UNIDAD 6

Ingeniería de Software

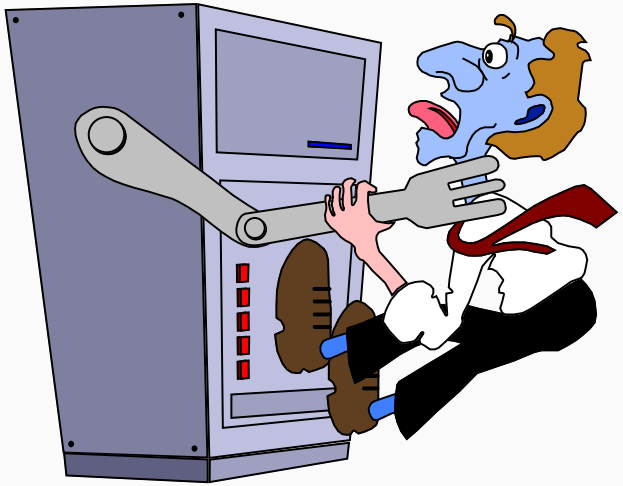
Testing de Software

v. 4.0

Conceptos De Testing de Software

¿Qué es un SW de “calidad”?

- *Aquel que cumpla con los requisitos*
- *Aquel que al ser usado NO tenga FALLAS*



Funcionamiento incorrecto

Aseguramiento de la Calidad

- *Un patrón planificado y sistemático de todas las acciones necesarias para brindar una adecuada confianza que un producto o componente cumple con los requerimientos técnicos establecidos*

- *La* **Prueba del Software** *es una de las actividades involucradas en el Aseguramiento de Calidad*

- **Aseguraniento de la Calidad** *implica la revisión y auditoría de los productos y actividades para verificar que cumplen los procedimientos y estándares aplicables y suministrando a los gerentes de proyecto y otros del resultado de estas revisiones y auditorías.*

Asegurar la Calidad vs Controlar la Calidad

Una vez definidos los requerimientos de calidad tengo que tener en cuenta que:

- La calidad no puede “inyectarse” al final
- La calidad del producto depende de tareas realizadas durante el proceso
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

Objetivo de la Prueba

¿Qué persigue el testing?

Básicamente, encontrar **fallas** en el producto

Hacerlo lo mas **eficiente** posible

* Lo mas rápido posible
* Lo mas barato posible

Hacerlo lo mas **eficazmente** posible

* Encontrar la mayor cantidad de fallas
* No detectar fallas que no son
* Encontrar las mas importantes

Prueba del SW

Según IEEE:

- Una actividad en la cual un sistema o componente es ***ejecutado*** bajo condiciones específicas, los resultados de dicha ejecución son observados o registrados y, a partir de los mismos, se realiza una evaluación de algún aspecto del sistema o componente

Deducimos que se trata de una actividad/proceso ***dinámico***

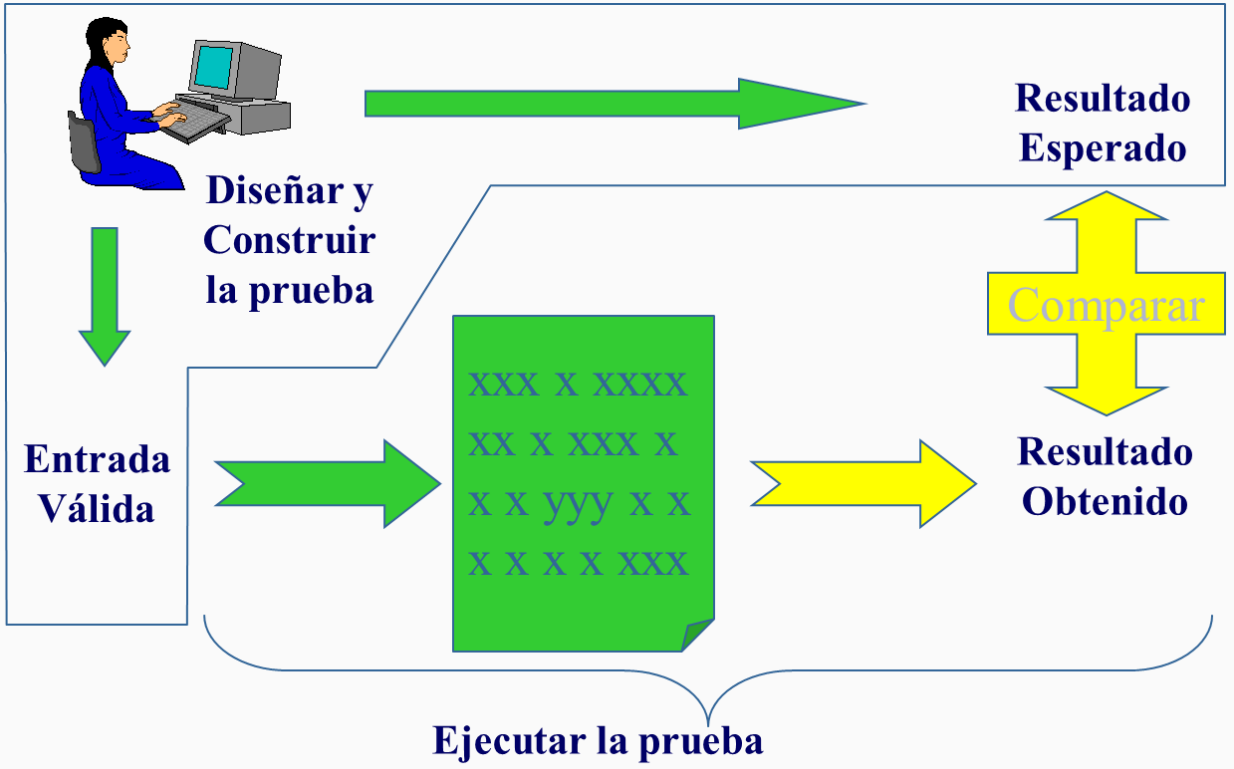
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

Prueba del SW

Informalmente …

- Probar es ejecutar un componente con el objetivo de producir fallas
- Una prueba es exitosa si encuentra fallas
  - *Si nuestro objetivo es mostrar que el SW no contiene errores, estaremos inconscientemente orientados a ese fin*

El Proceso de la Prueba del SW



El Proceso de la Prueba del SW

Las pruebas siempre se realizan contra un resultado esperado

Incidente de Testing

Según IEEE:

- Toda ocurrencia de un evento que sucede durante la ejecución de una prueba de software que requiere investigación

No toda incidencia es una falla

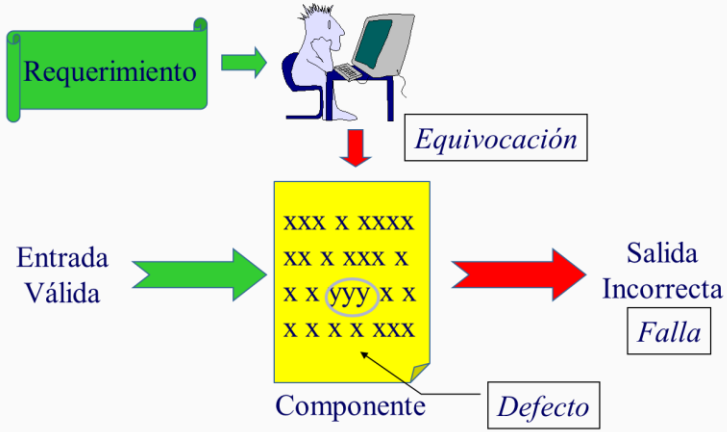
EJEMPLOS:

- Defectos en los casos
- Equivocaciones al ejecutar las pruebas
- Interpretaciones erróneas
- Dudas
- ... ...

Conceptos relacionados

Equivocación

Acción humana que produce un resultado incorrecto



Defecto

Paso, proceso o definición de dato incorrecto

Ausencia de cierta característica

Falla

Resultado de ejecución incorrecto. Es el producido por el SW distinto al resultado esperado

Conceptos relacionados

- *Una equivocación lleva a uno o mas defectos que están presentes en el código*
- *Un defecto lleva a cero, una o más fallas*
- *La falla es la manifestación del defecto*
- *Una falla tiene que ver con uno o mas defectos*

Conceptos relacionados

Condiciones de Prueba

Son descripciones de situaciones que quieren probarse ante las que el sistema debe responder

Crear condiciones es un proceso “creativo”

Casos de Prueba

Son lotes de datos necesarios para que se dé una determinada condición de prueba

Crear condiciones es un proceso “laborioso”

Conceptos relacionados

Criterio de Selección

Es una condición para seleccionar un conjunto de casos de prueba

De todas las combinaciones posibles, solo seleccionaremos algunas:

- *La menor cantidad de aquellas que tengan mayor probabilidad de encontrar un defecto no encontrado por otra prueba*

Partición

- Todos los posibles casos de prueba los dividimos en clases
- Todos los casos de una clase son equivalentes entre si → Detectan los mismos defectos
- Con solo ejemplos de cada clase cubrimos todas las pruebas
- El éxito está en la selección de la partición !!!

Proceso de Depuración

Depuración:

- Depurar es eliminar un defecto que posee el SW
- La depuración NO es una tarea de prueba aunque es consecuencia de ella
- La prueba detecta el falla (efecto) de un defecto (causa)

La depuración puede ser fuente de introducción de nuevos defectos

Proceso de Depuración

En la Depuración debemos:

- ***DETECTAR***
  - Dada la falla debemos hallar el defecto (dado el efecto debemos encontrar la causa)
- ***DEPURAR***
  - Encontrado el defecto debemos eliminarlo
  - Debemos encontrar la razón del defecto
  - Debemos encontrar una solución
  - Debemos aplicarla
- ***VOLVER A PROBAR***
  - Asegurar que sacamos el defecto
  - Asegurar que no hemos introducido otros (regresión)
- ***APRENDER PARA EL FUTURO***
  - Lecciones Aprendidas

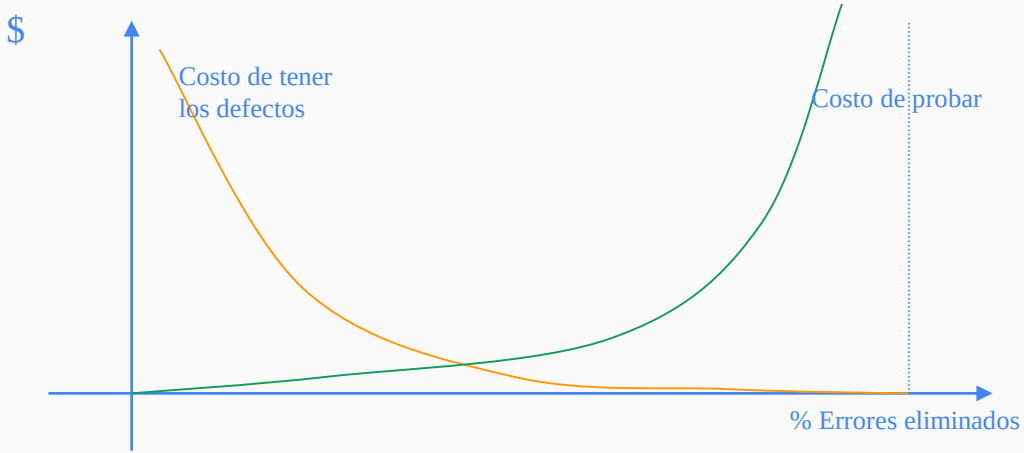
Economía del Testing

¿Hasta cuándo tengo que probar?

Se puede invertir mucho esfuerzo y tiempo en probar

PROBAR en definitiva es el proceso de establecer confianza en que un SW hace lo que se supone que tiene que hacer

Y ya que nunca se va a poder demostrar que un SW es correcto, continuar probando es una decisión económica



Economía del Testing

¿Cuándo detengo la prueba?

No hay una “única receta” y depende de cada situación en particular.

Algunos criterios pueden ser:

- *Para exitosamente el conjunto de pruebas que fue diseñado*
- ***“Good Enough”:*** *cierta cantidad de fallas no críticas es aceptable*
- *Cantidad de fallas detectadas es similar a la cantidad de fallas estimadas*

Economía del Testing

¿Cómo abarato la prueba?

- Hacer pruebas es caro y trabajoso
- La forma de abaratarlas y acelerarlas –sin degradar su utilidad- es:
  - DISEÑANDO EL SW PARA SER TESTEADO
- Algunas herramientas son:
  - *Diseño Modular*
  - *Ocultamiento de Información*
  - *Uso de Puntos de Control*
  - *Programación NO egoísta*

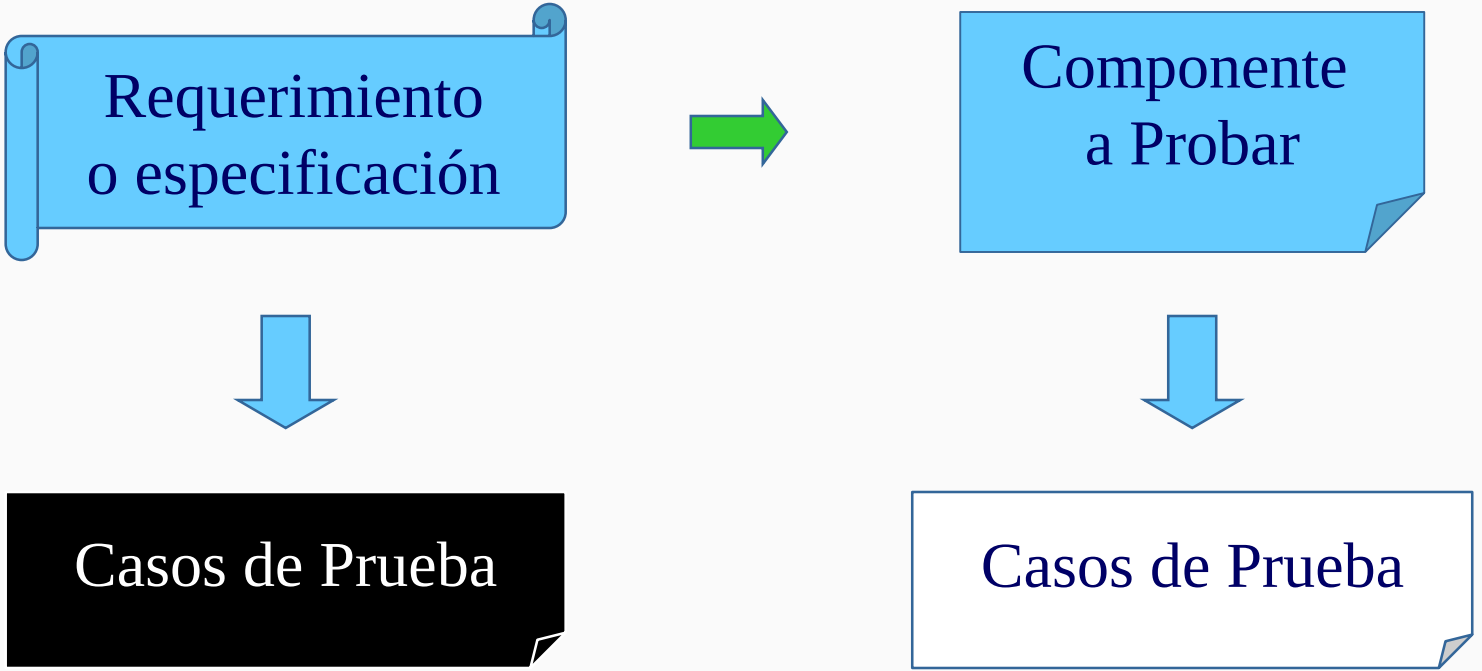
Primeras conclusiones …

- *Las pruebas no mejoran el SW, sólo muestran cuantas fallas se han producido debido a distintos tipos defectos*
- *El buen diseño y construcción no solo benefician a las pruebas, sino también a la corrección de los componentes y su mantenimiento*
- *El No probar No elimina los errores, ni acorta tiempos, ni abarata el proyecto*
- *Lo mas barato para encontrar y eliminar defectos es NO introducirlos*

PROBAR SW ES UNA ACTIVIDAD CREATIVA E INTELECTUALMENTE DESAFIANTE

Prueba Funcional

Enfoques de Prueba



Prueba de Caja Negra

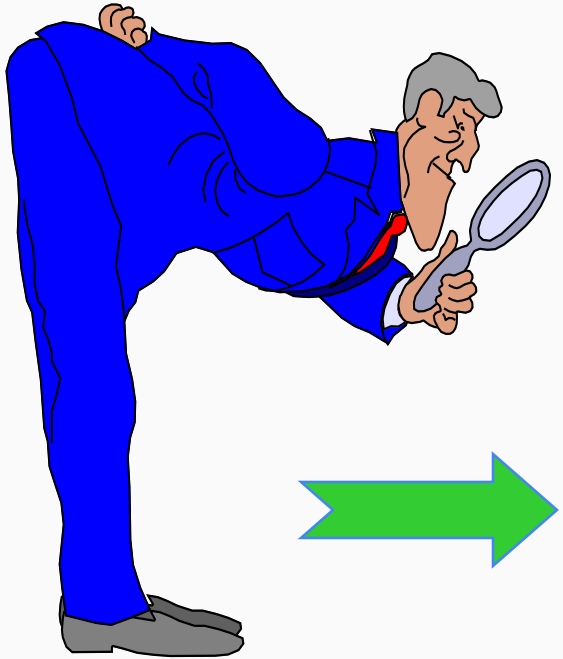
Prueba de Caja Blanca

Prueba de Caja Negra

- Prueba funcional, producida por los datos, o producida por la entrada/salida

- Prueba lo que el software debería hacer
  - *Se basa en la definición del módulo a probar (definición necesaria para construir el módulo)*
  - *Nos desentendemos completamente del comportamiento y estructura interna del componente*

Prueba de Caja Negra



?

“¿Qué condiciones & casos probarían?”

function FastExp(x,y: int): int;
{devuelve x a la y}

Prueba de Caja Negra

La prueba de caja negra exhaustiva es imposible de realizar

- *Tendría que probar todos los valores posibles de todos los datos de entrada*
- *Algo debemos hacer ...*

Prueba de Caja Negra

¿Qué hacemos?

- *Seleccionamos subconjuntos de los datos de entrada posibles, esperando que cubran un conjunto extenso de otros casos de prueba posibles*
- *Podemos suponer que la prueba de un valor representativo de cada clase es equivalente a la prueba de cualquier otro valor*
  - Llamamos a c/subconjunto “Clase de Equivalencia”
- *Estas pruebas son llamadas “Pruebas por Partición de Equivalencia” o “Pruebas basadas en subdominios”*

Prueba de Caja Negra

ALGUNOS CRITERIOS:

- Variaciones de Eventos
- Clase de Equivalencia
  - *De entrada*
  - *De salida*
- Condiciones de Borde
- Ingreso de valores de otro tipo
- Integridad del Modelo de datos
  - *De dominio*
  - *De entidad*
  - *De relación*

Prueba de Caja Negra

PARTICIÓN EN CLASE DE EQUIVALENCIA:

¿Cómo lo hacemos?

- *El proceso incluye dos pasos*
  - Identificar las clases de equivalencia
  - Definir casos de prueba

- *La identificación de clases de equivalencia se hace dividiendo cada condición de entrada en dos grupos*
  - CONDICIÓN DE ENTRADA
    - *Clases Válidas*
    - *Clases Inválidas*

Prueba de Caja Negra

PARTICIÓN EN CLASE DE EQUIVALENCIA:

- Por c/condición de entrada
  - *Rango de valores. Ej.: 100 < Nro. Sucursal < 200*
    - Una válida y dos inválidas
  - *Conjunto de valores. Ej.: DNI, CI, PAS*
    - Una válida y una inválida
  - *“Debe ser”. Ej.: Primera letra = “A”*
    - Una válida y una inválida
  - *Si creemos que los elementos de una clase de equivalencia no son tratados en forma idéntica, debemos dividir la clase en clases menores*
    - Ej.: Las suc. de Cap. Fed. son de la 100 a la 130

Prueba de Caja Negra

EJEMPLO:

Supongamos la transacción de alta de datos de un cliente (persona física) en un Banco.

- *Atributos considerados*
  - Apellido — Char (30) <> “ ”
  - Nombres — Char (30) <> “ ”
  - Documento
    - *Tipo Documento* — *Char (3) (DNI / CI / LE / LC / PAS)*
    - *Nro Documento* — *Num (9) > 0*
    - *Cod. Provincia* — *Num (2) (01 a 23) o 40*
  - Estado Civil — Char (1) (S / C / V / D / O)
  - Cantidad de Hijos — Num (2) (0 a 20)
  - Condición IVA — Char (3) (RI / RNI / EX)
  - Ingreso Mensual — Num (15) >= 0

Prueba de Caja Negra

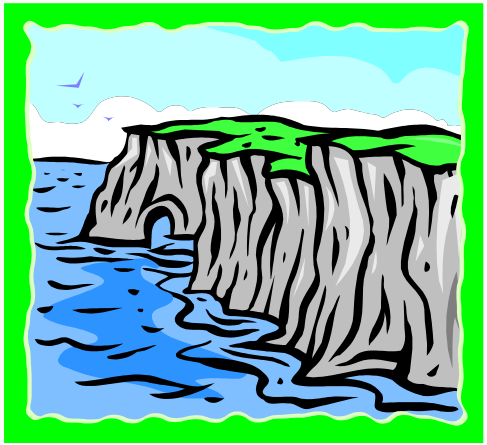
CONDICIONES DE BORDE:

La experiencia muestra que los casos de prueba que exploran las condiciones de borde producen mejor resultado que aquellas que no lo hacen

- *Además de mirar las condiciones de entrada, podemos mirar salida*
  - ej.: listados

El código postal es un número entre 1000 y 8000

998	7998
999	7999
1000	8000
1001	8001
1002	8002



Prueba de Caja Negra

CONDICIONES DE BORDE:

¿Cómo hacemos?

- *Rango de Valores*
  - Casos válidos para los extremos del rango y casos inválidos para los valores siguientes a los extremos
- *Aplicar lo mismo para los datos de salida*
- *Si la entrada o salida es un conjunto ordenado, enfocar la atención en el primero y en el último de los elementos del conjunto*
  - Prestar especial atención a los archivos/tablas vacíos, primer registro / fila, último registro / fila, fin del archivo / tabla.

Prueba de Caja Negra

CLASES INVÁLIDAS:

Hasta ahora hablamos del ingreso de clases inválidas del *mismo tipo que la clase válida*

- Pero también tenemos que probar el ingreso de valores de otro tipo
  - *Númericos en vez de alfabéticos*
  - *Alfabéticos en vez de numéricos*
  - *Combinaciones de ámbos*
  - *Fechas erróneas*

- En algunos casos, estas validaciones ya son resueltas por el entorno de desarrollo, y en consecuencia los casos de prueba no son necesarios

Prueba de Caja Negra

CLASES INVÁLIDAS:

- Muchas veces, la combinación de los datos de entrada es las que produce una clase válida o inválida
  - Ejemplo: Podría ser que si el estado civil es divorciado, los datos del cónyuge se deben ignorar
  - Ejemplo: El número de CUIT debe incluir el del documento
- Estas condiciones cruzadas deben agregarse a la lista

Prueba de Caja Negra

CONJETURA DE ERRORES:

- También llamada *Prueba de Sospechas*
- “Sospechamos” que algo puede andar mal
  - *Enumeramos una lista de errores posibles o de situaciones propensas a tener errores*
  - *Creamos casos de prueba basados en esas situaciones*
- Es un proceso muy efectivo. Formalizado a partir del análisis de las fallas.
- El programador es quien puede darnos información mas relevante

Prueba de Caja Negra

CONJETURA DE ERRORES:

- Dos orígenes
  - *Partes complejas de un componente*
  - *Circunstancias del desarrollo*

- La creatividad juega un papel clave
  - *No hay una técnica para la conjetura de errores*
  - *Es un proceso intuitivo y ad hoc*
  - *Se basa mucho en la experiencia*

Prueba de Caja Negra

CONJETURA DE ERRORES:

¿ Cuándo hacerlo ?

- *Un componente, o parte de él, hecho “a las apuradas”*
- *Un componente modificado por varias personas en distintos momentos*
- *Un componente con estructura anidadas, condiciones compuestas, etc...*
- *Un componente que sospechamos fue armado por la “técnica de copy & paste” de varios otros componentes*

Prueba de Caja Negra

DEFINIENDO CONDICIONES & CASOS:

Partamos de componentes generados por la etapa de Requerimientos. Un SDLC puede producir por ejemplo:

- *“Wish List”*
  - Statement 1
  - Statement 2
  - Statement 3
  - .....
- *Casos de Uso*
  - UC1
  - UC2
  - UC3
  - .....
- *Modelo de Datos*

Prueba de Caja Negra

TOMANDO UNA “WISH LIST”:

Partamos de componentes generados por la etapa de Requerimientos. Un SDLC puede producir por ejemplo:

- *Cada statement (declaración) es un requerimiento funcional*
  - Req 1: Debe permitir la publicación de una mesa de examen final de parte del Director del Departamento
  - Req 2: Debe permitir la inscripción de alumnos a las mesas de finales que están publicadas
- *Un requerimiento que no es testeable no es implementable*
  - Si no podemos definir como va a ser testeado tenemos un problema
  - No hay mejor manera que atacar la incompletitud de una definición de requerimientos que pensando cómo van a ser testeados
- *Pensar en “variaciones” de las declaraciones funciona muy bien.*
  - Revisar sustantivos / verbos

Prueba de Caja Negra

TOMANDO “USE CASES”:

- *La secuencia de eventos dentro de un caso de uso tienen variaciones indicadas por su texto*
- *Cada variación de un evento constituye una “condición de prueba”*
- *Cada condición debe ser ejercitada por, al menos, un caso de prueba*
- *Cada caso ejercitará uno o varios componentes involucrados*
- *El conjunto de condiciones y casos constituye la base de la prueba de aceptación funcional*
- *Este trabajo no puede hacerse si no se ha realizado el análisis de requerimientos*

Prueba de Caja Negra

TOMANDO UN “MODELO DE DATOS”:

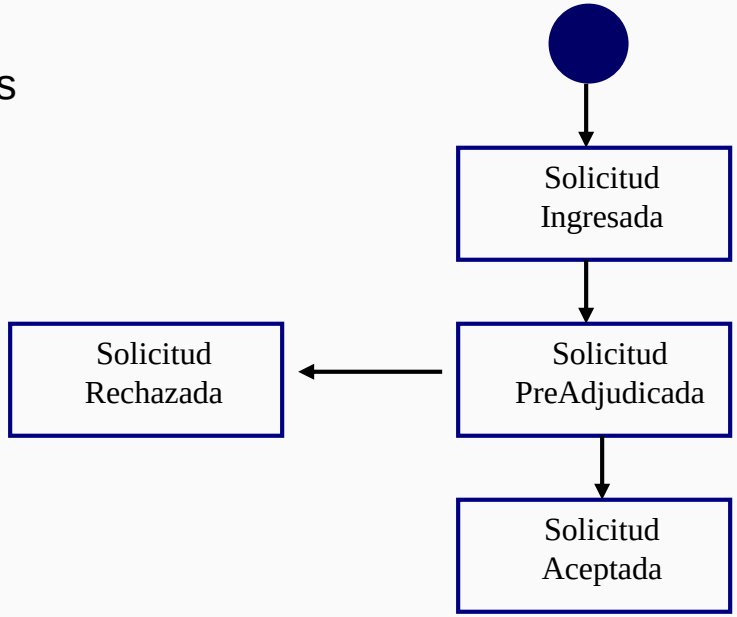
- *La integridad referencial entre tablas y la cardinalidad de las relaciones definen reglas de negocio que deben ser probadas*

- *Ejemplo:*
  - Una Mesa de Examen sin Alumnos
  - Una Mesa de Examen con un Alumno
  - Una Mesa de Examen con muchos (n) Alumnos
    - *Si el “n” estuviese acotado podría aplicar además el concepto de condiciones de borde*

Prueba de Caja Negra

TOMANDO UN “DIAGRAMA DE TRANSICIÓN DE ESTADOS”:

– *El ciclo de vida de un objeto define reglas de negocio que deben ser probadas*
  - Transiciones válidas
  - Transiciones inválidas

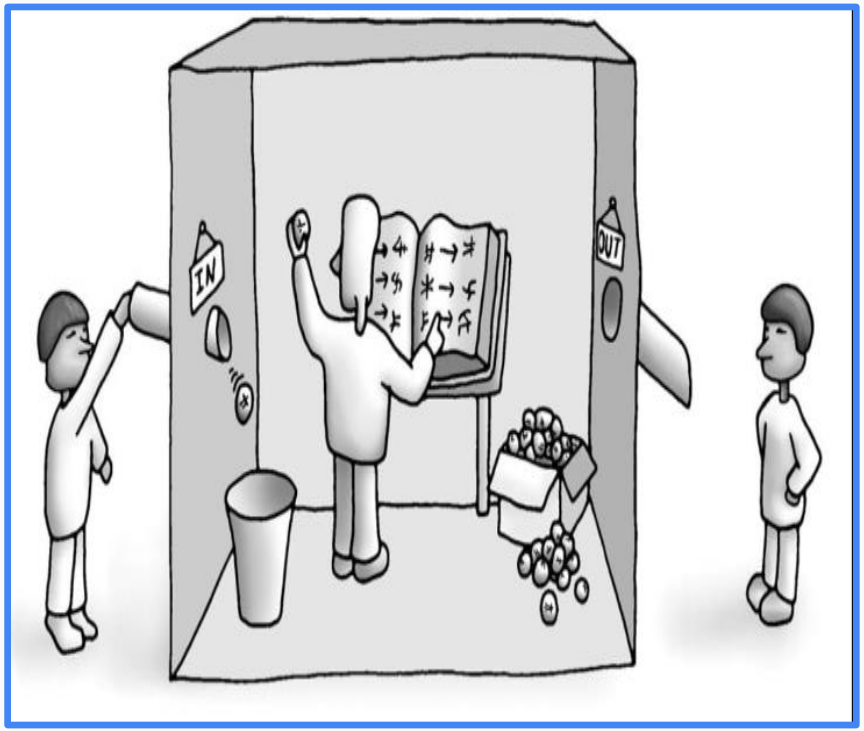


Prueba de Caja Negra

ALGUNAS CONCLUSIONES …

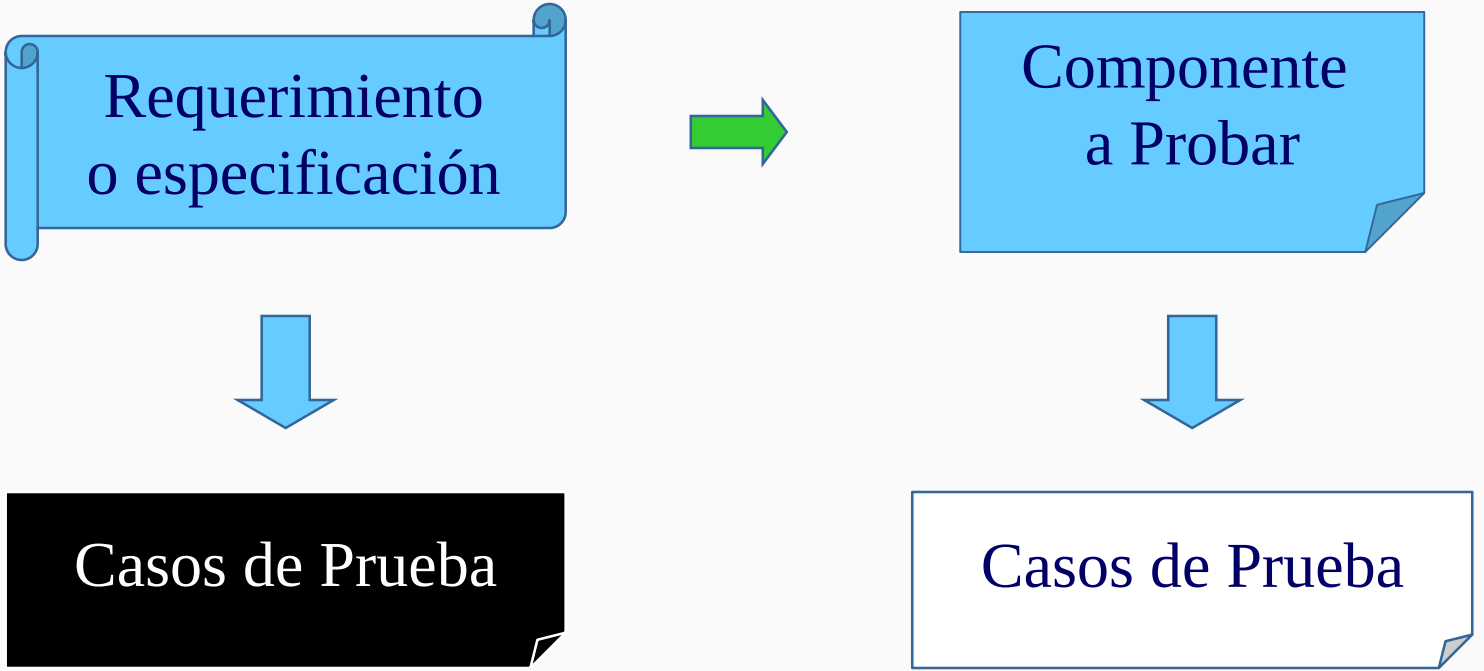
Acerca de la generación de condiciones & casos:

- *Ninguna técnica es completa*
- *Las técnicas atacan distintos problemas*
- *Lo mejor es combinar varias de estas técnicas para complementar las ventajas de c/u*
- *Sin especificaciones de reqs todo es muchísimo mas difícil*
- *Debemos tener en muy en cuenta la conjetura de errores*



Prueba Estructural

Enfoques de Prueba



Prueba de Caja Negra

Prueba de Caja Blanca

Prueba de Caja Blanca

- Prueba estructural
  - *También conocida como “clear box” o “glass box”*

- Prueba lo que el software hace

  - *Se basa en cómo está estructurado el componente internamente y su definición*
  - *Usada para incrementar el grado de cobertura de la lógica interna del componente*

Prueba de Caja Blanca

- Grados de Cobertura:
  - *Cobertura de Sentencias*
    - Prueba c/instrucción
  - *Cobertura de Decisiones*
    - Prueba c/salida de un “IF” o “WHILE”
  - *Cobertura de Condiciones*
    - Prueba cada expresión lógica (A AND B) de los IF, WHILE
  - *Prueba del Camino Básico*
    - Prueba todos los caminos independientes

Prueba de Caja Blanca

Grados de Cobertura:

- *Ejemplo Cobertura de Decisiones*
  (Cálculo Raíz Cuadrada)
  - *Ej. Caso de Clase Válida: 4*
  - *Ej. Caso de Clase Inválida: -10*

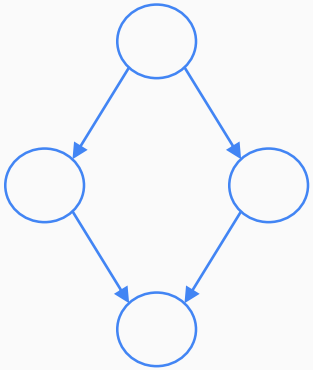
Un mismo conjunto de casos de prueba puede ofrecer distintos grados de cobertura en distintas implementaciones de una función

```
IF input <0 THEN
                CALL Print_Line "Square root error - illegal negative input"
ELSE
                USE maths co-processor to calculate the answer
                RETURN the answer
END_IF

*****************************************************************************************

IF input<0 THEN
                CALL Print_Line "Square root error - illegal negative input"
ELSE
                IF input=0 THEN
                                RETURN 0
                ELSE
                                USE maths co-processor to calculate the answer
                                RETURN the answer
                END_IF
END_IF

*****************************************************************************************

USE maths co-processor to calculate the answer
EXAMINE co-processor status registers
IF status=error THEN
                CALL Print_Line "Square root error - illegal negative input"
ELSE
                RETURN the answer
END_IF
```

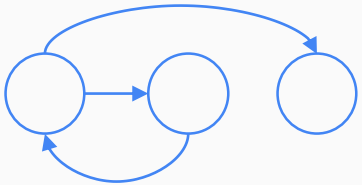
Prueba de Caja Blanca

- Camino Básico:
  - *Se representa el flujo de control de una pieza de código a través de un grafo de flujo*

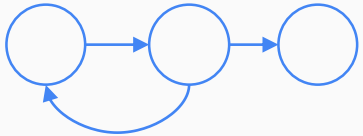
secuencia



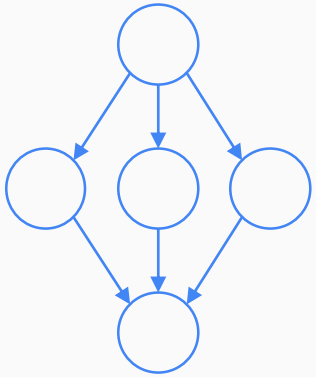
if then else



do while



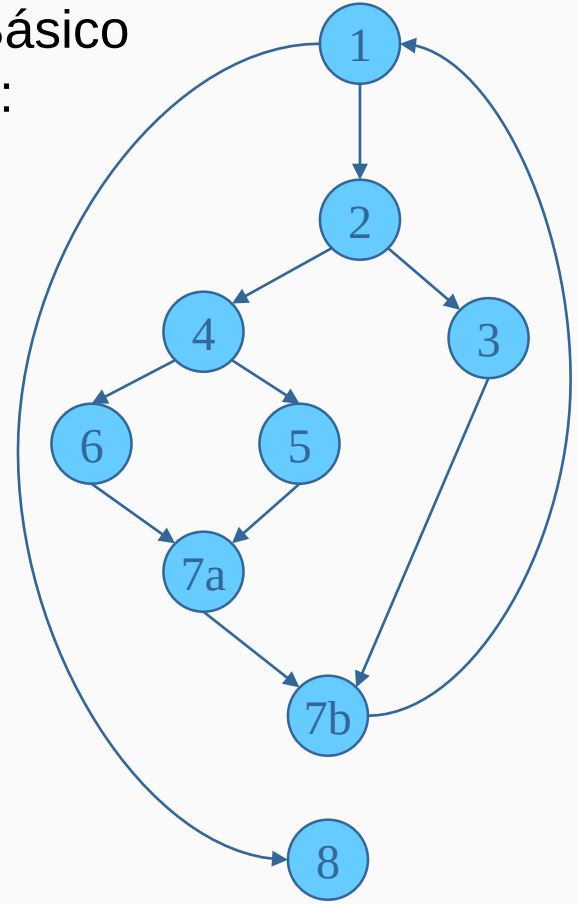
repeat until



case, o selección múltiple

Prueba de Caja Blanca

Camino Básico (Ejemplo):

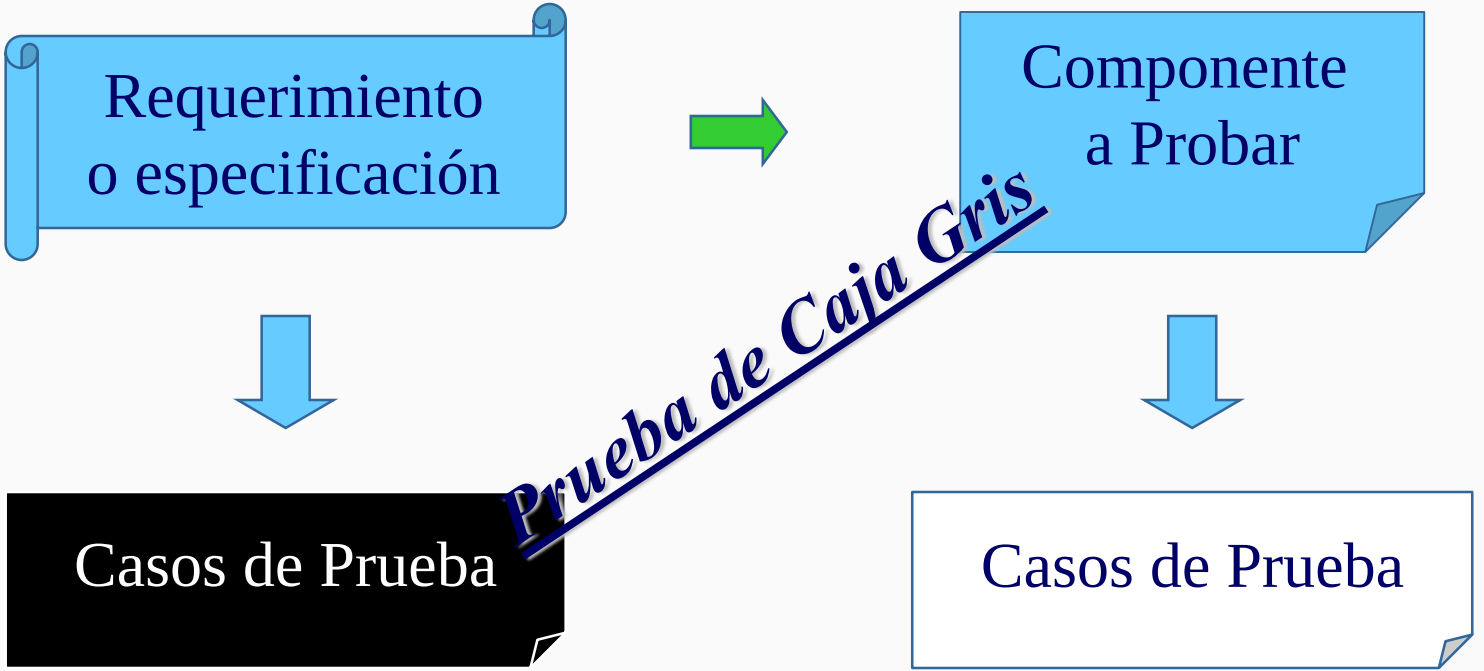



```
   procedimiento ordenar
1: do while no queden registros
     leer registro;
2:   if campo1 del registro =0
3:   then procesar registro
        guardar en buffer
        incrementar contador
4:   elsif campo2 del registro =0
5:      then reinicializar contador
6:      else procesar registro
        guardar en archivo
7a:     endif
     endif
7b:enddo
8: end
```

Prueba de Caja Blanca

- Complejidad Ciclomática:
  - *Métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa*
    - *cantidad de caminos independientes*
    - *camino independiente: agrega un nuevo conjunto de sentencias de procesamiento o una nueva condición*
  - *Formas de calcularla*
    - *número de regiones del grafo*
    - *V(g) = A - N + 2 (A = aristas, N = nodos)*

Enfoques de Prueba



Prueba de Caja Negra

Prueba de Caja Blanca

Prueba de Caja Gris

- Prueba que combina elementos de la caja negra y caja blanca
  - *No es caja negra porque se conoce parte de la implementación o estructura interna y se aprovecha ese conocimiento para generar condiciones y casos que no se generarían naturalmente en una prueba de caja negra*
  - *El conocimiento es “parcial”, no “total” (lo que sería caja blanca)*

Enfoques de Prueba - Conclusiones

- Las pruebas de **caja blanca** son un importante *complemento* a las pruebas de **caja negra**
  - *Hay defectos que serían casi imposibles detectarlos a través de caja negra*
  - *Los defectos que originan las fallas son encontrados mas rápidamente bajo caja blanca lo que deriva en una prueba mas económica*

- Las pruebas de **caja gris** prueban el SW como si fuera caja negra, pero suman condiciones y casos adicionales derivados del conocimiento de la operación e interacción de ciertos componentes de SW que componen la solución

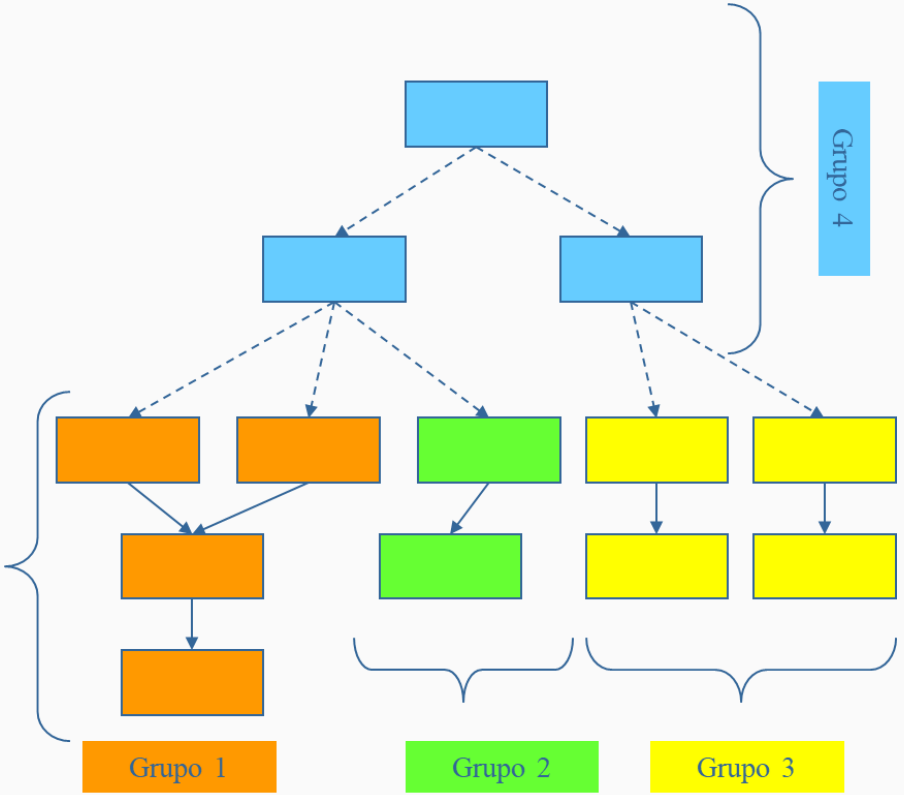
Tipos de Prueba

Prueba Unitaria

- Se realiza sobre una unidad de código claramente definida
- Generalmente lo realiza el área que construyó el módulo
- Se basa en el diseño detallado
- Comienza una vez codificado, compilado y revisado el módulo
- Los módulos altamente cohesivos son los más sencillos de probar

Prueba Integración

- Orientada a verificar que las partes de un sistema que funcionan bien aisladamente también lo hacen en su conjunto.
- TIPOS:
  - *No Incrementales*
    - BIG BANG
  - *Incrementales*
    - BOTTOM-UP
    - TOP-DOWN
    - “Sandwich”
- Los puntos clave son:
  - Conectar de a poco las partes mas complejas
  - Minimizar la necesidad de programas auxiliares



Prueba Aceptación de Usuario

- Prueba realizada por los usuarios para verificar que el sistema se ajusta a sus requerimientos

  - *Las condiciones de pruebas están basadas en el documento de requerimientos*
  - *Es una prueba de “caja negra”*

Pruebas No Funcionales

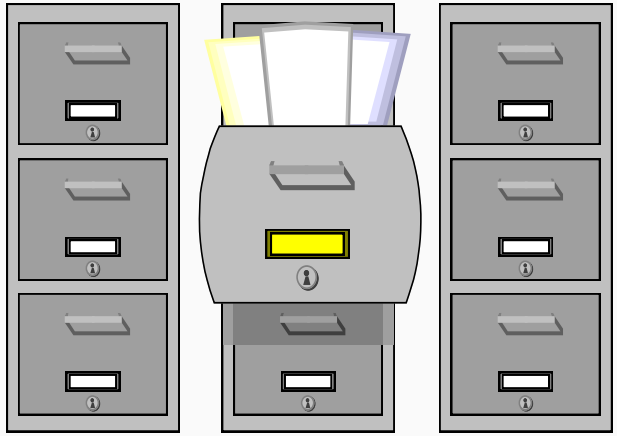
Las pruebas no funcional buscan comprobar la satisfacción de los requerimientos no funcionales del SW.

Se clasifican de acuerdo al req. no funcional bajo testeo:

- *Volumen*
- *Perfomance*
- *Stress*
- *Seguridad*
- *Usabilidad*
- *Otras pruebas:*
  - *Recuperación*
  - *Portabilidad*
  - *Escalabilidad*
  - *Etc …*

Pruebas No Funcionales

- **VOLUMEN:**
  - *Orientada a verificar a que el sistema soporta los volúmenes máximos definidos en la cuantificación de reqs.*
    - Capacidad de almacenamiento
    - Capacidad de procesamiento
    - Capacidad de transmisión

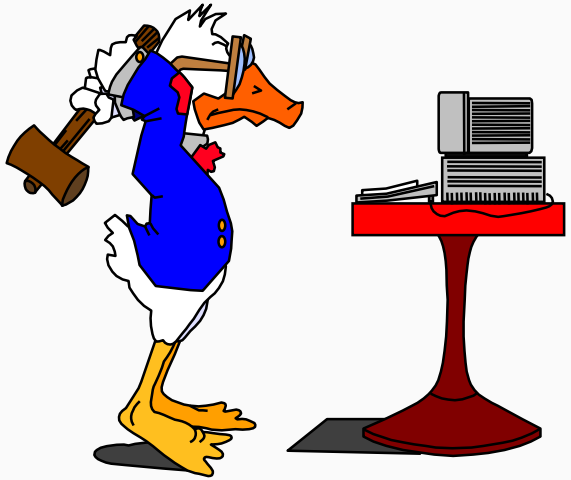


Pruebas No Funcionales

- **PERFORMANCE:**
  - *Orientada a verificar a que el sistema soporta los tiempos de respuesta definidos en la cuantificación de reqs. en las condiciones establecidas*
  - *Se evalúa la capacidad de respuesta con diferentes volúmenes de carga (x ej., demanda esperada, peaks de demanda, etc…)*
  - *Ayudan a identificar “cuellos de botella” y causas de degradación*
  - *Se realizan de la mano de las de “volumen”*

Pruebas No Funcionales

- **STRESS:**
  - *Orientada a someter al sistema excediendo los límites de su capacidad definidos en la cuantificación de reqs.*
    - Capacidad de almacenamiento
    - Capacidad de procesamiento
    - Capacidad de transmisión
  - *Se suele buscar el punto de ruptura*
  - *Busca ver el comportamiento en términos de:*
    - Estabilidad
    - Disponibilidad
    - Manejo de Errores

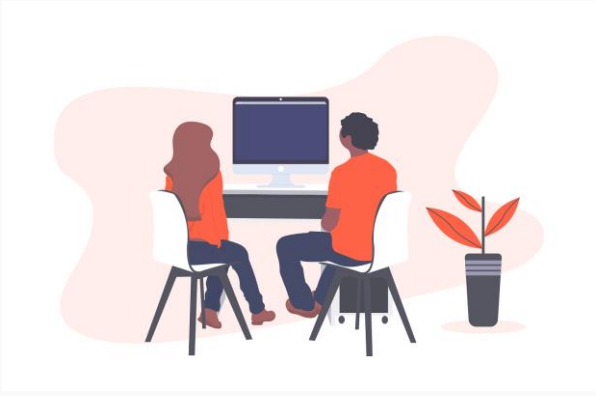


Pruebas No Funcionales

- **SEGURIDAD:**
  - *Orientada a probar los atributos/requerimientos de seguridad del sistema (si puede ser vulnerado, si el control de acceso es adecuado, etc…)*
  - *Ejemplo: Penetration Test*
    - Simula el accionar que puede realizar un intruso.
    - Persigue conocer el nivel de seguridad y exposición de los sistemas ante la posibilidad de ataques
    - Se basa en un conjunto de técnicas que permite realizar una evaluación integral de las debilidades de las aplicaciones
    - Se practica desde diferentes puntos de entrada:
      - *Internos*
      - *Externos*

Pruebas No Funcionales

- **USABILIDAD:**
  - *Orientada a probar los atributos de usabilidad definidos en los requerimientos del sistema*
  - *Las pruebas de usabilidad persiguen:*
    - ¿Cuáles son los principales problemas que evitan que el usuario complete su objetivo?
    - ¿Cómo la gente usa o usaría el producto para un fin determinado?
    - ¿Qué elementos o aspectos hacen sentir frustrado al usuario?
    - ¿Cuáles son los errores más frecuentes?
    - Se suele medir:
      - *Éxito en la tarea*
      - *Tiempo en la tarea*
      - *Errores en la tarea*
      - *Satisfacción “subjetiva”*



Pruebas de Regresión

- Orientada a verificar que, luego de introducido un cambio en el código, la funcionalidad original no ha sido alterada y se obtengan comportamientos no deseados o fallas en módulos no modificados

  - *Hay que probar “lo viejo”*
  - *Reuso de condiciones & casos*

  - *Se debería evaluar realizarlas en c/mantenimiento*
  - *La automatización es un muy buena opción*

Famosas últimas palabras...

Pruebas de Humo (Smoke Test)

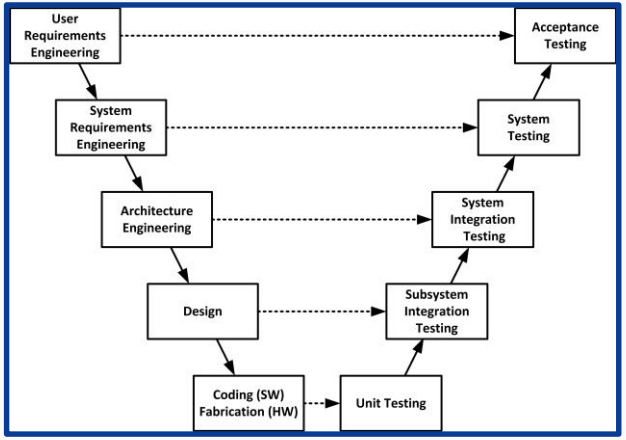
- Orientada a verificar de una manera muy rápida que en la funcionalidad del sistema no hay ninguna falla que interrumpa el funcionamiento básico del mismo

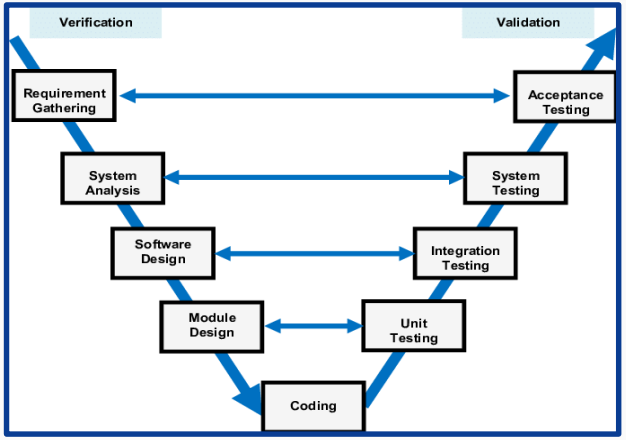
  - *El test es de muy alto nivel (solo las funcionalidades mas importantes)*
  - *No se entra “en detalles”*

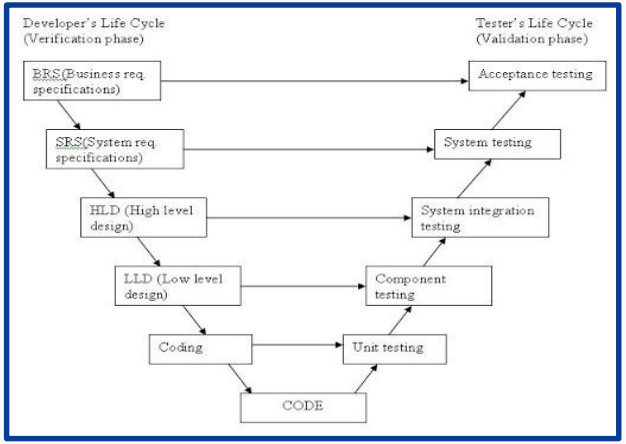
Pruebas Alfa & Beta

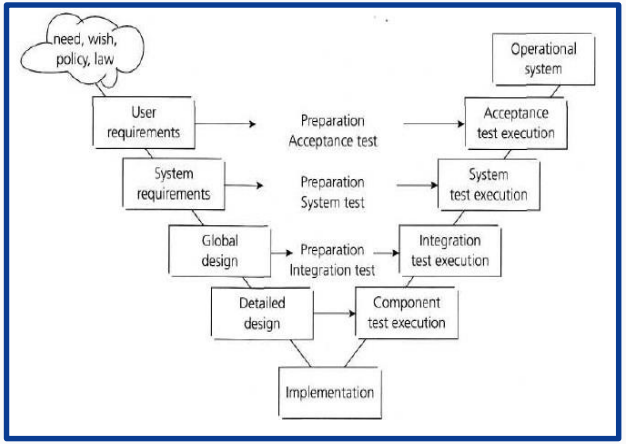
- Se entrega una primera versión al usuario que se considera está lista para ser probada por ellos
  - *Normalmente plagada de defectos*
  - *Una forma económica de identificarlos (ya que el trabajo lo hace otro)*
  - *En muchos casos no puede hacerse*
    - Alternativa válida: ejecutar en “paralelo”
  - *ALFA: la hace el usuario en mis instalaciones (en un entorno controlado y suele estar el developer)*
  - *BETA: la hace el usuario en sus instalaciones*
    - El entorno no es controlado por el desarrollador
    - El usuario es quien registra las fallas y reporta regularmente

El “V-Model”


User Requirements Engineering
Acceptance Testing
System Requirements Engineering
System Testing
Architecture Engineering
System Integration Testing
Design
Subsystem Integration Testing
Coding (SW) Fabrication (HW)
Unit Testing


Verification
Validation
Requirement Gathering
Acceptance Testing
System Analysis
System Testing
Software Design
Integration Testing
Module Design
Unit Testing
Coding


Developer's Life Cycle (Verification phase)
Tester's Life Cycle (Validation phase)
BRS(Business req. specifications)
Acceptance testing
SRS(System req. specifications)
System testing
HLD (High level design)
System integration testing
LLD (Low level design)
Component testing
Coding
Unit testing
CODE


need, wish, policy, law
Operational system
User requirements
Preparation Acceptance test
Acceptance test execution
System requirements
Preparation System test
System test execution
Global design
Preparation Integration test
Integration test execution
Detailed design
Component test execution
Implementation

Exploratory Testing

Concepto

- Exploratory Testing
  - *También conocido como “AdHoc Testing”*
  - *Es el aprendizaje, diseño y ejecución de la prueba en forma simultánea*
  - *Es “unscripted”*
    - Todas las condiciones y casos no son diseñados y documentados previamente para ser ejecutados
    - El tester va decidiendo tácticamente qué es lo mejor en c/paso de acuerdo al conocimiento que va adquiriendo de la ejecución de un test
  - *El tester es responsable en todo momento de la decisión del camino a tomar*
  - *“Unscripted” no significa “sin preparación”*
    - Apunta a tener mas variantes y no tener restricciones del “script driven”

Script Driven Testing & Unscripted Testing

- Script driven
  - *Se confeccionan las condiciones & casos tempranamente para luego ser ejecutados*
    - Por lo general, la actividad de creación de condiciones está destinada a profesionales con mas skill
    - La ejecución y reporte está asignada a personas con menor skill (mas operativas)
  - *Muchas veces, en cada ciclo de prueba se vuelven a repetir las condiciones y casos y no se retroalimentan*
    - C/ciclo “mira lo mismo”

- Unscripted
  - *Se confeccionan las condiciones & casos a medida que se aprende de las distintas ejecuciones, refocalizando las pruebas si fuera necesario (obteniendo ventajas de lo aprendido)*
  - *Las nuevas ideas ocurren “on the fly”*

Script Driven Testing

- Ventajas
  - *Puede ser objeto de revisión entre pares*
    - Otros Testers
    - Usuarios finales
  - *Puede ser reusado para una nueva ejecución fácilmente*
  - *Puede ser medible*
    - # de condiciones
    - # de casos

- Desventajas
  - *Laborioso*
  - *La etapa mas creativa se da en el diseño de las condiciones y casos y no en la etapa de ejecución*

Unscripted Testing

- Ventajas
  - *Se pueden variar los test sobre la marcha de acuerdo a lo que se considere mas apropiado*
    - El trabajo creativo se hace durante la ejecución (no antes)
  - *Permite mayor cobertura de situaciones sobre posibilidades difíciles de anticipar*

- Desventajas
  - *Puede perderse la capacidad de "reproducir" sino se sigue un órden (plan o charter) a la hora de probar*
    - El "plan" no significa "script"
  - *Muy dependiente de las personas*
    - Creatividad / Memoria / Intuición
    - Algunos pueden ser expertos en el dominio del negocio ("subject matter experts"), otros en la plataforma tecnológica que soporta la solución, otros pueden conocer el producto con anterioridad, etc …
    - ¿Está mal?

El Testing & su Contexto

- Hay *numerosos* y *variados* aspectos en c/proyecto que influyen a la hora de testear:
  - *Calendario / Presupuesto*
  - *Skill Testers*
  - *Herramientas / Ambientes*
  - *El tipo de producto a testear*
    - Conocimiento previo del dominio
    - Conocimiento previo del producto
  - *Objetivos de “calidad” (ponderación del cliente)*
  - *Etc … … …*

- Cada una de las consideraciones influyen a la hora de seleccionar cómo testear

Exploratory Testing

- Cuándo es apropiado el ET?

  - *Se necesita conocer el producto rápidamente*
  - *Se demanda feedback en poco tiempo*
  - *Se ejecutó "scripted testing" y se quiere diversificar la prueba*
  - *Se pretende chequear el testing realizado por un tercero a través de una breve prueba independiente*
  - *Hay que atacar un riesgo en particular*
  - *Hay que buscar un bug puntual previamente reportado*

Exploratory Testing

- Cuatro etapas básicas:
  - *Reconocimiento & Aprendizaje:*
    - Identificar toda la información que nos permita conocer qué es lo mas importante a probar y cómo hacerlo
    - Depende de la habilidad del tester para identificar los riesgos tanto de la aplicación como de la plataforma
  - *Diseño*
    - Crear una guía draft para probar
  - *Ejecución*
    - Ejecutar los casos y registrar los resultados
  - *Interpretación*
    - Obtener conclusiones de lo probado.
      - *Tanto a nivel del conocimiento adquirido del producto como de la forma en qué estamos probando*

Exploratory Testing

- ¿Cómo empezar? Hay distintos enfoques:
  - *Haciendo un draft de la arquitectura*
    - Distinto tipo de modelos
  - *Brainstorming the tipos de condiciones/casos a ejecutar*
  - *Imaginando todas las posibles formas de "fallar" que puede experimentar la aplicación*
  - *Haciendo preguntas "context-free"*
    - Quién? / Cuándo? / Qué? / Por qué? / Cómo / Dónde?
  - *Revisando especificaciones / manuales de usuario*
  - *Uso de heurísticas*
  - *Etc … … …*

Exploratory Testing

- Armado del “charter”:
  - *Un charter define la “misión” de la sesión de testing*
    - Qué se debería testear
    - Cómo se debería testear
    - Qué tipo de defectos buscar
  - *Un charter NO es un plan detallado*
  - *Ejemplo*
    - Opción “high level”: Analizar la función “Insertar Gráfico”
    - Opción “detailed level”: Ver el comportamiento al insertar varios tipos de gráficos en distintos docs. Poner foco en el consumo de recursos y el tiempo de respuesta ……..

Conclusión

- ET es una forma de encarar el testing
  - *Muchos testers de alguna manera en ciertos momentos han aplicado y aplican ET*
- ET es dependiente del tester
  - *Skill / Experiencia / "Personalidad"*
- ET es dependiente del conocimiento que se va obteniendo a medida que se ejecuta la prueba
- No se trata de "script based testing" vs "unscripted based testing"
  - *Ver lo conveniencia en c/situación*
  - *Buscar alternativas "mix"*

Métricas de Testing

Testing Metrics

Hay muchas métricas a utilizar en el testing de un proyecto:

- *Productividad Diseño*
  - Ej.: Condiciones Construídas x Unid. Tiempo
- *Productividad Ejecución*
  - Ej.: Casos Ejecutados x Unid. Tiempo
- *Eficacia (Pre-Release):*
  - Ej.: (Incidentes Reportados Aceptados / Total Incidentes Reportados x 100)
  - Es interesante medir también la eficacia en la resolución de defectos
- *Eficacia (Post-Release):*
  - Ej.: [Fallas Reportadas / (Fallas Reportados + Fallas Rep. User]x 100
- *“Calidad” del Desarrollo:*
  - Ej.: Indice de Severidad por Defectos Reportados
- *Release Readiness:*
  - Ej.: Indice de Severidad por Defectos Abiertos

Testing Automation

Primeras reflexiones

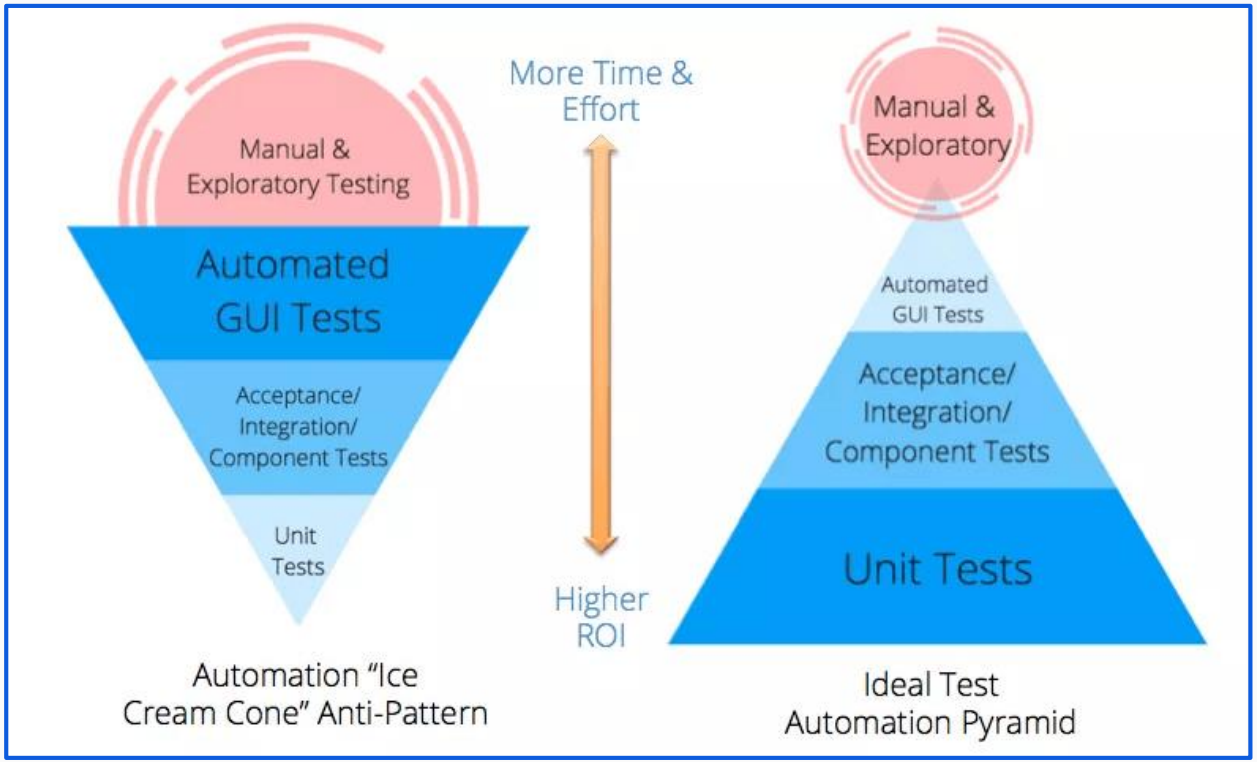
- Las pruebas automáticas tienen que ser un complemento de las manuales
  - No perseguir eliminar al testing manual
  - No buscar suplantar a los testers
  - La automatización no debe ser una meta, sino una alternativa
- Las automatizaciones abarcan un amplio espectro del proceso de testing
  - Desde la gestión de incidentes a las pruebas de regresión pasando por la generación de casos de prueba
- Automatizar -como cualquier proceso de desarrollo de software- lleva tiempo y esfuerzo; y un mantenimiento a medida que nuestro producto cambie
- Hay ciertos tipos de testing que por su magnitud/complejidad son excesivamente costosos (… hasta casi imposibles de realizar …) sin la ayuda de herramientas (Inviabilidad Técnica!)

Clasificación Herramientas Testing *

- **Herramientas para gestionar las pruebas y los test cases**
  - *Gestión de pruebas*
  - *Gestión de requerimientos*
  - *Gestión de incidentes*
  - *Gestión de configuración*
- ***Herramientas para testing estático***
  - *Revisión*
  - *Análisis estático*
  - *Modelado*
- **Herramientas de soporte para la especificación de pruebas**
  - *Diseño de pruebas*
  - *Preparación de datos de prueba*
- **Herramientas para soporte de ejecución y registración**
  - *Ejecución de pruebas*
  - *Pruebas Unitarias*
  - *Comparadores de pruebas*
  - *Medición de cobertura*
  - *Pruebas de seguridad*
- **Herramientas de soporte de Performance y monitoreo**
  - *Análisis dinámico*
  - *Pruebas de Performance / Carga / Stress*
  - *Monitoreo*
- **Herramientas para soporte a fines específicos**
  - *Evaluación de calidad de datos*
  - *Pruebas de Usabilidad*

* International Software Testing Qualification Board

¿Qué automatizar?



Pirámide de Cohn

Ventajas & Desventajas

- Ventajas
  - *Cobertura*
  - *Bajo costo de ejecución*
  - *Multiplicabilidad*
  - *Independencia del Tester*
  - *Consistencia (… si falla, falla siempre!)*
  - *Reuso*
  - *Rapidez*

- Desventajas
  - *En algunos casos, el costo de "licenciamiento" / "alquiler" es elevado*
  - *Costo de desarrollo & mantenimiento*
    - En particular el costo de Test GUI
    - Skill de developers
  - *Las fallas pueden deberse a la automatización*

Antes de arrancar …

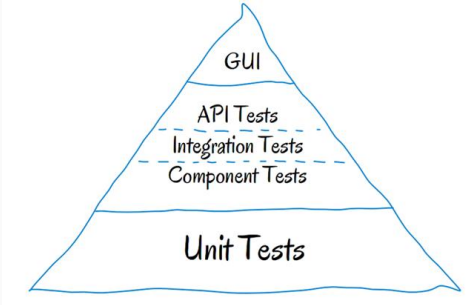
- Tener muy claro la madurez de mi “testing manual”
  - *¿Cuánto tiempo llevo testeando?*
  - *¿Tengo un proceso definido?*
  - *¿Funciona? ¿Estoy conforme? ¿Lo reviso y lo mejoro?*
  - *¿Poseo historia? ¿Guardé métricas?*
  - *Q de ciclos / Esfuerzo incurrido / Ventana de tiempo / Costos*
  - *¿Capitalicé las “lessons learned”?*
  - *Resguardo mis activos?*
    - Conocimiento
    - Condiciones & Casos de Prueba
    - Evidencias
    - Incidentes

Objetivo

- Lo primero a tener claro:
  - *Identificar la motivación*
  - *Expectativas*
  - *¿Cuál es mi driver?*
    - Umbral de Calidad – Not “Good Enough”
      - Mas profundidad / Mas cobertura / Mas repeticiones / Troubleshooting
    - *Costo*
    - *Ventana de tiempo*
    - *Equipo*
    - *¿Son varios?*
  - *¿Tengo restricciones?*
  - *¿Tengo flexibilidad?*

La Aplicación

- *Criticidad del producto / aplicación*
  - Relevancia para el negocio
- *¿Producto maduro/estable?*
- *“Volatilidad” de cambios*
  - ¿Qué parte de la aplicación es la que más cambia?
- *Rol de la interfaz de usuario*
- *Frecuencia del testing de regresión*
- *Importancia de los Reqs No Funcionales*
- *Desarrollo In House / Third party Dev / COTS*
  - ¿Tecnología propietaria?
- *Plataformas: OS / DB / Browsers / Devices / Integración / Cloud / …*
  - ¿Una o varias?
- *Frecuencia de upgrades de las partes*



El Caso de Negocio

La presión por el ROI
- Lo usual: Ahorro en esfuerzo = Ahorro en dinero … pero NO es todo

De la Prueba Manual:
- Costo del Diseño/Construcción de Condiciones & Casos de prueba
- Costo de un Ciclo de Prueba

De la Prueba Automática:
- Licencias Herramienta + HW + SW de Base + Training + Mantenimiento
- Costo de Diseño/Construcción de Scripts
- Costo de prueba de los scripts / Change Mgmt & Versionado / BackUp / Etc …..
- Costo de un Ciclo de Prueba

Por costo tener en cuenta esfuerzo y $$$

La automatización como soporte y liberar recursos (NO reemplazar)
- Nueva funcionalidad (crecimiento) = Mayor ejecución

Visión del "Costo/Beneficio"

Conclusiones

- *Acumular experiencia en testing manual*
- *Setear expectativas*
- *No esperar resultados de manera inmediata*
- *No pretender automatizar todo!!*
  - Automatizar lo que conviene!
  - Además, hay cosas que necesitan verificación visual
- *No perder vista que las condiciones y casos son el activo*
  - La automatización por si sola no encuentra fallas
- *Independizarse lo más posible de una herramienta en particular*

Conclusiones

- *Separar el equipo de automatización del equipo de ejecución*
  - Contar con los perfiles adecuados para automatizar
- *Analizar proyecto a proyecto (c/situación es particular)*
- *Esperar cualquier evaluación de automatizar a contar con un producto/aplicación con cierta estabilidad*
  - Cuidado con la volatilidad de la interfaz de usuario
- *Automatizar es un proyecto mas y su entregable es una aplicación mas*
  - Y como toda aplicación queremos que sea mantenible (… y demás atributos de calidad …)
  - Mas todo lo que ya conocemos de cualquier desarrollo …
- *Automatizar debe seguir un proceso → Medir & Evaluar*

Preguntas