Introducción al Testing de Software

1. Qué es el testing *4*
1.1 Roles en el proceso de Desarrollo de Software 4
1.2 Concepto de testing 4
2. Fundamentos del testing *5*
2.1. Conceptos de error, defecto (bug) y Fallo. 5
2.2. Concepto de Requisito. 5
2.3. Concepto de calidad 5
2.4. ¿Por qué son necesarias las pruebas? 6
2.5. Objetivos de las pruebas. 6
2.6. ¿Cuánto testing es necesario? 6
3. Proceso fundamental del testing *8*
3.1. Planificación y control 8
3.2. Análisis y diseño 9
3.3. Implementación y Ejecución 9
3.4. Evaluación de criterio de salida y reportes 10
3.5. Actividades de Cierre - Test closure 10
4. Modelo de desarrollo de software *11*
4.1. Modelos Iterativos 11
5. Diseño de pruebas *13*
5.1. ¿Qué es un caso de prueba? 13
5.2. Niveles de prueba 13
5.3. Tipos de prueba 16
6. Técnicas Estáticas *18*
6.1. Revisión 18
6.2. Análisis estático con herramientas 20
7. Técnicas Dinámicas *21*
7.1. Técnicas de Caja Blanca 21
7.2. Caja Negra 24
7.3. Basados en Experiencias 30
7.4. Cómo seleccionar la técnica correcta. 31
8. Creación de casos de prueba *32*
8.1. Cómo crear casos de prueba 32
8.2. Ciclo de vida de un defecto. 33

8.3. Cómo reportar un defecto 34
8.4. Severidad y prioridad.....35

1. Qué es el testing

1.1 Roles en el proceso de Desarrollo de Software

El desarrollo de software no es una actividad simple, se realiza en equipos: dos personas o más que trabajan juntas para lograr un objetivo común, donde a cada una de estas ellas se le asignan ciertas funciones específicas para lograr ese fin.

Los principales roles del equipo de desarrollo de software son:

Project Manager / Líder de proyecto / Administrador del Proyecto: coordina el equipo y asegura que todos los demás roles cumplan con su trabajo.

Analista Funcional: genera los requerimientos para el sistema para definir la estructura básica del mismo.

Diseñador: es quien en base a estos requerimientos genera el diseño del sistema y sus prototipos.

Desarrollador: es el que toma los requerimientos y prototipos y los traduce a código ejecutable.

Tester/Probador: es el que asegura la calidad del producto.

1.2 Concepto de testing

Existen muchas definiciones de Testing de Software. Pero básicamente, las pruebas de software (Testing en inglés) es el proceso que permite verificar la calidad de un producto de software. Permite identificar posibles fallos en la implementación, calidad o usabilidad de un programa. No solo significa ejecutar pruebas sino que incluye las actividades de planificación, preparación y evaluación de productos de software para determinar si cumple con los requerimientos especificados.

"El testing puede probar la presencia de errores, pero no la ausencia de ellos". Edsger Wybe Dijkstra

2. Fundamentos del testing

2.1. Conceptos de error, defecto (bug) y Fallo.

Para realizar un buen testing de software tenemos que tener en cuenta varios conceptos involucrados y la diferencia entre ellos.

Error ("Error"): Acción humana que produce un resultado incorrecto. Error es generado por una persona. Un error nos lleva a generar uno o más defecto. Ejemplo: error de programación, error de tipeo, requerimiento mal especificado.

Defecto ("Defect"): Desperfecto o error interno que se encuentra en un componente o sistema que puede causar que el componente o sistema falle en sus funciones. Un defecto causará un Fallo. Ejemplo: defecto es haber utilizado el operador "<" en vez de "<=".

Fallo ("Failure"): Manifestación **física** o **visible** de un defecto. Si un defecto es encontrado durante la ejecución de una aplicación puede producir un fallo. Es decir, una falla es un síntoma de un defecto. Ejemplo: crash del sistema.

Las causas de fallos se deben a 2 grandes grupos:

- Error humano: el defecto se introdujo en el código de software, datos o en los parámetros de configuración.

- Condiciones Ambientales: Cambios en las condiciones ambientales (por ejemplo: fallo en discos duros).

2.2. Concepto de Requisito.

Un requisito describe un atributo funcional deseado u obligatorio.

2.3. Concepto de calidad

La Calidad es el grado en el cual un componente, sistema o proceso satisface requisitos especificados y/o necesidades y expectativas del cliente. Es la suma de los atributos que hablan de la capacidad del software de satisfacer un conjunto de requisitos dados

2.3.1. Atributos de la calidad

Los atributos mencionados arriba, pueden ser categorizados en 2 grupos: Funcionales y NO Funcionales.

Dentro de los funcionales:
- Correctitud: Se satisfacen correctamente los atributos requeridos.

- Completitud: Se satisfacen todos los requisitos.

Dentro de los No Funcionales:
- Fiabilidad: Se mantiene la capacidad y funcionalidad del sistema a lo largo de un periodo de tiempo.

- Usabilidad: fácil de usar, fácil de aprender, uso intuitivo.

- Portabilidad: fácil de instalar y desinstalar y configurar parámetros.

- Eficiencia: El sistema necesita solo la utilización de un mínimo de recursos para ejecutar una tarea determinada.

- Mantenibilidad: Esfuerzo necesario para realizar cambios en los componentes de un sistema.

Dependiendo del sistema, habrá que priorizar los atributos que necesitemos utilizar, y en base a esto determinaremos qué tipo de testing se aplicará.

2.4. ¿Por qué son necesarias las pruebas?

Hay muchas razones por las cuales un software debe ser probado, pero las más importantes son:

1. El software es realizado por seres humanos que comenten errores.
2. A medida que se incrementa la presión en las entregas, no hay tiempo para chequear y se comienza a asumir, con lo cual se cometen errores.
3. Para medir la estabilidad del Software.
4. Para demostrar que el software no tiene fallas y asegurarnos que hace lo que debe hacer.
5. Porque las fallas pueden ser costosas: Es más barato y rápido encontrar las fallas antes de poner el sistema en producción.

2.5. Objetivos de las pruebas.

Los objetivos más importantes son los siguientes:

- Adquirir conocimiento: Realizamos pruebas para conocer los defectos que tiene un objeto de prueba y describirlos de tal forma que facilite su corrección.

- Confirmación de la funcionalidad: Con las pruebas verificamos que la funcionalidad del sistema ha sido implementada como se ha especificado.

- Generación de Información: Proporcionamos información sobre los posibles riesgos relativos al sistema antes de la entrega al usuario final.

- Confianza: Ganamos confianza ya que sabemos que el sistema cumple con la funcionalidad esperada.

2.6. ¿Cuánto testing es necesario?

Probar todas las combinaciones posibles es algo casi imposible. Para determinar cuándo terminar las pruebas, existen los riesgos y las prioridades. Es bastante subjetivo y depende de cada proyecto.

- No encontrar más defectos puede ser un criterio para terminar con las actividades de pruebas. Sin embargo, cada prueba debe contar con criterios de salida (“exit criteria”). Al alcanzar estos criterios de salida concluirán las actividades de prueba.

- Pruebas basadas en riesgos: El nivel de riesgo determinará el grado en el cual se harán las pruebas.

- Pruebas basadas en plazos y presupuesto: La disponibilidad de recursos de personal, tiempo, presupuesto.

3. Proceso fundamental del testing

La ejecución de pruebas es sólo **una** parte del proceso de las pruebas. Dependiendo del enfoque seleccionado el proceso de pruebas se realizará en diferentes puntos del proceso de desarrollo. Es decir, además del proceso de desarrollo, tenemos un proceso para el testing.

Si bien tenemos diferentes fases en este proceso, éstas fases pueden superponerse y la fase de control se realiza en todas las fases para poder saber en cada momento en qué estado nos encontramos.

3.1. Planificación y control

El Control de las Pruebas es una actividad continua que influye en la planificación de las pruebas. El plan de pruebas maestro ("master test plan") será modificado de acuerdo a la información adquirida a partir del control de prueba

Para saber el estado del proceso de pruebas comparamos el progreso logrado con respecto al plan de pruebas. Se inician medidas correctivas y esto permite tomar decisiones.

Tareas del Control de Pruebas:

- Medir y analizar los resultados de las pruebas: De esta manera tendremos conocimiento de la cantidad, el tipo y la importancia de los defectos que hemos encontrados.

- Monitorear y documentar el progreso: Esta tarea nos sirve para determinar cuántas pruebas se completaron, cuáles fueron los resultados, y qué riesgos se encontraron para evaluar.

- Brindar información de las pruebas a personas interesadas: Estos reportes les sirven a las personas interesadas (stakeholders) para saber el estado de las pruebas y así poder tomar las acciones necesarias.

- Iniciar acciones correctivas dependiendo de lo que se necesite corregir: Dependiendo de lo que necesitemos corregir, es posible ajustar los criterios de salida para los defectos detectados, o priorizar defectos bloqueantes.

- Tomar decisiones que definimos si seguimos o paramos las pruebas .

Tareas de la Planificación de Pruebas:

-Determinar el alcance y riesgos: Nos preguntamos si vamos a probar un software completo, un componente, o algún otro producto.

-Identificar los objetivos de las pruebas y los criterios de salida de pruebas: Qué es lo más importante en este caso? Prevenir defectos, verificar que el software cumple con los requerimientos, medir la calidad?

-Determinar el enfoque: Todas estas preguntas nos permiten definir el enfoque de nuestras pruebas. Cómo ejecutaremos las pruebas, qué técnicas usaremos, qué se probará y cuán extensamente, es decir la cobertura de pruebas, quienes van a participar en el equipo de pruebas y durante cuánto tiempo?

-Implementar la estrategia de pruebas, y planificación del período de tiempo para el desarrollo de las actividades a seguir.

-Adquirir recursos necesarios para las pruebas como personas, computadoras, software, entorno de pruebas, presupuesto de pruebas, fechas.

-Selección de condiciones de entrada y de salida: Una condición de entrada será, por ejemplo, que el entorno de pruebas está listo y estable, que las herramientas necesarias están instaladas, el equipo de pruebas está completo. Los criterios de salida pueden ser: que todos los casos de pruebas diseñados sean ejecutados, que se hayan ejecutado pruebas de regresión, que no existan defectos bloqueantes, que los defectos de criticidad mediana no superen un determinado número. Los criterios de salida son condiciones acordadas con la gente involucrada, para que el proceso sea considerado formalmente concluido.

Lo que se concluya en estas tareas se plasma En documentos que crearemos también en esta fase y que usaremos en todo el proceso. Los documentos son los siguientes:

El documento de Plan de pruebas describe alcance, enfoque, recursos y calendario de las actividades de pruebas previstas.

El documento de Estrategias de prueba describe a alto nivel los niveles de prueba a realizar.

El documento de Enfoque de pruebas incluye análisis de riesgo, técnicas de diseño de pruebas a aplicar, criterios de salida y tipos de prueba a ejecutar.

3.2. Análisis y diseño

Las tareas más importantes:

- Revisión de elementos: Se revisan los elementos básicos para las pruebas (requerimientos, arquitectura, especificaciones de diseño, interfaces). Utilizamos estas bases para poder comenzar con el diseño de pruebas.

- Testeabilidad: Evaluación de si los elementos básicos del punto anterior pueden generar casos de prueba.

- Condiciones de Pruebas: Identificación y revisión de las condiciones de las pruebas basándonos en los puntos anteriores: Esto nos brinda una lista a alto nivel de lo que nos interesa probar.

- Diseño de Casos: tanto los positivos (que dan muestra de la funcionalidad en sí) como los negativos (comprueban situaciones en las que hay tratamiento de errores).

- Entorno: Esta actividad trata sobre la puesta a punto del entorno de pruebas: disponibilidad del entorno, administración de usuarios, carga de datos.

- Herramientas: Seleccionar, proveer e instalar las herramientas de pruebas, procesos, procedimientos y responsabilidades.

3.3. Implementación y Ejecución

En esta etapa realizamos la ejecución de las pruebas, manual o mediante el uso de alguna herramienta:

- Se finalizan, implementan y priorizan los casos de prueba y los procedimientos de prueba.

- Trazabilidad: Se verifica que se pueda realizar la trazabilidad entre los elementos básicos de prueba y los casos de prueba.

- Ejecución: Se ejecutan los casos de manera automática o manual.

- Registro de los resultados: Una vez que hemos ejecutado, ya sea manual o automáticamente, los casos de pruebas, entonces podemos registrar los resultados obtenidos, esto incluye: identidades y versiones del software, las herramientas de pruebas utilizados, los productos de soporte de pruebas y también incluye comparar los resultados reales con los resultados que eran esperados.

- Comparación: Se comparan los resultados obtenidos con los resultados esperados.

- Informe: Se informan discrepancias en un defecto para que sea arreglado.

- Repetición: Se repiten las pruebas para confirmar que el defecto ha sido corregido (re testing). También incluye realizar pruebas de regresión para asegurarnos que los cambios nos han introducido nuevos defectos (testing de regresión)

3.4. Evaluación de criterio de salida y reportes

Los resultados de las pruebas son evaluados contra los objetivos definidos. Tareas:

- Comparación: Comparar los registros de prueba contra los criterios de salida especificados en la fase de planificación de prueba. Es decir evaluamos la evidencia que se tiene de las pruebas que se ejecutaron, fallas reportadas, solucionadas o pendientes para así confirmar si se completaron los criterios de salida que dan fin a las pruebas.

- Evaluación: Con resultados de la tarea anterior, se evalúa si se necesitan ejecutar más pruebas o si el criterio de salida debe modificarse. Es decir, aquí definimos si ejecutamos más pruebas en caso de que no se hayan ejecutad todos los casos de prueba diseñados; definimos si continuamos con la ejecución de las pruebas si es que no se alcanzó la cobertura deseada; o definimos que necesitamos diseñar mas casos de prueba en caso de que nuevos riesgos se han descubierto.

- Reporte: Lo que hacemos aquí es preparar un resumen de las pruebas ejecutadas para que las partes interesadas (stakeholders) estén al tanto de nuestro trabajo y de nuestro avance.

3.5. Actividades de Cierre - Test closure

Tareas:

- Recolección: Recolectar información sobre las pruebas completadas.

- Verificación: Verificamos que las pruebas y la documentación acordadas hayan sido entregadas al cliente según lo definido en el plan de pruebas.

- Documentación: Se deben documentar todos los casos de pruebas corridos, todos los resultados, todos los bugs que se hayan encontrados, vamos a documentar hasta la aceptación del sistema.

- Analizar lecciones aprendidas para futuros proyectos. Entre ellos, mejoras a los procesos del ciclo de desarrollo de software. También verificaremos dónde hubo problemas y el número de fallas para realizar mejoras de diseño, ejecución y revisión de las pruebas.

4. Modelo de desarrollo de software.

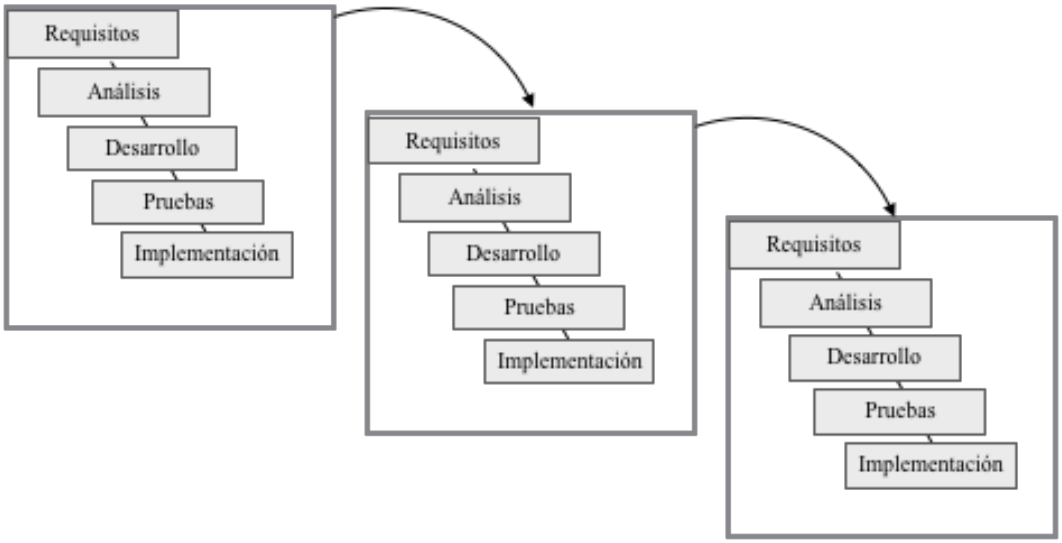
Los modelos de desarrollo software son utilizados para el **desarrollo de software** incluyendo las actividades del proceso de pruebas. Las pruebas no existen de manera aislada, están relacionadas con las actividades de desarrollo de software. Los diferentes modelos de ciclo de vida de desarrollo necesitan diferentes enfoques hacia la prueba. El más utilizado hoy es el siguiente:

4.1. Modelos Iterativos

Un modelo Iterativo busca reducir el riesgo que existe entre lo que el usuario necesita y el producto final.

ITERACION: es una secuencia de actividades dentro de un plan, que se organiza con el objetivo de entregar parte de funcionalidad del producto. Es decir, en la primera iteración, le entrego al cliente parte de la funcionalidad, en la segunda iteración le entrego una mejora (de la misma funcionalidad o más funcionalidad).

Cada iteración contribuye con una **característica adicional** del sistema a desarrollar. Cada una de las iteraciones puede ser probada por separado. En cada iteración, la verificación (relación con el nivel precedente) y la validación (grado de corrección del producto dentro del nivel actual) se pueden efectuar por separado.



Algunos modelos iterativos:

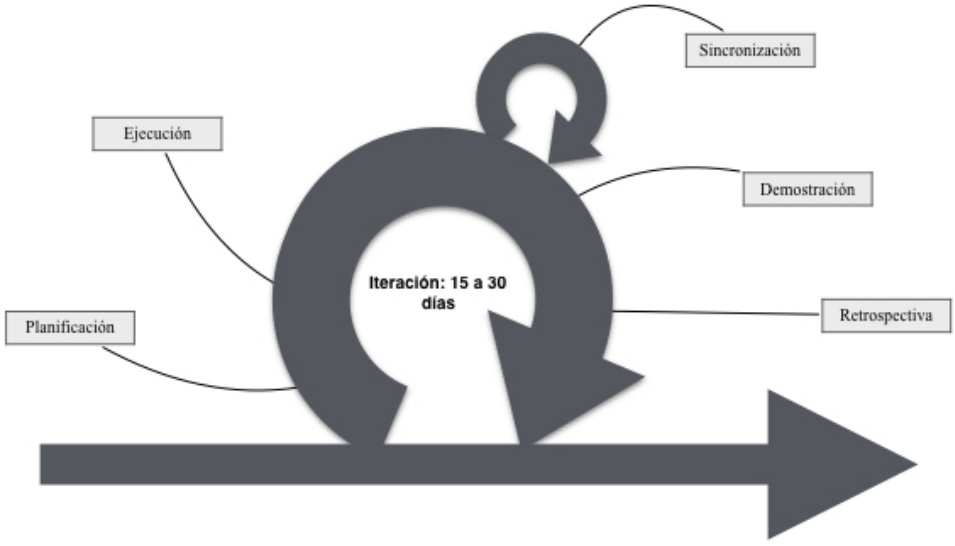
Proceso unificado (“Rational Unified Process” - (RUP))

Programación extrema (“Extreme Programming” - (XP))

SCRUM: es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para **trabajar en equipo**, y obtener el mejor resultado posible de un proyecto.

En esta metodología ágil se realizan entregas parciales del producto final. Está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos.

Proceso:



Un proyecto se ejecuta en iteraciones de aproximadamente de 15 días a 1 mes de duración. Cada una de ellas tiene que proporcionar un resultado completo, un incremento de producto final.

Empezamos con la lista de requerimientos priorizada o Product Backlog (el cliente es quien va definiendo la priorización de acuerdo a lo q aportarían)

Sprint Planning: Es la reunión que se realiza el primer día de la iteración, la cual cuenta con 2 partes importantes: la selección de requisitos y la planificación propiamente dicha.

Sprint (ejecución): A partir de este momento, ya se comienza a trabajar. Todos los días, todo el equipo se toma 15 minutos para realizar entre todos una **reunión de sincronización**

Esta reunión, o scrum meeting se realiza para que cada miembro del equipo sepa en que están trabajando los demás miembros. Cada uno, responderá a 3 preguntas: Qué he hecho desde la última reunión? Qué voy a hacer a partir de este momento? Qué impedimentos tengo o voy a tener?

Sprint Review (Demostración): El último día de la iteración, el equipo le presenta al cliente los requisitos completados. Aquí el cliente le puede exponer cambios o no.

Sprint Retrospective (Retrospectiva): El equipo analiza cómo fue su trabajo y cuáles son los problemas que podrían impedirle progresar, para así realizar los cambios necesarios.

5. Diseño de pruebas

5.1. ¿Qué es un caso de prueba?

Son un conjunto de condiciones y variables bajo las cuales el analista determinará si el requisito de una aplicación es parcial o completamente satisfactorio.

Hay que tener en cuenta casos de prueba positivos y negativos. Los **casos de prueba positivos** muestran de la funcionalidad, y los **casos de prueba negativos** comprueban situaciones en las que hay tratamiento de errores.

Ejemplo: tengo un requisito que dice que el precio de una producto debe ser mayor que 0.

- casos de prueba positivos: precio igual a 0, precio igual a 10 , precio igual a 15

- caso de prueba negativo: precio igual a -5. En este caso, el sistema tiene que poder determinar que es una situación de error y enviar el error a la aplicación.

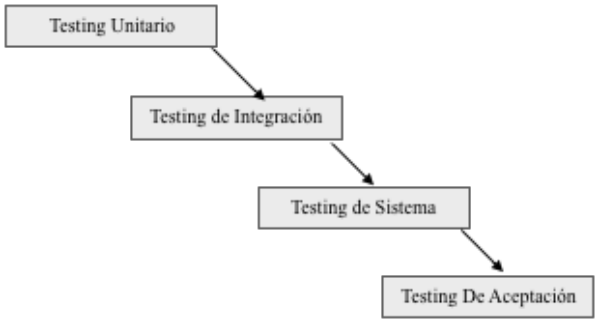
Se pueden hacer varios casos de prueba para determinar si un requisito es satisfactorio, pero debe haber al menos un caso de prueba por requisito.

La característica más importante de un caso de prueba es que hay una entrada conocida y una salida esperada. En el ejemplo anterior, tenemos que si la entrada es “10” , entonces la salida esperada es que permita continuar con la aplicación. En el caso de prueba donde el precio igual a “-5”, la entrada es el -5 y la salida esperada es un error.

Los casos de prueba deben ser trazables con los requisitos, si cambia un requisito, debe ser fácil determinar qué casos de prueba modificar. A esto se le llama TRAZABILIDAD.

5.2. Niveles de prueba

Dependiendo de en qué momento del proceso de desarrollo son ejecutadas las pruebas, tenemos diferentes niveles:



5.2.1. Pruebas de componentes (unitarias)

Algunas de sus características:

- Prueba luego de su construcción: El desarrollador termina de construir el componerte, y es allí cuando se realizan las pruebas unitarias. Los componentes también pueden ser mencionados como módulos, clases o unidades dependiendo del lenguaje que se haya usado para el desarrollo del mismo. Es decir, que las pruebas unitarias nos permiten asegurar que los pequeños fragmentos de código funcionan bien de manera aislada.

- Incrementa la confianza: En el mantenimiento del código. Significa que si las pruebas son correctas y se corren con cada cambio que se realiza en el código, entonces tenemos más posibilidades de que encontremos los posibles errores en este nivel y no es un nivel más avanzado.

- Realizado por desarrolladores: Las pruebas unitarias normalmente son realizadas por los mismos desarrolladores, peros pueden ser ejecutadas por el equipo de pruebas también.

Los **casos de prueba** podrán ser obtenidos a partir de especificaciones de componente, diseño software y el modelo de datos, y se obtendrán mediante Métodos de Caja Blanca.

* Necesitamos al menos un caso de prueba que corrobore que la funcionalidad se realiza correctamente. ¿Se cumplen las especificaciones?

* ¿Tenemos resistencia a datos de entrada inválidos? ¿Qué sucede si tenemos entradas inválidas? Un defecto que se encuentra normalmente en este tipo de pruebas son los de procesamiento de datos, normalmente en el entorno de los **valores límite** (“boundary values”: si el precio de un producto tiene que ser mayor que cero, ¿qué pasa cuando es igual a cero?).

5.2.2. Pruebas de integración (interfaz)

Características:

- Prueba de interacción entre componentes: Es la prueba que comprueba la interacción entre elementos de software (los componentes) tras la integración del sistema. **Integración** significa construir grupos de componentes, pero también puede ser la Integración de sub-sistemas.

- Realizada luego del Testing Unitario: Cada componente ya ha sido probado en lo referente a su **funcionalidad interna** (prueba de componente). Las pruebas de integración comprueban las **funciones externas** tras las pruebas de componente. La pregunta en este momento sería: ¿Cómo interactúan estos componentes entre sí?

- Pueden ser realizadas por los desarrolladores o por testers: Los casos de prueba podrán ser obtenidos a partir de especificación de interfaces, diseño de la arquitectura (diseño), el modelo de datos.

- Se utilizan métodos de Caja Blanca o Caja Negra.

El propósito de las pruebas de integración es detectar defectos en las interfaces. Entre los defectos más comunes que encontramos en este tipo de nivel de prueba puedes ser:

- Pérdida de datos.

- Manipulación errónea de datos.

- Los componentes interpretan datos de entrada de manera diferente. Pueden ser que los componentes en sí funcionen bien, pero al interactuar con otros componentes esa interacción no sea la que se espera.

Hay varias estrategias para abordar este nivel de testing:

- Estrategia ascendente

- Estrategia descendente

- Bing bang: Significa que todos los test unitarios se combinan y se prueban.

5.2.3. Pruebas de sistemas

Características:

- Prueba de comportamiento de todo el sistema: Se refiere al comportamiento de todo el sistema/producto, definido en el alcance del proyecto.

- Prueba de requisitos funcionales y no funcionales: Las pruebas de sistema se refieren a Requisitos **funcionales** y **no funcionales**. Los casos de pruebas podrán ser obtenidos desde especificación de requisitos, procesos de negocio, casos de uso, evaluación de riesgos

- Punto de vista del usuario: Aquí probaremos el sistema integrado desde el punto de vista del usuario, que se hayan implementado completa y correctamente los requisitos. Por eso es importante que el entorno en donde estemos probando sea muy similar sino coincida totalmente con el entorno real.

- Método de Caja Negra: ¿Qué métodos vamos a utilizar en este nivel de prueba? Los métodos de Caja Negra

5.2.4. De aceptación

Características:

- Prueba de verificación formal: Pruebas realizadas para verificar de manera **formal** la conformidad del sistema con los requisitos. El objetivo aquí es validar que el sistema cumple con el funcionamiento esperado. Se determina si se cumplen los requisitos contractuales, que está listo para puesta en producción.

- Tipos: Diferentes tipos de prueba:

Testing de Aceptación Interno (*Internal Acceptance Testing - Alpha Testing*): es realizado por miembros de la organización donde se desarrollo el software.

Testing de Aceptación Externo (*External Acceptance Testing*): es realizado por gente externa a la organización donde se desarrollo el software. Puede ser el cliente (*Customer Acceptance Testing*) o los usuarios finales del sistema (*User Acceptance Testing - Beta Testing*)

¿Qué método usamos para este tipo de pruebas? Métodos de Caja Negra.

5.3. Tipos de prueba

Tipos de pruebas que se aplicarán durante los distintos niveles de prueba.

5.3.1. Funcional

Es un tipo de prueba que se puede realizar en todos los niveles de prueba. El objetivo de este tipo es que el objeto de prueba realmente funcione.

Las pruebas permiten verificar los requisitos funcionales (establecidos en las especificaciones, conceptos, casos de estudio, reglas de negocio o documentos relacionados).

En este tipo de pruebas se prueban 5 características:

Adecuación: ¿Las funciones implementadas son adecuadas para su uso?

Exactitud: ¿Las funciones presentan los resultados correctos?

Interoperabilidad: ¿Las interacciones con el entorno del sistema presentan algún problema?

Cumplimiento de funcionalidad: ¿El sistema cumple con normas y reglamentos aplicables?

Seguridad: ¿Están protegidos los datos/programas contra acceso no deseado o pérdida?

5.3.2. No funcional

El objetivo aquí es saber **cómo** funciona el sistema. En la mayor parte de los proyectos, están especificados de manera escasa. Este tipo de pruebas también se puede llevar a cabo en todos los niveles de pruebas. Se van a describir las pruebas necesarias para medir las características que se puedan cuantificar en un escala, por ejemplo, el tiempo de respuesta para las pruebas de rendimiento.

- Pruebas de Volumen: Procesamiento de grandes cantidades de datos. ¿Qué pasa cuando el sistema maneja grandes cantidades de datos?

- Pruebas de Estabilidad: Rendimiento en “modo de operación continua”.

- Pruebas de Robustez: Reacción a entradas erróneas.

- Pruebas de estrés: Reacción a la sobrecarga / recuperación tras el retorno a una carga normal.

- Pruebas de usabilidad: Para determinar que el sistema es estructurado, comprensible, fácil de aprender.

- Pruebas de Rendimiento o performance: Rapidez con la cual un sistema ejecuta una determinada función. Si para loguearme necesito 30 minutos, entonces el rendimiento no es bueno.

- Pruebas de Carga: Cómo reacciona el sistema bajo carga. Por ejemplo: si tiene muchos usuarios o muchas transacciones, ¿cómo mi sistema reacciona ante esto?

Respecto a los requisitos no funcionales, son difíciles de lograr conformidad ya que normalmente no están bien definidos. Por ejemplo, el cliente nos puede decir, “mi sistema tiene que ser fácil de operar, tiene que

tener una interfaz de usuario bien estructurada", pero cómo logramos eso? ¿qué es fácil de operar para el usuario?

5.3.3. Estructural

La finalidad de las pruebas es medir el grado en el cual la estructura del objeto de prueba ha sido cubierto por los casos de prueba.

Las pruebas estructurales (o de caja blanca) pueden ser realizadas en todos los niveles de pruebas, pero sobre todo en las pruebas de componente (unit test) y de integración.

5.3.4. Relacionado a cambios

El objetivo aquí es **repetir** una prueba de funcionalidad que ha sido verificada previamente. Se realiza en dos diferentes momentos:

- Confirmación: cuando un defecto es detectado y arreglado, entonces el software es probado de nuevo, para comprobar que el defecto original ha sido corregido con éxito.

- Regresión: son las pruebas repetidas de un programa ya probado, después de una modificación, para descubrir cualquier error introducido.

6. Técnicas Estáticas

<table>
<tr><td rowspan="4">QA</td><td rowspan="3">Dinámico</td><td>Caja negra</td><td>Partición de equivalencia
Análisis de valores límite
Pruebas de transición de estado
Tablas de decisión
Pruebas de casos de uso</td></tr>
<tr><td colspan="2">Técnicas basadas en la experiencia</td></tr>
<tr><td>Caja blanca</td><td>Cobertura de sentencia
Cobertura de rama
Cobertura de condición
Cobertura de camino
Entorno de desarrollo integrado (IDE)</td></tr>
<tr><td>Estático</td><td colspan="2">Revisiones/revisiones guiadas
Análisis del flujo de control
Análisis del flujo de datos
Métricas del compilador/analizador</td></tr>
</table>

Las técnicas dinámicas, las hemos mencionado en los NIVELES DE TESTING, hemos dicho que tal o cual nivel de testing puede utilizar técnicas de caja negra o técnicas de caja blanca.

Las técnicas estáticas, se basan en el examen manual o en el análisis automatizado del código, o pueden basarse también en cualquier otra documentación del proyecto en los que no tengamos que ejecutar el código propiamente dicho.

Las técnicas estáticas, suelen realizarse antes de las técnicas dinámicas. Porque los defectos que encontremos van a ser al principio del ciclo de vida del producto, por lo cual van a ser menos costosos de corregir que los detectados durante la ejecución de las pruebas dinámicas.

Ambas técnicas son complementarias ya que detectan diferentes tipos de errores. El test estático encuentra **defectos** mientras que el test dinámico encuentra **fallas** o desviaciones en el resultado esperado.

Las técnicas estáticas pueden realizarse antes de las dinámicas para encontrar defectos antes que se conviertan en fallas.

6.1. Revisión

Revisión es la evaluación de un producto para detectar discrepancias respecto de los resultados planificados y para recomendar mejoras.

El análisis estático consiste en analizar un determinado objeto de prueba (script, código fuente, requisito) sin ejecutarlo. Aquí se evalúan los estándares de programación, flujos de datos.

Las revisiones pueden ser formales o informales, y permiten descubrir defectos en las especificaciones, en el diseño, especificaciones de interfaces.

En una revisión Formal tendremos ciertas actividades por completar. Es decir, cada vez que mencione que una revisión es formal, sabremos que cuenta con estas actividades:

-Planificación: Durante la planificación definimos los criterios que se van a utilizar (el tipo de revisión que aplicaremos, la lista de comprobación, definimos quienes participan, roles y tiempos).

-Definición de Criterios de entrada y salida: o sea, qué estaremos revisando.

-Kick-Off: Esto es una reunión en donde iniciamos el proceso, distribuimos los documentos, donde les decimos a cada participante lo que debe hacer.

-Identificación de defectos, preguntas y/o comentarios. La persona encargada va a identificar los defectos sobre los documentos, va a armar preguntas, va a armar consultas, comentarios, etc.

-Reunión de revisión: En esta etapa se discuten y registran los resultados, y recomendaciones.

-Reconstrucción: Una vez tenida la reunión la persona encargada va a corregir los defectos, o si hubo una presentación de mejoras, va a evaluar estas mejoras y las va a implementar.

-Seguimiento: Comprobación de que los defectos o mejoras han sido tratados.

-Comprobación de criterios de salida: Una vez que el seguimiento ha terminado vamos a comprobar si los criterios de salida se cumplen. Si no se cumplen, entonces va a haber nuevamente una definición de criterios, una Kick-Off y así sucesivamente hasta que se lleguen al cumplimiento de los criterios de salida y se pueda terminar con la revisión.

6.1.1. Tipos de Revisión

-Revisión Informal o revisión entre pares: es la revisión más simple ya que no tiene un proceso formal. El objetivo es invertir poco para obtener un posible beneficio. Simplemente es iniciada por el autor del código, de la especificación, o del documento que se está revisando; y hay 1 o 2 personas más que están colaborando en esta revisión. Los resultados pueden ser registrados como una lista de acción o una lista de actividades a realizar.

- Revisión Guiada o Walkthrough: este tipo de revisión está liderada por el Autor del componente/ documento, quien a lo largo de la presentación va mostrando el artefacto a revisar, mientras que los revisores hacen preguntas y comentarios y tratan de detectar desviaciones y/o áreas que representen un problema. Es un tipo de revisión que puede llegar a ser formal o informal, de acuerdo si se ha preparado o no.

El objetivo principal aquí es ganar conocimiento sobre el objeto que se está probando, entender el objeto o detectar defectos.

Las ventajas que encontramos para este tipo de revisión: utiliza un esfuerzo reducido en la preparación de la sesión de revisión y es una sesión que puede ser iniciada a través de notificaciones realizadas con poca antelación.

- Revisión Técnica o Technical Review: Es una revisión formal o informal, donde participan técnicos expertos, preferentemente externos a la organización y es liderada por un moderador, no el autor. En este

tipo de revisión habrá una preparación previa a la reunión por parte de los revisores y al final estos expertos presentarán sus recomendaciones (mediante un informe de revisión).

El principal objetivo es discutir soluciones, tomar decisiones, evaluar alternativas, detectar defectos, comprobar estándares. Aquí la pregunta sería: ¿este objeto es apto para su uso?

- Inspección: Revisión formal (es decir, cuenta con procesos de preparación, ejecución, documentación y seguimiento), contiene roles determinados donde se utilizan listas de comprobación y métricas (como por ejemplo, errores por página). Tenemos criterios de entrada y salida para aceptar o no el producto a revisar.

El objetivo principal es detectar defectos utilizando un método sumamente estructurado.

Entre las ventajas de utilizar este tipo están:

6.2. Análisis estático con herramientas

Mediante el análisis estático con herramientas se comprueban:

- Reglas y estándares de programación
- Diseño de un programa (la herramienta es: análisis del **flujo de control**)
- Uso de datos (la herramienta es: análisis del **flujo de datos**)
- Complejidad de la estructura de un programa (la herramienta es: métrica, por ejemplo número ciclomático)

Lo que buscan estas herramientas es PREVENIR defectos antes de ejecutar las pruebas, para advertir aspectos sospechosos del código, detectando inconsistencias, etc.

Las herramientas son:

- Compiladores: Detecta errores sintácticos en el código fuente, comprueba consistencias entre tipos de variables, detecta variables no declaradas, etc.

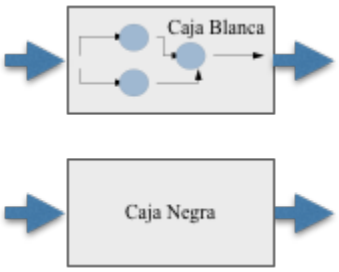
- Analizadores: trata convenciones y estándares, acoplamiento de objetos, etc

7. Técnicas Dinámicas

Las pruebas dinámicas se dividen en dos grandes categorías/grupos. La agrupación se realiza en función del carácter básico del método utilizado para obtener los casos de prueba

Identificar los diferentes tipos de defectos que se pueden encontrar en un sistema de software, requiere diferentes técnicas y tipos de prueba.

Para tener una buena cobertura en el sistema de software deberíamos aplicar las 3 técnicas (caja negra, caja blanca y basadas en experiencia). Ya que cada una descubre defectos diferentes.



7.1. Técnicas de Caja Blanca

Las técnicas de caja blanca se realizan cuando el tester tiene acceso al código fuente de la aplicación y a sus algoritmos y estructuras de datos utilizadas. La implementación de este tipo de pruebas requiere habilidades de programación, un conocimiento del framework de desarrollo y un cierto conocimiento funcional que permita conocer qué misión tienen determinadas clases y métodos. La estructura del programa es el foco.

Entre los objetivos más importantes de ejecutar tests de caja blanca, están:

- Alcanzar código que no se alcanzó con las pruebas de caja negra.
- Encontrar inconsistencias en el código o código inútil.

Esta técnica se utiliza en:

- Nivel de componente: la estructura de un componente software, es decir sentencias, decisiones, ramas, distintos caminos.
- Nivel de integración: la estructura puede ser un árbol de llamada (un diagrama en el cual unos módulos llaman a otros módulos, o una sus).
- Nivel de sistema: la estructura puede ser una estructura de un menú, un proceso de negocio o la estructura de una página web.

Ventaja: La principal ventaja de aplicar técnicas de caja blanca es que los métodos pueden aplicarse en etapas temprana del sistema, y como hemos mencionado anteriormente, encontrar fallas en etapas tempranas hace que el costo de corregir esta falla es mucho menos que corregirlo en etapas finales, además tenemos una cobertura casi total de la estructura del sistema.

Desventaja: La principal desventaja que tienen estas técnicas es que el tester o la persona que esta probando tiene que tener conocimientos en lenguajes de programación

7.1.1. Cobertura de Sentencia (statement coverage)

La cobertura es la medida en que un conjunto de pruebas ha probado una estructura, pero expresada como porcentaje (Por ejemplo: se probó el 15% o el 80% de la estructura del módulo).

Comprueba el número de sentencias ejecutables que se han ejecutado. Como el nombre lo dice, el foco aquí es la sentencia del código de un programa. (¿Qué casos de prueba son necesarios con el objeto de ejecutar todas (o un porcentaje determinado) las sentencias del código existentes?). Vamos a satisfacer el criterio de cobertura de sentencias si todas las sentencias del programa son ejecutadas por lo menos una vez.

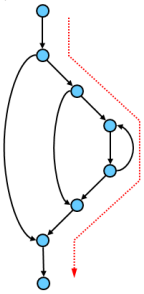
La base de este análisis es el gráfico (o diagrama) de flujo de control donde las instrucciones están representadas por nodos y el flujo de control entre instrucciones está representado por flechas.

Cobertura = (No. de sentencias ejecutadas / No. total de sentencias) * 100

El objetivo de estas pruebas es detectar aquellos bloques de código que no se utilizan (Si hay código muerto en el programa, no se podrá lograr una cobertura del 100%) aunque no permiten determinar la falta de código.

Tenemos el siguiente segmento de código:

```
if (i > 0) {
   j = f(i);
   if (j > 10) {
      for (k=i; k > 10; k --) {
            ...
      }
   }
}
```



Teniendo el código, podemos traducirlo a un diagrama. Tenemos 2 sentencias IF y un WHILE, sin embargo, Todas las sentencias de este programa pueden ser alcanzadas haciendo uso de este camino a la derecha. Por lo tanto solo necesitamos un caso de prueba para alcanzar el 100% de cobertura

7.1.2. Cobertura de Decisión (= cobertura de rama) (“decision coverage = branch coverage”)

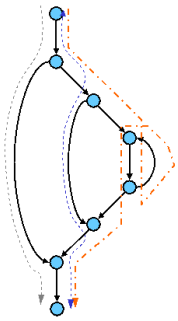
Una decisión es un punto en el código en el que se produce una ramificación. Entonces se satisface el criterio de cobertura de decisión si todas las condiciones del programa son ejecutadas ya sea por verdadero (True) o falso (False). La cobertura de decisión se centra en el flujo de control en un segmento de programa (no los nodos sino las aristas del diagrama de flujo de control). Todas ellas deben ser cubiertas por lo menos una vez.

Cobertura = (No. de decisiones ejecutadas / No. total de decisiones) * 100

Una cobertura de decisión del 100% siempre incluye una cobertura de sentencia del 100%

Tomando el ejemplo anterior, tenemos 3 caminos. La primera sentencia IF se divide en 2 caminos y uno de ellos se divide nuevamente en otros 2 caminos, uno de los cuales es un bucle.

Para tener una cobertura del 100% necesitamos 3 casos de prueba.



7.1.3. Cobertura de Camino (path coverage)

Le vamos a llamar camino/path a una combinación de segmentos de programa, en un diagrama de flujo también sería una determinada secuencia de nodos y aristas alternados.

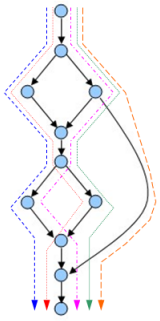
Comprueba el número de caminos linealmente independientes que se han ejecutado en el diagrama de flujo de la unidad que se está probando. El objetivo en esta prueba, es decir, el criterio de salida es alcanzar un porcentaje definido previamente de cobertura de camino.

Cobertura = (No. de caminos ejecutados / No. total de caminnos) * 100

Las sentencias son nodos, el flujo de control está representado por las aristas y cada camino es una vía única desde el inicio al fin del diagrama de flujo de control

La cobertura de camino es más exhaustiva que la cobertura de sentencia y de decisión, pero 100% de cobertura de camino sólo se puede lograr en programas muy simples

Si utilizáramos la Cobertura de decisión, solo necesitaríamos 3 caminos para lograr la cobertura completa. En este caso debemos ejecutar 5 caminos diferentes para cubrir 100% de cobertura de caminos



7.1.4. Pruebas de condición y cobertura:

Aquí mencionaré otro tipos de prueba que se pueden utilizar:

Como principal característica tienen que el porcentaje de todos los resultados individuales de condición afectan al resultado de una decisión.

Otra característica es que detectamos defectos que provienen de implementar condiciones múltiples.

Tipos de Condiciones y Cobertura

- Cobertura de Condición Simple
- Cobertura de Condición Múltiple
- Mínima Cobertura de Condición Múltiple

7.2. Caja Negra

Esta estrategia de testing se centra en la verificación de las funcionalidades de la aplicación: Datos que entran, resultados que se obtienen, interacción, funcionamiento de la interfaz de usuario y en general todo aquello que suponga estudiar el correcto comportamiento que se espera del sistema. No obstante, como el estudio de todas las posibles entradas y salidas de un programa sería impracticable se selecciona un conjunto de ellas sobre las que se realizan las pruebas.

Para seleccionar el conjunto de entradas y salidas sobre las que trabajar, hay que tener en cuenta que en todo programa existe un conjunto de entradas que causan un comportamiento erróneo en nuestro sistema, y como consecuencia producen una serie de salidas que revelan la presencia de defectos. Es necesario encontrar una serie de datos de entrada cuya probabilidad de pertenecer al conjunto de entradas que causan dicho comportamiento erróneo sea lo más alto posible.

Se observa el objeto de prueba como una caja. Aquí el sistema es tratado como un sistema cerrado del que se desconoce cómo fue desarrollado. Entonces, la diferencia fundamental respecto al testing de caja blanca es que en este caso no se trabaja con el código fuente sino con el programa.

Es la técnica más cercana a la experiencia del usuario. El principal objetivo es asegurar que el sistema funciona de acuerdo a los requerimientos y cumple las expectativas del usuario. También conocido como **Test Funcional o Test de comportamiento.**

El resultado de las pruebas de caja negra depende de la calidad de la especificación del sistema. por lo tanto si las especificaciones son erróneas, también lo serán los casos de prueba.

Errores de interfaz, comportamiento incorrecto, baja performance son ejemplos de defectos que podemos encontrar con esta técnica. Se puede utilizar en todos los niveles de prueba (testing unitario, de integración, de sistemas, de aceptación).

Ventajas:

- Que las pruebas se realizan desde un punto de vista del usuario buscando discrepancia con las especificaciones.
- Que el Tester no necesita conocimiento de lenguajes de programación.
- Las pruebas pueden diseñarse a medida que se terminan las especificaciones, no hace falta esperar a que el desarrollar termine con el código.

Desventajas:

- No hay forma de probar todos los caminos posibles en el sistema.
- Si las especificaciones no son claras o falta algo, los casos de pruebas no serán buenos o de buena cobertura.

7.2.1. Partición de equivalencia o clase de equivalencia (CE - Equivalence Partitioning)

Es el método que la mayoría de los testers hacen de forma intuitiva: dividimos los posibles valores en clases, mediante lo cual observamos valores de entrada de un programa y valores de salida.

La partición equivalente permite definir casos de pruebas que descubran clases de errores, reduciendo así el número total de casos de pruebas que hay que desarrollar. Entonces tenemos que con una mínima cantidad de casos de pruebas se puede esperar un valor de cobertura específico y además es aplicable a todos los niveles de prueba (componente, integración, sistema y de aceptación).

El rango de valores definido se agrupa en **clases de equivalencia** donde:

- todos los valores para los cuales **se espera** que el programa tenga un **comportamiento común** se agrupan en una clase de equivalencia (CE)

- Las clases de equivalencia pueden consistir en un **rango** de valores (por ejemplo, 0<x<10) o en un valor **aislado** (por ejemplo, x = Verdadero)

Entonces, las clases de equivalencia se dividirán también en CE validas y CE inválidas. Entre las inválidas nos encontramos con valores que están fuera de rango y valores de formato incorrecto.

Por ejemplo:

Si mi valor x está definido como $0 \leq x \leq 10$ tendremos 3 clases de equivalencias:

$0 \leq x \leq 10$ - valores de entradas válidos

$x < 0$ - valores de entradas inválidos

$x > 10$ - valores de entradas inválidos

También podemos definir otras clases de equivalencias no validas, por ejemplo: que x no sea numérico o que no tenga un formato numérico admitido.

Por último, hay que definir un representante para cada clase, el cual será nuestro caso de prueba:

Variable	Clase de Equivalencia	Representante
Valor válido	EC_1: $0 \leq x \leq 10$	+5
Valor inválido	EC_2: $x < 0$	-15
	EC_3: $x > 10$	+20
	EC_4: x no entero	Hello

Ejemplo 2

El precio final de un artículo se calcula en base a su precio de venta al público y un descuento expresado en %.

Sabemos que el precio de venta será un valor numérico mayor a 0 y que el porcentaje será un valor entre 0 y 100.

La definición de CE válidas e inválidas es la siguiente:

Variable	Clase de Equivalencia	Estado	Representante
Precio de Venta	EC1: x > 0	Válido	1000
	EC2: x =< 0	No Válido	-10
	EC3: x no numérico	No Válido	“Test”

Descuento	EC4: 0 =< x =< 100	Válido	15
	EC5: x < 0	No Válido	-20
	EC6: x>100	No Válido	200
	EC7: x no numérico	No Válido	‘Hola’

Se escriben los casos de prueba tal que cubran la mayor cantidad de clases de equivalencia válidas (TC1). En este caso necesitamos 1 solo caso de prueba para las CE válidas.

Y luego los casos de prueba para los CE NO válidos, combinando CE válidas con otros elementos, ya que no se hacen casos de prueba de combinaciones de Clases de Equivalencia en donde todas sean inválidas.

Variable	CE	Estado	Representante	TC1	TC2	TC3	TC4	TC5	T6
Precio de Venta	EC1: x > 0	Válido	1000	*	*	*	*		
	EC2: x =< 0	No Válido	-10					*	
	EC3: x no numérico	No Válido	“Test”						*
Descuento	EC4: 0 =< x =< 100	Válido	15	*				*	*
	EC5: x < 0	No Válido	-20		*				
	EC6: x>100	No Válido	200			*			
	EC7: x no numérico	No Válido	‘Hola’				*		

Así concluimos que disponemos de 6 casos de pruebas, 1 de ellos positivos y los demás negativos

7.2.2. Análisis de Valores Límite (Boundary Value Analysis)

Permite ampliar o complementar la técnica anterior introduciendo una regla para seleccionar representantes. Probamos con más énfasis los valores limites de cada clase ya que frecuentemente no están bien definidos o implementados. Esta técnica también puede ser utilizada en todos los niveles de prueba.

En esta técnica, también seleccionamos representantes. Por lo tanto vamos a tener 2 partes:

* Partición en clases de equivalencia: Evalúa un valor (representante) de la clase de equivalencia

* Análisis de valores límite: Evalúa los valores límite Y su entorno

Rango de valores para un descuento en %: $0,00 \leq x \leq 100,00$

- **Definición de CE**
  3 clases:
  - 1. CE: $x < 0,00$
  - 2. CE: $0,00 \leq x \leq 100,00$
  - 3. CE: $x > 100,00$
- **Análisis de valores límite**

  Extiende los representantes a:

  2. CE: *-0,01*; *0,00*; *0,01*; *99,99*; *100,00*; *100,01*

7.2.3. Tablas de Decisión

Todos los métodos anteriores no tienen en cuenta el efecto de dependencias y combinaciones. El uso del conjunto completo de las combinaciones de todas las clases de equivalencia de entrada conduce a un número muy alto de casos de prueba (explosión de casos de prueba).

Aquí identificamos de manera sistemática las combinaciones de entradas (llamadas combinaciones de causas) que no podrían ser identificadas utilizando otros métodos. Los casos de prueba son fáciles de obtener a partir de la tabla de decisión.

7.2.4. Pruebas de transición de estado

La mayoría de los métodos que hemos visto hasta ahora tienen en cuenta el sistema solo en términos de datos de entrada y de salida. Lo que muchas veces sucede es que el objeto de prueba en cuestión puede tener estados también, por lo tanto también hay que probarlo.

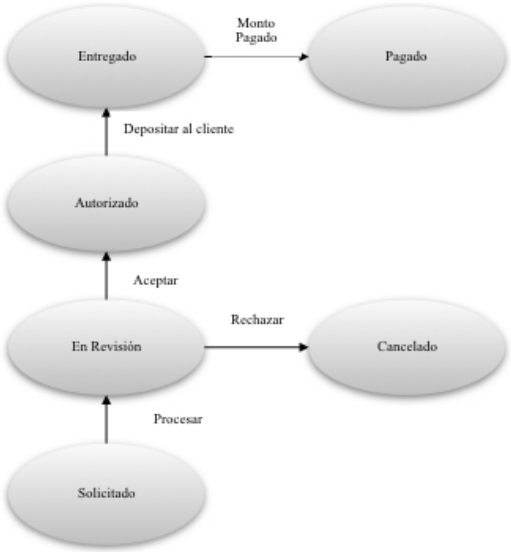
El principal objetivo es probar que los estados requeridos funcionan adecuadamente. Estos estados se modelan mediante diagramas de transición de estados

- El estado inicial es la raíz del árbol
- Para cada estado que pueda ser alcanzado desde el estado inicial se crea un nodo que está conectado a la raíz por una rama
- Esta operación se repite y finaliza cuando:
  - El estado del nodo es un estado final (una hoja del árbol) o

- El mismo nodo con el mismo estado ya es parte del árbol

En una aplicación de préstamos los estados serán: Solicitado, En revisión, Autorizado, Entregado, Cancelado, Pagado.

El diagrama sería:



Cada camino desde la raíz a una hoja entonces representa un caso de prueba de transición de estado.

Teniendo en cuenta el gráfico anterior, tendríamos 2 casos de prueba.

Estado 1 (E1)	Estado 2 (E2)	Estado 3 (E3)	Estado 4 (E4)	Estado 5 (E5)	Estado final (EF)
Solicitado	En revisión	Cancelado			Cancelado
Solicitado	En revisión	Autorizado	Entregado	Pagado	Pagado

Todos los casos de prueba habrán sido creados si:

Todo estado ha sido alcanzado, por lo menos, una vez

Toda transición ha sido ejecutada, por lo menos, una vez

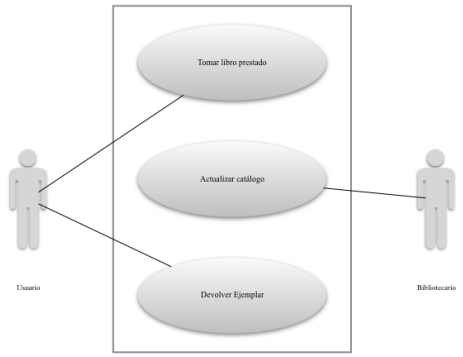
También podemos tener transiciones inválidas, para determinar cómo el sistema se comporta en estos casos, y con esto tendríamos casos de pruebas inválidos, es decir que deberían dar error al ejecutarlos. En el caso de aplicación de préstamo, un ejemplo sería que desde ENTREGADO haya una rama que sea de nuevo DEPOSITAR AL CLIENTE AUTORIZADO.

7.2.5. Pruebas basadas en Casos de Uso

Los casos de prueba se obtienen directamente a partir de los casos de uso del objeto de prueba. El objeto de prueba es visto como un **sistema** que reacciona con **actores**. Un caso de uso describe la interacción de todos los actores involucrados. Todo caso de uso tiene **precondiciones** (que deben se cumplidas con el caso de prueba de forma satisfactoria) y **poscondiciones** (que describen el sistema tras la ejecución del caso de prueba.

Este tipo de pruebas, son más apropiadas para **pruebas de aceptación** y **pruebas de sistema**, dado que cada caso de uso describe un escenario de usuario a probar.

En una biblioteca existen estos casos de uso:



La "caja" donde están los casos de uso define los límites del sistema de biblioteca es decir, los casos de uso representados son parte del sistema a modelar pero no los actores. También tenemos que todos estos elementos descriptivos son utilizados para definir los casos de prueba correspondientes. Entonces, a partir de un caso de uso voy a tener como mínimo un caso de prueba. La cantidad de casos de prueba derivados de cada caso de uso va a depender de la complejidad del caso de uso. Debido a que este sistema de biblioteca tiene 3 casos de uso, vamos a tener como mínimo 3 casos de prueba.

Los casos de uso se representan con óvalos y los actores con figuras de personas. El actor "usuario" podrá tomar prestado un libro y también devolverlo. Mientras que el actor "bibliotecario" actualiza los catálogos de los libros

Cada caso de uso puede es utilizado como la fuente para un caso de prueba.

7.3. Basados en Experiencias

El conocimiento de las personas involucradas es importante para generar casos de pruebas. Es un método complementario para los demás. Las preguntas aquí serían: ¿Dónde se han acumulado errores en el pasado? ¿Dónde falla normalmente el software? Por lo tanto el tester utilizará la predicción de errores y las pruebas exploratorias para la creación de casos de prueba. Es un método complementario para los demás.

Vamos a utilizar:

- Intuición: Conocimiento de Testers, usuarios, personas relacionadas con el software sobre su uso y el ambiente,

- Experiencia: Conocimiento de errores pasados en su distribución
- Pruebas exploratorias: pruebas iterativas basadas en el conocimiento adquirido respecto del sistema. Es muy útil cuando el tiempo disponible para pruebas es escaso.

7.4. Cómo seleccionar la técnica correcta.

¿Cómo sabemos que técnica de especificación elegir para crear los casos de prueba? Esto depende de muchos factores, entre ellos:

- Tipo de sistema: ¿Tenemos disponible el código fuente? ¿Tenemos personas que comprenden el código fuente para hacer casos de pruebas? ¿Tenemos buenas especificaciones para casos de prueba de caja negra o solo exploratorias?

- Requerimientos: ¿Tenemos requerimientos especificados? ¿Bien especificados? ¿Disponemos del cliente disponible para respondernos preguntas?

- Niveles de riesgo: ¿El objeto de prueba es muy usado? ¿Hay algún estándar contractual sobre la ejecución de pruebas que tiene que ser cumplido? ¿Cuánto tiempo disponemos? ¿Tenemos que entregarlo pronto?

- Objetivo del Test: ¿Qué pruebas no funcionales son necesarias? ¿Las funcionales han sido requeridas formalmente? ¿Qué debe ser probado sí o sí?

- Documentación: ¿Disponemos de documentación importante, como documentación sobre el negocio, documentación de usuario?

- Conocimiento del Tester/s: ¿Qué conocimiento tiene el tester? ¿Tiene conocimiento de código fuente?, ¿Tiene conocimiento de técnicas de pruebas? ¿Es principiante?

- Tiempo y presupuesto: ¿Cuánto tiempo tenemos?

8. Creación de casos de prueba

8.1. Cómo crear casos de prueba

- Hay que conocer la aplicación: Para esto, necesitamos obtener la mayor cantidad de información posible (requerimientos, casos de uso, guías de usuario, tutoriales, usar nosotros mismos el sistema). Y una vez que contamos con esto, haremos una lista de las funcionalidades.

- Estandarizar la forma de trabajo: Usando diferentes archivos para casos de prueba de diferentes funcionalidades, usando Excel o alguna herramienta. Estos archivos de casos de pruebas se deberían nombrar de acuerdo a la funcionalidad en la que se están basando.

- Identificar conjuntos de casos de pruebas (test suites). Esto se logra buscando conjuntos de casos de pruebas relacionados por funcionalidad, roles, o integración de ellos. Ciertos casos de pruebas específicos es mejor separarlos para obtener una mejor organización de los mismos, por ejemplo, casos de prueba específicos a navegadores, o sistemas operativos, de usabilidad, de interfaz, etc.

- Estructurar los casos de prueba. Cada caso de prueba contiene una cantidad de datos que permite determinar si una aplicación está funcionando correctamente. Entre ellos:

Nombre de la prueba: el nombre será dado en base al estándar definido en los puntos anteriores. El nombre debe ser único. Por ejemplo: Eliminar_Cliente.

Descripción: es el objetivo que se quiere alcanzar. Ejemplo: Verificar que el sistema solicita confirmación para la eliminación de un registro. Debe estar alineado con el resultado que estamos esperando.

Precondiciones: aquellas situaciones que deben cumplirse para que el Caso de Prueba se pueda ejecutar. Ejemplo: Para eliminar un registro, el usuario debe tener los permisos específicos sobre la aplicación.

Pasos: Secuencia de pasos a seguir para realizar la prueba y alcanzar el objetivo. Ejemplo: Realizar una búsqueda, seleccionar un registro, presionar el botón Eliminar

Resultado esperado: Estado que se espera luego de ejecutar los pasos del caso, los cambios que el usuario ve y los de Base de datos también. Ejemplo: Mensaje de confirmación de eliminación.

Resultado actual: Respuesta que se obtiene luego de ejecutar los pasos.

Estado: Los estados los obtendremos luego de correr los casos de prueba. No hay una definición exacta de cuáles son los estados correctos, cada proyecto o herramienta contiene sus propios nombres, pero en general son:

* Pasó = Los resultados esperado y actual coinciden

* Falló = Los resultados esperado y actual no coinciden

* No probado = No se ha ejecutado

* No aplica = Puede ser cuando el requerimiento ha cambiado

Comentarios: Cualquier información necesaria, una determinada condición.

Dependencias: Orden de ejecución de casos de prueba, como vimos previamente los casos de pruebas se unen en conjuntos o test suite, allí estará definido qué casos de pruebas se ejecutan primero y cuales después.

Referencias: Muestra a que requerimiento / caso de uso corresponde

Tomemos como ejemplo Enviar un correo electrónico:

Caso de Prueba	Nombre	Descripción	Precondición	Pasos	Resultado Esperado	Resultado Actual	Estado	Comentarios	Referencias
1	Crear correo	Componer el correo	Estar logueado	1 – Click en componetr 2 – Tipear	El usuario debe poder tipear el mensaje	El usuario puede tipear el mensaje	**PASO**	No aplcia	Caso de uso 15
2	Enviar correo	Enviar el correo electrónico	Correo ya compuesto y destinatario agregado	1- Click en el boton Enviar	El mensaje debe ser enviado al destinatario correcto	El correo no fue recibido por el destinatario	**FALLO**	Defecto nro 12254	Caso de uso 15

8.2. Ciclo de vida de un defecto.

Antes que nada repasemos el concepto de Defecto. Un defecto o bug, es una condición por la cual el producto de software no cumple con alguno de los requerimientos (o especificación) o expectativas del usuario final. Básicamente, un defecto es un error en el código o la lógica que causa un mal funcionamiento en el sistema. Un defecto es lo que encontramos luego de ejecutar un caso de prueba

El ciclo de vida de un defecto es el ciclo en el cual el defecto atraviesa durante su vida. Comienza cuando el defecto es encontrado, es decir cuando hemos corrido un caso de prueba y hemos encontrado un defecto y va va a terminar cuando es cerrado, luego de asegurarnos de que no puede ser reproducido de nuevo.

Dependiendo de si utilizamos alguna herramienta de seguimiento de defectos, podemos encontrarnos con una gran variedad de nombres para cada etapa del defecto. Es importante definir apropiadamente el ciclo de vida de un defecto, y añadirlo como un proceso más a la estrategia de pruebas. Por ejemplo, todo defecto, como mínimo, pasa por los siguientes estados:

Abierto o Nuevo: El tester encuentra un bug que no había sido detectado anteriormente y lo reporta.

Asignado: El defecto ha sido asignado a un desarrollador.

Arreglado: El desarrollador ha resuelto el problema.

Verificado o Cerrado: Una vez que el tester ha verificado que el defecto ha desaparecido, entonces el defecto se cierra.

Rechazado: Una vez que se ha encontrado el defecto y éste es revisado, si el desarrollador o el analista o quien corresponda considera que no es un defecto, puede llegar a pasar al estado rechazado.

Reabierto: El tester lo ubica en este estado cuando luego de ser arreglado, el defecto puede ser reproducido.

Para que la comunicación entre el equipo de desarrollo y el de pruebas sea correcto, es necesario reportar correctamente los defectos que se encuentran durante la ejecución de los casos de prueba. Para esto, tener en cuenta estas buenas prácticas:

- Asegurarse que realmente es un defecto, por ejemplo, que no sea problema de nuestra configuración.
- Revisar que no haya sido creado anteriormente: en este caso podemos agregar un comentario con mas información.
- Asegurarse que la versión del sistema que estamos utilizando sea la última, así no corremos riesgo de que haya sido ya corregido el problema

8.3. Cómo reportar un defecto

Reportar correctamente un defecto facilita a cualquier persona del proyecto a entender cuál es la falla. Debemos tener en cuenta estas "buenas prácticas":

- Descripción detallada: Se describe el escenario, cuál era el resultado esperado y cuál fue el resultado real, mencionar el caso de prueba que originó el defecto.

- Pasos para reproducir: si el desarrollador no puede seguir los pasos, no podrá corregir el defecto, es por este motivo que los pasos son la parte más importante del reporte.

- Defecto específico: Si se han encontrado 2 problemas aunque sean similares, se crean 2 defectos diferentes. Si el desarrollador descubre que la causa de los dos defectos es la misma, nos avisará, de lo contrario, es mejor tenerlos separados porque así el desarrollador sabe cuál es la causa de uno y cuál es la causa del otro.

- Entorno: Se necesita brindar toda la información posible del escenario en el que estamos (sistema operativo, navegador y versión, versión del sistema)

- Un buen título: El título del defecto debe ser especifico, brindando una especie de sumario de lo que contiene.

- Datos propios: En caso de que el desarrollador no comprenda alguna parte del problema, es necesario que pregunte a la persona que creó el defecto.

- Severidad: Dependiendo de la severidad que le asignemos al defecto, puede que el mismo sea corregido pronto o pasen varios días / meses.

8.4. Severidad y prioridad

Un error que cometemos frecuentemente es no diferenciar la prioridad de la severidad, cuando en realidad son dos conceptos muy diferentes.

Severidad:

La severidad de los defectos depende de las funcionalidades bajo prueba. Se define de acuerdo a en qué medida afecta el defecto al sistema en general.

Si no afecta las funcionalidades, entonces la severidad será trivial (Defectos Cosméticos), Menor (Defecto nivel interfaces) o Normal (Cualquier otro defecto que no interrumpa el uso de la aplicación)

En cambio si afecta las funcionalidades podrá ser Mayor (si las funcionalidades tienen un resultado erróneo o imprevisto) o Urgente (si la aplicación no es utilizable).

Prioridad:

En cambio la prioridad indica qué tan importante es para el negocio o el cliente que se corrija este defecto. Aquí nos vamos a preguntar cuán importante es para el negocio es que se arregle este defecto. Habrá casos en que los defectos tendrán alta severidad pero prioridad baja, o al revés, defectos con baja severidad pero con alta prioridad.