Pontificia Universidad Católica de Valparaíso

Facultad de Ingeniería

Escuela de Ingeniería Informática

PROCESO DE PLANIFICACIÓN DE PRUEBAS DE SOFTWARE APOYADO EN UNA HERRAMIENTA BASADA EN CASOS DE USO

Autor:

Francisco Javier Suárez Gallardo

Informe final del Proyecto para optar al Título profesional de

Ingeniero Civil en Informática

Profesor guía:

Broderick Crawford Labrín

Julio 2007

A quienes me han apoyado incondicionalmente durante toda mi vida, mi familia,
especialmente mis padres, mi madrina y abuelos.

Francisco Javier Suárez Gallardo

Agradecimientos.

Agradezco a Dios Padre, por estar siempre iluminando mi camino, por enseñarme a aprender de cada momento de la vida, y por apoyarme en cada desafío.

A mi familia, por apoyarme incondicionalmente en todo momento y ser la gran motivación para salir siempre adelante.

A la Universidad, por entregarme las vivencias, valores y conocimientos adquiridos durante todo este proceso, y a los profesores guías y correferentes que contribuyeron directamente en el desarrollo de este proyecto.

Francisco Javier Suárez Gallardo

Lista de Abreviaturas o Siglas.

SUT : System Under Test.

UML: Unified Modeling Language.

RUP: Rational Unified Process.

PLUTO: Product Lines Use case Test Optimization.

NDT: Navigational Development Technique.

RETNA: Requirements to Testing in a Natural Way.

RBC: Requirements by Contract.

TOTEM: A UML-Based Approach to System Testing.

TDEUML: Derivation of Domain Test Scenarios from Activity Diagrams.

ATCGDM: Automated Test Case Generation from Dynamic Models.

TCUC: Test Cases from Use Cases.

RBC: Requirement Base Testing.

UCPA: Testing from Use Cases Using Path Analysis Technique.

UCDTC: Use Case Derived Test Cases.

AT: Software Requirements and Acceptance Testing.

Resumen.

La generación sencilla de los casos de prueba de una aplicación es una aspiración antigua de los desarrolladores de software. A lo largo de la historia, han existido diversas tentativas de lograr esto incluso promoviendo entornos y herramientas que ayudan en la generación de pruebas a partir de distintos tipos de información descriptiva de la aplicación. Sin embargo, es habitual que tanto las descripciones necesarias para aplicar estas técnicas como los entornos propicios para ello no sean habituales en los proyectos

de desarrollo. En este trabajo se verá cómo utilizando información de una especificación de requisitos desarrollada con notación UML, y con la ayuda de una herramienta de apoyo, se puede generar un conjunto apropiado de pruebas para la correspondiente aplicación, los casos de uso se utilizan con mucha frecuencia para definir la funcionalidad de un sistema software en las etapas tempranas de su desarrollo. Este trabajo presenta una propuesta para obtener de manera sistemática objetivos de prueba a partir de los casos de uso del propio sistema, para que a partir de ello, a través de la medición de la Complejidad Ciclomática del modelo visto desde el punto de vista de un grafo, contar con la información acerca del límite superior de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia del caso de uso al menos una vez.

Abstract.

The simple generation of test cases of an application is an old aspiration of software developers. Throughout history, there have been various attempts to achieve this even promoting environments and tools that assist in generating evidence from different types of descriptive information on the application. However, it is customary for both descriptions necessary to implement these techniques as enabling environments for this are not common in development projects. This paper presents how to use information from a specification of requirements developed with UML notation, and with the help of a support tool, to generate an appropriate set of tests for the corresponding application, the use cases are used very frequently to define the functionality of a software system in the early stages of their development. In addition presents a proposal to generate test objectives based on the case of use of the system itself, so that on that basis, through the measurement of the complexity of the model seen from the point of view of a graph, to obtain information about the upper limit of tests to be carried out to ensure that runs each sentence in the use case at least once.

Capítulo 1
Introducción.

1.1 Introducción.

El aumento de la complejidad de los sistemas software pone de manifiesto la necesidad de garantizar la calidad de los mismos. Toda la capacidad de detección de defectos en la fase de pruebas se basa en la consideración de cualquier tipo de discrepancia entre la salida obtenida y la salida esperada (según lo indicado en las especificaciones) como síntoma de un problema en el software [3]. En consecuencia, todas las técnicas de diseño de casos de prueba existentes se basan en tomar las especificaciones como referencia de comportamiento de la aplicación. Nada impide que el diseño y la planificación de las pruebas se inicien en paralelo al trabajo de especificación. No sólo es algo posible sino que se trata de una práctica recomendada para lograr mayor calidad, productividad y disminución de riesgos.

Es así como, siguiendo la línea de pruebas tempranas del software existen metodologías de diseño orientados a que las pruebas sean la guía principal del desarrollo, como es el caso del test-driven development (TDD), práctica de programación que busca escribir las pruebas antes de implementar su código, buscando de esta forma lograr una rápida retroalimentación [2], ya que considera la refactorización como una etapa importante posterior a la codificación de las mismas. Este concepto será tomado como base para aplicarlo en el desarrollo de este proyecto, que tomará como punto de partida la captura de requisitos especificada en los casos de uso, desde el punto de vista de pruebas de sistema, debido a lo que contienen dichos diagramas, que para efectos de este proyecto, serán los relacionados con la funcionalidad.

Es posible desarrollar varios tipos de pruebas del sistema, como pruebas funcionales, de usabilidad, de seguridad y de navegación. Este trabajo está centrado en las pruebas funcionales desde el punto de vista de los actores del sistema y, especialmente, de actores

humanos que interactúan a través de interfaces gráficas. Así, un caso de prueba sustituye a un actor y simula su comportamiento para verificar que el sistema hace lo que se espera de él. Por esta razón, el principal artefacto para obtener pruebas del sistema son sus requisitos funcionales, ya que ellos describen todo el comportamiento esperado que debe ser probado.

La propuesta de este proyecto especifica el comportamiento del sistema mediante casos de uso. Los casos de uso ofrecen una visión general del sistema y son fáciles de estudiar y validar por parte de usuarios no expertos en ingeniería del software. Es también muy común en la industria del software identificar casos de prueba a partir de los casos de uso. En fases tempranas del desarrollo, cuando los requisitos están siendo descubiertos, definidos y negociados, es mucho más sencillo modificar los casos de uso definidos como texto libre o texto estructurado que requisitos formales.

Los objetivos de prueba obtenidos a partir de los casos de uso definen básicamente qué debe ser probado para asegurar la correcta y completa implementación de los mismos. El diseño de buenos casos de uso es una tarea vital para el proceso de pruebas, además para el proceso de desarrollo. Sin embargo, el perfil de pruebas de UML (Unified Modeling Language) no define ninguna notación para representar objetivos de prueba. Se ha resuelto esta carencia mediante el uso de diagramas de actividades y secuencias de actividades.

La generación automática de objetivos de prueba presentada más adelante continúa esta línea de investigación y es también un paso importante para obtener un proceso completo y sistemático para la generación de casos de prueba.

Otras propuestas se centran en los diagramas de estados que describen el comportamiento de cada clase [1]. De hecho, cualquiera de estas representaciones permite describir criterios de cobertura para la interfaz gráfica de usuario (GUI) o la estructura de la aplicación así como métodos de diseño sistemático de las pruebas o de verificación a partir de dichos modelos de especificación.

Uno de los problemas de estas aproximaciones es la peculiaridad de los modelos empleados y que los analistas deben generar cuando realizan el trabajo de especificación de un sistema. Al tratarse de notaciones no convencionales que requieren un considerable esfuerzo, la aplicación de estas técnicas resulta poco viable en la mayoría de los entornos. Frente a esta situación, esta propuesta se desea aproximar a enfoques metodológicos con un mayor grado de implantación en los proyectos de desarrollo profesional y comercial. Así, la implantación de UML como notación de desarrollo orientado a objetos ha permitido la difusión e implantación real de técnicas de especificación de software como los casos de uso. Más adelante, se describe el formato exacto de especificación basada en UML que servirá de base para el método sistemático de diseño de pruebas de software. También se podrá apreciar cómo un apropiado trabajo de especificación tiene como valor adicional la facilidad para el establecimiento de pruebas directamente relacionadas con los requisitos del software.

Este proyecto presenta un proceso sistemático para generar el conjunto de caminos de ejecución de los casos de uso del sistema llamados objetivos de prueba, gracias a ello se podrá visualizar además el modelo de actividades del sistema, y tratar su comportamiento como un grafo, para así poder posteriormente obtener una métrica que indica el número exacto de casos de prueba necesarios para probar cada punto de decisión del caso de uso del sistema, dicha medida es conocida también como complejidad ciclomática [26] [20]. Se propone además, un proceso sistemático para generar objetivos de prueba a partir de los casos de uso.

1.2 Objetivo General.

El objetivo global de este proyecto es la propuesta de un proceso de planificación de pruebas apoyado en una herramienta que asista y soporte dicha fase, en base al cálculo de una métrica conocida como complejidad ciclomática en la etapa de toma de requerimientos ya modelados del mismo, a nivel de Casos de Uso.

1.3 Objetivos Específicos.

- Analizar los conceptos fundamentales de pruebas de software en el contexto del aseguramiento de la calidad.

- Estudiar la posibilidad de establecer un proceso de pruebas tempranas de software a partir de la toma de requerimientos.

- Diferenciar etapas en el proceso de pruebas de software, enmarcando las que finalmente cubrirá el trabajo.

- Análisis y evaluación de estudios actuales que tengan como fin elaborar propuestas para la generación de pruebas a partir de los requisitos funcionales.

- Definir la información que deben tener los casos de uso, cómo deben estar redactados para poder generar pruebas a partir de ellos y desarrollar un prototipo de herramienta software.

- Propuesta concreta de metodología a aplicar en la herramienta.

- Modelar los requerimientos del sistema en una herramienta que sirva de soporte a la fase de planificación de pruebas.

- Dar una nueva instancia de refinamiento de las especificaciones del usuario, ya que a través de este proceso se pretende proponer además un método para la mejora de requerimientos.

- Implementación de la herramienta.

Capítulo 2

Investigación Previa.

2.1 Pruebas de Software.

Las Pruebas se integran dentro de las diferentes fases del ciclo de vida del software. Así se ejecuta un programa y mediante técnicas experimentales se trata de descubrir sus errores. La calidad de un sistema software es algo subjetivo que depende del contexto y del objeto que se pretenda conseguir [21]. Para determinar dicho nivel de calidad se deben efectuar unas medidas o pruebas que permitan comprobar el grado de cumplimiento respecto de las especificaciones iniciales del sistema.

Hay muchos planteamientos a la hora de abordar pruebas de software, pero para verificar productos complejos de forma efectiva se requiere de un proceso de investigación más que seguir un procedimiento al pie de la letra. Una definición de "testing" es: *proceso de evaluación de un producto desde un punto de vista crítico* [3], donde el "tester" (persona que realiza la prueba) somete el producto a una serie de acciones inquisitivas, y el producto responde con su comportamiento como reacción. Por supuesto, nunca se debe testear el software en un entorno de producción. Es necesario probar los nuevos programas en un entorno de pruebas separado físicamente del de producción. Para crear un entorno de pruebas en una máquina independiente de la de producción es necesario crear las mismas condiciones que en la máquina de producción. Existen a tal efecto varias herramientas vendidas por los mismos fabricantes de hardware. Esas utilidades reproducen automáticamente las bases de datos para simular un entorno de producción.

Es posible clasificar las pruebas en dos grandes grupos, en uno están las pruebas de caja negra y caja blanca y por otro están las pruebas manuales y automáticas.

Las pruebas de caja blanca tienen como objetivo recorrer la estructura del código comprobando la ejecución de todos los posibles caminos de ejecución. Las pruebas de caja negra, en cambio, se centran en los requisitos o resultados del sistema software. Las

pruebas automáticas son aquellas realizadas por un programa o herramienta que prueba el sistema sin necesidad de la interacción de una persona. Las pruebas manuales, en cambio, son aquellas pruebas realizadas por una o más personas que interactúan directamente con el sistema. Estas personas verifican si los resultados obtenidos son válidos o no.

2.1.1 La Importancia de la detección Oportuna.

En la cadena de valor del desarrollo de un software específico, el proceso de prueba es clave a la hora de detectar errores o fallas. Conceptos como estabilidad, escalabilidad, eficiencia y seguridad se relacionan a la calidad de un producto bien desarrollado. Las aplicaciones de software han crecido en complejidad y tamaño, y por consiguiente también en costos. Hoy en día es crucial verificar y evaluar la calidad de lo construido de modo de minimizar el costo de su reparación. Mientras antes se detecte una falla, más barato es su corrección.

El proceso de prueba es un proceso técnico especializado de investigación que requiere de profesionales altamente capacitados en lenguajes de desarrollo, métodos y técnicas de testeo y herramientas especializadas. El conocimiento que debe manejar un ingeniero de prueba es muchas veces superior al del desarrollador de software.

2.1.2 Proceso de prueba del software.

Tabla 1.2.1.2 Descripción de los tipos de prueba del software.

Tipos de pruebas	Fases de realización	Descripción
Unitarias	Durante la construcción del sistema	Prueban el diseño y el comportamiento de cada uno de los componentes del sistema una vez construidos.
Integración	Durante la construcción del sistema	Comprueban la correcta unión de los componentes entre sí a través de sus interfaces, y si cumplen con la funcionalidad establecida.

Sistema	Después de la construcción del sistema	Prueban a fondo el sistema, comprobando su funcionalidad e integridad globalmente, en un entorno lo más parecido posible al entorno final de producción.
Implantación	Durante la implantación en el entorno de producción	Comprueba el correcto funcionamiento del sistema dentro del entorno real de producción.
Aceptación	Después de la implantación en el entorno de producción	Verifican que el sistema cumple con todos los requisitos indicados y permite que los usuarios del sistema den el visto bueno definitivo.
Regresión	Después de realizar modificaciones al sistema	El objetivo es comprobar que los cambios sobre un componente del sistema, no generan errores adicionales en otros componentes no modificados.

Para que el proceso de pruebas del sistema sea eficaz debe estar integrado dentro del propio proceso de desarrollo. Como cualquier otra fase de dicho proceso, el proceso de prueba debe realizarse de manera sistemática, minimizando el factor experiencia o intuición. Esto se puede conseguir a través de metodologías que guíen el proceso de desarrollo de pruebas de sistema.

Es necesaria una fase de prueba del sistema. Las pruebas de sistema tienen como objetivo verificar que el sistema satisface sus especificaciones probando que el comportamiento del sistema es el recogido en sus requisitos. Para que el proceso de pruebas del sistema sea eficaz debe estar integrado dentro del propio proceso de desarrollo. Como cualquier otra fase de dicho proceso, el proceso de prueba debe realizarse de manera sistemática, minimizando el factor experiencia o intuición. Esto se puede conseguir a través de metodologías que guíen el proceso de desarrollo de pruebas de sistema.

Figura 1.2.1.2 Ciclo completo del proceso de pruebas.

2.1.3 Enfoques de Diseño de Pruebas.

Existen tres enfoques principales para el diseño de casos:

1.- El enfoque estructural o de caja blanca. Se centra en la estructura interna del programa (analiza los caminos de ejecución).

2.- El enfoque funcional o de caja negra. Se centra en las funciones, entradas y salidas.

3.- El enfoque aleatorio consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba

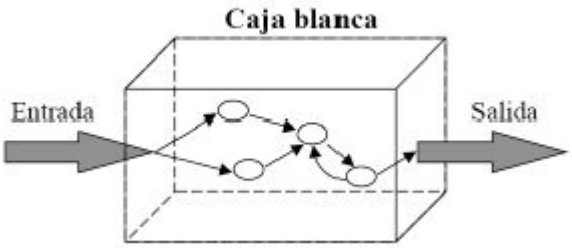


Figura 2.2.1.3 Enfoque estructural de diseño de pruebas.

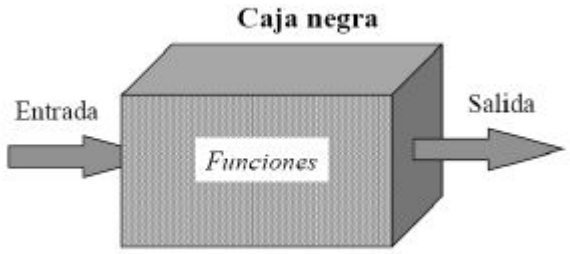


Figura 3.2.1.3 Enfoque funcional de diseño de pruebas.

2.2 Casos de prueba.

Los casos de prueba o Test Case son un conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio. Se pueden realizar muchos casos de prueba para determinar que un requisito es completamente satisfactorio. Con el propósito de comprobar que todos los requisitos de una aplicación son revisados, debe haber al menos un caso de prueba para cada requisito a menos que un requisito tenga requisitos secundarios.

Un caso de prueba chequea una situación particular o condición dentro del sistema, y está contenido de los siguientes elementos:

Identificación caso de prueba.
Precondiciones.
Dato de entrada.
Procedimiento de prueba.
Resultados esperados.

En ese caso, cada requisito secundario deberá tener por lo menos un caso de prueba. Algunas metodologías como RUP (Proceso Unificado de Rational) recomiendan el crear por lo menos dos casos de prueba para cada requisito. Uno de ellos debe realizar la prueba positiva de los requisitos y el otro debe realizar la prueba negativa.

Si la aplicación es creada sin requisitos formales, entonces los casos de prueba se escriben basados en la operación normal de programas de una clase similar. Lo que caracteriza un escrito formal de caso de prueba es que hay una entrada conocida y una salida esperada, los cuales son formulados antes de que se ejecute la prueba. La entrada conocida debe probar una precondición y la salida esperada debe probar una poscondición. Bajo circunstancias especiales, podría existir la necesidad de ejecutar la prueba, producir resultados, y luego un equipo de expertos evaluaría si los resultados se pueden considerar como "correctos". Esto sucede a menudo en la determinación del número del rendimiento de productos nuevos. La primera prueba se toma como línea base para los ciclos siguientes de pruebas/lanzamiento del producto. Los casos de prueba escritos, incluyen una descripción de la funcionalidad que se probará, la cuál es tomada ya sea de los requisitos o de los casos de uso, y la preparación requerida para asegurarse de que la prueba pueda ser dirigida.

2.3 Casos de Uso.

Es una técnica para la captura de requisitos potenciales de un nuevo sistema o una actualización software. Cada caso de uso proporciona uno o más escenarios que indican cómo debería interactuar el sistema con el usuario o con otro sistema para conseguir un objetivo específico. Normalmente, en los casos de uso se evita el empleo de jergas técnicas, prefiriendo en su lugar un lenguaje más cercano al usuario final. Los diagramas de casos de uso sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/o otros sistemas. Los casos de uso son una manera práctica de especificación del comportamiento de un sistema desde la perspectiva del usuario, éstos pueden ser una poderosa fuente de generación de casos de prueba, por lo tanto se examinará cómo derivar en éstos pruebas del sistema y adelantar en lo posible así esta fase que se deja comúnmente para el final del Ciclo de desarrollo de Software.

Entre las entidades y definiciones básicas que hay en los casos de uso están los actores, los cuales se pueden definir como un conjunto de personas o sistemas que interactúan con el sistema que estamos construyendo de la misma forma. Tiene la propiedad de ser externo al sistema. Hay que tener en cuenta que un usuario puede acceder al sistema

como distintos actores. Tienen una representación gráfica de un "hombre de palo" (stick man). Como se ha dicho antes, un caso de uso es una secuencia de interacciones entre un sistema y alguien o algo que usa alguno de sus servicios. Ese alguien o algo es el actor. Tienen una representación gráfica de óvalos.

Como características principales, los casos de uso:

Están expresados desde el punto de vista del actor.
Se documentan con texto informal describiendo la interacción.
Están acotados al uso de una funcionalidad claramente definida del sistema.

Los casos de uso son una manera práctica de especificación del comportamiento de un sistema desde la perspectiva del usuario, éstos pueden ser una poderosa fuente de generación de casos de prueba, por lo tanto se examinará cómo derivar en éstos pruebas del sistema y adelantar en lo posible así esta fase que se deja comúnmente para el final del Ciclo de desarrollo de Software.

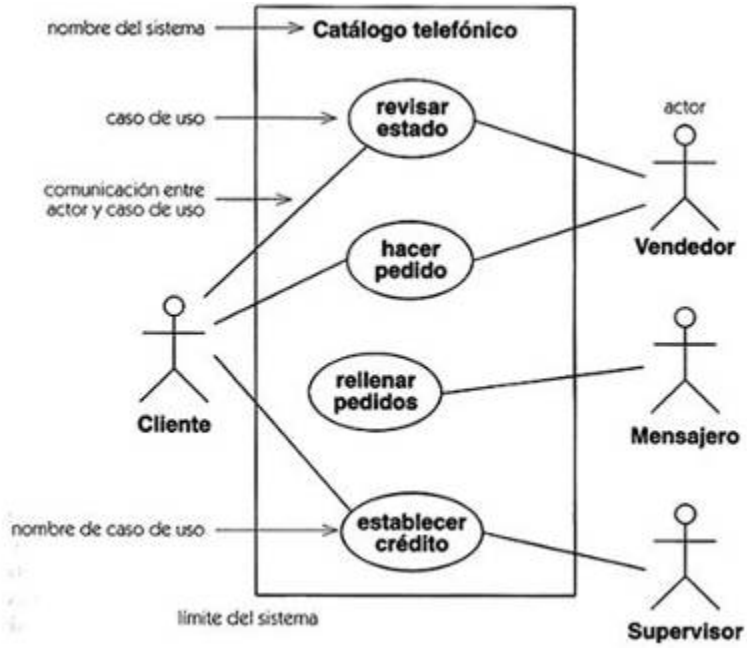


Figura 4.2.3 Ejemplo de diagrama de caso de uso.

La idea básica es simple, ya que se espera que un sistema desarrolle una función de forma segura, por lo tanto el caso de uso verifica que el sistema desarrolla su función según lo

esperado, de acuerdo a lo descrito en él. Primeramente eso sí, el caso de uso debe estar especificado de tal manera que cumpla con los requisitos para que se pueda probar. Para esto debe ser: Legible, Entendible, Correcto, Completo, Actualizado, Estable, Específico y Documentado.

2.4 Mejora de la calidad de los requisitos mediante la generación de pruebas.

Una de las piezas básicas para desarrollar sistemas software en el tiempo planificado y con la calidad adecuada es contar con buenos requisitos. Estos deben ser claros, precisos, no ambiguos y completos. Sin embargo, conseguir buenos requisitos es difícil. Para verificar y mejorar la calidad de los requisitos existen varias técnicas como revisiones formales, matrices de rastreo, ontologías, etc. Otra técnica poco utilizada es la generación de pruebas a partir de los requisitos. Muchos problemas en la construcción de sistemas software tienen su origen en los requisitos. Requisitos incompletos e inadecuados son una de las causas principales de fracaso en los proyectos software. Como hemos visto, adelantar el diseño y generación de pruebas del sistema tiene como ventaja añadida realizar una verificación adicional sobre los requisitos para corregir errores y mejorar su calidad. La identificación de caminos de ejecución de los casos de uso debe involucrar a todos los participantes en el sistema, incluyendo a los clientes o futuros usuarios y a los ingenieros de prueba y permitir el descubrimiento de nuevas características y elementos. La generación de pruebas en etapas tempranas del desarrollo permite corregir los siguientes defectos:

- Omisión de escenarios: Un caso de uso contiene un conjunto de posibles interacciones entre los actores implicados y el sistema. Cada uno de dichos conjuntos recibe el nombre de camino de ejecución o escenario.

- Resultados sin escenarios o escenarios sin resultado: Para elaborar las pruebas, no solo se necesita un camino de ejecución, sino el resultado de dicho camino, lo cual permite evaluar la ejecución satisfactoria o insatisfactoria del camino y la correcta implantación de dicho camino en el sistema. Al explorar todas las posibles combinaciones es posible

encontrar caminos de ejecución que no conduzcan a ningún resultado documentado y resultados esperados a los que no conduzca ningún camino de ejecución.

- Nuevos requisitos funcionales: Es posible encontrar caminos de ejecución y datos que resultados esperados que se expresen mejor mediante la definición de nuevos requisitos funcionales.

- Ambigüedad: Actualmente se emplea con mucha frecuencia tablas y plantillas en lenguaje natural para definir casos de uso. Uno de los principales problemas de los casos de uso descritos mediante lenguaje natural es la ambigüedad, ya que las mismas palabras pueden tener significados diferentes según quien las lea. Puesto que generar un conjunto de casos de prueba obliga a los ingenieros de prueba a interpretar y simular el funcionamiento del mismo, ayuda a detectar aquellos términos ambiguos o que puedan tener distintos significados.

2.4.1 Generación de pruebas del sistema.

La verificación de los elementos de un sistema software se realiza mediante pruebas unitarias y de integración (figura 4). Estas pruebas se aplican sobre los distintos componentes del sistema y garantizan su correcto funcionamiento, primero de manera aislada y, después, trabajando de forma conjunta.

Sin embargo estas pruebas no pueden garantizar el correcto funcionamiento del sistema [3]. Es necesaria una fase posterior de prueba del sistema. Las pruebas del sistema tienen como objetivo verificar que el sistema satisface sus especificaciones [3] probando que el comportamiento del sistema es el recogido en sus requisitos.

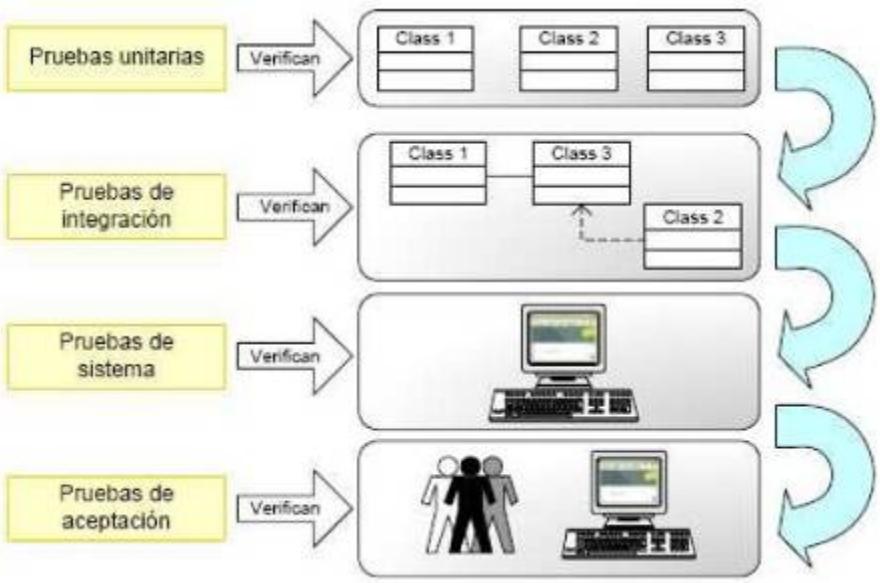


Figura 5.2.4.1 Fases de prueba de un sistema software.

Las pruebas del sistema pueden generarse a partir de los requisitos del sistema [1]. Esto permite adelantar la definición de las pruebas a etapas donde ya estén disponibles los requisitos. La generación de pruebas de sistema en etapas tempranas del desarrollo permite mejorar la calidad de los requisitos detectando errores, omisiones, inconsistencias y sobre-especificaciones en los requisitos funcionales cuando aún es fácil y económico corregirlos [4]. Para capturar los requisitos funcionales se utiliza mayoritariamente casos de uso.

Los casos de uso proporcionan un medio de expresar la interacción entre un sistema y su entorno. Esto permite estructurar los requisitos de acuerdo con los objetivos de los usuarios [1]. Esto permite estructurar los requisitos de acuerdo con los objetivos de los usuarios.

Existe un amplio número de propuestas para la generación de casos de prueba a partir de requisitos funcionales, en el cual se expuso una comparación de las más significativas durante la fase de avance de esa asignatura.

Varias se basan en descripciones en lenguaje natural de los requisitos funcionales, otras se apoyan en diagramas UML [17] y otras detallan su propio proceso para desarrollar y detallar los requisitos funcionales. En general, las pruebas de sistema generadas consisten

en una explosión combinatoria para identificar todos los caminos de ejecución posibles y generar una prueba por cada camino. Dichas pruebas se completan con datos de prueba, el resultado esperado, los actores participantes y las precondiciones y poscondiciones de cada requisito funcional [11]. A lo largo de este trabajo se utilizarán casos de uso para definir requisitos funcionales.

A continuación se expone cómo la aplicación de estas propuestas puede servir, además de para obtener un conjunto de pruebas, para mejorar los requisitos funcionales de un sistema en etapas tempranas del desarrollo.

Se hace necesario entonces, contar con una herramienta que permita la representación de las interacciones entre actores y el sistema para determinar los valores de prueba implicados y los resultados esperados. El problema que se pretende abordar en este proyecto es asegurar la satisfacción de requisitos tomando en cuenta el hecho de que una vez terminada la codificación no hay tiempo para probar. Para ello es necesario buscar un proceso para guiar y automatizar la obtención de pruebas a partir de los requisitos, adelantando el proceso, se pueden tener las pruebas muy avanzadas o ya listas cuando termine la codificación.

Así, para la generación de pruebas el punto de origen de este proyecto son los casos de uso. Las pruebas sólo pueden encontrar los errores que buscan, por lo tanto, un programa que pasa todos sus casos de prueba no es un programa sin errores, por esto es tan importante un buen diseño de pruebas.

La automatización permite reducir costos, en este caso concreto, automatización se puede definir como realizar un proceso de manera automática mediante herramientas software. Ahora bien, la automatización se puede hacer en todas las fases de la prueba: Diseño de las pruebas, Codificación de las pruebas (Menos habitual, pocas técnicas y herramientas), Ejecución de las pruebas, Evaluación de los resultados.

Con la automatización se logran grandes ventajas como por ejemplo: mayor rapidez de ejecución, disminución de los recursos y se evita el hecho de realizar pruebas que ya

están obsoletas. Los inconvenientes pueden ser que existen muy pocas herramientas, se necesita una “base” a menudo inexistente, un conjunto pobre de pruebas.

También es recomendable identificar previamente las condiciones que se deben cumplir para la automatización, para cumplir con lo anterior se debe contar con modelos y documentación del sistema, se deben controlar las herramientas de prueba y no pretender automatizar todo. Por otra parte, no se debe cuando se gasta más en la automatización que en las pruebas, y también en los casos en que se construye el sistema sobre la marcha.

2.5 Problemas de las pruebas manuales.

Tabla 2.2.5 Definición de verificación y validación.

El proceso de determinar cuándo los requisitos de un sistema son completos y correctos, cuándo los productos de cada una de las fases de desarrollo satisfacen los requisitos o condiciones impuestas por las fases anteriores y cuándo el sistema final o componente cumple los requisitos especificados.

Las primeras definiciones de pruebas no han variado hasta nuestros días. Ya en el año 1979 [22] definía el proceso de prueba del software como la búsqueda de fallos mediante la ejecución del sistema o de partes del mismo en un entorno y con valores de entrada bien definidos.

Sin embargo, este proceso aplicado a las pruebas del sistema puede resultar tedioso y repetitivo. Si estas pruebas se desarrollan manualmente por un conjunto de técnicos, el proceso de prueba consistiría en introducir manualmente diversas combinaciones de valores de prueba y comparar los resultados con los resultados esperados. Estos técnicos deben repetir el mismo conjunto de pruebas cada vez que se modifique el sistema. Este proceso provoca fatiga y falta de atención, lo cual repercute en su calidad y nivel de

detección de pruebas. Además requiere el gasto de los recursos necesarios para tener a una persona, o grupo de personas repitiendo las pruebas [23].

Los informes de fallos son elaborados por las personas en función de sus apreciaciones, por lo que si una persona está cansada o confunde los resultados de distintas pruebas puede dar lugar a informes erróneos.

La idea de que un proceso de desarrollo de software facilita la construcción del sistema y aumenta su calidad es mundialmente aceptada en la actualidad. Esta misma idea puede y debe aplicarse a los procesos de prueba y, en especial, a la identificación, selección y generación de casos de prueba.

2.6 Costes del proceso de prueba.

Un estudio de 1027 proyectos de sistemas de información revela que solo el 12,7% concluyó satisfactoriamente. La baja calidad de los requisitos fue considerado el principal motivo de fracaso en el 76% de los proyectos [3].

Los problemas más comunes en los requisitos son: ambigüedad, imprecisión, inexactitud e inconsistencia. Asegurar la calidad de los requisitos es una tarea casi imprescindible para culminar un proyecto de desarrollo de software con éxito. Existen varias técnicas ampliamente aceptadas y documentadas para mejorar la calidad de los requisitos, como pueden ser revisiones formales, inspecciones, listas de verificación, ontologías, auditorias, matrices de rastreabilidad, métricas de calidad, etc.

Desde mediados de los años setenta se estima que el coste de corregir un error en un sistema software aumenta a medida que avanza el desarrollo del sistema (ver figura). El coste de corregir un error en las últimas etapas se estima entre 50 y 100 veces por encima del coste de corregir en mismo error en las primeras etapas.

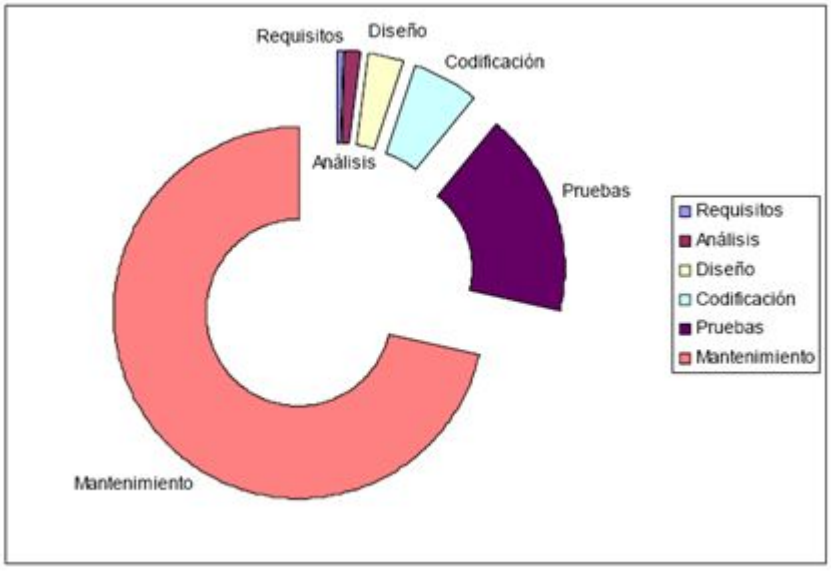


Figura 6.2.6 Coste de corrección de errores en las etapas del desarrollo del software.

Otras estimaciones posteriores [24] confirman esta tendencia. Cuanto antes sea identificado y corregido un error, menor será el coste de su corrección [24]. En la tabla siguiente se muestra el coste relativo de corregir un error, dependiendo de donde sea detectado, en dólares USA.

Tabla 3.2.6 Coste relativo en dólares de la corrección de errores (1987).

Fase	Coste
Definición	$1
Diseño de alto nivel	$2
Diseño de bajo nivel	$5
Codificación	$10
Pruebas unitarias	$15
Pruebas de integración	$22
Pruebas del sistema	$50

Producción	$100+

Otros estudios más actuales, como [26], siguen confirmando esta tendencia. Las estimaciones de [26] se muestran en la tabla siguiente.

Tabla 4.2.6 Coste de corrección de errores (2001).

Fase	Porcentaje
Requisitos	1
Diseño	3-6
Codificación	10
Pruebas unitarias / integración	15-40
Pruebas de sistema / aceptación	30-70
Producción	40-1000

Actualmente, estas estimaciones siguen teniendo plena vigencia y valores similares. Como se aprecia en la ilustración anterior, los costes de corregir errores en la fase de prueba son mucho mayores que los costes de corregir dichos errores en fases tempranas, como la fase de requisitos o de análisis. Adelantar la fase de pruebas a la fase de requisitos o de análisis permite reducir el coste de corrección de errores. Esto debe complementarse con estrategias que permitan, además, disminuir el número de errores potenciales.

Sin embargo, como se ha visto en la sección 1, la realización de una prueba implica la ejecución del código. Para probar es necesario disponer del código del sistema. Por este motivo no es posible adelantar toda la fase de pruebas a etapas tempranas como la fase de requisitos o análisis.

Sin embargo, sí es posible adelantar la generación de casos de prueba a etapas tempranas. La generación de pruebas en la etapa de requisitos o análisis va a permitir encontrar fallos, inconsistencias, omisiones, ambigüedades y sobre-especificaciones en las especificaciones del sistema cuando es más económico corregirlas.

Además, adelantando la generación de pruebas se evita dejar todo el proceso de prueba para final de la construcción, momento donde el tiempo y los recursos disponibles suelen ser insuficientes para realizar este proceso adecuadamente.

2.7 Acotación del problema de búsqueda.

Es imposible probar un sistema al 100% [2]. Por ejemplo, suponer una función suma que admite como parámetros dos valores de tipo entero y devuelve un valor de tipo entero que es la suma de dichos parámetros. Para probar ese método de manera exhaustiva y garantizar que no va a suceder ningún error con el 100% de exactitud, se necesitaría probar el método con cada uno de los valores enteros posibles.

Como, en teoría, existen infinitos valores enteros se obtendrían infinitos casos de prueba. Sin embargo, todo microprocesador y sistema operativo manipula números enteros dentro de un rango de valores concreto. Por ejemplo, en el lenguaje Java existen 2^32 posibles valores enteros.

Probando todas las posibles combinaciones se obtendrían 2^32 * 2^32 posibles casos de prueba, por lo que el resultado final serían:

18.446.744.073.709.551.616 casos de prueba. Realizando 100 millones de comprobaciones por segundo, se tardarían más de 5800 años. Un valor demasiado alto para ejecutarlos en la práctica.

Muchas de esas pruebas desbordarán el tamaño de un entero, por lo que se pueden seleccionar sólo aquellas combinaciones de valores que no desborden el valor de un entero y añadir una prueba adicional que sí lo desborde. En este caso se tiene:

4,294.967.296 + 1 casos de prueba.

De una forma sencilla se ha conseguido reducir el número de valores de prueba de infinito a algo más de 4.000 millones, un número mucho más manejable. Sin embargo aún sigue siendo un número demasiado alto. ¿Por qué 4.000 millones de pruebas es un número de pruebas demasiado elevado para utilizarlas en la práctica?.

Se podría responder que tantas pruebas consumirían mucho tiempo tanto preparándolas como ejecutándolas. Pero esto no es cierto. Los procesadores cada vez son más rápidos y potentes, por lo que solo sería cuestión de adquirir uno más potente y construir un generador automático de pruebas.

La verdadera razón que responde a la pregunta de porqué 4.000 millones de pruebas es un número demasiado alto e imposible de llevar a la práctica es porque no son necesarias

tantas pruebas. Se puede encontrar fácilmente, utilizando el sentido común, un conjunto de no más de 6 o 12 pruebas que prueben todas las distintas combinaciones posibles. Si con 12 pruebas es posible verificar adecuadamente la función, ¿para qué realizar 4.000 millones de pruebas?.

Un posible listado de estas pruebas se muestra en la tabla siguiente.

Tabla 5.2.7 Conjunto de pruebas.

Sumando A	Sumando B	Resultado
Positivo	Positivo	Positivo
Positivo	Negativo	Positivo
Positivo	Negativo	Negativo
Negativo	Positivo	Positivo
Negativo	Positivo	Negativo
Negativo	Negativo	Negativo

Positivo	Positivo	Desbordamiento
Negativo	Negativo	Desbordamiento

¿Por qué hacer 4.000 millones de pruebas cuando solo con 8 es suficiente?. No hay ninguna razón para sobreprobar (overtesting) el sistema. Sin embargo, localizar el conjunto mínimo de pruebas que verifican, casi al 100%, un sistema no es una tarea sencilla. Por este motivo necesitamos técnicas y metodologías que nos guíen en dicho proceso.

Capítulo 3

Estudio de la Situación Actual.

Existen varias propuestas para la generación de objetivos de prueba. En total, los estudios revisados han sido de 12 propuestas, de las cuales, para efectos de este proyecto se han expuesto en profundidad cuatro de ellos, por ser los que han tenido como resultado un nivel más alto de aceptación, para tomar a partir de éstos, ideas claves para la propuesta que se explicará a continuación.

Los siguientes párrafos resumen las propuestas estudiadas y sus principales puntos fuertes y débiles.

Se considera justificado mostrar el estudio comparativo de propuestas de generación de casos de prueba a partir de especificaciones funcionales, debido a la falta de estudios que integren y evalúen las propuestas que se mostrarán, y de manera de servir de base para la propuesta final que se planteará, a partir de los resultados obtenidos y expuestos en el informe de avance anterior.

En este punto se van a citar brevemente algunos de los trabajos más relevantes a nivel académico en la generación de casos de prueba.

3.1. Requirements by Contract.

Esta propuesta, llamada a partir de ahora RBC, se divide en dos partes. En la primera propone extender los casos de uso de UML mediante contratos. Estos contratos incluyen pre y poscondiciones y casos de uso parametrizados. En la segunda parte se muestra cómo generar automáticamente casos de prueba a partir de los casos de uso extendidos. El trabajo describe cómo aplicar esta propuesta a familias de productos.

El punto de partida de esta propuesta es un diagrama de casos de uso según la notación de UML. Al final del proceso descrito en esta propuesta se obtiene, un modelo de casos de uso extendido con contratos y un conjunto de casos de prueba que verifican la

implementación de dicho modelo de caso de uso. Los casos de uso se expresan como caminos de ejecución que recorren secuencias de casos de uso que satisfagan sus precondiciones y poscondiciones. El conjunto de pasos de esta propuesta y su resultado se muestra en la tabla siguiente:

Tabla 6.3.1 Conjunto de pasos para la generación de pruebas.

Paso	Descripción	Resultado
1	Extensión de los modelos de caso de uso mediante contratos.	Casos de uso con parámetros, precondiciones y poscondiciones.
2	Construcción del modelo de ejecución de casos de uso.	Modelo de ejecución de casos de uso.
3	Selección del criterio de cobertura.	Criterio de cobertura para recorrer el modelo de ejecución.
4	Aplicación del criterio al modelo de ejecución.	Secuencias de casos de uso válidas.
5	Generación de pruebas.	Pruebas ejecutables sobre el sistema.

RESUMEN DE VENTAJAS E INCONVENIENTES.

Ventajas.

• Ofrece una herramienta experimental de libre descarga con la implementación de los criterios propuestos.

• Permite generar pruebas basadas en secuencias de casos de uso.

• Los contratos son un medio muy flexible para expresar las dependencias de un caso de uso respecto de otros casos de uso.

Inconvenientes.

• Extiende los casos de uso de una manera no estándar según los mecanismos de extensión de UML.

• No describe como generar pruebas que verifiquen la implementación de un caso de uso de manera aislada.

• No ofrece ningún criterio que indique cuantos parámetros distintos debe utilizar un caso de uso.

• No existe ninguna relación entre los parámetros simbólicos utilizados y sus valores reales.

3.2. PLUTO y Category Partition Method.

La propuesta Product Lines Use case Test Optimization (PLUTO) [1] describe como generar casos de pruebas para familias de productos. El punto de partida son casos de uso para familias de productos. Esta propuesta extiende la notación de casos de uso con plantillas y lenguaje natural, para expresar los elementos comunes a todos los productos y los puntos de variabilidad donde cada producto presenta unas características concretas distintas de los demás productos.

El resultado de esta propuesta es un conjunto de casos de prueba comunes para todos los productos de la familia y, además, un conjunto de casos de prueba específicos para cada producto de la familia. Estos casos de prueba se describen mediante plantillas en lenguaje natural.

Las pruebas generadas por esta propuesta son descripciones abstractas de un caso de prueba que deberá ser refinado para obtener un caso de prueba ejecutable sobre el sistema. Sin embargo PLUTO no indica ni ofrece referencias de cómo realizar este proceso [1].

El proceso de generación de casos de prueba propuesto en PLUTO es una adaptación del proceso de partición de categorías (Category Partition o CP) puesto al día para trabajar

con especificaciones modeladas mediante casos de uso para familias de productos. Los 8 pasos de esta propuesta son básicamente los mismos que los pasos de CP y se describen en la tabla siguiente.

Tabla 7.3.2 Conjunto de pasos para la generación de pruebas.

Paso	Descripción	Resultado
1	Descomposición de la especificación funcional del sistema en unidades funcionales.	Unidades funcionales que pueden probarse de manera independiente.
2	Identificación de parámetros y condiciones del entorno que afectan a la ejecución de cada unidad funcional.	Conjunto de entradas del sistema y estado del sistema necesario para la ejecución de cada unidad funcional
3	Búsqueda de categorías.	Una categoría para cada parámetro y condición del sistema que indicará el rango que de valores que podrá tomar.
4	División de cada categoría en elecciones.	Conjunto de elecciones de cada categoría.
5	Identificación de restricciones.	Lista de restricciones de cada elección.
6	Redacción de especificaciones de prueba.	Listado de especificaciones de prueba.
7	Generación de marcos de prueba.	Marcos de prueba a partir de las especificaciones.
8	Generación de scripts de prueba.	Conjunto de scripts, cada uno de ellos conteniendo varias de las pruebas generadas.

RESUMEN DE VENTAJAS E INCONVENIENTES.

Ventajas.

• Ofrece un ejemplo práctico no trivial.

• Permite verificar el comportamiento de un caso de uso teniendo en cuenta sus dependencias con otros casos de uso.

• Se puede aplicar en etapas tempranas, como la elicitación de requisitos.

• Indica como reducir el número de casos de prueba generados (mediante restricciones, por ejemplo, erróneas).

Inconvenientes.

• Las elecciones están muy basadas en la experiencia de los ingenieros de pruebas y el conocimiento que estos tienen sobre el sistema.

• No ofrece información sobre la cobertura de las pruebas. Al dejar la elección de valores al criterio de los ingenieros de pruebas puede no ofrecer una cobertura adecuada.

• No hay referencias a herramientas de soporte existentes

• Aunque puede ser automatizada, al dejar muchos elementos a la decisión de los ingenieros de pruebas no puede alcanzar la automatización total.

• Si los casos de uso no manipulan datos, esta propuesta se muestra inútil.

• Puesto que las elecciones dependen de los criterios de los ingenieros de pruebas no se puede estimar a priori el número de casos de prueba necesarios para verificar la implementación de un caso de uso.

• Falta una guía sistemática que indique como identificar categorías y elecciones.

• La inclusión de clasificaciones erróneas en un número mayor o menor de casos de prueba depende del criterio del ingeniero de prueba.

• No explica como mezclar y agrupar las pruebas en scripts.

3.3. Extended Use Case Test Design Pattern.

Esta propuesta muestra cómo generar pruebas a partir de casos de uso extendidos con información adicional. El punto de partida son casos de uso descritos en lenguaje natural. Esta propuesta no exige ningún formalismo para representar los casos de uso, pero sí exige un conjunto mínimo de información.

Al final de la aplicación de esta propuesta se obtiene una tabla con todos los posibles casos de prueba para verificar cada uno de los casos de uso. El conjunto de pasos de esta propuesta y su resultado se muestra en la tabla siguiente.

Tabla 8.3.3 Conjunto de pasos para la generación de pruebas.

Paso	Descripción	**Resultado**
1	Identificación de variables operacionales.	Listado de variables operacionales para un caso de uso.
2	Definición del dominio de variables operacionales.	Dominio de cada una de las variables Operacionales.
3	Identificación de relaciones operacionales Lista de relaciones existentes entre las distintas variables operacionales.	Lista de relaciones existentes entre las distintas variables operacionales.
4	Generación de casos de prueba	Casos de prueba.

RESUMEN DE VENTAJAS E INCONVENIENTES.

Ventajas.

• Esta propuesta ofrece una referencia clara de cuando puede comenzar el proceso de generación de pruebas.

• Incluye una referencia al estándar IEEE 982.1 como referencia para evaluar la cobertura de las pruebas generadas.

Inconvenientes.

• La propuesta describe la información que debe aparecer en un EUC, pero no dice como añadirla un caso de uso. No da ninguna notación, plantilla o extensión a UML.

• Está completamente basada en lenguaje natural, por lo que sería difícil desarrollarla de manera automática con una herramienta software.

• No se ofrecen referencias a aplicaciones reales.

• No detalla qué se obtiene al final de su aplicación ni cómo generar pruebas ejecutables.

3.4. A Model-based Approach to Improve System Testing of Interactive Applications.

Esta propuesta (TDEUML) está basada en los trabajos sobre generación de casos de prueba desarrollados por la empresa Siemens desde 1992 [28], año de aparición de la primera versión de su herramienta TDE centrada en las pruebas unitarias. El punto de partida de esta propuesta es la documentación de casos de uso en lenguaje natural. Al final de la aplicación de esta propuesta se obtiene un conjunto de pruebas de un sistema a través de su interfaz gráfica. Estas pruebas son ejecutables en herramientas que permitan interactuar con una interfaz gráfica a partir de las instrucciones almacenadas en un script de prueba.

En el paso de modelado del comportamiento del sistema se construye un modelo de comportamiento del sistema. Esta propuesta propone los diagramas de actividades de UML para describir el comportamiento interno de cada caso de uso. Una vez construidos los diagramas de actividades, se completan con la especificación de requisitos de prueba. Esta propuesta define los requisitos de prueba como un conjunto de estereotipos que

servirán de guía a la herramienta para indicarle como debe interpretar cada actividad. Cada actividad debe ser anotada con uno de los estereotipos siguientes: <<UserAction>>, <<SystemResponse>> o <<Include>>. Mediante estos estereotipos es posible indicar si es una actividad del usuario o del sistema o está descrita mediante otro diagrama de actividades. Estos estereotipos serán interpretados por el generador de pruebas a la hora de construir pruebas ejecutables. A partir de los diagramas de actividades y sus estereotipos se aplica el segundo paso de la tabla anterior. En primer lugar se construye el diseño de pruebas. Esta propuesta define el diseño de pruebas como la descripción de los diagramas de actividades en lenguaje TSL. A partir de este diseño en lenguaje TSL es posible generar guiones o scripts de prueba. Esta propuesta propone la herramienta TDE para realizar la traducción de TSL a scripts de prueba.

El resultado de la aplicación de esta herramienta es un conjunto de guiones de pruebas ejecutables en una herramienta de verificación de interfaces gráficas.

El tercer paso consiste en la ejecución de estos guiones sobre el sistema en pruebas. Las primeras ejecuciones pueden servir también para refinar y mejorar el conjunto de scripts de pruebas generado. El proceso de generación de casos de prueba se puede resumir en tres pasos. Estos pasos se describen en la tabla siguiente.

Tabla 9.3.4 Conjunto de pasos para la generación de pruebas.

Paso	Descripción	Resultado
1	Modelado del comportamiento del sistema	Diagramas de actividades con requisitos de prueba.
2	Diseño de casos de prueba	Conjunto de scripts de prueba.
3	Ejecución automática de casos de prueba	Conjunto de scripts de prueba refinados.

RESUMEN DE VENTAJAS E INCONVENIENTES.

Ventajas.

• Indica como generar pruebas ejecutables sobre el sistema.

• El uso de estereotipos permite añadir información necesaria para la automatización del proceso de generación de pruebas respetando el estándar UML.

• El uso de un lenguaje formal facilita también la automatización.

Inconvenientes.

• Se centra en interfaces gráficas, por lo que no permite generar casos de prueba para aplicar a otro tipo de interfaces externas de un sistema que permitan la comunicación, no con humanos, sino con otros sistemas.

• No describe con detalle como construir los diagramas de actividades,

• Tampoco describe si es necesario un diagrama de actividades por cada caso de uso o un único diagrama para todo el sistema.

• No explica cómo incluir en los diagramas de actividades dependencias entre casos de uso.

• No se ha encontrado ninguna versión de evaluación o con licencia educativa para evaluar la herramienta.

3.5. Resultados del Análisis.

Tabla 10.3.5 Investigaciones Analizadas.

Título	Código
Requirements to Testing in a Natural Way	RETNA
Requirements by Contract	RBC

PLUTO y Category Partition Method	PLUTO
A UML-Based Approach to System Testing	TOTEM
Derivation of Domain Test Scenarios from Activity Diagrams.	TDEUML
Automated Test Case Generation from Dynamic Models [	ATCGDM
Test Cases from Use Cases	TCUC
Scenario-Based Validation and Test of Software (SCENT)	SCENT
Requirement Base Testing	RBC
Software Requirements and Acceptance Testing	AT
Testing from Use Cases Using Path Analysis Technique	UCPA
Use Case Derived Test Cases	UCDTC
Extended Use Case Test Design Pattern	EUC

Tabla 11.3.5 Análisis Comparativo Investigaciones Pruebas a partir de requerimientos.

	ATCGDM	SCENT	TCUC	UCPA	PLUTO	RBC	TOTEM	RBT	EUC	AT	UCDTC	TDEUML	RETNA
Punto Partida	CU	Usuario	CU	CU	CU(17)	CU	Varios (13)	CU	CU	Usuario	CU	CU	Usuario
Nueva Notación	No	Si	No	Si	No	Si (11)	Si (12)	No	No	Si (2)	No	Si (3)	No
Notación Gráfica	Si	Si	No	Si	No	Si	Si	Si	No	Si	No	Si	No
Caso Estudio	No	Si	No	No	No	No	No	No	No	Si	No	No	No
Uso de Standards	Si	No	No	No	No	Si	Si	Si (8)	No	No	Si	Si	Si
Herramientas	Si (14)	No	No	No	Si (15)	Si	No	Si (9)	No	No	No	Si (9)	Si
Casos Prácticos	Si	Si	Si	Si	Si	Si	Si	No	Si	Si	Si	Si	Si
Criterio Cobertura	Varios (6)	Análisis Caminos	Análisis Caminos	Análisis Caminos	Partición Categorías	Varios	Varios	No (10)	Análisis Caminos	Varios	Análisis Caminos	Varios	Análisis Caminos
Valores Prueba	No	No	No	No	Si	No	No	No	Si	No	No	Si	No
Selección de Casos de Prueba	No	No	No	Si	Si	Si	No	No	IEEE**	No	No	No	No
Dependencias Casos de Uso	Si	Si	No	No	Si	Si	Si	No	No	Si	No	No	No Indicado
Modelo Comportamiento	Si	Si	No	Si	No	Si	Si	Si	No	Si	No	Si	Si
Prioridad Casos de Uso	No	No	No	Si	Si	No	Si	No	No	No	No	Si	Si
Enfoque activo (5)	?	No	?	?	?	Si	No	?	?	?	?	Si	No
Comienzo	Requerimientos	Requs	Requs	Requs	Requs	Requs	Análisis (18)	Requs (16)	Requs	Requs	Requs	Requs	Requs
Resultados	Test Por cada Caso de Uso	Escenarios Casos Prueba lenguaje Natural	Casos Prueba lenguaje natural	Casos Prueba lenguaje natural	Casos Prueba lenguaje natural	Secuencias Transición	Modelos Prueba	Casos Prueba lenguaje natural	Casos Prueba lenguaje natural	Modelos Prueba	Casos Prueba lenguaje natural	Scripts Prueba	Scripts Prueba
Calidad Documentación	Media	Media	Media	Media	Media	Media	Alta (7)	Baja	Media	Baja	Baja	Media (4)	Baja
Implementación Caso Prueba Formato Caso de Uso													
Actividades	7	15+3 (1)	3	5	8	4	8	12	4	6	2	3	6
Año	2000	1999 / 2000	2002	2002	2004	2000 / 2001	2002	2001	1999	1997	2002	2004	2004

** - Incluido en [1] y [13].

(1) 15 actividades para obtener escenarios y 3 para obtener casos de prueba.

(2) Una notación para uso de árboles y otra para máquinas de estados finitos.

(3) Define sus propios estereotipos para diagramas de actividades UML.

(4) Aunque hay suficiente documentación, el enfoque ha sido diseñado para usarse con herramientas de soporte.

(5) “?” significa que no hubo contacto con los autores.

(6) Todos los estados o todas las transiciones.

(7) Sin embargo es difícil de poner en práctica debido a que la documentación es incompleta.

(8) Se asume que los diagramas causa – efecto son una herramienta ampliamente usada.

(9) No está disponible una versión de evaluación.

(10) Criterios de cobertura han sido implementados directamente en una herramienta de soporte. No se ha encontrado alguna referencia a ese criterio.

(11) Diagramas de transición de casos de uso.

(12) Define una notación textual para representar secuencias de ejecución de casos de uso.

(13) Diagrama de casos de uso UML , descripciones textuales de casos de uso, diagramas de secuencia o colaboración para cada caso de uso, diagrama de dominio de clases, diccionario de datos con descripciones para cada clase, método, atributo y restricción en OCL (Object Constraint Language).

(14) Herramientas de Planificación.

(15) Herramientas para generar plantillas de prueba.

(16) Después de la toma, verificación y validación de requerimientos.

(17) Para familia de productos.

(18) Requerimientos estables.

En esta sección se exponen las conclusiones extraídas del análisis realizado según las propuestas estudiadas, de las cuales se mostró un breve resumen en el punto anterior.

La primera conclusión obtenida a partir del análisis del estado de la situación actual realizado es que ninguna propuesta está completa. Esto significa que ninguna ofrece todo lo necesario para generar un conjunto de pruebas del sistema ejecutables sobre el sistema bajo prueba a partir de la especificación funcional del mismo.

Como se ha visto en la descripción de cada propuesta, la mayoría tienen algunos puntos fuertes. Sin embargo, todas presentan algún tipo de carencia que dificulta su puesta en práctica.

También se ha detectado que, en general, las propuestas realizan un estudio incompleto del problema que quieren resolver. Muchas no contemplan elementos que se consideran fundamentales en cualquier propuesta de generación de prueba, como la generación de valores concretos de prueba o la evaluación del grado de cobertura de los requisitos.

La carencia de documentación también es un hecho importante, la mayoría de las propuestas tienen una calidad media en su documentación. Esto significa que, a la hora de ponerlas en práctica, aparecen situaciones o dudas no resueltas por la documentación. Estas situaciones o dudas deben ser resueltas a partir de la experiencia, conocimientos o intuición del equipo humano que las aplique, lo cual les resta efectividad.

En general, ninguna propuesta parte de otras propuestas. Las propuestas incluidas en este trabajo han sido desarrolladas de manera aislada, sin estudiar el estado del arte, investigar las propuestas ya existentes y sin aprovechar los puntos fuertes y corregir los puntos débiles de otras propuestas. Esto queda patente en un conjunto de propuestas similares, como EUC que está basada en el análisis de caminos y prácticamente no aporta nada nuevo respecto a las demás.

Una carencia grave detectada es la falta de consistencia de los resultados en las propuestas analizadas. La mayoría de las propuestas, en sus resúmenes o títulos prometen que permiten obtener pruebas. Tal y como se han definido las pruebas en la sección 1 de este trabajo, esto consiste en obtener un conjunto de valores de prueba, un conjunto de interacciones con el sistema y un conjunto de resultados esperados. Sin embargo, al final de la mayoría de las propuestas, se obtiene un conjunto de tablas según el formato particular de cada propuesta. En ninguna propuesta se obtienen pruebas directamente ejecutables (salvo TDEUML y PLUTO pero contando con las herramientas adecuadas), ni contemplan la generación de los resultados esperados.

Ninguna propuesta incluye referencias ni información sobre cómo evaluar la calidad de las pruebas generadas, excepto EUC. Todas las propuestas parten de la premisa de que, si se aplican adecuadamente sus pasos, se obtiene un conjunto de pruebas con la máxima calidad o cobertura de los requisitos.

Un aspecto a considerar y que está poco o nada recogido en las propuestas estudiadas, es la automatización del proceso de generación de pruebas mediante herramientas software. El análisis pone de relieve que, en general, no es posible una automatización completa debido a que los requisitos están expresados en lenguaje natural. Cada una de las propuestas tiene un grado de automatización distinto, en función del detalle en que se definen sus pasos y el uso de formalismos y modelos. Muchas de las propuestas no siguen ningún estándar, por ejemplo el estándar IEEE para el formato de requisitos o casos de uso, o modelos ampliamente aceptados como los modelos propuestos por UML.

Las propuestas existentes pueden clasificarse en tres grupos dependiendo de los artefactos usados para la generación de objetivos de prueba. El primer grupo engloba las propuestas que generan objetivos de prueba directamente a partir de los casos de uso. El segundo grupo engloba las propuestas que generan un modelo de comportamiento a partir de los casos de uso y genera casos de prueba a partir de él, como [18], [13] o [17]. Algunas de las notaciones utilizadas para el modelo de comportamiento son: diagramas de actividades, diagramas de estados o árboles de escenario. El tercer grupo engloba propuestas centrabas en variables operacionales y valores de prueba. Estas propuestas identifican las variables de un caso de uso y realizan particiones de los dominios de dichas variables, como [1] y [16]. Las propuestas basadas directamente en los casos de uso escritos en prosa son difíciles de sistematizar y automatizar. Tampoco se ha encontrado ninguna propuesta sistemática para identificar variables, dominios y particiones. Las propuestas basadas en modelos de comportamiento están descritas con poco detalle y no ofrecen herramientas de soporte.

Las propuestas existentes también pueden clasificarse en dos grupos dependiendo del alcance de los objetivos generados. El primer grupo engloba las propuestas que generan objetivos de prueba a partir de casos de uso de manera aislada, como [1] o [17]. El

segundo grupo engloba las propuestas que generan objetivos de prueba a partir de secuencias de casos de uso, como [18] o [13]. Sin embargo, ninguna de las propuestas del primer grupo permite obtener de manera automática modelos de comportamiento ni objetivos de prueba a partir de casos de uso. Los párrafos anteriores justifican la necesidad de ésta propuesta para generar un modelo de comportamiento y obtener casos de prueba de manera automática.

Capítulo 4

Elaboración Proceso de Pruebas.

4.1 Elaboración Proyecto.

Al tratarse de planificación de pruebas a partir de la etapa de toma de requerimientos a través de casos de uso, se visualiza el diseño de pruebas como de caja negra, ya que un caso de uso tiene como característica el hecho de describir la funcionalidad del sistema y el comportamiento del mismo mediante su interacción con los usuarios y/o otros sistema, escondiendo detalles de la implementación, y de las estructuras de control que hay involucradas en el funcionamiento de cada paso, acción o decisión que se realizan.

Por otra parte, lo que se generará a partir de los casos de uso serán los llamados objetivos de prueba lo que corresponde a las acciones del sistema a nivel de su interacción con un actor externo esto se logrará gracias al diagrama de actividades relacionado al caso de uso desde donde se podrá identificar el conjunto total de escenarios, es decir, todas las posibilidades de caminos de ejecución, además dicho modelo se podrá visualizar como un grafo, lo que permitirá calcular el número mínimo de casos de prueba necesarios para cubrir todas las acciones a realizar en un casos de uso, medida conocida como la Complejidad Ciclomática [26], dicha información será relevante para quien esté enfocado en el proceso de planificar las pruebas, teniendo como característica fundamental el hecho de que se podrá hacer desde etapas tempranas en el desarrollo del sistema.

Por lo dicho anteriormente, se puede decir que este es un enfoque “mezclado” para el diseño de las pruebas, ya que lo relacionado al cálculo del conjunto de pruebas mínimo tiene relación a un enfoque de caja blanca que se puede complementar de muy buena forma con el enfoque de caja negra descrito recientemente.

4.2 Marco de Trabajo.

Para comenzar a enmarcar lo que será la propuesta de proceso de planificación de pruebas se verán los conceptos que se integrarán, los de pruebas estructurales referidos a la complejidad ciclomática [26] y los de pruebas funcionales a los que hacen mención los casos de uso.

4.1.1 Pruebas Estructurales.

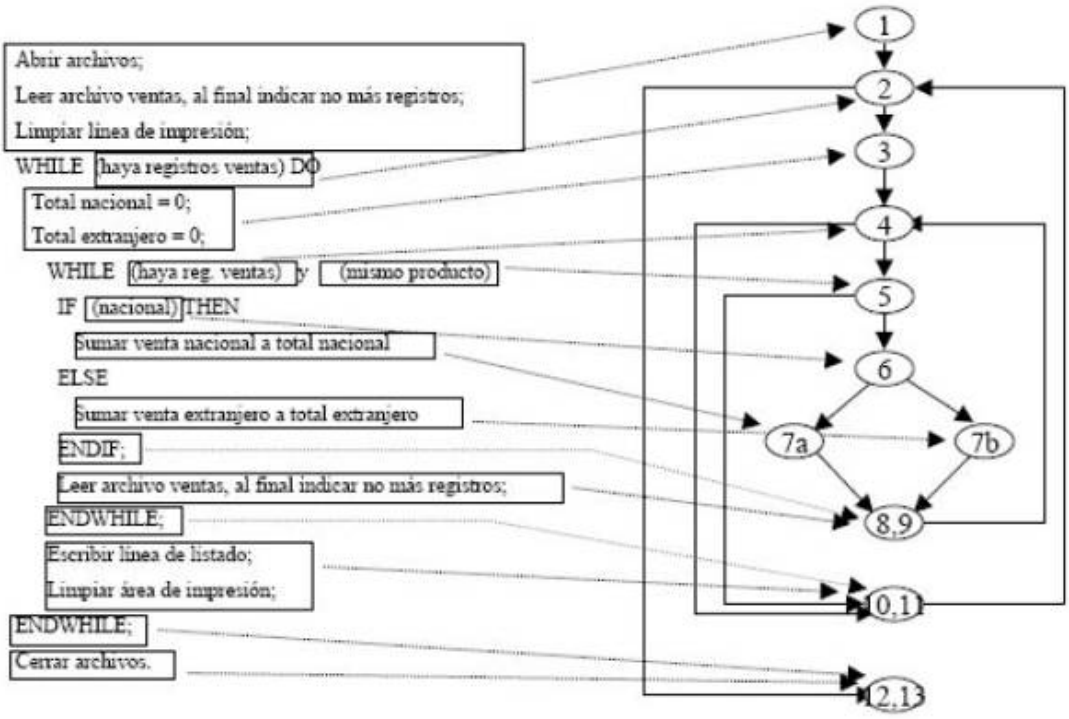


Figura 7.4.1.1 Grafo de flujo de un programa (Pseudocódigo)

El diseño de casos de prueba tiene que estar basado en la elección de caminos importantes que ofrezcan una seguridad aceptable de que se descubren defectos (un programa de 50 líneas con 25 sentencias if en serie, da lugar a 33,5 millones de secuencias posibles), para los que se usan los criterios de cobertura lógica.

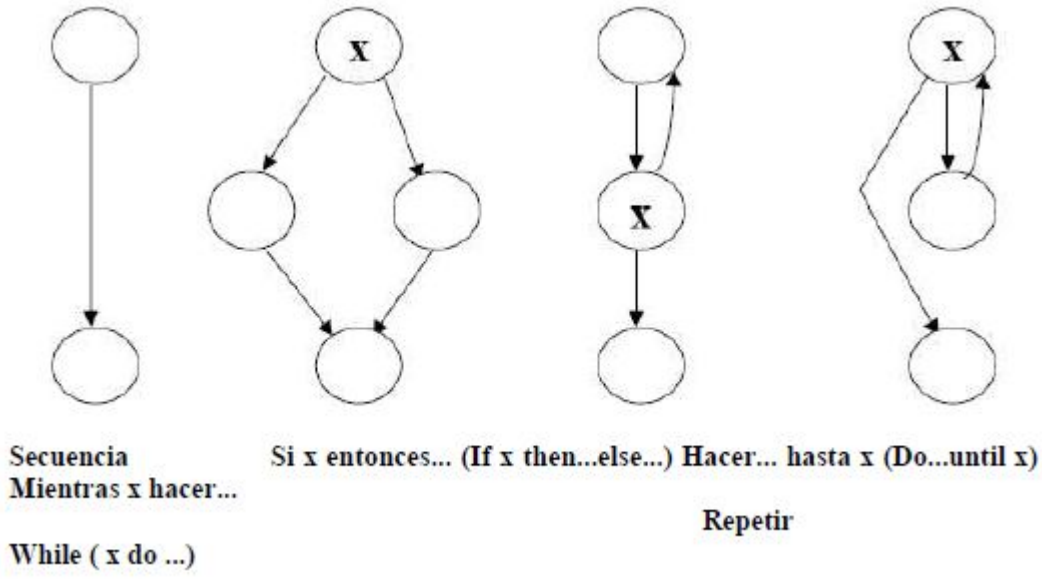


Figura 8.4.1.1 Grafo de flujo de las estructuras básicas de programa.

Consejos:

-Separar todas las condiciones

-Agrupar sentencias ‘simples’ en bloques

-Numerar todos los bloques de sentencias y también las condiciones

4.1.2 Criterios de Cobertura Lógica.

Cobertura de sentencias. Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.

Cobertura de decisiones. Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso. (Incluye a la cobertura de sentencias)

Cobertura de condiciones. Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez. (No incluye cobertura de condiciones).

Criterio de decisión/condición. Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.

Criterio de condición múltiple. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea, se puede considerar que cada decisión multicondicional se descompone en varias condiciones unicondicionales.

Criterio de cobertura de caminos. Se recorren todos los caminos (impracticable)

4.2 La Complejidad de McCabe V(G) (Complejidad Ciclomática).

La Complejidad Ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. La métrica, propuesta por Thomas McCabe en 1976 [20], se basa en la representación gráfica del flujo de control del programa. De dicho análisis se desprende una medida cuantitativa de la dificultad de prueba y una indicación de la fiabilidad final. Cuando se utiliza en el contexto del método de prueba del camino básico, el valor calculado como complejidad ciclomática define el número de caminos independientes del conjunto básico de un programa y nos da el límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez. Es una de las métricas de software mas ampliamente aceptada, ya que ha sido concebida para ser independiente del lenguaje. Se ha medido un gran número de programas, de modo de establecer rangos de complejidad que ayuden al ingeniero de software a determinar la estabilidad y riesgo inherente de un programa. La medida resultante puede ser utilizada en el desarrollo, mantenimiento y reingeniería para estimar el riesgo, costo y estabilidad. Algunos estudios experimentales indican la existencia de distintas relaciones entre la métrica de McCabe y el número de errores existentes en el código fuente, así como el tiempo requerido para encontrar y corregir esos errores. Se suele comparar la complejidad ciclomática obtenida contra un conjunto de valores límite como se observa en la tabla siguiente:

Tabla 12.4.2 Complejidad ciclomática vs. evaluación de riesgo

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

La Complejidad ciclomática es moderadamente sensitiva a cambios en el programa.

4.2.1 Ámbito de utilización de la Complejidad Ciclomática.

La complejidad ciclomática puede ser aplicada en varias áreas incluyendo:

• Análisis de Riesgo en desarrollo de código: Mientras el código esta en desarrollo, su complejidad puede ser medida para estimar el riesgo inherente.

• Análisis de riesgo de cambio durante la fase de mantenimiento: La complejidad del código tiende a incrementarse a medida que es mantenido durante el tiempo [26]. Midiendo

la complejidad antes y después de un cambio propuesto, puede ayudar a decidir cómo minimizar el riesgo del cambio.

• Planificación de Pruebas: El análisis matemático ha demostrado que la complejidad ciclomática indica el número exacto de casos de prueba necesarios para probar cada punto de decisión en un programa.

• Reingeniería: Provee conocimiento de la estructura del código operacional de un sistema. El riesgo involucrado en la reingeniería de una pieza de código está relacionado con su complejidad.

Como se mencionó en la sección anterior, el ámbito más común de utilización es la prueba estructural (caja blanca) para probar módulos unitarios. McCabe también expone que se puede utilizar la complejidad ciclomática para dar una indicación cuantitativa del tamaño máximo de un módulo. A partir del análisis de muchos proyectos encontró que un valor 10 es un límite superior práctico para el tamaño de un módulo. Cuando la complejidad supera dicho valor se hace muy difícil probarlo, entenderlo y modificarlo. La limitación deliberada de la complejidad en todas las fases del desarrollo ayuda a evitar los problemas asociados a proyectos de alta complejidad. El límite propuesto por McCabe sin embargo es fuente de controversias. Algunas organizaciones han utilizado el valor 15 con bastante éxito.

No obstante, un valor superior a 10 debería ser reservado para proyectos que tengan ventajas operacionales con respecto a proyectos típicos, por ejemplo staff con gran experiencia, diseño formal, lenguaje de programación moderno, programación estructurada, entre otros.

La complejidad de McCabe se puede calcular de cualquiera de estas 3 formas

1. V (G) = a - n + 2, siendo a el número de arcos o aristas del grafo y n el número de nodos.

2. V (G) = r, siendo r el número de regiones cerradas del grafo.

3. V(G) = c + 1, siendo c el número de nodos de condición.

El criterio de prueba de McCabe es: Elegir tantos casos de prueba como caminos independientes (calculados como V(G)) La experiencia en este campo asegura que: V(G) marca el límite mínimo de casos de prueba para un programa. Cuando V(G) > 10 la probabilidad de defectos en el módulo o programa crece mucho quizás sea interesante dividir el módulo.

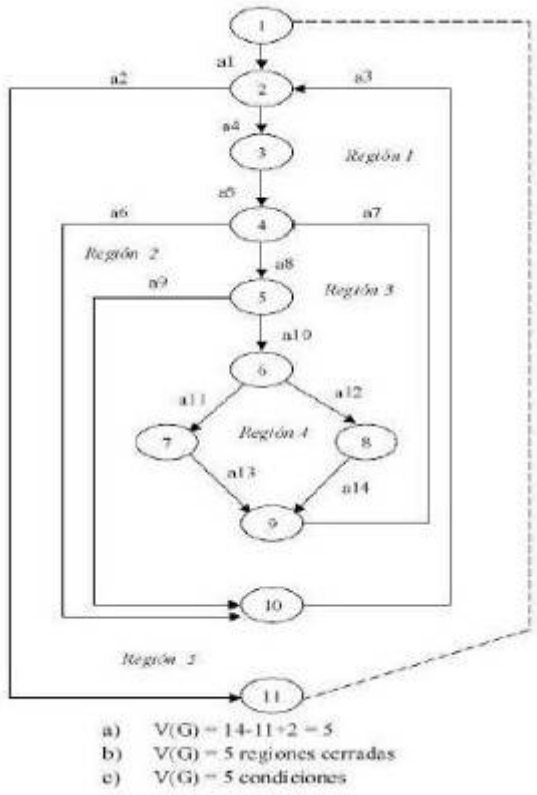


Figura 9.4.2.1 Cálculo de la complejidad ciclomática sobre un grafo.

El objetivo final para el cálculo de la complejidad ciclomática es obtener el modelo de comportamiento del sistema y asemejarlo a un grafo como el anterior, dicho modelo para efectos de este proyecto será el diagrama de actividades de cada caso de uso como el que se muestra a continuación.

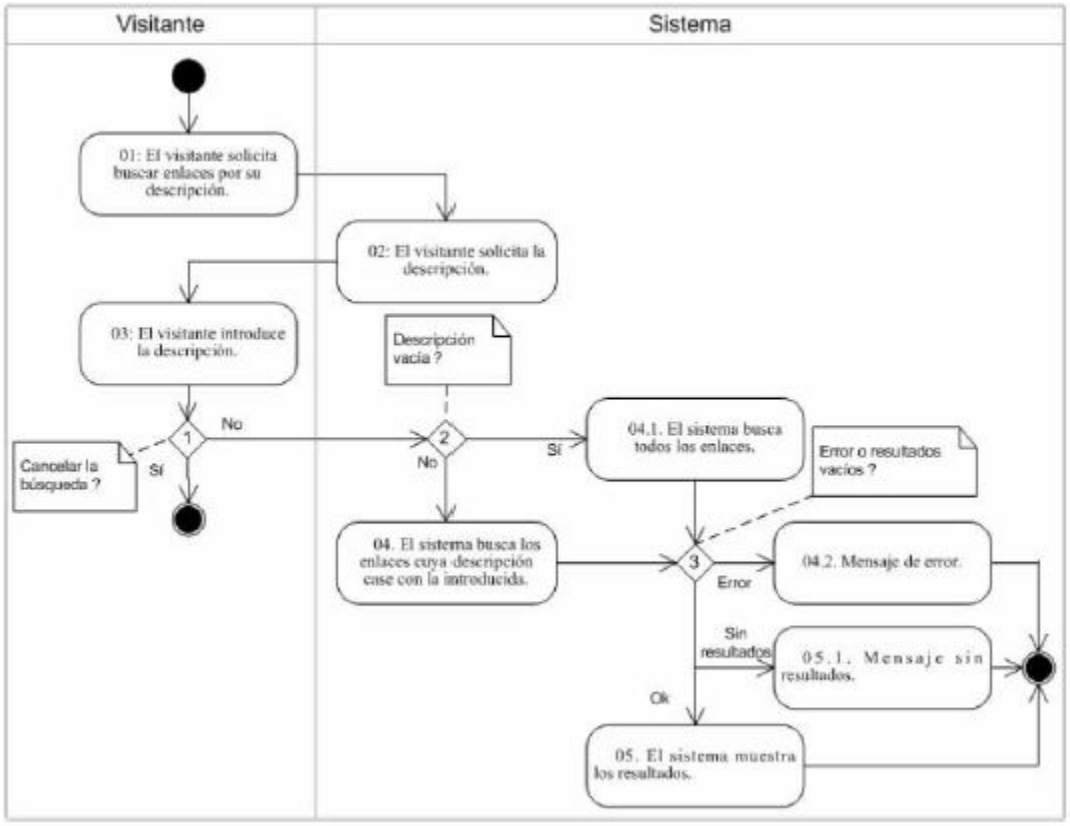


Figura 10.4.2.1 Modelo de Comportamiento del sistema visualizado como un grafo.
Se puede observar que existe relación al observar ambos diagramas, por lo tanto se puede decir que un diagrama de actividades tiene un recorrido del tipo de un grafo.

Además dicho diagrama presenta como característica el hecho de que se puede observar claramente quién está ejecutando la acción de cada paso del caso de uso, para este caso ya sea el actor o el sistema.

Para el diagrama de actividades la complejidad de McCabe se puede calcular de cualquiera de la siguiente forma.

V(G) = c + 1, siendo c el número de nodos de condición.

Para este caso, los nodos de condición serán las actividades de decisión.

Capítulo 5

Proceso de Planificación de Pruebas de Sistema.

5.1 Definición de Requisitos Funcionales.

5.1.2 Definición de casos de uso para procesos de generación automática de pruebas del sistema.

A la hora de estudiar cómo obtener casos de prueba a partir de los requisitos de un sistema, es vital plantearse cómo deben estar descritos dichos requisitos. Actualmente una gran cantidad de desarrollos software parten de requisitos expresados en lenguaje natural, mediante plantillas, completados con modelos de casos de uso según la notación propuesta por UML. Los trabajos que componen la base de este proceso de prueba (principalmente los trabajos relacionados con NDT) [8] también abordan los requisitos funcionales expresados con casos de uso. Estos casos de uso se definen mediante lenguaje natural y plantillas. Por este motivo se ha adoptado casos de uso, en lenguaje natural y plantillas, y complementados con diagramas de casos de uso como el punto de partida de este proceso de generación de pruebas.

Uno de los objetivos de este trabajo es definir la información que deben tener los casos de uso, cómo deben estar redactados para poder generar pruebas a partir de ellos y desarrollar un prototipo de herramienta software.

Para lograr estos objetivos se han desarrollado las siguientes actividades:

• Estudio de distintas propuestas de clasificación de requisitos funcionales y casos de uso con el fin de encontrar el nivel adecuado para el proceso de generación de pruebas. (Análisis realizado sección 3).

• Estudio de distintas propuestas de definición de casos de uso con el fin de evaluar los distintos elementos que un caso de uso puede tener. Los resultados son:

• Definición del nivel de detalle adecuado para los casos de uso.

• Propuesta de una plantilla de casos de uso y de normas para la definición de los elementos.

• Casos prácticos.

• Desarrollo de una herramienta para la generación de un diagrama de actividades a partir de un caso de uso definido con esta plantilla.

Como se ha indicado anteriormente. Los requisitos funcionales o casos de uso no son los únicos requisitos implicados en la generación de pruebas. Sin embargo este trabajo está centrado sólo en requisitos funcionales.

En esta sección se expone la propuesta elegida para este proyecto con el fin de definir requisitos funcionales mediante casos de uso. En el siguiente punto se resume la propuesta relevante para este proyecto centrada en casos de uso.

5.2 Propuesta NDT (Navigational Development Techniques).

NDT [8] es una técnica para especificar, analizar y diseñar el aspecto de la navegación en aplicaciones web. Para este trabajo, solo es relevante la propuesta que ofrece para la definición y captura de requisitos. El flujo de especificación de requisitos de NDT comienza con la fase de captura de requisitos y estudio del entorno. Para ello, plantea el uso de técnicas como las entrevistas y JAD (joint application design) [7]. Tras esta fase, se propone la definición de los objetivos del sistema. En base a estos objetivos, el proceso continúa definiendo los requisitos que el sistema debe cumplir para cubrir los objetivos marcados. NDT [8] clasifica los requisitos en:

- Requisitos de almacenamiento de información

- Requisitos de actores

- Requisitos funcionales

- Requisitos de interacción, representados mediante:
  - Frases, que recogen cómo se va a recuperar la información del sistema utilizando un lenguaje especial denominado BNL (Bounded Natural Language)
  - Prototipos de visualización, que representan la navegación del sistema, la visualización de los datos y la interacción con el usuario.

- Requisitos no funcionales

Todo el proceso de definición y captura de requisitos y objetivos que propone NDT se basa principalmente en plantillas o patrones, pero también hace uso de otras técnicas de definición de requisitos como son los casos de uso y los glosarios. La propuesta ofrece una plantilla para cada tipo de requisito, lo que permite describir los requisitos y objetivos de una forma estructurada y detallada. Algunos de los campos de los patrones son cerrados, es decir, solo pueden tomar valores predeterminados. Estos campos permiten que en el resto del proceso del ciclo de vida de NDT se puedan conseguir resultados de forma sistemática. El flujo de trabajo de especificación de requisitos termina proponiendo la revisión del catálogo de requisitos y el desarrollo de una matriz de trazabilidad que permite evaluar si todos los objetivos han sido cubiertos en la especificación [8].

A continuación se muestra la propuesta relevante del proyecto que describe los elementos que debe tener un caso de uso y cómo definirlos. A continuación se enumera brevemente los elementos propuestos por NDT para la definición de requisitos funcionales.

• Identificador.

• Descripción.

• Precondiciones.

• Actores.

• Secuencia normal.

• Post-condición.

• Alternativas / Excepciones.

• Rendimiento.

• Frecuencia esperada.

A partir de lo anterior, se define en este punto un modelo de requisito funcional que permita obtener pruebas del sistema de una manera automática.

5.2.1 Extensión del modelo de requisitos de NDT.

Como se ha visto en el informe de avance anterior, las propuestas de generación de pruebas utilizan un modelo de casos de uso con elementos similares. El modelo de requisitos funcionales y plantillas de NDT ya incluye todos esos elementos, por lo que es un punto de partida válido.

Sin embargo, se considera necesario modificar alguno de los elementos existentes en el modelo de NDT y extenderlo con nuevos elementos. Los elementos del modelo NDT que se extenderán son: la secuencia normal y las alternativas / excepciones. Los nuevos elementos son: los resultados, prioridad y los disparadores. Estos elementos se detallan en los siguientes párrafos.

Disparador:

Los casos de uso siempre deben comenzar por la acción de un actor o un evento externo al propio sistema. El disparador (o trigger) es el nombre que recibe dicha acción o evento. Una excepción son los casos de nivel de sub-función los cuales son invocados desde otro caso de uso y nunca directamente por ningún actor.

Cuando sólo existe un único disparador, no es necesario indicarlo explícitamente en el caso de uso. En este caso, el primer paso de la secuencia principal debe ser el disparador. Si existieran varios disparadores distintos (por ejemplo una acción de un actor y un evento de tiempo), sí deben añadirse explícitamente.

El elemento disparador, se puede definir con una frase en lenguaje natural, similar a la redacción de precondiciones o post-condiciones.

Resultados:

Uno de los tres elementos imprescindibles en cualquier caso de prueba son los resultados esperados. Para facilitar la generación de resultados, se va a añadir un nuevo elemento al modelo de requisitos de NDT que permitan definir con precisión cuáles son los resultados esperados tras la ejecución de un caso de uso, tanto si se logra la meta como si no. Estos resultados pueden expresarse como la ejecución de otros casos de uso o la obtención de una información o mensaje del sistema [8]. La siguiente tabla muestra los distintos resultados posibles. Estos resultados pueden extenderse con nuevos tipos.

Los resultados también indican qué pasos de un requisito funcional dan por concluida la ejecución del requisito. Un resultado siempre es una respuesta del sistema, aunque el paso pertenezca al actor.

Tabla 13.5.2.1. Tipos de resultados de para de uso.

Resultado	Descripción
Ejecución de X.	El caso de uso termina y, justo a continuación, se ejecuta el caso de uso X. No confundir con una extensión o inclusión que no son resultados dado que el caso de uso original no termina.
El caso de uso termina.	La ejecución del caso de uso se da por concluida.
Vacío.	No indicar ningún resultado significa que se realiza el siguiente paso de

	la secuencia principal.

Un resultado no es lo mismo que una post-condición. Una post-condición es una condición que el sistema cumple a la finalización exitosa del caso de uso. Esa condición puede no ser apreciable por el actor. Sin embargo un resultado es una acción del sistema observable por el actor.

Secuencia normal.

Para poder manipular un requisito funcional de manera automática mediante una herramienta software es necesario evitar la ambigüedad del lenguaje natural. Para ello se propone el uso de un patrón lingüístico en la definición de pasos de la secuencia principal. Este patrón lingüístico se muestra a continuación. Todos los elementos entre corchetes son obligatorios.

Tabla 14.5.2.1 Patrón lingüístico para la secuencia normal.

Español:	[Número]. [Artículo] [actor / sistema / el evento] [verbo de acción] [lo que hace].
Inglés:	[Number]. [actor / system / event] [action verb][action performed].

En la tabla siguiente, se muestra un ejemplo de secuencia normal redactada mediante el patrón de la tabla anterior. Dicha secuencia corresponde al caso de uso añadir un nuevo enlace, del sistema para el mantenimiento de un catálogo on-line de enlaces.

Tabla 15.5.2.1 Ejemplo de secuencia normal redactada con el patrón lingüístico.

Id	Paso

1	El actor usuario selecciona la opción para añadir un nuevo enlace.
2	El sistema selecciona la categoría “Top” y solicita la información del nuevo enlace.
3	El actor usuario introduce la información sobre el enlace.
4	El sistema valida el nuevo enlace y lo almacena.

Secuencias alternativas.

También se propone para este proyecto un patrón para expresar los pasos alternativos a la secuencia principal de un caso de uso. Dicho patrón se muestra en la tabla siguiente y se describe a continuación. El elemento {"con el resultado" Z} aparece entre llaves en lugar de corchetes porque es opcional.

Tabla 16.5.2.1 Patrón para secuencias alternativas o erróneas.

Español:	[Identificador]. [Tipo] [Cuando / Si] X "entonces" Y {"con el resultado" Z}
Inglés:	[Identifier]. [Type] [When / If] X “then” Y {“with the result” Z}

El primer elemento del patrón es el identificador. Este identificador está compuesto por el número del paso de la secuencia normal al que complementa seguido de un número que identifica a una alternativa concreta (dado que un mismo paso de la secuencia principal puede tener varias alternativas). El segundo elemento del patrón es el tipo de la alternativa. El tipo se asigna en función del orden de ejecución de la alternativa con respecto al paso principal. Los distintos tipos propuestos así como su significado se enumeran en la tabla siguiente. Pueden añadirse más tipos si fuera necesario.

Tabla 17.5.2.1 Tipos de alternativas.

Tipo	Descripción
[Pre]	Una alternativa de este tipo se evalúa antes de la ejecución del paso.
[Inv]	Una alternativa de este tipo se evalúa durante la ejecución del paso. En el instante en que esta alternativa se cumple, se detiene la ejecución del paso y se realiza la alternativa.

El tercer elemento es la palabra “Cuando” o “Sí”. Se puede elegir la que calce mejor con el texto de la alternativa. El elemento X del patrón describe en lenguaje natural la condición que debe ser cierta para que se ejecute la alternativa. El elemento Y del patrón describe las acciones a realizar cuando se ejecuta la alternativa. Opcionalmente, se puede añadir un elemento Z que indica qué debe pasar después de ejecutar el elemento Y del patrón. El elemento Z generalmente tomará uno de los valores de la tabla de resultados (tabla 16). Cuando el resultado de la acción ya se indica en el elemento Y no siendo necesario incluir el elemento Z.

Prioridad:

Este elemento valora la importancia y relevancia del caso de uso en el sistema. Si la prioridad la establecen los usuarios, indica la importancia que los usuarios otorgan al caso de uso. Si la prioridad la establecen los desarrolladores, indica la relevancia del caso de uso para determinar la arquitectura del sistema o la complejidad de diseño e implantación del caso de uso. La prioridad puede clasificarse mediante un conjunto de términos (por ejemplo baja, media, alta) o bien de manera numérica (por ejemplo con un valor entre 1 y 5 ambos inclusive).

5.2.2 Plantilla de casos de uso.

En la tabla siguiente se define el modelo de plantilla de casos de uso para recoger todos los elementos propuestos por NDT junto a las extensiones descritas en el punto anterior.

Tabla 18.5.2.2 Plantilla para casos de uso.

Name	UC-01. …
Precondition	…
Triggers *	…

Main sequence	1	The actor….. **
	2	The system …
	…	…
	4	The system stores the new link.
Errors alternatives	1.1.i	If the system … then … and the result is ….
	2.1.p	If the actor … then … and the result is ….
	3.1.i	At any time, the [system/actor] may …. then …. and the result is …
Results	1	System …
	…	…

Postcondition	…

Reliability	…
Priority	…

* Sólo cuando existan más de uno.

** Si sólo hay un disparador, se coloca aquí.

En la sección siguiente se muestra un ejemplo de casos de uso redactado con la plantilla de la tabla anterior.

Capítulo 6

Generación de Pruebas de Sistema a partir de Casos de Uso.

6.1. El Proceso de Generación de Pruebas del Sistema.

Toda prueba consta tradicionalmente de tres elementos: interacciones entre el sistema y la prueba, valores de prueba y resultados esperados. Los dos primeros elementos permiten realizar la prueba y el tercer elemento permite evaluar si la prueba se superó con éxito o no.

Un proceso de pruebas consta generalmente de cuatro fases: la fase de diseño de pruebas, la fase de codificación, la fase de ejecución y la fase de análisis de los resultados. El objetivo de un proceso de generación de pruebas del sistema es desarrollar las dos primeras fases y obtener esos tres elementos a partir del modelo de requisitos del propio sistema bajo prueba. Dicho proceso toma como punto de partida los requisitos y, a partir de ellos genera los resultados y construye las pruebas. La figura siguiente ilustra un proceso genérico que recoge las ideas principales extraídas después de realizar un estudio comparativo sobre 12 propuestas. A partir de este estudio comparativo y de varios casos prácticos, se ha identificado un conjunto de actividades pertenecientes al proceso de generación de pruebas de la figura siguiente que son independientes de la plataforma de la implementación. Es decir, dichas actividades no se ven afectadas si, por ejemplo, el sistema a prueba es un sistema web o un sistema de escritorio monousuario. De esta manera, es posible generar un conjunto de pruebas independientes de la plataforma. Sólo es necesario conocer los detalles de la plataforma a la hora de implementar las pruebas generadas.

6.1.1 Modelos de Prueba.

En este punto se describen un conjunto de modelos de prueba independientes y dependientes de la plataforma (PITs y PDTs). Los modelos descritos se muestran en la figura.

Los óvalos de la figura representan los distintos modelos implicados. Los óvalos sombreados representan los modelos de requisitos, los óvalos claros representan los modelos independientes de prueba y los óvalos a rayas representan los modelos dependientes. Las líneas entre modelos implican las dependencias y las futuras

transformaciones. Todos los modelos siguen el Testing Profile de UML 2.0 [19] siempre que ha sido posible. Los modelos de la figura 2 se describen en los siguientes puntos.

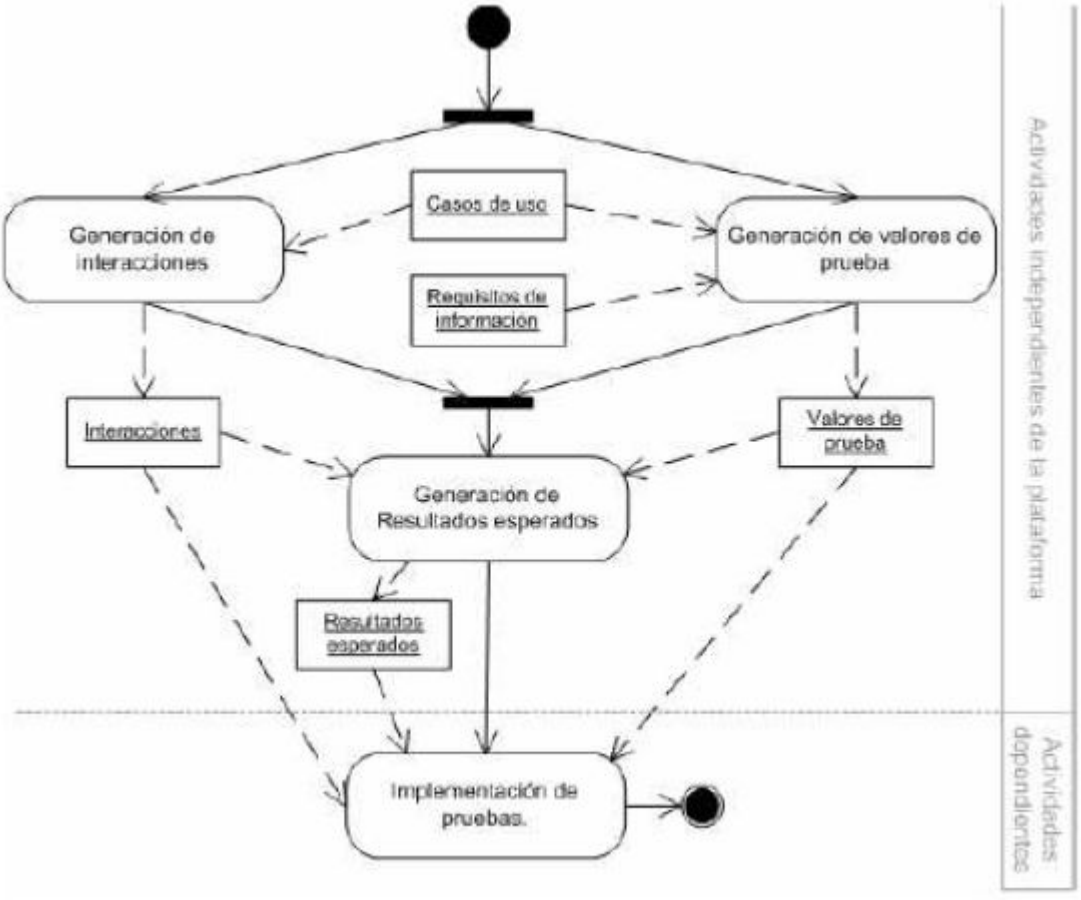


Figura 11.6.1.1 Proceso de generación de pruebas.

A lo largo de este trabajo se muestran ejemplos basados en uno de los casos prácticos realizado durante el desarrollo de estos modelos. El sistema a prueba es una aplicación web para la gestión de un catálogo de enlaces en-línea (www.codecharge.com). En la siguiente sección se definen los modelos de prueba necesarios.

6.1.2 Modelos de requisitos.

Los únicos modelos de requisitos necesarios son los casos de uso y los requisitos de almacenamiento, aunque otros modelos, como por ejemplo modelos de interfaces o modelos de navegación [7] pueden enriquecer el proceso de prueba. Actualmente existen varias propuestas de modelos de requisitos. En concreto, la propuesta que se utiliza en este trabajo, está basada en Navigational Development Techniques (NDT) [7].

6.1.3 Modelo de comportamiento.

Un gran número de técnicas de requisitos están basadas en casos de uso definidos en prosa [5]. Pero no es sencillo manipular programáticamente casos de uso escritos en prosa. Por este motivo, el primer paso de este proceso sistemático de generación de pruebas consiste en expresar dicha prosa mediante un modelo formal manipulable de manera automática.

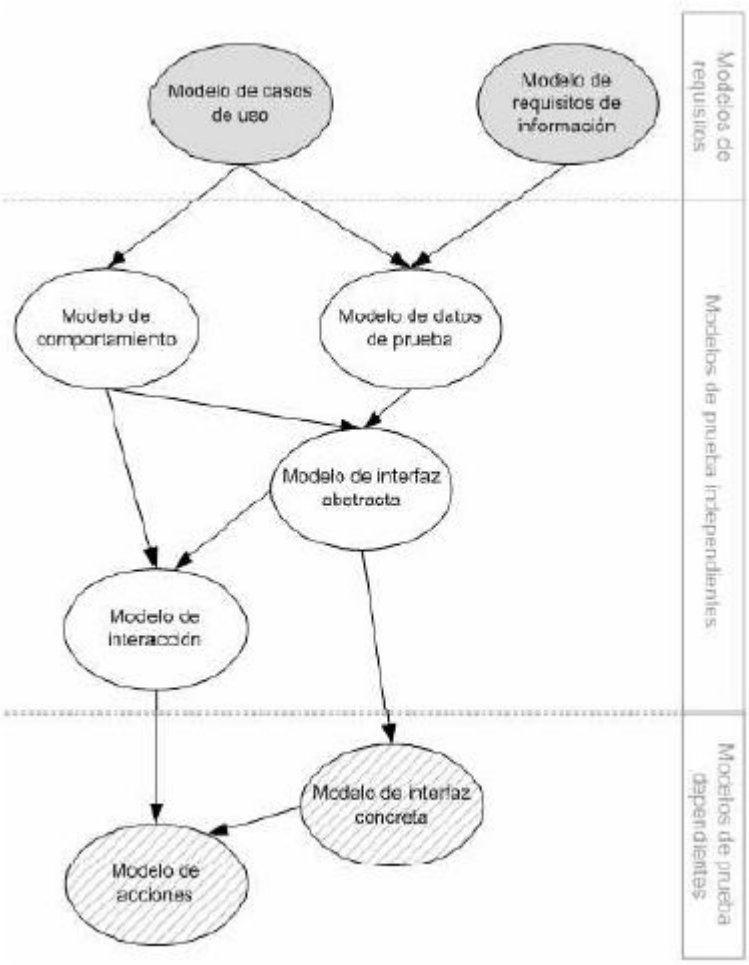


Figura 12.6.1.3 Modelos para la generación de pruebas.

El objetivo del modelo de comportamiento es expresar la misma información contenida en una plantilla de caso de uso de una forma fácilmente manipulable.

Las propuestas estudiadas utilizan como modelos de comportamiento diagramas UML de estados, diagramas UML de secuencia o diagramas UML de actividades [6]. En este trabajo se han seleccionado diagramas de actividades ya que, a diferencia de los diagramas de secuencia, permiten expresar caminos alternativos fácilmente y, a diferencia de los

diagramas de estados, permiten expresar la interacción entre el sistema y los actores externos identificando claramente a cada uno de los participantes.

6.1.4 Modelo de Datos de Prueba.

Los casos de uso contienen elementos variables cuyos valores o comportamiento difiere de una ejecución de un caso de uso a otra [2]. Algunos ejemplos son la información suministrada por un actor, una opción seleccionada por un actor, o la información mostrada por el sistema como resultado del caso de uso.

Los objetivos del modelo de datos de prueba son dos. En primer lugar, el modelo de datos de prueba expresa todas las variables del caso de uso [2], su estructura si son tipos complejos (como clientes o compras), las restricciones que puedan existir entre ellos y las particiones de sus respectivos dominios. Esto se realiza mediante un diagrama de clases según la notación propuesta en el Testing Profile de UML [15]. Dicho diagrama de clases puede extraerse automáticamente. Las clases se obtienen a partir de los requisitos funcionales y las distintas particiones se obtienen a partir de las condiciones evaluadas en las alternativas del diagrama de comportamiento. Este diagrama de clases puede refinarse posteriormente añadiendo particiones adicionales si fuera necesario.

En segundo lugar el modelo de datos de prueba expresa los valores de prueba del sistema y los resultados esperados del mismo. Esto se modela mediante un diagrama de objetos, instanciando las clases identificadas en el modelo de clases.

Por lo tanto, El primer paso será transformar los requisitos en modelos manipulables automáticamente, donde el punto de partida del proyecto son requisitos escritos en lenguaje informal, por lo tanto se hace difícil manipularlos de manera automática. A partir de dichos modelos se obtendrán los objetivos de prueba, donde, como se dijo anteriormente, un objetivo de prueba es algo que se tiene que probar, por lo tanto, se sabe entonces desde este momento que se tiene que probar.

Toda prueba consta tradicionalmente de tres elementos: interacciones entre el sistema y la prueba, valores de prueba y resultados esperados. Los dos primeros elementos permiten realizar la prueba y el tercer elemento permite evaluar si la prueba se superó con éxito o no. Un proceso de pruebas consta generalmente de cuatro fases: la fase de diseño de pruebas, la Fase de codificación, la fase de ejecución y la fase de análisis de los resultados. El objetivo de un proceso de generación de pruebas del sistema es desarrollar

las dos primeras fases y obtener esos tres elementos a partir del modelo de requisitos del propio sistema bajo prueba. Dicho proceso toma como punto de partida los requisitos y a partir de ellos genera los resultados y construye las pruebas.

6.2. Actividades para la generación de casos de prueba.

A continuación se presenta un proceso de seis actividades para la generación de pruebas del sistema a partir de casos de uso. Dichas actividades se muestran en la figura 1 y se describen en los siguientes párrafos.

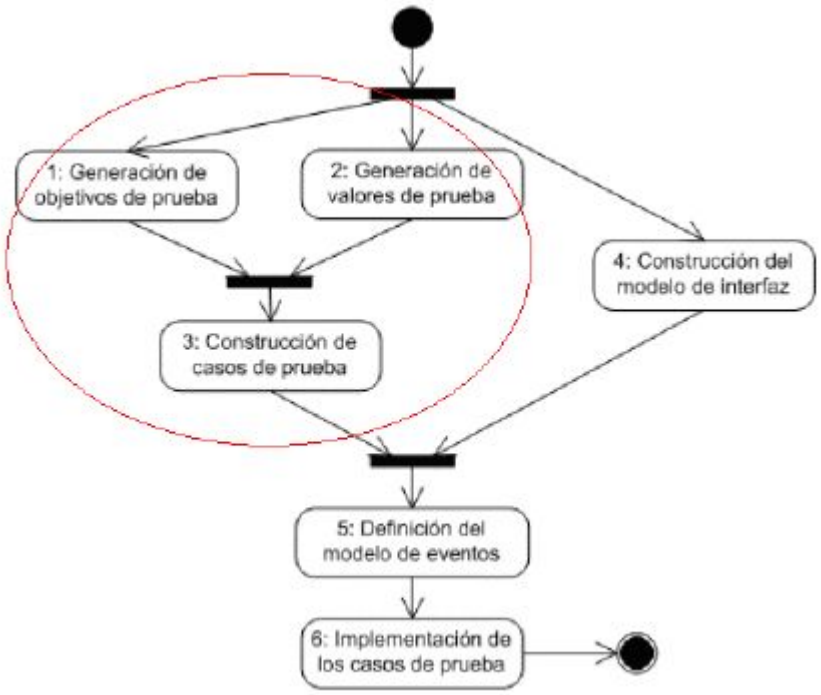


Figura 13.6.2 Actividades para la Construcción de pruebas tempranas.

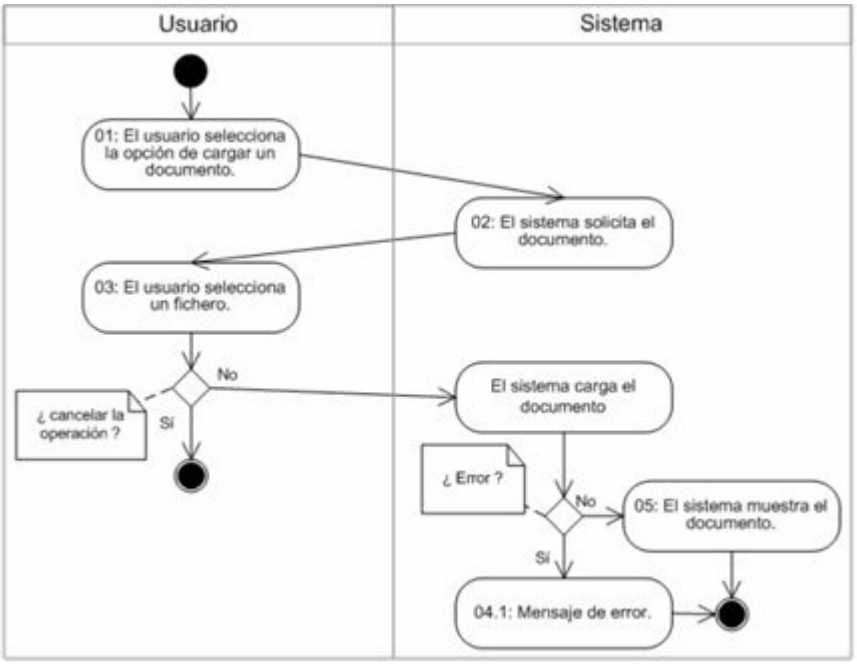
La primera actividad consiste en construir un modelo de comportamiento y obtener objetivos de prueba a partir de él. A continuación, en la segunda actividad, se identifican las variables del caso de uso y se definen los datos de prueba. En la tercera actividad se generan casos de prueba combinando los objetivos de prueba con los valores de prueba. A continuación, en la cuarta actividad se construye el modelo de interfaz. En la quinta actividad, los objetivos de prueba se refinan y se construyen los árbitros que comprobarán si el resultado del caso de prueba es el esperado. Finalmente, toda la información generada se implementa en pruebas ejecutables y test harness (un programa o script que

permite la ejecución y la secuenciación de los casos de prueba). A continuación se definen con mayor detalle las actividades y modelos.

6.2.1 Generación de Objetivos de Prueba.

Un gran número de técnicas de requisitos están basadas en casos de uso definidos en prosa. Pero no es sencillo manipular programáticamente casos de uso escritos en prosa. Por este motivo, el primer paso de este proceso sistemático de generación de pruebas consiste en expresar dicha prosa mediante un modelo formal manipulable de manera automática.

El primer paso es la construcción del modelo de comportamiento. El objetivo del modelo de comportamiento es expresar la información contenida en una plantilla de caso de uso de una forma manipulable sistemáticamente. En este proyecto se han seleccionado diagramas de actividades ya que, a diferencia de los diagramas de secuencia, permiten expresar caminos alternativos fácilmente y, a diferencia de los diagramas de estados, permiten expresar la interacción entre el sistema y los actores externos identificando claramente a cada uno de los participantes. El modelo de comportamiento correspondiente a un caso de uso de cargar un documento se muestra en la figura 12. La generación del modelo de comportamiento puede realizarse de manera automática.



Id	Objetivo de Prueba
1	01,02,03,04,04.1
2	01,02,03,04,04.2
3	01,02,03

Figura 14.6.2.1 Modelo de comportamiento y objetivos de prueba.

En la figura anterior se muestran todos los posibles caminos obtenidos con el criterio de todos las actividades y todas las transiciones. Los objetivos de prueba, en este caso, son la cobertura de transiciones. En la tabla se muestra la definición en XML de un caso de uso de una aplicación para gestionar un catálogo de enlaces. En la figura SIGUIENTE se muestra el diagrama de actividades correspondiente. Dicho diagrama se ha obtenido automáticamente con la herramienta obtenida, que se describirá en el próximo capítulo. Dicha herramienta sólo soporta casos de uso redactados en inglés.

6.2.2 Generación de valores de prueba.

Como se ha mencionado, en primer lugar se identifican las variables. El modelo de comportamiento de la figura anterior revela dos variables. La primera, llamada OpcionUsuario, indica si el usuario selecciona la opción de cargar el fichero o cancelar la operación. La segunda, llamada Fichero, indica el fichero elegido por el usuario. Ambas variables se definen en la figura 15.6.2.2 a y 15.6.2.2 b.

A continuación, el dominio de las variables se divide en categorías. En la figura 15.6.2.2 a y 15.6.2.2 b se muestra una posible partición utilizando la notación propuesta por el UMLTP. Finalmente, para cada partición que se desee probar se selecciona al menos un valor de prueba. Los valores de prueba del caso práctico se muestran en el diagramas de objetos de la figura 15.6.2.2 c.

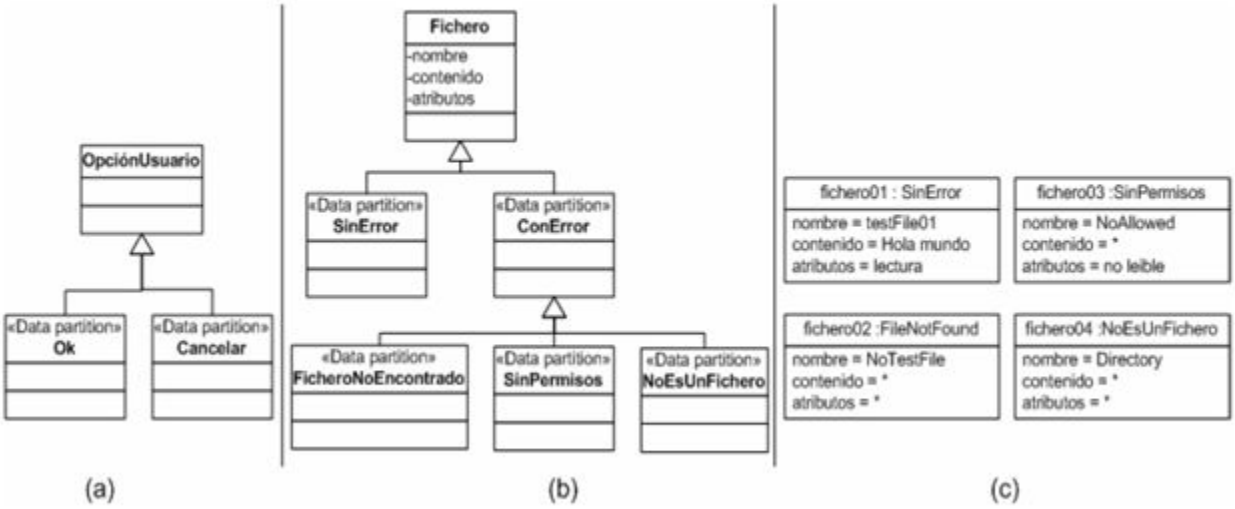


Figura 15.6.2.2 Datos de prueba.

Los casos de uso contienen elementos variables cuyos valores o comportamiento difiere de una ejecución de un caso de uso a otra. Algunos ejemplos son la información suministrada por un actor, una opción seleccionada por un actor, o la información mostrada por el sistema como resultado del caso de uso.

Los objetivos del modelo de datos de prueba son dos. En primer lugar, el modelo de datos de prueba expresa todas las variables del caso de uso, su estructura si son tipos complejos (como clientes o compras), las restricciones que puedan existir entre ellos y las particiones de sus respectivos dominios. Esto se realiza mediante un diagrama de clases según la notación propuesta en el Testing Profile de UML [23] (figura 15.6.2.2 a). Dicho diagrama

de clases puede extraerse automáticamente. Las clases se obtienen a partir de los requisitos funcionales y las distintas particiones se obtienen a partir de las condiciones evaluadas en las alternativas del diagrama de comportamiento. Este diagrama de clases puede refinarse posteriormente añadiendo particiones adicionales si fuera necesario.

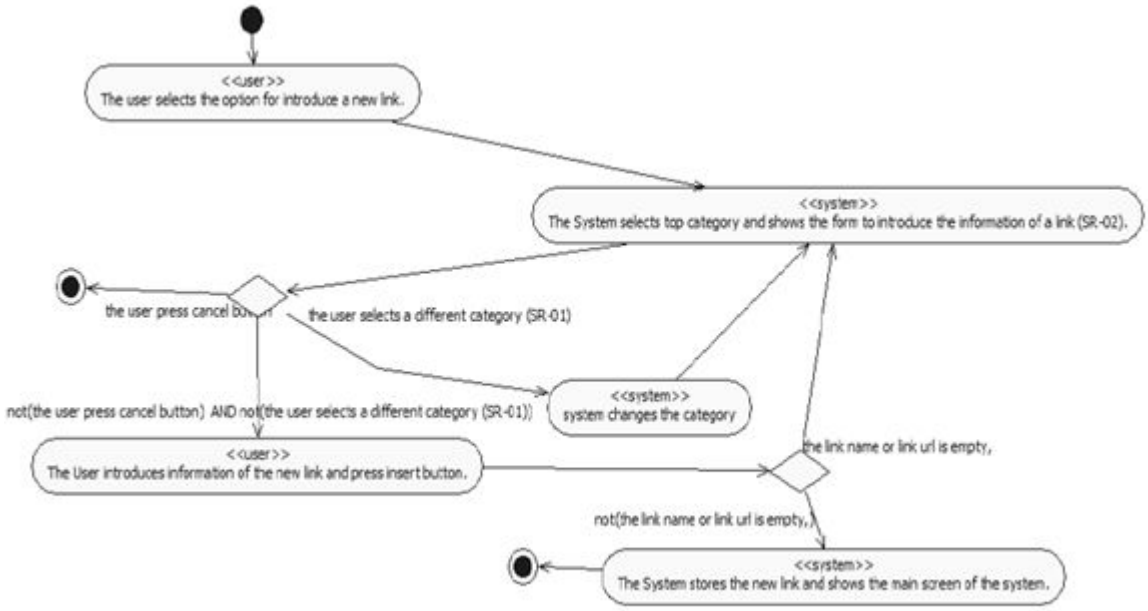


Figura 16.6.2.2 Modelo de comportamiento.

6.2.3 Construcción de los casos de prueba.

En esta actividad se combinan los objetivos y los valores de prueba. A partir de tres objetivos y dos variables se han obtenido un total de 5 casos de prueba. Todas las combinaciones se muestran en la tabla siguiente.

Tabla 19.6.2.3 Combinación de Objetivos y Valores de Prueba.

Id	Objetivo	Valor de Prueba
1	1	OpcionUsuario = Ok, Fichero = fichero01
2	2	OpcionUsuario = Ok, Fichero = fichero02
3	2	OpcionUsuario = Ok, Fichero = fichero03
4	2	OpcionUsuario = Ok, Fichero = fichero04
5	3	OpcionUsuario = Cancelar, Fichero = *

En esta sección se ha mostrado un proceso para la prueba temprana del sistema a partir de los casos de uso del SUT (sistema bajo prueba). Aunque este proceso se ha desarrollado para probar la funcionalidad desde la perspectiva de actores humanos, puede también usarse para probar otros tipos de actores. Este proceso puede aplicarse en etapas tempranas del desarrollo. De hecho, en el caso de estudio todas las pruebas han sido diseñadas antes de elegir una implementación real.

Capítulo 7

Aplicación caso práctico para la generación de objetivos de prueba.

7.1 Un Modelo de Caso de Uso y Plantilla para Pruebas.

Una técnica ampliamente utilizada en la industria para definir casos de uso son las plantillas. Una plantilla combina la prosa con una estructura concreta. La generación sistemática de objetivos de prueba implica una serie de restricciones. Una de ellas es la necesidad de definir un modelo concreto de plantillas de casos de uso que pueda ser manipulado de manera sistemática y automática sin perder la ventaja de usar texto en prosa. Como modelo se ha utilizado el modelo de requisitos propuesto en la metodología Navigational Development Technique (NDT) [8], explicado en detalle en el punto anterior, la cual ofrece un modelo de requisitos completo, formal y flexible.

7.2 Caso práctico

7.2.1 Sistema de almacenamiento y consulta de enlaces.

A continuación se muestra un ejemplo referente a un sistema de almacenamiento y consulta de enlaces, este sistema permite gestionar un catálogo de enlaces en línea. La definición de la operación para añadir nuevos enlaces se define en la tabla siguiente

Tabla 20.7.2.1 Caso de uso (caso práctico).

Name	UC-01. Añadir nuevo enlace
Precondition	No

Main sequence	1 System select “top” category and shows the form to introduce the information of a link (SR-02). 2 User introduces information of the new link and press insert button. 3 System stores the new link.
Error / Alternative sequences	1 At any time, user can press cancel button and exit of the form. 2 If the user selects a different category (SR-01), system changes the category and shows the form again. 3 If link name or link url is empty, system shows an error message and ask the information again
Post condition	No.

En la tabla siguiente se muestra el mismo caso de uso redactado como se ha explicado en la sección anterior.

El sistema a prueba (SUT) es una aplicación web que permite gestionar un catálogo de enlaces “on-line” (de libre descarga desde www.codecharge.com). El SUT contempla dos actores, el actor visitante y el administrador. Sin embargo, este caso de estudio se centrará únicamente en el actor visitante. El diagrama de casos de uso para este actor se muestra en la tabla siguiente.

Tabla 21.7.2.1 Diagrama de casos de uso y caso de uso.

Nombre	UC-02. Buscar enlaces por descripción
Precondición	No.

Secuencia Principal	1. El visitante solicita al sistema buscar enlaces por su descripción. 2. El sistema solicita la descripción. 3. El visitante introduce la descripción. 4. El sistema busca los enlaces cuya descripción case con la introducida por el visitante. 5. El sistema muestra los resultados.
Errores / alternativas	3.1.i. En cualquier momento, el visitante puede cancelar la búsqueda, entonces el caso de uso termina. 4.1.p. Si el visitante introduce una descripción vacía, entonces el sistema busca todos los enlaces almacenados y el resultado es continuar la ejecución del caso de uso. 4.2.i. Si el sistema encuentra un error realizando la búsqueda, entonces muestra un mensaje de error y este caso de uso termina. 5.1.p. Si el resultado de la búsqueda está vacío, entonces el sistema muestra un mensaje y este caso de uso termina.
Resultados	5. El sistema muestra los resultados de la búsqueda. 3.1.i. Fuera de los límites de este caso de uso. 4.2.i. Mensaje de error. 5.1.p. Mensaje de no resultados.
Post condiciones	No

El caso de uso a usar en el caso práctico es “Buscar enlace por descripción”. No se entrará en detalles sobre cómo se muestran los enlaces encontrados. La definición de este caso de uso se muestra en la plantilla de la tabla 17. En dicha plantilla las precondiciones se indican con una letra ‘p’ y las invariantes con una letra ‘i’. Los componentes de la plantilla que no son relevantes en este caso práctico han sido omitidos. En este caso, el resultado del paso 3 sí es redundante, aunque se podría quitar del paso 3 de la secuencia principal.

En cambio, en el paso 2.1.i no se indica cuál es el resultado, por lo que sí está aportando valor de la secuencia principal. El segundo número permite distinguir entre las diferentes alternativas o errores de un paso de la secuencia principal. Cada paso de error o de alternativa debe incluir una condición para indicar cuando se ejecutará el paso. También deben incluir el resultado del paso, por ejemplo, finalizar el caso de uso, continuar el caso de uso o repetir un conjunto de pasos de la secuencia principal.

Capítulo 8

Objetivos de Prueba a partir de Casos de Uso (Caso práctico).

Para obtener objetivos de prueba, en primer lugar se construye un modelo de comportamiento a partir de un caso de uso. En segundo lugar, dicho modelo se recorre para identificar los objetivos de prueba.

8.1. Construcción del modelo de comportamiento.

La primera tarea es construir el modelo de comportamiento a partir del caso de uso. Como se mencionó en la sección 5, se utiliza un diagrama de actividades para representar el comportamiento del caso de uso. Este modelo representa los distintos escenarios o instancias de un caso de uso. Los siguientes párrafos describen los pasos para generar un modelo de comportamiento a partir de un caso de uso definido mediante la plantilla descrita en la sección 5.

Cada paso de la secuencia principal es una actividad del modelo de comportamiento. Las transiciones se añaden entre las actividades consecutivas. Un modelo de comportamiento tiene un único punto de origen, el cuál es la primera actividad del caso de uso.

Un modelo de comportamiento tiene tantos puntos de terminación como resultados. Cada paso de error o alternativa es una decisión en el diagrama de actividades. Dicha decisión evalúa la condición de la alternativa.

Si hay una secuencia de decisiones que pertenecen al mismo actor o al sistema, pueden unirse en una sola decisión. Si el paso alternativo o erróneo incluye alguna acción, se representa como una nueva actividad.

Finalmente, las actividades se agrupan mediante clasificadores. Cada clasificador contiene todas las actividades y decisiones realizadas por un actor o por el sistema. La

figura 6 muestra el modelo de comportamiento obtenido a partir del caso de uso de la figura 5. El algoritmo “BuildBehaviourModel” describe el proceso de generación de los casos de prueba.

Tabla 22.8.1. Proceso de generación de casos de prueba

```
algorithm BuildBehaviourModel
var
        model : ACTIVITYGRAPH
        alternativeSteps : LIST[USECASESTEP]
        step : USECASESTEP
init
foreach (step in useMase.mainSequence)
        alternativeSteps = useCase.getAlternatives(pre, step)
        if ( not_empty(alternativeSteps) )
                traverse_alternativeSteps(behaviourModel, alternativeSteps)
        end if
        behaviourModel.addActivity(step)
        alternativeSteps = useCase.getAlternatives(inv, step)
        if ( not_empty(alternativeSteps) )
                traverse_alternativeSteps(behaviourModel, alternativeSteps)
        end if
end foreach

function traverse_alternativeSteps(behaviourModel, alternativeSteps)
        alternative : STEP
        decision = behaviourModel.addDesicion(alternativeSteps))
        foreach ( alternative in alsternativeSteps)
                if ( is_activity(alternative.action) )
                        node = behaviourModel.addActivity(alternative.action)
                end if
                if ( is_end(alternative.action) )
                        node = behaviourModel.addActivity(activityEnd)
                end if
                if ( is_gotoActivity(alternative.action) )
                        node = behaviourModel.getActivity(alternative.action)
                end if
                behaviouralModel.addTransition(decision, node)
        end foreach
end function
```

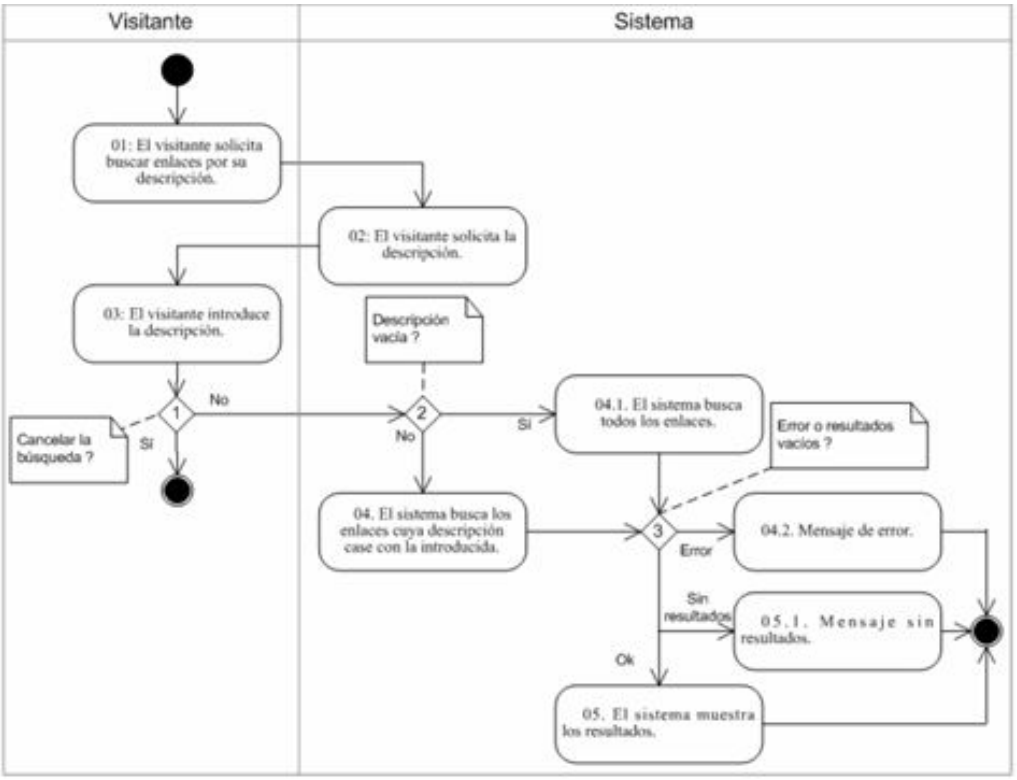


Figura 17.8.1. Modelo de Comportamiento.

Si cada paso del caso de uso incluye una única actividad, entonces el número máximo de nodos (actividades y decisiones) de un modelo de comportamiento es el número de pasos en la secuencia principal y el número de los pasos alternativos multiplicado por 2. Para el caso de uso de la figura 1, el máximo número de nodos es 5 + (4 x 2) = 13. El modelo de comportamiento de la figura 21.7.2.1 contiene sólo 11 nodos ya que el paso 3.1.i no genera ninguna actividad y las condiciones de los pasos 4.2.i y 5.1.p se han unido en la decisión 3.

8.2. Descripción del proceso.

A continuación se describe, el proceso para generar diagramas de actividades a partir de las definiciones de casos de uso. Las tablas muestran dos casos de uso, definidos de una manera abstracta, usados como ejemplo.

Tabla 23.8.2 Definición Abstracta de Caso de Uso.

Name	UC-01. X
Precondition	Not Relevant
Main Sequence	1 The Actor A1 2 The System S1 3 The Actor A2 4 The System S2
Error/ Alternative Sequences	2.1.i If the system X1 then Y1 and the result is to repeat step2. 2.2.i If the system X2 then System Y2 and the result is to repeat step 2. 3.1.i If the actor X3 then Y3 and the result is to repeat step 3 .
Results	4 R1
Post Condition	Not Relevant

Las reglas para convertir un caso de uso en un diagrama de actividades se enumeran a continuación.

1. El diagrama comienza en el paso 1 de la secuencia principal, el cuál debe ser realizado por un actor.

2. Cada uno de los pasos de la secuencia principal será una actividad. Las actividades siguen la misma secuencia de ejecución que el orden de los pasos.

3. Cada uno de los pasos de la secuencia alternativa será una decisión gobernada por la condición definida en el paso.

a. Si un paso de la secuencia alternativa es una precondición, debe evaluarse antes de ejecutar el paso, por tanto, el símbolo de decisión en el diagrama de actividades se dibuja antes de la actividad.

b. Si un paso de la secuencia alternativa es una invariante, el símbolo de decisión en el diagrama de actividades se dibuja después de la actividad.

c. Una secuencia de decisiones correlativas pueden unirse en una única decisión siempre que hagan referencia al mismo paso y sean del mismo tipo (invariante o precondición).

4. Si los elementos entonces de un paso de una secuencia alternativa implican la realización de alguna acción, dicha acción será una actividad.

5. El diagrama termina después de ejecutar los pasos marcados como resultado.

Aplicando estas reglas al caso de uso definido en la tabla 20, se ha obtenido el diagrama de actividades mostrado en la figura 17.8.1. Se puede observar como no es posible unir los dos nodos decisión consecutivos en uno solo.

El siguiente punto describe la aplicación que se desarrollará con el fin de implementar estas reglas.

8.3. Automatización de la generación automática.

Uno de los objetivos de la propuesta de definición de requisitos es que permita generar diagramas de actividades de forma automática. Estos diagramas de actividades permiten expresar la información relevante de un caso de uso de una manera formal y gestionable por herramientas software.

Por lo tanto hay que centrarse en la generación de actividades y decisiones de un diagrama de actividades a partir de la secuencia principal y alternativas de un requisito. De momento no se tendrá en cuenta la identificación y representación de la información implicada (variables operacionales).

En primer lugar, los casos de uso serán expresados mediante plantillas en XML. Esto facilita su tratamiento por parte de la herramienta generadora. Esta herramienta construirá un documento XML que contendrá un diagrama de actividades expresado siguiendo las reglas de la especificación XMI. Este documento podrá visualizarse y manipularse en cualquier aplicación con soporte XMI para visualizarlo de manera gráfica (por ejemplo ArgoUML y StarUML, ambas de libre descarga).

Para posibilitar el proceso, se van a expresar los casos de uso mediante plantillas UML. Después de una búsqueda preliminar solo se ha encontrado un formato de caso de uso para documentos XML en una sencilla y desconocida aplicación, por lo que creemos que no existe ninguna propuesta relevante. Por ello, se ha definido un modelo propio. Un ejemplo de este modelo se recoge en la tabla 23. Este modelo recoge el caso de uso de alta de un enlace definido en el caso práctico.

Tabla 24.8.3 Ejemplo de plantilla XML para casos de uso.

<useCase id="02-AddLink"> <mainSequence> <step id="1"> The user select the option for introduce a new link. </step> <step id="2"> The System select “top” category and shows the form to introduce the information of a link (SR-02). </step> <step id="3">

The User introduces information of the new link and press insert button.

</step>

<step id="4">

The System stores the new link and shows the main screen of the system.

</step>

</mainSequence>

<alternative>

<step id="2.1.i"

mainStep="2"

type="inv"

result="end">

If the user press cancel button then the use case ends.

</step>

<step id="2.2.i"

mainStep="2"

type="inv"

result="2">

If the user selects a different category (SR-01) then system changes the category and the result is to show the form again and execute step 2.

</step>

```
<step id="3.1.i"
   mainStep="3"
   type="inv"
   result="2">
If the link name or link url is empty the system shows an error message with the result of execute step 2.
</step>
</alternative>
</useCase>
```

La plantilla de la tabla es un prototipo. Por este motivo se han añadido atributos adicionales para simplificar la implementación de la herramienta de conversión. OMG define una especificación para definir modelos y elementos de UML mediante documentos en XML. Por cuestiones de simplicidad se ha optado por utilizar este propio modelo para diagramas de actividades.

8.4. El Prototipo.

Para demostrar que es posible manipular los casos de uso mediante herramientas software (y que es posible obtener un diagrama de actividades a partir de la tabla 22 con una herramienta), se implementará el prototipo para generar un diagrama de actividades a partir de una plantilla de caso de uso en XML mostrada en la tabla 22.

La idea final es que una vez generado el código del diagrama de actividades pueda ser visualizado por cualquier herramienta que permita importar diagramas en XMI.

Un problema detectado que no corresponde al sistema es que, a la hora de cargar los diagramas estos no aparecen dibujados, a pesar de que las actividades aparecen adecuadamente en el visor del diagrama.

La mayoría de los elementos de nuestro modelo de requisitos ya existen en otros modelos y trabajos. En este sentido las aportaciones originales son escasas. La principal es la clasificación de las excepciones y alternativas a una secuencia principal como precondiciones e invariantes.

Dado que la mayoría de propuestas utilizan formatos similares, la clave no está tanto en el formato de los casos de uso, sino en cómo definir sus elementos y en los requisitos complementarios. En general, las clasificaciones expuestas, son demasiado amplias. Por ejemplo, dentro del nivel de objetivos de usuario es posible redactar casos de uso muy diferentes en cuanto al nivel de detalle. Esto no ha sido abordado en profundidad en este trabajo, por lo que se propone para futuros nuevos trabajos.

8.4.1 Descripción de la Herramienta.

Es un prototipo de Generación de objetivos de prueba a partir de casos de uso. Los casos de uso son escritos en archivos XML. Primero, un diagrama de actividad (archivo XMI), es generado desde cada caso de uso. Luego, los objetivos de prueba son generados recorriendo el diagrama de actividad.

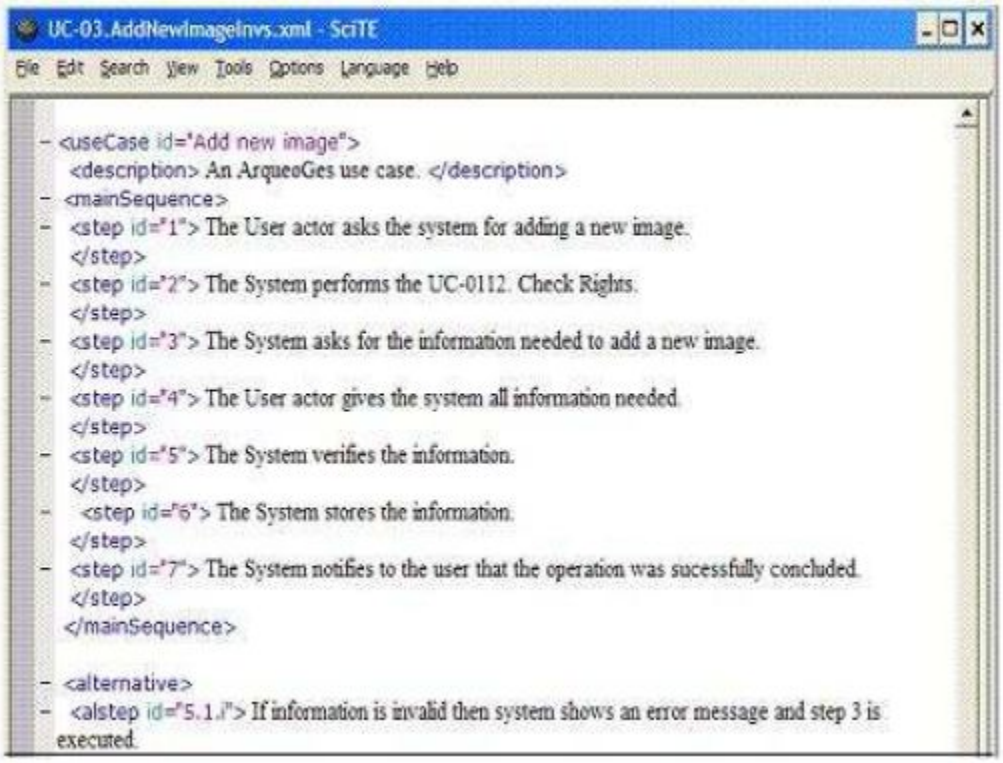


Figura 18.8.4.1 Plantilla de caso de uso en formato XML.

Aquí hay algunos ejemplos de casos de uso y diagrama de actividades generados con la Herramienta. Los archivos XMI han sido ilustrados usando la herramienta Open Source StarUml. También hay ejemplos de Objetivos generados.

- Se recibe un Archivo Xml como entrada:

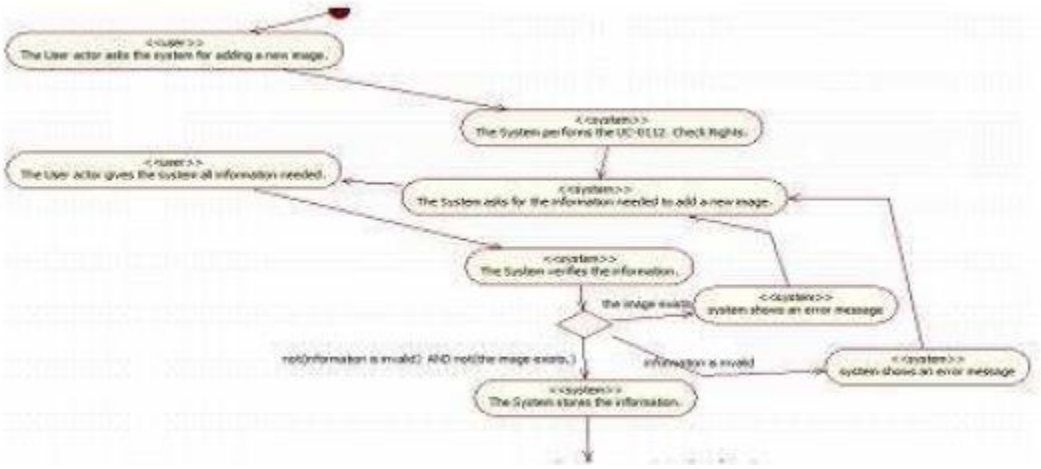


Figura 19.8.4.1 Modelo de comportamiento sistema (StarUml).

Luego se genera el diagrama de actividades en formato Xmi, que importado por la herramienta StarUml puede visualizarse como el diagrama de actividades de la figura anterior.

Finalmente la Herramienta genera el conjunto de objetivos de prueba a partir del caso de uso expresado en Xml. Ahi se pueden identificar los puntos críticos y lo que se debe probar a partir de los requerimientos del sistema.

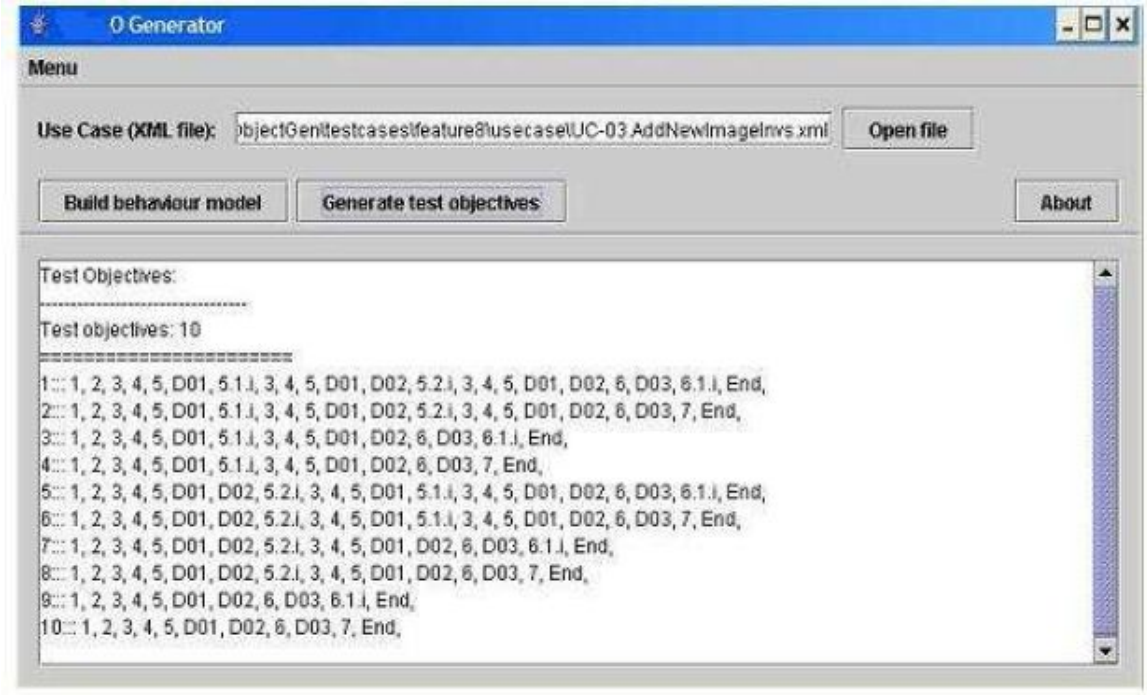


Figura 20.8.4.1 Generación Objetivos de Prueba de la Herramienta.
Para la implementación de esta herramienta se han utilizado las siguientes librerías Externas:

- Jakarta-Regexp : Paquete de Expresiones Regulares para Java.

- Log4j: Una herramienta muy util que se utiliza para crear logs de los eventos en una aplicacion Java. Esta librería trabaja con 3 componentes principales que permiten: Establecer prioridades, establecer el formato de los mensajes, establecer donde se escriben.

Es el responsable de dar un formato de presentación de los mensajes. Permite presentar el mensaje con el formato necesario para almacenarlo simplemente en un archivo de texto o uno XML (XMLLayout).

- Swing-layout: Librería para crear ventanas en la aplicación.

8.5. Generación de Objetivos de Prueba.

Después de la construcción del modelo de comportamiento, los objetivos de prueba se obtienen de manera sistemática. Como se ha mencionado, un objetivo de prueba es un conjunto de interacciones entre el sistema y un caso de prueba (que reemplaza a un actor) para verificar que el comportamiento del sistema es el comportamiento definido en sus casos de uso. Los objetivos de prueba se definen como caminos a través del modelo de comportamiento o como diagramas de actividades dónde sólo existe un único camino.

Es posible utilizar varios criterios de cobertura para generar secuencias a partir de un diagrama de actividades. Por ejemplo, dos criterios clásicos son los criterios de todos los nodos y de todas las transiciones. En este caso, el criterio de selección escogido es el criterio de todos los escenarios (AS).

Un conjunto de objetivos de prueba satisface el criterio AS si cada posible escenario del caso de uso es probado por un y solo un objetivo de prueba. Un escenario es una instancia de un caso de uso. El criterio AS asegura que todos los objetivos obtenidos son alcanzables y que ninguno de los objetivos está repetido. Dos objetivos de prueba son los mismos cuando tienen las mismas actividades en el mismo orden. El algoritmo “BuildTestObjectives” describe cómo generar los objetivos de prueba. Al igual que en el algoritmo “BuildBehaviourModel”, las funciones auxiliares tienen un nombre descriptivo y su definición ha sido omitida.

Tabla 25.8.5 Construcción del modelo de comportamiento.

```
algorithm BuildTestObjectives
var     objective : PATH
        objectives : LIST(PATH)
init
        objective = < empty >
        objectives = < empty >
        traverse(initialNode, path)
function traverse(in node, inout objective)
        if ( is_desicion(node) )
                traverse_desicion(node, objective)
                exit function
        end if
        objective.add(node)
        if ( isEnd(node) )
                objectives.add(objective)
                exit function
        end if
        nextNode = next_node(node)
        traverse(nextNode, objective)
end function
function traverse_desicion(in node, inout objective)
        foreach (alternative in node.alternatives)
                path.add(alternative)
                nextNode = next_node(alternative)
                traverse(nextNode, objective)
        end foreach
end function
```

Los objetivos de prueba obtenidos para el modelo de comportamiento del caso práctico se muestran en la tabla siguiente. El primer objetivo de prueba es el objetivo de prueba de la secuencia principal del caso de uso.

Tabla 26.8.5 Objetivos de Prueba.

Id	Caminos / Objetivos de prueba
1	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(Sin error & Resultados) -> 05
2	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(Sin error & Sin resultados) -> 05.1

3	01 -> 02 -> 03 -> D1(No) -> D2(No)-> 04 -> D3(Error) -> 04.2
4	01 -> 02 -> 03 -> D1(No) -> D2(Sí)-> 04.1 -> D3(Sin error & Resultados) -> 05
5	01 -> 02 -> 03 -> D1(No) -> D2(Sí)-> 04.1 -> D3(Sin error & Sin Resultados) -> 05.1
6	01 -> 02 -> 03 -> D1(No) -> D2(Sí)-> 04.1 -> D3(Error) -> 04.2
7	01 -> 02 -> 03 -> D1(Sí)

Como se ha mencionado, los objetivos de prueba también pueden expresarse como diagramas de actividades. La figura a muestra el diagrama de actividad correspondientes al camino 1 y la figura b muestra el diagrama de actividad correspondiente al camino 7 de la tabla 26.8.5.

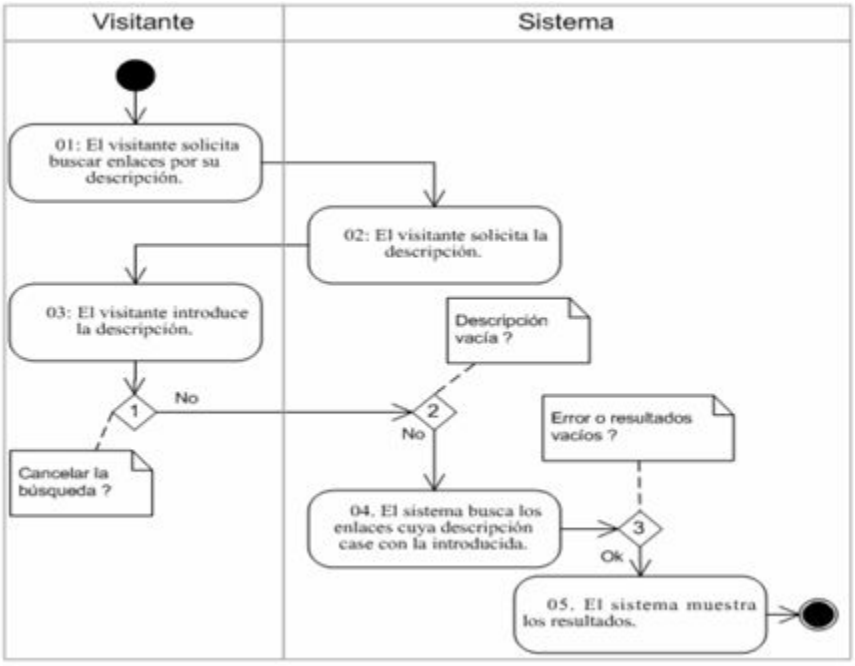


Figura 21.8.5 Diagrama Actividad camino 1 (ejemplo)

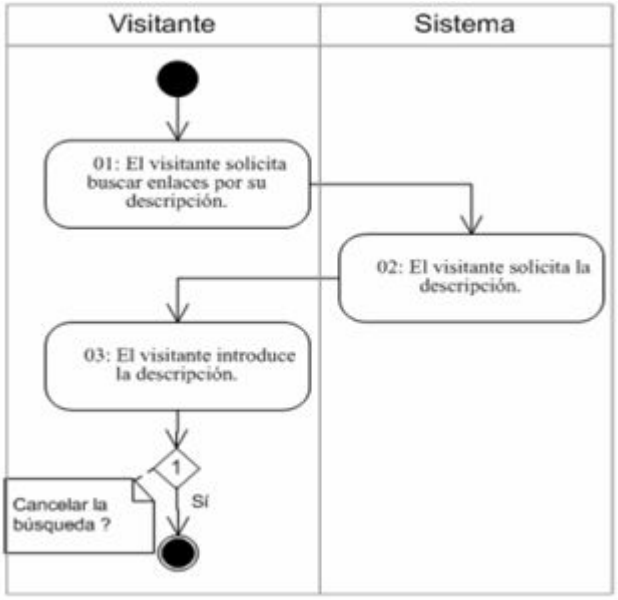


Figura 22.8.5 Diagrama Actividad camino 7 (ejemplo)

8.6. Cobertura de los Casos de Uso.

La cobertura de los objetivos de prueba mide el número de escenarios de un caso de uso que cuenta con un caso de prueba. Los objetivos de la tabla 13 cubren el 100% de los escenarios del caso de uso.

Objetivos de prueba = Cobertura.

Escenarios

Prioridad	Cobertura
0	No se genera ningún objetivo para el caso de uso.
1	Solo se genera un objetivo para la secuencia principal.
2	Se generan objetivos para la secuencia principal y todas las alternativas de los actores.
3	Se generan todos los objetivos posibles.

Figura 23.8.6 Medida de la cobertura de los objetivos de prueba.

La cobertura de los casos de uso por parte de los objetivos de prueba puede calcularse a partir de la fórmula mostrada anteriormente. Una mayor cobertura implica más objetivos de prueba y, por tanto más casos de prueba. Una cobertura de 1 (el 100%) significa que cualquier escenario posible tendrá un caso de prueba.

La cobertura y el número de objetivos de prueba generados pueden determinarse por la relevancia o la frecuencia del caso de uso. Ambos elementos están incluidos en la propuesta de plantilla de NDT y vista en la sección 5. Un criterio de cobertura basado en la prioridad del caso de uso se muestra en la figura anterior.

Conclusiones

Los requisitos funcionales expresados como casos de uso tienen un papel fundamental en el proceso de generación de pruebas del sistema. Los casos de uso permiten obtener los siguientes elementos relacionados directamente con las pruebas:

- Objetivos de prueba.
- Las acciones que debe realizar una prueba (pero no cómo realizarlas).
- La información que una prueba debe proporcionar al sistema (pero no cómo está definida ni valores concretos de prueba).
- Categorías en las que se clasifica dicha información (aunque pueden identificarse más categorías).
- Los resultados esperados (pero no cómo están definidos esos resultados ni valores concretos).

Además, los casos de uso también están relacionados con otros aspectos relevantes en un proceso de pruebas, como los que se enumera a continuación:

- Medida de la cobertura (aunque no en su totalidad, también hay que tener en cuenta las particiones).
- Determinación del orden de los casos de prueba.

Como se ha puesto de relieve en este documento, los casos de uso son muy importantes pero también necesitamos el concurso de otros tipos de requisitos, principalmente requisitos de información y requisitos de interfaz.Los requisitos de información definen la estructura de la información que maneja el sistema, así como sus restricciones. Estos requisitos nos permiten identificar particiones y generar valores concretos de prueba.

Se necesita conocer detalles de las interfaces del sistema para poder definir casos de prueba que puedan implementarse y ejecutarse. Algunos de los detalles que se necesita

conocer son: ¿Qué pantallas muestra el sistema cuando termina un caso de uso?, ¿cuál es el formato de la información que el sistema ofrece al actor o actores?, ¿qué elementos tiene el sistema para que los actores puedan trabajar con él?.

Se considera que estos detalles no deben incluirse en los casos de uso. Las plantillas de casos de uso deben abstraerse de detalles de implementación y de interfaces, tal y como recomiendan varios autores cuyas propuestas se han estudiado.

Algunos modelos de requisitos sí incluyen elementos relacionados, como el canal al actor principal y el canal al actor secundario. Una alternativa para poder gestionar la información necesaria son los requisitos de interfaz. Se han encontrado documentos para obtener interfaces a partir de los requisitos funcionales del sistema pero ninguna propuesta para definir requisitos de interfaz.

El modelo de requisitos de NDT [8] aborda satisfactoriamente los requisitos funcionales y los requisitos de información. Sin embargo sus requisitos de navegación no contienen toda la información necesaria para el proceso de prueba.

La mayoría de los elementos de este modelo de requisitos ya existen en otros modelos y trabajos. En este sentido las aportaciones originales son escasas. La principal es la clasificación de las excepciones y alternativas a una secuencia principal como precondiciones e invariantes. El modelo de requisitos de NDT [8] es adecuado para la definición de requisitos que puedan ser utilizados en un proceso de generación de casos de prueba. Las modificaciones a este modelo propuestas en esta sección son pocas y sencillas de realizar. El uso de patrones facilita la implementación de herramientas que manipulen de manera automática los casos de uso.

Dado que la mayoría de propuestas utilizan formatos similares, la clave no está tanto en el formato de los casos de uso, sino en cómo definir sus elementos y en los requisitos complementarios.

Los objetivos de prueba son básicos para realizar un proceso de pruebas con éxito. Los objetivos de prueba indican qué casos de prueba deben ser construidos para probar la implementación de los casos de uso. Este trabajo ha presentado una propuesta concreta y

completa para la generación sistemática de objetivos de prueba a partir de casos de uso y el diseño completo del prototipo final, ya implementado, que muestra toda la teoría expuesta. En la sección 3 se ha justificado la necesidad de esta nueva propuesta debido a que no se ha encontrado ningún otro trabajo anterior a esta propuesta que indique como derivar de manera automática objetivos de prueba a partir de casos de uso.

Esta propuesta soluciona esas carencias y genera los mismos objetivos de prueba que los trabajos citados en la sección 3. Sin embargo, los principales beneficios de este trabajo son la presentación de un modelo semiformal para la definición de casos de uso y la generación automática de objetivos de prueba. Todos los objetivos generados son alcanzables y únicos.

Referencias.

[1] Bertolino, A., Gnesi, S. 2004. PLUTO: A Test Methodology for Product Families. Lecture Notes in Computer Science. Springer-Verlag Heidelberg. 3014 / 2004. pp 181-197.

[2] Beck K., 2002. Test Driven Development. Addison Wesley.

[3] Binder, R.V. 1999. Testing Object-Oriented Systems. Addison Wesley.

[4] L. Briand, Y. Labiche. 2002. An UML Based Approach To System Testing. Journal of Software and Systems odelling, p 10-42.

[5] Burnstein, I. 2003. Practical software Testing. Springer Professional Computing. USA.

[6] Cockburn, A. 2000. Writing Effective Use Cases. Addison-Wesley 1st edition. USA.

[7] Denger, C. Medina M. 2003. Test Case Derived from Requirement Specifications. Fraunhofer IESE Report.

[8] Escalona M.J. 2004. Models and Techniques for the Specification and Analysis of Navigation in Software Systems. Ph. European Thesis. University of Seville. Seville, Spain.

[9] Escalona M.J. Gutiérrez J.J. Villadiego D. León A. Torres A.H. 2006. Practical Experiences in Web Enginering. 15th International Conference On Information Systems Development. Budapest, Hungary, 31 August – 2 September

[10] Gary Mogyorodi. 2001. Requirements-Based Testing: An Overview. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39). pp. 0286

[11] Glenford J. Myers. 1979. The art of software Testing. John Wiley & Sons. New York. USA.

[12] Gutierrez J.J. Escalona M.J. Mejías M. Torres J. 2004. Aplicando técnicas de testing en sistemas para la difusión Patrimonial. TURITEC'2004. pp. 237-252. Málaga, Spain.

[13] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. 2006. Generation of test cases from functional requirements. A survey. 4º Workshop on System Testing and Validation. Potsdam. Germany.

[14] Heumann , J. 2002. Generating Test Cases from Use Cases. Journal of Software Testing Professionals.

[15] Jacobs, Frits. 2004. Automatic generation of test cases from use cases. ICSTEST'04. Bilbao. Spain.

[16] Java, Varios. 2004. Java Language Specification Third Edition. Sun MicroSystems.

[17] Koch N. Zhang G. Escalona M. J. 2006. Model Transformations from Requirements to Web System Design. Webist 06. Portugal.

[18] Labiche Y., Briand, L.C. 2002. A UML-Based Approach to System Testing, Journal of Software and Systems Modelling (SoSyM) Vol. 1 No.1 pp. 10-42.

[19] Littlewood B. , ed., Software Reliability: Achievement and Assessment (Henley-on-Thames, England: Alfred Waller, Ltd., November 1987).

[20] McCabe, Thomas, A Software Complexity Measure. IEEE Transactions on Software Engineering, vol.2, nro 6, December 1976.

[21] Myers G. 2004. The art of software testing. Second edition. Addison-Wesley. USA.

[22] Nebut C. Fleury F. Le Traon Y. Jézéquel J. M. 2006. Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering Vol. 32. 3. March.

[23] Object Management Group. 2003. The UML Testing Profile. www.omg.org

[24] Object Management Group. 2003. Unified Modelling Language 2.0.

[25] Offutt, J. et-al. 2003. Generating Test Data from Sate-based Specifications. Software Testing, Verification and Reliability. 13, 25-53. USA.

[26] Piattini Velthuis Mario G. 2003. Análisis y diseño de aplicaciones informáticas de gestión. Una perspectiva de ingeniería del software. Editorial Ra-Ma.

[27] Robinson, Harry. 2000. Intelligent Test Automation. Software Testing & Quality Engineering. Pp 24-32.

[28] Ruder A. 2004. UML-based Test Generation and Execution. Rückblick Meeting. Berlin.

[29] 2003. OMG Unified Modelling Language Specification 1.5. http://www.omg.org

Trabajos Futuros.

El camino hacia la automatización total aún es largo. Como se puede ver en el caso

práctico, para la generación automática es necesario tener datos específicos adicionales aún, un ejemplo pueden ser los de la interfaz. Será necesario no sólo tener esos datos, sino definirlos de una manera procesable automáticamente y enriquecerlos con una semántica para que el sistema sepa lo que son.

Por lo tanto un camino muy válido para continuar esta línea del proceso de pruebas, sería enfocar los futuros trabajos en la convención de nombres, desde los requisitos de almacenamiento hasta la implementación, con la idea de que al igual como se logró con los casos de uso, establecer ciertas plantillas que sirvan para poder aplicar generación automática al proceso completo de pruebas.