Testing de Software

Objetivos

- Analizar los conceptos fundamentales de pruebas de software en el contexto del aseguramiento de calidad del software.
- Diseñar casos de prueba, planes de prueba y especificaciones de prueba utilizando técnicas apropiadas.
- Planificar, especificar, ejecutar y evaluar pruebas de software.
- Utilizar herramientas para mejorar la efectividad de las pruebas de software.
- Analizar los elementos críticos para la gestión del proceso de pruebas de software.

Contenidos

Introducción

- *software quality assurance*, objetivos, visiones y modelos de calidad, definiciones, revisiones, aspectos económicos
- conceptos básicos de testing, temas esenciales, principios, aspectos sicológicos y económicos, proceso, estado del arte v/s práctica
- taxonomía de errores, *debugging*

Fundamentos

- testing: definiciones, objetivos, casos de prueba
- diseño de casos de prueba (*black-box*, *white-box*)
- cobertura de *white-box* testing
- estrategias, métodos particulares de testing, *testability*, *cleanroom*
- plan, especificación, ejecución y evaluación de pruebas

Gestión

- enfoques organizacionales, prácticas, tendencias, desafíos
- mejoramiento de procesos, costos y beneficios, mediciones
- herramientas prácticas en el mercado, estándares, documentación de testing

Temas Avanzados

- pruebas de mutación
- pruebas de regresión

Primera parte

Introducción

- *software quality assurance*, objetivos, visiones y modelos de calidad, definiciones, revisiones, aspectos económicos
- conceptos básicos de testing, temas esenciales, principios, aspectos sicológicos y económicos, proceso, estado del arte v/s práctica
- taxonomía de errores, *debugging*

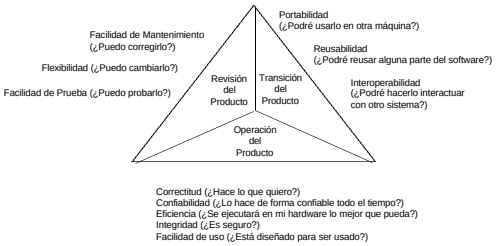
Calidad de software

- Objetivo último de Ingeniería de Software: producir software de calidad
- Calidad engloba todo el proceso, y está determinada por factores directos e indirectos
- Calidad es un concepto complejo y multifacético, que puede describirse desde diversas perspectivas

Visiones de calidad

- Visión trascendental
  - puede ser reconocida pero no definida
- Visión del usuario
  - grado de adecuación al propósito
- Visión del productor
  - conformidad con la especificación
- Visión del producto
  - ligada a características inherentes del mismo
- Visión basada en valor
  - ¿cuánto el cliente está dispuesto a pagar?

Modelo de calidad de McCall



Factores de calidad de software de McCall

Aseguramiento de calidad de software

- ***Software Quality Assurance (SQA)***

acciones sistemáticas y planificadas requeridas para asegurar la calidad de software

Grupo de SQA

- **Objetivos: planificar, desarrollar y controlar el proceso de verificación y validación**
- **Actividades: aplicación de métodos, revisiones e inspecciones, testing, aplicación de estándares, control de cambios, mediciones, registro**

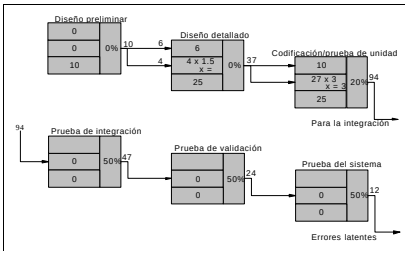
Verificación y validación

- **Verificación -- ¿estamos construyendo el producto correctamente?**
  are we building the product right?
- **Validación -- ¿estamos construyendo el producto correcto?**
  are we building the right product?

Revisiones de software

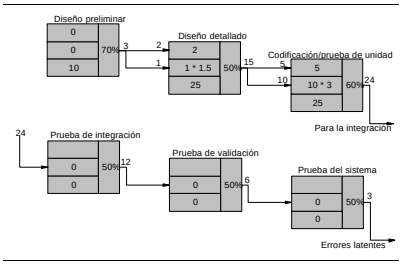
- **Actúan como un filtro**
- **Descubrimiento temprano de defectos -- gran impacto en los costos de testing y mantención**
- **Defectos de software presentan un efecto de amplificación**

Efecto de amplificación de defectos



Sin revisiones

Efecto de amplificación de defectos



Con revisiones

Revisiones de software
objetivo

- **Detectar defectos en la lógica, función o implementación**
- **Otros beneficios *gratuitos***
  - **Verificar satisfacción de requerimientos**
  - **Asegurar cumplimiento de estándares**
  - **Fomentar uniformidad**
  - **Hacer proyectos más manejables**

Revisiones de software
guías de acción

- **Definir tamaño, conformación y duración**
- **Revisar producto, no productor**
- **Establecer agenda**
- **Limitar debates y rebates**
- **Enunciar problemas, no resolverlos**
- **Llevar registro**

Revisiones de software
guías de acción

- **Limitar tamaño del grupo**
- **Exigir preparación previa**
- **Definir *checklists***
- **Asignar recursos**
- **Entrenar a los revisores**
- **Revisar las revisiones**

Inspecciones Fagan

- **Desarrolladas en 1972, por Michael Fagan**
- **Beneficios: prevención y reducción de defectos, y por ende de costos**
- **Proceso de inspección en 6 etapas**

Inspecciones Fagan
etapas

- **Planeación -- preparación logística (materiales, disponibilidades, fechas)**
- ***Overview* -- conocimiento general, asignación de roles (autor, lector, *tester*, moderador)**
- **Preparación -- estudio del material a ser inspeccionado**
- **Inspección -- búsqueda de defectos**
- ***Rework* -- hacer las correcciones**
- **Iteración -- verificación de lo realizado, y de la no introducción de efectos secundarios**

Medición de calidad

- Visión del usuario
  - confiabilidad (mtbf)
  - usabilidad (instalación, aprendizaje, uso)
- Visión del productor
  - defectos
  - *rework*

Norma ISO 9126

- Estándar *reciente* para medir calidad
- 6 características básicas, descompuestas en subcaracterísticas
- Refleja una visión de usuario, mientras el modelo de McCall refleja una visión de producto

Norma ISO 9126

- Funcionalidad -- adecuación, precisión, interoperabilidad, seguridad
- Confiabilidad - madurez, tolerancia a fallas, recuperación de fallas
- Usabilidad -- facilidad de comprensión, facilidad de aprendizaje, facilidad de operación

Norma ISO 9126

- Eficiencia -- comportamiento temporal, comportamiento de recursos
- Facilidad de mantención -- facilidad de análisis, facilidad de cambio, estabilidad, facilidad de prueba
- Portabilidad -- facilidad de instalación, conformidad, facilidad de reemplazo

Análisis costo/beneficio de SQA

- Costos de SQA -- prevención y evaluación
- Beneficios de SQA -- asociados a disminución de *rework* (menos defectos y mayor confiabilidad)
- Importante que beneficios superen a los costos
- Implicancia: para ganar hay que invertir

Conceptos básicos de testing

- Una falla (*failure*) ocurre cuando un programa no se comporta adecuadamente - representa una propiedad del sistema en ejecución
- Un defecto (*fault*) existe en el código - si se encuentra puede causar una falla - no hay defecto si el sistema no puede fallar
- Un error (*error*,*mistake*) es una acción humana que resulta en software que contiene un defecto - un error puede llevar a incluir un defecto en el sistema, haciéndolo fallar
- Nunca se puede probar que un programa no fallará, solo se puede probar que contiene defectos

Conceptos básicos de testing

- Un test exitoso es aquel que encuentra muchos defectos, no lo opuesto - por lo tanto, desarrollo exitoso puede llevar a testing no exitoso (no encuentra defectos)
- El propósito del testing es encontrar defectos - es un proceso destructivo, se debe mostrar que algo es incorrecto - no es recomendable probar las propios desarrollos
- Cada vez que se corrigen defectos detectados, se pueden introducir nuevos defectos al sistema

Conceptos básicos de testing

- Testing es el proceso de ejecutar un programa con la intención de encontrar defectos - proceso destructivo que determina el diseño de los casos de prueba y la asignación de responsabilidades
- Testing exitoso: aquel que descubre defectos
  - caso de prueba *bueno*: aquel que tiene una alta probabilidad de detectar un defecto aun no descubierto
  - caso de prueba *exitoso*: aquel que detecta un defecto aun no descubierto

Conceptos básicos de testing

- Testing no es:
  - demostración de que no hay errores
  - demostración de que el software desempeña correctamente sus funciones
  - establecimiento de *confianza* que un programa hace lo que debe hacer
- Visión más apropiada de testing: proceso destructivo de tratar de encontrar defectos (cuya presencia se asume!) en un programa

Ejercicio

- Considere el siguiente programa

  un programa lee tres valores enteros, que representan las longitudes de los lados de un triángulo, e imprime un mensaje que establece si el triángulo es escaleno, isósceles, o equilátero
- Escriba un conjunto de casos de prueba que prueben adecuadamente este programa. Cada caso de prueba es de la forma (x,y,z) donde x,y,z representan los lados del triángulo

Principios de testing

- Una parte necesaria de un caso de prueba es la definición de la salida o resultado esperado
- Un programador debería evitar probar su propio código
- Una unidad de programación no debería probar sus propios programas
- Los resultados de cada prueba deben ser acuciosamente inspeccionados
- Los casos de prueba deben diseñarse para condiciones de entrada inválidas e inesperadas, no solo para aquellas válidas y esperadas

Principios de testing

- Examinar un programa para ver si no hace lo que debe hacer es solo la mitad de la tarea; la otra mitad es ver si hace lo que no debe hacer
- Evitar casos de prueba *espontáneos* y que no dejan registro - es solo pérdida de tiempo
- No planificar un esfuerzo de prueba bajo el supuesto tácito que no se encontrará defectos
- La probabilidad de existencia de más defectos en una sección de un programa es proporcional al número de defectos ya detectados en dicha sección
- Testing es una tarea extremadamente creativa e intelectualmente desafiante

Aspectos económicos

- **Es posible probar un programa para encontrar todos los defectos? ---- no, ni siquiera para los programas más triviales..**
- **En general, es impráctico y a menudo imposible encontrar todos los defectos en un programa**
  - **habría que probar todas las combinaciones de entrada, correctas e incorrectas ..... número infinito de casos de prueba**
  - **habría que probar todos los caminos posibles dentro de un programa, que pueden contener loops ... el número de casos de prueba no es infinito, pero usualmente puede considerarse como tal**

Aspectos económicos

- **La mayor parte de los costos de software corresponde a los costos de los defectos: diseño de las pruebas para descubrirlos, ejecución de dichas pruebas, detección del defecto, corrección del defecto**

Proceso de testing

- **Planificación**
  Al comienzo del desarrollo, se establecen guías, métodos, y niveles de ambición, si será automático o manual, estimación de recursos requeridos, estándares involucrados
- **Identificación**
  Estimación más detallada de los recursos requeridos
- **Especificación**
  Descripción de las pruebas a nivel funcional (propósito), y a nivel detallado (ejecución paso a paso)
- **Ejecución**
  Desarrollo de las pruebas tanto automatizadas como manuales, registrando los resultados

Proceso de testing

- **Análisis de defectos**
  Identificación del defecto y sus causas, y de las acciones correctivas
- **Completación**
  Preparación del equipamiento y los casos de prueba para uso posterior, registro de documentación

Aspectos esenciales del proceso de testing

- **La calidad del proceso de testing determina el éxito del esfuerzo de testing**
  testing tiene su ciclo propio, que comienza con la fase de requerimientos del software y desde allí va en paralelo con el proceso de desarrollo de software; para cada fase del proceso de desarrollo existe una tarea de testing importante
- **Prevención de la migración de los defectos utilizando técnicas de prueba temprano en el ciclo de vida**
  más de la mitad de los defectos se introducen usualmente en la fase de requerimientos, y el costo de los defectos se minimiza si se detectan en la misma fase en que son introducidos; inspecciones y revisiones periódicas constituyen una herramienta efectiva y rentable

Aspectos esenciales del proceso de testing

- **El momento para las herramientas de testing es ahora**
  existe una amplia variedad de productos para elegir, algunos más apropiados, o más fáciles de usar que otros, para distintas plataformas, para determinar cobertura estructural; es importante contar con una estrategia para adquirir herramientas, y procedimientos para seleccionarlas, que aunque se basan en el sentido común deben aplicarse sistemáticamente
- **Una persona *real* debe ser responsable por mejorar el proceso de testing**
  buenas especificaciones, revisiones e inspecciones afectan positivamente la calidad de las pruebas; su mejoramiento no es conceptualmente difícil, pero requiere esfuerzo y tiempo, por lo que se requiere una responsabilidad clara que planifique y gestione el avance

Aspectos esenciales del proceso de testing

- **Testing es una disciplina profesional que requiere gente entrenada y competente**
  - **para tener éxito se requiere de profesionales competentes y entrenados con el apoyo adecuado de la administración superior; no debe ser tratado como un nivel de entrada o *trampolín*; debe ser independiente, imparcial, y organizado para que cuente con el reconocimiento justo de su contribución a la calidad del producto**
- **Se debe cultivar una actitud de equipo positiva para la destrucción creativa**
  - **se necesita considerable creatividad para destruir algo en una forma controlada y sistemática; una buena prueba debe desarmar un producto metódicamente, encontrar sus debilidades, empujar hacia los límites**

Estado del arte v/s estado de la práctica

- **Testing todavía no es parte fundamental de la formación profesional, ni en la universidad ni en la industria**
- **Mucha investigación no probada en la vida real, sin consideraciones de retornos sobre la inversión**
- **Muchos métodos probados no utilizados en la industria**
- **Desarrollo de software es caro, muchos errores, plazos excedidos, altos costos de mantención, productos más complejos y críticos, más del 50% del esfuerzo de desarrollo frecuentemente ocupado en testing**

Estado del arte v/s estado de la práctica (cont)

- **Incicialmente, testing era *debugging***
- **Durante los 80, aparecen estándares (IEEE, ANSI, ISO)**
- **En los 90, aparecen las herramientas no solo como algo útil sino que vital para probar los sistemas que hoy se desarrollan**
- **A pesar de los avances, el proceso de testing todavía es muy inmaduro prácticamente en todas partes. Si se considera que los problemas a resolver hoy son más complejos y críticos, que las plataformas son más complejas también, no es ninguna sorpresa la dificultad para producir métodos, herramientas y profesionales efectivos, haciendo el testing cada vez más difícil**

Establecimiento de una perspectiva práctica

- **El software lo desarrollan las personas, y éstas cometen errores; no se puede prevenir completamente la introducción de estos defectos, pero sí se puede trabajar para localizarlos, especialmente los más críticos**
- **Decisiones de pruebas deberían basarse en satisfacción del cliente -- este es el objetivo último**
- **Mayor parte de los defectos se concentran en las etapas tempranas del proceso de desarrollo, y el costo de corrección aumenta a medida que permanece no detectado**
- **De *perogrullo*: los defectos más costosos son aquellos que no se detectan**

Actitud adecuada

- **Foco en la *cacería* de defectos**
- **Esfuerzo positivo y creativo de destrucción**
- **Persecución de defectos, no gente**
- **Agregación de valor**

Cómo se detectan defectos?

- **Examinando estructuras y diseños internos**
- **Examinando interfaces usuarias**
- **Examinando objetivos de diseño**
- **Examinando requerimientos de los usuarios**
- **Ejecutando código**

Taxonomía de defectos

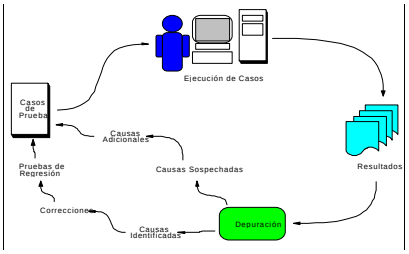
- La importancia de un defecto depende de la frecuencia, el costo de corrección, el costo de instalación, y sus consecuencias
- No hay una manera universalmente correcta de categorizar defectos; las siguientes categorías son típicamente utilizadas
  - requerimientos
  - características y funcionalidad
  - estructura
  - datos
  - implementación y código
  - integración
  - arquitectura del sistema y del software
  - definición y ejecución de pruebas
  - otros

Testing v/s debugging

- El propósito del testing es mostrar que un programa tiene defectos, mientras el propósito del *debugging* es encontrar el error o mala concepción que llevó a la falla, y diseñar e implementar los cambios que corrijan el error
- El *debugging* usualmente sigue al testing
- El testing lo puede hacer un externo, el *debugging* no

Debugging

- Consecuencia del buen testing
- Difícil ligar síntoma y causa



Segunda parte

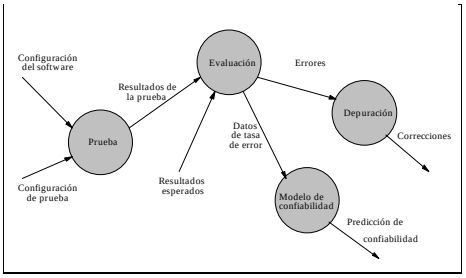
Fundamentos

testing: definiciones, objetivos, casos de prueba
diseño de casos de prueba (*black-box, white-box*)
cobertura de *white-box* testing
estrategias, métodos particulares de testing, *testability*, *cleanroom*
plan, especificación, ejecución y evaluación de pruebas

Testing de software

- Objetivo: demoler el producto de software con el propósito de detectar defectos
- Es una actividad esencialmente destructiva, pero que requiere mucha creatividad
- Consiste en ejecutar un programa con la intención de encontrar un defecto

Testing de software



Principios de testing

- Todas las pruebas deben ser trazables a requerimientos del usuario
- Las pruebas deben ser planificadas mucho antes de comenzarlas
- La ley de Pareto aplica al testing de software
- Testing debería comenzar en pequeña escala y progresar
- Testing exhaustivo no es posible
- Para mayor efectividad, testing debería ser desarrollado por una tercera parte independiente

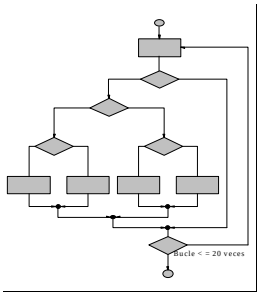
Caso de prueba

- Un buen *test case* es aquel que tiene una alta probabilidad de encontrar un defecto no descubierto, no aquel en que el programa funciona correctamente
- Un buen *test case* no es redundante, ni muy simple, ni muy complejo .... idealmente, el *mejor de su clase*
- Un exitoso *test case* es aquel que descubre un defecto no descubierto
- El diseño de *test cases* es muy difícil

Testing no puede mostrar la ausencia de defectos, solo puede mostrar que los defectos están presentes

Diseño de casos de prueba

- Caja blanca
  - caminos básicos
  - estructuras de control
  - otros
- Caja negra
  - clases de equivalencia
  - análisis de fronteras
  - otros



Diseño de casos de prueba

- Existen dos estrategias fundamentales
  - *white-box* testing o pruebas de caja blanca
    - » pretende un examen en detalle de los procedimientos internos, ejecutando caminos lógicos con condiciones o loops específicos
  - *black-box* testing o pruebas de caja negra
    - » las pruebas se conducen a nivel de la interfaz, sin mayor interés por los detalles internos de estructura lógica

A simple vista da la impresión que *white-box* testing puede entregar programas 100% correctos, si se definen y ejecutan todos los caminos (*path*) lógicos, es decir, se prueba exhaustivamente la lógica del programa ....

Diseño de casos de prueba

- Ejercicio

Un programa en C de 100 líneas, contiene 2 *loops* anidados que ejecutan de 1 a 20 veces cada uno, dependiendo de ciertas condiciones especificadas en la entrada. Dentro del *loop* interior, se requieren 4 constructores *if-then-else*.

Calcule cuántos caminos posibles existen (y que deberían ejercitarse para probar exhaustivamente el programa)?

Suponga que un *procesador mágico* es capaz de desarrollar un caso de prueba, ejecutarlo, y evaluar el resultado en solo un milisegundo, y que trabaja 24 horas diarias durante los 365 días del año. Cuánto tiempo se necesita para probar exhaustivamente dicho programa?

Diseño de casos de prueba

- *White-box* testing no debería descartarse por impráctico. Un número limitado de caminos lógicos importantes puede seleccionarse y ejercitarse, y puede probarse la validez de importantes estructuras de datos
- Los atributos de *black-box* y *white-box* testing se pueden combinar para proveer un enfoque que valide la interfaz y selectivamente asegure que los aspectos internos son los correctos

Black-box testing

- Los siguientes métodos son comúnmente utilizados
  - particionamiento de equivalencias (*equivalence partitioning*)
  - análisis de valores frontera (*boundary-value analyisis*)
  - adivinanza de defectos (*error guessing*)
- Los siguientes son menos utilizados
  - diagramas de causa-efecto (*cause-effect graphing*)
  - pruebas sintácticas (*syntax* testing)
  - pruebas de transición de estado (*state transition* testing)
  - matriz de grafos (*graph matrix*)

Black-box testing
equivalence partitioning

- Es un proceso sistemático que identifica un conjunto de clases de pruebas representativas de grandes conjuntos de otras pruebas posibles; la idea es que el producto bajo prueba se comportará de la misma manera para todos los miembros de la clase
- Dos pasos
  - identificar las clases de equivalencia
  - identificar los casos de prueba

Black-box testing
equivalence partitioning

- Identificar las clases de equivalencia
  - rango de valores válidos: se define una clase válida (dentro del rango) y dos inválidas (en cada lado fuera del rango)
  - número de valores válidos: idem
  - conjunto de valores válidos: una clase válida (dentro del conjunto) y una clase inválida (fuera)
  - si se cree que cada input válido es manejado de manera diferente, crear una clase por cada input válido
  - si se especifica una situación *must be*, crear una clase válida y una inválida

Black-box testing
equivalence partitioning

- Identificar los casos de prueba
  - asignar un número único a cada clase
  - hasta cubrir todas las clases válidas, diseñar un caso de prueba nuevo que cubra la mayor cantidad de clases aun no cubiertas
  - hasta cubrir todas las clases inválidas, diseñar un caso de prueba nuevo que cubra solo una clase inválida no cubierta
  - si múltiples clases inválidas se prueban con la misma prueba, algunas pruebas pueden no ejecutarse ya que la primera prueba puede enmascarar otras pruebas o terminar la ejecución del caso de prueba

Black-box testing
equivalence partitioning

- Ejercicio: GOLFSCORE
  - para la especificación de diseño funcional entregada en hoja aparte determine las clases de equivalencia

Black-box testing
equivalence partitioning

- Ejercicio

Black-box testing
boundary-value analysis

- Es una variante del particionamiento de equivalencias en que, en vez de seleccionar cualquier elemento como representativo de una clase de equivalencia, se seleccionan los *bordes* de una clase
- Además, se definen clases de equivalencia de salida, no solo de entrada como en el caso anterior
- Guía
  - rango o número de valores válidos: casos de prueba para los extremos y valores inválidos en su vecindad
  - ejemplo 1: rango 0.0 a 90.0, probar para 0.0, 90.0, -0.001 y 90.001
  - ejemplo 2: número entre 3 y 8, probar para 2, 3, 8, y 9

Black-box testing
boundary-value analysis

- Ejercicio: GOLFSCORE
  - para la especificación de diseño funcional entregada en hoja aparte determine una lista de valores frontera a ser probados

Black-box testing
boundary-value analysis

- Ejercicio

Black-box testing
error guessing

- Enfoque *ad hoc*, basado en intuición y experiencia, para identificar pruebas que probablemente expondrán defectos
- Idea básica: lista de defectos posibles o situaciones propensas a error, desarrollo de pruebas basadas en la lista
- *Historias* de los defectos pueden ser muy útiles
- Posibilidades
  - listas o strings nulos/vacíos
  - cero ocurrencias/instancias
  - caracteres blancos o nulos en strings
  - números negativos

Black-box testing
error guessing

- Ejercicio

Black-box testing
cause-effect graphing

- Enfoque sistemático para seleccionar conjuntos de casos de prueba de alto rendimiento que exploran combinaciones en las condiciones de entrada
- Derivación de las pruebas
  - descomponer la especificación en partes funcionales
  - identificar causas y sus efectos
  - crear grafos (booleanos) de causa efecto
  - registrar restricciones describiendo combinaciones de causas y/o efectos imposibles
  - convertir el grafo en una tabla de decisión de entrada limitada trazando condiciones de estado en el grafo; cada columna representa un caso de prueba
- Util para especificación funcional, difícil de implementar

Black-box testing
otros

- *Syntax* testing
  - enfoque sistemático para generar datos de entrada a un programa válidos e inválidos, aplicable a programas que tienen un lenguaje *oculto* para definir los datos (comandos interactivos a sistemas operativos o subsistemas), usualmente inefectivo para lenguajes explícitos y desarrollados formalmente
- *State transition* testing
  - Método analítico, que utiliza máquinas de estado finito, para diseñar pruebas para programas que tienen muchas funciones de control similar pero sutilmente diferentes, útil para verificación de diseño funcional
- *Graph matrix*
  - Es solo una representación más simple de un grafo para organizar los datos

Black-box testing
state transition testing

- Defectos potenciales
  - número incorrecto de estados
  - transición incorrecta para una combinación dada estado-entrada
  - salida incorrecta para una salida dada
  - pares de estados o conjuntos de estados que se hacen equivalentes inadvertidamente (factor perdido)
  - estados o conjuntos de estados que se dividen para crear duplicados no equivalentes
  - estados o conjuntos de estados que se han trasnformado en *muertos*
  - estados o conjuntos de estados que se han transformado en inalcanzables

Black-box testing
state transition testing

- Puede ser impráctico recorrer cada *path* en un diagrama de transición de estados
- La noción de cobertura indica ejercitar cada transición
- Aunque en muchos casos se puede recorrer todos los estados en un solo caso de prueba, es recomendable
  - definir un conjunto de secuencias de entrada que vuelvan al estado inicial
  - para cada paso en cada secuencia de entrada definir el próximo estado esperado, la transición esperada, y la salida esperada
- Entonces, un conjunto de pruebas consiste de tres conjuntos de secuencias: secuencias de entrada, transiciones correspondientes, y secuencias de salida

Black-box testing
state transition testing

- Ejercicio

White-box testing

- Utiliza la estructura de control del diseño procedural para derivar casos de prueba que
  - garanticen que todos los caminos independientes dentro de un módulo han sido ejercitados por lo menos una vez
  - ejerciten todas las decisiones lógicas en sus lados verdaderos y falsos
  - ejecuten todos los loops en sus fronteras y dentro de sus límites operacionales
  - ejerciten sus estructuras de datos internas para asegurar su validez

White-box testing

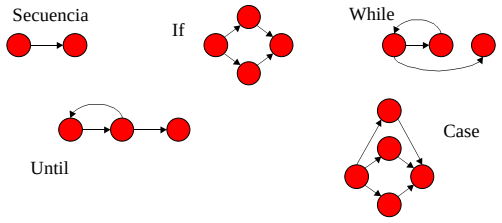
- Razones para hacer *white-box* testing
  - los errores lógicos y supuestos incorrectos son inversamente proporcionales a la probabilidad de ejecutar un camino en un programa
  - a menudo se cree que un camino lógico probablemente no se ejecutará cuando de hecho puede ser ejecutado en forma regular
  - los errores tipográficos son aleatorios
- Los tipos de errores indicados arriba pueden no ser detectados vía *black-box* testing

White-box testing
basis path testing

- ***Basis path*** **testing o pruebas de caminos básicos es una técnica que permite derivar una medición de complejidad lógica de un diseño procedural y usar esta medición como una guía para definir un conjunto básico de caminos (*path*) de ejecución**
- **Los casos de prueba derivados para ejercitar el conjunto básico garantizan la ejecución de cada instrucción en el programa al menos una vez durante las pruebas**

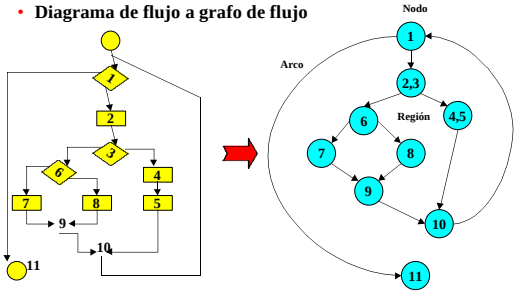
White-box testing
basis path testing

- **Notación de grafo de flujo**



White-box testing
basis path testing

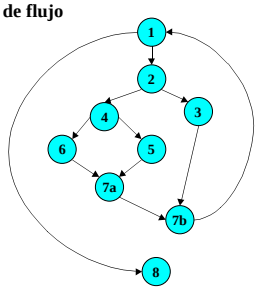
- **Diagrama de flujo a grafo de flujo**



White-box testing
basis path testing

- **Traducción de PDL a grafo de flujo**

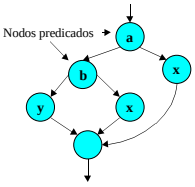
```
PDL

  procedure: sort

1:  do while records remain
      read  record;
2:    if record field 1 = 0
3:      then process record;
            store in buffer;
            increment counter;
4:    elseif record field 2 = 0
5:      then reset counter;
6:      else process record;
            store in  file;
7a:   endif
      endif
7b: enddo
8: end
```



White-box testing
basis path testing

- **Traducción de PDL a grafo de flujo (condiciones compuestas)**

```
.....
if a or b
  then procedure x
  else procedure y
endif
.......
```

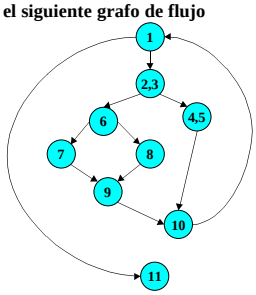


White-box testing
basis path testing

- **Complejidad ciclómatica es una métrica de software que provee una medición cuantitativa de la complejidad lógica de un programa**
- **Usada en el contexto de testing, define el número de caminos independientes en el conjunto básico y entrega un límite superior para el número de casos necesarios para ejecutar todas las instrucciones al menos una vez**
- **Un camino independiente es cualquier camino a través del programa que introduce al menos un nuevo conjunto de instrucciones o una nueva condición; en términos de un grafo de flujo, debe incluir al menos un arco no utilizado antes de definir el camino**

White-box testing
basis path testing

- Considere el siguiente grafo de flujo



White-box testing
basis path testing

- Un conjunto de caminos independientes es el siguiente:
  - path 1: 1-11
  - path 2: 1-2-3-4-5-10-1-11
  - path 3: 1-2-3-6-8-9-10-1-11
  - path 4: 1-2-3-6-7-9-10-1-11
- Cada nuevo camino introduce un arco nuevo
- El siguiente camino
  - 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

  no es considerado independiente porque es una combinación de caminos ya especificados y no utiliza nuevos arcos
- Los 4 caminos constituyen un conjunto básico

White-box testing
basis path testing

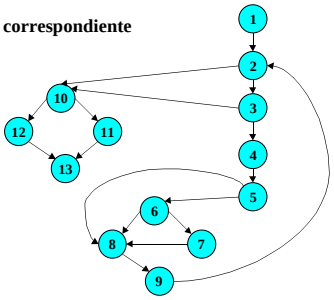
- Si se puede diseñar casos de prueba para forzar la ejecución de estos caminos, cada instrucción del programa se ejecutará al menos una vez y cada condición se habrá ejecutado en su lado verdadero y falso
- El conjunto básico no es único
- Cómo se calcula la complejidad ciclomática?
  - número de regiones
  - V(G) = e - n + 2, e son los arcos y n los nodos
  - V(G) = p + 1, p son los nodos predicados
- En ejemplo anterior en todos los casos el resultado es 4

White-box testing
basis path testing

- Ejemplo en PDL para derivar casos de prueba
  - ver procedimiento AVERAGE entregado en hoja aparte

White-box testing
basis path testing

- Grafo de flujo correspondiente



White-box testing
basis path testing

- Conjunto básico de caminos linealmente independientes
  - path 1: 1-2-10-11-13
  - path 2: 1-2-10-12-13
  - path 3: 1-2-3-10-11-13
  - path 4: 1-2-3-4-5-8-9-2- ....
  - path 5: 1-2-3-4-5-6-8-9-2- ....
  - path 6: 1-2-3-4-5-6-7-8-9-2- ....
- (....) significa que cualquier camino de lo que resta de la estructura de control servirá
- Nodos 2, 3, 5, 6, y 10 son nodos predicados
- Complejidad ciclomática: 6

White-box testing
basis path testing

- Casos de prueba
  - path 1: *value*(k) = input válido, donde k < i definido abajo
    value(i) = -999, donde 2 <= i <= 100
    resultados esperados: *average* correcto basado en k valores y *totals* apropiados
    caso 1 debe probarse como parte de los casos 4, 5, y 6
  - path 2: value(1) = -999
    resultados esperados: *average* = -999, otros *totals* al valor inicial
  - path 3: intentar procesar 101 o más valores, los primeros 100 valores deben ser válidos
    resultados esperados: los mismos del caso 1

White-box testing
basis path testing

- Casos de prueba
  - path 4: *value*(i) = input válido, donde i < 100
    value(k) < minimum, donde k < i
    resultados esperados: *average* correcto basado en k valores y *totals* apropiados
  - path 5: *value*(i) = input válido, donde i < 100
    value(k) > maximum, donde k <= i
    resultados esperados: *average* correcto basado en n valores y *totals* apropiados
  - path 6: *value*(i) = input válido, donde i < 100
    resultados esperados: *average* correcto basado en n valores y *totals* apropiados

White-box testing
basis path testing

- Cada caso de prueba se ejecuta y se compara con los resultados esperados. Una vez que todos los casos de prueba han sido completados, el *tester* puede estar seguro que todos las instrucciones han sido ejecutadas al menos una vez
- Algunos caminos independientes (como el *path* 1) requieren ser probados como parte de otros caminos, para generar la combinación de datos necesarios

White-box testing
basis path testing

- Ejercicio

White-box testing
control structure testing

- *Control structure testing* o pruebas de estructuras de control son las siguientes
  - pruebas de caminos básicos (*basis path* testing)
  - pruebas de condiciones (*condition* testing)
  - pruebas de flujo de datos (*data flow* testing)
  - pruebas de *loop* (*loop* testing)

White-box testing
condition testing

- Es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en un módulo de programa
- Tipos de error:
  - operador booleano (or, and, not)
  - variable booleana (p, q, r)
  - paréntesis booleano ()
  - operador relacional (<, =, >, ≠)
- El método se focaliza a probar cada condición en un programa
- Si un conjunto de pruebas es efectivo para detectar defectos en las condiciones, probablemente también lo es para detectar otros defectos

White-box testing
condition testing

- Una estrategia de prueba de condiciones es *branch testing* o prueba de ramificación, en que para una condición compuesta cada rama verdadera y falsa y cada condición simple es ejecutada al menos una vez
- Otra estrategia es prueba de dominio o *domain testing* que requiere 3 o 4 pruebas para una expresión relacional de la forma E_1(rel-op)E_2
- 3 casos de prueba ($E_1 > E_2$, $E_1 = E_2$, $E_1 < E_2$) se requieren para detectar un defecto en el operador relacional, asumiendo que E_1 y E_2 son correctos
- Para una expresión booleana con n variables, se requieren 2^n pruebas para detectar defectos en los operadores booleanos, variables booleanas, y paréntesis booleanos, pero solo es práctico para n pequeño

White-box testing
condition testing

- Una estrategia de prueba llamado BRO *(branch and relational operator)* testing garantiza la detección de defectos en las ramificaciones o en los operadores relacionales de una condición, siempre que todas las variables booleanas y operadores relacionales ocurran solo una vez y no tengan variables comunes
- Esta estrategia utiliza restricciones de condición para una condición dada, de la forma (D1, D2, D3, ..., Dn) para una condición con n condiciones simples, donde Di es un símbolo que especifica una restricción en el resultado de la i-ésima condición simple

White-box testing
condition testing

- Una restricción de condición D para una condición C se dice que está cubierta por una ejecución de C si durante esta ejecución el resultado de cada condición simple en C satisface la restricción correspondiente en D
- Para una variable booleana, la restricción en el resultado establece que debe ser *true* (t) o *false* (f)
- Para expresiones relacionales, los símbolos <, =, > se usan para especificar restricciones en el resultado de la expresión

White-box testing
condition testing

- Ejemplos de BRO testing
  - condición C_1: B_1 & B_2 , B_1 y B_2 variables booleanas

    set de restricciones a cubrir: (t,t), (f,t), (t,f), pues si la condición es incorrecta debido a uno o más errores de operadores booleanos, al menos un miembro del set de restricciones la forzará a fallar
  - condición C_2: B_1 & ($E_3 = E_4$) , B_1 variable booleana, E_i expresiones aritméticas

    set de restricciones a cubrir: (t,=), (f,=), (t,<), (t,>) pues si la condición es incorrecta debido a uno o más errores de operadores booleanos o relacionales, al menos un miembro del set de restricciones la forzará a fallar
  - condición C_3: ($E_1 > E_2$) & ($E_3 = E_4$) , E_i expresiones aritméticas

    set de restricciones a cubrir: (>,=), (=,=), (<,=),(>,<), (>,>) pues si la condición es incorrecta debido a uno o más errores de operadores relacionales, al menos un miembro del set de restricciones la forzará a fallar

White-box testing
condition testing

- Ejercicio

White-box testing
data flow testing

- Es un método de diseño de casos de prueba que selecciona caminos de un programa de acuerdo a la ubicación de las definiciones y usos de variables en él
- Suponga que a cada instrucción en un programa se le asigna un número de instrucción único y que cada función no modifica sus parámetros o variables globales
- Para una instrucción cuyo número es S

  DEF (S) = { X| instrucción S contiene una definicion de X}

  USE (S) = {X | instrucción S contiene un uso de X}
- Si la instrucción S es un *if* o un *loop*, su DEF es vacío y su USE se basa en la condición de la instrucción S

White-box testing
data flow testing

- La definición de variable X en la instrucción S se dice que está viva (*live*) en la instrucción S' si existe un camino desde la instrucción S a la instrucción S' que no contiene otra definición de X
- Una cadena definición-uso (DU *chain*) de la variable X es de la forma [X, S, S'], donde S y S' son números de instrucciones, X está en DEF(S) y USE(S'), y la definición de X en la instrucción S está viva en la instrucción S'

White-box testing
data flow testing

- Una estrategia simple es requerir que cada DU *chain* se cubra por lo menos una vez, estrategia conocida como DU testing. No hay garantías de cubrir todas las ramas de un programa, aunque esto es muy raro (*if-then-else* con *then* sin definición de variables y *else* inexistente, en cuyo caso la rama *else* no está garantizado que se cubra)
- Esta estrategia es util para seleccionar caminos de prueba de un programa que contengan instrucciones *if* o *loops* anidados

White-box testing
data flow testing

- Ejemplo de aplicación de DU testing

```
proc x
    B1;
    do  while C1
                if C2
                            then
                                        if C4
                                                    then B4;
                                                    else B5;
                                        endif;
                            else
                                        if C3
                                                    then B2;
                                                    else B3;
                                        endif;
                endif;
    enddo;
    B6;
end proc;
```

White-box testing
data flow testing

- Se requiere saber las definiciones y usos de variables en cada condición o block
- Suponga que la variable X es definida en la última instrucción de los blocks B1, B2, B3, B4 y B5 y es usada en la primera instrucción de los blocks B2, B3, B4, B5 y B6
- La estrategia DU testing requiere la ejecución del camino más corto de cada Bi (0 < i <= 5) a cada Bj (1 < j <= 6); también cubre el uso de la variable X en las condiciones C1, C2, C3, y C4
- Aunque hay 25 DU *chains* de la variable X, solo se requieren 5 caminos

White-box testing
data flow testing

- La razón es que 5 caminos se requieren para cubrir las DU *chains* de X desde Bi, 0 < i <= 5, a B6, y otras DU *chains* pueden ser cubiertas mediante iteraciones del *loop*
- Para seleccionar caminos que probar utilizando *branch* testing strategy no se requiere información adicional; para utilizar BRO testing se necesita conocer la estructura de cada condición o block
- Dado que las instrucciones de un programa están relacionadas entre sí de acuerdo a las definiciones y usos de variables, el enfoque de *data flow* testing es muy efectivo, aunque la selección de caminos y cálculo de cobertura es mas complejo que en el caso de *condition* testing

White-box testing
data flow testing

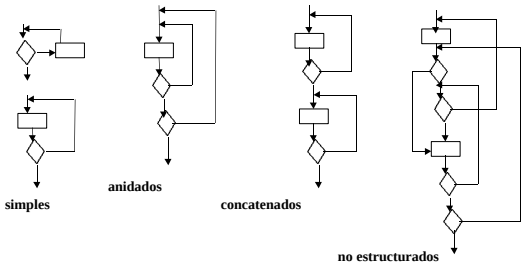
- Ejercicio

White-box testing
loop testing

- Corresponde a una técnica que se focaliza exclusivamente a la validez de las construcciones de *loops*
- 4 clases de *loops* pueden definirse
  - simples
  - concatenados
  - anidados
  - no estructurados

White-box testing
loop testing

- Ejemplos de las clases de *loops*



White-box testing
loop testing

- *Loops* simples (n es el máximo de iteraciones)
  - debe generarse el siguiente conjunto de pruebas
    - » saltarse el *loop* completamente
    - » solo una iteración a través del *loop*
    - » dos iteraciones a través del *loop*
    - » m iteraciones a través del *loop*, donde m < n
    - » n - 1, n, n + 1 iteraciones a través del *loop*

White-box testing
loop testing

- *Loops* anidados
  - la estrategia anterior haría crecer geométricamente el número de casos de prueba, transformándose en impráctica. Un enfoque para reducir el número de casos de prueba es el siguiente
    - » comenzar en el *loop* interior, fijar los otros *loops* en valores mínimos
    - » conducir pruebas de *loop* simple para el *loop* interno, manteniendo los otros *loops* fijos con el valor mínimo de su parámetro de iteración; agregar pruebas para valores fuera del rango o excluidos
    - » trabajar hacia fuera, conduciendo pruebas para el siguiente *loop*, pero manteniendo todos los otros *loops* externos fijos en valores mínimos, y otros *loops* internos en valores *típicos*
    - » continuar hasta que todos los *loops* hayan sido probados

White-box testing
loop testing

- *Loops* concatenados
  - si cada *loop* es independiente de los otros, utilizar la estrategia para *loops* simples
  - si el parámetro de iteración de uno depende de otro, estos *loops* están concatenados pero no son independientes; utilizar el enfoque de *loops* anidados
- *Loops* no estructurados
  - en la medida de lo posible, estos *loops* deben ser rediseñados

White-box testing
loop testing

- Ejercicio

White-box testing
test coverage

- **Cobertura (*test coverage*)**
  - **la ejecución de un caso de prueba considera ciertos requerimientos, utiliza ciertas partes de la funcionalidad, y ejercita ciertas partes de la lógica interna de un programa**
  - **hay que asegurarse de tener suficientes pruebas de cada uno de estos niveles**
  - **tiene tres componentes**
    - » **cobertura de requerimientos**
    - » **cobertura funcional**
    - » **cobertura lógica**
  - **Forma simplista de entenderlo: % de instrucciones ejecutadas por un conjunto de pruebas**
  - **Observación general: la probabilidad de encontrar nuevos defectos es inversamente proporcional a la cantidad de código no cubierto**

White-box testing
test coverage

- **Los métodos *white-box* se utilizan para aumentar la cobertura lógica**
- **Existen las siguientes formas básicas de cobertura lógica**
  - **cobertura de instrucción (statement coverage)**
    - » **cada instrucción es ejecutada al menos una vez**
  - **cobertura de decisión (ramificación) (decision (branch) coverage)**
    - » **cada instrucción es ejecutada al menos una vez, cada decisión toma todos los resultados posibles al menos una vez**
  - **cobertura de condición (condition coverage)**
    - » **cada instrucción es ejecutada al menos una vez, cada condición en una decisión toma todos los resultados posibles al menos una vez**

White-box testing
test coverage

- **Existen las siguientes formas básicas de cobertura lógica**
  - **cobertura de decisión/condición (decision/condition coverage)**
    - » **cada instrucción es ejecutada al menos una vez, cada decisión toma todos los resultados posibles al menos una vez, cada condición en una decisión toma todos los resultados posibles al menos una vez**
  - **cobertura de múltiples condiciones (multiple/condition coverage)**
    - » **cada instrucción es ejecutada al menos una vez, todas las combinaciones posibles de resultados de condiciones en cada decisión ocurre al menos una vez**
  - **cobertura de camino (path coverage)**
    - » **cobertura exhaustiva es generalmente impráctica**

White-box testing
test coverage

- **Ejercicio: LIABILITY**

```
procedure liability (age, sex, married, premium);
begin
   premium:=500;
   if ((age < 25) and (sex = male) and (not married)) then
                        premium :=premium + 1500;
   else (if (married or (sex = female)) then premium := premium - 200;
         if ((age > 45) and (age < 65)) then premium :=premium -100;)
end;
```

White-box testing
test coverage

- **Ejercicio: LIABILITY**
  - **listar un conjunto posible (mínimo) de casos de prueba que satisfaga cada uno de los criterios de cobertura**

Casos especiales de testing

- **Testing GUIs**
  - **las siguientes preguntas pueden utilizarse para crear una serie de pruebas genéricas**
    - » **ventanas**
      - **se abren apropiadamente basado en comandos digitados o de menú?**
      - **se pueden modificar su tamaño, mover y *scroll*?**
      - **todos los contenidos dentro de la ventana son apropiadamente direccionables con un *mouse*, teclas funcionales, flechas de dirección, y teclado?**
      - **se regenera apropiadamente cuando es sobreescrita y despues re-llamada?**
      - **están todas las funciones disponibles cuando se necesitan?**
      - **están todas las funciones operacionales?**
      - **están disponibles y se despliegan apropiadamente los menús, barras, diálogos, botones, iconos, y otros controles?**
      - **cuando se despliegan múltiples ventanas, se representan apropiadamente sus nombres?**

Casos especiales de testing

- **Testing GUIs**
  - » **ventanas (cont)**
    - **se destaca apropiadamente la ventana activa?**
    - **en caso de multitareas, se actualizan todas las ventanas en momentos apropiados?**
    - ***clicks* del *mouse* múltiples o incorrectos dentro de la ventana causan efectos laterales no esperados?**
    - ***prompts* de audio o color en la ventana o a consecuencia de operaciones de la ventana, se presentan de acuerdo a las especificaciones?**
    - **cierran apropiadamente?**

Casos especiales de testing

- **Testing GUIs**
  - » **menús y operaciones del *mouse***
    - **se despliega la barra de menú apropiada y en el contexto apropiado?**
    - **despliega la barra de menú de aplicación características del sistema? (ej. reloj)**
    - **funcionan apropiadamente las operaciones de *pull-down*?**
    - **funcionan apropiadamente los menús descomponibles, paletas y barras de herramientas?**
    - **están apropiadamente listadas las funciones y subfunciones desplegables de los menús?**
    - **son todas las funciones de menú direccionables apropiadamente por el *mouse*?**
    - **es correcto el tipo, tamaño y formato del texto?**
    - **es posible invocar cada función de menú usando su comando de texto alternativo?**
    - **se destacan las funciones de menú según el contexto de la operación actual dentro de una ventana?**

Casos especiales de testing

- **Testing GUIs**
  - » **menus y operaciones del mouse (cont)**
    - **se desempeña cada función del menú según se anuncia?**
    - **son los nombres de los menús autoexplicativos?**
    - **existe ayuda disponible para cada item de menú, y es sensible al contexto?**
    - **se reconocen las operaciones del *mouse* en forma apropiada a través del contexto interactivo?**
    - **si se requieren múltiples *clicks*, se reconocen apropiadamente en el contexto?**
    - **si el *mouse* tiene múltiples botones, se reconocen apropiadamente en el contexto?**
    - **cambian apropiadamente el cursor, indicador de procesamiento, puntero según se invocan diferentes operaciones?**

Casos especiales de testing

- **Testing GUIs**
  - » **entrada de datos**
    - **la entrada alfanumérica se despliega e ingresa al sistema apropiadamente?**
    - **funcionan adecuadamente los modos gráficos de entrada de datos?**
    - **se reconocen apropiadamente los datos inválidos?**
    - **son entendibles los mensajes de entrada de datos?**

Casos especiales de testing

- **Testing de arquitecturas cliente servidor**

  representan un desafío significativo, dada la naturaleza distribuida de sus ambientes, los aspectos de rendimiento asociados con procesamiento de transacciones, la presencia potencial de un número significativo de plataformas diferentes de hardware, la complejidad de las comunicaciones de redes, la necesidad de servir a múltiples clientes desde una base de datos centralizada (o distribuida), y los requerimientos de coordinación impuestos en el servidor; todo esto se combina para hacer del testing de estas arquitecturas y el software que en ellas reside considerablemente mas difícil que probar aplicaciones independientes, lo que es indicativo de aumentos significativos de tiempos y costos de prueba

Casos especiales de testing

- **Testing de documentación y facilidades de ayuda**
  - **puede enfrentarse en dos fases: la primera mediante revisiones técnicas formales, y la segúnda, prueba *viva*, mediante el uso conjunto de la documentación con el programa real**
  - **la prueba *viva* puede enfrentarse utilizando un enfoque similar al de *black-box* testing discutido**
  - **el uso del programa se analiza a traves de la documentación**
    - » **describe con precisión la documentacion cómo cumplir cada modo de uso?**
    - » **es precisa la descripción de cada secuencia de interacción?**
    - » **son precisos los ejemplos?**
    - » **son consistentes la terminología, descripciones de menú, y respuestas del sistema con el programa real?**

Casos especiales de testing

- **Testing de documentación y facilidades de ayuda**
  - **el uso del programa se analiza a traves de la documentación (cont)**
    - » **es relativamente fácil localizar guías dentro de la documentación?**
    - » **se pueden resolver problemas fácilmente con la documentación?**
    - » **son precisos la tabla de contenidos y el índice de la documentación?**
    - » **es el diseño de la documentación (*layout*, tipos, indentación, gráfica) conducente a la comprensión y rápida asimilación de información?**
    - » **todos los mensajes de error desplegados para el usuario, están descritos en mayor detalle en el documento?**
    - » **si se usan enlaces de hipertexto, son precisos y completos?**

Estas pruebas deben hacerlas usuarios *selectos* e independientes

Casos especiales de testing

- **Testing de sistemas de tiempo real**
  - **en adición a los métodos descritos, debe considerarse la manipulación de eventos (i.e. procesamiento de interrupciones), el *timing* de los datos, y el paralelismo de las tareas (procesos) que manejan los datos**
  - **además debe considerarse la relación existente entre software de tiempo real y el hardware, lo que causa tambien muchos problemas de testing**
  - **se ha propuesto una estrategia de 4 pasos**
    - » **pruebas de tareas independientes - descubre defectos en lógica y función, pero no en *timing* o comportamiento**
    - » **pruebas de comportamiento - examen del comportamiento como consecuencia de eventos externos**

Casos especiales de testing

- **Testing de sistemas de tiempo real**
  - **se ha propuesto una estrategia de 4 pasos (cont)**
    - » **pruebas intertareas - descubre defectos de sincronización de tareas que se comunican**
    - » **pruebas de sistema - hardware y software integrado, se descubren defectos de interfaz entre ellos**

Casos especiales de testing

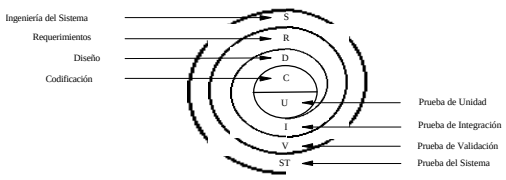
- **Testing de sistemas de tiempo real**
  - **tambien se generan casos de prueba para las siguientes características de los sistemas asociadas con las interrupciones**
    - » **se asignan y se manejan apropiadamente las prioridades de interrupción?**
    - » **se maneja correctamente el procesamiento de cada interrupción?**
    - » **conforma con los requerimientos el desempeño (tiempo de procesamiento) de cada procedimiento de manejo de interrupciones?**
    - » **crea problemas en función o desempeño en momentos críticos el alto volumen de interrupciones?**

Estrategias de testing

- **Una estrategia de testing debe incorporar planificación, diseño de casos de prueba, ejecución, recolección de datos y evaluación**
- **El testing es un conjunto de actividades que pueden planificarse por adelantado y conducirse sistemáticamente**
- **Diversas estrategias de testing proveen *templates* con las siguientes características genéricas**
  - **comenzar en el nivel más bajo y trabajar hacia fuera hasta integrar completamente el sistema**
  - **diferentes técnicas son apropiadas en diversas circunstancias**
  - **las pruebas las conduce el desarrollador y un grupo independiente**
  - **el *debugging* es reconocido como una actividad diferente**

Estrategias de testing

- **Una estrategia de testing puede ser vista como una espiral en el que la cronología de eventos es inversa a la que se da en el desarrollo de software**



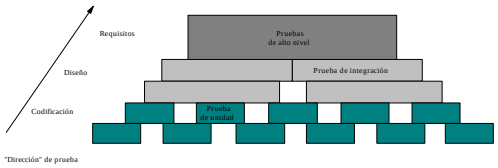
Estrategias de testing

- **Inicialmente, las pruebas se focalizan en cada módulo individualmente, asegurando que funciona apropiadamente como una unidad; principalmente se utilizan técnicas de *white-box***
- **Después, los módulos deben integrarse o ensamblarse para formar un paquete completo; las técnicas de *black-box* son las más importantes**

Estrategias de testing

- **Finalmente, corresponde una serie de pruebas de orden superior orientadas a asegurar que el software satisfaga todas los requerimientos de uso, funcionales, de comportamiento y de desempeño, y a verificar que todos los elementos del sistema computacional se acoplan adecuadamente y que la funcionalidad y desempeño globales se alcanzan**

Estrategias de testing



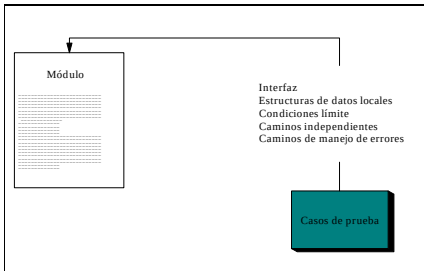
Estrategias de testing

- **Para una implementación exitosa de una estrategia de testing se debe**
  - **especificar requerimientos del producto en forma cuantificable mucho antes de comenzar el testing**
  - **establecer explícitamente los objetivos del testing**
  - **comprender a los usuarios y desarrollar un perfil para cada categoría**
  - **desarrollar un plan que enfatice pruebas de ciclo rápido**
  - **construir software robusto que se diseñe para probarse a sí mismo**
  - **usar revisiones técnicas formales como un filtro previo al testing**
  - **conducir revisiones técnicas formales para evaluar la estrategia de testing y los casos de prueba**
  - **desarrollar un enfoque de mejoramiento continuo para el proceso de testing**

Estrategias de testing

- ***Black-box* y *white-box* testing v/s cobertura**
  - **pruebas de requerimientos (cobertura de requerimientos) y funcionales (cobertura funcional) deberían utilizar *black-box* testing**
  - **pruebas internas (cobertura lógica) deben emplear *white-box* testing**
- **En todos los casos las pruebas deben definir sus resultados y ser repetibles**

Testing unitario



Testing unitario

- Ejemplo de *checklist* para pruebas de interfaz
  - número de parámetros de entrada igual al número de argumentos?
  - calzan los atributos de parámetros y argumentos?
  - número de argumentos transmitidos a módulos llamados igual al número de parámetros?
- Categorías de error para estructuras de datos locales
  - tipos inconsistentes o impropios
  - valores erróneos de inicialización o *default*
  - nombres de variable incorrectos
  - tipos de datos inconsitentes
  - *underflow*, *overflow*, excepciones de direccionamiento

Testing unitario

- Categorías de error de computación (cálculo)
  - precedencia aritmética incorrecta o mal entendida
  - operaciones modales mixtas
  - inicializaciones incorrectas
  - imprecisión
  - representaciones simbólicas incorrectas
- Categorías de error de comparaciones o flujo de control
  - comparación de tipos de datos diferentes
  - precedencia u operadores lógicos incorrectos
  - expectativas de igualdad, errores de precisión la hacen improbable
  - comparaciones o variables incorrectas
  - terminaciones de *loop* impropias o no existentes, iteraciones divergentes
  - variables de *loop* modificadas inapropiadamente

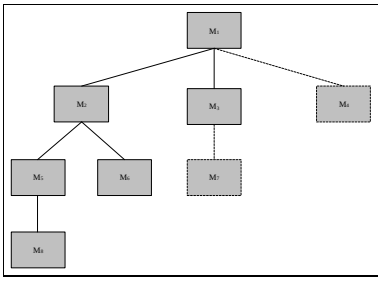
Testing unitario

- Categorías de error en el manejo de los errores
  - descripción de error no es entendible
  - error notado no corresponde al error encontrado
  - condición de error causa intervención del sistema antes de manejar el error
  - procesamiento de condiciones excepcionales es incorrecto
  - descripción de error no es informativa para ayudar a localizar su causa
- Consideraciones
  - *overhead* asociado con módulos *stubs* y *drivers*
  - módulos con alta cohesión simplifican las pruebas unitarias

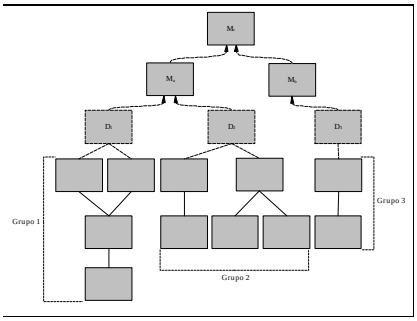
Testing de integración

- Es una técnica sistemática para construir la estructura del programa mientras se conducen pruebas para descubrir defectos asociados a las interfaces, para lo que se toman módulos probados (testing unitario) y se construye una estructura de programa de acuerdo al diseño
- Cualquier estrategia de integración debe ser incremental, para lo que existen dos esquemas principales
  - integración *top-down*
  - integración *bottom-up*

Integración top-down



Integración bottom-up



Testing de integración

- En general, las ventajas de ambos enfoques tienden a ser desvemtajas del otro
- Integración *top-down*
  - ventaja: probar tempranamente funciones principales
  - desventaja: necesidad de *stubs*
- Integración *bottom-up*
  - ventaja: diseño más fácil de casos de prueba, no uso de *stubs*
  - desventaja: el programa como entidad no existe hasta el final
- Un buen compromiso puede ser un enfoque combinado

Testing de integración

- Finalmente, es importante indentificar módulos críticos según se conducen las pruebas de integración
  - consideran varios requerimientos de software
  - tienen altos niveles de control (*arriba* en la estructura de programa)
  - son complejos o propensos a los errores (potencial indicador: V(G))
  - tienen requerimientos de desempeño
- Los módulos críticos deben ser probados lo más tempranamente posible

Testing de integración

- Estructura de una especificación de prueba
  - ámbito de testing
  - plan de testing
    - » fases de prueba e incrementos
    - » calendario
    - » software adicional
    - » ambiente y recursos
  - procedimiento de prueba n (descripción para incremento n)
    - » orden de integración
      - propósito
      - módulos a ser probados

Testing de integración

- Estructura de una especificación de prueba
  - » pruebas unitarias para módulos en el incremento
    - descripción de prueba para módulo n
    - descripción de software adicional
    - resultados esperados
  - » ambiente de prueba
    - herramientas o técnicas especiales
    - descripción de software adicional
  - » casos de prueba
  - » resultados esperados para incremento n
  - resultados reales de las pruebas
  - referencias
  - apéndices

Testing de regresión

- Es la actividad que ayuda a asegurar que los cambios no introduzcan comportamientos no pretendidos o defectos adicionales
- Los casos de prueba para regresión (*regression test suite*) corresponden a tres clases diferentes
  - un conjunto representativo de pruebas para ejercitar todas las funciones del software
  - pruebas adicionales que se focalizan en funciones probablemente afectadas por el cambio
  - pruebas que se focalizan en los componentes que han cambiado

Testing de regresión

- Dado que el número de pruebas puede crecer demasiado, y no es posible ejecutar cada prueba para cada función después de un cambio, estas pruebas deben diseñarse de tal manera de incluir solo pruebas de clases de errores de las principales funciones
- Las pruebas de regresión se pueden ejecutar manualmente o usando herramientas automatizadas de captura y ejecución (*capture-playback tools*)

Testing de usabilidad

- Involucra el hacer que los usuarios trabajen con un producto de software y observar sus respuestas a él
- Históricamente ha sido uno de los muchos componentes de las pruebas de sistema ya que están basados en requerimientos, pero la tendencia es hacia darle un rol más prominente
- Características a probar
  - accesibilidad - puede el usuario entrar, navegar, salir con facilidad?
  - respuesta - puede hacer lo que desea, cuando desea, en forma clara?
  - eficiencia - puede hacerlo con un mínimo uso de recursos?
  - comprensibilidad - entiende la estructura, ayudas, y documentación?

Testing de validación o aceptación

- Objetivo
  - determinar si el funcionamiento satisface al cliente/usuario
- Pruebas alfa
  - conducidas en laboratorios del desarrollador por el usuario final
- Pruebas beta
  - conducidas en ambientes de trabajo *reales* por usuarios finales, generalmente *amistosos*; la prueba se desarrolla antes de la distribución general

Testing de sistemas

- Objetivo: determinar si el sistema en su globalidad opera satisfactoriamente (recuperación de fallas, seguridad y protección, *stress*, performance)

Criterios de completación de testing

- ¿Cómo saber cuándo dejar de hacer testing?
- Normalmente, se abandona al agotarse los recursos
- En la práctica el testing nunca acaba, solo cambia de *tester* ... del profesional al usuario
- Idealmente, utilizar modelos de fallas de software como función del tiempo de ejecución, basados en modelación estadística y teorías de confiabilidad para poder hacer afirmaciones como la siguiente:

  .... hemos hecho suficiente testing para afirmar con un 95% de confianza que la probabilidad de 1000 horas de CPU de operación libre de fallas en un ambiente probabilísticamente definido es por lo menos 0.995 ....

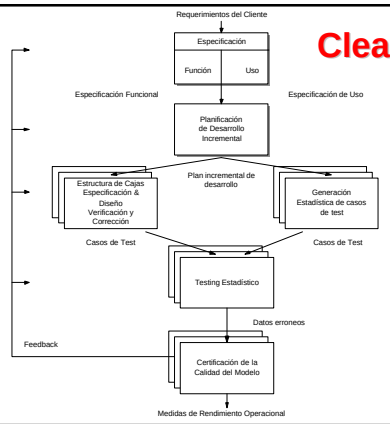
Testability

- Representa cuán fácilmente se puede probar un programa
- Algunas características deseables
  - operabilidad - mejor funciona, más eficientemente se puede probar
  - observabilidad - lo que ve es lo que prueba
  - controlabilidad - mejor se controla, más se puede automatizar y optimizar
  - descomponibilidad - ámbito controlado, más rápidamente se puede aislar porblemas y desarrollar *re-testing* más inteligente
  - simpleza - menos que probar, más rápidamente se hace
  - estabilidad - menos cambios, menos problemas
  - comprensibilidad - más información, mejor se prueba

Cleanroom

- Fundamentos teóricos
  - especificación y diseño formal
  - prueba de correctitud
  - testing estadístico

Cleanroom



Plan de pruebas

- **Consideraciones para generar un plan de pruebas**
  - **métodos de testing**
  - **infraestructura (para desarrollo de *testware* y ejecución de las pruebas)**
  - **automatización de las pruebas**
  - **software de apoyo**
  - **administración de la configuración**
  - **riesgos (presupuesto, recursos, plazos, entrenamiento, etc)**
- **Estas consideraciones aplican al esfuerzo global y a cada actividad individual de testing**
- **Se requiere un plan global, y uno detallado para cada actividad de testing (pruebas unitarias, de integración, de usabilidad, funcionales, de sistema, de aceptación)**

Plan de pruebas

- **Plan de pruebas**
  - **propósito**
    - **prescribir el ámbito, enfoque, recursos, y plazos de las actividades de testing**
  - **esquema**
    - **identificador**
    - **introducción**
    - **ítemes de prueba**
    - **características a ser probadas**
    - **características a no ser probadas**
    - **enfoque**
    - **criterios de aprobación por ítem**
    - **criterios de suspensión y requerimientos de continuación**

Plan de pruebas

- **Plan de prueba**
  - **esquema (cont)**
    - ***deliverables* de testing**
    - **tareas de testing**
    - **necesidades ambientales**
    - **necesidades de personal y entrenamiento**
    - **calendario**
    - **riesgos y contingencias**
    - **aprobaciones**

Plan de pruebas

- **Ejercicio**

Especificación de pruebas

- **Un complemento importante del plan de pruebas es el diseño de la arquitectura de las pruebas, es decir una estructura significativa y útil de las pruebas como un todo**
- **Debe existir una especificación de arquitectura por producto, que puede verse como el documento *raíz* del repositorio de prueba**
- **Consideraciones básicas**
  - **organización de las pruebas (requerimientos, función, internas)**
  - **categorización de las pruebas y convenciones de agrupación**
  - **estructura y convenciones de nombres para el repositorio**
  - **agrupación de pruebas para períodos razonables de ejecución**

Especificación de pruebas

- Especificación de arquitectura de prueba
  - esquema
    - » identificador
    - » introducción
    - » ubicación del repositorio
    - » convenciones de almacenamiento y acceso del repositorio
    - » estructura/organización del repositorio
    - » estándares
    - » convenciones de agrupamiento y nombres para todos los archivos

Especificación de pruebas

- Ejercicio

Especificación de pruebas

- A continuación se debe diseñar en detalle e implementar las pruebas
- Consideraciones
  - identificar los ítemes a ser probados
  - asignar prioridades a estos ítemes, basado en riesgo
  - desarrollar diseños de pruebas de alto nivel para grupos de ítemes de prueba relacionados
  - diseñar casos de prueba individuales desde los diseños de alto nivel
- Una especificación de diseño de pruebas ayuda a identificar casos de prueba, para los que se usa una especificación
- La implementación traduce la especificación de casos de prueba en casos de prueba listos para ejecutar

Especificación de pruebas

- Especificación de diseño de pruebas
  - propósito
    - » especificar refinamientos del enfoque de prueba e identificar las características a ser cubiertas por el diseñoy sus pruebas asociadas; también identifica los casos de prueba y los procedimientos de prueba, si existen, requeridos para cumplir las pruebas y especifica los criterios de aprobación/reprobación de las características
  - esquema
    - » identificador
    - » características a ser probadas
    - » refinamientos del enfoque
    - » identificación de casos de prueba
    - » criterios de aprobación/reprobación de las características

Especificación de pruebas

- Ejercicio

Especificación de pruebas

- Especificación de caso de prueba
  - propósito
    - » definir un caso de prueba identificado por la especificación de diseño de pruebas; documenta los valores reales usados como entrada junto con los resultados esperados; identifica restricciones en los procedimientos de prueba resultantes del uso de ese caso de prueba específico; los casos de prueba se separan de los diseños de prueba para permitir su uso en más de un diseño y permitir su reuso
  - esquema
    - » identificador
    - » ítemes de prueba
    - » especificaciones de entrada
    - » especificaciones de salida
    - » necesidades ambientales
    - » requermientos procedurales especiales
    - » dependencias intercasos

Especificación de pruebas

- **Ejercicio**

Especificación de pruebas

- **Especificación de procedimiento de prueba**
  - **propósito**
    - » **identificar todos los pasos requeridos para operar el sistema y ejercitar los casos de prueba especificados para implementar los diseños de prueba asociados; los procedimientos se separan de las especificaciones de diseño de prueba ya que se espera que sean seguidos paso a paso**
  - **esquema**
    - » **identificador**
    - » **propósito**
    - » **requerimientos especiales**
    - » **pasos del procedimiento**

Especificación de pruebas

- **Ejercicio**

Ejecución de pruebas

- **A continuación se debe ejecutar las pruebas, desarrollando las siguientes tareas**
  - **selección de casos de prueba**
  - **setup previo, ejecución, análisis posterior**
  - **registro de actividades, resultados e incidentes**
  - **determinar si las fallas se deben a errores en el producto o en las pruebas**
  - **medir cobertura lógica interna**
- **Organizaciones maduras exigen que ciertos criterios se cumplan antes de autorizar la ejecución de pruebas, como una forma de evitar costosas pérdidas de tiempo**
- **Potenciales *deliverables* son *logs* de pruebas, informes de incidentes, e informes de cobertura lógica (automático)**

Ejecución de pruebas

- ***Logs* de prueba**
  - **propósito**
    - » **proveer un registro cronológico de detalles relevantes acerca de la ejecución de las pruebas**
  - **esquema**
    - » **identificador**
    - » **descripción**
    - » **entradas por actividades y eventos**

Ejecución de pruebas

- **Ejercicio**

Ejecución de pruebas

- Informes de incidentes de prueba
  - propósito
    - » documentar cualquier evento de la ejecución de las pruebas que requiera mayor investigación
  - esquema
    - » identificador
    - » resumen
    - » descripción de incidentes
    - » impacto

Ejecución de pruebas

- Ejercicio

Evaluación de pruebas

- Finalmente se debe evaluar las pruebas, mediante
  - evaluación de cobertura
  - evaluación de defectos
  - evaluación de efectividad de las pruebas
- Resultado de esta evaluación lo constituyen
  - pruebas adicionales
  - decisiones de detención del esfuerzo de testing
- Preguntas claves
  - se continua o se detiene las pruebas?
  - qué pruebas adicionales se requieren, en caso de continuar?
  - cómo se crea el informe sumario final, en caso de detener las pruebas?

Tercera parte

Gestión

enfoques organizacionales, prácticas, tendencias, desafíos

mejoramiento de procesos, costos y beneficios, mediciones

herramientas prácticas en el mercado, estándares, documentación de testing

Enfoques organizacionales

- Una organización de test (grupo de test) es un recurso o conjunto de recursos dedicados a desarrollar actividades de prueba
- Requiere adecuada gestión
  - comprender y evaluar procesos, estándares, políticas, herramientas, entrenamiento, mediciones de testing de software
  - mantener una organización de prueba fuerte, independiente, formal e imparcial
  - reclutar y retener profesionales calificados
  - liderar, comunicar, apoyar, controlar
  - tiempo para proveer el cuidado necesario para gestionar grupos de test
- Requiere estructura organizacional adecuada

Enfoques organizacionales

- Elementos estructurales de diseño organizacional
  - alto o plano
    - » niveles que separan la toma de decisión de la acción
  - mercado o producto
    - » servir a diferentes mercados o a diferentes productos
  - centralizado o descentralizado
    - » aspecto clave para la organización de testing
  - jerárquico o difuso
    - » niveles sucesivos de autoridad y rango, o *esparcido*
  - línea o staff
    - » mezcla de roles de línea o staff
  - funcional o proyecto
    - » orientación por funciones o por proyecto

Enfoques organizacionales

- **Testing es la responsabilidad de cada persona**
  - **lo que ocurre a menudo en el mundo real, cuando la compañía es pequeña y no se ha reflexionado mucho sobre testing. Existe un grupo de desarrolladores cuya principal responsabilidad es construir un producto, pero también son responsables de probar su propio código**
    - » **ventaja: parece una solución natural, obvia**
    - » **desventaja: existe una incapacidad inherente de probar efectivamente el trabajo propio**

Enfoques organizacionales

- **Testing es la responsabilidad de cada unidad**
  - **se asigna a los profesionales dentro de una unidad la tarea de probar el trabajo de otro. Existe un grupo de desarrolladores cuya principal responsabilidad es construir un producto, pero también son responsables de probar el código del compañero.**
    - » **ventajas: resuelve el problema de incapacidad de probar código propio**
    - » **desventajas: *ancho de banda*, i.e. pretender demasiado de una persona en términos metodológicos, competencias, entrenamiento**

Enfoques organizacionales

- **Testing es desarrollado por recursos dedicados**
  - **se asigna a un desarrollador la tarea de *tester* permanente, lo complejo es asignar a la persona adecuada a la tarea, y gestionar.**
    - » **ventajas: resuelve el problema de ancho de banda del desarrollador, además de que hay un equipo único**
    - » **deventajas: problema de ancho de banda en la administración, que debe proveer procesos, estándares, políticas, herramientas, entrenamiento, mediciones de testing de software**

A esta altura se hace evidente la necesidad de un grupo de testing, i.e. un conjunto de recursos dedicados para desarrollar actividades de testing y gestionados por un *test manager*. La pregunta es dónde ponerlo organizacionalmente

Enfoques organizacionales

- **Testing es parte de QA**
  - **una solución común es hacer que la nueva organización de testing sea parte de QA, que también audita el proceso de desarrollo. Puede ocurrir que el QA *manager* no entienda de testing de software, por lo que las capacidades de gestión del grupo de testing son críticas.**
    - » **ventajas: resuelve el problema de gestión creando una organización formal que es parte de QA**
    - » **desventajas: potenciales problemas para el trabajo en equipo, testing puede extraviarse en QA con otras responsabilidades, el grupo de desarrollo no es completamente responsable por la producción de productos de calidad**

Enfoques organizacionales

- **Testing es parte del desarrollo**
  - **trata de resolver problemas de trabajo en equipo y responsabilidad sobre la calidad creando una organización de testing que es parte de la organización de desarrollo.**
    - » **ventajas: resuelve problemas de gestión y trabajo en equipo asignando la organización de testing a la organización de desarrollo**
    - » **desventajas: satisface la gestión superior de desarrollolos requeirmientos de gestión de testing? problema de ancho de banda a este nivel**

Enfoques organizacionales

- **Organización centralizada de testing**
  - **se resuelve el problema de gestión del caso anterior creando una organización central de testing que depende de una división de desarrollo de producto, creando una oportunidad de impactar significativamente una organización**
    - » **ventajas: resuelve los problemas de gestión superior de desarrollo por medio de la centralización de la organización de testing, que es parte del desarrollo; además crea oportunidades de promoción**
    - » **desventajas: organización depende absolutamente de la división y su administración superior, potencial carencia de trabajo en equipo a nivel individual o de proyecto, potencial carencia de metodologías consistentes en diferentes divisiones**

Enfoques organizacionales

- Organización centralizada de testing con un centro tecnológico de test
  - intenta resolver los problemas de consistencia del caso anterior creando un grupo tecnológico, encargado de liderar y gestionar procesos de testing y esfuerzos de mejoramiento de productividad, conducir y coordinar programas de entrenamiento, coordinar la planificación e implantación de programas de herramientas de testing, documentar procesos de testing, así como estándares, políticas, y guías según se requiera, reorganizar y aprovechar las mejores prácticas de los diversos grupos, recomendar e implantar mediciones clave
    - » ventajas: resuelve problemas de consistencia creando un grupo tecnológico de testing
    - » desventajas: organización depende absolutamente de la división y su administración superior, potencial carencia de trabajo en equipo a nivel individual o de proyecto

Enfoques organizacionales

- Criterios de selección
  - habilidad para rápida toma de decisiones
  - mejoramiento del trabajo en equipo
  - independencia, formalidad, imparcialidad, fortaleza, adecuación de la organización de testing
  - coordinación de responsabilidades de testing y calidad
  - ayuda con los requerimientos de gestión de testing
  - *propiedad* de la tecnología de testing
  - aprovechamiento de los recursos disponibles, especialmente humanos
  - impacto en motivación y proyecciones profesionales

Dato importante

- Tasas *tester* a desarrollador
  - históricamente, 1:5-10
  - tendencias actuales, 1:3-4

Prácticas y desafíos

- Estudios de mejores prácticas en la industria indican que el éxito en testing depende menos de tecnologías *estado del arte*, herramientas, etc, que de aspectos no técnicos
- Las claves para el éxito son cosas básicas
  - buena administración
  - buena comunicación
  - buena gente
  - buenos controles
  - mediciones permanentes

Mejoramiento del proceso de testing

- Algunos datos (USA)
  - 6 a 8 defectos por cada 100 LOC
  - 12 a 20 defectos por cada 100 LOC (código no estructurado, pobremente documentado)
  - 2 a 3 defectos por cada 100 LOC (código estructurado y documentado)
  - 1 a 1.5 defectos por cada 100 LOC después de la *típica* prueba unitaria
  - 70% de defectos introducidos durante análisis y diseño
  - 4 a 6 defectos por 1000 LOC entregados a los clientes
- Qué se necesita para mejorar el proceso?
  - prácticas efectivas de gestión de testing
  - mejoramiento de técnicas para testing funcional (basado en requerimientos) y para testing estructural (basado en código)
  - testing de regresión efectivo

Prácticas efectivas de gestión de testing

- Planificación de pruebas
- Definición y gestión de objetivos de prueba (requerimientos)
- Monitoreo de avance de las pruebas
- Seguimiento de calidad de productos y pruebas
- Control de configuración de las pruebas

Control de costos

- Un objetivo fundamental es maximizar el rendimiento del testing, i.e. maximizar el potencial para detectar defectos y minimizar el número de pruebas realizadas
- Además de la detección de defectos, se agregan otros objetivos:
  - minimizar el costo de ejecutar las pruebas
  - minimizar el costo de mantener las pruebas
  - minimizar el costo de desarrollar las pruebas

Costo de ejecutar pruebas

- Incluye
  - costo de setup previo
    - » minimizar el tiempo y esfuerzo requerido para tareas como configuración de hardrware y software, establecimiento del ambiente de prueba, identificación de las pruebas a ejecutar
  - costo de ejecución
    - » minimizar el tiempo total de ejecución, asistido (con intervención manual) y no asistido; determinar cuánto testing de regresión se ejecutará (riesgo v/s costo, costo de selección de un subset v/s reducción de costo de ejecución)
  - costo posterior a la ejecución
    - » minimizar el tiempo y esfuerzo requerido para el análisis y documentación de los resultados de las pruebas, desarme del ambiente de prueba y restauración del ambiente original

Costo de ejecutar pruebas

- Recomendaciones
  - considerar el uso de hardware dedicado para las pruebas
  - automatizar configuración de software y ambiente de pruebas
  - automatizar la identificación y selección de las pruebas a ejecutar
  - automatizar lo más posible la ejecución de las pruebas
  - automatizar comparación de resultados y expectativas

Costo de mantener las pruebas

- Pruebas de regresión deben mantenerse bajo el control de un sistema de gestión de configuración
- Actividades clave de mantención
  - agregar una prueba para cada problema informado
  - agregar pruebas progresivas para probar nuevos cambios
  - revisar periódicamente todos los casos de prueba y su efectividad continua, esto es
    - » que cada caso de prueba es ejecutable
    - » que las pruebas de requerimientos funcionales son válidas
    - » que cada caso de prueba agrega valor

Costo de desarrollar las pruebas

- Requiere la misma disciplina que ingeniería de software en general
- Se debe planificar y comprender qué se debe hacer antes de diseñar
- Se debe usar inspecciones y revisiones tal como se haría con otros productos de software
- Se debe generar la documentación necesaria
- Principio básico: reusar si es posible, comprar si hay disponibilidad, desarrollar solo si es necesario

Mediciones de testing

- Algunas preguntas previas:
  - cuándo se puede detener el testing?
  - cuántos defectos se pueden esperar?
  - cuál técnica de testing es más efectiva?
  - se está probando con inteligencia?
  - no se detectan defectos .... se tiene un programa sólido o una prueba débil?

Mediciones de testing

- **Pueden servir para**
  - **cuál es el tamaño real del esfuerzo de testing para un producto dado?**
  - **cuán eficientes fueron los esfuerzos de verificación, de validación?**
  - **cuál fue la calidad del producto, durante las pruebas? en producción? comparado con otros productos?**
  - **cuántos defectos existen en el producto, antes de las pruebas? después en la operación?**
  - **cuándo se deja de hacer testing?**
- **Medición da una idea de la complejidad real de un programa, ayudando a planear el esfuerzo de prueba, a predecir cuántos errores existen y dónde pueden ocurrir**
- **Medición del número y tipo de defectos detectados provee una idea precisa de la eficiencia en la verificación y las debilidades del proceso de desarrollo**

Mediciones de testing

- **Medición de cobertura de pruebas provee una evaluación cuantitiva de su acuciosidad y alcance**
- **Monitoreo del estado de ejecución de las pruebas señala la convergencia de las categorías de prueba y provee información cuantitativa de cuándo dejar de hacer testing**
- **Las mediciones también**
  - **proveen un indicador importante de calidad del producto**
  - **proveen criterios significativos para *release readiness***
  - **se correlacionan con satisfacción del usuario**
  - **sirven como predictor del número de errores que quedan**
  - **normalizadas, permiten comparar**
  - **proveen una indicación de eficiencia**

Mediciones de testing

- **Las mediciones producen respuestas, pero se debe priorizar tomando como base qué es lo crítico y qué será realmente utilizado**
- **Mediciones útiles:**
  - **complejidad**
  - **eficiencia de verificación**
  - **cobertura de pruebas**
  - **estado de ejecución de pruebas**
  - **informes de incidentes (defectos)**
  - **basadas en informes de incidentes**

Herramientas de testing

- **El uso de herramientas de testing puede hacer el testing más fácil, más efectivo y más productivo**
- **Qué herramientas se necesitan?**
- **Existen muchas herramientas disponibles, que se refieren a múltiples aspectos del proceso de testing**
- **Se requiere una estrategia para evaluar, adquirir, entrenar, implantar y mantener herramientas**
- **Las herramientas se pueden caracterizar según la actividad en la que se emplea, la función específica que desarrolla, o por áreas de clasificación**
- **A continuación se presentan tipos de herramientas por actividad**

Herramientas de testing

- **Para revisiones e inspecciones**
  - **análisis de complejidad**
  - **comprensión de código**
  - **análisis sintáctico y semántico**
- **Para planificación de pruebas**
  - ***templates* para documentación de planes de prueba**
  - **estimación de esfuerzo y calendarización de pruebas**
  - **analizador de complejidad**
- **Para diseño y desarrollo de pruebas**
  - **generador de casos de prueba**
  - **diseño de prueba basado en requerimientos**
  - **captura/*playback***
  - **análisis de cobertura**

Herramientas de testing

- **Para ejecución de pruebas y evaluación**
  - **captura/*playback***
  - **análisis de cobertura**
  - **pruebas de memoria**
  - **administración de casos de prueba**
  - **simuladores y rendimiento**
- **Para soporte**
  - **administración de problemas**
  - **administración de la configuración**

Herramientas de testing

- **Adquisición**
  - establecer los costos verdaderos - licencia, instalación, operación, entrenamiento, mantención, soporte, reorganización de procedimientos
  - establecer cómo calza la herramienta en el procesode pruebas
  - establecer ámbito de necesidades de uso inicial
  - encontrar la herramienta correcta y el proveedor correcto

Herramientas de testing

- **Fuentes**
  - lista compilada por Software Development Technologies, consultora líder en herramientas de testing; disponible en Software TestWorld™, CD-ROM con herramientas actuales de testing
  - Internet

Estándares para testing

- **Claves**
  - IEEE/ANSI Standard 829-1983, Software Test Documentation
  - IEEE/ANSI Standard 1008-1987, Software Unit Testing
- **Otros relacionados**
  - IEEE/ANSI Standard 1012-1986, Software Verification and Validation Plans
  - IEEE/ANSI Standard 1028-1988, Software Reviews and Audits
  - IEEE/ANSI Standard 730-1989, Software Quality Assurance Plans
  - IEEE/ANSI Standard 610.12-1990, Glossary of Software Engineering Terminology

Bibliografía básica

- **The art of software testing, Glenford Myers. John Wiley 1979**
- **The complete guide to software testing, Bill Hetzel. John Wiley 1984, 2nd edition 1988**
- **Software testing techniques, Boris Beizer. Van Nostrand Reinhold Company Inc. 1983, 2nd edition 1990**
- **Software testing in the real world, improving the process, Edward Kit. Addison Wesley 1995**

Catálogos de herramientas de testing

- **Testing Tools Reference Guide**
  - Software Quality Engineering
- **Software Test Technologies Report**
  - Software Technology Support Center, Hill Air Force Base
- **Software TestWorld CD-ROM**
  - Software Development Technologies
- **Software Management Technology Reference Guide**
  - Software Maintenance News, Inc.
- **The CAST Report**
  - Cambridge Market Intelligence Ltd.
- **Ovum Evaluates: Software Testing Tools**
  - Ovum Ltd.

Otras fuentes

- **Conferencias**
  - International Software Testing Analysis & Review (STAR)
  - International Conference & Exposition on Testing Computer Software
  - International Software Quality Week (SQW)
  - International Conference on Software Testing
- **Journals y newsletters**
  - Testing Techniques Newsletter, Software Research Inc.
  - Software Testing, Verification and Reliability, Wiley and Sons Ltd.
  - Software Quality Management Magazine, Software Quality Engineering
  - Software Quality World, ProQual Inc.
  - Software Management News
- **Internet**
  - *newsgroup* comp.software.testing
  - múltiples *web sites*