

UNIDAD 4

Software Testing

Controlando la calidad del producto construido

v1.1 - REV 2019_08_12



Conceptos Generales de Software Testing

¿Qué es el testing de Software?

- Según la 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
- Software Testing es usualmente definida como una actividad para evaluar si los resultados obtenidos en la ejecución del software coinciden los los resultados esperados, con el propósito de determinar si el sistema está libre de defectos Involucra la ejecución de un componente de software o de un sistema para evaluar una o varias propiedades de interés. ¿

¿Qué es el testing de Software?

- Otra definición:

Es una investigación técnica empírica ejecutada para proveer a los stakeholders con información de la calidad del producto o servicio que se encuentra a prueba.

- *Professor Cem Kaner* - Director of Florida Tech's Center for Software Testing Education & Research
- Las pruebas por sí solas encuentran fallas, NO defectos (a lo cual solo pueden demostrar su presencia pero nunca su ausencia)
- Partimos de la suposición que el SW a probar contiene defectos (lo cual casi siempre es válido) y se trata de encontrar tantos como sea posible
- Por el contrario, si nuestro objetivo es mostrar que el SW no contiene errores, estaremos inconscientemente orientados a ese fin
- El adoptar una u otra actitud tiene profundas implicancias en el éxito de la prueba

¿Qué es el testing de Software?

- Las pruebas por sí solas encuentran fallas, NO defectos (a lo cual solo pueden demostrar su presencia pero nunca su ausencia)
- Partimos de la suposición que el SW a probar contiene defectos (lo cual casi siempre es válido) y se trata de encontrar tantos como sea posible
- Por el contrario, si nuestro objetivo es mostrar que el SW no contiene errores, estaremos inconscientemente orientados a ese fin
- El adoptar una u otra actitud tiene profundas implicancias en el éxito de la prueba

¿Por qué es importante el Testing?

El Testing es importante porque los bugs pueden ser costosos o incluso peligrosos. Los bugs de software pueden causar potencialmente problemas económicos o de pérdidas humanas.

- Cada vez existe software más crítico y más complejo
- El Software está escrito por personas, y las personas cometen errores
 - Por lo tanto, el software puede cometer defectos que llevarán potencialmente a fallas
- Marc Andreessen: Why Software is eating the World (<http://a16z.com/2016/08/20/why-software-is-eating-the-world/>)

¿Por qué es importante el Testing?

Ejemplos de casos emblemáticos de fallas en un Software:

- En Abril 2015, una terminal de Bloomberg en Londres colapso debido a una falla de software que afectó a más de 300.000 inversionistas en mercados financieros. El hecho forzó al gobierno ingles a posponer un pago de 3BN de libras esterlinas de un bono.
<https://www.theguardian.com/business/2015/apr/17/uk-halts-bond-sale-bloomberg-terminals-crash-worldwide>
- Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they unable to process the transaction.
- En mayo de 1996, un bug de software causó que las cuentas bancarias de 823 clientes de un banco estadounidense sean acreditadas con 920 Millones de Dólares.
- En un Bingo de Argentina, una persona “gano” 35 Millones de pesos por un error de software de la máquina tragamonedas.
- Knight Capital: En agosto de 2012, un error de programa casi provocó la quiebra de la [empresa](#) de inversión Knight Capital. La compañía perdió 500 millones de dólares en media hora debido a que sus computadoras comenzaron a comprar y vender millones de acciones sin ningún tipo de control humano. Como resultado, el precio de las acciones de Knight Capital cayó un 75% en dos días.

¿Qué es la crisis del Software?

Este término fue acuñado ya en los años 70, cuando la industria del software ya había producido los suficientes programas para darse cuenta de que había algo que fallaba. En concreto estas eran sus principales inquietudes:

- ¿Por qué lleva tanto tiempo terminar los programas?
- ¿Por qué es tan elevado el coste?
- ¿Por qué no podemos encontrar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué es tan difícil constatar el progreso durante el desarrollo?
- ¿Por qué es tan difícil calcular cuánto tiempo va a costar?

Esto pasaba entonces y pasa ahora, y por eso se dice que el software está en crisis. Ante dicha situación se planteó el aplicar métodos científicos y rigurosos al proceso de desarrollo de programas, apareciendo en escena [la Ingeniería del Software](#) y la Ingeniería de Requerimientos

Aunque quizás:

- •“El software no está en crisis. La crisis le viene desde que nació. Lo que hay que plantearse es por qué no ha salido de esa crisis en todo este tiempo.”

¿Cual es el objetivo del Testing?

- Básicamente, encontrar fallas en el producto
- De forma eficiente
 - Hacerlo lo más rápido posible
 - Hacerlo lo más barato posible
- De forma eficaz
 - Encontrar la mayor cantidad de fallas
 - No detectar fallas que en realidad no son
 - Encontrar las más importantes

¿Por qué se producen fallas?

- Requerimientos poco claros
- Requerimientos incorrectos
- Requerimientos cambiantes
- Diseño Complejo
- Desarrolladores sin experiencia
- Lógica Complicada
- Falta de tiempo
- Falta de conocimiento del producto
- Testers sin experiencia
- Problemas del entorno o de sistemas



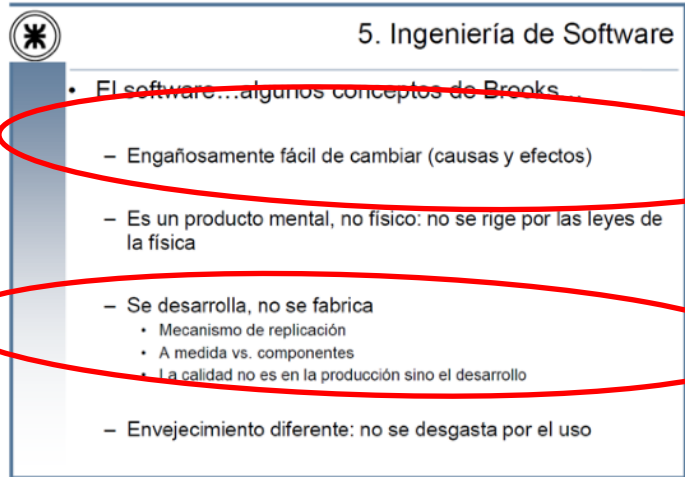
¿Qué es Software de Calidad?

- Aquel que cumpla con los requisitos
- El que al ser usado NO tenga fallas



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



5. Ingeniería de Software

- El software... algunos conceptos de Brooks ...
 - Engañosamente fácil de cambiar (causas y efectos)
 - Es un producto mental, no físico: no se rige por las leyes de la física
 - Se desarrolla, no se fabrica
 - Mecanismo de replicación
 - A medida vs. componentes
 - La calidad no es en la producción sino el desarrollo
 - Envejecimiento diferente: no se desgasta por el uso

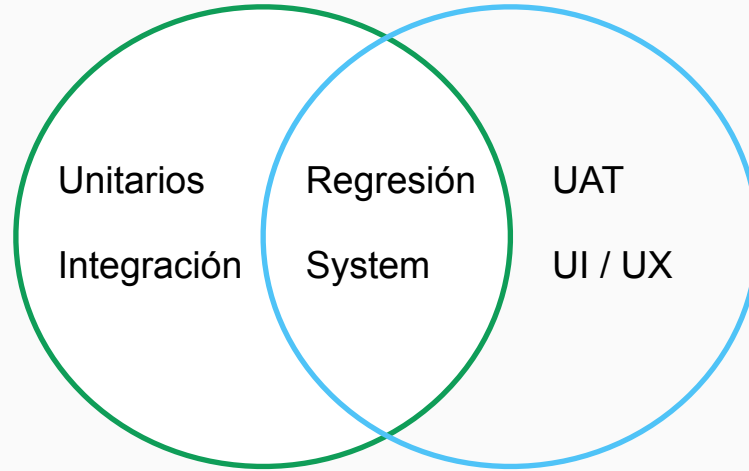
Asegurar Vs Controlar la Calidad

- 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
- SQA vs Testing

Verificación y Validación

Verificación

¿Estoy
construyendo el
producto
correctamente?



Validación

¿Estoy
construyendo el
producto correcto?



El Proceso de Software Testing

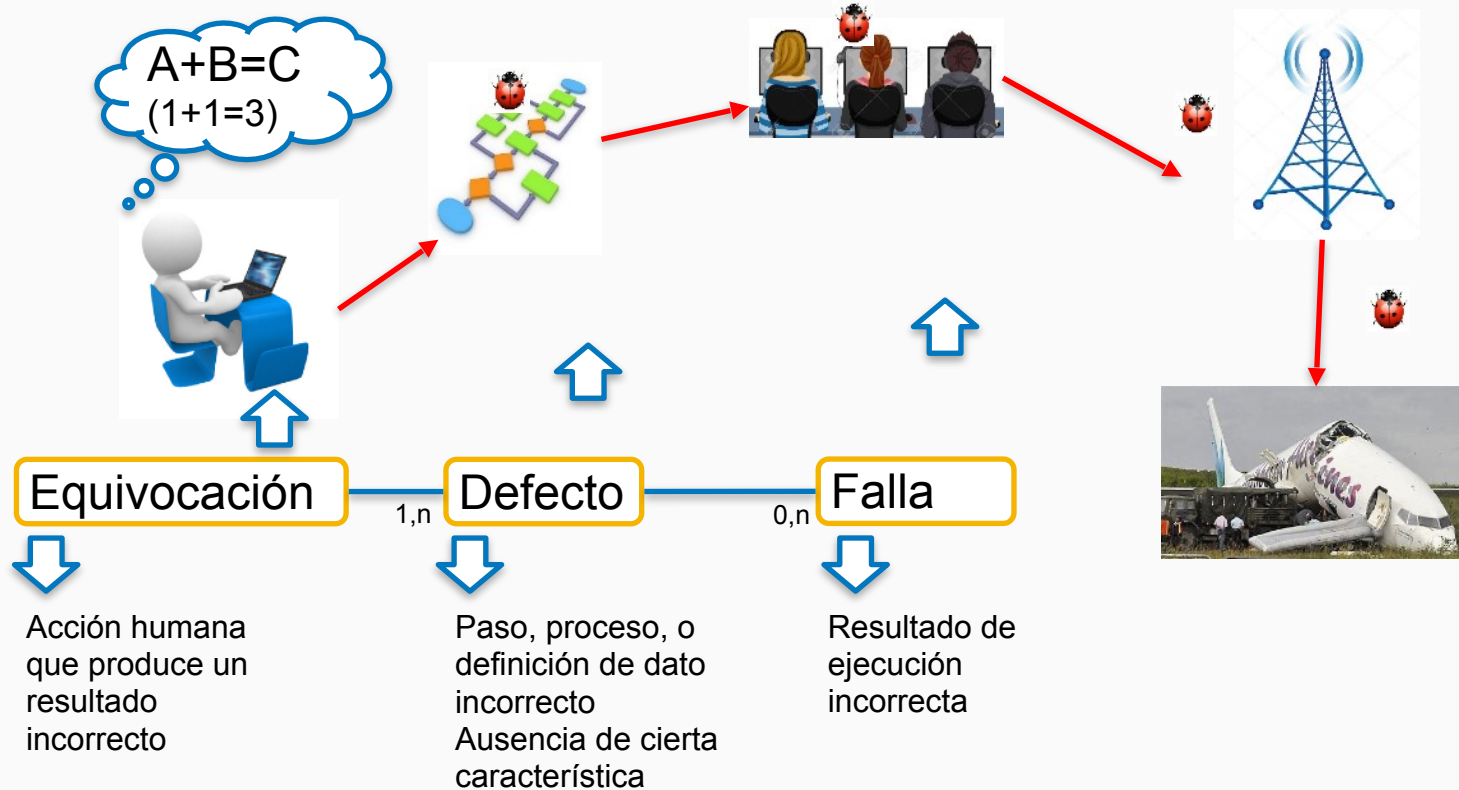
Informalmente hablando del proceso de Testing...

- Probar es ejecutar un componente con el objetivo de producir fallas
- Una prueba es exitosa si encuentra fallas

El proceso de Testing



Conceptos relacionados al proceso



Incidentes en Testing

Según IEEE, un incidente es:

- Toda ocurrencia de un evento que ocurre durante la ejecución de una prueba de software que requiere investigación
 - Un incidente puede deberse a:
 - Un defecto en el SW
 - Un defecto en los casos de prueba
 - Un defecto en el ambiente
 - entre otros eventos...
- Entonces no todo incidente es una falla, por ejemplo
 - Defectos en los casos
 - Equivocaciones al ejecutar las pruebas
 - Interpretaciones erróneas
 - Dudas

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 en probar
- Probar es el proceso de establecer confianza en que un programa hace lo que se supone que tiene que hacer
- Ya que nunca voy a poder demostrar que un programa es correcto, continuar probando es una decisión económica

¿Cuándo parar de probar?

- Nuestro Problema:
 - Imposibilidad en la práctica de estimar la intensidad del testing para alcanzar determinada densidad de defectos
- Estrategias:
 - Pasa exitosamente el conjunto de pruebas diseñado
 - “Good Enough”: cierta cantidad de fallas no críticas es aceptable (umbral de fallas no críticas por unidad de testing)
 - Cantidad de fallas detectadas es similar a la cantidad de fallas estimadas

Abaratamiento del Testing

- 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



Enfoques de Prueba

Enfoque 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
- **¿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”

Condiciones y Casos

Condiciones de Prueba

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

Casos de Prueba

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

Criterio de Selección de Casos

- Es una condición para seleccionar un conjunto de casos de prueba
- De todas las combinaciones posibles sólo seleccionaremos algunas:
 - La menor cantidad de aquellas que tengan mayor probabilidad de encontrar un defecto no encontrado por otra prueba

Partición de Casos

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

Partición de Casos

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

Enfoque de Caja Negra - Criterios de Selección

- 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

Caja Negra - Criterios de Selección de Casos

- 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

Criterios de Selección - Partición 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*
 - Ejemplo de elección Por c/condición de entrada
 - *Rango de valores. Ej.: $100 < \text{Nro. Sucursal} < 200$*
 - 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

Partición de equivalencia - Ejemplo

- ***Supongamos la transacción de alta de datos de un cliente (persona física) en un Banco.***
- ***Definir las particiones para los atributos seleccionados***
 - **Trabajaremos solo con algunos al solo efecto de incorporar el concepto de partición**
 - *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
 - ***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

Criterios de Selección - Condición 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
 - *Si creemos que los elementos de una clase de equivalencia no son tratados en forma idéntica, debemos dividir la clase en clases menores*
 - Ejemplos: Listados

El código postal es
un número entre
1000 y 8000.

998	7998
999	7999
1000	8000
1001	8001
1002	8002



Criterios de Selección - Condición 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.

Criterios de Selección - 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 ambos*
 - *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
- Muchas veces, la combinación de los datos de entrada es la 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

Criterios de Selección - 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 más relevante
- Tiene generalmente 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

Criterios de Selección - 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

Enfoque de Caja Blanca

- Prueba estructural del Software
 - *También conocida como “Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing”*
 - Es un método de Software Testing en el cual la estructura interna, el diseño y la implementación del software es conocido por el Tester.
 - The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.
- El Tester elige qué valores de entrada utilizar para ejecutar ciertos caminos en el código, conociendo la potencial salida de ese camino.
- En este tipo de pruebas, es esencial que el Tester conozca el lenguaje de programación con el que el Software fue construido, debido a que se basa en cómo está estructurado el componente internamente y su definición
- Se lo conoce como caja blanca porque desde el punto de vista del Tester, se puede ver como el Software está construido.
- Se lo utiliza también para incrementar el grado de cobertura de la lógica interna del componente

Enfoque 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

Caja Blanca - Cobertura de Decisiones

- 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
```

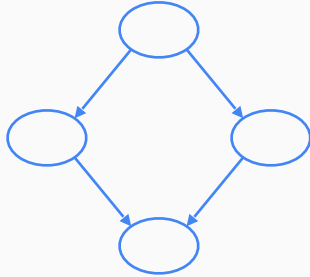
```
Use maths co-processor to calculate the answer
Examine co-processor status registers
If status=error THEN
    CALL Print_Line "Square root error - illegal negative
input"
ELSE
    RETURN the answer
END_IF
```

Caja Blanca - Camino Básico

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



secuencia



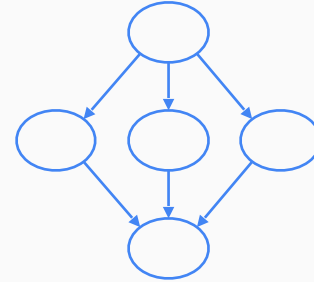
if then
else



do
while

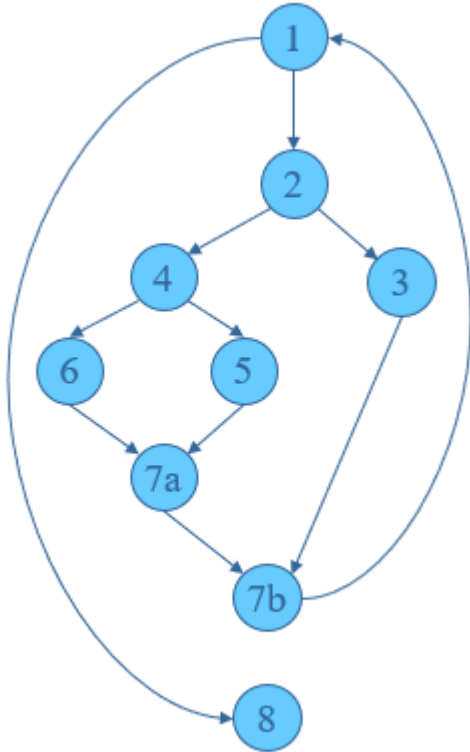


repeat
until



case, o
selección
múltiple

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
```

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)

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 requerimientos todo es muchísimo más difícil
- Debemos tener muy en cuenta la conjetura de errores



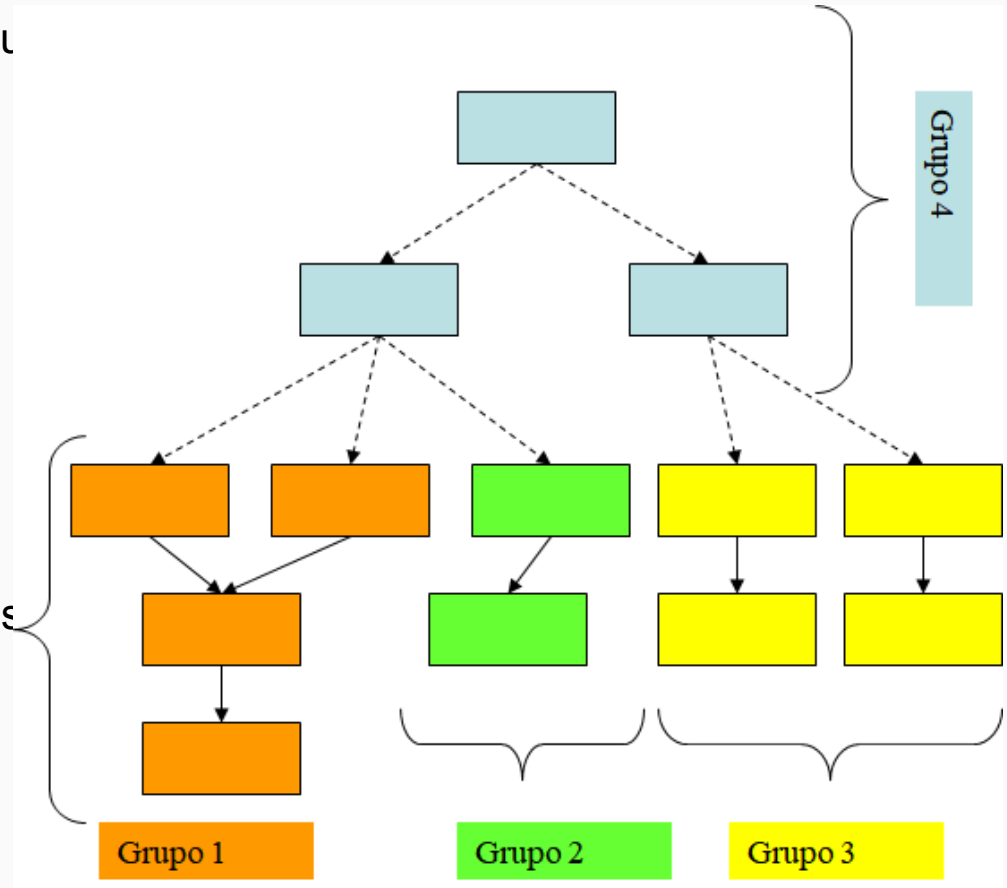
Tipos de Prueba

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 de 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 de 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”



Prueba de Stress

- Orientada a someter al sistema excediendo los límites de su capacidad de procesamiento y almacenamiento.
- Teniendo en cuenta situaciones NO previstas originalmente

Pen Testing / Ethical Hacking

- Es la práctica de testear un software, red o aplicación web con el fin de encontrar vulnerabilidades de seguridad que un hacker pudiera explotar.
- El test en sí puede automatizarse utilizando aplicaciones software o ser realizado en forma manual. De cualquier manera, el proceso involucra obtener información acerca del software o servicio a probar, identificar potenciales puntos de entrada y reportar lo encontrado

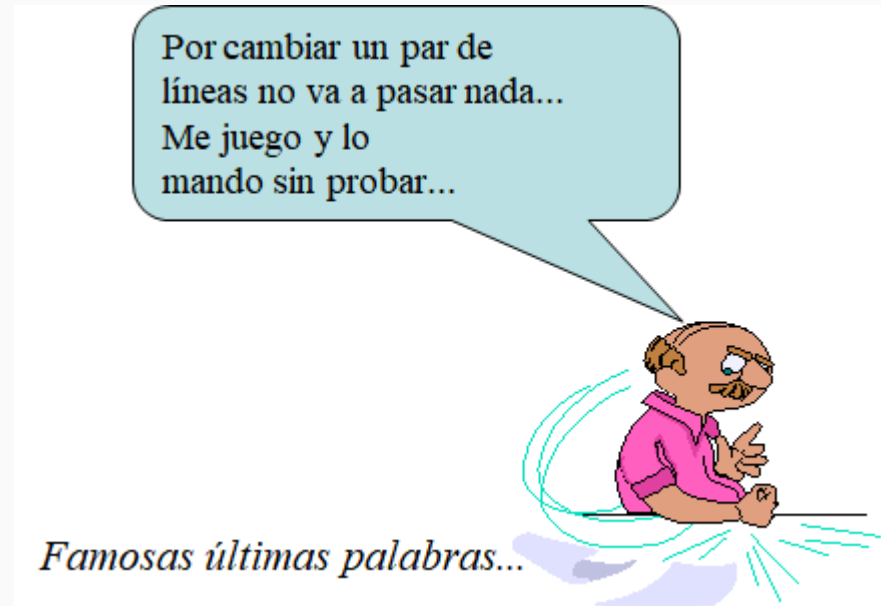
Prueba de Volumen

- Orientada a verificar a que el sistema soporta los volúmenes máximos definidos en la cuantificación de reqs.
 - Capacidad de Almacenamiento
 - Capacidad de Procesamiento



Prueba de regresión

- Orientada a verificar que, luego de introducido un cambio en el código, la funcionalidad original no ha sido alterada.
 - Hay que probar “lo viejo”
 - Reuso de condiciones & casos



Prueba Alfa y 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
 - Beta: la hace el usuario en sus instalaciones



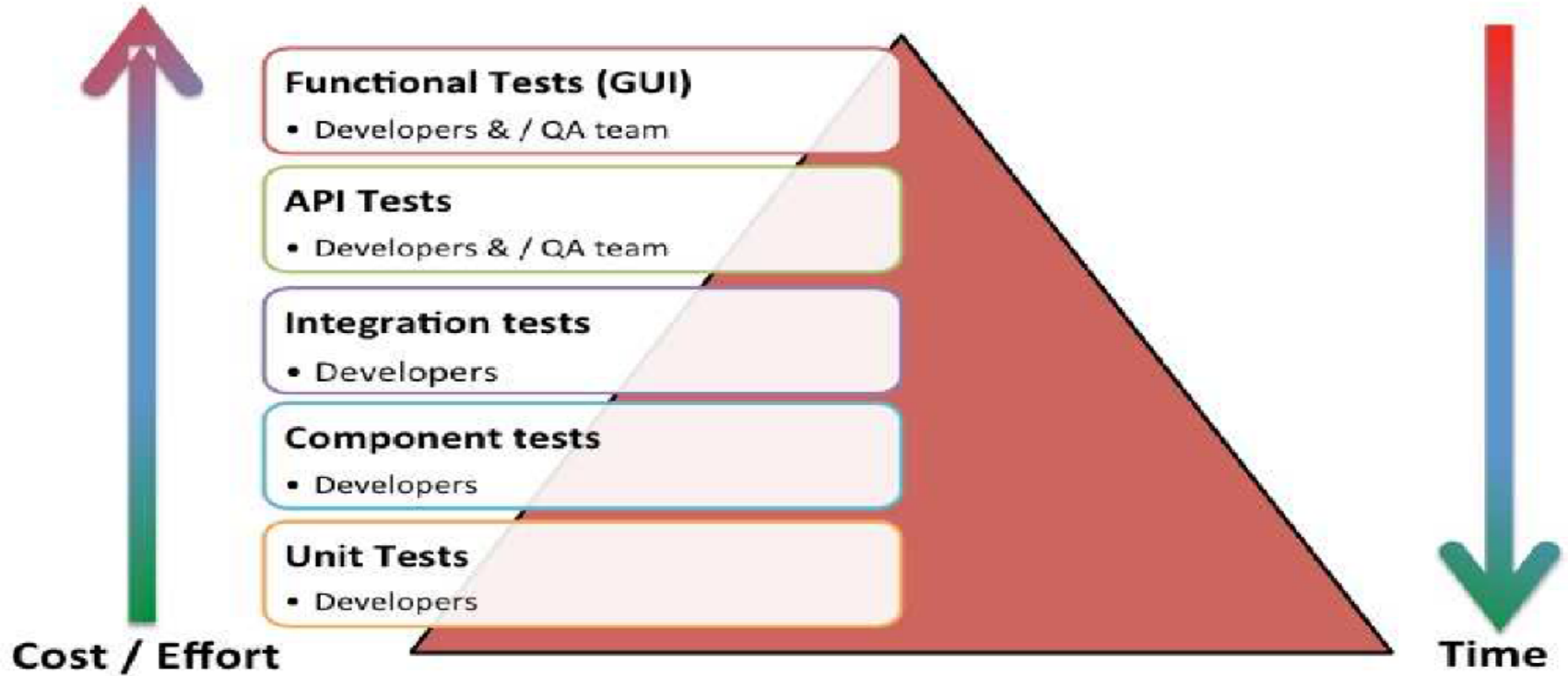
Modelos de Ciclos de Testing

Clasificación de las pruebas

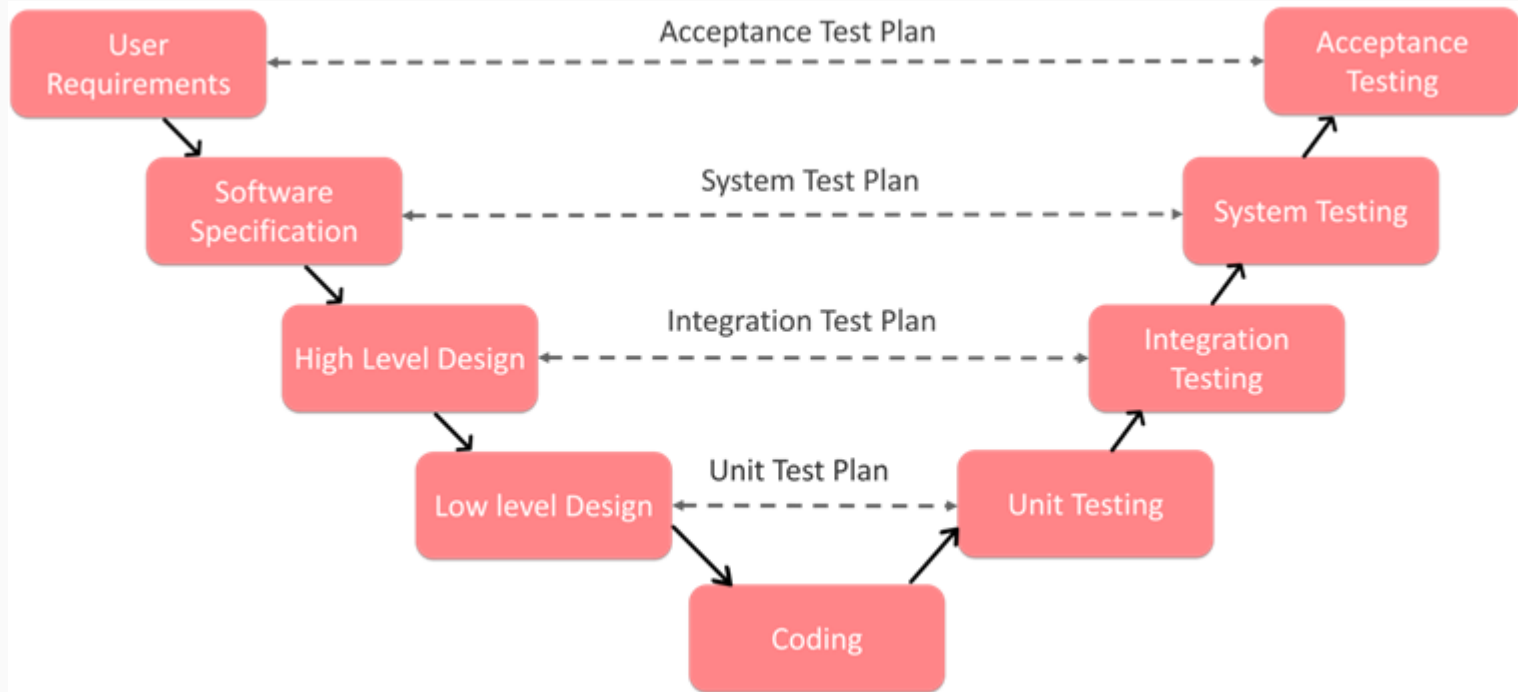
- Por el conocimiento
 - Caja blanca
 - Caja negra
 - Caja gris
- Por el tipo de ejecución
 - Automatizadas
 - Manuales
 - Semi-automatizadas
- Por el tipo de prueba
 - Funcionales
 - No Funcionales
- Por el alcance o nivel
 - Unitarias
 - De integración
 - De sistemas
 - De aceptación de usuario



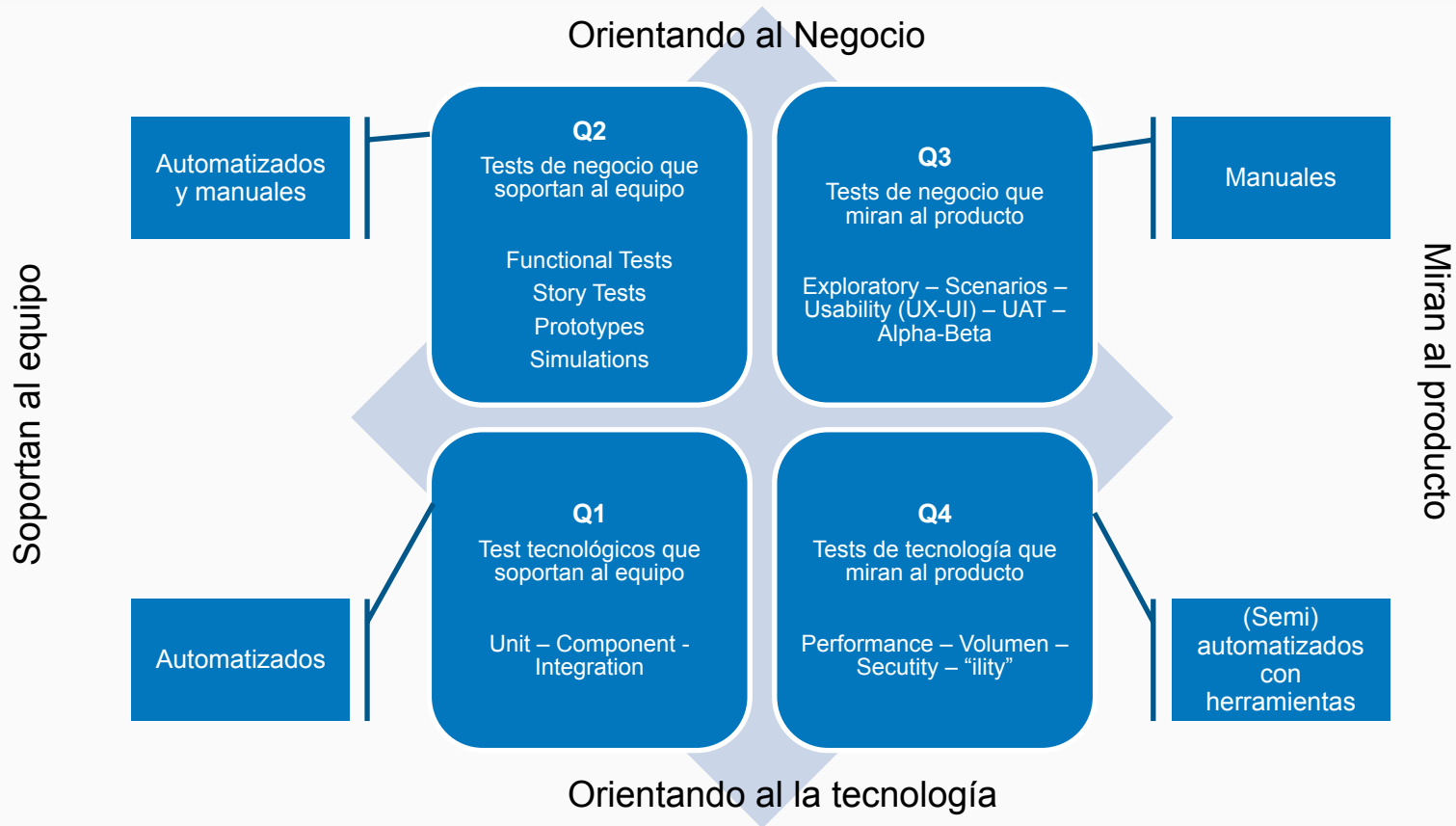
Pirámide del Testing



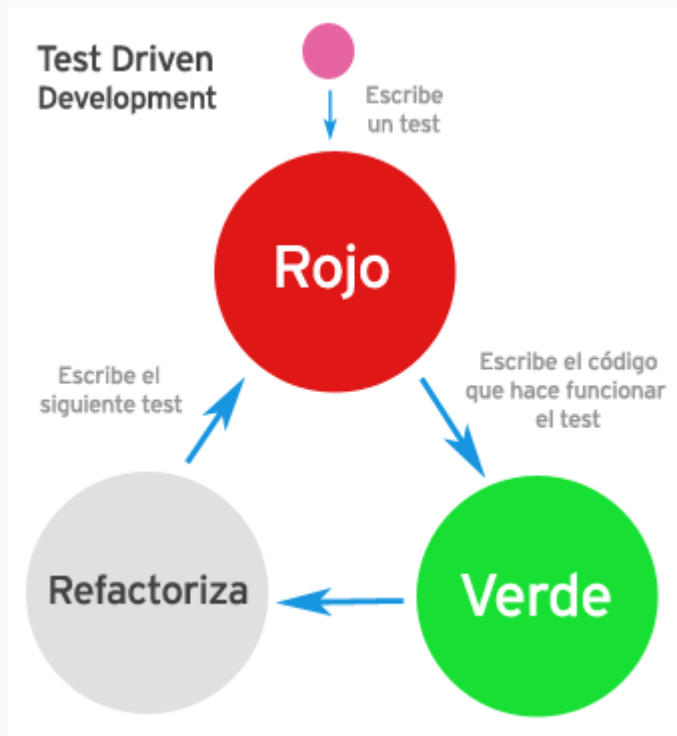
Modelo de Ciclo de Vida V en Testing



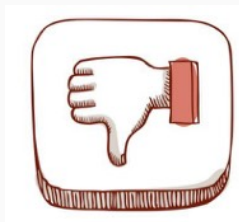
Cuadrantes, niveles y tipos de testing



Agile testing



- Aplicaciones más modulares, flexibles, y escalables
- Código menos acoplado y más mantenible
- El codificador puede hacer refactoring en cualquier momento con bajo riesgo
- Soporta la regresión de bugs
- Mejor cobertura de código



- Similares errores en el código y en el test case
- Algunas funcionalidades son difíciles de probar con estos tests
- Mantenimiento de los tests
- Puede llevar a un falso nivel de confianza sobre la calidad del código

Principios básicos para tener en cuenta

- La definición del resultado esperado es parte integrante y necesaria de la prueba
- Un programador debe evitar probar su propio componente (TDD?)
- Es necesario revisar a fondo el resultado de la prueba
- Evite condiciones y casos de prueba descartables
- No suponer a priori que no se encontrarán errores
- Probar es un desafío intelectual
- El objetivo de la prueba es encontrar fallas, no demostrar que el SW no las tiene

Conclusiones

- Las pruebas no mejoran el SW, sólo muestran cuántas 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 más barato para encontrar y eliminar defectos es NO introducirlos
- Hay herramientas que ayudan a automatizar las pruebas, pero requieren esfuerzo para crear y mantener y ayudan aunque no son suficientes