad y consultar los registros de tiempo.

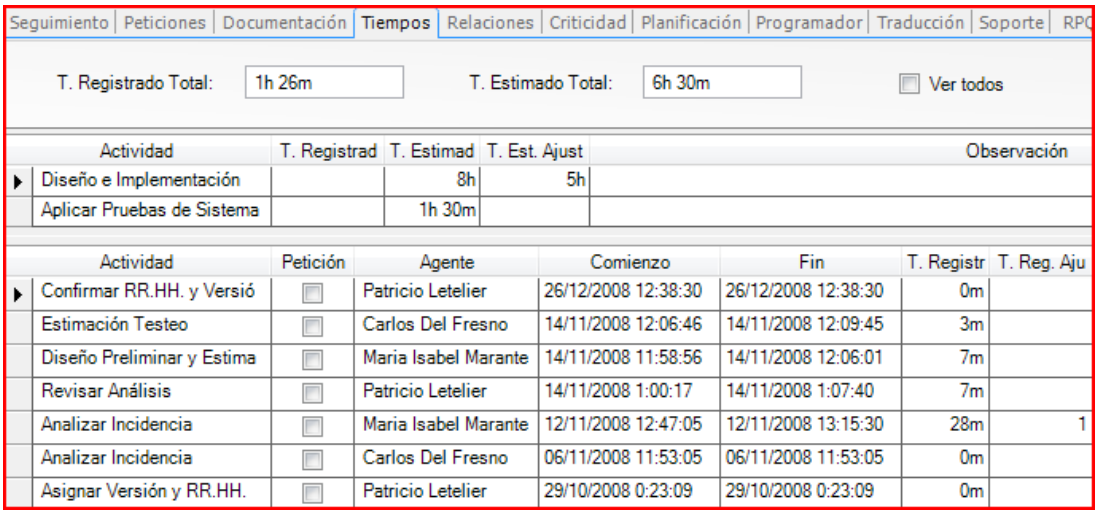


Figura 27. Pestaña de Tiempos del GUT

El **Planificador de Versiones** (PV) permite gestionar: los productos, sus versiones, los workflows disponibles para el producto, los agentes por defecto en las actividades y,

en cada versión, balancear la carga de los agentes. El Product Manager puede planificar una versión y en cualquier momento añadir o quitar unidades de trabajo. La Figura 28 muestra un fragmento con las unidades de trabajo incluidas en una versión.

Figura 28. PV - Unidades de Trabajo en un Versión

En la Figura 29 se muestra la Pestaña Carga de Agentes en Versión. En ella se puede conocer en cualquier momento la holgura simple de los agentes respecto de sus actividades en una versión del producto. En esta interfaz se ofrecen potentes mecanismos de filtros y agrupaciones por columnas. Cuando una versión tiene problemas de holgura en esta misma interfaz el Product Manager puede cambiar el agente asignado a la actividad (para balancear la carga de un agente) o cambiar de versión alguna unidad de trabajo. Otras alternativas son modificar la fecha de término de la versión, asignar más recursos humanos a la versión o partir unidades de trabajo para realizarlas incrementalmente en varias versiones.

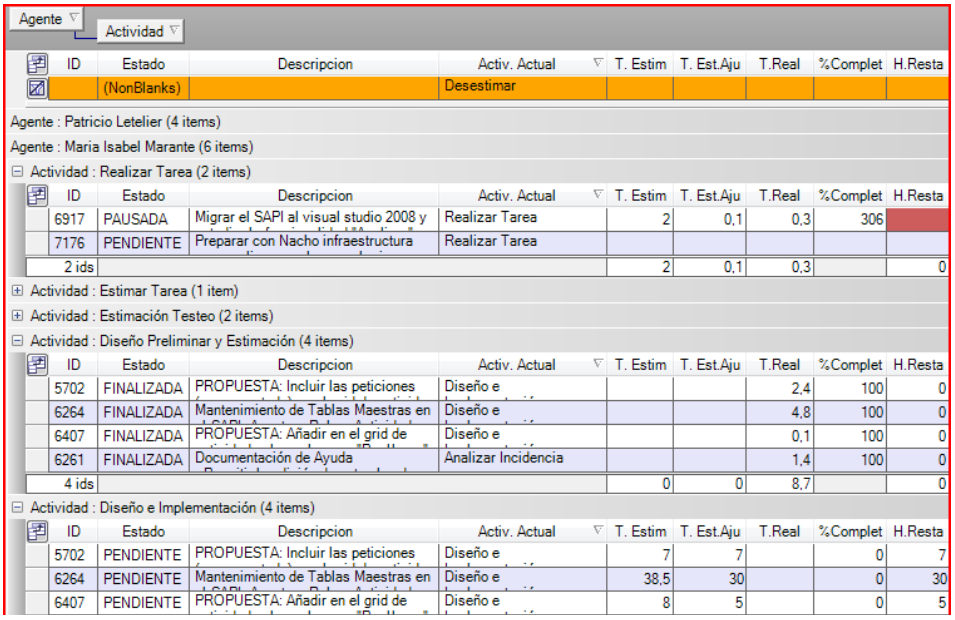


Figura 29. Fragmento de interfaz Gestión de Productos – Carga de Agentes en Versión

3.2 Gestión de requisitos en TUNE-UP

El Gestor de Requisitos permite gestionar la estructura de requisitos y las pruebas contenidas en cada nodo.

Para acceder al Gestor de Requisitos en el TUNE-UP, es necesario pulsar en el botón correspondiente del Gestor de Incidencias (Figura 30) o del Planificador Personal (Figura 31), según si queremos acceder en el contexto de una incidencia o no, respectivamente.

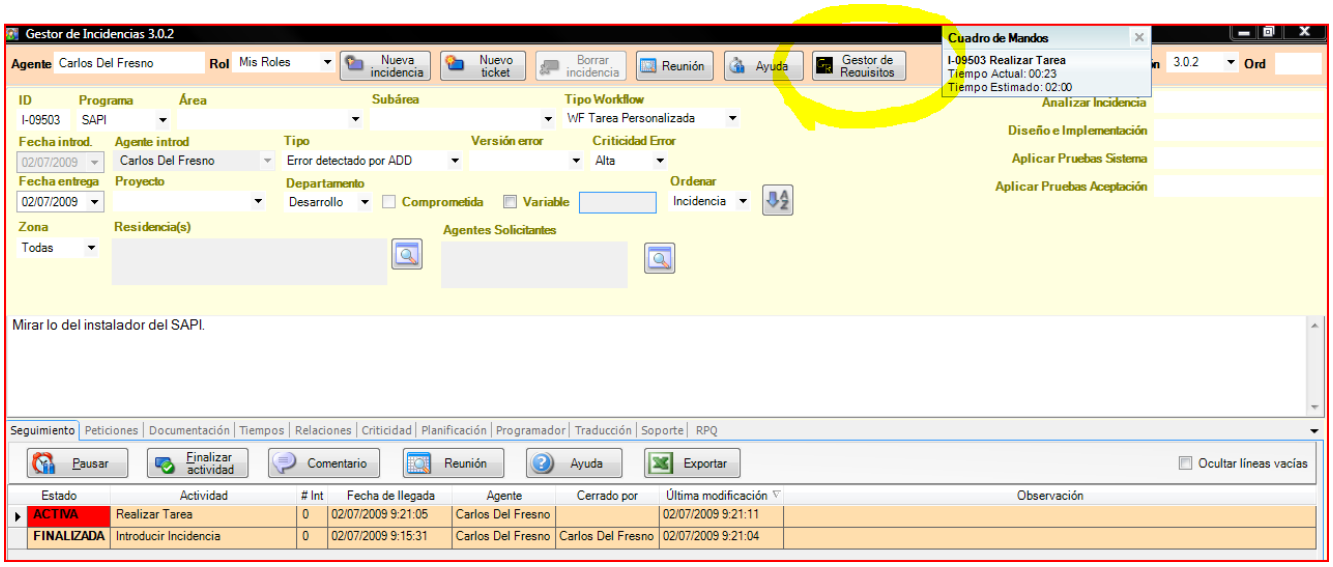


Figura 30. Gestor de Requisitos en el Gestor de Incidencias

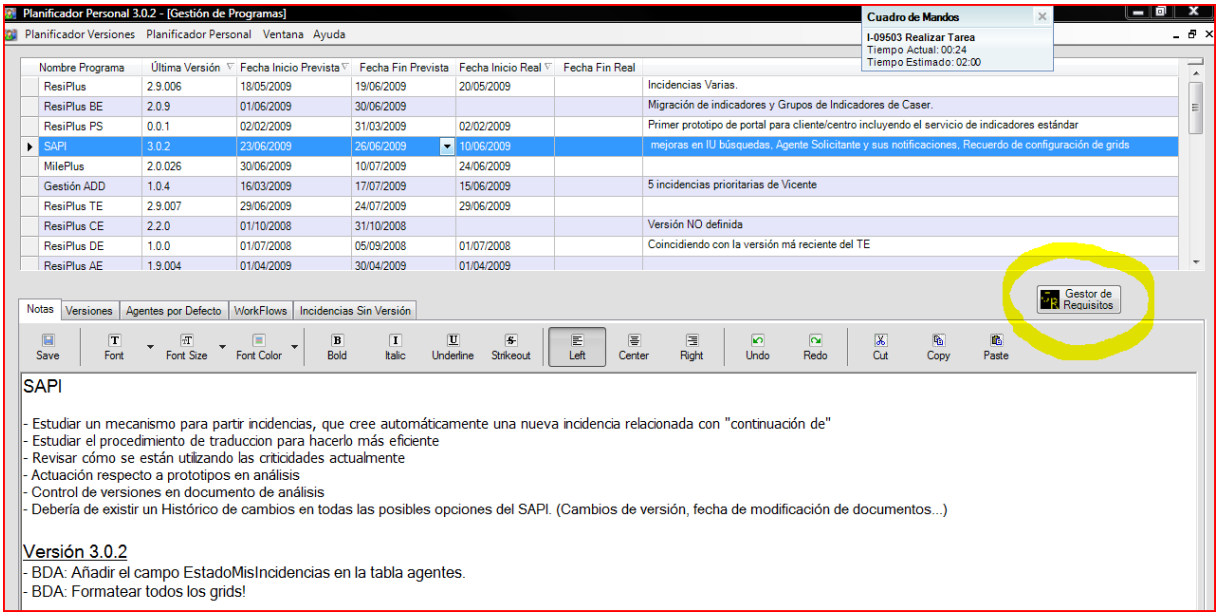


Figura 31. Gestor de Requisitos en el Planificador Personal

El módulo se ejecutará con un aspecto u otro y unos permisos determinados [Tabla 2] según el rol con el que esté conectado el agente en ese momento (product manager, analista, tester, etc).

	RR. Funcional		Analista		Otros	
	Fuera Incidencia	**Contexto Incidencia**	**Fuera Incidencia**	**Contexto Incidencia**	**Fuera Incidencia**	**Contexto Incidencia**
Mover/Copiar nodos	✓					
Marcar nodos como afectados. Ponerles descripción.		✓		✓		Sólo si acaba de crear la incidencia.
Modificar datos nodo	✓	✓				
Añadir/borrar nodos	✓					
Consultar todo. Buscar nodos.	✓	✓	✓	✓	✓	✓
Cambiar filtros nodos y programas	✓		✓		✓	
Gestión PA e IU actuales.	✓		✓			
Gestión PA e IU propuestas.		✓		✓		
Gestión Pruebas de Sistema	✓		✓		Sólo si es tester	
Añadir comentarios	✓	✓	✓	✓	✓	✓
Marcar comentarios como leidos/no leidos			✓	✓		

Tabla 2. Permisos por roles según contexto

Una vez dentro del Gestor de Requisitos podemos diferenciar cinco zonas de contenido, éstas son las marcadas en la Figura 32 con recuadros de colores.

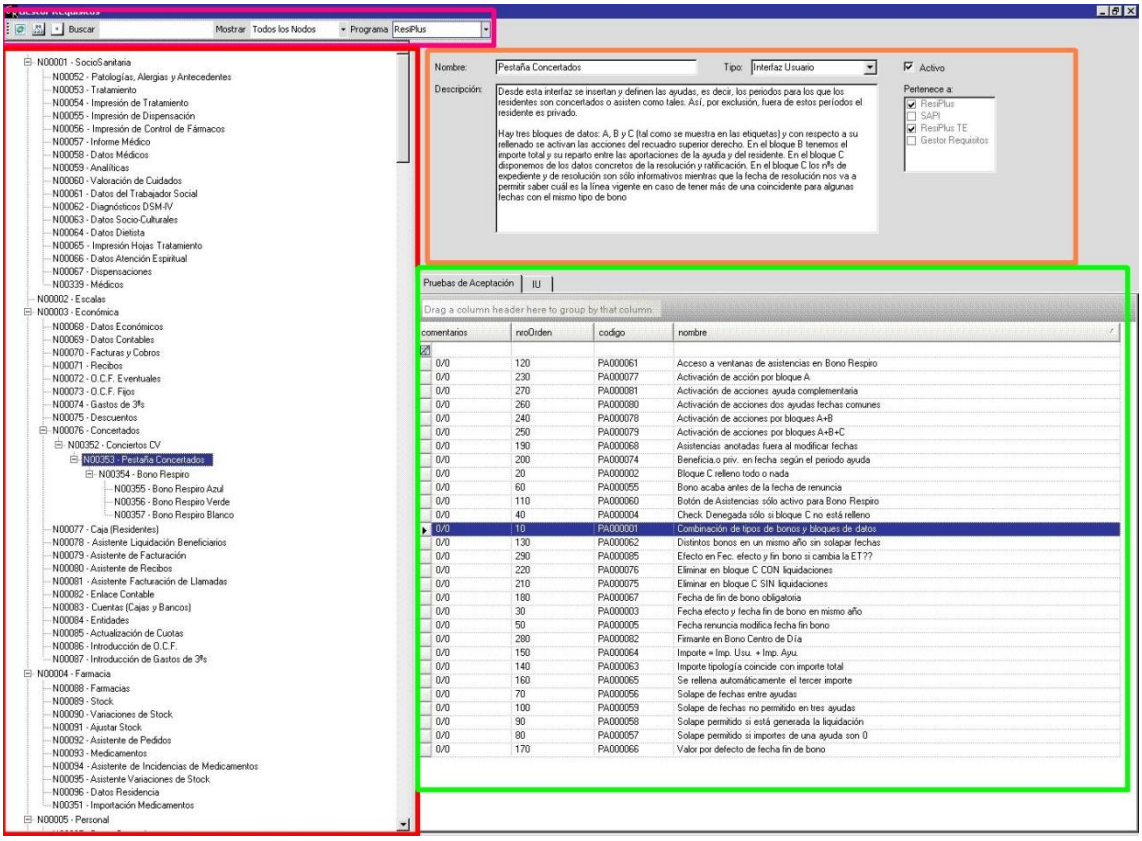


Figura 32. Formulario principal

3.2.1 El grafo de nodos

Corresponde a la región delimitada por el recuadro rojo en la Figura 32, y ampliado en la Figura 33.

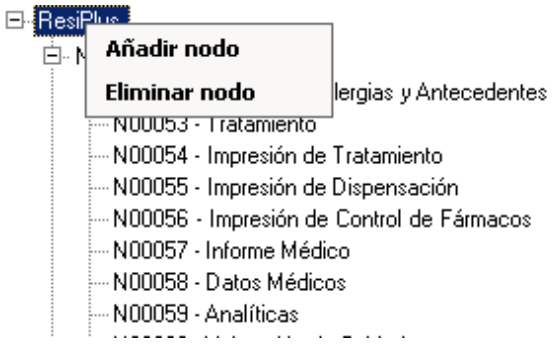


Figura 33. Añadir y eliminar nodo del GR

Éste es el órgano principal del módulo. Los nodos del grafo equivalen a lo que serían los requisitos del programa. Como tal, se pueden manipular de cualquier forma (añadir, eliminar, editar y mover) para facilitar la organización, el entendimiento y el mantenimiento de los requisitos, lo cual es uno de los principales objetivos del módulo.

Para añadir nuestros primeros nodos (requisitos) al programa hay que estar conectados en el módulo con el rol de RR. Funcional. Pulsamos con el botón derecho del ratón sobre el nodo raíz del grafo (cuyo nombre coincide con el del programa sobre el que estamos trabajando) y seleccionamos "*Añadir nodo*" (Figura 33).

Se abrirá una ventana (Figura 34) con los campos a rellenar sobre el nuevo nodo que queremos crear, así como con el código de identificativo autogenerado que se le asignará (en el caso de crearse definitivamente). Tras elegir un nombre, una descripción, y el tipo de nodo que estamos añadiendo (Requisito Funcional, Requisito No Funcional o Interfaz de Usuario), pulsamos sobre el botón "Añadir" y éste aparecerá en el grafo, bajo el nodo que seleccionamos inicialmente.

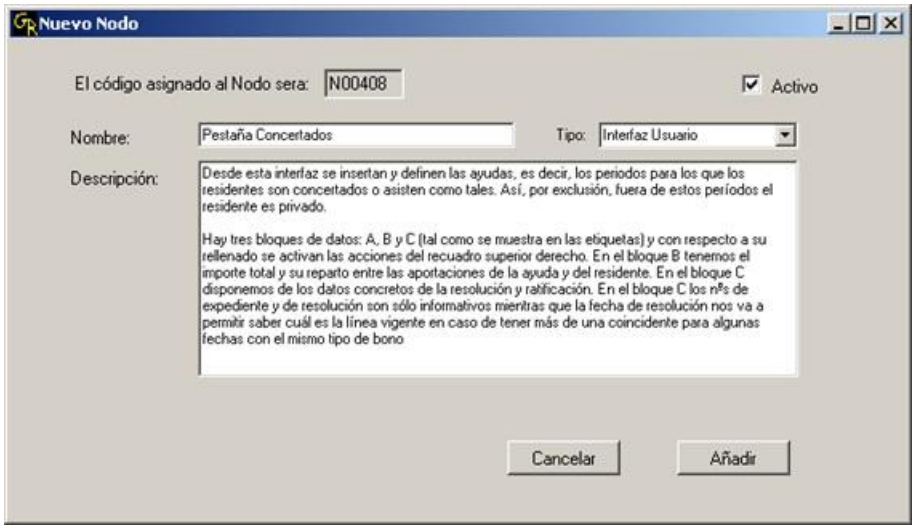


Figura 34. Nuevo Nodo

A partir de ahora podemos añadir todos los nodos que queramos, colgando de los nodos que deseemos, según la organización del programa.

Si nuestros requisitos cambian, el programa evoluciona y se requiere una modificación en el orden éstos, se pueden mover para estructurarlo en la forma deseada (hay que tener en cuenta que estamos ante un grafo, y como tal puede tener todas las ramificaciones que queramos) tan sólo seleccionando uno de ellos y, sin soltar el botón izquierdo del ratón, arrastrarlo sobre el nuevo nodo del que queremos que cuelgue y sólo entonces soltar el botón izquierdo.

Saldrá una ventana emergente preguntando si deseamos mover o añadir un nuevo padre (Figura 35). En caso de seleccionar lo primero, el nodo que has arrastrado, y todos sus hijos, desaparecerán de donde estaban y aparecerán bajo el nodo donde lo arrastramos.
En caso de haber seleccionado la segunda opción, el nodo será duplicado en el grafo, quedando el original, más una nueva copia de él bajo el nuevo padre. Es decir, que el mismo requisito podrá tener más de un padre. Ésta situación se puede dar en algunos programas, es por eso que este módulo lo soporta y cuenta para ello con un grafo acíclico de nodos, en lugar de un árbol.

No obstante, al realizar un cambio (de nombre, descripción, etc.) sobre el nodo que ha sido duplicado, el cambio se efectuará sobre él mismo, y sobre el nodo original del que fue copiado.

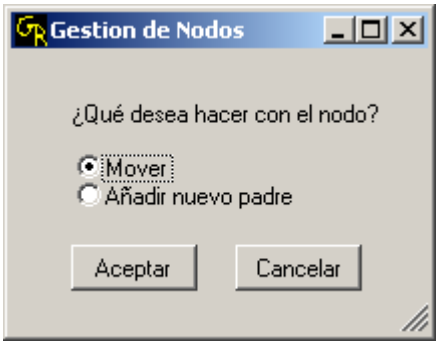


Figura 35. Mover nodos

Hay que notar que no se pueden mover nodos padre dentro de sus nodos hijos, ya que eso crearía recursividad e inconsistencias. Para evitarlo, el módulo impide esta acción.

En caso de que algún requisito desaparezca o se haya añadido por error, podemos eliminarlo seleccionándolo, pulsando el botón derecho del ratón y eligiendo “Eliminar Nodo” (Figura 33).

3.2.2 Las herramientas y el buscador

Ésta es la región delimitada por el recuadro rosa en la Figura 32. Lo vemos ampliado en la Figura 36:

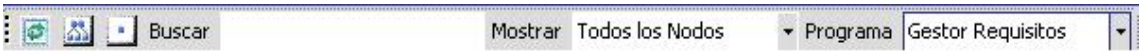


Figura 36. Barra de herramientas del GR

En esta zona, de izquierda a derecha, podemos realizar las siguientes opciones:

El primer tooltip, llamado “Actualizar”, permite refrescar todo el grafo de nodos, mostrando cambios en los nodos que aún no estaban reflejados.

El segundo tooltip, llamado “Expandir todo” hace que todos los nodos del grafo se expandan, es decir, que los nodos que tienen hijos, pero estaban ocultos porque el padre estaba plegado, se desplegarán y se podrán ver todos los hijos.

El tercer tooltip, llamado “Contraer todo” hace lo contrario al anterior. Contrae todos los nodos de forma que no se puedan ver los hijos que éstos tienen.

El siguiente campo es el buscador de nodos. Situando el puntero sobre ése área y escribiendo un texto, se puede ver cómo se van marcando en el grafo todos aquellos

nodos que contienen la cadena especificada de texto en su nombre. Con esta utilidad se puede encontrar fácilmente un nodo en concreto que estemos buscando dentro del vasto grafo.

El cuarto campo se trata de un filtro de nodos. Al hacer clic en el desplegable, tendremos las opciones de mostrar: “Todos los Nodos”, “Nodos Activos” o bien “Nodos No Activos”. De elegirse la primera opción, se mostrarán todos los requisitos del programa en el grafo sin restricción alguna. Si se elige la segunda opción, sólo los nodos cuyo estado sea activo se mostrarán en el grafo, y viceversa si elegimos sólo mostrar “Nodos No Activos”.

La última opción del módulo es un filtro por programas. Si estamos realizando un proyecto que conlleva más de un programa a la vez, o varios programas independientes, todos se pueden gestionar desde la misma ventana, no siendo necesario cerrar el módulo para volverlo a abrir con otro programa. De esta forma, en el desplegable “Programa”, podremos elegir el programa cuyos requisitos queremos ver, o bien elegir “Todos”, en cuyo caso veremos los requisitos de todos los programas en el grafo, separados en grupos según el nombre del programa. Esto puede ser útil si hay programas que tienen requisitos en común y así poderlo ver claramente.

3.2.3 La zona de información

Es la región delimitada por el recuadro naranja en la Figura 32, y ampliado en la Figura 37:

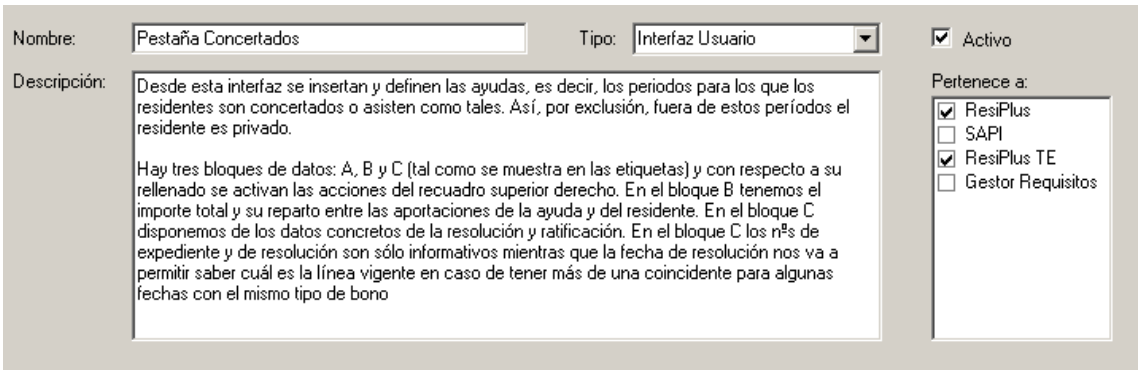


Figura 37. Zona de información en el GR

Al pulsar sobre un nodo en el grafo podemos ver aquí sus datos generales: nombre, descripción, tipo de requisito, programas a los que pertenece, así como su estado: si se encuentra activo o no.

Si accedimos al módulo con el rol de RR. Funcional podemos alterar esta información directamente sobre las cajas de texto. Así pues, podemos cambiar el texto que aparece en la descripción, el nombre del nodo, cambiar su tipo o su estado. La próxima vez que

se cargue el grafo de nodos, o bien si pulsamos sobre el botón “Actualizar”, podremos ver efectivo el cambio.

3.2.4 Las Pruebas de Aceptación

Es la región delimitada por el recuadro verde en la Figura 32, y ampliado en las Figura 38.

En esta zona es donde se gestionan las pruebas de aceptación y las interfaces de usuario de cada nodo.

Para añadir una nueva prueba de aceptación, seleccionaremos un nodo del grafo, a continuación en el grid de las pruebas de aceptación pulsaremos con el botón derecho del ratón y elegiremos “Añadir Nueva PA” (Figura 38).

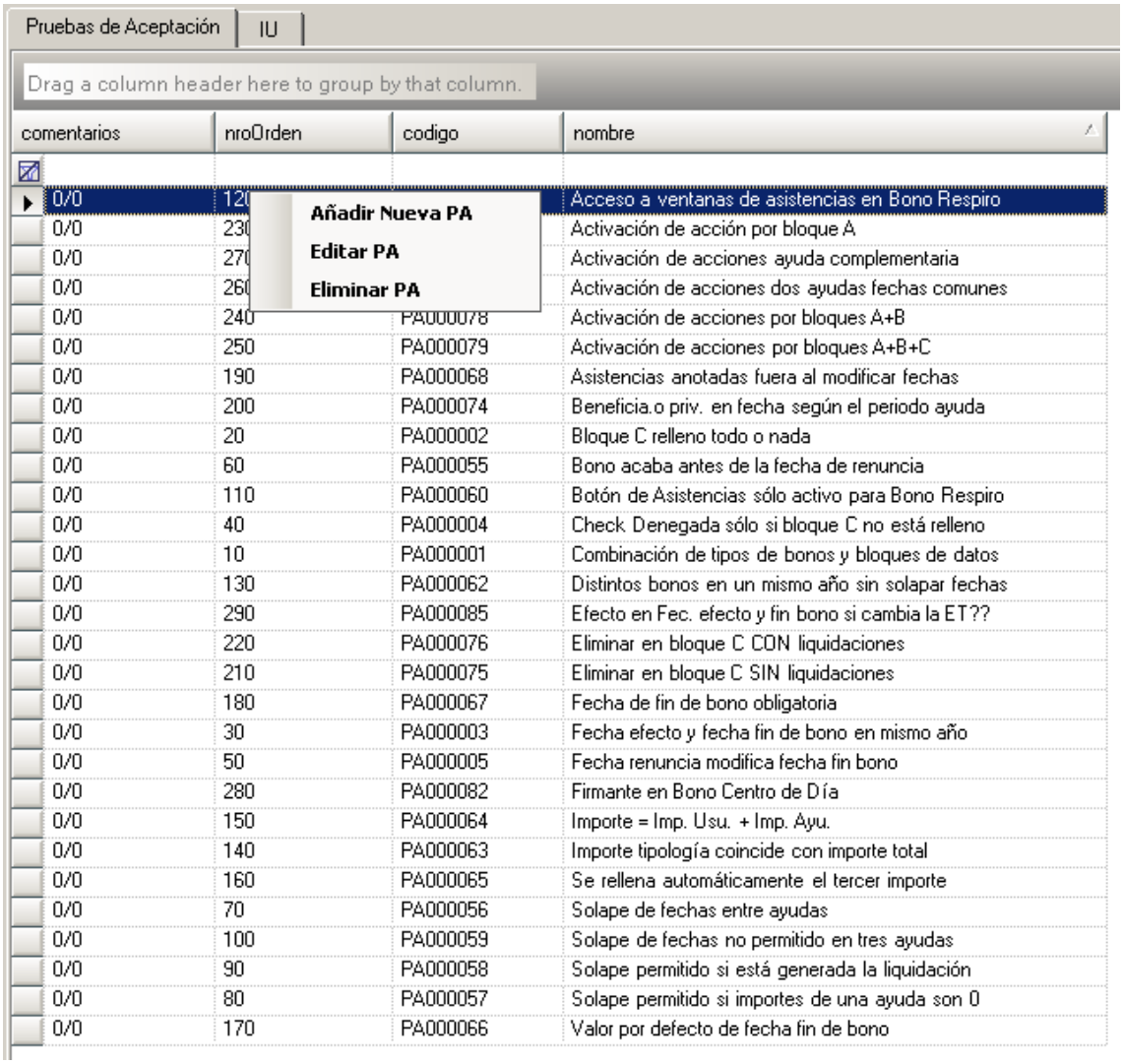


Figura 38. Pruebas de Aceptación en el GR

Una vez hecho esto, se abrirá una nueva ventana (Figura 39) con los campos necesarios para definir una prueba de aceptación. Estos son: el nombre de la prueba, el número de orden en caso de requerirse y la descripción, la cual está dividida en 4 partes:

- "Condición": que se tiene que dar para que se cumpla la prueba de aceptación.
- "Pasos": que hay que seguir para comprobar la prueba.
- "Resultado esperado": explicación de qué es lo que debería suceder en caso de que la prueba fuera correcta.
- "Observaciones": anotaciones extra que puedan ser necesarios tener en cuenta.

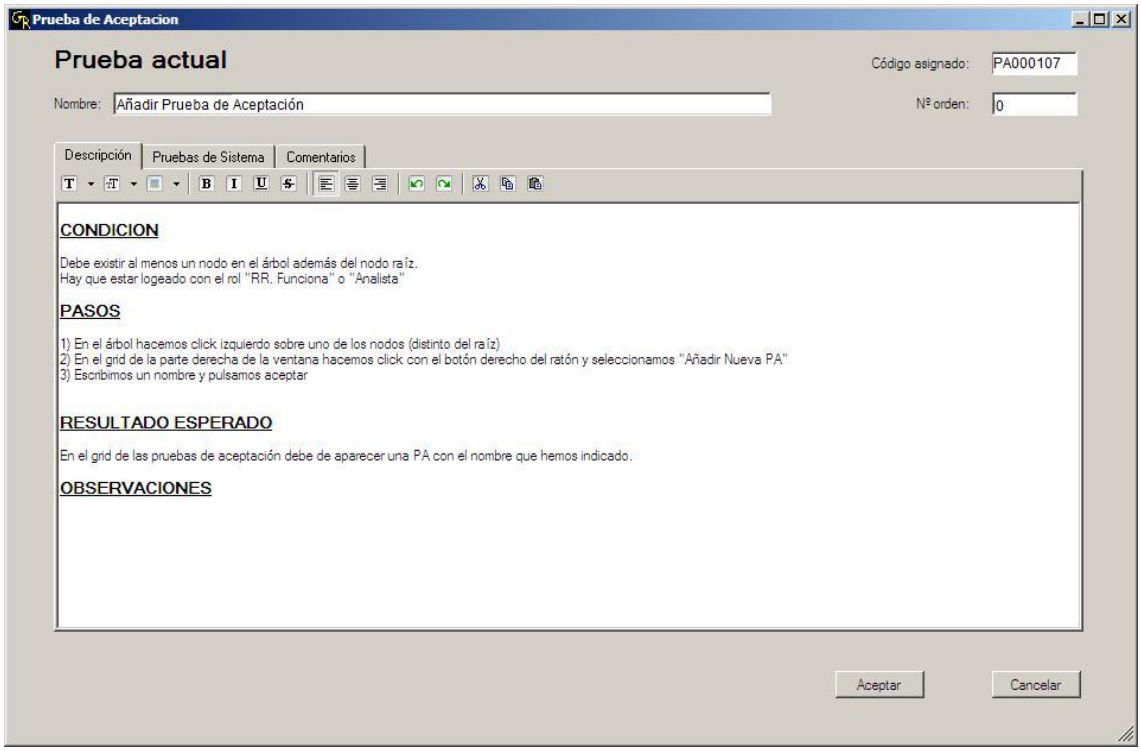


Figura 39. Nueva PA en el GR

Además, si pulsamos sobre la pestaña "Pruebas de Sistema", nos es posible añadir pruebas de sistema a la prueba de aceptación, sólo haciendo botón derecho sobre el grid, eligiendo "Añadir PSistema" (Figura 40), y posteriormente escribiendo una descripción en la nueva fila que aparece en el grid.

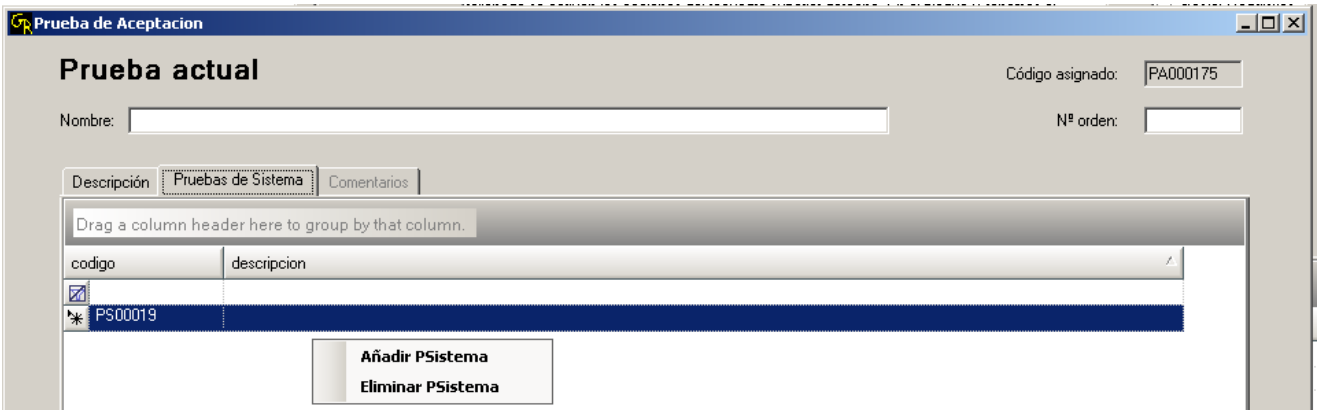


Figura 40. Pruebas de Sistema en GR

Podemos eliminar Pruebas de Aceptación seleccionándolas en el grid de las pruebas (Figura 38), haciendo clic derecho y eligiendo “Eliminar PA”.

Si seguimos los mismos pasos que para eliminar, pero en lugar de seleccionar “Eliminar PA”, escogemos “Editar PA”, de estar conectados con un usuario con los permisos necesarios (ver Tabla 2), se abrirá la información de la PA en modo editable, pudiendo cambiar su nombre, descripción, número de orden y pruebas de sistema.

Otra función interesante es la posibilidad de añadir comentarios a cualquier prueba de aceptación y desde cualquier rol de usuario. Siempre que alguien identifique un problema o tenga algo que aportar respecto a una prueba, puede hacer doble clic sobre dicha prueba y se abrirá su información. Vamos a la pestaña “Comentarios” y en el grid pulsamos botón derecho y elegimos “Nuevo comentario”. Esto abrirá una ventana simple como la que se puede ver en la Figura 41, donde introduciremos el comentario y pulsaremos “Aceptar” a continuación.

El comentario quedará añadido y enlazado a la prueba de aceptación, junto con la fecha y hora en la que fue introducido y el nombre del agente que lo comentó. Al conectarse al módulo el analista podrá ver si tiene comentarios sin leer (indicado en la segunda columna del grid de las pruebas de aceptación, llamada “comentarios” (Figura 38)).

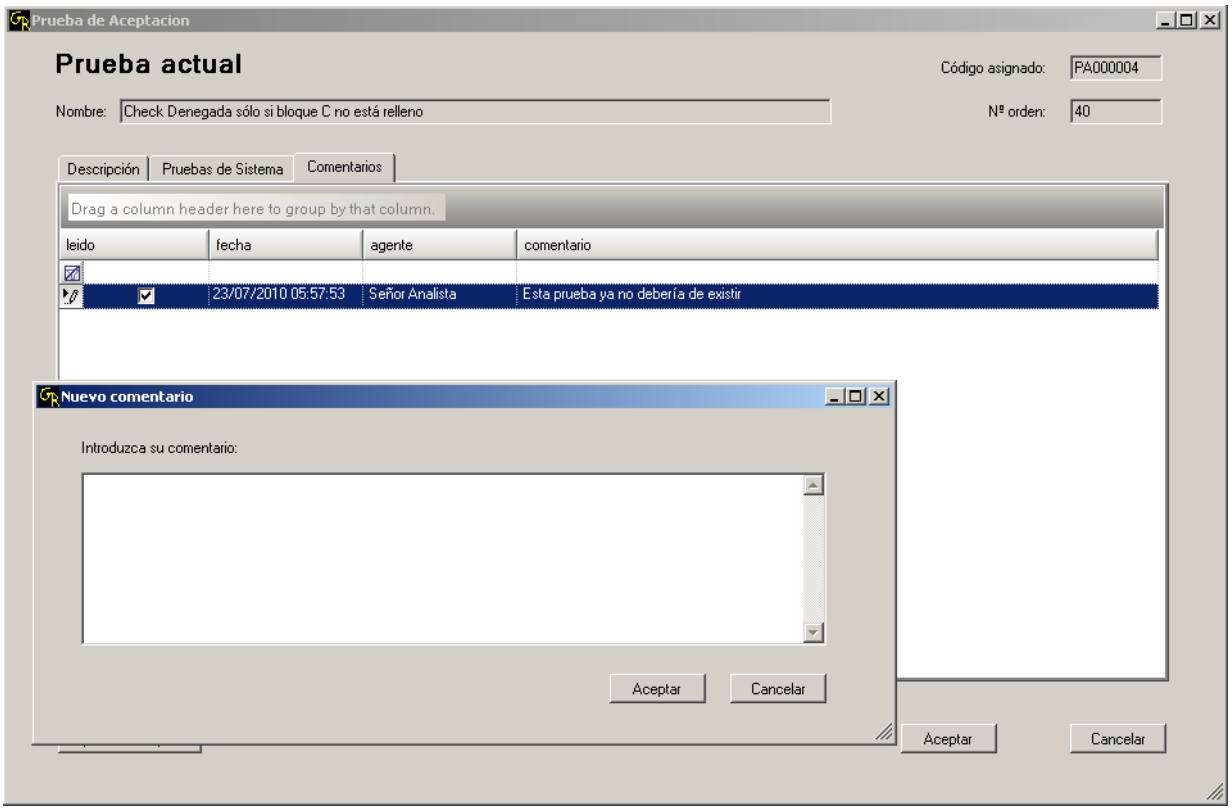


Figura 41. Comentarios en el GR

Cabe destacar que, tanto en el grid de las pruebas de aceptación, como en el de las pruebas de sistema y los comentarios se ofrece la posibilidad de filtrar; esto es, escribiendo algún texto en la primera fila del grid (el filtro) desaparecen las filas que no contengan dicho texto. Es muy cómodo en el momento en que haya muchas pruebas de aceptación pero, por ejemplo, sólo interesen las que hablen de algo en concreto.

3.2.5 Las Interfaces de Usuario

En esa zona se muestran las capturas asociadas a los requisitos, en caso de que estos sean del tipo “Interfaz de Usuario”. Accedemos a ella pulsando sobre la pestaña “IU”, justo al lado de las pruebas de aceptación. Al hacerlo nos encontramos con un grid como el de la Figura 42.

Para añadir capturas, habrá que seleccionar en el árbol un requisito de dicho tipo, luego ir a la pestaña IU, y en el grid pulsar botón derecho del ratón y elegir “Añadir captura”.

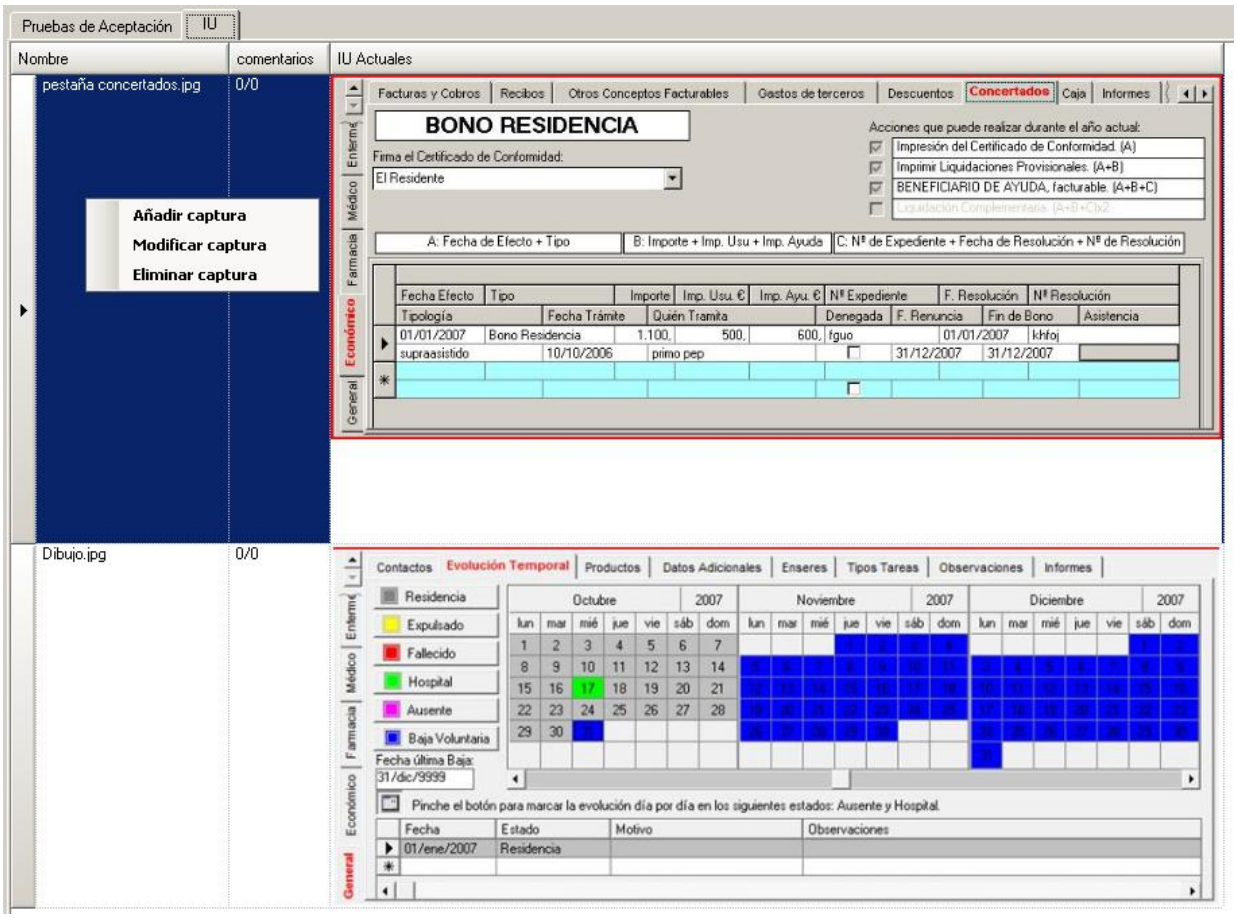


Figura 42. Interfaces de Usuario en el GR

Aparecerá un formulario como el de la Figura 43, en la página siguiente.
Pulsando en examinar podemos buscar en nuestro disco duro cuál es la imagen que queremos asociar al requisito. Al elegir una y pulsar aceptar el nombre que se seleccionará por defecto será el mismo nombre de la imagen que hemos seleccionado, no obstante, podemos cambiar el nombre.

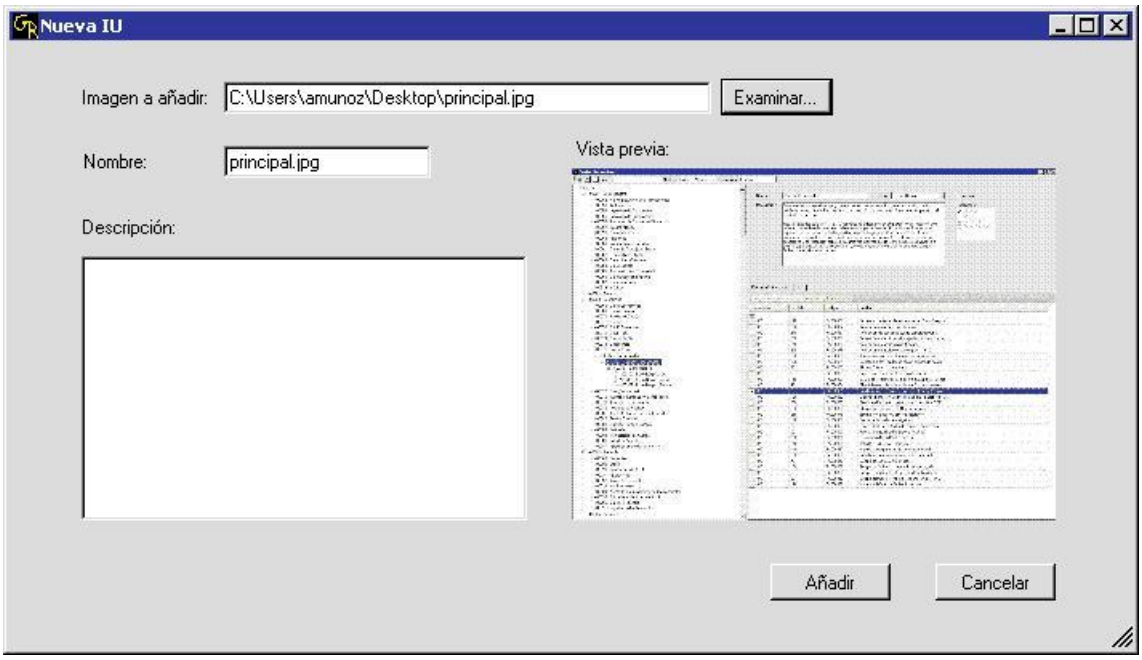


Figura 43. Nueva IU en el GR

Por último podemos indicar la descripción que creamos conveniente para la captura y pulsar “Aceptar”.

La captura, con su información, se añadirá al grid de las Interfaces de Usuario, tal como se veía en la Figura 42.

Una vez hecho esto, haciendo doble clic sobre la imagen, se abrirá una ventana (Figura 44), donde podemos editar la descripción o el nombre (si tenemos permisos), así como añadir pruebas de sistema y comentarios, de igual manera que se explicó para las Pruebas de Aceptación.

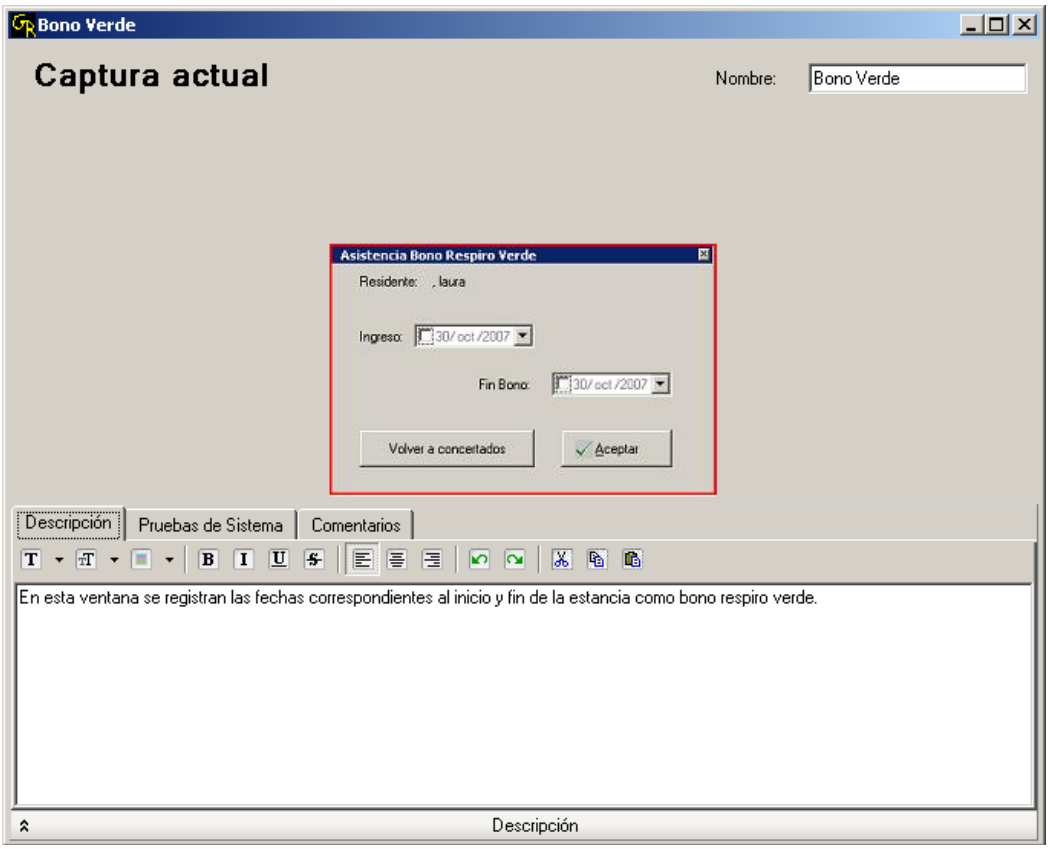


Figura 44. Editar IU

3.2.6 Incidencias

Si lo que se desea es informar sobre alguna incidencia, es decir, algún cambio que haya que realizar, ya sea dar de baja un requisito, modificar alguna descripción o prueba de aceptación o dar de alta una nueva; entraremos en el Gestor de Requisitos en el contexto de la incidencia, con un usuario cuyo rol sea analista.

En el contexto de una incidencia, el Gestor de Requisitos apoya a los analistas en la definición del cambio de comportamiento.

Para el equipo de desarrollo es muy importante conocer qué nodos afecta cada incidencia. Esta información permite detectar posibles conflictos o solapes entre las incidencias, contenidas en la misma versión o entre incidencias de diferentes versiones. Por ejemplo, si un analista está definiendo un cambio de comportamiento en un nodo, le interesará conocer cómo ha evolucionado su comportamiento (conocer las incidencias realizadas en el nodo con sus correspondientes cambios en las PAs), el comportamiento actual del nodo (las PAs vigentes del nodo) y los cambios pendientes en el nodo (incidencias pendientes que afectarán al nodo, con sus correspondientes cambios propuestos para sus PAs).

En TUNE-UP el módulo Gestor de Incidencias apoya a los agentes en sus actividades sobre una determinada incidencia. Desde este módulo y para cada incidencia, se puede acceder al Gestor de Requisito, donde el analista determina los nodos que se verán afectados y define los cambios en las PAs de dichos nodos.

En la Figura 45 se observa la estructura de requisitos del producto (desplegada de manera parcial), donde se ven marcados los nodos afectados.

Marcaremos un checkbox para indicar que el nodo se ve afectado por esa incidencia; en ese momento nos será posible añadir un comentario (a la derecha del nombre del nodo) describiendo cuál es el motivo por el que el nodo se ve afectado por la incidencia con la que estemos actualmente conectados.

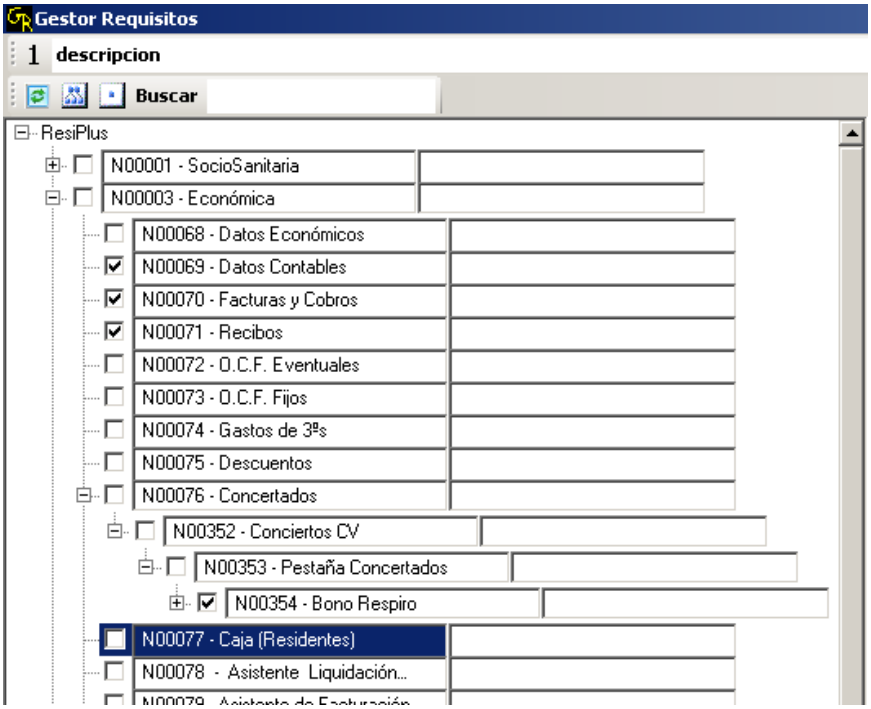


Figura 45. Nodos en el contexto de una incidencia

Si es necesario, también podemos añadir, modificar o eliminar PAs y/o IUs. Estos cambios quedarán indicados como "*propuestos*", no haciéndose efectivos hasta que se indique.

En la Figura 46 se observa la lista de PAs del nodo seleccionado. Tal y como se muestra, las consecuencias de la implantación de una incidencia son reflejadas claramente por los cambios en las pruebas del nodo. De abajo a arriba, las PA que salen en el grid de la figura son: "una propuesta nueva" (), "una propuesta de eliminación" () y "una propuesta de modificación" (). Los dibujos que aparecen en la primera columna, "Acción", son los correspondientes a cada una de esas propuestas. Además, las PAs que no tienen icono en esta columna son pruebas actuales del nodo e indican un comportamiento ya existente que se mantiene.

Haciendo clic sobre la segunda columna, “comentarios PA Propuesta” se pueden añadir comentarios respecto a esa propuesta, de la misma manera que se ha explicado con anterioridad.

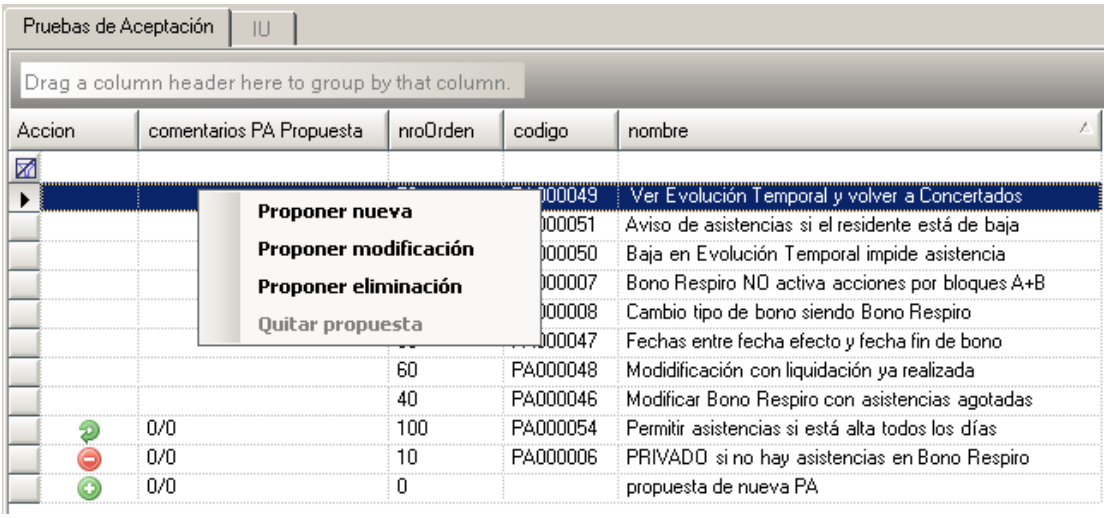


Figura 46. PA en el contexto de una incidencia

En el caso de las IU, la interfaz queda de la siguiente manera al estar en el contexto de una incidencia:

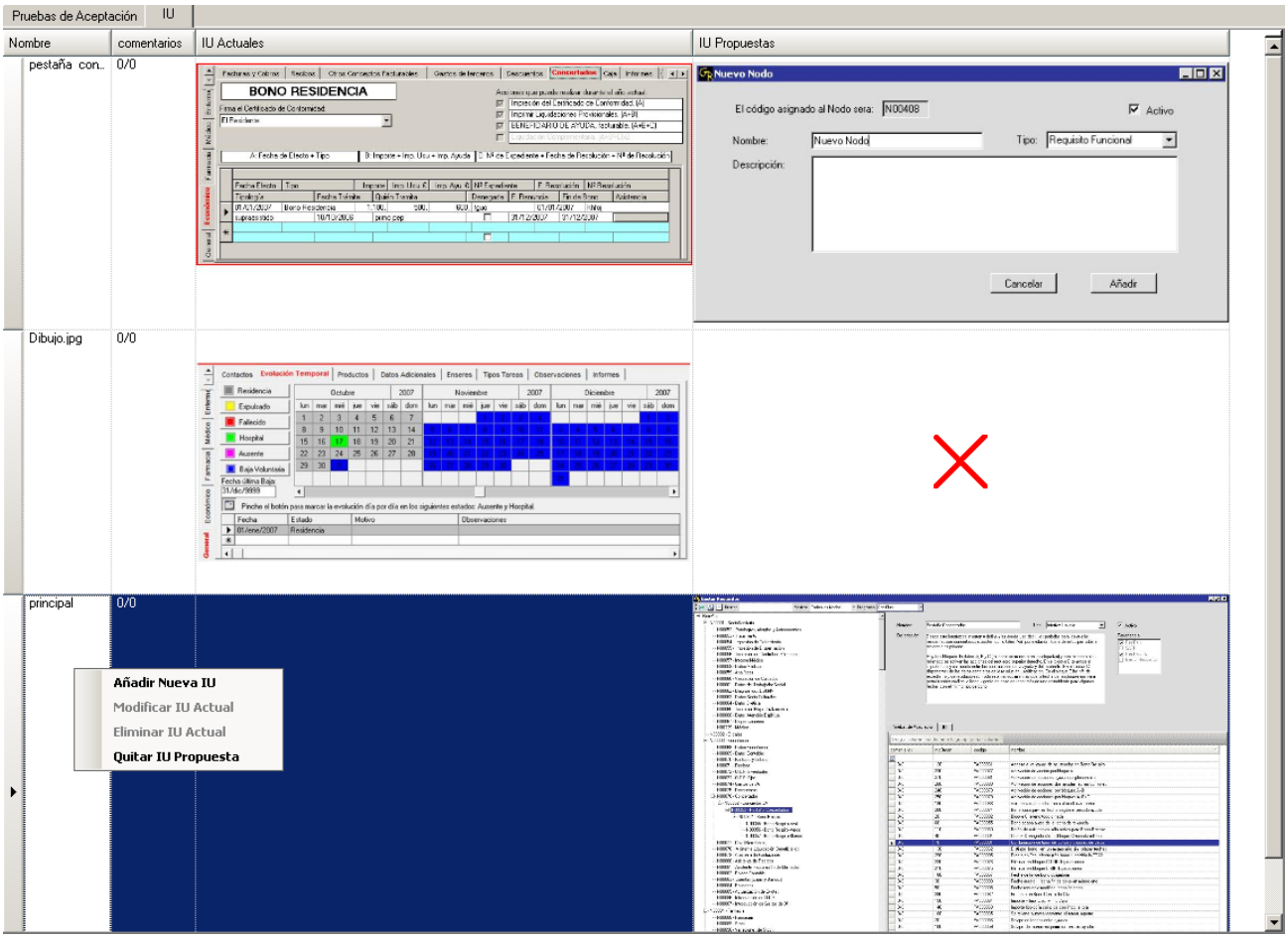


Figura 47. IU en el contexto de una incidencia

De arriba abajo, se presentan: “una propuesta de modificación” (con una captura propuesta en la columna “IU Propuesta”), “una propuesta de eliminación” de la interfaz (indicado con el símbolo: ╳) y “una nueva propuesta” de interfaz para el programa (denotado porque no hay actualmente ninguna captura bajo la columna “IU Actual”, pero sí que hay una propuesta).

Para hacer una propuesta de las anteriores es tan sencillo como hacer clic derecho sobre la interfaz que quieres cambiar y elegir una de las opciones que se pueden ver en menú contextual de la Figura 47: “Añadir Nueva IU”, “Modificar IU Actual” o “Eliminar IU Actual”. La opción “Quitar IU Propuesta” sirve para eliminar una propuesta de cualquiera de los 3 tipos anteriores que se haya hecho a alguna interfaz, bien porque haya sido descartada o bien porque se pusiera por error.

Capítulo 4. Módulo de requisitos para TUNE-UP Process Tool

En este capítulo se presenta el análisis del módulo de requisitos, en el cual se describe su estructura y funcionalidad mediante diagramas que permiten comprender cómo funciona. Estos diagramas describen cada una de las funciones que lleva a cabo el módulo y las clases de las que se compone el mismo (diagrama de clases).

4.1 Requisitos del módulo

Los requisitos del módulo del Gestor de Requisitos se han especificado en el mismo. Así que fueron sus propios requisitos los primeros con los que se probó el GR, antes de ser implantado en TUNE-UP.

Las Pruebas de Aceptación, que fueron exportadas desde el módulo, pueden encontrarse en el Anexo A.

4.2 Modelo Conceptual

Un diagrama de clases presenta las clases del sistema con sus relaciones estructurales y de herencia.

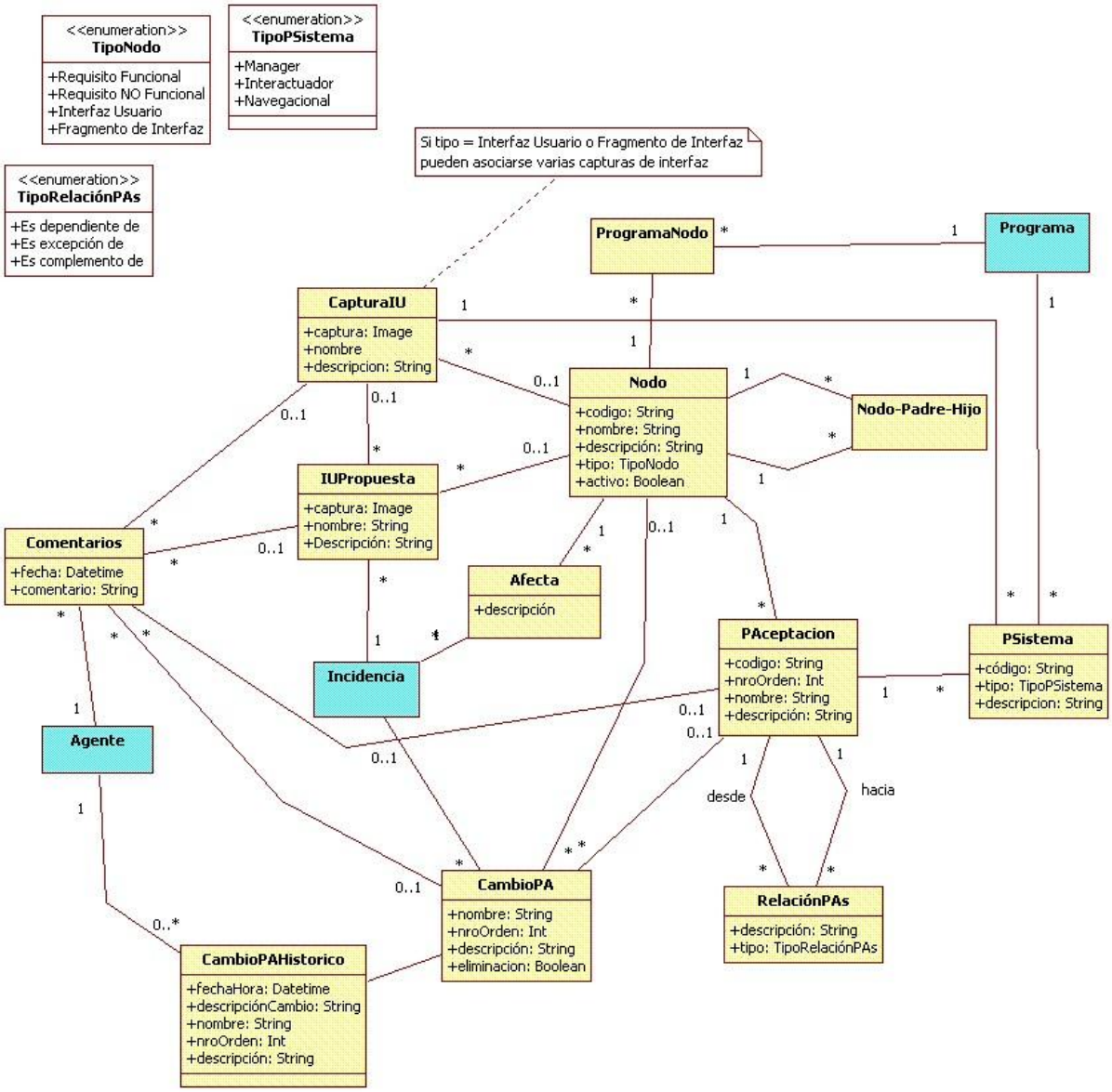


Figura 48. Diagrama de clases.

Del diagrama de la Figura 48 se puede extraer que cada programa que se gestione tendrá una serie de Nodos. Dichos nodos, que pueden ser requisitos o interfaces de usuario (según vemos en la clase enumerada “TipoNodo”), tendrán asociados un conjunto de Pruebas de Aceptación (clase “PAceptacion”) y, en el caso de tratarse del segundo tipo, una serie de capturas de pantalla (clase “CapturaIU”).

No incorporaremos ningún fragmento de código de estas clases debido a su extensa longitud; no obstante, en las páginas 96 – 97, se incluyen unos fragmentos de código alternativos para el conjunto de clases comprendidas por “ProgramaNodo”, “Nodo” y “NodoPadreHijo”.

Cabe notar que las clases “Nodo”, “PAceptacion”, “CapturaIU” y “PSistema” tienen un atributo de tipo String llamado “código”, el cual actúa como identificador. Dicho

atributo es calculado de manera automática e incremental y, aunque cada uno de ellos difiere del otro ligeramente, para identificarlos más claramente incluiremos aquí el fragmento de código de cómo generamos el identificador de las Pruebas de Aceptación a modo de ejemplo:

```
//genera el código que se asignará a las nuevas PA
private string generaCodigoPrueba()
{
      if (db.GRPAceptacion.Count() == 0) idFuturaPA = 1;
      else idFuturaPA =
           Convert.ToInt32(db.UltimoID("GRPAceptacion").Single().lastID)
           + 1; //el último nodo que HUBO más el incremento

      string codigo = "PA";

      //añadimos 0s hasta que tenga 7 cifras en total
      while (codigo.Length < 8 - idFuturaPA.ToString().Length)
            codigo += 0;
      return codigo += idFuturaPA.ToString();
}
```

Los Nodos tienen una representación jerárquica, de manera que se pueden ramificar en forma de grafo, esto se puede intuir en la relación de la clase “Nodo” con “Nodo-Padre-Hijo”.

Lo siguiente es un fragmento para ejemplificar[7] cómo generamos el grafo de nodos en el módulo, teniendo en cuenta esta relación Padre-Hijo:

[7] Con el fin de simplificar, hemos omitido del código los filtros (en los que se permite mostrar sólo nodos activos, no activos, o todos) así como el código correspondiente a conectarse en la aplicación en el contexto de una incidencia, y la creación del nodo raíz o inicial que obviamos.

```
//va haciendo hijos a los nodos en forma de arbol
public void insertarNodos(UltraTreeNode punteroPadre){
      ultraProgressBar1.PerformStep();
      GRequisitosDataSet.GRNodoDataTable tablaNodos = new
      GRequisitosDataSet.GRNodoDataTable();
      UltraTreeNode punteroNodo;
      bool iterar = false, activo1 = true, activo2 = true;
      //se muestran los nodos activos, no activos o todos
      switch (VerDefault){
            case Ver.Todos: { activo1 = true; activo2 = false; break; }
            case Ver.SoloActivos:{activo1= true; activo2 = true; break;}
            case Ver.SoloInactivos:{activo1=false; activo2=false;break;}
      }

      if (punteroPadre.IsRootLevelNode){//caso base, estamos en la cima
            this.nodoTableAdapter.NodosPrincipales(tablaNodos, activo1,
            activo2, idPrograma); //cogemos los nodos del nivel 1

            iterar = true;
      }
      //sino comprobamos que el nodo tenga hijos
      else if (this.nodoTableAdapter.NodosHijos(tablaNodos,
      int.Parse(punteroPadre.Key.TrimStart('+')), activo1,
      activo2,idPrograma) > 0)

            iterar = true;

      if (iterar) //si hay nodos se recorren los hermanos
      foreach(GestorIncidencias.GestorRequisitos.GRequisitosDataSet.GRN
odoRow nodo in tablaNodos){
            string claveNodo = nodo.IDNodo.ToString();
            punteroNodo = null;
            bool exito = false;
            do{
                  try{
                  punteroNodo = punteroPadre.Nodes.Add(claveNodo,
                  nodo.codigo + " - " + nodo.nombre);
                  //si llega aquí es que no ha saltado la excepción
                  exito = true;
                  }
                  catch (ArgumentException){
                        claveNodo = "+" + claveNodo;
                  }
             } while (!exito);

insertarNodos(punteroNodo); //para cada hermano se explorarán sus hijos
}
```

Tanto las Pruebas de Aceptación como las capturas de interfaz de usuario podrán tener: propuestas de cambio por cada incidencia (clases “CambioPA” e “IUPropuesta”, respectivamente), un conjunto de pruebas de sistema (clase “PSistema”) y unos comentarios que puede realizar cualquiera de los agentes sobre ellos (representado por la clase “Comentarios”).

A continuación, los fragmentos de código correspondientes a “añadir pruebas de sistema” y “añadir comentarios”, respectivamente. Dicho código es el mismo tanto para las Pruebas de Aceptación, como para las Interfaces de Usuario.

```
//añade una PSistema
private void añadirPSistemaToolStripMenuItem_Clic(object sender,
EventArgs e)
{
      Infragistics.Win.UltraWinGrid.UltraGridRow row =
      ultraGrid1.Rows.Band.AddNew();

      int nuevasInserciones = 0;

      foreach (Infragistics.Win.UltraWinGrid.UltraGridRow fila in
      ultraGrid1.Rows)
            if ((int)fila.Cells["IDPSistema"].Value < 0 )
                  nuevasInserciones++;
      row.Cells["codigo"].Value =
      GRPSistema.generaCodigoPSistema((int)db.UltimoID("GRPSistema").Si
      ngle().lastID + nuevasInserciones);

      row.Cells["tipo"].Value = 1;
}
```

Podemos observar en ambos códigos, sobre y bajo este texto, el uso de los componentes de Infragistics, los cuales dan más riqueza al módulo, gracias a sus posibilidades no aportadas por los componentes básicos de Visual Studio.

En ambos casos se trata del uso sobre un "grid", donde listamos las pruebas de sistema (arriba) y los comentarios (abajo).

En el caso del comentario, podemos ver cómo queda reflejado en la fila, al introducirse el texto deseado, cuál fue el momento en el que se introdujo dicho comentario, así como quién fue el usuario que lo hizo.

```
private void nuevoComentarioToolStripMenuItem_Clic(object sender,
EventArgs e)
{
      FormNuevoComentario f = new FormNuevoComentario();
      if (f.ShowDialog() == DialogResult.OK)
      {
            Infragistics.Win.UltraWinGrid.UltraGridRow row =
            ultraGrid2.Rows.Band.AddNew();

            row.Cells["IDComentario"].Value = aux; aux--;
            row.Cells["fecha"].Value = DateTime.Now;
            row.Cells["agente"].Value = nombreAgente;
            row.Cells["comentario"].Value = f.getComentario();
            comentariosChanged = true;
       }
}
```

Sabremos qué nodos son afectados en cada incidencia por la relación entre "Incidencia" – "Nodo". En la clase "Afecta" podremos guardar información sobre el nodo en concreto, la incidencia por la que está afectado, así como una descripción de en qué o por qué se ve afectado.

Los siguientes dos fragmentos de código hacen referencia a esta funcionalidad:

Éste es el fragmento correspondiente a la activación de un nodo en una incidencia de manera manual, por así decirlo. De esta forma el analista o el product manager deben entrar en el Gestor de Requisitos desde el contexto de una incidencia y, del grafo de nodos, seleccionar los que se vean afectados por esa incidencia. En ese momento se podrá escribir una descripción junto al nombre del nodo, especificando el por qué de ese cambio. Eso es lo que se representa a continuación:

```
//si se edita la columna con el checkbox y es el analista o PM
if (e.Column.Key == "checkboxColumn" && rol<2){
//si marcamos el nodo significa que la incidencia actual le afecta, se
puede añadir descripción
      if (e.Node.Text == "False"){ //si estaba desmarcado significa que
ahora estará marcado
            db.ExecuteCommand("INSERT INTO GRAfecta (IDIncidencia,
            IDNodo) VALUES ({0}, {1})", idIncidencia,
            e.Node.Key.TrimStart('+'));
                  e.Node.Cells[2].AllowEdit = AllowCellEdit.Full;
      }
      else { //si desmarcamos quitamos el nodo de la incidencia
            db.ExecuteCommand("DELETE FROM GRAfecta WHERE IDIncidencia =
            {0} AND IDNodo = {1} ", idIncidencia,
            e.Node.Key.TrimStart('+'));

            e.Node.SetCellValue(ultraTree1.ColumnSettings.ColumnSets[0].
            Columns[2], "");

            e.Node.Cells[2].AllowEdit = AllowCellEdit.ReadOnly;
      }
}
```

El siguiente fragmento hace que los nodos se vean afectados por una incidencia en concreto de forma automática. Esto sucedería de la siguiente manera:

Al entrar el analista o el PM en una incidencia, si en un nodo que inicialmente no se veía afectado por esa incidencia, le introducen propuestas nuevas o de cambios en Pruebas de Aceptación o en Interfaces de Usuario, en ese momento, el nodo quedaría marcado en el grafo. Y el cambio se introduciría en la base de datos, quedando a partir de entonces reflejado que ése nodo ha sido “afectado” por una incidencia.

```
//marca los nodos como afectado en la incidencia si se hace algún
cambio en las PA o capturas
private void contextMenuPropuesta_ItemCliced(object sender,
ToolStripItemClicedEventArgs e){

if (rol < 2){ //solo se marca si es el analista o PM
      if (ultraTree1.ActiveNode.Cells["checkboxColumn"].Text ==
      "False") //estaba desmarcado
      {
            ultraTree1.ActiveNode.Cells["checkboxColumn"].Value = true;

            db.ExecuteCommand("INSERT INTO GRAfecta (IDIncidencia,
            IDNodo) VALUES ({0}, {1})", idIncidencia,
            ultraTree1.ActiveNode.Key.TrimStart('+'));

         ultraTree1.ActiveNode.Cells[2].AllowEdit = AllowCellEdit.Full;
       }
}}
```

Al modelar la base de datos, el diagrama de clases quedaría tal como se muestra en la Figura 49.

La base de datos del módulo se compone de 20 tablas necesarias para almacenar toda la información. Todas ellas, así como sus relaciones, se muestran en dicha figura.

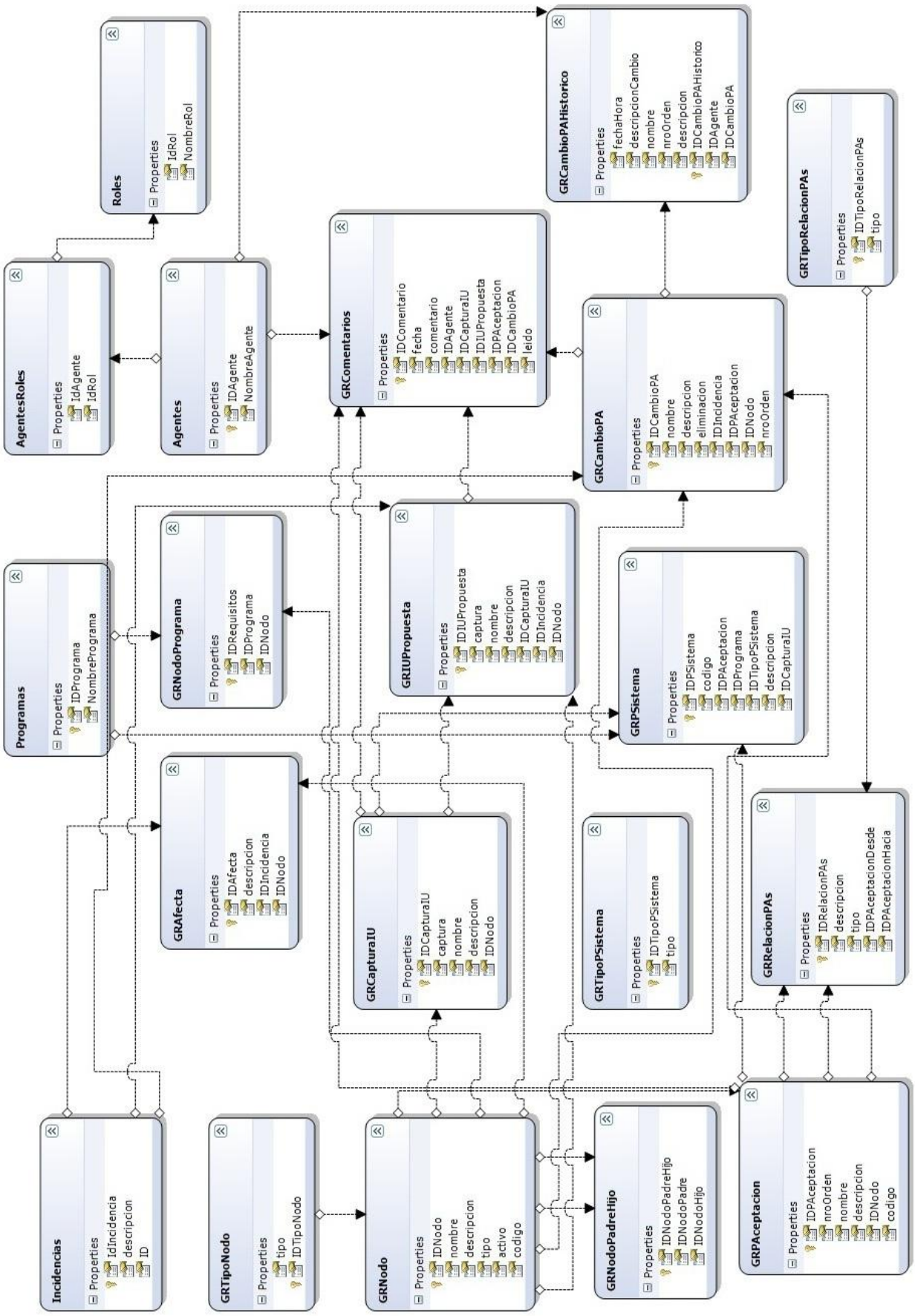


Figura 49. Diagrama de base de datos

4.3 Arquitectura de la solución

La arquitectura del sistema se puede diferenciar en 3 capas:

- Nivel de Presentación

Es el nivel que interacciona con el usuario, presenta el sistema al usuario, le comunica la información y captura la información del usuario dando un mínimo de proceso (realiza un filtrado previo para comprobar que no hay errores de formato). Esta capa se comunica únicamente con el nivel lógico. Está formada por el conjunto de formularios y ventanas que permiten la interacción a los diferentes agentes (al product manager, el analista, etc).

La siguiente captura de pantalla (Figura 50), consistente en el formulario que se usa para consultar, crear y editar las pruebas de aceptación, es un ejemplo donde puede verse el nivel de presentación.

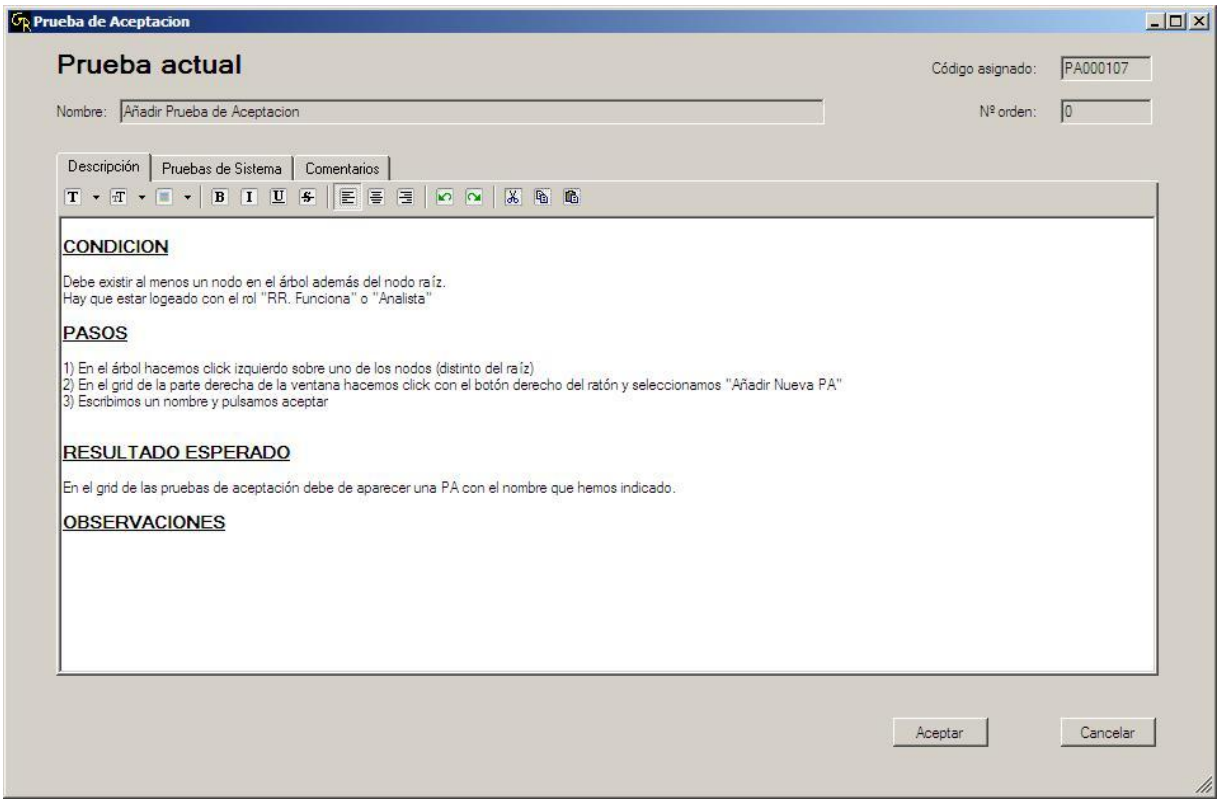


Figura 50. Formulario pruebas de aceptación

El resto de formularios se pueden encontrar en el subcapítulo 3.2 Gestión de requisitos en TUNE-UP.

- Nivel Lógico

Implementa las operaciones descritas en los subcapítulos anteriores. Es donde residen los programas que se ejecutan, recibiendo las peticiones del usuario y enviando las respuestas tras el proceso. Se denomina también capa de negocio, pues es aquí donde se establecen todas las reglas que deben cumplirse. Esta capa se comunica con la capa de presentación, para recibir las solicitudes y presentar los resultados, y con la capa de datos (nivel de persistencia), para solicitar al gestor de base de datos para almacenar o recuperar datos de él.

- Nivel de Persistencia

En él, residen los datos del módulo. Está formada por un gestor de bases de datos que realiza todo el almacenamiento de datos, y recibe solicitudes de almacenamiento o recuperación de información desde el nivel lógico.

Para el acceso a la base de datos se ha optado por usar tres tecnologías diferentes, aprovechando las ventajas de cada una:

- Para el rellenado de los grids se ha hecho uso de DataSets (o conjuntos de datos), usando las facilidades que aportan sus DataTables y DataAdapters (Figura 51)

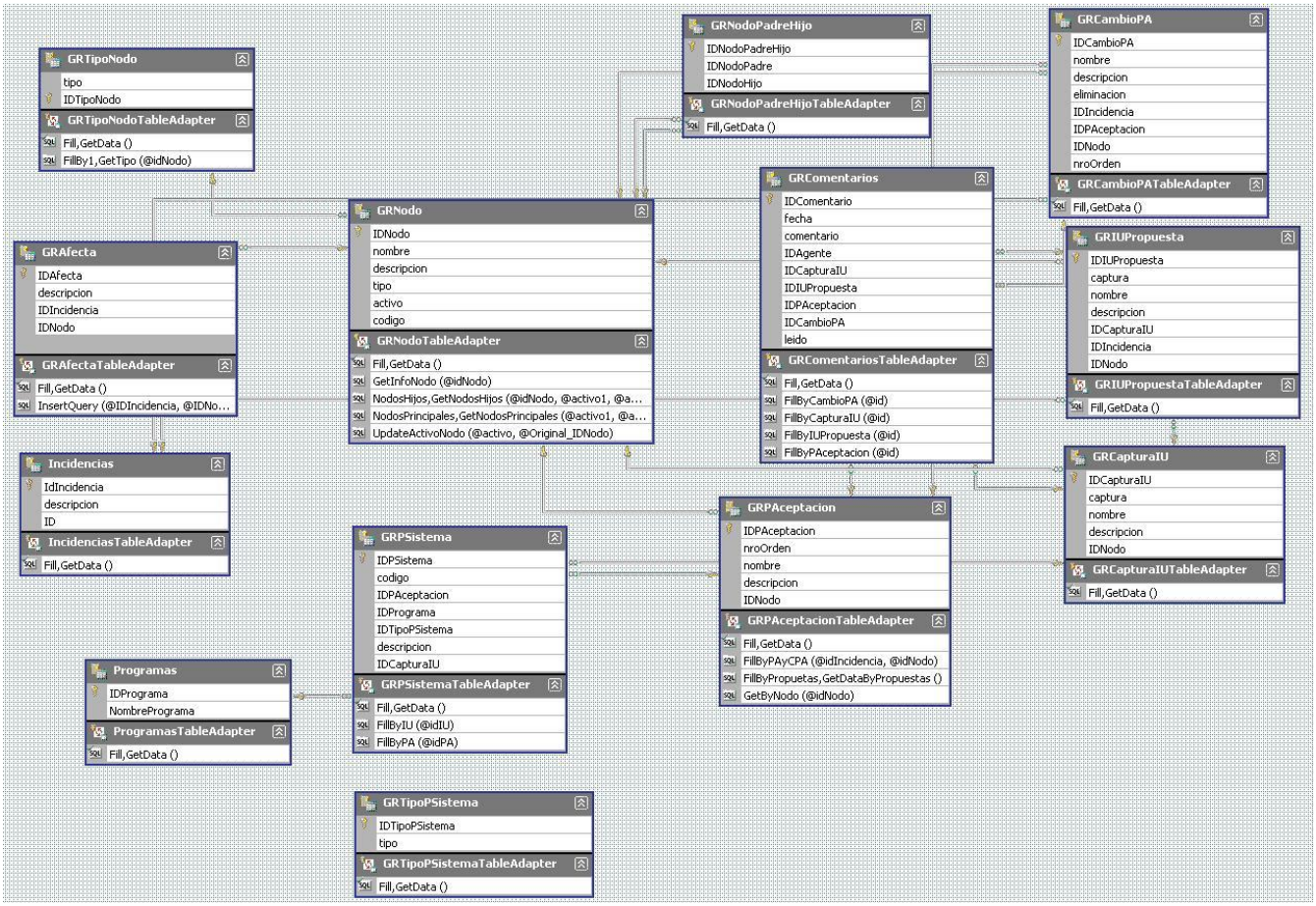


Figura 51. Datasets

De esta forma sólo hay que asociar el grid a una fuente de datos, y rellenar los datos usando el método necesario (los que se pueden ver en la Figura 51), tal como se muestra en esta línea de código:

```
this.gRComentariosTableAdapter.FillByCambioPA(this.gRequisitosDataSet.G
RComentarios, idPrueba);
```

Para otras consultas, normalmente necesarias para completar las funciones de la lógica de negocio, se ha usado la tecnología LINQ y LINQ-to-SQL (comentaremos más sobre LINQ en el siguiente subcapítulo, 4.4 Tecnología utilizada.

Un ejemplo de consulta con LINQ y su uso sería el siguiente:

```
var propuesta = from prop in db.GRCambioPA
                where prop.IDCambioPA == idPrueba
                select new
                  { prop.nombre, prop.descripcion, prop.nroOrden };

textBox1.Text = propuesta.First().nombre;
richTextBoxExtended1.RichTextBox.Rtf = propuesta.First().descripcion;
textBox4.Text = propuesta.First().nroOrden.ToString();
```

- Para realizar inserciones, eliminaciones y modificaciones de la base de datos, se ha usado el lenguaje SQL plano, haciendo los comandos a mano; aunque a través del objeto de clases mapeable que crea LINQ-to-SQL, en el siguiente ejemplo "*db*":

```
db.ExecuteCommand("DELETE FROM GRNodoPadreHijo WHERE IDNodoHijo = {0}
AND IDNodoPadre = {1}", aNode.Key.TrimStart('+'),
aNode.Parent.Key.TrimStart('+'));
```

El motivo de haber usado tres métodos diferentes para interactuar con la base de datos es, entre otros, debido al aprendizaje. Es decir: según íbamos avanzando en el desarrollo, íbamos aprendiendo más sobre .Net y sus características y aplicando estos nuevos conocimientos según conviniera. De esta forma, de partida ya sabíamos hacer consultas SQL planas, pero aprendimos a usar los DataSets y los usamos al inicio, aunque poco después dimos con LINQ, el cuál finalmente fue al que más uso dimos, tal vez por ser la tecnología más moderna de la versión 3.5 de .Net ha sido la mejor, por su comodidad y simplicidad.

A continuación mostraremos dos fragmentos de código de cómo debería ser el código, de empezar de nuevo con el desarrollo. Dichos fragmentos son los involucrados en la funcionalidad de añadir nodos al grafo.

El primero corresponde al código que podríamos encontrar en el formulario de añadir Nuevo Nodo (Figura 34) y los siguientes fragmentos, al código con el que se conecta el primero.

```
public partial class FormNuevoNodo : Form{
    GRNodo nuevoNodo = new GRNodo();
    private string idPadre;

    public FormNuevoNodo(string keyNodoPadre){
         InitializeComponent();
         idPadre = keyNodoPadre;
         textBox3.Text = GRNodo.generaCodigoRequisito();
    }

    public GRNodo getNuevoNodo(){
         return nuevoNodo;
    }

    private void FormNuevoNodo_Load(object sender, EventArgs e){
     this.tipoNodoTableAdapter.Fill(this.gRequisitosDataSet.GRTipoNodo);
    }

    private void botonAceptar_Clic(object sender, EventArgs e){
         nuevoNodo = GRNodo.insertarNodo(textBox1.Text, textBox2.Text,
         (short)comboBox1.SelectedValue, checkBox1.Checked,
         textBox3.Text);

         if (int.Parse(idPadre.TrimStart('+')) > 0){
            //es hijo de un nodo

            //añadimos la relacion padre - hijo
            GRNodoPadreHijo.insertarRelacionPadreHijo(idPadre.TrimStart(
            '+'), nuevoNodo.IDNodo);

            //añadimos el nodo en los mismos programas donde estaba el
            padre
            GRNodoPrograma.insertarNodoEnProgramasDelPadre(int.Parse(idP
            adre.TrimStart('+')), nuevoNodo.IDNodo);
          }
          else //es hijo de un programa
            GRNodoPrograma.insertarNodoEnPrograma(int.Parse(idPadre.Trim
            Start('-')), nuevoNodo.IDNodo);

          this.DialogResult = DialogResult.OK;
          Close();
      }
}
```

Creamos un objeto de la clase “GRNodo”, con el código (identificador) autogenerado que le corresponde.

Al rellenar el usuario la información referente al nodo y pulsar el botón aceptar del formulario, se guardará dicha información en la base de datos, llamando al método “insertarNodo” de la clase “GRNodo”.

Comprobamos si el nuevo nodo es hijo de otro nodo padre, en caso de serlo, creamos la relación. Esto se hace mediante la clase “GRNodoPadreHijo” encargada de estas

relaciones. De no ser así, colgaría del nodo raíz, es decir, el nombre del programa sobre el que estemos trabajando; para eso usaremos la clase “GRNodoPrograma”.

La clase GRNodo contiene todos los atributos de los nodos. La usamos para crear los objetos de tipo nodo para trabajar con ellos, generar su código e insertarlos en la base de datos.

```
partial class GRNodo{
     public static string generaCodigoRequisito(){
         DataClasses1DataContext db = new DataClasses1DataContext();

         //el último nodo que HUBO más el incremento
         int idFuturoNodo =
         (int)db.UltimoID("GRNodo").Single().lastID.Value  + 1;
         string codigo = "N";

         //añadimos 0s hata que tenga 6 cifras en total
         while (codigo.Length < 6 - idFuturoNodo.ToString().Length)
             codigo += 0;
         return codigo += idFuturoNodo.ToString();
     }

     public static GRNodo insertarNodo(string nombre, string
     descripcion, short tipo, bool activo, string codigo)
     {
          DataClasses1DataContext db = new DataClasses1DataContext();
          db.ExecuteCommand("INSERT INTO GRNodo (nombre, descripcion,
          tipo, activo,codigo) VALUES ({0},{1},{2},{3},{4})",
          nombre, descripcion, tipo, activo, codigo);

          return db.GRNodo.Single(n => n.IDNodo == db.GRNodo.Max(id =>
          id.IDNodo));
     }
}
```

La clase “GRNodoPadreHijo” se encarga de las relaciones jerárquicas entre los nodos padres e hijos.

```
partial class GRNodoPadreHijo{
    public static void insertarRelacionPadreHijo(string IDNodoPadre,
    int IDNodoHijo){
         DataClasses1DataContext db = new DataClasses1DataContext();

         db.ExecuteCommand("INSERT INTO GRNodoPadreHijo (IDNodoPadre,
         IDNodoHijo) " + "VALUES ({0},{1}) ", IDNodoPadre, IDNodoHijo);
    }
}
```

La clase “GRNodoPrograma” tiene dos misiones: una es añadir los nodos que no tienen padre en la tabla de la base de datos del mismo nombre (GRNodoPrograma), usada para la primera iteración al crear el grafo de nodos.

La otra misión es, para el caso de los nodos que sí que tengan padre, añadirlos en la base de datos junto a todos los programas a los que pertenezca el padre, ya que si cuelgan del padre, deben estar en los mismos programas en los que éste se encuentra.

```
partial class GRNodoPrograma{
    public static void insertarNodoEnProgramasDelPadre(int idPadre, int
    idNodo)
    {
        DataClasses1DataContext db = new DataClasses1DataContext();
        var programas =  from prg in db.GRNodoPrograma
                         where prg.IDNodo == idPadre
                         select prg.IDPrograma;

        foreach (var programa in programas)
            db.ExecuteCommand("INSERT INTO GRNodoPrograma (IDPrograma,
            IDNodo) VALUES ({0},{1})", programa, idNodo);
     }

     public static void insertarNodoEnPrograma(int idPrograma, int
     idNodo)
     {
        DataClasses1DataContext db = new DataClasses1DataContext();
        db.ExecuteCommand("INSERT INTO GRNodoPrograma
        (IDPrograma,IDNodo) VALUES ({0},{1}) ", idPrograma, idNodo);
     }
}
```

4.4 Tecnología utilizada

El módulo se ha desarrollado usando el lenguaje de programación C#, con el entorno de desarrollo Microsoft Visual Studio 2008 Professional SP1.

La base de datos con la que se conecta se encuentra en un servidor con SQL Server 2008, la cual se administra con la herramienta gráfica SQL Server Management Studio 2008 (SSMSE).

Para el desarrollo inicial del Diagrama de Clases fue útil el uso de StarUML [51]. Éste siguió el proceso de desarrollo hasta el final, siendo necesario recurrir de nuevo a él en varias ocasiones para modificar el diagrama según se iba perfilando el módulo del Gestor de Requisitos.

Con el fin de aumentar el potencial de C#, y facilitar la conexión con la base de datos, se ha usado la tecnología de LINQ (Language Integrated Query) [52]**.** A groso modo lo que hace LINQ es agregar consultas nativas semejantes a las de SQL a los lenguajes de la plataforma .NET, inicialmente a los lenguajes Visual Basic .NET y C#.

Cabe recalcar el uso de esta tecnología innovadora en el proyecto, pues supuso todo un desafío ya que cuando se utilizó era relativamente nueva. Hacía apenas un año que había aparecido en el mercado (junto con la versión 3.5 del .NET Framework) y la documentación sobre ésta era escasa (sólo las guías básicas que Microsoft distribuyó inicialmente).

Trataremos de explicarla aquí lo mejor posible, con ejemplos sencillos y cómo lo hemos utilizado en nuestro código.

LINQ define operadores de consulta estándar que permiten a lenguajes habilitados con LINQ filtrar, enumerar y crear proyecciones de varios tipos de colecciones usando la misma sintaxis. Tales colecciones pueden incluir arreglos, clases enumerables, XML [53], conjuntos de datos desde bases de datos relacionales y orígenes de datos de terceros.

Aunque LINQ soporta inicialmente consultas en colecciones en memoria, bases de datos relacionales y datos XML, es una arquitectura extensible que permite a desarrolladores de orígenes de datos adicionales el uso del LINQ. Así, implementando los operadores de consulta estándar como métodos extensores para sus orígenes de datos, o mediante la implementación de la interfaz IQueryable [54], permite convertir un árbol de expresión en tiempo de ejecución para transformarlo en algún lenguaje de consultas.

Los operadores de consulta estándar son usados también para objetos y permiten consultarlos en la memoria con la misma sintaxis LINQ.

LINQ usa varias características nuevas para permitir a lenguajes como C# el uso de la sintaxis de consultas nativas: tipos anónimos, métodos extensores, expresiones lambda, árboles de expresión, operadores de consulta estándar.
Estas características hacen que la consulta de datos sea un concepto de primera clase.

Veámoslas de una en una:

- Tipos anónimos

Los tipos anónimos nos permiten trabajar con resultados de consultas al vuelo, es decir, sin tener que definir explícitamente clases para representarlos. Cuando el compilador encuentra algo como:

```
var capturasOriginales = from capturaIU in db.GRCapturaIU
                         where capturaIU.IDNodo == int.Parse(idNodo)
                         select new { capturaIU.captura,
                        capturaIU.nombre, capturaIU.IDCapturaIU };
```

Transparentemente crea una nueva clase con tres propiedades, una para cada parámetro de la sentencia “new”.

Hay que recordar que los tipos anónimos por sí mismos no pueden referenciarse desde el código. ¿Cómo es posible acceder al resultado de una consulta si no se sabe el nombre del nuevo tipo? El compilador se encarga infiriendo el tipo.

- Métodos extensores

Como el nombre implica, los métodos extensores extienden los tipos de .NET con nuevos métodos. Por ejemplo, usando los métodos extensores con un string, es posible añadir un nuevo método que convierta cada espacio de un string en un subrayado.

- Expresiones lambda

Esta característica simplifica el código de los delegados y los métodos anónimos.
Una de las lambda expresiones más sencillas que podríamos encontrar en nuestro código sería la siguiente:

```
string comentariosTotales = db.GRComentarios.Count(c => c.IDCambioPA ==
(int)row.Cells["IDCambioPA"].Value).ToString();
```

En ella, calculamos el total de comentarios que tiene una Prueba de Aceptación determinada.

Las expresiones lambda nos permiten escribir funciones que se pueden pasar como argumentos a métodos, por ejemplo, suministrando predicados para una posterior evaluación.

Otra ventaja de las expresiones lambda es que te dan la habilidad de ejecutar análisis de expresiones usando árboles de expresión.

- Árboles de expresión

LINQ puede tratar expresiones lambda como datos en tiempo de ejecución. El tipo “Expression<T>“ representa un árbol de expresión que puede ser evaluado y cambiado en tiempo de ejecución. Es una representación jerárquica de datos en memoria, donde cada nodo del árbol es parte de la expresión de consulta entera. Habrá nodos representando las condiciones, la parte izquierda y derecha de la expresión, etc.

Los árboles de expresión hacen posible personalizar la forma en la que LINQ trabaja cuando construye consultas. Por ejemplo, un proveedor de bases de datos que no soporte nativamente LINQ, podría proveer librerías para traducir las expresiones de árboles de LINQ en consultas de la base de datos.
En nuestro caso esto no fue necesario, ya que como hemos mencionado usamos una base de datos SQL Server 2008, que soportaba LINQ y toda su funcionalidad.
De todas formas, un ejemplo de cómo representar una lambda expresión con un árbol de expresión podría ser este:

```
    Expression<Func<Person, bool>> e = p => p.ID == 1;
    BinaryExpression body = (BinaryExpression)e.Body;
    MemberExpression left = (MemberExpression)body.Left;
    ConstantExpression right = (ConstantExpression)body.Right;

    Console.WriteLine(left.ToString());
    Console.WriteLine(body.NodeType.ToString());
    Console.WriteLine(right.Value.ToString());
```

Primero se define una “ Expression<T> “ variable “ e “, y se asigna a la lambda expresión que quieres evaluar. Después, se obtiene el “cuerpo” de la expresión de la propiedad “Body” del objeto “Expression <T> “. Sus propiedades “Left” y “Right” contienen los operandos izquierdo y derecho de la expresión.
Dependiendo de la expresión, estas propiedades asumirán el tipo expresado en la fórmula. En un caso más complejo no se sabe el tipo a convertir, así que se tiene que usar una expresión “switch” para implementar cualquier caso posible.

```
La salida por pantalla del ejemplo anterior sería:
p.ID
EQ
1
```

El resultado está claro: la propiedad “Left” provee la parte izquierda de la expresión, la cual es: p.ID. La propiedad “Right” provee la parte derecha de la expresión: 1. Finalmente la propiedad “Body”, provee un símbolo describiendo la condición de la expresión. En este caso EQ, el cual se refiere a “equals”.

- Operadores de consulta estándar

LINQ provee de una API conocida como “Standard Query Operations” (sQOs) para soportar los tipos de operaciones a los que estamos acostumbrados en SQL. A continuación daremos un ejemplo sencillo para cada tipo de operador de LINQ:

- Operador de restricción (where)

```
public void Linq1(){
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    var lowNums =
        from n in numbers
        where n < 5
        select n;

    Console.WriteLine("Números < 5:");
    foreach (var x in lowNums){
        Console.WriteLine(x);
    }
}
```

```
El resultado de esta función son los números menores que 5, en este
orden:
4, 1, 3, 2, 0
```

- Operador de proyección (select)

```
public void Linq6(){
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    var numsPlusOne =
        from n in numbers
        select n + 1;

    Console.WriteLine("Números + 1:");
    foreach (var i in numsPlusOne){
        Console.WriteLine(i);
    }
}
```

```
La salida será la cadena de entrada + 1:
6, 5, 2, 4, 10, 9, 7, 8, 3, 1
```

- Operador de partición (take)

```
public void Linq20(){
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    var first3Numbers = numbers.Take(3);
    Console.WriteLine("Primeros 3 números:");

    foreach (var n in first3Numbers){
        Console.WriteLine(n);
    }
}
```

```
El resultado serán los 3 primeros números:
5, 4, 1
```

- Operador de orden (OrderBy)

```
public void Linq28(){
    string[] words = { "cereza", "manzana", "pera" };
    var sortedWords =
        from w in words
        orderby w
        select w;

    Console.WriteLine("Lista ordenada de palabras:");
    foreach (var w in sortedWords){
        Console.WriteLine(w);
    }
}
```

```
La lista ordenada de palabras será:
cereza, manzana, pera
```

- Operador de agrupación (GroupBy)

```
public void Linq41(){
    string[] words = { "manzana", "mono", "rosa", "magenta", "rojo",
"rocío" };

    var wordGroups =
        from w in words
        group w by w[0] into g
        select new { FirstLetter = g.Key, Words = g };

    foreach (var g in wordGroups){
        Console.WriteLine("Palabras que empiezan con la letra '{0}':",
g.FirstLetter);

        foreach (var w in g.Words){
            Console.WriteLine(w);
        }
    }
}
```

```
La salida por pantalla sería:

Palabras que empiezan con la letra 'm':
manzana
mono
magenta
Palabras que empiezan con la letra 'r':
rosa
rojo
rocío
```

- Operador de conjuntos (distinct)

```
public void Linq46(){
    int[] factorsOf300 = { 2, 2, 3, 5, 5 };
    var uniqueFactors = factorsOf300.Distinct();
    Console.WriteLine("Prime factors of 300:");
    foreach (var f in uniqueFactors){
        Console.WriteLine(f);
    }
}
```

```
El resultado sería:
2, 3, 5
```

- Operador de conversión (ToArray)

```
public void Linq54(){
    double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
    var sortedDoubles =
        from d in doubles
        orderby d descending
        select d;

    var doublesArray = sortedDoubles.ToArray();
    Console.WriteLine("Every other double from highest to lowest:");
    for (int d = 0; d < doublesArray.Length; d += 2){
        Console.WriteLine(doublesArray[d]);
    }
}
```

```
Salida:
4.1
2.3
1.7
```

- Operador de elementos (first)

```
public void Linq59(){
    string[] strings = { "cero", "uno", "dos", "tres", "cuatro",
"cinco", "seis", "siete", "ocho", "nueve" };

    string startsWithO = strings.First(s => s[0] == 'u');
    Console.WriteLine("String empezando por 'u': {0}", startsWithO);
}
```

```
Salida:
String empezando por 'u': uno
```

- Operador de generación (repeat)

```
public void Linq66(){
    var numbers = Enumerable.Repeat(7, 3);
    foreach (var n in numbers){
        Console.WriteLine(n);
    }
}
```

```
Salida:
7
7
7
```

- Cuantificadores

```
public void Linq67(){
    string[] words = { "creer", "nacer", "recibir", "campo" };
    bool rAfterE = words.Any(w => w.Contains("er"));
    Console.WriteLine("Hay una palabra en la lista que contiene 'er':
{0}", rAfterE);
}
```

```
Salida:
Hay una palabra en la lista que contiene 'er': True
```

- Operador de agregación

```
public void Linq74(){
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    int oddNumbers = numbers.Count(n => n % 2 == 1);
    Console.WriteLine("Hay {0} números impares en la lista.",
oddNumbers);
}
```

```
Salida:
Hay 5 números impares en la lista.
```

- Otros operadores

```
public void Linq96(){
    var wordsA = new string[] { "cereza", "manzana", "pera" };
    var wordsB = new string[] { "cereza", "manzana", "pera" };
    bool match = wordsA.SequenceEqual(wordsB);
    Console.WriteLine("La secuencia es igual: {0}", match);
}
```

```
Salida:
La secuencia es igual: True
```

Para más ejemplos de operadores, se pueden consultar en [55].

Los desarrolladores pueden usar LINQ con cualquier fuente de datos. Pueden expresar consultas eficientemente en los lenguajes de programación que elijan, opcionalmente transformar/incrustar los resultados de las consultas en el formato que quieran, y entonces manipular fácilmente los resultados. Los lenguajes habilitados para LINQ pueden aportar seguridad de tipos y chequeo en tiempo de compilación en las expresiones de consulta, y desarrollar herramientas que aporten intelisense, debugging, y un gran soporte para refactoring cuando escriban código de LINQ.

Adicionalmente a LINQ, también hemos utilizado la tecnología de LINQ to SQL. Se trata de una implementación de O/RM (Object Relational Mapping) [56] que nos permite modelar bases de datos relacionales con clases de .NET. Podemos consultar bases de datos con LINQ, así como actualizar/añadir/borrar datos de ellas.

El Diseñador Relacional de Objetos proporciona una superficie de diseño visual para crear clases de entidad y asociaciones (relaciones) de LINQ to SQL basadas en los objetos de una base de datos. Es decir, el Diseñador relacional de objetos se usa para crear un modelo de objetos en una aplicación que se asigna a los objetos de una base de datos. También genera una clase "DataContext" con establecimiento inflexible de tipos que se usa para enviar y recibir datos entre las clases de entidad y la base de datos. El O/RM también proporciona la funcionalidad para asignar los procedimientos almacenados y funciones a los métodos de "DataContext" con el fin de devolver datos y rellenar las clases de entidad. Por último, el O/RM permite diseñar relaciones de herencia entre las clases de entidad.

El O/RM genera el archivo .dbml que proporciona la asignación entre las clases de LINQ to SQL y los objetos de base de datos, la Figura 49 corresponde a una captura de dicho archivo. El O/RM también genera las clases "DataContext" con tipo y las clases de entidad.

El O/RM tiene dos áreas distintas en su superficie de diseño: a la izquierda, el panel de entidades y, a la derecha, el panel de métodos. El panel de entidades es la superficie de diseño principal que muestra las clases de entidad, asociaciones y jerarquías de herencia. El panel de métodos es la superficie de diseño que muestra los métodos "DataContext" que están asignados a procedimientos almacenados y funciones.

Capítulo 5. Conclusiones y trabajo futuro

Se ha desarrollado el módulo del Gestor de Requisitos y se ha implantado en la herramienta de TUNE-UP (para cubrir todo el proceso de desarrollo de software) y en la empresa. Después, y tras haber validado la efectividad del módulo, podemos decir que el Gestor de Requisitos cumple las expectativas para las que fue diseñado. Aunque al principio hubo quien se mostró reacio a dejar los documentos en Word (pese a la problemática que ello suponía), no fue así al ser finalmente acogido el módulo, donde el personal se mostró agradecido con las nuevas herramientas y posibilidades. Aunque todavía no se hayan derivado todas las pruebas de la empresa al GR, las pruebas de aceptación están ahora organizadas de manera más clara e intuitiva. Las características del GR están totalmente orientadas al tratamiento de las pruebas de aceptación, y el mantenimiento de las pruebas se hace ahora de manera más eficaz.

Aunque la creación del módulo se llevó a cabo en el tiempo establecido (durante 6 meses), no fue así con la implantación, habiendo sido ésta más lenta de lo esperado inicialmente.

Los motivos de esta lenta implantación han sido varios, incluyendo tanto problemas como desafíos aparecidos:

Para empezar, hubo que dedicar un tiempo considerable a la formación del equipo, dicho equipo consta de 3 analistas, 8 programadores y 6 testers.

El volumen del proyecto que se maneja en la empresa es de entre 50 y 100 cambios en el proyecto por versión. Esto hace que se tarde más en realizar la implantación por completo. No es recomendable pasar directamente todos los cambios al Gestor de Requisitos, así que se optó por hacerlo de manera progresiva, pasando incidencias al GR poco a poco. Actualmente en la empresa hay un 20% de las pruebas en el Gestor de Requisitos y el otro 80% sigue usándose sin el gestor.

Como problema surgió que el Gestor de Requisitos estaba demasiado enfocado al rol del analista. Dado que el equipo lo componen también programadores y testers, y cada uno de ellos tiene unas demandas particulares, hubo que acomodar el gestor para los otros agentes. Dicha “acomodación” se hizo ad-hoc, mediante el uso de una nueva interfaz, cuyo desarrollo no fue trivial, debido a que tenía que estar bien integrado.

Un inconveniente que hubo en el desarrollo es que se atendieron los requisitos de un único analista como cliente. Dichos requisitos cada vez eran más específicos a la par que complicados. Al iniciar la implantación del módulo surgieron las necesidades de otros analistas. Esto ayudó a ver realmente cuáles eran las mejoras más notables e importantes para todos.

Como desafío cabe notar la petición de muchas ampliaciones por parte de los agentes. Una especialmente destacable sería la de explotar el módulo del Gestor de Requisitos en el ámbito del seguimiento del proyecto. Esto es, ver claramente en qué porcentaje

está finalizado el producto, lo cual es una gran ayuda desde el punto de vista del Product Manager para la planificación del desarrollo del proyecto. Este seguimiento consiste en comprobar, para cada prueba de aceptación, y para cada agente que tiene que ver con ella (en el siguiente orden gradual: programador, tester, automatización tester, soporte) si ha terminado de trabajar con la prueba o no. En caso de haberla pasado todos, la prueba de aceptación estaría completa en un 100%. Teniendo en cuenta el estado de todas las pruebas de aceptación se puede extraer una idea del nivel de desarrollo completo del proyecto.

En la siguiente tabla se ilustra un ejemplo de cómo sería este seguimiento.

	Programador	Tester	Aut. tester	Soporte
PA1	OK?	OK?	OK?	KO
PA2	OK	OK	OK	OK

Tabla 3. Ejemplo seguimiento versión

A su vez, dicho seguimiento y el resto de ampliaciones demandadas por los usuarios sirve para dar a entender la buena acogida y convencimiento de todos los agentes respecto al módulo, que han adoptado con motivación y buenas expectativas.

No podemos determinar una evaluación cuantitativa, puesto que de momento no tenemos suficientes datos como para determinar, por ejemplo, el aumento de la productividad del equipo. Entre otras cosas, esto es debido a que la implementación empezó a finales del 2009 y a día de hoy, todavía se encuentra en fase de implantación. Sin embargo, sí que podemos determinar con total seguridad que el Gestor de Requisitos ha provisto al equipo de una ventaja, además de nueva experiencia.

El desarrollo llevado a cabo para esta tesis, no es más que un primer paso hacia lo que podría ser en el futuro el gestor de requisitos más importante para la metodología de trabajo aquí presentada. De modo que sus futuras ampliaciones son muchas, ya que podría convertirse en un producto muy potente.

A continuación se citan algunas de estas posibles futuras ampliaciones:

- El módulo debería permitir el versionado. Esta primera versión del módulo, sólo permitiría gestionar una primera iteración de un proyecto software; pero en el futuro debería incorporar la capacidad de versionado, es decir, poder indicar que se ha finalizado la versión, y se produzcan los cambios adecuados en los nodos y las pruebas de aceptación.

- Histórico de cambios en Incidencias. Esto permitiría mantener un registro actualizado de las incidencias que han aparecido y desaparecido y los cambios que se han llevado a cabo. Actualmente si se propone una incidencia de

cambio, una vez cambiado no queda rastro de lo anterior, y puede ser útil saber cómo era al menos una versión anterior.

- Búsquedas específicas. Actualmente no se permite más que realizar búsquedas por nombres o por código de Nodos, y en todo caso se pueden usar los filtros en los grids para buscar por contenido; pero lo verdaderamente adecuado sería poder realizar la búsqueda entre todos los nodos de todas las pruebas de aceptación, por su descripción, nombre, código…

- Diagramas de Actividad o de Estado. Como se explica, las pruebas de aceptación son adecuadas para especificar los requisitos, pero en algunos casos es muy útil usar diagramas de actividad o de estado para aclarar algunos conceptos. Añadir un apartado en el módulo en el que tuvieran cabida estos diagramas sería muy adecuado.

- Nuevo tipo de nodo: Componente Interno. Además de los tipos ya existentes (Requisito Funcional, Requisito no Funcional e Interfaz de usuario), sería útil identificar algunos nodos como componentes internos.

Finalmente quisiera destacar la experiencia profesional que ha significado para mí el desarrollo de esta tesis. Desde que terminé la carrera ha sido el primer trabajo real que he realizado, así como la primera participación en una empresa, con la que mantuve contacto a través de mi director, Patricio Letelier.

Durante un cuatrimestre he estado trabajando desarrollando y mejorando la metodología TUNE-UP a través del Gestor de Requisitos. Además, hemos podido comprobar la integración parcial en la empresa, así como interactuar con los usuarios de la herramienta, pese a que la implantación del Gestor de Requisitos en TUNE-UP ha sido difícil puesto que nos hemos encontrado con muchos agentes reacios al cambio debido a la costumbre que tenían tras varios años de proceder de la misma manera con los documentos de Word. Aun así hemos podido clasificar sus opiniones al respecto y recoger ideas para futuras mejoras y ampliaciones.

Actualmente, tanto la herramienta como la metodología, que incluye el Gestor de Requisitos, están fuertemente implantadas y resultan indispensables para organizar los proyectos que llevan a cabo los agentes.

La realización de esta tesis, además de permitir al autor trabajar en un proyecto real en colaboración con una PYME de desarrollo de software, ha tenido que profundizar en tecnologías de programación como LINQ, en diferentes enfoques y las herramientas que las soportan; mejorando notablemente las habilidades como desarrollador, aumentando el nivel de conocimientos de .Net así como de metodologías para la gestión de requisitos.

En definitiva, mi estancia en la empresa puedo considerarla además de provechosa en mi aprendizaje como informático, satisfactoria en la parte personal. En todo momento había un ambiente correcto para la realización de mi trabajo. Y existía un ambiente agradable entre los compañeros de trabajo que hacía más llevadero el trabajo. Otra de las cosas que más destaco de la empresa es las facilidades que me ofrecieron para poder seguir estudiando, gracias a esto he podido sacar adelante la carrear sin ningún tipo de problema.

Referencias

[1] M., Marante, Letelier P., and Company M. "Gestión de requisitos basada en pruebas de aceptación: Test-Driven en su máxima expresión." Edited by Universidad Politécnica de Valencia. *XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2010).* Valencia, Septiembre de 2010.

[2] Haugset, B, and Hanssen G. "Automated Acceptance Testing: A Literature Review and an Insutrial Case Study." *Proc. of Agile.* 2008. 27-38.

[3] P., Kroll; P., Kruchten; G., Booch. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Addison-Wesley Professional, 2003.

[4] K., Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[5] K., Forsberg, and Harold M. "The Relationship of Systems Engineering to the Project Cycle." *Engineering Management Journal* 4, no. 3 (1992): 36-43.

[6] 830-1993, IEEE. "IEEE Recommended Practice for Software Requirements Specifications."

[7] Sommerville, Ian. *Ingeniería del Software. Séptima edición.* Pearson Addison Wesley, 2006.

[8] Corbett, Nancy. *Test Driven Development, a Portable Methodology.* QuinStreet Inc. Julio 26, 2006. http://www.developer.com/tech/article.php/3622546/Test-Driven-Development-a-Portable-Methodology.htm (accedido Abril 2009).

[9] Mackinnon, Tim. *XUnit Testing – A plea for assertEquals.* Londres, Inglaterra: Connextra Ltd., 2000.

[10] Martin, Robert C. *JUnit.org Resources for Test Driven Development.* Object Mentor. 1999. http://www.junit.org/ (accedido Julio 2010).

[11] Beck, Kent. *Test-Driven Development by Example.* Addison Wesley, 2003.

[12] Koskela, Lasse. *Test Driven. Practical TDD and Acceptance TDD for Java Developers.* Manning Publisher, 2007.

[13] Ambler, Scott W. *Agile Modeling. Effective Practices for Extreme Programming and the Unified Process.* John Wiley, 2002.

[14] Elssamadisy, Amr. *Patterns of Agile Practice Adoption. The Technical Cluster.* C4Media Inc., 2007.

[15] Elssamdisy, Amr. *Recognizing and Responding to Change.* http://www.elssamadisy.com/ (accedido Julio 2010).

[16] Fleischer, Georg. *Continuous Integration.* Venlo, Países Bajos: Fontys University of Applied Sciences, 2009.

[17] Cunningham, Ward. *Fit: Framework for Integrated Test.* Cunningham & Cunningham, Inc. Octubre 12, 2002. http://fit.c2.com/ (accedido Julio 14, 2010).

[18] Martin, Robert C. *Object Mentor.* Object Mentor Inc. 2006. http://blog.objectmentor.com/articles/2008/10/02/slim (accedido Julio 14, 2010).

[19] Adams, Tom. *Better Testing Through Behaviour.* Workingmouse, 2007.

[20] North, Dan. *Behaviour-Driven Development.* http://behaviour-driven.org/

(accedido Abril 2009).

[21] North, Dan. "Better Software." (StickyMinds) I (Marzo 2006).

[22] Goh, Tim. *Revelling in unraveling complexity.* ProgProg. Agosto 18, 2007. http://www.progprog.com/articles/2007/08/17/tdd-vs-bdd (accedido Abril 2009).

[23] Astels, Dave. *A new look at Test-Driven Development.* Asterls Consulting, 2007.

[24] Hruby, Pavel. "Domain-Driven Development with Ontologies and Aspects." Vedbaek, Denmark: ACM, 2005.

[25] North, Dan, Liz Keogh, Mauro Talevi, Paul Hammant, and Shane Duan. *JBehave.* jbehave.org. 2008. http://jbehave.org/ (accedido Julio 2010).

[26] Chelimsky, David, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, and Dan North. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends.* Noruega: The Pragmatic Bookshelf, 2010.

[27] Haughton, Tim. *NSpec.* collabnet. 2007. http://nspec.tigris.org/ (accedido Julio 2010).

[28] Boal, John E. *Test Driven Developer.* Abril 1, 2008. http://testdrivendeveloper.com/2008/04/02/AcceptanceTestDrivenDevelopmentExplained.aspx (accedido Abril 2009).

[29] Taha, Walid. *Domain-Specific Languages.* Houston, USA: Rice University, 2008.

[30] Mitra, T. *Business-driven development.* IBM. 2005. http://www.ibm.com/developerworks/webservices/library/ws-bdd (accedido Julio 2010).

[31] Fielding, R., et al. *Hypertext Transfer Protocol -- HTTP/1.1.* w3, Junio, 1999.

[32] Peterson, David. *Concordion.* 2006. http://www.concordion.org/ (accedido Abril 2010).

[33] Hellesøy, Aslak. *Cucumber.* http://cukes.info/ (accedido Julio 2010).

[34] Crispin, Lisa. "Using Customer Tests to Drive Development." (Methods & Tools) 13, no. 2 (2005).

[35] Nicolette, Dave. *Functional Test Driven Development.* Noviembre 2, 2005. http://www.davenicolette.net/articles/functional_tdd.html.

[36] Mack, Preston. "Better Software." (StickyMinds) I (Julio/Agosto 2004): 18-23.

[37] Ruby, Sam, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails, Third Edition.* Raleigh, North Carolina: Pragmatic Bookshelf, 2009.

[38] Beck, Kent. *Simple Smalltalk Testing: With Patterns.* First Class Software, Inc., 1999.

[39] Poole, Charlie, Jamie Cansdale, and Gary Feldman. *NUnit.* 2002. http://www.nunit.org/ (accedido Julio 2010).

[40] Gold, Russell. *HttpUnit.* 2000. http://httpunit.sourceforge.net/ (accedido Julio 2010).

[41] Henry, Julien, Julien Henry, and Julien Henry. *JWebUnit.* JWebUnit team. 2002. http://jwebunit.sourceforge.net/ (accedido Julio 2010).

[42] Martin, Robert C., Micah D. Martin, and Patrick Wilson-Welsh. *FitNesse.* Octubre 2008. http://fitnesse.org/ (accedido Julio 2010).

[43] Antonioli, Denis N., Denis N. Antonioli, and Marc Guillemot. *WebTest.* Canoo Engineering AG. 2002. http://webtest.canoo.com (accedido Julio 2010).

[44] Scott, Alister. *Watir.com.* 2005. http://watir.com/ (accedido Julio 2010).

[45] IBM. *IBM.* http://www-01.ibm.com/software/awdtools/reqpro/ (accedido Noviembre 2011).

[46] INCOSE Requirements Working Group. "INCOSE Requirements Management Tools Survey." *International Council on System Engineering.* 2010. http://www.incose.org/ProductsPubs/products/toolsdatabase.aspx (accedido Diciembre 2010).

[47] Staff, IBM Rational. "RequisitePro technical FAQs overview." IBM, 2004, 9-12.

[48] Goossen, Mark. "An introduction to Rational RequisitePro extensibility." 2003. http://www.ibm.com/developerworks/rational/library/445.html (accedido Noviembre 2010).

[49] Gause, D.; Weinberg, G.. *Exploring Requirements: Quality Before Design.* Dorset House, 1989.

[50] Marante, M, P Letelier, and F Suarez. "TUNE-UP: Un enfoque pragmático para la planificación y seguimiento de proyectos de desarrollo y mantenimiento de software." Valencia: Congreso Iberoamericano SOCOTE - Soporte al Conocimiento con la Tecnología (SOCOTE 2009), Noviembre del 2009.

[51] Cheon, Park Yong, Niklaus Lee, and Kim Juho. *StarUML.* 2005. http://staruml.sourceforge.net/ (accedido Abril 2009).

[52] Fabrice Marguerie, Steve Eichert, Jim Wooley. *Linq In Action.* Wiley India Pvt. Ltd., 2008.

[53] DuCharme, Bob. *XML: The Annotated Specification.* Nueva York: Prentice Hall, 1998.

[54] Mayo, Joe. "LINQ Programming." 222. McGraw-Hill Osborne Media, 2008.

[55] Ferracchiati, Fabio Claudio. *LINQ for Visual C# 2008.* Milan, Italia: Apress, 2008.

[56] Mehta, Vijay P. *Pro LINQ Object Relational Mapping in C# 2008.* Apress, 2008.

[57] Deng, Chengyao. *FitClipse: a Testing Tool for Supporting Executable Acceptance Test Driven Development.* University of Calgary, 2007.

[58] Chengyao, Deng, Patrick Wilson, and Frank Maurer. "FitClipse: A Fit-Based Eclipse Plug-In for Executable Acceptance Test Driven Development." Edited by XP 2007, Como, Italia, Junio 18-22 8th International Conference. *Agile Processes in Software Engineering and Extreme Programming.* Springer Berlin / Heidelberg, 2007. 93-100.

[59] Letelier, P., M. Marante, and F. Suarez. "TUNE-UP: Seguimiento de proyectos software dirigido por la gestión de tiempos." San Sebastián: XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2009), Septiembre del 2009.

[60] Ambler, Scott W. *Agile Database Techniques. Effective Strategies for the Agile Software Developer.* Wiley, 2003.

[61] Astels, David. *Test-Driven Development: A Practical Guide.* Prentice Hall, 2003.

[62] Adams, Tom. *instinct.* 2002. http://code.google.com/p/instinct/ (accedido Abril 2010).

[63] Microsoft Corporation. *C# Language Specification 3.0.* Redmond, Washington, USA: Microsoft , 2007.

[64] Object Management Group, Inc. *OMG.* 1997. http://www.omg.org/ (accedido Agosto 2010).

[65] Anneke G. Kleppe, Jos Warmer, Jos B. Warmer, Wim Bast. *MDA explained: the model driven architecture : practice and promise.* Addison-Wesley, 2003.

[66] Robin Pars, Laurence Moroney, John Grieb. *Foundations of ASP.NET AJAX.* Nueva York: Apress, 2007.

[67] Brown, Samantha. *INCOSE.* INCOSE Foundation. http://www.incose.org/ (accedido Agosto 2010).

ANEXO A: Pruebas de Aceptación

Las Pruebas de Aceptación se agrupan por nodos. Cada nodo hace referencia a un requisito del sistema.

Podemos ver en la siguiente captura del módulo cuál es nuestra jerarquía de nodos:

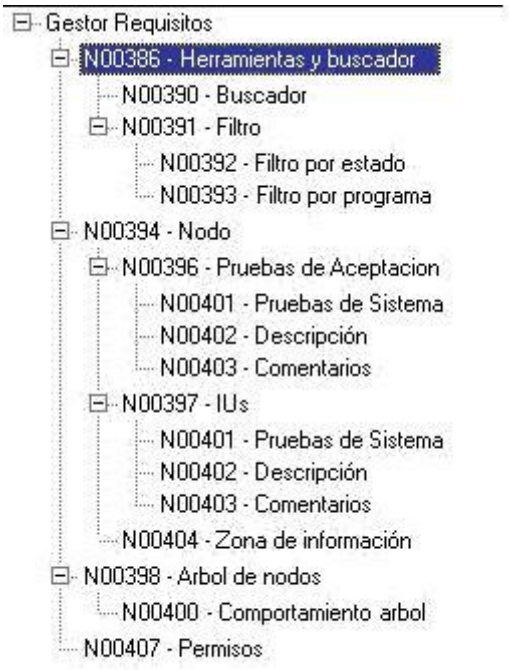


Figura 52. Nodos Gestor Requisitos

Hay que destacar que no solamente se creó el grafo con los nodos del módulo, sino que la especificación de requisitos del módulo se ha hecho al completo (todos los nodos, pruebas e interfaces de usuario) con el mismo módulo en sí. De hecho, todas las pruebas de aceptación que vienen en los siguientes subcapítulos (agrupadas por nodos, tal como se verían en el GR) han sido directamente exportadas desde el Gestor de Requisitos a texto plano para añadirlas a continuación.

8.1 Nodo: Herramientas y buscador

PRUEBA	PA000108 : Expandir y contraer nodos
NODO	N00386 : Herramientas y buscador
CONDICION	
Debe haber nodos añadidos en el grafo, con al menos 2 niveles de jerarquía (nodos con hijos, que tienen hijos a su vez)	
PASOS	
1. En la barra superior, pulsamos en el tooltip "Expandir todo". 2. Pulsamos en el tooltip “Contraer todo”.	
RESULTADO ESPERADO	
PASO 1: Los nodos del grafo deben expandirse, mostrándose todos los nodos y no quedar ningún "+" en el grafo. PASO 2: Los nodos deben contraerse y quedar como estaban inicialmente: mostrándose sólo el nodo del programa al que pertenecen.	
OBSERVACIONES	

PRUEBA	PA000109 : Refrescar todo
NODO	N00386 : Herramientas y buscador
CONDICION	
Debe existir al menos un nodo en el grafo.	
PASOS	
1) Seleccionamos un nodo del grafo. 2) En la zona de información del nodo, cambiamos el nombre. 3) Cambiamos el foco. 4) Pulsamos el tooltip "refrescar todo".	
RESULTADO ESPERADO	
PASO 4: En el grafo debería de mostrarse el nuevo nombre del nodo.	
OBSERVACIONES	

8.1.1 Nodo: Buscador

PRUEBA	PA000095 Selección de nodos buscando por nombre
NODO	N00390: Buscador
CONDICION	
Debe haber nodos añadidos en el grafo, con al menos 2 niveles de jerarquía (nodos con hijos, que tienen hijos a su vez)	
PASOS	
1) En la barra superior, en "Buscar" escribimos una cadena de texto que coincida con parte del nombre de alguno de los nodos del grafo 2) Cambiamos la cadena de texto a una que no coincida con el nombre de ningún nodo del grafo	
RESULTADO ESPERADO	
PASO 1 : En el grafo, deberán quedar marcados y visibles (si estaban contraídos deben expandirse) todos los nodos del grafo que contengan la cadena introducida en el nombre. PASO 2 : Deberán volver a contraerse los nodos que se habían expandido y no quedar ninguno marcado.	
OBSERVACIONES	
No es relevante si la cadena se introduce con mayúsculas o minúsculas. Así mismo, los espacios también deberían ser reconocidos.	

8.1.2 Nodo: Filtro

PRUEBA	PA000110 : Visualización de nodos activos y no activos
NODO	N00392: Filtro por estado
CONDICION	
Debe haber nodos añadidos en el grafo, con al menos 2 niveles de jerarquía (nodos con hijos, que tienen hijos a su vez) Alguno de esos nodos debe encontrarse en el estado "no activo" y el resto como "activo"	
PASOS	
1) En la barra superior, en "Mostrar" elegimos "Nodos Activos". 2) En el mismo desplegable elegimos ahora "Todos los Nodos". 3) Elegimos ahora "Nodos No Activos".	
RESULTADO ESPERADO	
PASO 1: Deben desaparecer del grafo tanto los nodos que se encontraran en el estado "No Activo", como los hijos de éstos. PASO 2: Deben volver a aparecer en el grafo los nodos que habían desaparecido; es decir, se deben volver a mostrar todos los nodos que había inicialmente. PASO 3: Deben desaparecer del grafo tanto los nodos que se encontraran en el estado "Activo", como los hijos de éstos.	
OBSERVACIONES	

PRUEBA	PA000096 : Selección de programa
NODO	N00393 : Filtro por programa
CONDICION	
Deben existir al menos 2 programas en la base de datos, cada uno de ellos con una serie de nodos asociados.	
PASOS	
1) En la barra superior, en "Programa", seleccionamos uno de los programas que aparezcan. 2) En el mismo desplegable, seleccionamos "Todos" 3) Elegimos ahora otro programa diferente al del paso 1.	
RESULTADO ESPERADO	
PASO 1: Deben desaparecer todos los nodos que se mostraban inicialmente, y en su lugar aparecer los relacionados con el programa elegido. PASO 2 : Deben aparecer como nodos raíz todos los programas de la base de datos, y bajo ellos sus nodos relacionados. PASO 3 : Deben desaparecer todos los nodos de nuevo y mostrarse sólo los relacionados con el programa elegido.	
OBSERVACIONES	

8.2 Nodo: Grafo de Nodos

PRUEBA	PA000168 : Añadir descripciones a nodos que afectan a la incidencia
NODO	N00398: Grafo de nodos
CONDICION	
Debe existir un programa con nodos relacionados. Debemos conectar a la aplicación desde el contexto de una incidencia que tenga nodos afectados.	
PASOS	
1) Entrar en la aplicación. 2) Desplegar el grafo 3) Escribimos en el textbox a la derecha del nombre del nodo en el grafo una descripción.	
RESULTADO ESPERADO	
PASO 3 : Debe permitirse la inserción de la descripción, siempre y cuando el nodo esté afectado por la incidencia. La próxima vez que entremos en la aplicación desde el mismo programa y la misma incidencia, las descripciones que pusimos deberían estar.	
OBSERVACIONES	

PRUEBA	PA000166 : Carga de grafo en el contexto de una incidencia
NODO	N00398 : Grafo de nodos
CONDICION	
Debe existir un programa con nodos relacionados y con varios niveles jerárquicos. Debemos conectar a la aplicación desde el contexto de una incidencia. La incidencia debe de tener varios nodos marcados como afectados por la incidencia.	
PASOS	
1) Elegir una incidencia con nodos afectados y entrar en la aplicación. 2) Pulsar botón refrescar	
RESULTADO ESPERADO	
En ambos pasos debe de aparecer la barra de carga mientras se carga el grafo de nodos. Al finalizar la carga deben aparecer en el grafo todos los nodos relacionados con el programa escogido que se encuentren en estado ACTIVO. Deberán aparecer marcados todos los nodos que se vean afectados por la incidencia. Además, deben de aparecer desplegados los nodos que tienen hijos afectados por la incidencia, de forma que todos los afectados sean visibles a primera vista.	
OBSERVACIONES	

PRUEBA	PA000136 : Cargar grafo fuera del contexto de una incidencia
NODO	N00398 : Grafo de nodos
CONDICION	
Debe existir un programa con nodos relacionados.	
PASOS	
1) Entrar en la aplicación. 2) Pulsar botón refrescar	
RESULTADO ESPERADO	
En ambos pasos debe de aparecer la barra de carga mientras se carga el grafo de nodos. Al finalizar la carga deben aparecer en el grafo todos los nodos relacionados con el programa escogido y respetando su jerarquía.	
OBSERVACIONES	

PRUEBA	PA000169 : Desmarcar nodos como afectados por una incidencia
NODO	N00398 : Grafo de nodos
CONDICION	
Debe existir un programa con nodos relacionados. Debemos conectar a la aplicación desde el contexto de una incidencia, que tenga nodos marcados como afectados, algunos de ellos con descripciones asociadas.	
PASOS	
1) Entrar en la aplicación. 2) Desplegar el grafo 3) En un nodo marcado como afectado, pero sin descripción: lo desmarcamos. 4) En un nodo marcado como afectado y con descripción: lo desmarcamos. 5) En el mensaje de advertencia, pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 5 y 6: Debe desaparecer la marca del check y ya no aparecer como afectado. PASO 6: La descripción que tenia el nodo en el grafo debe haber desaparecido.	
OBSERVACIONES	

PRUEBA	PA000167 : Marcar nodos como afectados por incidencia
NODO	N00398 : Grafo de nodos
CONDICION	
Debe existir un programa con nodos relacionados. Debemos conectar a la aplicación desde el contexto de una incidencia.	
PASOS	
1) Entrar en la aplicación. 2) Desplegar el grafo 3) Marcamos el check que sale al lado del nombre de los nodos en el grafo	
RESULTADO ESPERADO	
PASO 3: La próxima vez que entremos en la aplicación desde el mismo programa y la misma incidencia, los nodos que marcamos deben mantenerse marcados.	
OBSERVACIONES	

PRUEBA	PA000101 : Añadir nodo
NODO	N00394 : Nodo
CONDICION	
Debe existir un programa en la base de datos. Hay que estar logueado con rol "RR. Funcional"	
PASOS	
1) Debemos iniciar la aplicación. 2) En el grafo hacemos clic izquierdo sobre el nodo raíz (debe ser el programa que hemos elegido en el paso 1) 3) Hacemos clic derecho y seleccionamos "Añadir Nodo" 4) En la ventana que aparece, escribimos un nombre y pulsamos "Añadir"	
RESULTADO ESPERADO	
PASO 4: En el grafo, bajo el nodo raíz, habrá aparecido un nodo con el nombre que hemos introducido.	
OBSERVACIONES	

8.3 Nodo: Nodo

PRUEBA	PA000105 : Duplicación de nodo
NODO	N00394 : Nodo
CONDICION	
Debe haber varios nodos en el grafo. Tenemos que estar logeados en la aplicación con rol "RR. Funcional".	
PASOS	
1) En el grafo hacemos clic izquierdo sobre uno de los nodos (que no sea el raíz) 2) Volvemos a hacer clic sobre el nodo y sin soltar el botón del ratón lo arrastramos encima de otro de los nodos 3) Soltamos el botón. 4) En la ventana emergente marcamos "Añadir nuevo padre" y pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 4: El nodo que hemos seleccionado debe aparecer ahora como hijo de dos nodos: el del que ya era hijo antes, y el nodo sobre el que lo hemos arrastrado	
OBSERVACIONES	
Si el nodo se arrastra sobre un nodo de tipo programa (de la raíz del grafo, el caso de mostrarse todos los programas a la vez), en el listbox "Pertenece a" deberán estar ahora marcados tanto el programa del que ya era hijo, como el nuevo programa.	

PRUEBA	PA000104 : Eliminación de nodo
NODO	N00394 : Nodo
CONDICION	
Debe haber al menos un nodo (a parte del raíz) en el grafo. Debemos estar logeados con el rol "RR. Funcional"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre un nodo. 2) Hacemos clic derecho y seleccionamos "Eliminar Nodo"	
RESULTADO ESPERADO	
PASO 2: El nodo que hemos seleccionado debe desaparecer del grafo.	
OBSERVACIONES	

PRUEBA	PA000106 : Mover nodo
NODO	N00394 : Nodo
CONDICION	
Debe haber varios nodos en el grafo. Hay que estar logeado con el rol "RR. Funcional".	
PASOS	
1) En el grafo hacemos clic izquierdo sobre uno de los nodos (excepto el raíz) 2) Volvemos a hacer clic sobre el nodo y sin soltar el botón del ratón lo arrastramos encima de otro de los nodos y soltamos el botón. 3) En la ventana emergente pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 3: El nodo que hemos seleccionado, debe haber desaparecido como hijo del que era y ser ahora hijo del nodo sobre el que se arrastró. Si el nodo que hemos arrastrado tenía hijos, todos ellos deben seguir siendo sus hijos.	
OBSERVACIONES	
Si al mover el nodo se elige como nuevo padre un programa diferente al original, en el listbox "Pertenece a" deberá haberse desmarcado el padre anterior y aparecer marcado el nuevo programa padre.	

PRUEBA	PA000111 : No permitir eliminación de nodo
NODO	N00394 : Nodo
CONDICION	
Debe haber al menos dos nodo (a parte del raíz) en el grafo. Uno de ellos tiene que tener pruebas de aceptación asociadas y el otro al menos un nodo hijo. Debemos estar logeados con el rol "RR. Funcional"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación 2) Hacemos clic derecho y seleccionamos "Eliminar Nodo" 3) En el grafo hacemos clic izquierdo sobre el nodo con hijos 4) Hacemos clic derecho y seleccionamos "Eliminar Nodo"	
RESULTADO ESPERADO	
PASO 2: Debe aparecer un mensaje de error informando sobre la imposibilidad de eliminar el nodo debido a que tiene pruebas de aceptación. PASO 4: Debe aparecer un mensaje de error informando sobre la imposibilidad de eliminar el nodo debido a que tiene hijos.	
OBSERVACIONES	

8.3.1 Nodo: Pruebas de Aceptación

PRUEBA	PA000130 : Añadir Propuesta de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo.	
PASOS	
1) Seleccionamos un nodo del grafo (diferente al nodo raíz) 2) En el grid de las pruebas de aceptación, hacemos clic derecho. 3) Seleccionamos "Proponer nueva" 4) En la ventana emergente, escribimos un nombre. 5) Pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 5: La nueva propuesta de PA debe aparecer en el grid de las PAs. En su fila, en la columna "Acción", tiene que aparecer un icono de un "+" . Si el nodo no estaba marcado como que afectaba a la incidencia se marcará.	
OBSERVACIONES	
Debe mostrarse el nroOrden de la propuesta en el grid	

PRUEBA	PA000107 : Añadir Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre uno de los nodos (distinto del raíz) 2) En el grid de la parte derecha de la ventana hacemos clic con el botón derecho del ratón y seleccionamos "Añadir Nueva PA" 3) Escribimos un nombre y pulsamos aceptar	
RESULTADO ESPERADO	
PASO 3: En el grid de las pruebas de aceptación debe de aparecer una PA con el nombre que hemos indicado.	
OBSERVACIONES	

PRUEBA	PA000145 : Consultar propuesta de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociadas 1 PA con propuesta, y una propuesta nueva.	
PASOS	
1) Seleccionamos un nodo del grafo. En el grid de las pruebas de aceptación: 2) Para la fila con una PA con una propuesta, hacemos doble clic en las columnas "comentarios Propuesta PA" o "Acción" 3) Para la fila con sólo una propuesta, hacemos doble clic en cualquier parte de la fila.	
RESULTADO ESPERADO	
PASOS 2 y 3: Debe abrirse una ventana con toda la información de la propuesta (arriba estará indicado que es una Propuesta)	
OBSERVACIONES	

PRUEBA	PA000116 : Consultar Prueba de Aceptación en Incidencia
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos entrar a la aplicación en el contexto de una incidencia Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que tener 2 pruebas de aceptación asociadas, una de ellas con una propuesta de modificación.	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación. En el grid de la parte derecha de la ventana: 2) Hacemos doble clic en la fila con sólo una pruebas de aceptación. 3) En la fila con una prueba de aceptación y una propuesta de modificación: hacemos clic en las columnas: nroOrden, codigo o nombre	
RESULTADO ESPERADO	
PASOS 2 y 3: Debe abrirse una ventana con toda la información relacionada con la PA. En dicha ventana debe dejar claro arriba que es una "Prueba actual"	
OBSERVACIONES	

PRUEBA	PA000158 : Consultar Prueba de Aceptación fuera de Incidencia
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Hay que entrar en la aplicación desde fuera del contexto de una Incidencia. Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que tener al menos una prueba de aceptación asociada.	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación. 2) En el grid de la parte derecha de la ventana hacemos doble clic sobre una de las pruebas de aceptación.	
RESULTADO ESPERADO	
PASO 2: Debe abrirse una ventana con toda la información relacionada con la PA	
OBSERVACIONES	

PRUEBA	PA000115 : Eliminar Pruebas de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que tener al menos una prueba de aceptación asociada. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación. 2) En el grid de la parte derecha de la ventana hacemos clic sobre una de las pruebas de aceptación. 3) Hacemos clic derecho y seleccionamos "Eliminar PA".	
RESULTADO ESPERADO	
PASO 3: La PA seleccionada debe desaparecer del grid.	
OBSERVACIONES	

PRUEBA	PA000118 : Filtrar pruebas de aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que tener al menos una prueba de aceptación asociada.	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación. 2) En el grid de la parte derecha de la ventana hacemos clic sobre una de las columnas de la primera fila e introducimos el patrón de búsqueda deseado.	
RESULTADO ESPERADO	
PASO 2: Deberán desaparecer todas las PAs del grid que no cumplan con lo especificado en el filtro.	
OBSERVACIONES	
Al seleccionar otra fila, la primera fila del filtro debe quedar marcada en naranja para indicar que hay un filtro activo.	

PRUEBA	PA000138 : Imposible añadir Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre uno de los nodos (distinto del raíz) 2) En el grid de la parte derecha de la ventana hacemos clic con el botón derecho del ratón y seleccionamos "Añadir Nueva PA" 3) Escribimos una descripción y pulsamos aceptar	
RESULTADO ESPERADO	
PASO 3: Debe aparecer un mensaje de error indicando que debes elegir un nombre para la PA.	
OBSERVACIONES	

PRUEBA	PA000144 : Imposible consultar una propuesta de eliminación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos conectarnos a la aplicación en el contexto de una incidencia. Debe haber al menos un nodo en el grafo con una prueba de aceptación asociada y una propuesta de eliminación para esta prueba.	
PASOS	
1) Hacemos clic en el nodo con la PA 2) En el grid de las PAs, hacemos doble clic sobre el "-" de la propuesta de eliminación.	
RESULTADO ESPERADO	
PASO 2: No debe de suceder nada.	
OBSERVACIONES	

PRUEBA	PA000131 : Modificar Propuesta de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociadas 1 PA con propuesta.	
PASOS	
1) Seleccionamos un nodo del grafo. En el grid de las pruebas de aceptación: 2) Hacemos clic en la fila con una PA con una propuesta 3) Clic derecho y elegimos "Proponer modificación" 4) Cambiamos el nombre y/o descripción y pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 4: Al consultar la propuesta modificada los cambios deberían verse reflejados.	
OBSERVACIONES	

PRUEBA	PA000117 : Modificar Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que tener al menos una prueba de aceptación asociada. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con pruebas de aceptación. 2) En el grid de la parte derecha de la ventana hacemos clic sobre una de las pruebas de aceptación. 3) Hacemos clic derecho y seleccionamos "Editar PA". 4) En la ventana emergente cambiamos el nombre de la PA 5) Pulsamos aceptar	
RESULTADO ESPERADO	
PASO 5: En el grid de las pruebas de aceptación, la que habíamos seleccionado debe haber cambiado su nombre al nuevo que introducimos.	
OBSERVACIONES	

PRUEBA	PA000133 : Proponer Eliminación de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociada alguna prueba de aceptación.	
PASOS	
1) Seleccionamos un nodo del grafo. En el grid de las pruebas de aceptación: 2) Hacemos clic en la fila con una PA sin propuesta. 3) Clic derecho y elegimos "Proponer eliminación"	
RESULTADO ESPERADO	
PASO 3: En el grid, en la columna "Acción" de la fila modificada debe aparecer un nuevo símbolo (" - ") indicando que hay una propuesta de eliminación para la prueba.	
OBSERVACIONES	

PRUEBA	PA000162 : Proponer modificación de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociada alguna prueba de aceptación.	
PASOS	
1) Seleccionamos un nodo del grafo. En el grid de las pruebas de aceptación: 2) Hacemos clic en la fila con una PA sin propuesta. 3) Clic derecho y elegimos "Proponer modificación" 4) Cambiamos el nombre y/o descripción y pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 4: En el grid, en la columna "Acción" de la fila modificada debe aparecer un nuevo símbolo indicando que hay una propuesta para esa prueba. Al consultar la propuesta modificada los cambios deberían verse reflejados.	
OBSERVACIONES	

PRUEBA	PA000132 : Quitar Propuesta de Prueba de Aceptación
NODO	N00396 : Pruebas de Aceptación
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociadas 1 PA con propuesta.	
PASOS	
1) Seleccionamos un nodo del grafo. En el grid de las pruebas de aceptación: 2) Hacemos clic en la fila con una PA con una propuesta 3) Clic derecho y elegimos "Quitar propuesta"	
RESULTADO ESPERADO	
PASO 3: El símbolo que aparecía en la columna acción de la fila seleccionada debe haber desaparecido.	
OBSERVACIONES	

PRUEBA	PA000150 : Añadir Prueba de Sistema a PA o IU
NODO	N00401 : Pruebas de Sistema
CONDICION	
Entramos en la aplicación como RR. Funcional o Analista y desde fuera del contexto de una incidencia. Debe haber al menos un nodo en el grafo. Con una PA y IU asociada.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Seleccionamos una prueba del grid de Pruebas de Aceptación 3) Hacemos clic derecho y pulsamos "Editar PA" 4) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema" 5) Hacemos clic derecho en el grid y pulsamos "Añadir PSistema" 6) Introducimos la descripción deseada. 7) Pulsamos aceptar. 8) Vamos a la pestaña IU 9) Hacemos doble clic sobre una captura. 10) Repetimos los pasos 4 a 6	
RESULTADO ESPERADO	
PASO 5: Debe añadirse una nueva Prueba de Sistema con un código autogenerado al grid. PASO 7: Al volver a consultar y ver la pestaña Pruebas de Sistema, debe seguir viéndose la Prueba de Sistema que añadimos en 5.	
OBSERVACIONES	

PRUEBA	PA000151 : Eliminar Prueba de Sistema a PA o IU
NODO	N00401 : Pruebas de Sistema
CONDICION	
Entramos en la aplicación como RR. Funcional o Analista y desde fuera del contexto de una incidencia. Debe haber al menos un nodo en el grafo. Con una PA y IU asociada. Tanto la PA como la IU tienen que tener una prueba de sistema.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Seleccionamos una prueba del grid de Pruebas de Aceptación 3) Hacemos clic derecho y pulsamos "Editar PA" 4) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema" 5) Seleccionamos una de las pruebas de sistema del grid. 6) Hacemos clic derecho y pulsamos "Eliminar PSistema" 7) Pulsamos aceptar 8) Vamos a la pestaña IU 9) Hacemos doble clic sobre una captura. 10) Repetimos los pasos 4 a 6	
RESULTADO ESPERADO	
PASO 7: La prueba de sistema debe desaparecer del grid.	
OBSERVACIONES	

PRUEBA	PA000153 : Filtrar Pruebas de Sistema
NODO	N00401 : Pruebas de Sistema
CONDICION	
Entramos en la aplicación desde fuera del contexto de una incidencia. Debe haber al menos un nodo en el grafo. Con una PA y IU asociada. Tanto la PA como la IU tienen que tener una prueba de sistema.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Seleccionamos una prueba del grid de Pruebas de Aceptación 3) Hacemos clic derecho y pulsamos "Editar PA" 4) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema" 5) En el grid, hacemos clic sobre una de las columnas de la primera fila e introducimos el patrón de búsqueda deseado. 6) Pulsamos aceptar 7) Vamos a la pestaña IU 8) Hacemos doble clic sobre una captura. 9) Repetimos los pasos 4 y 5	
RESULTADO ESPERADO	
PASO 5: Deberán desaparecer todas las pruebas de sistema del grid que no cumplan con lo especificado en el filtro.	
OBSERVACIONES	
Al seleccionar otra fila, la primera fila del filtro debe quedar marcada en naranja para indicar que hay un filtro activo.	

PRUEBA	PA000152 : Modificar Prueba de Sistema de PA o IU
NODO	N00401 : Pruebas de Sistema

CONDICION

Entramos en la aplicación como RR. Funcional o Analista y desde fuera del contexto de una incidencia.
Debe haber al menos un nodo en el grafo. Con una PA y IU asociada. Tanto la PA como la IU tienen que tener una prueba de sistema.

PASOS

1) Seleccionamos el nodo del grafo.
2) Seleccionamos una prueba del grid de Pruebas de Aceptación
3) Hacemos clic derecho y pulsamos "Editar PA"
4) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema"
5) Seleccionamos una de las pruebas de sistema del grid.
6) Cambios su código / descripción.
7) Pulsamos aceptar
8) Vamos a la pestaña IU
9) Hacemos doble clic sobre una captura.
10) Repetimos los pasos 4 a 6

RESULTADO ESPERADO

PASO 7: El cambio debe ser efectuado

OBSERVACIONES

PRUEBA	PA000157 : No permitir códigos duplicados para las PSistema
NODO	N00401 : Pruebas de Sistema

CONDICION

Entramos en la aplicación como RR. Funcional o Analista y desde fuera del contexto de una incidencia.
Debe haber al menos un nodo en el grafo. Con una PA y IU asociada.
Tiene que existir al menos una prueba de sistema.

PASOS

1) Seleccionamos el nodo del grafo.
2) Seleccionamos una prueba del grid de Pruebas de Aceptación
3) Hacemos clic derecho y pulsamos "Editar PA"
4) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema"
5) Hacemos clic derecho en el grid y pulsamos "Añadir PSistema"
6) Introducimos la descripción deseada.
7) Cambiamos el código por el de una prueba de sistema ya existente.
8) Pulsamos aceptar.
9) Vamos a la pestaña IU
10) Hacemos doble clic sobre una captura.
11) Repetimos los pasos 4 a 7

RESULTADO ESPERADO

PASO 8: Un error debe avisar de que el código está duplicado. La ventana no se puede cerrar hasta cambiar el código duplicado.

OBSERVACIONES

PRUEBA	PA000163 : No permitir ver Pruebas de Sistema en el contexto de una incidencia
NODO	N00401 : Pruebas de Sistema
CONDICION	
Entramos en la aplicación desde el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Con una PA y IU asociada.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Hacemos doble clic sobre alguna de sus pruebas de aceptación. 3) En la ventana emergente, vamos a la pestaña "Pruebas de Sistema" 4) Pulsamos aceptar. 5) Vamos a la pestaña IU 6) Hacemos doble clic sobre una captura. 7) Repetimos el paso 3	
RESULTADO ESPERADO	
PASO 3: No se debe permitir clicar en la pestaña "Pruebas de Sistema"	
OBSERVACIONES	

PRUEBA	PA000139 : Añadir comentarios a PA o IU
NODO	N00403 : Comentarios
CONDICION	
Debe haber al menos un nodo en el grafo. Con una PA e IU asociada.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Seleccionamos una prueba del grid de Pruebas de Aceptación 3) Hacemos doble clic sobre la columna "Comentarios" 4) En la ventana emergente, hacemos clic derecho en el grid y pulsamos "Añadir comentario" 5) Introducimos la descripción deseada. 6) Pulsamos aceptar. 7) Vamos a la pestaña IU 8) Repetimos los pasos 3, 4 y 5	
RESULTADO ESPERADO	
PASO 4: Debe aparecer el comentario en el grid con la fecha y el nombre del autor. PASO 6: El contador de comentarios del grid de las PA debe haberse incrementado.	
OBSERVACIONES	

PRUEBA	PA000171 : Marcar comentarios como leídos / no leídos

NODO	N00403 : Comentarios
CONDICION	
Hay que loguearse en la aplicación como analista. Debe haber al menos un nodo en el grafo. Con una PA e IU asociada. Cada uno de ellos con algunos comentarios.	
PASOS	
1) Seleccionamos el nodo del grafo. 2) Seleccionamos una prueba del grid de Pruebas de Aceptación 3) Hacemos doble clic sobre la columna "Comentarios" . 4) En la ventana emergente, hacemos doble clic sobre alguno de los comentarios. 5) Pulsamos aceptar. 6) Vamos a la pestaña IU 7) Repetimos los pasos 3 y 4.	
RESULTADO ESPERADO	
PASO 4: Los comentarios deben marcarse como leídos (check de la columna leido) si estaban como no leídos y viceversa. PASO 5: En el grid, el contador de comentarios debe haber cambiado, mostrando correctamente cuántos comentarios hay leídos y sin leer del total.	
OBSERVACIONES	
En el contador de comentarios del grid, se muestran los comentarios de las Prueba de Aceptación actual si estamos fuera del contexto de una incidencia; y se muestra el contador de comentarios leídos de la propuestas de prueba cuando estamos en el contexto de una incidencia.	

8.3.2 Nodo: Interfaz Usuario

PRUEBA	PA000119 : Añadir Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz. Dicho nodo debe ser del tipo Interfaz de Usuario. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre uno de los nodos de tipo Interfaz de Usuario. 2) En la parte derecha, vamos a la pestaña "IU" 3) En el grid, hacemos clic con el botón derecho del ratón y seleccionamos "Añadir captura" 4) En la ventana emergente, pulsamos examinar y seleccionamos una imagen. 5) Pulsamos "Añadir"	
RESULTADO ESPERADO	
PASO 5: En el grid de las interfaces de usuario debe de aparecer una captura con el nombre que hemos indicado.	
OBSERVACIONES	

PRUEBA	PA000127 : Añadir Propuesta de Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo.	
PASOS	
1) Seleccionamos un nodo del grafo (diferente al nodo raíz) 2) Vamos a la pestaña "IU" 3) En el grid, hacemos clic derecho y seleccionamos "Añadir nueva IU" 4) En la ventana emergente, pulsamos Examinar y elegimos una imagen. 5) Pulsamos aceptar.	
RESULTADO ESPERADO	
PASO 5: La nueva propuestas de IU debe aparecer en el grid. La imagen debe aparecer en la columna "IU Propuesta". Si el nodo no estaba marcado como que afectaba a la incidencia se marcará.	
OBSERVACIONES	

PRUEBA	PA000122 : Consultar Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que ser del tipo fragmento de interfaz y tener al menos una captura asociada.	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con capturas. 2) Seleccionamos la pestaña "IU" 3) Hacemos doble clic sobre la captura que queremos consultar.	
RESULTADO ESPERADO	
PASO 3: Debe aparecer una ventana emergente con la captura a pantalla completa.	
OBSERVACIONES	

PRUEBA	PA000120 : Eliminar Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que ser del tipo fragmento de interfaz y tener al menos una captura asociada. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con capturas. 2) Seleccionamos la pestaña "IU" 3) Hacemos clic sobre la captura que deseamos eliminar. 4) Hacemos clic derecho y seleccionamos "Eliminar IU"	
RESULTADO ESPERADO	
PASO 4: Debe desaparecer del grid la captura seleccionada.	
OBSERVACIONES	

PRUEBA	PA000121 : Modificar Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debe existir al menos un nodo en el grafo además del nodo raíz, dicho nodo tiene que ser del tipo fragmento de interfaz y tener al menos una captura asociada. Hay que estar logeado con el rol "RR. Funciona" o "Analista"	
PASOS	
1) En el grafo hacemos clic izquierdo sobre el nodo con capturas. 2) Seleccionamos la pestaña "IU" 3) Hacemos doble clic sobre la captura que deseamos modificar. 4) En la ventana emergente, cambiamos el nombre que aparece 5) Cerramos la ventana.	
RESULTADO ESPERADO	
PASO 5: En el grid debe haber cambiado el nombre de la captura que seleccionamos.	
OBSERVACIONES	

PRUEBA	PA000137 : Proponer eliminación de Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociada alguna IU.	
PASOS	
1) Seleccionamos un nodo del grafo. 2) Vamos a la pestaña "IU" 3) Hacemos clic en una fila con una IU actual. 4) Clic derecho y elegimos "Eliminar IU Actual"	
RESULTADO ESPERADO	
PASO 4: En el grid, en la columna "IU Propuesta" de la fila modificada debe aparecer una imagen (" X ") indicando que hay una propuesta de eliminación para la captura.	
OBSERVACIONES	

PRUEBA	PA000129 : Proponer modificación de Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociada al menos una IU.	
PASOS	
1) Seleccionamos un nodo del grafo. 2) Vamos a la pestaña "IU" 3) Hacemos clic en una fila con una IU actual. 4) Clic derecho y elegimos "Modificar IU Actual" 5) Pulsamos examinar y elegimos una nueva captura. 6) Pulsamos añadir.	
RESULTADO ESPERADO	
PASO 6: En la fila de la IU que seleccionamos, debe haber aparecido la nueva captura bajo la columna IU Propuesta.	
OBSERVACIONES	

PRUEBA	PA000128 : Quitar Propuesta de Interfaz de Usuario
NODO	N00397 : IUs
CONDICION	
Debemos estar logueados en la aplicación como Analista o RR. Funcional. Debemos estar en el contexto de una incidencia. Debe haber al menos un nodo en el grafo. Dicho nodo tiene que tener asociadas 1 IU con propuesta.	
PASOS	
1) Seleccionamos un nodo del grafo. 2) Vamos a la pestaña "IU" 3) Hacemos clic en la fila con una IU con una propuesta 4) Clic derecho y elegimos "Quitar IU propuesta"	
RESULTADO ESPERADO	
PASO 4: La captura que aparecía en la fila, en la columna "IU Propuesta" debe haber desaparecido.	
OBSERVACIONES	

8.3.3 Nodo: Zona de información del nodo

PRUEBA	PA000103 : Modificación de información de nodo
NODO	N00404 : Ficha del nodo
CONDICION	
Debe existir un programa en la base de datos. Debe haber un nodo hijo del programa, que se encuentre activo. Hay que estar conectado como RR. Funcional.	
PASOS	
1) Elegimos el programa y entramos en la aplicación. 2) Expandimos el nodo raíz y pulsamos sobre el nodo con información. 3) En la parte derecha de la ventana, cambiamos el nombre del nodo 4) En la parte superior izquierda, pulsamos en "Refrescar".	
RESULTADO ESPERADO	
PASO 4: En el grafo, el nodo en cuestión debe haber cambiado su nombre.	
OBSERVACIONES	

PRUEBA	PA000134 : Modificar estado del nodo
NODO	N00404 : Ficha del nodo
CONDICION	
Debe existir un program en la base de datos. Debe haber un nodo hijo del programa, que se encuentre activo. Debe estar activo el filtro de "Mostrar sólo Nodos Activos"	
PASOS	
1) Elegimos el programa y entramos en la aplicación. 2) Expandimos el nodo raíz y pulsamos sobre el nodo con información. 3) En la parte derecha de la ventana, cambiamos el nombre del nodo 4) En la parte superior izquierda, pulsamos en "Refrescar".	
RESULTADO ESPERADO	
PASO 4: En el grafo, el nodo en cuestión debe haber desaparecido.	
OBSERVACIONES	

PRUEBA	PA000102 : Ver información de nodo
NODO	N00404 : Ficha del nodo
CONDICION	
Debe existir un programa en la base de datos. Dicho programa tiene que tener al menos un nodo relacionado, que se encuentre activo y con una descripción.	
PASOS	
1) Elegimos el programa y entramos en la aplicación. 2) Expandimos el nodo raíz y pulsamos sobre uno de sus nodos	
RESULTADO ESPERADO	
PASO 2: En la parte derecha de la ventana debe aparecer el nombre del nodo, su descripción asociada, su tipo (Requisito Funciona, Requisitos No Funcional o Interfaz de Usuario), el estado en el que se encuentra (activo o no activo) y debe estar indicado a qué programas pertenece.	
OBSERVACIONES	