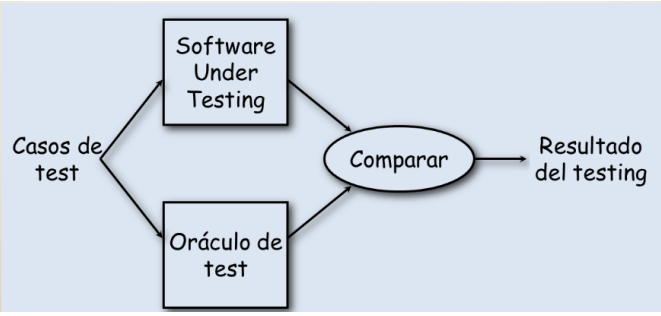
Clase 9-Testing

Defecto y desperfecto

- **Desperfecto**: un desperfecto de software ocurre si el comportamiento de este es distinto del esperado/especificado.
- **Defecto:** es lo que genera el desperfecto. = bug.
  - Un desperfecto implica la presencia del defecto
  - la existencia del defecto no implica la ocurrencia del desperfecto. Pero igual el defecto tiene el potencial para causar el desperfecto
  - Lo que se considere un desperfecto depende del proyecto.
- La idea del testing es poder identificar los defectos del sw para garantizar calidad.
  - Durante el testing, el programa se lo ejecuta en un conjunto de casos de test. Buscando causar desperfectos para así detectar la presencia de defectos. Luego para identificar el defecto real, se debe debuggear.

Oráculos de tests

Un oraculo es una entidad que conoce el resultado esperado de los casos de test. Su funcion es poder verificar la ocurrencia de un desperfecto en la ejecucion de un caso de test.




```
El oraculo puede ser generado automaticamente a partir de la especificacion. (y en pocos casos es humano). Lo ideal es que sea algo generado automaticamente.
```

Casos de test y criterios de seleccion

A la hora de realizar el testing, deseamos poder construir un conjunto de test tal que la ejecucion satisfactoria de todos ellos imploque la ausencia de defectos. Como testear es costoso, se busca que sea un conjunto reducido.

A la hora de elegirlos se usa algun **criterio de seleccion de tests**

```
El criterio de seleccion especifica las condiciones que el conjunto de casos de test
debe satisfacer con respecto al programa y/o a la especificacion.
```

Se busca que los casos de test cumplan dos propiedades, **confiabilidad** y **validez**.

Testing de caja negra

Se basa en que el software a testear se trata como una **caja negra**:

- Donde la especificación esta dada
- Para diseñar los casos de test, se utiliza el comportamiento esperad del sistema. Es decir, los casos de test se seleccionan sólo a partir de la especificación
- No se utiliza la estructura interna del código.

La **premisa** es que el comportamiento esperado está especificado. Luego solo se definen test para el comportamiento esperado especificado.

- En el testing de módulos: la especificación producida por en el diseño define el comportamiento esperado.
- Para el testing del sistema: la SRS define el comportamiento esperado

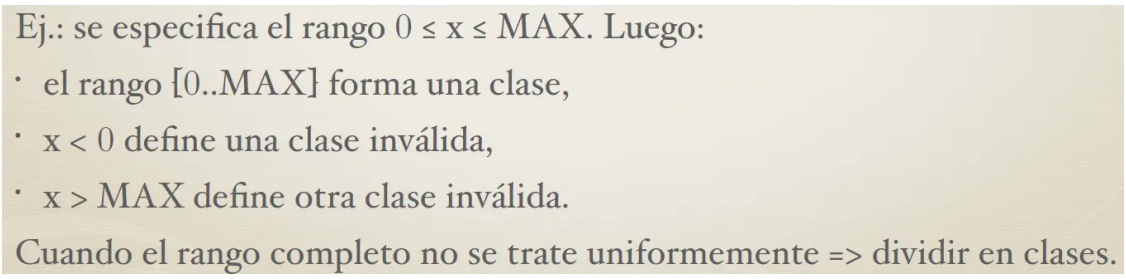
El testing **exhaustivo** es el mas minucioso. Ya que un sw esta diseñado para trabajar sobre un espacio de entrada -> Se testea el sw con todos los elementos del espacio de entrada. Esto es algo claramente muy costoso e inviable.

Clases de equivalencia

Se divide el espacio de entrada en clases de equivalencias

- De modo que si el software funciona para un caso de test en una clase, muy probablemente funcione de la misma manera para todos los elementos de la misma clase.
  - obtener un particionado ideal es imposible. Pero se aproxima identificando las clases para las cuales se especifican dsitintos comportamientos
- La especificacion requiere el mismo comportamiento en todos los elementos de una misma clase. Si falla para uno, falla para todos.

Cada condicion especificada como entrada es una clase de equivalencia. Por ello para lograr robustez, se deben armar clases de equivalencias para entradas inválidas.



Ademas se deben considerar las clases de equivalencia de los datos de salida y generar los casos de test para estas clases eligiendo apropiadamente las entradas. Una vez elegidas las clases de

equivalencia, se deben seleccionar los casos de test:

1. Seleccionar cada caso de test cubriendo tantas clases como sea posible
2. Dar un caso de tes que cubra a lo sumo una clase válida por cada entrada.

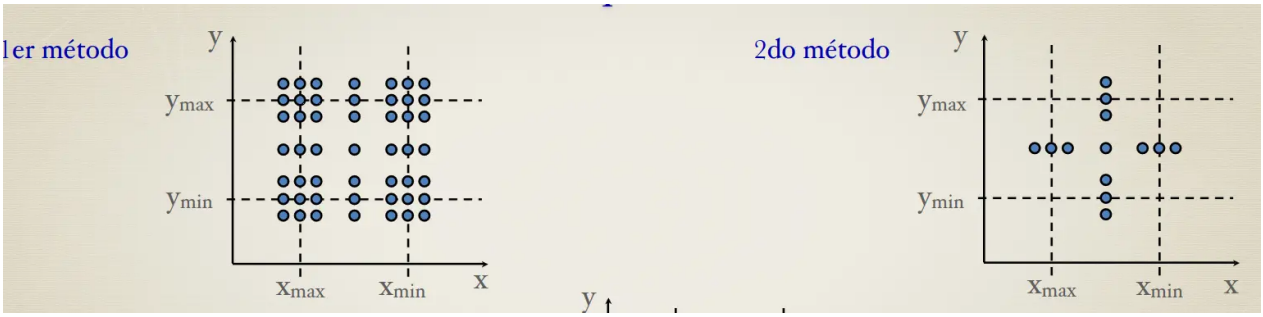
- Ademas de los casos de test separados por cada clase inválida.

Análisis de valores limites

Los programas generalmente fallan sobre valores especiales. Estos valores suelen estar en los limites de las clases de equivalencia. (también llamados casos extremos).

Un caso de test de valores límites es un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida. Los casos de test de valores limites tienen alto rendimiento.

- Primero se determinan los valores a utilizar para cada variable. Si la entrada tiene un rango definido hay 6 valores del limite (2 afura, 2 adentro, 2 del limite) mas el caso normal.
- Si tenemos multiples entradas hay dos alternativas
  1) ejecutar todas lass combinaciones posibles de las variables 7^n casos de test.
  2) Seleccionar los casos limites para una varaible y mantener las demas en casos normales. mas el caso de todo normal. $6n + 1$ casos de test.



Grafo causa efecto

Los analisis de clase de equivalencia y valores limites consideran cada entrada separadamente. Para manipular las entradas, distintas combinaciones de las clases de equivalencia deben ser ejecutadas. Si hay n condiciones en la entrada que puden ser validas o invalidas hay 2^n clases de eq.

El grafo causa-efecto ayuda a seleccionar las combinaciones como condiciones de entrada

- **Causa**: distintas condiciones en la entrada que pueden ser V o F
- **Efecto**: distintas condiciones de salidas (V o F)
  Se busca identificar cuales causas pueden producir que efectos; las causas se pueden combinar.

Las causas y efectos son nodos en el grafo, y las aristas determinan dependancia (positivas o negativas) tambien hay nodos and y or para combinar causalidad.

El grafo sirve para poder crear **tabla de decision**. Que esta se usa para armar distintos casos de test.

Testing de a pares

El comportamiento del sistema suele estar determinado por muchos parametros. Estos puden ser entradas o seteos y toman distintos valores o rango de valores.

- Tipos de defectos
  - De modo simple: involucran solo una condicion. Los defectos de modo simple pueden detectarse verifiando distintos valores de los distintos parametros. Si hay n parametros con m tipos de valores c/u. Podemos testear cada valor distinto en cada test por cada parametro. **m** casos de test en total
  - Los defectos de modo multiple se revelan con casos de test que contemplen las combinaciones apropiadas. Para esto surge el test de a pares
  Hacer test combinatorio (osea testear cada combinacion posible de parametros) no es factible ya que demora mucho tiempo. Y como la mayoria de defectos se revelan con interaccion de apares. Se testea en cada par posible.

Todos los pares de valores deben ser ejercitados. Si tenemos n parametros con m valores. Para cada par de parametros tenemos m *m pares. Luego el total de test que habra por hacer sera $m^2* n *(n-1)/2$. Y en el mejor caso un test cubre $n*(n-1)/2$ pares. Luego en el mejor caso total, tendremos m^2 casos de test distintos que proveen cobertura completa.

Testing basado en estados

Esta dirigido para los sistemas que según su estado cambian su comportamiento. Un estado del sistema representa el impacto acumulado de las entradas pasadas. Estos sistemas pueden modelarse con un modelo de estados , suele construirse a partir de las especificaciones o los requerimientos. El desafío mas importante es a partir de la especificación/requerimientos identificar el conjunto de estados que captura las propiedades claves. Tiene 4 componentes.

- Un conjunto de estados: son estados lógicos representando el impacto acumulativo del sistema
- Un conjunto de transiciones: representa el cambio de estado en respuesta a algún evento de entrada
- Un conjunto de eventos: entradas del sistema
- Un conjunto de acciones: son las salidas producidas en respuesta a los eventos de acuerdo al estado actual
  Los casos de test que se generan para los sistemas que tienen modelos de estados, pueden tener diferentes coberturas.
- Cobertura de transiciones: el conjunto T de casos de test debe asegurar que todas transición sea ejecutada
- Cobertura de par de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado.
- Cobertura de árbol de transiciones: T debe ejecutar todos los caminos simples. (del estado inicial al final o a uno visitado)

Testing de caja blanca

- Se enfoca en el código.
- El objetivo es ejecutar las distintas estructuras del programa con el fin de descubrir errores.
- Los casos de test se derivan a partir de código

Flujo de control

Se considera al programa como un grado de flujo de control.
- Los nodos representan bloques de código.
- Una arista i,j. Representa una posible transferencia de control del nodo i a j.

Cobertura de sentencia

Busca que cada sentencia se ejecute al menos una vez durante el testing. El conjunto de caminos ejecutados durante el testing debe incluir todos los nodos.
- No es posible garantizar 100% de cobertura debido a que puede haber nodos inalcanzables

Cobertura de ramificaciones

- Cada arista debe ejecutarse al menos una vez en el testing.(Cada decision debe ejercitarse como verdadera y como falsa durante el testing.)
- Implica la cobertura de sentencias.
- Si hay múltiples condiciones en una decisión luego no todas las condiciones se ejercitan como verdadera y falsa.

Cobertura de caminos

- Todos los posibles caminos del estado inicial al final deben ser ejercitados
- Cobertura de caminos implica cobertura de bifurcación
- La cantidad de caminos puede ser infinita y algunos no realizables.

En general el criterio de flujo de control se utiliza para proveer alguna idea cuantitativa de la "amplitud" y cobertura del conjunto de casos de test. Se suele utilizar mas para evaluar el nivel de testing que para seleccionar los casos de test.

Flujo de datos

Se construye un grafo de definicón-uso etiquetando apropiadamente el grado de flujo de control
Una sentencia en el grafo de flujo de control puede ser de tres tipos

- Def: representa la definición de una variable
- uso-c: cuando la variable se usa para cómputo
- uso-p: cuando la variable se utiliza en un predicado para transferencia de control

El grafo de def-uso se construye asociando variables a nodos y aristas del grafo de flujo de control:

- Por cada nodo i, def(i) es el conjunto de variables para el cual hay una definición en i.
- Por cada nodo i, c-use(i) es el conjunto de variables para el cual hay un uso-c.
- Para una arista (i,j), p-use(i,j) es el conjunto de variables para el cual hay un uso-p.

Un camino de i a j se dice libre de definiciones con respecto a una var x si no hay definiciones de x en todos los nodos intermedios.

Algunos criterios son:

- todas las definiciones: por cada cono i y cada x en def(i) hay un camino libre de definiciones con respecto a x hasta un uso-c o uso-p de x.
- Todos los usos-p: todos los usos-p de todas las definiciones deben testearse
- otros criterios: todos los usos-c, algunos usos-p, algunos usos-c.

Proceso de testing

Los objetivos del testing son:

- detectar tantos defectos como sea posible a bajo costo.
  El testing incremental busca agregar partes no testeadas incrementalmente a la parte ya testeada. Esto ayuda a encontrar más defectos y ayuda a la identificación y eliminación de los mismos.
  Existen diferentes niveles de testing para revelar los distintos tipos de defectos.

Testing de unidad

- Cada modulo del programa se testean separademente contra el diseño.
- Se enfoca en los defectos inyectados durante la codificación.
- En gral son realizados por el mismo programador que realizo el modulo.

Testing de integración

- Se enfoca en la interacción de módulos de un subsistema
- Los módulos ya testeados unitariamente se combinan para formar subsistemas. Y se someten a test de integracion
- Los casos de test deben generarse con el objeto de ejercitar de distinta manera la interaccion entre los módulos.

Testing del sistema

- Se testea el sistema de sw completo.
- Se busca ver si implementa los requerimientos, forma parte de la validacion del sistema.
- Se realiza antes de entregar el mismo y por personal independiente.

Testing de aceptación

- Se enfoca en verificar que el software satisfaga las necesidades del usuario.
- Es realizado por un usuario en el entorno del cliente y con datos reales.
- El plan de test de aceptacion se basa en el criterio del test de aceptacion y la SRS.

- Testing de desempeño:
  - Requiere de herramientas para medir el desempeño.
- Testing de estrés (stress testing):
  - El sistema se sobre carga al máximo; requiere de herramientas de generación de carga.
- Testing de regresión:
  - Se realiza cuando se introduce algún cambio al software (¡es importante de realizar!).
  - Verifica que las funcionalidades previas continúen funcionando bien.
  - Se necesitan los registros previos para poder comparar.
    => tests deben quedar apropiadamente documentados (+ scripts para automatizarlos).
  - Priorizar los casos de tests necesarios cuando el test suite completo no pueda ejecutarse cada vez que se realiza un cambio.

El plan de test

El testing usualmente comienza ocn la realizacion del plan de test y finaliza con el testing de aceptación. Es un documento que toma como entradas el plan del proyecto, la SRS y el diseño. Define el alcance y el enfoque del testing para el proyecto completo, debe ser consistente con el plan de calidad del proyecto y el cronograma de testing debe ser acorde al del proyecto.

- Identifica que niveles de testing se realizarán, qué unidades serán testeadas, etc.
- Se puede realizar antes de comenzar con la tarea del testing conjuntamente con las actividades de diseño y codificación.

Contiene:

1. Especificación de la unidad de test: que unidad necesita testearse separadamente
   - Una unidad de test es un conjunto de uno o más modulos conjuntamente con datos asociados que son el objeto del testing.
2. Características a testear: esto incluye funcionalidad, desempeño, usabilidad, etc.
   - Establece el nivel de testing
3. Enfoque: criterios a utilizarse, cuando detenerse, como evaluar, etc.
4. "Entregables"(Testing deliberables)
   - ej: lista de casos utilizados, resultados detallados. Reporte resumido, cobertura, etc..
5. Cronograma y asignación de tareas.

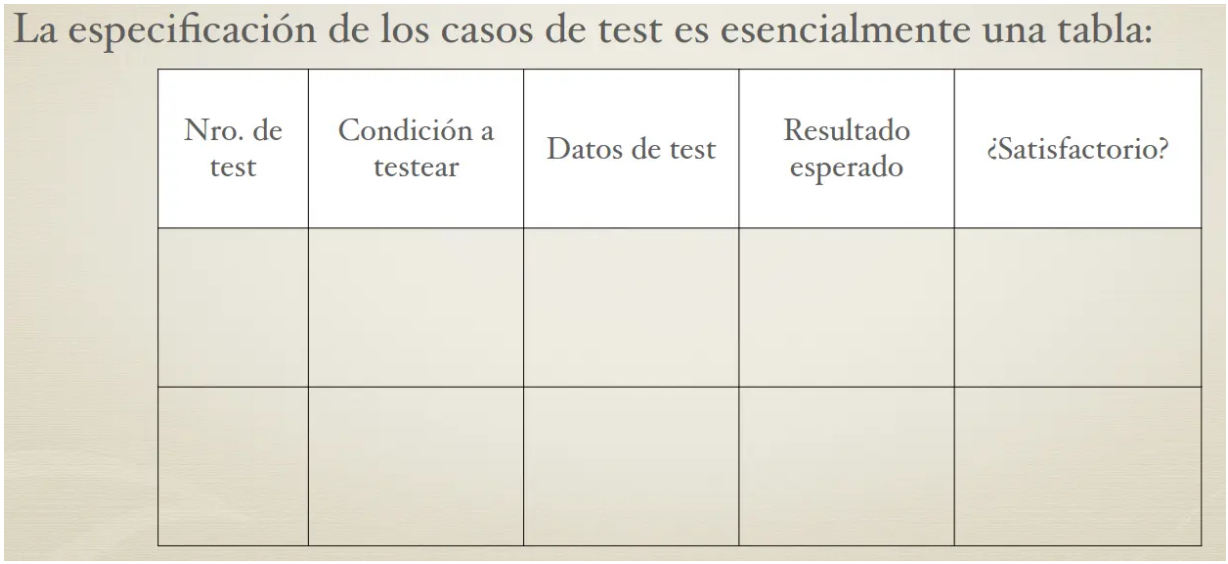
El plan de test se enfoca en cómo proceder y qué testear pero no trata detalles del testeo de una unidad. La especifiación de casos de test se tiene que realizar separadamente para cada unidad. Por cada unidad de test se determinan los casos de te st de acuerdo con el plan. Con cada caso de test se especifica:

- entradas a utilizar
- condiciones en las que se testeará
- resultado esperado.
  Es una justifiación del caso de test.

Especificacion de casos de test

La **efectividad** y **costo** del testing dependen del conjunto de casos de test seleccionados. Como no es posible detectar si un caso de test es bueno o malo , es por lo que se necesitan especificaciones de cada

caso de test para que sean analizados.



Para preparar la especificacion se puede:

- utilizar cirterios para casos de test
- utilizar casos especiales y escenarios
  Una vez especificados, la ejecucion y verificacion del resultado se puede automatizar mediante scripts.

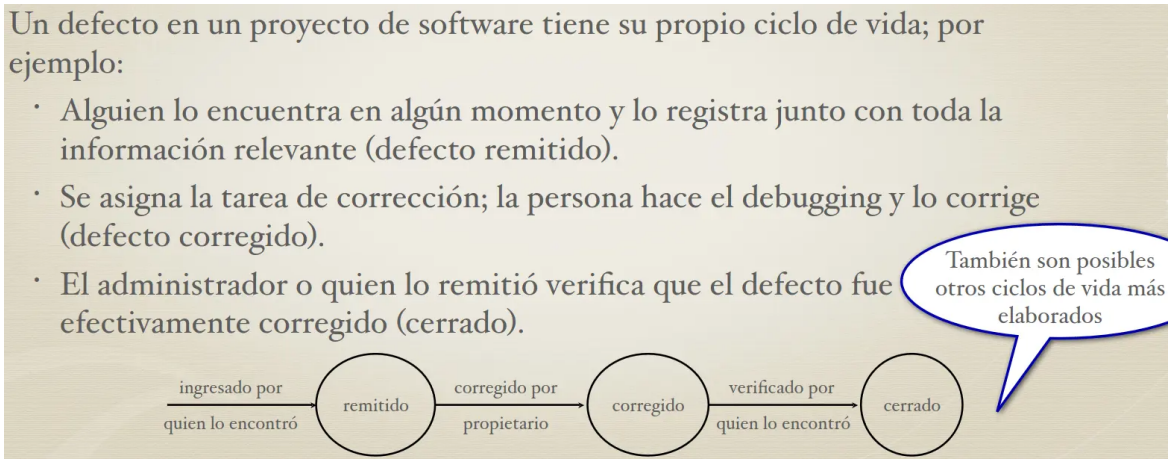
Una vez especificado un caso de test, se lo ejecuta. Puede que se requera de ecribir "drivers" o "stubs" o incluso modulos estras para preparar el entorno acorde a las condicinoes establecidas en la especificación del caso de test.

Una vez ejecutados se realizan reportes con resúmenes del test:
- reporta un resumen de los casos de test ejecutados, el esfuerzo y defectos encontrados.

El seguimiento y el control del esfuerzo del testing es importante para asegurar que se invirtió el tiempo suficiente.

Registro de defectos y seguimiento

En gral las personas que encuentran defectos no son las mismas que los corrigen. Es por esto que los defectos se registran en un **sistema seguidor de defectos** que permite rastrearlos hasta que se "cierren".



Los defectos se suelen categorizar. Algunas categorías son: funcional, lógica, standard, asignación, interfaz de usuario, desempeño, documentación, etc.

Ademas se los clasifica por gravedad:

- Critico: puede demorar el proceso, afecta muchos usuarios
- Mayor: tiene mucho impacto pero posee soluciones provisorias, requiere de mucho esfuerzo para corregirlo pero tiene menor impacto
- Menor: defecto aislado que se manifiesta raramente y tiene poco impacto
- Cosmético: pequeños errores sin impacto en el funcionamiento

No siempre se entrega un software 100% libre de defectos, se los suele entregar con defectos conocidos. Cada org tiene un estándar para determinar cuando un producto se puede entregar.