9 - Testing

El principal objetivo de un **proyecto** es desarrollar un **producto de alta calidad con alta productividad.** La calidad tiene muchas dimensiones: Confiabilidad, mantenibilidad, interoperabilidad, etc. La confiabilidad es la más conocida, habla de las posibilidades que el software falle: Más defectos implica más chances de que falle, lo cual significa menor confiabilidad. Así, nos centraremos en el objetivo de calidad de que el producto entregado tenga la **menor cantidad de defectos** como sea posible.
El **objetivo del testing** es encontrar la **mayor cantidad de errores posible**.

Conceptos fundamentales

Desperfecto y defecto (Failure & Fault)

Un **desperfecto** de software ocurre si el comportamiento de éste es distinto del esperado/especificado. Por otro lado, un **defecto** (o bug) es lo que causa el desperfecto.

Un desperfecto implica la presencia del defecto (porque es la observación del mismo), pero la existencia del defecto no implica la ocurrencia del desperfecto, aunque un defecto tiene el potencial para causar el desperfecto.
Qué se considera un desperfecto depende del proyecto que se está llevando a cabo.

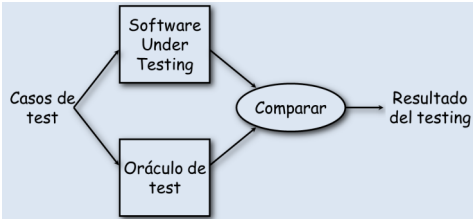
Los defectos tienen que identificarse por medio del testing, jugando así un rol crítico cuando se trata de garantizar calidad.

Durante el testing, un programa se ejecuta siguiendo un conjunto de casos de test. Si hay desperfectos en la ejecución de un test, entonces hay defectos en el software. Si no ocurren desperfectos, entonces la confianza en el software crece, pero no podemos negar la presencia de defectos. **Los defectos se detectan a través de los desperfectos**. Para detectar los defectos debemos causar desperfectos durante el testing. Para identificar el defecto real (que causa el desperfecto) debemos recurrir al debugging.

Oráculo

El oráculo es el ente que sabe la **respuesta esperada** o considerada correcta.
Sirve para **detectar desperfectos** y es usado para chequear la correctitud de una salida para los casos de test.

Idealmente pretendemos que el oráculo siempre dé el resultado correcto. El oráculo humano usa la especificación para decidir el “comportamiento correcto”. Pero hay que notar que las mismas especificaciones pueden contener errores. En algunos casos, los oráculos pueden generarse automáticamente dependiendo de la especificación.



Casos de test y criterios de selección

Si existen defectos, deseamos que los casos de test los evidencien a través de fallas. Deseamos construir un conjunto de casos de test tal que la ejecución satisfactoria de todos ellos **implique la ausencia de defectos**. Además, como el testing es costoso, deseamos que sea un **conjunto reducido.** Vemos que estos dos **deseos son contradictorios.**

Es necesario usar algún criterio de selección de tests, que especifique las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o a la especificación.

Hay dos **propiedades fundamentales que esperamos de los criterios de test:**

- **Confiabilidad**: Un criterio es confiable si todos los conjuntos de casos test que satisfacen el criterio detectan los mismos errores.
- **Validez**: Un criterio es válido si para cualquier error en el programa hay un conjunto de casos de test que satisfagan tal criterio y detecten el error.

Es prácticamente imposible obtener un criterio que sea confiable y válido al mismo tiempo, y que también sea satisfecho por una cantidad manejable de casos de test.

Psicología y enfoques

La psicología del testing es importante: debe revelar defectos (contrariamente a mostrar que funciona), **los casos de test deben ser "destructivos"**. Dos enfoques para diseñar casos de test: de **caja negra** o funcional, de **caja blanca** o estructural. Ambos son complementarios.

El testing de caja negra se enfoca sólo en la funcionalidad: Lo que el programa hace no en cómo éste implementado.
El testing de caja blanca se enfoca en la implementación del código: El objetivo es ejecutar las distintas estructuras del programa para descubrir errores. **A diferencia del testing de caja negra, el testing de caja blanca tiene acceso al código.**

Testing de caja negra

El software a testear se trata como una caja negra. La especificación de la caja negra está dada, y **para diseñar los casos de test, se utiliza el comportamiento esperado del sistema.** Así, los casos de test se seleccionan **sólo a partir de la especificación**. No se utiliza la estructura interna del código.

Premisa: el comportamiento esperado está especificado. Entonces sólo se definen test para el comportamiento esperado especificado.

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

El testing funcional más minucioso es el exhaustivo: El software está diseñado para trabajar sobre un espacio de entrada, por lo que se debería testear el software con todos los elementos del espacio de entrada. Sin embargo, esto no es viable, es demasiado costoso. Necesitamos mejores métodos para elegir los casos de test, veremos diferentes enfoques.

Particionado por clases de equivalencia

Se debe dividir el espacio de entrada en clases de equivalencias. La obtención del particionado ideal es imposible, se debe aproximar identificando las clases para las cuales se especifican distintos comportamientos.

La base lógica es que la especificación requiere el **mismo comportamiento en todos los elementos de una misma clase**, por lo que es muy probable que el **software se construya de manera tal que falle para todos o para ninguno.**

Ej.: si una función no fue diseñada para números negativos, fallará para todos los números negativos. Cada condición especificada como entrada es una clase de equivalencia. Para lograr robustez, se deben armar clases de equivalencias para entradas inválidas.

Ej.: se especifica el rango $0 \leq x \leq MAX$. Luego: el rango [0..MAX] forma una clase, $x < 0$ define una clase inválida, $x >$ MAX define otra clase inválida**. Cuando el rango completo no se trate uniformemente, se debe dividir en clases.**

También se deben considerar las **clases de equivalencia de los datos de salida**. Generar los casos de test para estas clases eligiendo apropiadamente las entradas.

Una vez elegidas las clases de equivalencia, hay que elegir los casos de test, con alguno de los siguientes dos métodos:

Métodos para elegir los casos de test

1. Seleccionar cada caso de test cubriendo tantas clases como sea posible.
2. O dar **un** caso de test que cubra a lo sumo una clase **válida** por cada entrada. Además de los **casos de test separados por cada clase inválida.**

Análisis de valores límites (respecto a las clases de equivalencia)

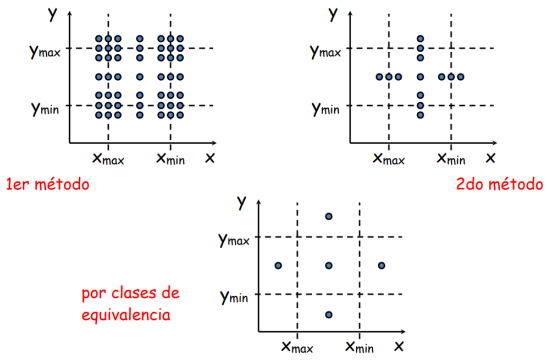
Los programas generalmente **fallan sobre valores especiales**, que usualmente se encuentran en los límites de las clases de equivalencia. Los casos de test que tienen valores límites tienen alto rendimiento.
Un **caso de test de valores límites** es un **conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida**.También se denominan casos extremos.
Para cada clase de equivalencia:

1. Elegir valores en los límites de la clase.
2. Elegir valores justo fuera y dentro de los límites.
3. (Además del valor normal)

Hay que considerar las **salidas** también y producir casos de test que generen salidas sobre los límites.

En primer lugar se determinan los valores a utilizar para cada variable. Si la entrada tiene un rango definido, entonces hay 6 valores de límite más un valor normal (total: 7). Para el caso de múltiples entradas hay dos estrategias para combinarlas en un caso de test:

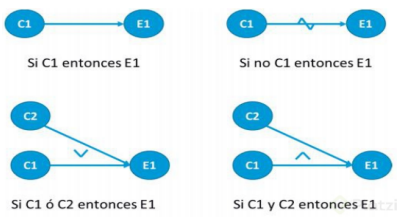
- Ejecutar todas las combinaciones de las distintas variables (7^n casos de test!).
- Seleccionar los casos límites para una variable y mantener las otras en casos normales y considerar el caso de todo normal (total 6n + 1).



Grafo de causa efecto

Los análisis de clase de equivalencia y valores límites consideran cada entrada separadamente. Para manipular las entradas, distintas combinaciones de las clases de equivalencia deben ser ejecutadas. La cantidad de combinaciones puede ser grande: si hay n condiciones distintas en la entrada que puedan hacerse válidas o inválidas, entonces hay 2^n clases de equivalencia. El grafo de causa-efecto ayuda a seleccionar las combinaciones como condiciones de entrada.
Para crearlo, debemos:

- Identificar las causas y efectos en el sistema
  - **Causa**: distintas condiciones en la entrada que pueden ser V o F.
  - **Efecto**: distintas condiciones de salidas (V/F también).
- Identificar cuáles causas pueden producir qué efectos; las causas se pueden combinar.
- Causas y efectos son nodos en el grafo.
- Las aristas determinan dependencia: hay aristas “positivas” y “negativas”.
- Existen nodos “and” y “or” para combinar la causalidad.



A partir del grafo de causa-efecto se puede armar una tabla de decisión, que lista las combinaciones de condiciones que hacen efectivo cada efecto. La tabla de decisión puede usarse para armar los distintos casos de test.

Testing de a pares

Usualmente muchos parámetros determinan el comportamiento del sistema. Los parámetros pueden ser entradas o seteos y pueden tomar distintos valores (o distintos rangos de valores). Muchos defectos involucran sólo una condición (defecto de modo simple). Los defectos de modo simple pueden detectarse verificando distintos valores de los distintos parámetros. Si hay n parámetros con m (tipos de) valores c/u, podemos testear cada valor distinto en cada test por cada parámetro. i.e: m casos de test en total.

Pero no todas los defectos son de modo simple: el software puede fallar en combinaciones: Los defectos de modo múltiple se revelan con casos de test que contemplen las combinaciones apropiadas. Esto se denomina test combinatorio. Pero **el test combinatorio no es factible**: n parámetros / m valores => n^m casos de test. **Se investigó que la mayoría de tales defectos se revelan con la interacción de pares de valores. i.e. la gran mayoría de los defectos tienden a ser de modo simple o de modo doble.** Para modo doble necesitamos ejercitar cada par, esto se denomina testing de a pares.

En testing de a pares, todos los pares de valores deben ser ejercitados. Si tenemos n parámetros / m valores, para cada par de parámetros tenemos m*m pares: el 1er param. tendrá m*m contra n-1 otros el 2do param. tendrá m*m contra n-2 otros el 3er param. tendrá m*m contra n-3 otros etc. Total: m^2 * n * (n-1) / 2

Un caso de test consiste en algún seteo de los n parámetros**. El menor conjunto de casos de test se obtiene cuando cada par es cubierto sólo una vez.** Un caso de test puede cubrir hasta (n-1)+(n-2)+...= n*(n-1)/2 pares. En el mejor caso, cuando cada par es cubierto exactamente una vez, tendremos m^2 casos de test distintos que proveen una cobertura completa.
La generación del conjunto de casos de test más chico que provea cobertura completa de los pares no es trivial. Existen algoritmos eficientes para generación de casos de test que pueden reducir el esfuerzo de testing considerablemente. El testing de a pares es un enfoque práctico y muy utilizado en la industria.

Testing basado en estados

Algunos sistemas no tienen estados: para las mismas entradas, se exhiben siempre las mismas salidas. Sin embargo, en muchos sistemas, el comportamiento depende del estado del sistema, i.e. para la misma entrada, el comportamiento podría diferir en distintas ocasiones esto es, el comportamiento y la salida depende tanto de la entrada como del estado actual del sistema. El estado del sistema representa el impacto acumulado de las entradas pasadas. El testing basado en estado está dirigido a tales sistemas.

Un sistema puede modelarse como una máquina de estados. Si bien el espacio de estados puede ser demasiado grande (es el producto cartesiano de todos los dominios de todas las variables), puede particionarse en pocos estados, cada uno **representando un estado lógico de interés del sistema**.El modelo de estado se construye generalmente a partir de tales estados.

Un **modelo de estados** tiene cuatro 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:** son las entradas al sistema.
- **Un conjunto de acciones:** son las salidas producidas en respuesta a los eventos de acuerdo al estado actual.

El **desafío** más importante es, a partir de la especificación/ requerimientos, identificar el conjunto de estados que captura las propiedades claves pero es lo suficientemente pequeño como para modelarlo.

El modelo de estado muestra la ocurrencia de las transiciones y las acciones que se realizan. Usualmente el modelo de estado se construye a partir de las **especificaciones o los requerimientos.**
El modelo de estado puede crearse a partir de la especificación o del diseño. En el caso de los objetos, los modelos de estado se construyen usualmente durante el proceso del diseño. Los casos de test se seleccionan con el modelo de estado y se utilizan posteriormente para testear la implementación. Existen varios criterios para generar los casos de test:

- **Cobertura de transiciones:** el conjunto T de casos de test debe asegurar que toda 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).

El test basado en estados se enfoca en el testing de estados y transiciones. Se testean distintos escenarios que de otra manera podrían pasarse por alto. El modelo de estado se realiza usualmente luego de que la información de diseño se hace disponible. En este sentido, se habla a veces de **testing de caja gris** (dado que no es de caja negra puro)

Testing de caja blanca

Los casos de test se derivan a partir del código, se denomina también testing estructural. Existen varios criterios para seleccionar el conjunto de casos de test.

Tipos de testing estructural:

- Criterio basado en el **flujo de control.** Observa la cobertura del grafo de flujo de control.
- Criterio basado en el **flujo de datos.** Observa la cobertura de la relación definición-uso en las variables.
- Criterio basado en **mutación**. Observa a diversos mutantes del programa original.

Criterio basado en flujo de control

Considerar al programa como un grafo de flujo de control:

- Los nodos representan **bloques de código**, i.e., **conjuntos de sentencias** que siempre se ejecutan juntas.
- Una arista (i,j) representa una **posible transferencia de control del nodo i al j.**

Suponemos la existencia de un nodo inicial y un nodo final. **Un camino es una secuencia del nodo inicial al nodo final.** Ahora, se subdivide en tres criterios:

Criterio de cobertura de sentencias

Cada sentencia se ejecuta al menos una vez durante el testing. i.e. el conjunto de caminos ejecutados durante el testing debe incluir todos los nodos.

Limitación: Depende de la definición de sentencia que se esté usando. Puede no requerir que una decisión evalúe a falso en un if si no hay else. Aunque estemos cubriendo el 100% todavía pueden haber defectos no detectados.

Criterio de cobertura de ramificaciones

Cada arista debe ejecutarse al menos una vez en el testing. i.e. cada decisión debe ejercitarse como verdadera y como falsa durante el testing. La cobertura de ramificaciones implica cobertura de sentencias.

Limitación: Si hay múltiples condiciones en una decisión luego no todas las condiciones se ejercitan como verdadera y falsa. Por ejemplo, para las condiciones (x < 0 or y and y < 0) se puede testar x< 0 y de todos modos todavía no se cubre todo perfectamente.

Criterio de cobertura de caminos

Todos los posibles caminos del estado inicial al final deben ser ejecutados. Cobertura de caminos implica cobertura de bifurcación.
Problema: la cantidad de caminos puede ser infinita (considerar loops). Notar además que puede haber caminos que no son realizables.

Existen criterios intermedios (entre el de caminos y el de bifurcación): cobertura de predicados, basado en complejidad ciclomática, etc.
Ninguno es suficiente para detectar todos los tipos de defectos (ej.: se ejecutan todos los caminos pero no se detecta una división por 0).
Proveen alguna idea cuantitativa de la “amplitud” del conjunto de casos de test. Se utiliza más para evaluar el nivel de testing que para seleccionar los casos de test.

Criterio basado en flujo de datos

Se construye un grafo de definición-uso etiquetando apropiadamente el grafo de flujo de control. Una sentencia en el grafo de flujo de control puede ser de tres tipos:

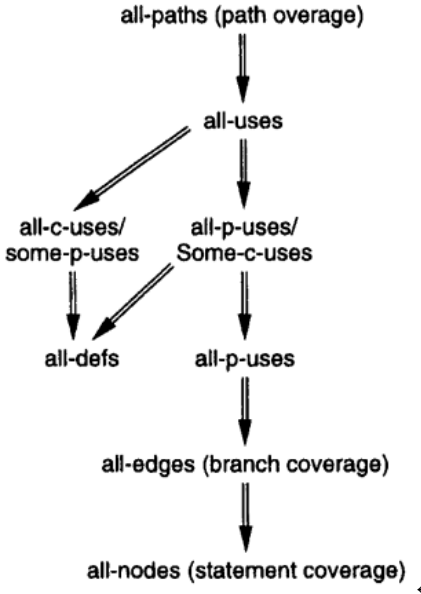
- **def**: representa la definición de una variable (i.e. cuando la var está a la izquierda de la asignación).

- **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 conj. 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.

Criterios:
- todas las definiciones: por cada nodo 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.



← Relaciones entre los distintos criterios

Comparación y uso

Se deben utilizar tanto test funcionales (caja negra) como estructurales (caja blanca). Ambas técnicas son complementarias:
- Caja **blanca** es bueno para **detectar errores en la lógica del programa** (i.e. errores estructurales del programa).
- Caja **negra** es bueno para **detectar errores de entrada/salida** (i.e. errores funcionales).

Los métodos estructurales son útiles a bajo nivel solamente, donde el programa es “manejable” (ej. test de unidad). Los métodos funcionales son útiles a alto nivel, donde se busca analizar el comportamiento funcional del sistema o partes de éste.

Proceso de testing

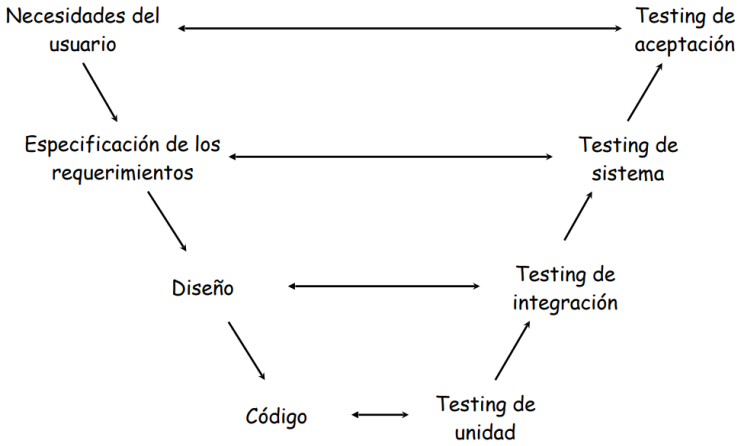
Testing incremental

Los objetivos del testing son: detectar tantos defectos como sea posible, y hacerlo a bajo costo.
Objetivos contrapuestos: incrementar el testing permite encontrar más defectos pero a la vez incrementa el costo.

El testing incremental consiste en agregar partes no testeadas incrementalmente a la parte ya testeada. Es esencial para conseguir los objetivos antedichos: ayuda a encontrar más defectos y ayuda a la identificación y eliminación. El testing de grandes sistemas se realiza siempre de manera incremental.
Sirve para encontrar el error y también **saber dónde pasó**.

Niveles de testing

El código contiene defectos de requerimiento, de diseño y de codificación. La naturaleza de los defectos es diferente para cada etapa de inyección del defecto. Un sólo tipo de testing sería incapaz de detectar los distintos tipos de defectos. Por lo tanto se utilizan distintos niveles de testing para revelar los distintos tipos de defectos.



Cuanto más abajo se esté en estos niveles, más permitido está que el tester sea el mismo desarrollador. Al ir subiendo, no corresponde que sea la misma persona.

Testing de unidad:

Los distintos **módulos del programa se testean separadamente** contra el diseño, que actúa como especificación del módulo. **Se enfoca en los defectos inyectados durante la codificación.** El objetivo es testear la lógica interna de los módulos.

Frecuentemente el mismo programador realiza el test de unidad. La fase de codificación se suele denominar también de “codificación y testing de unidad”.

Testing de integración:

Se enfoca en la **interacción de módulos de un subsistema**. Los módulos que ya fueron testeados unitariamente se combinan para formar subsistemas, los que son sujetos a testing de integración.

Los casos de tests deben generarse con el objeto de ejercitar de distinta manera la interacción entre los módulos. **Hace énfasis en el testing de las interfaces entre los módulos**. Se podría omitir si el sistema no es muy grande.

Testing del sistema:

El sistema de **software completo** es testeado. **Se enfoca en verificar si el software implementa los requerimientos.** Generalmente es la etapa final del testing antes de que el software sea entregado. Debería ser realizado por personal independiente, pero los defectos son eliminados por los desarrolladores. Es la fase de testing que consume más tiempo.

Testing de aceptación:

Se enfoca en verificar que el software satisfaga las necesidades del usuario. Generalmente se realiza por el usuario/cliente en el entorno del cliente y con datos reales, pero los defectos son eliminados por los desarrolladores. Sólo después de que el testing de aceptación resulte satisfactorio, el software es puesto en ejecución (“deployed”). El plan del test de aceptación se basa en el criterio del test de aceptación y la SRS.

Otras formas de testing:

- **Testing de desempeño:** Requiere de herramientas para medir el desempeño.
- **Testing de estrés (stress testing):** El sistema se sobrecarga 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. Los 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.

Ejecución de los casos de test y análisis

La ejecución de los casos de test puede requerir de la escritura de “drivers” o “stubs”. También se requerirán módulos extras para preparar el entorno acorde a las condiciones establecidas en la especificación del caso de test. Algunos test pueden automatizarse, otros necesitan ser manuales: En ambos casos se puede **preparar un documento separado respecto del procedimiento de test.**

Se realizan **reportes con resúmenes del test**:

- casos de test ejecutados,
- el esfuerzo,
- los defectos encontrados.

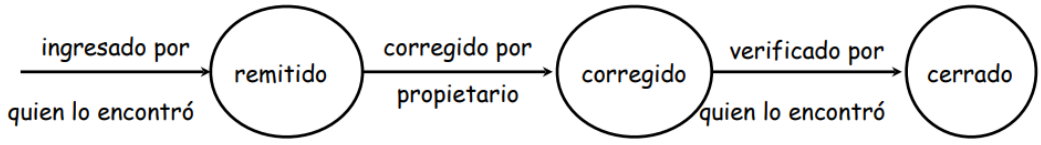
El seguimiento y control del esfuerzo del testing es importante para asegurar que se invirtió el tiempo suficiente. El tiempo de computadora también es indicador de cómo está procediendo el testing.

Registro de defectos y seguimiento

Un software grande puede tener muchos defectos, encontrados por personas distintas. Usualmente quienes los corrigen no son los mismos que lo encontraron. Debido a este gran alcance, el registro y la corrección de los defectos no puede ser informal.

Los defectos encontrados suelen registrarse en un **sistema seguidor de defectos** (“tracking”) que permite rastrearlos hasta que se “cierren”. El registro de defectos y su seguimiento es una de las mejores prácticas en la industria. Un defecto en un proyecto de software tiene su propio ciclo de vida; por ejemplo (hay otros más elaborados):

- Alguien lo **encuentra** en algún momento y lo **registra** junto con toda la información relevante (defecto remitido).
- Se **asigna la tarea de corrección**; la persona hace el debugging y lo corrige (defecto corregido).
- El administrador o quien lo remitió **verifica** que el defecto fue corregido (cerrado).



Durante el ciclo de vida, se registra información sobre el defecto en las distintas etapas para ayudar al debugging y al análisis.

Los defectos se categorizan generalmente en algunos tipos; los tipos de los defectos son registrados. Una posible clasificación es la denominada “*Orthogonal defect classification*”. Algunas categorías: funcional, lógica, estándares, asignación, interfaz de usuario, interfaz de componente, desempeño, documentación, etc.

También se registra la **severidad del defecto** en términos de su impacto en el software. La severidad es útil para priorizar la corrección. Una posible categorización:

- **Crítico**: puede demorar el proceso; afecta a muchos usuarios.
- **Mayor:** tiene mucho impacto pero posee soluciones provisorias; requiere de mucho esfuerzo para corregirlo, pero tiene menor impacto en el cronograma.
- **Menor:** defecto aislado que se manifiesta raramente y que tiene poco impacto.
- **Cosmético:** pequeños errores sin impacto en el funcionamiento correcto del sistema.

Idealmente, todos los defectos deben cerrarse. Algunas veces, las organizaciones entregan software con defectos conocidos (ej.: no hay defectos críticos o mayores, y menores < X). Las organizaciones tienen estándares para determinar cuándo un producto se puede entregar (o poner a la venta). El registro de defectos puede utilizarse para seguir la tendencia de cómo ocurren los arribos de los defectos y sus correcciones.