El Testing como parte del proceso de Calidad del Software

Guión de jornadas

Escaleta de días

Bienvenida

Introducción

El presente guión de jornadas corresponde al dictado del curso **El testing como parte del proceso de Calidad del Software** desarrollado por el equipo del **Laboratorio de Testing y Aseguramiento de la Calidad** del centro **INTI Córdoba**.

En él encontraremos una serie de temas e informaciones que apoyan y complementan el dictado del curso de carácter práctico, enfocado a generar en quienes lo atiendan una sensibilización sobre la importancia de la Calidad y el Testing dentro del proceso de desarrollo de Software y del mismo proceso de Calidad.

Así, reforzando la orientación pragmática del curso y el equipo capacitador, se tiene como objetivo que los asistentes finalicen el curso con una experiencia de caso real derivada directamente de la industria, de modo que puedan impactar en sus organizaciones y procesos los beneficios de estas temáticas con resultados tangibles y de provecho.

Consideraciones del material de jornadas

El formato de **guión** seleccionado para el material responde a una necesidad de dictado. Esto es, se establecen pautas, temáticas e informaciones de referencia pero el curso en desarrollo tomará los caminos que el grupo conformado para recibirlo y dictarlo consideren más oportunos y de criterio al momento.

Esto busca una participación activa y a modo de “coaching”, asegurando exponer realidades, acercar a los participantes y potenciar conocimientos a partir de esa apertura.

El caso **práctico** a tratarse será convenido por el equipo capacitador según requerimientos particulares de las jornadas y el grupo receptor.

En este caso se trabajará sobre el Software **Espiga**, que consta de un sistema integral para la industria de la panificación, orientado a la producción, distribución y gestión. Nace en el marco de una tesis de grado presentada en el año 2010 en la Universidad Tecnológica Nacional Facultad Regional Córdoba para la carrera de Ingeniería en Sistemas de Información, la cual fue aprobada y liberada como proyecto de código abierto.

Día 1

Conceptos Fundamentales

Esta sección inicial apunta a establecer los conceptos básicos y darles un contenido para el desarrollo del curso **El Testing como parte del proceso de Calidad del Software**.

Como bienvenida se recorrerá un camino de sensibilización y casos históricos que abran puerta a la importancia del tema, posicionen las ideas que guiarán el resto del desarrollo de la capacitación y expongan las principales misiones, tareas y componentes del testing y la calidad del Software.

¿Por qué asegurar la calidad?

Poder asegurar la calidad de un producto, proceso, herramienta o solución significa poseer la suficiente capacidad para tomar un compromiso sobre los resultados y el rendimiento del mismo frente a un tercero.

Esa capacidad se basa en información, en conocimiento y en una cobertura previa que de alguna manera certifique, más allá de las palabras, lo que se está afirmando.

En sistemas críticos, una falla puede ocasionar pérdida de vidas humanas y repercusiones legales y económicas severas.

En sistemas de un impacto menor quizás los riesgos no son tan altos como los de la pérdida de la vida humana, pero de igual modo implicarán una exposición desventajosa que producirá, al menos, un deterioro en la imagen de quien dejó pasar la oportunidad de asegurar la calidad.

Su importancia es crucial y lleva sus particularidades en el mundo del Software. Veamos algunos casos históricos que ilustran lo planteado.

Casos Históricos

Microsoft Excel 2007

El conocido producto de Microsoft fue lanzado al mercado con un error (bug) en la operación de multiplicación.

Aunque computaba y almacenaba correctamente operaciones que daban como resultado el valor **65535** en ocasiones mostraba el valor **100000**.

Esto generó una pérdida de imagen, haciendo que los contadores y usuarios del programa comiencen a corroborar las operaciones con una calculadora tradicional.

La **Solución** ofrecida por Microsoft para resolver el problema fue la liberación de un parche oficial que arreglaba el error.

Nasa Climate Orbiter

Otro paradigmático caso de errores costosos por fallas en el proceso de testing y calidad fue el de la sonda de la NASA *Mars Climate Orbiter* lanzada el 11 de Diciembre de 1998.

La Mars Climate Orbiter se destruyó debido a un error de navegación, consistente en que el equipo de control en la Tierra hacía uso del Sistema Anglosajón de Unidades para calcular los parámetros de inserción y envió los datos a la nave, que realizaba los cálculos con el sistema métrico decimal. Es decir, un defecto en la traducción entre las unidades de medición inglesas y las decimales.

Esto produjo un fallo de cálculo de trayectoria cuando la sonda se aproximó a la superficie del planeta Marte, haciendo que esta vuele mucho más cerca de lo esperado y sea destruida por la fricción de la atmósfera del planeta.

El costo de este error de programación no detectado fue de **125 millones de dólares** y el fracaso de la misión.

Cohete Ariane 5

El 4 de junio 1996, el cohete no tripulado Ariane 5, lanzado por la Agencia Espacial Europea, explotó sólo cuarenta segundos después de su despegue desde Kourou, en la Guayana francesa.

Era el primer vuelo del lanzador que por motivos de un error en el sistema inercial referencial produjo una pérdida de **500 millones de dólares** y el trabajo de una década de desarrollo con un costo total de **7 mil millones de dólares**.

El error particular que produjo la desviación de su trayectoria y la destrucción del mismo fue un **bug** en el software, según informó la comisión investigadora: Una excepción no capturada al intentar convertir un flotante de 64 bits a entero con signo de 16 bits en el cálculo de la trayectoria.

El software de guiado inercial del Ariane 5 era heredado del Ariane 4, pero en el caso anterior no se había manifestado ya que las trayectorias de los lanzadores eran distintas y no se produjeron las situaciones específicas para el error.

Las causas de la falla fueron apuntadas a las fases de diseño y especificación, sumando a que la prueba del módulo de guiado no se realizó de manera correcta, siendo el software de vuelo y el software de navegación probados por separado.

Es uno de los errores de software más costosos de la historia, sin haber producido pérdida de vidas humanas, pero con un costo material y de imagen por demás relevante.

Instituto del Cáncer de Panamá

La combinación de una falla de software y una falla humana al seguir los procedimientos causaron que se proporcione una sobredosis de radiación a los pacientes con cáncer en el instituto de la ciudad de Panamá.

El tratamiento de 28 pacientes fue modificado a pedido de un oncólogo de radiación, manipulando los ingresos de datos al Sistema de Planeamiento de Tratamiento (SPT) que calcula la distribución de dosis resultantes y determina el tiempo de tratamiento.

En uno de los métodos de introducción de datos a la computadora, se obtuvo un resultado que indicó un tiempo de tratamiento sustancialmente más largo del que debería. Como resultado, los 28 pacientes recibieron una dosis más alta que la prescrita entre Agosto del 2000 y Marzo del 2001.

El protocolo modificado fue utilizado sin una prueba de verificación, por ejemplo un cálculo manual del tiempo de tratamiento en comparación con el tiempo de tratamiento calculado por la computadora, o la simulación de tratamiento por irradiación de agua fantasma y midiendo la dosis alcanzada. La exposición accidental pasó desapercibida por varios meses.

La mayoría de los pacientes expuestos han muerto, algunos debido a la radiación, otros por el estado avanzado del cáncer. El gobierno de Panamá acordó compartir urgentemente las conclusiones del informe para ayudar a prevenir accidentes similares. Los físicos del Instituto Oncológico Nacional involucrados fueron llevados a juicio por los familiares de los pacientes.

Los peores bugs de la historia

Los bugs o errores en informática son una constante que viene acompañando los avances sin relegar espacio. A través de los años se han identificado y desarrollado técnicas para evitarlos, solucionarlos y detectarlos, pero su presencia sigue siendo una realidad.

Resumimos una lista de los 10 defectos más famosos de la historia del software:

- 1962: Error de trayectoria en la sonda espacial Mariner I
- 1982: Explosión del gaseoducto soviético por saboteo de Estados Unidos
- 1985 – 1987: Muertes por radiación del acelerador médico Therac-25
- 1988: Demonio finger de UNIX
- 1988 – 1996: Generador de números aleatorios para el sistema de seguridad Kerberos
- 1990: Caída de la red AT&T por un bug en los switches de larga distancia
- 1993: División de punto flotante del Intel Pentium

- 1995 – 1996: El ping de la muerte a terminales Windows
- 1996: Explosión del transbordador Ariane 5
- 2000: Instituto Nacional del Cáncer de Panamá

Impacto económico de una infraestructura inadecuada para testing

Dentro del mercado estadounidense de software, del cual se obtuvieron los datos, el total de las ventas del año 2000 fueron de $180 mil millones de dólares.

El costo de los bugs de software le han implicado a su economía entera un costo total de $59,5 millones mil millones de dólares anuales, afrontándola en un 50% los usuarios y otro 50% los productores.

Otro dato interesante ofrecido por el NIST (National Institute of Standards and Technology – Instituto Nacional de Estándares y Tecnología) es que el 80% del costo total de un desarrollo se utiliza en identificar y corregir defectos. La identificación y remoción temprana de los defectos eliminaría un tercio de este costo total, hablamos de $22 mil millones de dólares.

Calidad de software

Antes de comenzar expondremos una serie de definiciones del concepto:

- Es la capacidad del producto software de satisfacer las necesidades establecidas e implícitas cuando se utiliza bajo condiciones específicas. *(ISO/IEC 25000:2005)*
- Es la totalidad de características relacionadas con su habilidad para satisfacer necesidades explícitas o implícitas *(ISO/IEC 9126)*
- Es el grado en que un componente, sistema o proceso satisface las necesidades o expectativas especificadas, implícitas u obligatorias de la organización, sus clientes y demás partes interesadas *(ISO 9000:2000)*

Según la IEEE, los factores que afectan a la calidad de Software son:

- Ausencia de defectos, comprobable mediante:
  - Tasa de defectos
  - Confiabilidad
- Satisfacción de usuarios
- Conformidad con los requerimientos

Según la IEEE la calidad debe ser **medible** y **predecible**.

Verificación y Validación

Uno de los objetivos principales en el desarrollo de software es

conseguir productos de alta calidad.

La calidad involucra confiabilidad, mantenibilidad, interoperabilidad, etc.

Las tareas de validación y verificación permiten analizar la calidad del software y tomar acciones para mejorarla.

Verificación: el software debería realizar lo que su especificación indica. Responde a la pregunta *¿Construimos el producto correctamente?*

Validación: el software debería hacer lo que el usuario requiere de él: *¿Construimos el producto correcto?*

Una manera de realizar las tareas de Verificación y Validación es a través del análisis del código fuente, de modelos, de especificaciones, de documentación, etcétera. En particular disponemos de técnicas de **análisis estático** y **análisis dinámico**.

El análisis **estático** infiere propiedades del software mediante el examen de la documentación y el código fuente. Generalmente es impreciso; en sus versiones “automáticas” puede dar lugar a falsos positivos/negativos y con frecuencia es conservador.

Ejemplo paradigmático: inspección manual.

Otros: chequeo de tipos (en compilación)

El análisis **dinámico** examina las propiedades del software mediante su ejecución extrayendo información de las corridas, los modelos, ejecutables, prototipos y programas. Es sumamente “preciso” pero intrínsecamente incompleto.

Ejemplo paradigmático: testing funcional.

Otros: profiling de consumo de recursos, chequeo en tiempo de ejecución de propiedades del programa, etcétera.

¿Qué es testing?

Testing **NO** es:

- Depurar código
- Verificar que las funciones del software se implementen
- Demostrar que no hay defectos

Testing **SI** es:

un proceso destructivo que trata de encontrar defectos, poniendo al tester en una posición negativa para demostrar que algo es incorrecto. El testing detecta fallas pero no ahonda en los motivos de ellas, el objetivo es encontrarlas y una vez detectadas, si existe la necesidad de corregirlas es el desarrollador el que estará encargado de descubrir su origen y repararlas.

Algunas definiciones

- Es el proceso de ejecutar un programa o sistema con la intención de encontrar defectos. *(The art of software testing, Glenford Myers)*
- El proceso de analizar un ítem de software para detectar la diferencia entre las condiciones existentes y las condiciones requeridas y evaluar las características de los ítems del software. *(IEEE 829-1998 - Standard for Software Test Documentation)*
- El proceso de ejecutar un sistema o componente bajo condiciones específicas, observando o registrando sus resultados, y evaluando algún aspecto del sistema o componente. *(ANSI – 1990 – Std 610.12)*
- El testing es un proceso de ingeniería, un ciclo de vida concurrente que usa y mantiene el testware con el fin de medir la calidad del software bajo prueba. *(Systematic Software Testing, Rick Craig y Stefan Jaskiel)*

Misión del testing

La misión principal del proceso de testing es la de **generar información**.

La principal razón por la que los testers existen es para proveer información que pueda ser utilizada por otros y éstos la utilicen para generar cosas de valor. *(James Bach)*

El testing es una forma de medir la calidad.

Terminología básica

Un **error** genera ningún, uno o más **defectos** que generan ninguna, una o más **fallas**.

- **Error:** Un error es una acción humana que produce un resultado incorrecto. Éste puede derivar en uno o más defectos y puede ser introducido en cualquier momento del ciclo de desarrollo.
- **Defecto (bug, fault)**: Es un desperfecto en un componente o sistema y al ejecutarse puede causar una falla en la función requerida. Puede ocurrir también que exista un defecto pero que no produzca una falla.
- **Falla:** Es una diferencia entre los resultados esperados y los reales, implica que se ha generado una desviación en la respuesta, servicio o resultado esperado de un componente o sistema. Esto ocurre cuando un programa no se comporta adecuadamente.

Existen una serie de normas y estándares para la nomenclatura del Testing y la Calidad de Software:

- **BS 7925-1:98 -** Glosario de términos de testing de software, basado principalmente en terminología de testing

de componentes.

- **IEEE 1008:1993 –** Estándar para testing unitario de software.
- **IEEE 829:1998 –** Estándar para la documentación de pruebas de software.
- **IEEE 610.12:1990 –** Glosario de terminología de ingeniería de software.

Actualmente la **ISQTB** (International Software Testing Qualifications Board – Junta Internacional de Calificación de Prueba de Software) trabaja en la unificación de los términos usados en testing de software, habiendo liberado un Glosario de términos en su versión 2.2 liberada en Octubre de 2012.

Ejemplo de error, defecto y falla

A fines del ejemplo vamos a plantear primeramente el requerimiento y el caso de prueba que de inicio a la prueba:

- La función **doblar(int param)** devuelve el doble del valor pasado por parámetro cada vez que se ejecuta.
- El caso de prueba será **doblar(3)** y deberá retornar el valorsalto flujo de texto **6** para que sea exitosa.
- Planteamos el método:

```
int doblar(int param)
{
    int res;
    res = param * param;
    return(res);
}
```

- La llamada al método **doblar(3)** retorna el valor **9**
- El resultado representa una **FALLA**
- Esta falla se debe a un **DEFECTO** en la línea 3 del código
- El **ERROR** es tipográfico

Principios del testing

El testing es efectivo para demostrar la presencia de defectos en un software, pero no puede asegurar ni demostrar de manera alguna la ausencia de los mismos. Puede lograr la reducción de probabilidad de existencia de defectos no descubiertos pero el hecho de NO encontrar defectos en un software implica que esa no es una prueba de corrección.

El testing no es exhaustivo

El testing exhaustivo implica la prueba de todos los caminos posibles de ejecución de un software.

Supone ejecutar realmente todas las posibilidades de entradas y salidas de un sistema y sus diferentes combinaciones, lo que aseguraría una cobertura total y completa del funcionamiento y podría asegurar al 100% la ausencia de errores.

Esto es, claramente, imposible en la práctica debido a que la capacidad de procesamiento de cualquier computador es limitada, aún siendo que se trabaja a un nivel de rendimiento altísimo esta técnica llevaría, con seguridad, a una ejecución de tiempos infinitos.

Testing temprano

Las actividades de testing deberían iniciarse lo antes posible, refiriéndose a esta tarea como **testing temprano**.

Esto implica que comenzar a pensar en cómo vamos a probar un software, incluso cuando este todavía no ha sido desarrollado, tiene un sentido fuerte. Al empezar a definir pruebas y buscar formas de destruir la confiabilidad de lo que se está por construir nos permite tener una seguridad y una cobertura previa y predecible que no existiría si iniciamos el testing sobre el final del proceso.

Tiene como objetivo claro dar visibilidad de manera temprana al área de testing de cómo se va a probar lo desarrollado y disminuir los costos por correcciones de defectos.

Existen algunos requerimientos para realizar el testing temprano de manera eficiente:

- Definición de la estrategia y alcances de la prueba
- Organización del equipo de prueba
- Capacitación del equipo
- Establecimiento del esquema de reportes
- Provisión de recursos de hardware y de software

El testing debe validar al cliente

Además de la capacidad de encontrar y reparar defectos, el software probado debe satisfacer las necesidades y las expectativas del usuario para poder referirnos al concepto de **calidad**. El testing temprano ayuda también a esto ya que ataca la naturaleza del problema a resolver desde una perspectiva distinta a la del desarrollo, del diseño y del análisis

Contexto

La dependencia del contexto a la hora de plantear una prueba es fundamental, su proceso es diferente cuando varía este

contexto.

Por ejemplo, a la hora de probar un sistema médico crítico no tendremos las mismas consideraciones que al probar un sistema de comercio electrónico. Los impactos de los errores son considerablemente distintos, pero no sólo eso, incluso la infraestructura, el proceso y la capacidad de quien realiza la prueba varía según el contexto.

Si probamos una aplicación de Televisión Digital nos encontramos con un conjunto de actores interventores (tanto humanos como tecnológicos) en el proceso mucho más amplio que a la hora de efectuar el test sobre una terminal magnética de boletos de transporte.

Esto nos exige una consideración especial del contexto y un análisis particular en cada caso a la hora de realizar una prueba de software.

Aseguramiento de la calidad

El aseguramiento de la calidad es la parte de la gestión de calidad que se enfoca en proveer confianza de que los requerimientos de calidad serán alcanzados (BS ISO 9000:2000).

Consta del diseño y planificación de acciones sistemáticas que se requieren para asegurar dicha calidad de software.

Las **visiones** permiten definir **modelos de calidad** (conjunto de características). Estos modelos permiten definir **propiedades** del producto y del proceso. En base a esas **propiedades** definimos **métricas** (indicadores), y en base a las métricas realizamos la **gestión de calidad** para asegurarla.

La **calidad de software** está fuertemente determinada por la **calidad del proceso** usado para desarrollarlo y mantenerlo, sin embargo la calidad del proceso no garantiza la calidad del producto. El aseguramiento de la calidad del producto necesita ser parte del proceso de desarrollo y mantenimiento. Las características de calidad necesitan ser localizadas y verificadas en los productos intermedios del proceso de desarrollo.

Rol del testing dentro del Proceso de Aseguramiento de Calidad

El testing no mejora directamente la calidad del software, él se plantea como un medio para determinar el nivel de calidad del software que se prueba, ayudando a encontrar defectos y aumentando la confianza al otorgar más información.

Un programa de aseguramiento de la calidad o de mejora continua, ayuda a aprender de esos errores y tomar medidas para que no se repitan en el futuro.

El testing de software **no puede asegurar que el software está libre de defectos**.

Actualmente, el proceso de aseguramiento de calidad se compone mayormente de testing.

“El propósito del testing es encontrar defectos

Encontrar defectos, destruye la ***confianza***

Entonces

El propósito del testing es destruir la confianza

Pero

Otro propósito del testing es dar ***confianza***

Por lo que

La mejor forma de dar confianza, es destruyéndola*”*

Los procesos en la historia

A lo largo de la historia del desarrollo tecnológico, y en particular del software, se han generado diversas maneras de llegar a la concreción de proyectos.

Mientras se diseminaba la novedad de la informática y exigía cada vez mayores esfuerzos de trabajo por su crecimiento e inserción en el mundo, aparecieron formalizaciones en las maneras de producir o crear software para enfrentar las demandas y problemas.

Así hacen aparición las metodologías o procesos de desarrollo de Software, buscando organizar y mejorar las prácticas y los resultados.

Si bien estos procesos han dado una significativa mejora y un marco dentro del cual surgieron conceptos y pudieron adoptarse otros, el desarrollo de software sigue teniendo sus matices, con procesos y herramientas diversos que continúan también un camino propio.

Detallaremos los procesos más destacados, centrándonos en el rol que juega el **testing** dentro del mismo:

Desarrollo en cascada

El modelo o proceso en cascada define una serie de fases a seguir que conforman las tareas necesarias para dar vida a un software. Estas fases son independientes en la teoría y para comenzar con una debe finalizarse la anterior.

Las fases son:

1. Especificación de requisitos y análisis
2. Diseño y arquitectura del software
3. Construcción o implementación
4. Testing
5. Mantenimiento

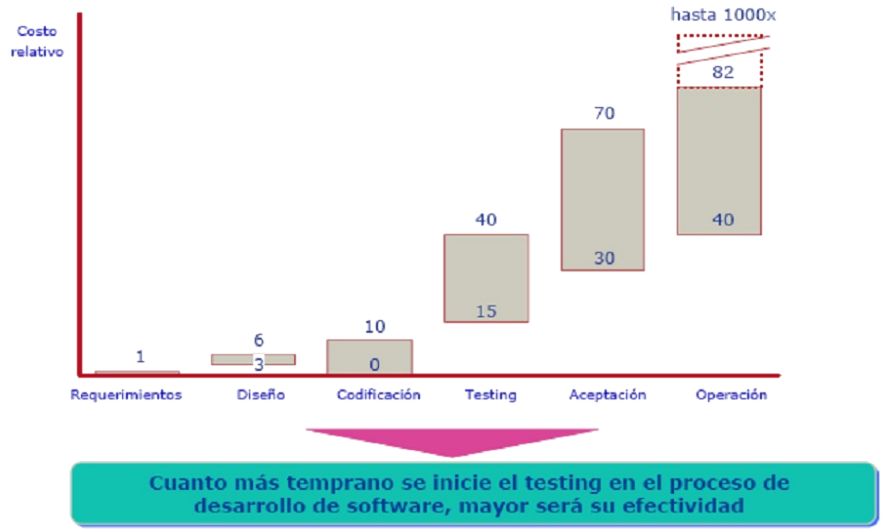
Este planteo implica que no se comenzarán las tareas de prueba

hasta no tener completo el producto en su totalidad. A su vez, no se comenzará a construir el software hasta que el análisis haya sido "exhaustivo", y así.

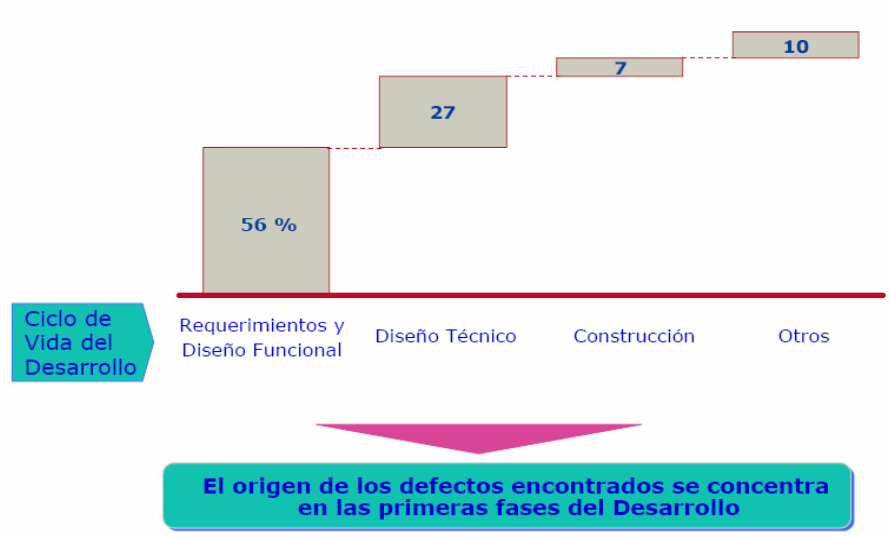
El problema aquí recae en los costos y complicaciones que genera la metodología. Si encontramos un error en la fase de codificación y es realmente un error de especificación de requisitos, eso implica que se han surcado todas las fases arrastrando un error y el único momento de encontrarlo y repararlos es sobre la etapa final de testing, generando una carga de re-trabajo más que considerable. Caso similar sería el de las modificaciones que pueden llegar a surgir en la especificación, luego de que esta etapa haya sido cerrada, dando como resultado avances sin validaciones y pérdidas de tiempo y dinero.

El problema central en esta metodología, como se expresa, es el **costo**. Hablamos de *costo relativo* refiriéndonos a lo que necesitará gastarse en tiempo, dinero o esfuerzo para reparar o corregir una falla según el momento en el que se ha detectado.

Estas relaciones alcanzan una evolución exponencial a lo largo del ciclo del proceso en cascada:



Es interesante también saber que las principales fuentes de generación de defectos se encuentran en las etapas tempranas del proceso, antes que en las etapas finales. Para verlo gráficamente:



Por último, vemos un interesante efecto que suele generarse dentro del proceso, que es el de amplificación de los defectos. Esta amplificación se da por una naturaleza propia del arrastre de los errores a medida que se avanza en el tiempo y de la aparición de nuevos propios de cada fase particular.

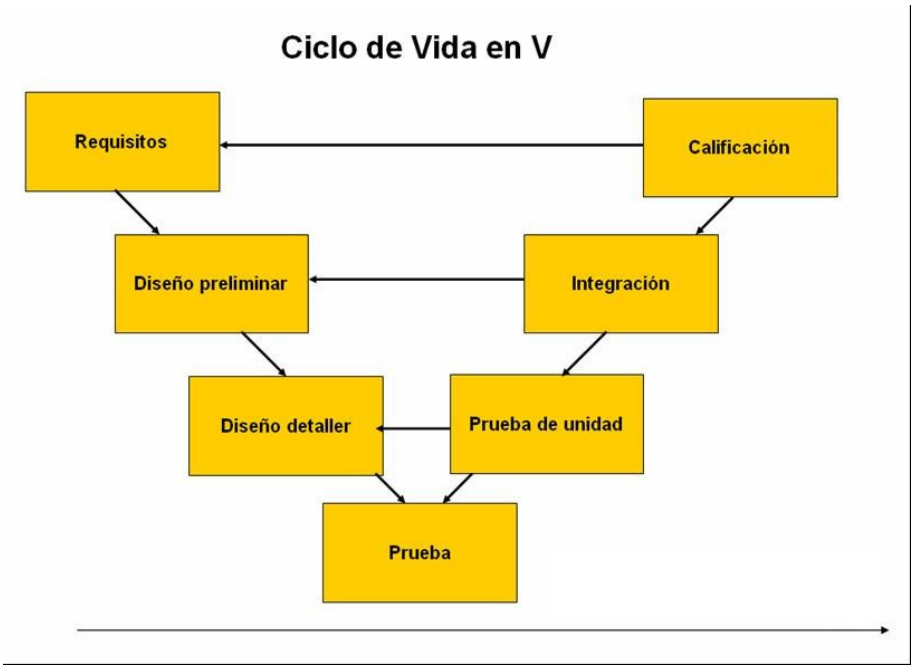
Desarrollo en V

El desarrollo en V implica una descripción de actividades y resultados que deben producirse durante el desarrollo del producto, identificando un lado “izquierdo” de la V y un lado “derecho”. El lado izquierdo representa la descomposición de las necesidades, y la creación de las especificaciones del sistema. El lado derecho de la **V** representa la integración de las piezas y su verificación.

Se utiliza también la **V** para dar significado de «Verificación y validación». Existen muchas similitudes con el modelo en cascada clásico por su rigidez y gran cantidad de iteraciones, pero se diferencia ampliamente en que este modelo introduce el uso del principio de **testing temprano.**

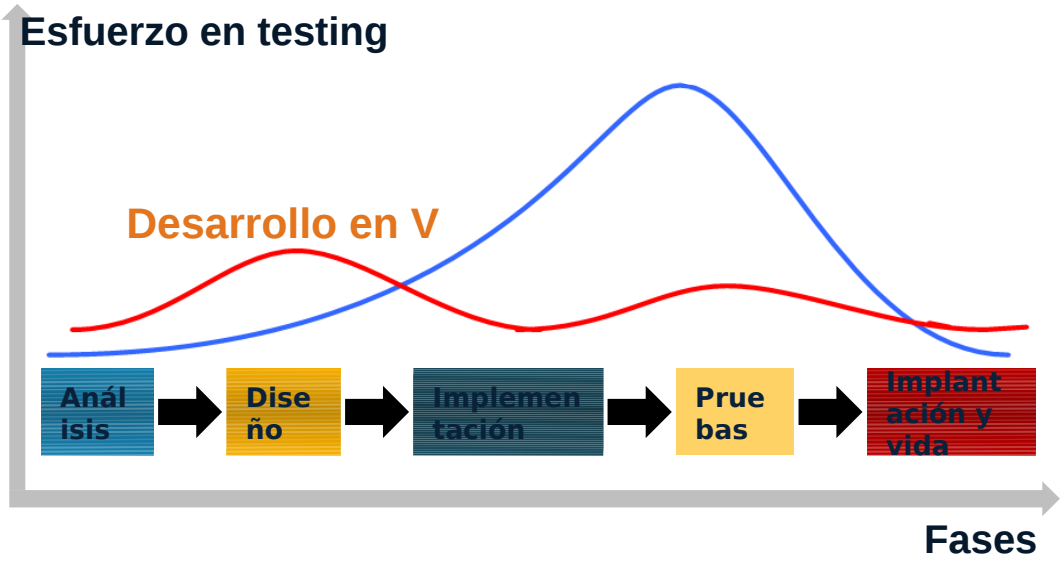
Posee una facilidad práctica para entenderlo, donde existen niveles que muestran la conexión entre las actividades de desarrollo y su correspondiente de testing, o que se encargaría de cubrir en cierta forma lo realizado.

Se va probando en paralelo a medida que se avanza en la rama izquierda, haciendo que la rama derecha vaya ejecutando las pruebas variando sus **niveles** según la fase en la que nos encontremos:



Ventajas principales:

- No es necesario esperar hasta que las fases del producto estén completas para iniciar el diseño de las pruebas
- Esto permite realizar correcciones de defectos antes de que sean trasladados al siguiente nivel



Rup

El RUP es el Proceso Unificado de Rational, creado por la empresa Rational Software adquirida por IBM, que junto con el lenguaje de modelado UML constituye la metodología estándar

más usada para el análisis, diseño, implementación y documentación de sistemas orientados a objetos, al menos durante fines de los años 90 y principios del 2000.

Se basa en un conjunto de metodologías adaptables al contexto y necesidades de cada organización, teniendo seis principios clave:

1. Adaptar el proceso
2. Equilibrar prioridades
3. Demostrar valor iterativamente
4. Colaboración entre equipos
5. Elevar el nivel de abstracción
6. Enfocarse en la calidad

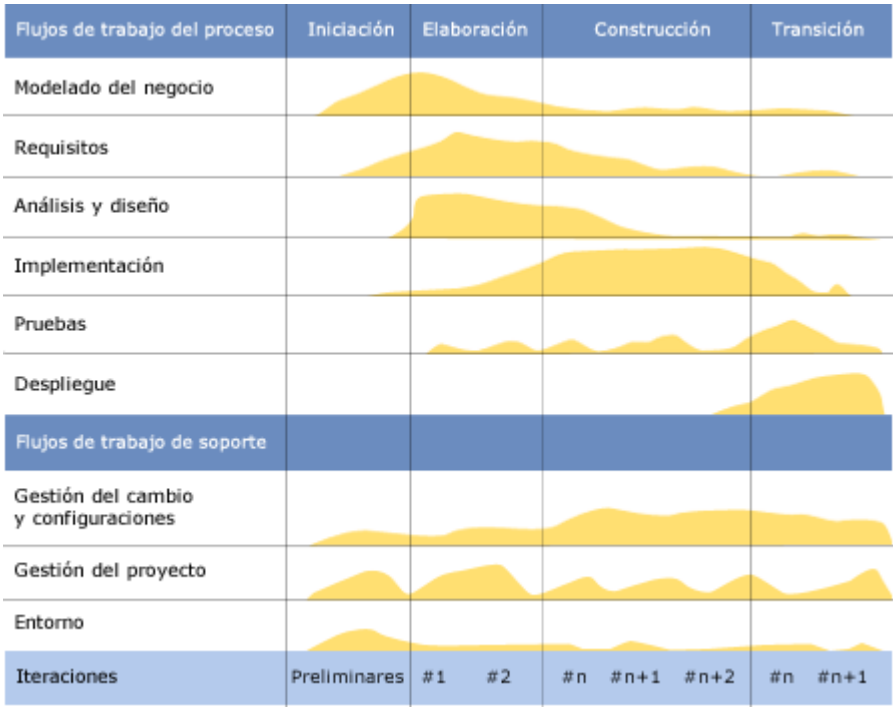
Es una actividad llevada a cabo a través de todo el ciclo iterativo de desarrollo donde cada iteración tiene una misión o meta diferente, tiene un carácter exploratorio, porque sus especificaciones tienden a cambiar con frecuencia.

Toma como base de pruebas no sólo las especificaciones, sino una colección de fuentes diversas, incluso no documentadas, pero debe evitar producir más documentación de la estrictamente necesaria.

El plan de prueba y un plan detallado de esfuerzo de prueba debería ser todo lo producido antes de la ejecución de pruebas.

Su ciclo de vida identifica fases pero no las limita a una dependencia estricta del resto, pudiendo dar inicio, por ejemplo, al testing en un momento temprano.

Paralelamente cada fase se va desarrollando e iterando sobre si misma, alimentando el avance y las tareas de las otras fases. Lo vemos gráficamente:



Existen 3 conceptos claves que dominan la metodología:

1. Dirigido por casos de uso
2. Centrado en la arquitectura
3. Iterativo e incremental

En lo referente a dirigido por los casos de uso, significa que los requerimientos están enfocado a dar valor al cliente y que el proceso debe garantizar que todo el desarrollo, pruebas, planificación, documentación etc, está orientado a cubrir estas expectativas del cliente y asegurar que los requerimientos de valor se ponen en producción.

En lo referente a centrado en arquitectura, significa que hay un énfasis a diseñar una arquitectura de calidad, y es la arquitectura también la que guía la forma cómo se debe planear y hacer el desarrollo.

En lo referente a iterativo e incremental, significa que el proyecto se divide en varios ciclos de vida (llamadas iteraciones) que deben dar como resultado un ejecutable. Por cada una de las iteraciones se va agregando requerimientos y sobre todo valor al cliente; por este motivo es incremental.

Ágiles

Los “ágiles” son métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requerimientos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios.

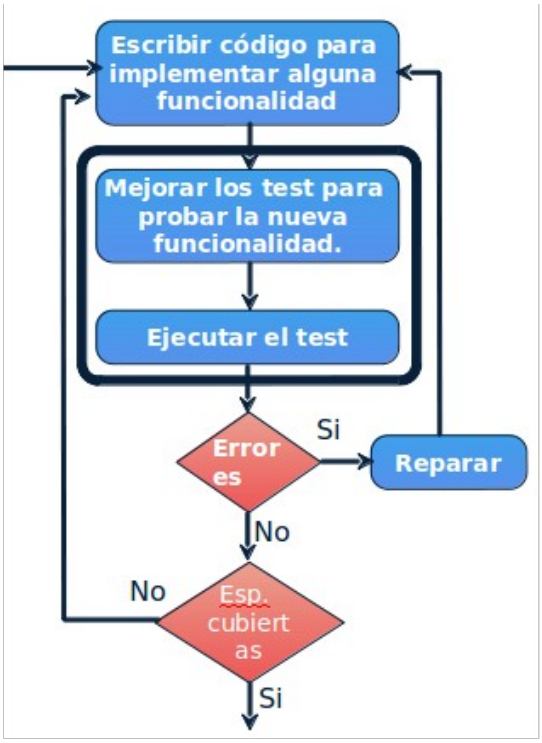
Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en lapsos cortos. El software desarrollado en una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Éstas no deben agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, sino que la meta es tener una demo (o producto entregable, sin errores) al final de cada iteración. Luego de eso el equipo vuelve a evaluar las prioridades del proyecto.

Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación. La mayoría de los equipos ágiles están localizados en una simple oficina abierta. Se define al software funcional como la primer medida del progreso.

Combinado con la preferencia por las comunicaciones cara a cara, generalmente los métodos ágiles son criticados y tratados como "indisciplinados" por la falta de documentación técnica.

Dentro de estas metodologías el testing está presente en varias fases del proceso de desarrollo, principalmente como parte de la codificación. Se realiza el testing como parte de la implementación de cada funcionalidad.

Gráficamente:



Desarrollo guiado por tests (TDD)

De las siglas en inglés *Testing Driven Development*, esta metodología plantea centrar el desarrollo en las pruebas. Y no sólo eso, si no que pretende crear y ejecutar primero las pruebas antes que el código que ellas probarán.

¿Tiene esto sentido?

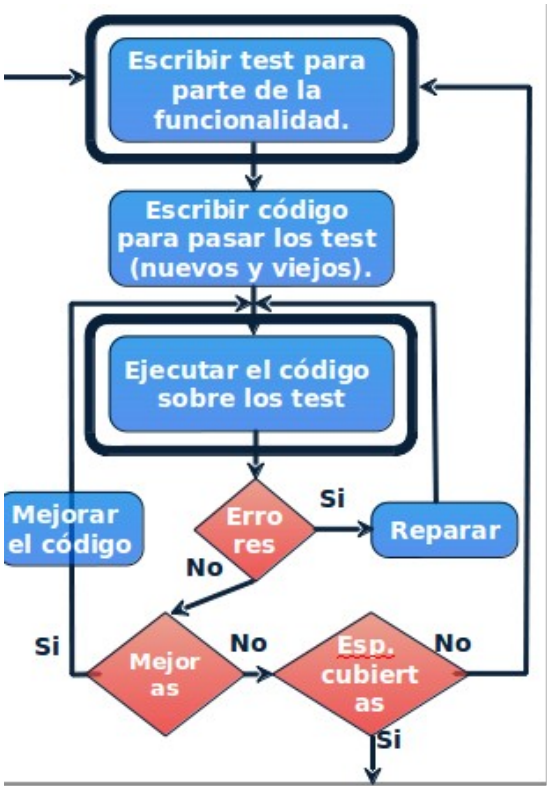
A un primer vistazo parece no tenerlo, pero una vez incorporada la metodología y practicada, cobra importante relevancia.

La intención es generar primero un requerimiento, del que se parte para generar una serie de casos de prueba, los suficientes como para asegurarnos una cobertura plena de lo que las funcionalidades deberían responder y abarcar. Luego de esto, el desarrollador toma un requerimiento, selecciona los casos de prueba correspondientes y comienza a escribir su prueba unitaria, la porción de código que se ejecutará automáticamente para probar el código funcional del sistema, desde lo más general hacia lo más particular. Paso siguiente es ejecutar esa prueba unitaria y, como es de esperarse, fallará al no tener una implementación del código. Esa falla, nos dictamina el *pequeño paso* que debemos dar en codificación, guiando al desarrollo a través de la prueba. Se construye la solución, también general, para que esa prueba sea exitosa y se vuelve a correr el test, verificando su éxito. Es este el momento de la refactorización, que será la que determine el nivel de cobertura que se tendrá sobre las pruebas y el nivel de reutilización que pueda tener lo que estamos construyendo.

Resumiendo los pasos:

1. Elegir un requisitos
2. Escribir una prueba
3. Verificar que la prueba falla
4. Escribir la implementación
5. Ejecutar las pruebas automatizadas
6. Eliminación de la duplicación o refactorización
7. Actualizar la lista de requerimientos

Gráficamente:



¿Cómo probar el software?

Metodología de testing

Pruebas no sistemáticas

Son pruebas habitualmente realizadas “ad-hoc”, sin una planificación previa que guíe sus pasos u oriente sus objetivos, siendo desarrolladas en un proceso caótico y sin posibilidad de repetición. Al no estar sistematizadas, sus resultados no pueden utilizarse para definir métricas y el momento para realizar este tipo

de pruebas es una vez finalizado el desarrollo.

Esto genera un testing incompleto, sin recabar información ni planificar su cobertura, dando como consecuencias:

- Que no haya una adecuada identificación de los requerimientos
- Que haya una indefinición respecto a la completitud de la prueba y su cobertura
- Un desconocimiento completo de la tasa actual de defectos
- Imposibilidad para decidir cuándo liberar el producto, o cómo hacerlo

Su objetivo dominante es demostrar que el sistema funciona, lo que se conoce como "camino feliz".

**** EJEMPLO DE PRUEBA NO SISTEMÁTICA ****

Pruebas sistemáticas

A contraposición de las pruebas NO sistemáticas, están las pruebas sistemáticas, que se desarrollan en base a una metodología formal, con las siguientes características:

- Son fácilmente repetibles, en múltiples circunstancias
- Se provee un encuadre formal para todas las actividades que las integran
- Incluyen actividades de seguimiento
- Se ejecutan en paralelo al ciclo de desarrollo, empezando lo más temprano posible

El objetivo de las pruebas sistemáticas se focaliza en la detección e identificación de defectos del sistema.

**** EJEMPLO DE PRUEBA SISTEMÁTICA ****

El rol del tester

Estas pruebas exigen ciertas "posiciones" o cualidades que deben poseer quienes realizan las pruebas:

- **Independencia:** el tester debe estar comprometido con la búsqueda de errores y no con la defensa del producto.
- **Idoneidad:** el tester debe ser un profesional preparado en las funciones de probar y con los conocimientos necesarios de la metodología de pruebas, sus técnicas y estrategias.

Existen dos encargados de realizar las pruebas del software inicialmente:

- **Desarrollador:** él entiende el sistema pero está dirigido por entregables y analizando una tarea propia o de un colega, sin necesidad de tener entendimiento pleno del propósito que el usuario tiene sobre el sistema.
- **Tester:** debe entender el sistema pero intentará provocar

fallas y está dirigido por la calidad.

El testing es una disciplina profesional que requiere personas capacitadas y entrenadas para realizarlo.

Tipos de prueba

En cuanto a la tipificación de las pruebas que pueden realizarse al software encontramos dos grandes categorías principales dentro de las que se identifican otras tipificaciones más particulares.

Estas son la de pruebas **funcionales** y pruebas **no funcionales**.

Pruebas Funcionales

Cuando hablamos de pruebas funcionales estamos apuntando a que lo que se probará del software, es funcionalidad; es decir, su capacidad de responder a requerimientos relacionados con la operatoria del sistema, con los intereses del usuario estrictamente relacionado a la información que él necesita procesar u obtener.

Apuntan a verificar que los comportamientos específicos o funciones del software cumplan con las necesidades, establecidas e implícitas, cuando el software es utilizado en condiciones normales.

Para confeccionarlas se atiende como base al análisis de la especificación de funcionalidad del componente o sistema; hablamos aquí de requerimientos funcionales determinando una relación entre ellos y las pruebas funcionales.

Estas pruebas consisten en accionar, aun de forma inválida, el objeto de prueba y verificar que las respuestas obtenidas del mismo sean las esperadas, evaluando las siguientes características:

- **Adecuación:** que el objeto provea el conjunto apropiado de funcionalidad para la tarea especificada y los objetivos de su usuario.
- **Precisión:** que las respuestas esperadas sean correctas y cumplan con el grado de precisión acordado.
- **Interoperabilidad:** que pueda interactuar con uno o más componentes, sistemas o entornos.
- **Seguridad:** que los datos/programas están protegidos contra accesos no deseados o pérdida de información.
- **Conformidad:** que cumpla con normas establecidas.

**** PROBAMOS FUNCIONALIDAD ****

Pruebas No funcionales

Las pruebas no funcionales funcionan en otro sentido, como su nombre lo indica. Éstas se centran y operan sobre requerimientos

no funcionales, es a ellos que intentará probar. Estos requerimientos especifican un criterio para juzgar la operación del sistema como un todo, y no un comportamiento específico. Dejarán de lado la funcionalidad y lo que el usuario espera en cuanto a información y salidas del sistema y se centrarán en las capacidades de rendimiento, de cómputo, de recursos, etc.

La división de la clasificación es aquí mayor que en las pruebas funcionales y encontramos los siguientes tipos de pruebas no funcionales:

Fiabilidad

Estas pruebas tienen por objetivo medir la madurez del software a través del tiempo y compararla con la confiabilidad deseada. Al tomar métricas se utiliza mucho el tiempo medio entre fallas (MTBF) y tiempo medio de reparación (MTTR). Su importancia o significado toma un tiempo en hacerse visible y claro, debido a que dependen fuertemente del tiempo que lleve el software funcionando, y por lo general se continúan trabajando en producción.

Encontramos también estas acciones para probar fiabilidad:

- **Pruebas de robustez o tolerancia a fallas:** Orientadas a evaluar la tolerancia de un componente o sistema a fallas externas.
- **Pruebas de recuperación:** Orientadas a evaluar la habilidad del sistema de software a recuperarse de fallas en hardware o software de modo que permita reanudar las operaciones normales.

Rendimiento

Éstas están enfocadas en la habilidad del componente o sistema de responder dentro del tiempo establecido y bajo las condiciones establecidas.

Sus medidas pueden variar desde ciclos de CPU hasta tiempos de respuesta.

En el rendimiento habrá que tenerse en cuenta la arquitectura donde se está desplegando el software y la tecnología utilizada para desarrollarlo, donde ambas deberían ir en sintonía con los requerimientos iniciales del sistema. Esta sintonía debería determinar los tiempos de respuestas esperados y posibles para que luego mediante pruebas de rendimiento se verifiquen y pueda certificarse si el rendimiento es exitoso o no.

Carga

Las pruebas de carga están enfocadas en la habilidad del sistema para manejar niveles crecientes de carga real y previsible, resultante de pedidos de transacciones generadas por usuarios en simultáneo.

Las medidas de estas pruebas suelen realizarse sobre tiempo de respuesta promedio y tasa de transferencia de información.

Existen varias estrategias para llevar adelante este tipo de pruebas, que buscan verificar la capacidad de respuesta del software a un acceso simultáneo y transaccional “pesado”, con niveles de tolerancia definidos en acuerdo con el usuario y los encargados técnicos:

- Producir fuertes cargas de trabajo como tráfico excesivo o cargas elevadas de transacciones simultáneas
- Generar tráfico de red artificial
- Acceder con múltiples usuarios simultáneos
- Generar transacciones con herramientas de automatización

Volumen

Éstas pruebas se orientan a verificar la capacidad del sistema de procesar grandes cantidades de datos y/o archivos manteniendo su normal funcionamiento.

Como la mayoría de las pruebas no funcionales esta también exige un acuerdo entre quien solicita el software y el equipo técnico que dará respuesta, ya que existen diversos factores que limitan la capacidad de procesamiento de grandes volúmenes de datos.

Estrés

Estas pruebas están enfocadas en la habilidad del sistema para manejar picos de carga en el límite o superiores a su capacidad máxima.

Es esperable que el sistema empiece a degradarse lentamente y de forma predecible, sin producir fallas a medida que se aumentan los niveles de estrés.

El objetivo de estas pruebas es verificar que el sistema no pierda su integridad funcional mientras se encuentra en estas condiciones de estrés.

Permiten lo siguiente:

- Encontrar fallas debido a escasez de recursos
  - Falta de memoria
  - Falta de espacio en disco
- Encontrar errores debido a la existencia de recursos compartidos
  - Recursos del sistema
  - Bloqueos de la base de datos
  - Ancho de banda de la red

Portabilidad

La portabilidad hace referencia a la facilidad con la que el software puede ser transferido y desplegado al ambiente

pretendido, las pruebas de este tipo se centrarán en verificar esa posibilidad:

- **Pruebas de coexistencia:** evalúan la capacidad del sistema para coexistir con otros sistemas dentro del mismo ambiente sin afectar su comportamiento.
- **Pruebas de instalación:** evalúan la capacidad para instalar el software en el ambiente pretendido.
- **Pruebas de adaptabilidad:** evalúan la capacidad del sistema para funcionar correctamente en todos los ambientes objetivo.
- **Pruebas de reemplazo:** se enfocan en la capacidad para intercambiar componentes de software del sistema por otros.

Usabilidad

Las pruebas de usabilidad realizan una medición de cómo el software se adecúa a sus usuarios. Esa medición incluye evaluar las siguientes capacidades del producto dentro de un contexto específico de uso:

- **Efectividad:** de permitir a los usuarios alcanzar sus metas
- **Eficiencia:** de permitir a los usuarios utilizar una cantidad apropiada de recursos en relación con la efectividad alcanzada
- **Satisfacción:** de satisfacer a sus usuarios
- **Entendimiento:** aquellos que afecten el esfuerzo requerido para reconocer el concepto lógico y su aplicabilidad
- **Aprendizaje:** aquellos que afectan el esfuerzo requerido del usuario para aprender la aplicación
- **Operabilidad:** aquellos que afectan el esfuerzo requerido del usuario para realizar sus tareas efectiva y eficientemente
- **Atracción:** la capacidad de que el usuario guste del producto

Mantenimiento

Las pruebas de mantenimiento buscan probar y recabar información sobre las capacidades del software de ser mantenido con un bajo costo y una rápida respuesta.

Existen dos categorías de pruebas de mantenimiento conocidas como:

- **Mantenimiento correctivo**
  - **Analizabilidad:** enfocadas en medir el esfuerzo requerido para diagnosticar y corregir problemas identificados en el sistema
- **Mantenimiento adaptativo**

- **Modificabilidad:** enfocadas en medir el esfuerzo requerido para realizar un cambio al sistema
- **Estabilidad:** enfocadas en evaluar la respuesta del sistema al cambio
- **Testeabilidad:** enfocadas en medir el esfuerzo requerido para probar los cambios realizados

Escalabilidad

Estas pruebas están enfocadas en la habilidad del sistema de cumplir con futuros requerimientos de eficiencia, que suelen estar por encima de los actualmente exigidos.

Utilización de recursos

Enfocadas en evaluar el uso de recursos (espacio en memoria, capacidad de disco, ancho de banda de red) del sistema en diferentes circunstancias.

Casos de prueba

Un caso de prueba es un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y poscondiciones de ejecución, desarrollados para un objetivo específico, tal como ejercitar un camino particular de un programa o verificar el cumplimiento de un requerimiento puntual.

Requerimientos de prueba: es un aspecto, un evento, un componente o un sistema que puede ser verificado por uno o más casos de pruebas.

Un caso de prueba, entonces, consta de:

- Una precondición
- Datos de entrada
- Resultados esperados
- Criterio para el veredicto “pasa – falla”
- Una poscondición

Precondición

Son las condiciones de entorno y de estado, que deben ser cumplidas antes de que el componente o sistema puede ser ejecutado con un valor particular.

Se presenta como la situación previa a la ejecución de la prueba o como la característica de un objeto de prueba antes de la ejecución de una prueba.

Datos de prueba: son los datos que deben existir antes de que una prueba pueda ser ejecutada, y que afectan o son afectados por el componente o sistema bajo prueba. Un ejemplo de esto puede ser una base de datos que contenga instancias específicas de una entidad.

Entorno de prueba: son los diferentes elementos del objeto de prueba que son necesarios para conducir la prueba, detallando sus configuraciones. Ejemplos de esto son el hardware necesario, herramientas de testing, etc

Entrada

La entrada es el dato o estímulo recibido por un objeto a prueba durante la ejecución de la prueba desde una fuente externa. Esta fuente puede ser tanto hardware, como software o una acción humana.

Resultado esperado

Es el comportamiento predicho por la especificación u otra fuente, bajo condiciones específicas. Necesita también especificar cómo se obtendrá dicho resultado esperado.

Oráculo: se llama así a una fuente para determinar resultados esperados de un caso de prueba, pudiendo esta ser un sistema existente, un usuario manual o el conocimiento especializado de un individuo. No debería ser nunca todo o parte del código probado. Ejemplos son la simulación, la utilización de una hoja de cálculo, experto en el dominio, etc.

Veredicto "pasa – falla"

Son las reglas de decisión usadas para determinar si un ítem de prueba ha pasado o fallado la prueba en base al **resultado esperado**. Es también el comportamiento producido/observado del objeto de prueba como resultado del procesamiento de las entradas del caso. Este veredicto no es parte del caso de prueba, si no que es el resultado de la ejecución de la prueba sobre una versión específica del objeto a prueba.

Poscondición

Son las condiciones del entorno y estado que se deben cumplir después de la ejecución de una prueba o un caso de prueba, o un procedimiento de prueba.

**** EJEMPLO CLÁSICO DE CASO DE PRUEBA ****

Niveles de casos de prueba

A su vez, dentro de los casos de prueba, encontramos de distintos niveles:

- **Casos de prueba de alto nivel:** es un caso de prueba que no tiene una implementación concreta, o valores concretos para la entrada de datos y el resultado esperado. Es también conocido como "Caso de prueba lógico".
- **Casos de prueba de bajo nivel:** Es un caso de prueba con valores concretos, que existen a nivel de implementación, para datos de entrada y resultado esperado. Los operadores lógicos de los casos de prueba de

alto nivel son reemplazados por valores reales que corresponden al objetivo de los operadores lógicos. Es también conocido como “Caso de prueba físico”.

Especificación de un caso de prueba

A la hora de realizar la especificación o confección de un caso de prueba nos enfrentamos a un conjunto de preguntas a responder para la tarea: **¿qué tan preciso tienen que estar especificadas las acciones, los datos, el resultado esperado, etc. de un caso de prueba?**

Ésta es una decisión por demás importante que debe tomarse cuando se piensa en documentar los casos de prueba.

El resultado de un buen nivel de detalle en la especificación de un caso de prueba puede resumirse en:

- Una mejora de la capacidad de reproducir fallas, ya que nada es dejado al juicio de un tester en particular
- La posibilidad de que los casos de pruebas sean revisados por el área de desarrollo
- Una alta carga de trabajo a la hora de la especificación
- La posibilidad de que testers no expertos ejecuten los casos de pruebas sin una capacitación tan fuerte

Características deseables de un caso de prueba

Para poder hablar de un buen caso de prueba, completo y efectivo, éste debería:

- Ser representativo
  - Recordemos que el testing exhaustivo no es factible, con lo que esto exige que el caso de prueba sea representativo de un conjunto grande de ejecuciones
- Ayudar a identificar fácilmente cuál es el defecto
- Ser fácil de repetir
  - Se debería tener un punto de inicio y de fin preciso y el estado del sistema la momento de introducirse la entrada debería ser conocido
  - Se debería conocer exactamente qué datos de entrada fueron introducidos y en qué orden
- Ser fácil de usar, entender y de ejecutar
- Ser fácil de mantener
- Ser trazable

Así podemos definir que una prueba será **exitosa** en los siguientes casos:

- Cuando detecta un defecto aún no descubierto
- Cuando encuentra muchos defectos

- Se comprueba que de alguna manera el producto posee defectos, lo que de ser contrario, aseguraría el fracaso del área de testing

Definimos también lo que sería un **conjunto de pruebas ideal** de la siguiente manera:

- Es el menor subconjunto de todos los casos de prueba posibles que encuentra todos los defectos sobre un componente o sistema
- En general existe una imposibilidad de definir un conjunto de casos de prueba ideal, pero se intenta aproximar mediante el uso de *criterios de selección* de pruebas
- El criterio de selección de pruebas intenta aproximar esta noción, eligiendo el subconjunto de comportamientos a probar

Terminología básica

- **Prueba o conjunto de pruebas:** es un conjunto de uno o más casos de prueba
- **Suite de prueba:** es un conjunto de varios casos de prueba para un componente o sistema bajo prueba, donde la poscondición de una prueba es frecuentemente usada como la precondición del próximo caso de prueba. Se usa para agrupar casos de prueba similares
- **Procedimiento de prueba o script:** es un documento que especifica una secuencia de acciones para la ejecución de una prueba o una suite de pruebas. Contiene acciones, verificaciones y casos de pruebas relacionados e indica la secuencia de ejecución. Se utiliza generalmente para un procedimiento automatizado

Niveles de prueba

Según el nivel de abstracción de las pruebas, encontramos que ellas pueden clasificarse de la siguiente forma:

Pruebas de componentes

Éstas pruebas son las encargadas de realizar el test sobre componentes de software de manera individual o independiente, buscando asegurar que el componente funciona como está especificado. Son conocidas también como “Pruebas unitarias”

Llamamos *componente* a un ítem de software para el cual existe una especificación.

Para este tipo de pruebas normalmente se utilizan frameworks automatizados del tipo *Xunit*.

Pruebas de integración

Las pruebas de integración procuran exponer defectos entre las interfaces de los componentes y en su interacción. Implica un nivel "superior" de prueba, haciendo que, una vez probados los componentes individuales, se verifiquen las interacciones entre ellos e intenten detectar fallas que las pruebas de componentes no han podido encontrar.

Pruebas de sistema

Las pruebas de sistema verifican que el sistema integrado cumple con los requerimientos especificados. Intentarán detectar fallas de funcionalidad o desvío de requerimientos una vez las pruebas de componentes y de integración hayan alcanzado un nivel de madurez aceptable para este punto.

Pruebas de aceptación

Estas pruebas aseguran que el sistema cumpla con los requerimientos de "negocio" y, consecuentemente, que la lógica del mismo funcione correctamente.

Buenas prácticas para realizar estas pruebas son:

- Realizar una planificación
- Involucrar al usuario desde un comienzo de las pruebas
- Realizar versiones de Alpha y Beta testing
- Utilizar contratos de aceptación

Retesting y pruebas de regresión

Retesting

El retesting consta del siguiente proceso:

1. Se ejecuta una prueba y falla
2. Se reporta el defecto
3. El área de testing recibe una nueva versión del software con el defecto corregido
4. Se re-ejecuta la misma prueba, en el mismo entorno, las mismas entradas, las mismas precondiciones
5. Si la prueba pasa, el defecto ha sido corregido correctamente

No podemos asegurar que el sistema no tiene nuevos defectos.

Pruebas de regresión

Las pruebas de regresión son una estrategia de pruebas que

implica la repetición total de casos de prueba diseñados para una aplicación, a medida que se avanza en la prueba de nuevos módulos.

La estrategia implica el reconocimiento de que la corrección de defectos detectados en una aplicación, puede conllevar la aparición de errores de modo indirecto y en áreas supuestamente libre de errores.

Derivación de pruebas

La derivación de pruebas implica generar pruebas para medir la calidad y encontrar defectos en un producto, proceso o herramienta a partir de material ya existente del mismo.

Derivación de casos de prueba

La derivación de casos de prueba es una actividad que debe realizarse de manera específica dentro del proceso de desarrollo de software. Su base está en la documentación del proyecto, que depende de la política particular de cada organización en cuanto a la documentación y la metodología que se utilice para el desarrollo.

La derivación llevará a la detección de defectos en la documentación en la que se basa.

Para comenzar a realizarla partimos de una **base de pruebas** conformada por:

- Documentos del cliente
- Documentos de relevamiento
- Casos de uso
- Especificaciones de programación
- Código
- Otras

Criterios de selección y técnicas de diseño

Criterios de selección

Éste es un mecanismo para decidir si una prueba es adecuada o no frente a las necesidades que existan. De un criterio de selección se espera que éste sea:

- **Regular:** si todas las pruebas que satisfacen el criterio detectan en general los mismos errores
- **Válido:** si para cualquier error en el programa hay un conjunto de pruebas que satisfacen el criterio y detectan el error

Conseguir criterios con buenas características de regularidad y validez suele ser una tarea compleja.

Los criterios surgen a partir de entender que la forma más efectiva de hacer testing sería a través de una ejecución de pruebas exhaustivas, pero sabemos que esto es impracticable y es allí donde los criterios dan su aporte, son necesarios para realizar la selección de las pruebas.

El proceso guiado por un criterio identifica un conjunto de pruebas (clase) que es representativo de todas las pruebas posibles, haciendo que, por ejemplo, si dos pruebas encontrasen el mismo defecto, éstas deberían pertenecer a la misma clase y el producto bajo prueba se comportará de la misma manera para todas las de la misma clase.

Generalmente la definición de criterios la haremos una vez que los casos de prueba estén identificados y escritos, para luego ser agrupados en clases que respondan a los criterios de selección considerados.

Técnicas de diseño

Una técnica de diseño de pruebas es un método estandarizado para derivar, de una base de prueba específica, casos de prueba que realicen una cobertura específica.

Dentro de sus características encontramos que son fundamentales para el desarrollo del testing metodológico y profesional, facilitan la comprensión de la calidad y la cobertura de las pruebas, existen diferentes técnicas enfocadas a encontrar diferentes tipos de fallas y que, utilizando una combinación de técnicas, es posible prevenir y detectar fallas de una manera más eficiente.

Tiene como ventaja los siguientes puntos:

- Permiten una implementación efectiva de una estrategia de prueba
- Los defectos relevantes son detectados con mayor eficiencia
- Los casos de prueba resultan más fácilmente reproducibles
- Proveen independencia respecto de los ejecutables
- Aseguran que los casos de prueba sean mantenibles
- Permiten dividir la especificación de las pruebas de la ejecución de las mismas, haciendo el proceso de prueba más fácil de planificar y administrar

Técnicas de caja negra

Las técnicas de caja negra están basadas en la definición de requerimientos o de una descripción funcional del producto bajo prueba y se centran en *“qué hace”* el programa y no en *“cómo lo hace”*. Es por eso que son llamadas “de caja negra”, porque no observan el comportamiento interno y generalmente las pruebas se conducen a nivel de interfaz de usuario.

Encontramos las siguientes técnicas de caja negra para la

derivación de casos de prueba:

Casos de uso

La técnica pide crear al menos un caso de prueba para el escenario exitoso y uno para cada alternativo permitiendo definir una medida de cobertura de las pruebas.

El problema es que un mismo escenario puede ser accionado con más de una combinación de entradas haciendo que muchas de las combinaciones posibles queden sin probar. Hay que prestar especial atención también al cuidado de no sobreestimar la calidad del sistema.

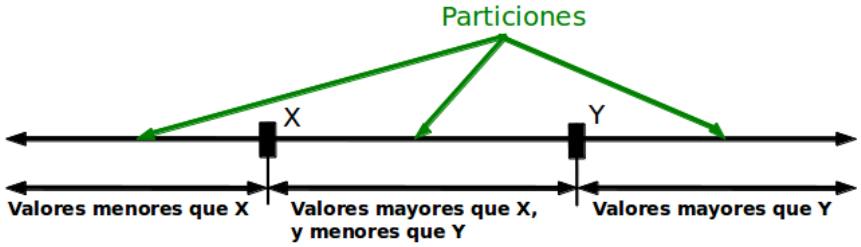
**** EJEMPLO DE CASO DE USO ****

Particiones de equivalencia

Esta técnica usa un modelo del componente que divide los valores de entrada y de salida en particiones de equivalencia. Esas particiones deben corresponder a casos similares, en los que el producto bajo prueba se comporta de la misma manera.

Entiende que si el producto funciona correctamente para una prueba en una partición, se supone que lo hará para el resto de las pruebas de la misma partición.

Ejemplo de entrada numérica:



Deben considerarse las siguientes entradas:

- **Válidas:** que NO deberían ser rechazadas por el componente o sistema
- **Inválidas:** que deberían ser rechazadas por el componente o sistema

Las instrucciones para ejecutar la técnica son:

1. Identificar las particiones
    1. Si la entrada es sobre un rango de valores
        1. una partición válida para valores dentro del rango
        2. dos inválidas para cada lado fuera del rango
    2. Si la entrada es sobre un conjunto de valores
        1. una partición válida para valores dentro del conjunto
        2. una partición inválida para valores fuera del conjunto

3. Asignar un número único a cada partición de equivalencia

2. Diseñar casos de prueba que cubran la mayor cantidad de particiones válidas

3. Diseñar un caso de prueba para cada clase inválida

    1. Si probamos múltiples clases inválidas con el mismo caso, el rechazo de una clase inválida puede enmascarar una falla en el rechazo de las otras entradas inválidas

    2. Existen dos aproximaciones:

        1. Mantener los mismos valores válidos para todas las pruebas que cubren clases inválidas

        2. Variar tanto valores válidos como inválidos

**** EJEMPLO ****

Valores frontera

Existe una frase enunciada por Boris Beizer, teórico de Software y de las ciencias de computación, que dice *"Los defectos se esconden en las esquinas y se congregan en los límites"*.

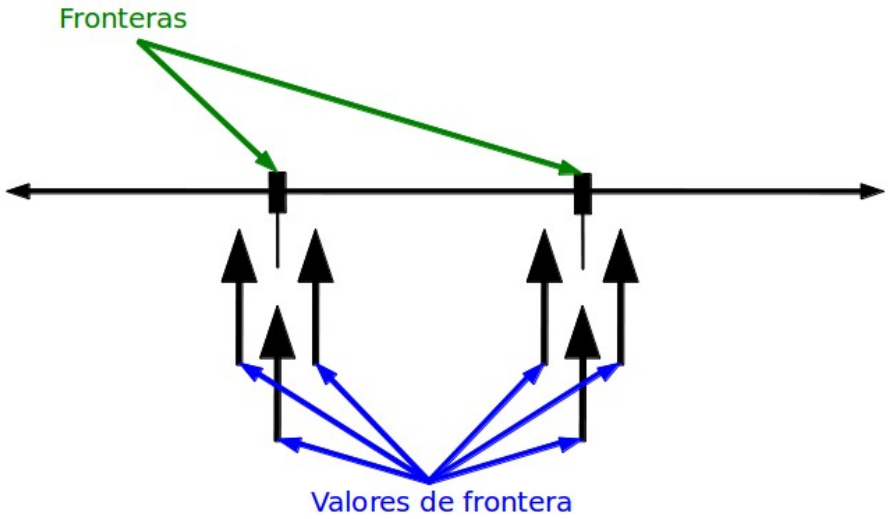
La técnica *valores de frontera* es una especialización de la de particiones de equivalencia orientada a entradas con valores ordenados.

La técnica parte de la identificación de las particiones de equivalencia y seleccionando los valores frontera de dos maneras:

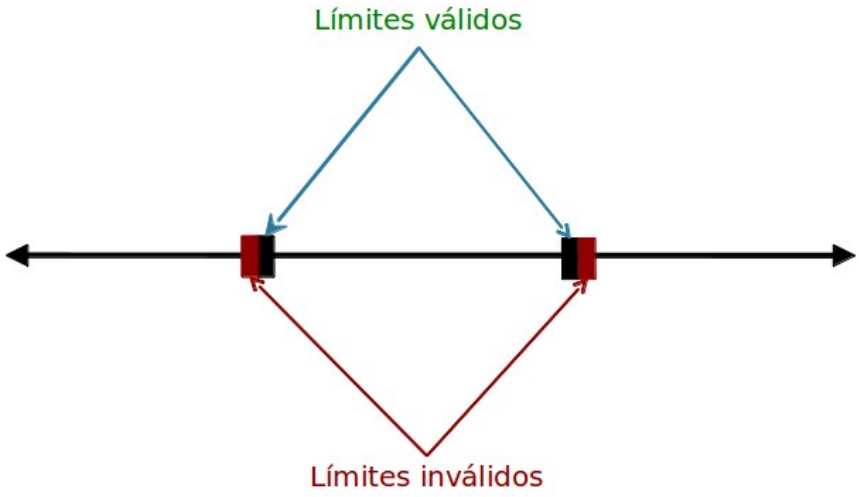
1. Por cada entrada y salida con fronteras identificables se definen 3 casos de prueba por frontera:

    1. uno en la frontera

    2. uno de cada lado de la frontera con el menor incremento/decremento posible



2. Por cada entrada y salida con particiones identificables se definen 2 casos de prueba:
    1. uno para el máximo de cada partición válida o inválida
    2. uno para el mínimo de cada partición válida o inválida



La técnica permite agregar más casos de pruebas que se encuentren alejados de las fronteras, pero estos raramente encuentran más defectos.

Cuando el hecho de crear pruebas individuales para cada límite trae un problema de gasto de tiempo, se crean casos que prueban varias fronteras a la vez pero siempre buscando probar un límite inválido por vez.

Árbol de clasificación

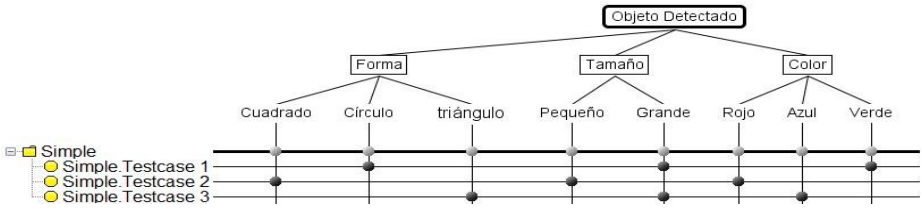
Esta técnica es también una especialización de la de particiones por equivalencia.

Los pasos de la técnica:

1. Para cada entrada se determinan los diferentes aspectos funcionales relevantes
    1. Se define una partición para cada aspecto
    2. Los aspectos se representan en forma de árbol
    3. Los casos de prueba se diseñan combinando las particiones de los diferentes aspectos

El árbol de clasificación organiza la información y permite seleccionar casos que cubran la mayor cantidad de combinaciones de los aspectos.

Por ejemplo:



Prueba de pares

Es una técnica para cubrir una cantidad significativa de combinaciones de valores (entradas, configuraciones).

Podemos pensar que un sistema a probar puede ejecutarse bajo una gran diversidad de configuraciones, con diferentes entornos, agregados, etc.

Al momento de calcular la cantidad posible de combinaciones podemos encontrarnos con que el número hace que sea complicado hacer un recorrido por todas ellas, por lo que se plantea el enfoque de probar todas las combinaciones de valores sólo entre dos variables.

Existen al menos dos métodos para seleccionar situaciones con todas las combinaciones de a pares:

- Arreglos ortogonales
- Arreglos de pares

Son métodos complejos pero existen algunas herramientas automáticas para computar las situaciones que facilitan la tarea.

Por ejemplo, supongamos que nuestro sistema a testear es web y podemos encontrar la siguiente variabilidad de entornos y configuraciones:

- 6 navegadores web
- 3 plug-ins necesarios para que el sistema funcione correctamente
- 7 diferentes Sistemas Operativos clientes que accederán a él
- 3 distintos servidores web en los que podría correr
- 3 distintos Sistemas Operativos que pueden alojar a los servidores
- Total: **1134 combinaciones posibles**

Sin embargo, esta técnica nos dice que es posible encontrar muchas de las posibles fallas con sólo 42 combinaciones, probando todas las combinaciones sólo entre 2 variables.

Manejo de datos

En los sistemas **ABML** (Alta, Baja, Modificación y Lectura) los

datos tienen un ciclo de vida dentro de la aplicación que:

- Comienza cuando la entidad es creada (Alta)
- Durante su ciclo de vida, es modificada (Modificación) o leída (Lectura)
- Termina cuando es destruida (Baja)

Esta técnica se ejecuta de la siguiente manera:

1. Se crea una matriz ABML
    1. Por cada función del sistema determinamos:
        1. entidades usada en la función
        2. acciones (A, B, M o L) llevadas a cabo por esas entidades
    2. El resultado es registrado en la matriz
        1. una columna por cada entidad
        2. una fila por cada función
        3. las acciones de una función sobre una entidad específica en la celda correspondiente
2. Probar completitud (prueba estática):
    1. Examinar si por cada entidad se pueden realizar las 4 acciones
    2. Que se encuentren incompletas no implica un error, pero si efectúa un llamado de atención
        1. Entidades no creables no deberían ser destruibles y raramente modificables
        2. Entidades no destruibles implican un aumento constante del volumen de datos
        3. Entidades no modificables implican imposibilidades de corregir errores de tipo
        4. Entidades no legibles suelen no tener sentido
    3. Probar consistencia (prueba dinámica);
        1. Verificar que las funciones usen consistentemente una entidad
            1. Cada prueba comienza con una A, sigue con todas las posibles M y finaliza con una B
            2. Luego de cada acción (A, B o M), se realiza la L correspondiente, para verificar que la entidad fue correctamente procesada
            3. Para cada entidad relevante, todas las ocurrencias de las acciones (A, B, M o L) en todas las funciones deben ser cubiertas
        2. Variante más intensiva: realizar <u>todas</u> las L luego de un A, B o M.

*** EJEMPLO CON ABML ***

Tabla de decisión

Las tablas de decisión como técnica son utilizadas para especificar las reglas de negocio de un sistema, las entradas y las posibles respuestas que el sistema tendrá frente a ellas.

Estas tablas consisten en:

- Una serie de condiciones de entrada
- Una serie de acciones a ejecutar y sus características
- Un conjunto de reglas que definen qué acciones se ejecutan para cada combinación de condiciones de entrada

La técnica se desarrolla de la siguiente manera:

1. Construir la tabla en función de los requisitos
2. Diseñar casos de prueba en función del tipo de condición:
    1. Condición binaria: un caso de prueba por cada valor
    2. Rango de valores: aplicar la técnica de partición de equivalencia o, mejor aún, de análisis de frontera

Una columna colapsada debe tratarse como dos columnas separadas.

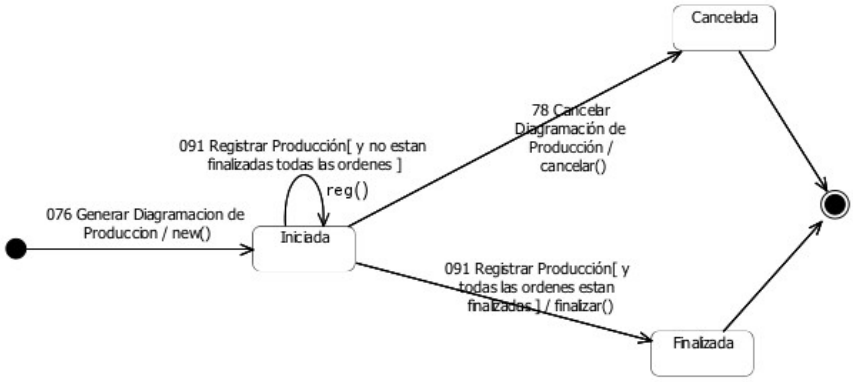
Diagrama de transición

Esta técnica es una manera de modelar programas reactivos, contando con los siguientes componentes:

- **Estado:** condición distinguible del sistema que persiste por un período de tiempo significativo
- **Evento:** acontecimiento, interno o externo, reconocible por el sistema
    - Pueden ser señales, invocaciones, paso del tiempo, etc.
    - En cada estado los eventos no explicitados son considerados inválidos para ese estado
- **Transición:** Relación entre estados que representa el paso de uno hacia el otro
    - El estado resultante puede ser el estado de partida
    - Son disparadas por eventos
    - Su ejecución puede estar restringida por condiciones
    - Su ejecución es ininterrumpible y su duración en el tiempo no es significativa
- **Acción:** comportamiento no interrumpible ejecutado a la entrada o salida de un estado
    - Su ejecución no dura una cantidad significativa de tiempo
    - Las acciones son ejecutadas sin importar cómo se ingresa o se egresa de un estado

Los casos de prueba derivados por esta técnica están guiados por los siguientes principios:

- Cubrir todas las transiciones
- Cubrir todos los estados
- Cubrir algunas secuencias de transiciones
- Considerar eventos inválidos



Pasos de la técnica:

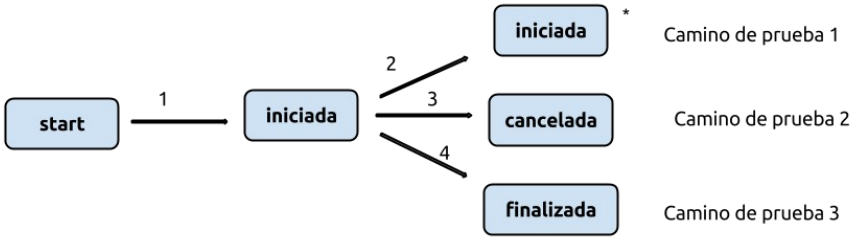
1. Componer la tabla de estado-evento
    1. Listar eventos por filas
    2. Agregar una columna para el estado inicial
    3. En cada celda de combinación evento-estado legal colocar el estado resultante con un identificador único de transición
        1. Las combinaciones ilegales se marcan con un punto
    4. Consecutivamente ir agregando una columna por cada estado resultante, y proseguir de igual manera hasta que todos los estados estén listados

Evento	Estado			
	start	iniciada	cancelada	finalizada
`new()`	1 - iniciada	•	•	•
`reg()`	•	2 - iniciada	•	•
`cancelar()`	•	3 - cancelada	•	•
`finalizar()`	•	4 - finalizada	•	•

2. Componer el árbol de transición
    1. Los nodos del árbol son estados y las aristas son transiciones
    2. La raíz del árbol es el estado inicial
    3. Siguiendo la tabla agregamos una arista y un nodo (estado resultante) por cada transición. La arista es

marcada con el identificador

4. Por cada nodo del siguiente nivel, que no aparezca en los niveles precedentes, se procede de igual manera
5. Las “hojas” del árbol que no sean un estado final o el estado inicial son puntos terminales provisionales y se marcan con *



3. Derivar casos de prueba legales
    1. Con la ayuda del árbol de transiciones y la tabla de estado-evento, se definen los casos de pruebas legales
    2. Cada camino en el árbol de prueba es un caso de prueba. Cada camino posee el evento, las acciones esperadas y el estado resultante.
4. Derivar casos de prueba ilegales
    1. Las combinaciones ilegales de estado-evento se pueden obtener de la tabla
    2. El resultado esperado es que el sistema no reaccione
    3. Si el estado no es inicial, se puede combinar con un caso de prueba legal que conduzca a dicho estado.

Técnicas de caja blanca

En los criterios o técnicas de caja blanca, a diferencia de los de caja negra, el análisis se realiza teniendo acceso al código fuente que da vida al producto a probar. Es, entonces, foco de estas técnicas la implementación.

Muchos de este tipo de criterios exploran la estructura del código, buscando a través de ese análisis generar suites que ejerciten dicho código de diferentes maneras y permitan una automatización y mensurabilidad.

Cobertura de sentencias

Para satisfacer el criterio de cobertura de sentencias se debe asegurar que se ejecutan todas las sentencias del programa al menos una vez por algún test de la suite, siendo éste uno de los criterios de caja blanca más débiles. Puede dar problemas como errores en condiciones compuestas y ramificaciones que son ignoradas por la ejecución de las pruebas.

Entendemos a una sentencia ejecutable como a aquella que esté asociada al código máquina (asignaciones, evaluaciones de guardas, llamadas a funciones, inicialización de variables, etc.).

En una gran mayoría de casos este criterio puede ser satisfecho con suites pequeñas.

Cobertura de decisiones

Una decisión es un punto en el código en el que se produce una ramificación o bifurcación (Ej: condiciones de ciclos, if-then-else).

La técnica de cobertura de decisión se satisface si todos los resultados de las decisiones del programa son ejecutados al menos una vez por al menos un test de la suite. Ella es más fuerte que la cobertura de sentencias y su satisfacción incluye la satisfacción de esta última.

Cobertura de condiciones

Una decisión puede estar compuesta por una o más condiciones, donde una condición es una expresión booleana que es evaluada para determinar el resultado de una decisión.

Ej:

```
if (index < count(list) && !found …
```

La satisfacción de esta técnica se dará cuando cada condición (de cada decisión) es ejecutada por verdadero y por falso por al menos un test de la suite, lo que no implica que sean todas las combinaciones. Cobertura de condición no es más fuerte que cobertura de decisión, estos métodos son incomparables.

Cobertura de caminos

El grafo de flujo de control de un programa es una representación, mediante grafos dirigidos, del flujo de control del programa:

- Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente
- Los arcos del grafo representan transferencias de control entre nodos

La técnica se satisface cuando todos los caminos del grafo de flujo de control son recorridos por al menos un test de la suite.

Es un criterio muy fuerte pero que para ser alcanzado puede requerir la creación de suites muy grandes, por lo que para hacerlo practicable se le suelen imponer restricciones como:

- Cobertura de caminos simples: requiere cubrir caminos sin repetición de arcos
- Cobertura de caminos elementales: requiere cubrir caminos sin repetición de nodos

Si una suite satisface cobertura de caminos, ésta satisface cobertura de decisiones y, por lo tanto, de sentencias.

Complejidad ciclomática

La complejidad ciclomática es una métrica que estima la complejidad lógica de un componente de software. Se basa en el diagrama de flujo determinado por las estructuras de control de un determinado código. De dicho análisis se puede obtener una medida cuantitativa de la dificultad de crear pruebas automáticas del código y también es una medición orientativa de la fiabilidad del mismo.

Cálculo:

```
CC = arcos - nodos + 2 * componentes

//Con un punto de entrada y un punto de salida:
CC = decisiones + 1

//Con un punto de entrada y múltiples salidas
CC = decisiones – salidas + 2

//CC sirve como cota superior del nº de test
necesario para cobertura de decisiones.
```

Diversos estudios de la industria del software han indicado que módulos con complejidad ciclomática alta, tienen mayor probabilidad de contener defectos.

Una herramienta para el testing: Testlink

La herramienta **Testlink** es un software basado en la web que se utiliza para facilitar el aseguramiento de la calidad del software. Su plataforma ofrece soporte para casos de prueba, suites de prueba, proyectos de prueba y administración de usuario, así como una gran variedad de informes y estadísticas.

Es de licencia GPL (licencia de software libre) y su última versión estable es la 1.9.5 lanzada el 08/12/2013. Al ser un sistema web, se necesita para su ejecución un servidor de aplicaciones y un motor de base de datos, así como también un navegador web en el cliente que quiera accederlo. Su arquitectura es multiplataforma.

Puede encontrarse el proyecto en los siguientes sitios:

- http://sourceforge.net/projects/testlink/
- http://www.teamst.org/

Día 2

Cerrando: ¿Cómo probarlo?

Finalizando el espacio para práctica de lo visto en el día 1 haciendo foco sobre metodología del testing y derivación de pruebas de caja negra, comentando experiencias y lo aprendido hasta ahora.

¿Cómo compone el ciclo de vida lo hecho hasta ahora?

Lo avanzado hasta el momento plantea una serie de conceptos, tareas y técnicas para llevar adelante la acción de prueba y aseguramiento de calidad, pero esto no ha sido enmarcado o guiado de la mano de un proceso o ciclo de vida.

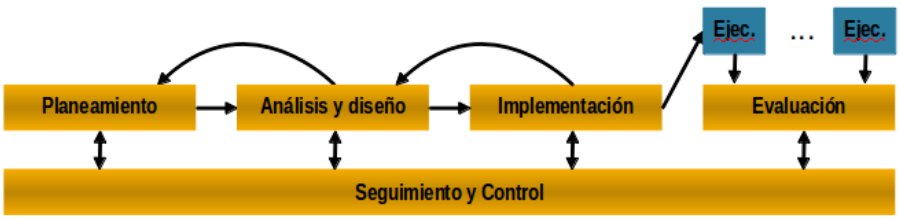
Ordenando el proceso de testing

Realizar testing profesionalmente implica la necesidad de llegar a una formalización del proceso distinguiendo diferentes fases que consistan de un número finito de actividades y que de las cuales se defina lo siguiente:

- Un objetivo preciso
- Los productos que deberán ser entregados
- Cómo la actividad debe ser desarrollada

El hecho de disponer de un proceso ordenado y un ciclo de vida claramente definido permite lo siguiente:

- Hacer gestionable el proceso de testing
- Definir roles, objetivos y cronogramas
- Planificar el esfuerzo del testing
- Conocer el progreso del proceso de testing



Ciclo de vida

El ciclo de vida del testing está compuesto por las siguientes etapas:

Planeamiento

La fase de planeamiento tiene como objetivo:

- Delinear el proceso de prueba para que pueda ser correctamente ejecutado, medido y controlado
- Anticiparse a los problemas que puedan ocurrir durante todo el ciclo de vida
- Discutir y comprender:
  - La calidad esperada del objeto de prueba
  - La organización de las diferentes tareas
  - La disponibilidad de personal, infraestructura y tiempo

Un plan de pruebas puede ser un simple documento de algunas páginas, o uno de tamaño considerable de 200 páginas. La formalidad del plan de pruebas debe ser acorde a la magnitud del proyecto.

Las **actividades** dentro del planeamiento de las pruebas son:

1. Formular la misión de pruebas para determinar:
   - Responsables del proyecto
   - Alcance
   - Objetivos
   - Precondiciones del proceso de testing
   - Restricciones del proceso de testing
2. Revisar y estudiar globalmente el proyecto, para informarse sobre:
   - Las generalidades del proyecto
   - El objeto a probar
   - Las necesidades del usuario final
   - Las opiniones de otros miembros del equipo de testing
   - La historia del grupo de trabajo
3. Establecer la base de pruebas para determinar:
   - La documentación relevante
   - La calidad requerida de la documentación
4. Determinar la estrategia de pruebas, que:
   - Describa el abordaje de la misión de prueba
   - Defina cómo son cubiertos los requerimientos y los

riesgos del objeto de prueba

Se puede definir una estrategia de prueba por cada nivel y luego complementarlas para obtener una estrategia global.

Estrategia de pruebas

- La **estrategia basada en riesgos** responde a los siguientes interrogantes:
  - ¿Cuáles son los riesgos del producto?
  - ¿Cómo son clasificados los riesgos?
  - ¿Cuál es su importancia relativa?
  - ¿Cuánto conocimiento posee el grupo de testing de los riesgos de producto?

La importancia y costo de los defectos se determina en base a los riesgos de que el producto manifieste fallas.

- La **estrategia basada en requerimientos** responde a los siguientes interrogantes:
  - ¿Cuál es la relevancia relativa de los requerimientos?
  - ¿Cuál es la relevancia relativa de los subsistemas?
  - ¿Cuál es la relevancia relativa de los atributos de calidad?

Definir una estrategia incluye:

- Determinar qué pruebas realizar por cada nivel de prueba
- Producir trazabilidad de pruebas a riesgos de producto
- Producir trazabilidad de pruebas a requerimientos del producto
- Esclarecer y determinar balance entre calidad deseada del producto y monto de dinero/tiempo requerido
- Asignar técnicas de diseño de pruebas de acuerdo a la profundidad deseada
- Evaluar el uso de herramientas y automatización de pruebas
- Estudiar la reusabilidad de las pruebas
- Considerar experiencia previa del grupo de trabajo
- Coordinar estratégicamente con el equipo de desarrollo de:
  - Testing regresivo completo
  - Testing regresivo puntual: por defecto, funcionalidad o subsistema

5. Establecer el equipo de pruebas:
   - Determinando roles
   - Asignando tareas, autoridades y responsabilidades

- Describiendo los puestos dentro del equipo y asignando personal
- Determinando la capacitación (dominio, tecnología del desarrollo, herramientas de prueba)
- Determinando la estructura de comunicación y líneas de reporte

Equipo de pruebas

- **Gerente de pruebas**
  - Responsabilidades:
    - Conducción estratégica y coordinación
    - Facilitador
    - Dirección total – Planeamiento de la prueba
    - Adquisición de recursos
    - Reportes del proyecto – evaluación y seguimiento
  - Requerimientos:
    - Conocimiento profundo del proceso de prueba
    - Familiaridad con las herramientas de prueba
    - Capacidad de liderazgo
    - Capacidad para el manejo de proyectos
- **Ingeniero de pruebas**
  - Responsabilidades:
    - Descomposición de los requerimientos de pruebas
    - Diseño de las pruebas
    - Implementación y ejecución de las pruebas
    - Evaluación de resultados
    - Recuperación de errores
  - Requerimientos:
    - Conocimiento de los requerimientos de la aplicación
    - Familiaridad con las herramientas de prueba
    - Destreza en programación (opcional)
    - Capacidad de diagnóstico
- **Administrador del sistema de pruebas**
  - Responsabilidades:
    - Administrar el sistema de prueba
    - Instalar nuevos usuarios
    - Manejar las necesidades del usuario

- Requerimientos:
  - Experiencia y conocimientos de administración de sistemas
  - Conocimiento de las herramientas de prueba

6. Especificar la infraestructura de pruebas, determinando:
   - Ambiente de prueba
   - Herramientas de gestión
   - Configuración de la oficina
   - Planificación de la infraestructura
7. Especificar los entregables, determinando:
   - Los productos de cada fase del proceso de testing

y desarrollando:

- Normas
- Estándares
- Guías de estilo

8. Organizar la gestión y el control, definiendo un proceso de seguimiento de las pruebas que incluya:
   - Criterios generales de entrada, salida y continuación de las pruebas
   - Contenido, periodicidad y destino de informes de progreso, calidad y métricas
   - Procedimientos de control y auditoría de las pruebas
   - Gestión de la infraestructura
   - Administración de entregables de prueba
   - Administración de defectos
9. Programar el proceso de prueba, determinando:
   - El plan global
   - El presupuesto
   - El cronograma
10. Consolidar el plan de pruebas
    - Determinando:
      - Criterio de finalización
      - Riesgos
      - Desafíos
      - Medidas
      - Manejo de contingencias y estrategias de mitigación

- Documentando el plan
- Consiguiendo el aval del sponsor del proyecto

Criterios de finalización

Existen diversos criterios para determinar la finalización del plan de pruebas:

- Ejecución exitosa de casos seleccionados
- Porcentaje de cobertura alcanzada por casos de prueba ejecutados
- Umbral de defectos detectados por unidad de tiempo
- Cantidad de defectos remanentes por severidad
- Finalización por el absurdo
  - Cantidad total de defectos detectados
  - Tiempo de testing transcurrido.
- Normalmente se deja de hacer testing cuando se acaban los recursos , ya que siempre es posible seguir probando y encontrar nuevos defectos no detectados hasta ese momento
- Podemos decir que, en la práctica, el testing no termina nunca, sólo cambia quién lo realiza, del profesional de testing al usuario final

Planeamiento de las pruebas

A partir de la información existente sobre el proyecto y la documentación del mismo es que se parte para confeccionar un plan de pruebas acabado y efectivo que otorga las siguientes **salidas**:

- Plan de Prueba:
  - Misión de pruebas
  - Estrategia de prueba
  - Recursos necesarios
  - Infraestructura
  - Organización
  - Entregables
  - Cronograma
  - Presupuesto

Análisis y diseño

Los objetivos de esta fase son:

- Derivar los casos de prueba de acuerdo a las técnicas

apropiadas

- Determinar si la base de pruebas tiene la suficiente calidad para el diseño de las pruebas
- Definir los casos de prueba que cumplen la misión de acuerdo a la estrategia definida, evaluando las posibilidades de automatizarlos

Las **actividades** de esta fase son:

1. Revisar la base de pruebas, verificando si es acorde a la estrategia definida y posee la calidad suficiente:
    - Recopilando la documentación relevante
    - Diseñando checklists de revisión
2. Definir los requerimientos de pruebas
    - Conducen el proceso de pruebas
    - Deben ser trazables a requerimientos y riesgos del producto
    - Permiten refinar el esfuerzo estimado del proceso en el planeamiento
    - Son la base para fijar un criterio de alcance de la prueba y para la implementación de los casos de prueba
3. Derivar los casos de prueba lógicos, determinando
    - El orden en que deben ser ejecutadas, indicando cuando esto sea o no relevante
    - Los puntos de verificación adecuados a cada momento de la ejecución
4. Definir la automatización evaluando las necesidades y oportunidades:
    - Por factibilidad
    - Por precisión
    - Por fiabilidad
    - Por frecuencia
    - Por costo
5. Especificar la infraestructura, estableciendo las configuraciones necesarias para ejecutar los casos de prueba definidos

Esta fase toma como entrada la información del proyecto y la documentación existente del software para producir las siguientes salidas:

- Reporte de testeabilidad
- Requerimientos de prueba
- Casos de prueba lógicos con trazabilidad a requerimientos de prueba

- Análisis de cobertura de los requerimientos de prueba
- Conjunto de pruebas a automatizar

Implementación

Esta fase tiene como objetivo construir los conjuntos o bancos de pruebas desarrollando:

- Las pruebas físicas (automatizadas o no)
- Las herramientas, utilidades y desarrollos especiales necesarios

Las **actividades** de la fase son:

1. Desarrollar los casos de prueba físicos:
    - Definiendo los datos de entrada y resultados esperados
    - Escribiendo los procedimientos manuales o los scripts automatizados
2. Producir los escenarios de pruebas:
    - Conjuntos de casos de prueba cuya ejecución debe ser coordinada, de acuerdo a un orden establecido y puntos de verificación
3. Especificar las verificaciones previas
    - Creando un checklist de completitud de objeto de prueba
    - Creando un checklist de completitud de infraestructura de prueba
    - Creando test scripts de pre-prueba (prueba piloto)
4. Instalar la infraestructura
    - Configurando el ambiente de pruebas
    - Configurando las herramientas
    - Realizando la prueba piloto

La fase toma como entrada los siguientes elementos:

- Requerimientos de prueba
- Casos de prueba lógicos
- Conjunto de pruebas a automatizar
- Prototipo o entrega de software

Y ofrece como salida:

- Scripts de prueba
- Utilitarios para la ejecución de la prueba
- Controles de ejecución de pruebas

Ejecución

El objetivo de esta fase es ejecutar el conjunto de pruebas contra una versión del producto, registrando sus resultados.

Las **actividades** de la fase son:

1. Verificar el objeto de prueba e infraestructura realizando las pruebas piloto
2. (Re)Ejecutar las pruebas, ya sean manuales o automatizadas
3. Analizar y registrar los resultados evaluando las desviaciones y registrando particularmente los defectos encontrados

Análisis y registro de los resultados

- Por cada caso (o etapa de la prueba) debe estar establecido cuál es la información que vamos a registrar
- Recordar que a mayor cantidad de datos:
  - Mejor información
  - Mayor tiempo utilizado en esta tarea
  - Mayor probabilidad de que tengan imprecisiones o sean incompletos
- Recomendación: usar procedimientos estandarizados para registrar la ejecución de la prueba

Ante la falla de un caso de prueba, asegurarnos que es una real falla mediante:

- Determinar si se ha procedido correctamente a la ejecución de la prueba
- Ver si el resultado inesperado no es consecuencia del ambiente
- Comparar con los datos fuente

Por cada defecto encontrado se debe registrar:

- Identificación del defecto
- Sistema o componente afectado
- Versión y fecha en que se detectó
- Casos de prueba involucrados
- Ambiente/tipo de prueba que se detectó
- Persona que reporta y quién lo detectó
- Descripción corta y detallada
- Pasos para reproducirlo

Además, los defectos suelen clasificarse de acuerdo a una serie de atributos:

- **Prioridad**: califica el defecto según tiempo y oportunidad de realizar la corrección desde el punto de vista del testing
  - Inmediata
  - Urgente
  - Media
  - Baja
- **Severidad**: especifica la forma en la que el defecto afecta al producto bajo testeo
  - Invalidante
  - Grave
  - Comúm
  - Leve
  - Mejora
  - Consulta
- **Síntomas**: Especifican la forma en que el defecto se manifiesta
  - Caída del sistema
  - Corrupción de datos
  - Operación incorrecta
  - Pérdida de datos
  - Comportamiento inesperado
  - Comportamiento poco amistoso
  - Rendimiento
  - Presentación
  - Problema de entorno
- **Resolución (al cierre)**: es la forma en que se da por terminado un defecto
  - Duplicado
  - Corregido
  - Corregido indirectamente
  - Funciona según diseño
  - Limitación de hardware
  - Limitación de software
  - Necesita más información
  - Corrección no planificada
  - Diferido
- **Condiciones del error**:

- Defectos en las especificaciones
- Defectos en la definición de los eventos
- Fallas ajenas a la aplicación

4. Mantener los escenarios de prueba
   - El escenario de prueba ejecutado probablemente sea utilizado nuevamente
   - Por lo tanto todos los problemas detectados durante la ejecución del mismo deben ser corregidos

La entrada de esta fase se compone de:

- Escenarios de pruebas
- Procedimientos de pruebas
- Orden de ejecución de la prueba
- Especificaciones de ambiente de la prueba

Y su salida es:

- Registro de ejecución de pruebas
- Informe de defectos

Evaluación

Los objetivos de esta fase son:

- Analizar los resultados de la ejecución de las pruebas para determinar si los requerimientos son satisfechos
- Determinar la calidad del objeto a probar y el progreso de las pruebas diseñadas

Las **actividades** de la fase son:

1. Evaluar el objeto de prueba
   - Determinando fallas no resueltas
   - Determinando tendencias
   - Determinando riesgos en la entrega
   - Formulando consejos u opiniones

Las fallas nos proveen una indicación de la calidad de corrección del software y su cantidad puede ser reportada como una función del tiempo y/o una función de algún atributo de la falla (la gravedad, el estado).

Se pueden utilizar diferentes reportes:

- De distribución de fallas
- De edad de la falla

- De tendencia de las fallas
- De progreso del testeo
- De cobertura en la ejecución de las pruebas

La cobertura pretende determinar qué tan completo es el conjunto de casos de prueba:

- Cobertura de requerimientos: verifica que todos los requerimientos de prueba sean cubiertos
- Cobertura lógica: verifica que cada instrucción sea ejecutada al menos una vez, que cada condición en una decisión tome todos los resultados posibles al menos una vez, etc.
- Cobertura funcional

2. Evaluar el proceso de pruebas, revisando:
   - La estrategia de prueba
   - La planificación contra la realización
   - La utilización de recursos
3. Archivar el testware
   - Seleccionar qué conservar
   - Coleccionar y actualizar el testware (para reutilización en otros procesos de prueba)
   - Entregar el testware

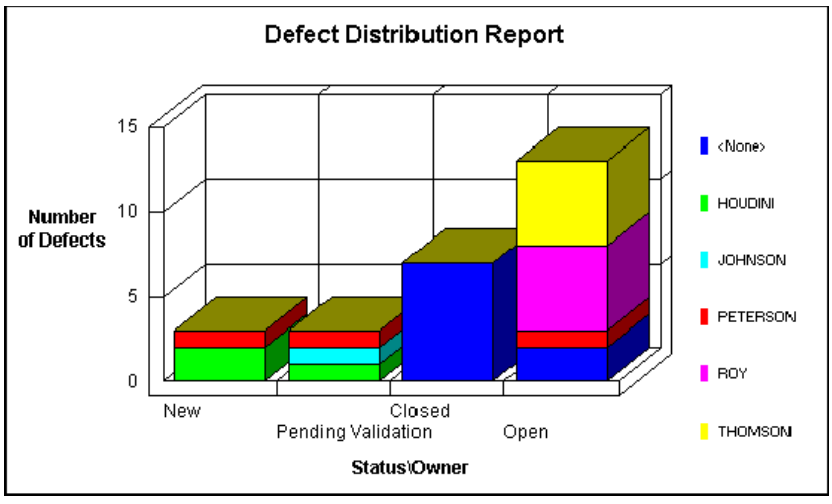
Como entrada de la fase de Evaluación tenemos:

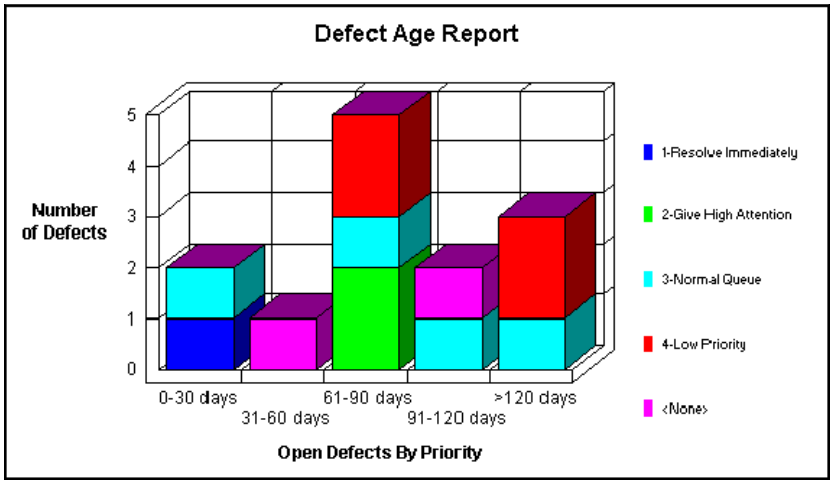
- Registros de ejecuciones de las pruebas
- Incidentes reportados
- Plan de prueba
- Procedimientos de prueba

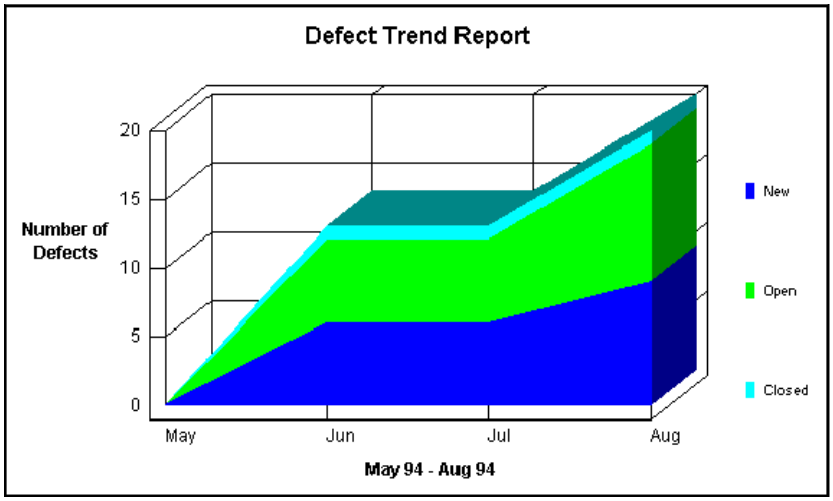
Y la salida será:

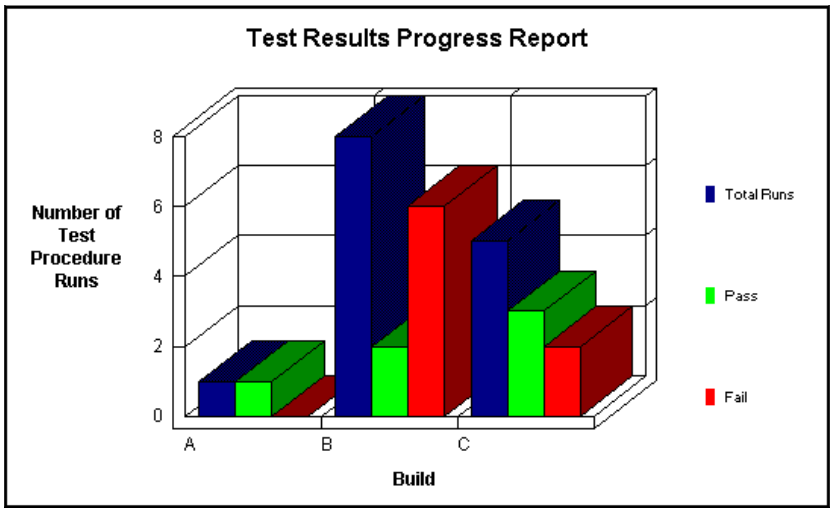
- Medida de la cobertura de la prueba
- Reporte de análisis de defectos

Algunos gráficos (a modo de ejemplo) de la fase de evaluación de pruebas pueden ser:









Seguimiento y control

La etapa de seguimiento y control registra los incidentes de las pruebas, el proceso de prueba y los problemas del objeto a probar, investigándolos y creando estructuras que permitan solucionarlos.

Tiene por objetivo la coordinación, monitoreo y control del

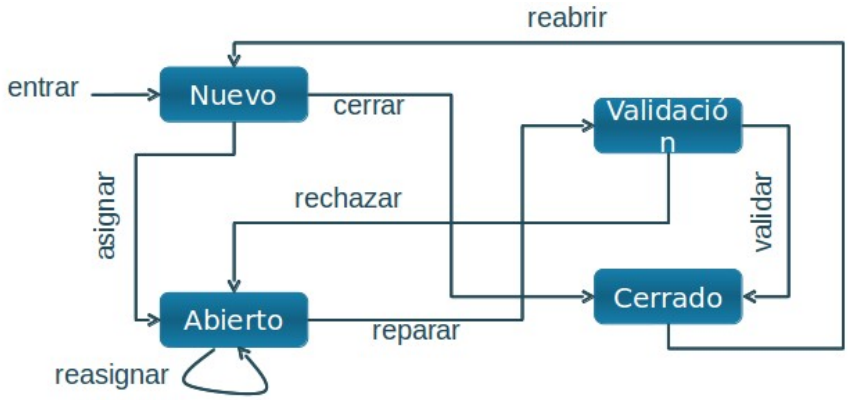
proceso de prueba a fin de entender la calidad del objeto a prueba dentro del tiempo, presupuesto dado y con el personal apropiado.

Las **actividades** de la fase son:

1. Mantener el plan de pruebas
   - Ajustar estrategia y/o plan de pruebas
   - Mantener cronograma detallado
2. Controlar las pruebas
   - Auditar que las tareas se desarrollen de la manera prevista
   - Verificar el cumplimiento de las entregas de y a desarrollo
   - Llevar a cabo la gestión de los defectos

Gestión de defectos

Es el procedimiento de seguimiento de los defectos desde su inicio hasta su cierre. Se define un flujo de trabajo entre los diferentes miembros del equipo de testing:



- El equipo de pruebas genera un defecto y lo asigna al gerente de pruebas
- El gerente de pruebas revisa los nuevos defectos y:
  - Los abre, asignándolos
  - O Los resuelve
- Un miembro del equipo de desarrollo ve la lista de defectos abiertos y:
  - Los corrige
  - O re-asigna a otro miembro
- Personal de pruebas revisa la lista de defectos corregidos y los prueba con la nueva versión del programa, y:
  - Rechaza no corregidos y re-asigna defectos

- Cierra los corregidos
- Además puede re-abrir defectos que aparezcan nuevamente

3. Reportar
    1. Proveer a la organización reportes sobre el avance sobre el proceso de testing y la calidad del objeto bajo prueba
    2. Los informes pueden ser:
        - Periódicos
        - A pedido

Como salida de la fase tenemos lo siguiente:

- Reporte final del objeto bajo prueba
- Reporte final del proceso de prueba

Una herramienta para la gestión: Redmine

Redmine es una herramienta para la gestión de proyectos que incluye un sistema de seguimiento de incidentes con seguimiento de errores. Otras herramientas que incluye son calendario de actividades, diagramas de Gantt para la representación visual de la línea del tiempo de los proyectos, wiki, foro, visor del repositorio de control de versiones, RSS, control de flujo de trabajo basado en roles, integración con correo electrónico, etcétera.

Está escrito usando el framework Ruby on Rails. Es software libre y de código abierto, disponible bajo la Licencia Pública General de GNU v2.

Para encontrarlo en la web: www.redmine.org

Características

- Soporta múltiples proyectos.
- Roles flexibles basados en control de acceso.
- Sistema de seguimiento de errores flexible.
- Diagramas de Gantt y calendario.

- Administración de noticias, documentos y archivos.
- Fuentes web y notificaciones por correo electrónico.
- Integración SCM (Subversion, CVS, Git, Mercurial, Bazaar y Darcs).
- Soporta diferentes bases de datos (MySQL, PostgreSQL y SQLite).
- Plugins

Ambiente de prueba

Conceptos fundamentales

Para que la tarea de testing sea efectiva es esencial conocer cómo, cuándo, dónde y con qué configuración se obtuvieron los resultados de las pruebas, registrando, entre otros, los siguientes datos:

- Identificación de la configuración del hardware y software
- Fecha y hora
- Lugar
- Datos de calibración para el equipo de captura de datos
- Número de corridas de la prueba

Para ello, el testing debe realizarse en un entorno en particular, lo que llamaríamos Ambiente de Prueba.

Un ambiente de prueba es un conjunto de elementos de hardware, software y demás dispositivos que reproducen de la mejor manera el entorno de producción del software a probar.

El ambiente de pruebas incluye:

- Todo el hardware necesario para la prueba (procesadores, periféricos, dispositivos de red, etc)
- Todo el software necesario para la prueba (sistema operativo, software de base de datos, componentes necesarios, software a probar)
- Interfaces en general (ej: archivos recibidos de otros sistemas)
- Simuladores, prototipos, emuladores

El ambiente de pruebas debe:

- Ser independiente del resto de las áreas (desarrollo y producción)
- Garantizar estabilidad en los datos y elementos a probar de modo que los resultados obtenidos sean objetivamente representativos. (Esto es especialmente crítico en pruebas de rendimiento)

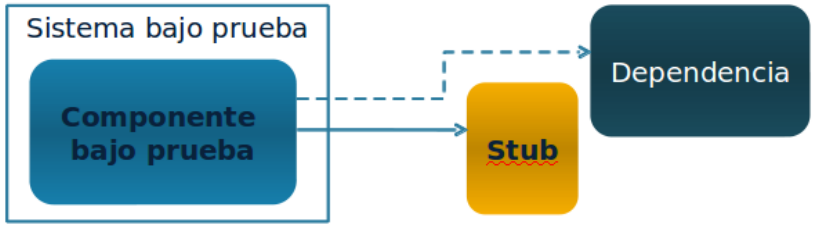
Arnés de prueba

Es un ambiente particular que incluye componentes de software para controlar la ejecución del producto bajo prueba:

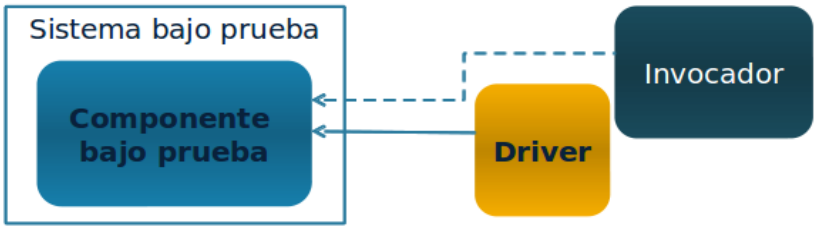
- Iniciar el programa
- Ejecutar las pruebas
- Informar sobre los resultados de las mismas

Suele ser un framework automatizado que incluye stubs y drivers.

- **Stub:** es un esqueleto o una implementación de un componente *A* para un propósito especial, usado para desarrollar o probar un componente *B*, que llama o depende del componente *A.* El stub reemplaza un componente invocado.



- **Driver:** Es un componente que reemplaza a un componente *A* que invoca y/o controla al componente o sistema *B* que se desea probar.



Características

Las características de un ambiente de pruebas son:

- Debe ser reproducible (o reinstalable o reseteable)
  - Se debe especificar qué se debe hacer en los casos en que algún componente no se pueda resetear
- El ambiente de pruebas debe estar documentado
  - Deben especificarse claramente las dificultades o problemas para su mantenimiento
- Deben enumerarse las diferencias existentes entre el ambiente de pruebas y el ambiente de producción y el

posible impacto de las mismas en las pruebas.

Actividades

Preparación y aseguramiento

En esta tarea se preparan todos los recursos necesarios para realizar las pruebas de acuerdo a la documentación del plan de pruebas y de las pre-condiciones de los casos de pruebas.

Se asegura la disponibilidad del ambiente y de los datos necesarios para ejecutar las pruebas.

Se debe:

- Establecer el estado inicial necesario para la ejecución de un conjunto de pruebas:
  - Fuentes de datos (restablecer la base de datos)
  - Entorno de la aplicación
- Asegurar un ámbito dedicado al testing:
  - Apagado de utilitarios
  - Apagado de aplicaciones que generan mensajes con pedidos de confirmación

Seguimiento y control

Esta actividad comprende todas las tareas tendientes a mantener un control estricto del estado del ambiente de pruebas, en todo momento.

Se debe mantener un seguimiento de:

- Procesos ejecutados
- Orden de ejecución de los casos de prueba
- Necesidades de modificaciones dentro del ambiente para la exitosa ejecución del plan

Documentación

Se debe documentar:

- Requisitos básicos de hardware y software base:
  - Ej: Sistemas operativos, gestores de bases de datos, monitores de teleproceso, etc.
- Requisitos de configuración de ambiente:
  - Ej: librerías, bases de datos, ficheros, procesos, comunicaciones, necesidades de almacenamiento, configuración de accesos, etc.
- Herramientas auxiliares

- Ej: Análisis de rendimiento

Día 3

Cierre de ciclo de vida

Espacio para práctica de lo visto en el día 2 haciendo foco sobre el ciclo de vida del proceso de testing y calidad que se llevó adelante, haciendo un repaso general que legitime las nuevas herramientas y conocimientos incorporados.

Automatización

A medida que aumenta la productividad en el desarrollo de software y disminuyen los tiempos entre la generación de nuevas versiones, crece la incertidumbre por la calidad final del producto. Muchas veces disminuye la calidad del núcleo estable del producto y aumentan los errores detectados y reportados en producción.

La automatización de las pruebas es una alternativa interesante para asegurar cierto nivel de calidad antes de cada liberación de productos o versiones. Aportan tranquilidad al ajustar y mejorar las principales funcionalidades, ya que brindan información sobre el impacto de los cambios realizados.

Es relevante seleccionar la herramienta que mejor se adapte al producto. La automatización es una inversión. Es tan importante el costo de la herramienta, como el tiempo que requiere la generación de los “scripts” de pruebas.

Motivaciones

- La velocidad del proceso de testing no alcanza a la velocidad de puesta en producción
- No es posible probar adecuadamente cada versión
- Mientras avanza el proceso de desarrollo, la cobertura de testing decrece y el riesgo aumenta
- El testing frecuente y amplio ayuda a asegurar la calidad
- La automatización de pruebas:
  - Permite aumentar la cobertura y por lo tanto, la seguridad
  - Permite ejecutar todas las pruebas sobre cada nueva versión del producto
  - Reduce al mínimo el esfuerzo en pruebas de regresión
  - Promueve las pruebas consistentes y repetibles

- Facilita la ejecución de las pruebas complejas, disminuyendo la posibilidad de cometer errores
- Evita la repetición de procedimientos rutinarios y aburridos
- Promueve una disciplina de documentación de pruebas

¿Cuándo automatizar?

- Cuando las pruebas tienen que ser repetidas muchas veces
- Cuando se cuenta con casos de prueba básicos pero con una amplia variación de las entradas
  - Los mismos pasos se ejecutan muchas veces pero con diferentes datos (pruebas dirigidas por datos)
- Cuando existe una alta probabilidad de cometer errores si las pruebas se realizan en forma manual
- En principio, cualquier caso de prueba puede ser automatizado, pero en la práctica sólo una pequeña parte de las pruebas son automatizadas

Herramientas

Las herramientas de automatización permiten la ejecución (semi) automática de pruebas, usando entradas previstas y verificando las salidas esperadas, utilizando algún lenguaje de scripting:

- Frameworks Xunit
- Herramientas para tipos de pruebas específicos: rendimiento, seguridad, etc
- Herramientas “Capture and Replay” o “Record and Playback”

Frameworks Xunit

Dan soporte a la metodología para probar componentes de forma aislada y suelen probar requerimientos funcionales o de integración. Los propios programadores son los que se encargan de crear estas pruebas y usual y preferentemente son desarrolladas al mismo tiempo (o incluso antes) que el código.

Un framework Xunit:

- Funciona para pruebas a nivel de Integración, de componente y funcionales
- Define la estructura básica de una prueba
- Permite organizar las pruebas en suites
- Ofrecen entornos para ejecutar las suites
- Reportan información detallada sobre los resultados, en

especial sobre las fallas

- Simulan las dependencias de los componentes bajo pruebas
  - Stubs, mocks, drivers
  - Porque aún no están disponibles
  - Para aislar el componente y tener aún mayor control
- Están disponibles para diferentes lenguajes de programación: Junit, Nunit, CppUnit, Vbunit, etc.

Herramientas para rendimiento

Las herramientas de rendimiento se utilizan en pruebas de nivel de Sistema y de Rendimiento y permiten responder las siguientes preguntas:

- ¿El sistema responde lo suficientemente rápido?
- ¿La infraestructura tiene la capacidad adecuada?
- ¿El sistema puede crecer para manejar mayores volúmenes en el futuro?
- ¿El sistema se comporta correctamente bajo una gran carga?

Las herramientas de rendimiento:

- Simulan y monitorean el comportamiento para evaluar carga, volumen, estrés, respuesta, etc.
  - Impacto de la configuración: procesador, memoria, cachés, adaptadores, etc.
  - Métricas de red: tiempo de respuesta punta a punta, cliente a servidor, latencia, utilización de ancho de banda, etc.
  - Métricas web: impactos por segundo, tasa de transferencia, respuestas HTTP por segundo, etc.
- Se basan en la ejecución repetitiva en diferentes condiciones de uso
  - Múltiples conexiones
  - Datos y volúmenes variables

Como herramienta para el rendimiento particular tenemos una solución open source interesante y útil: **Jmeter**

- Puede probar rendimiento de recursos estáticos y dinámicos: archivos, servlets, scripts pearl, objetos java, bases de datos, servidores FTP, etc.
- Puede simular cargas en un servidor, red u objeto bajo prueba, para probar su fortaleza o analizar su rendimiento global
- Se puede combinar con algún Xunit para realizar pruebas con mayor penetración

Capture and replay

- Permite pruebas a nivel de Aceptación, de Sistema y Funcional
- Permite grabar interactivamente las acciones del usuario y luego reproducirlas cuantas veces se desee
  - Las pruebas resultan muy sensibles a los cambios en interfaces
  - Usualmente incorporan actividades irrelevantes
- Están diseñadas para aplicaciones con GUI o páginas web
- Requieren mínimo conocimiento sobre desarrollo de software
- Selenium IDE, Selenium WebDriver, QALiber
- Las pruebas resultantes poseen muchas desventajas:
  - Sensibilidad al comportamiento
  - Sensibilidad a la interfaz
  - Sensibilidad a los datos
  - Sensibilidad al contexto
- Aún así son útiles para prototipar o como base para el desarrollo posterior de casos de pruebas

Ciclo de vida y mantenimiento

- Robotización del ciclo 0:
  - Las diversas funciones a probar son “robotizadas” para que, al recibir una nueva versión del sistema, el ciclo 0 se ejecute automáticamente
- Generalización del ciclo 0:
  - Se generalizan los “robots” para aceptar diferentes conjuntos de datos, o para que se ejecuten en diferentes circunstancias
- Ejecución del ciclo 0:
- Repetición de las pruebas y mantención:
  - Se ejecutan los robots construidos, y para los casos en que hay cambio de especificación o de funcionalidad se repiten los pasos anteriores

El proceso de automatización es similar a cualquier proceso de desarrollo, con fases de requerimientos, diseño, validación, liberación y mantenimiento. Las pruebas automatizadas ahorran dinero y tiempo **si se pueden reutilizar varias veces**. Las consecuencias de cambios en el sistema, diseño o requerimientos son una amenaza a la usabilidad de las pruebas automatizadas, las que deben ser implementadas de forma de minimizar su reimplementación, frente a cualquier cambio del objeto de prueba.

- Los cambios en el sistema pueden ser diversos:
  - Cambio en la funcionalidad
  - Cambio en el criterio de validación
  - Cambio de interfaces (de programación o gráficas)
  - Cambio de entorno de ejecución
- Las consecuencias de los cambios pueden ser:
  - La necesidad de diseñar casos de prueba adicionales
  - La necesidad de modificar los casos de prueba actuales

Una herramienta para automatización: Selenium

Selenium es un entorno de desarrollo de pruebas integrado para aplicaciones web, basado en la metáfora “Capture and Replay”. Está implementado como un plugin de Firefox y permite grabar, ejecutar, editar y debuggear casos de prueba.

Para construir un caso de prueba podemos utilizar tres técnicas (posiblemente combinadas):

- **Grabar:** Selenium inserta en el Script un comando cada click de mouse, ingreso de valores por teclado, etc. Se debe tener precaución con los tiempos de espera
- **Insertar aserciones o verificaciones:** para comprobar ciertas propiedades en cualquier punto de ejecución del script
- **Editar:** el script (resultante) para programar comandos específicos

Los comandos se dividen en tres clases:

- **Acciones:** manipulan el estado de la aplicación. Si falla, la ejecución del caso de prueba se detiene
- **Accessors:** examinan el estado de la aplicación y guardan el resultado en variables
- **Aserciones:** verifican que el estado de la aplicación coincida con el estado esperado. Se divide en tres modos:
  - **Assert:** cuando falla, la ejecución del caso de prueba aborta, por ej: assertText
  - **Verify:** cuando falla, la ejecución del caso de prueba

continúa y se registra la falla, por ej: verifyText

- **WaitFor:** espera que se cumpla una condición por un determinado tiempo. Si no se cumple, falla y la ejecución del caso de prueba aborta, por ej: waitForText

Los comandos más utilizados son:

- **open:** abre una página usando la URL como parámetro
- **click / clickAndWait:** realiza la operación de click, opcionalmente espera la carga del destino del link
- **verifyTitle / assertTitle:** verifica que el título de la página sea el esperado
- **verifyTextPresent:** verifica que el texto que se le pasa como argumento exista en la página
- **verifyElementPresent:** verifica que un determinado elemento UI identificado por el tag HTML esté presente en la página
- **verifyTable:** verifica el contenido de una tabla
- **waitForPageToLoad:** detiene la ejecución hasta que una nueva página es cargada. Es ejecutada automáticamente luego del comando clickAndWait
- **waitForElementPresent:** detiene la ejecución hasta que el elemento UI deseado se encuentre presente en la página

La mayor parte de los comandos necesitan referenciar algún elemento dentro de la página, para ello existen diversos métodos de localización:

- Por Id
- Por Name
- Por identificador
- Por texto del vínculo
- Por DOM
- Por CSS
- Por Xpath

Herramientas como **Firebug** nos permiten inspeccionar los localizadores. Los valores esperados pueden ser tanto constantes como expresiones regulares.

Cierre del curso

Bibliografía

1. Perry, William E., “Effective Methods for Software Testing”, *Wiley*, 3ra edición, 2006
2. Black, Rex, “Managing the Testing Process”, *Wiley,* 2a edición, 2002
3. Kaner, Cem, Falk, Jack, et. al, “Testing computer Software”, *Wiley*, 2a edición, 1999