S5. Métodos para corregir errores del código

Sitio: Agencia de Habilidades para el Futuro
Curso: Metodología de Pruebas de Sistemas 2° D
Libro: S5. Métodos para corregir errores del código

Imprimido por: Eduardo Moreno
Día: lunes, 8 de septiembre de 2025, 00:04

Descripción

Tabla de contenidos

Introducción

La primera semana hablamos sobre los enfoques de *testing* que ayudarán con la verificación y validación del software. Estos enfoques nos ayudarán a ver que herramientas utilizaremos para encontrar los errores y defectos que queremos arreglar.

Enfoque estático

Recordemos la primera definición:

Con el enfoque estático buscamos errores o defectos sin la necesidad de ejecutar el sistema y se aplica a todas las etapas del proceso de desarrollo de un software. Consiste en:

- Realizar un análisis
- Comprobar de las representaciones del sistema con:
  - el documento de requerimientos,
  - los diagramas de diseño y;
  - el código fuente del programa.
- Usar herramientas para el análisis:
  - Crucible: permite una revisión del código colaborativa y el inicio de conversaciones en porciones específicas del código para dejar comentarios o anotaciones puntuales.
  - GitHub: contiene un analizador de código que permite hacer una comparación entre código anterior y código agregado recientemente.

Las estrategias del enfoque dinámico, caja negra y caja blanca, ya vistas; y las técnicas del enfoque estático - refactorig, mocking, stubbing- que se verán esta semana, son pruebas humanas. Es decir que no son automáticas o basadas en computadoras.

Los métodos que utilicemos en cada enfoque permiten encontrar errores tempranos, disminuyendo el costo de corregir estos errores y aumentando la probabilidad de corregirlos correctamente.

Recorramos algunos de estos métodos.

Inspecciones y *Walkthroughs*

Los 3 métodos de pruebas estáticas humanas son:

- **Inspecciones de código**
- ***Walkthoughs***
- **Pruebas de escritorio**

Los tres métodos están orientados al código y pueden utilizarse en cualquier etapa de desarrollo de un software.

Comencemos con los dos primeros: inspecciones de código y *Walkthoughs*.

¿En qué se parecen? Tanto el método de inspecciones y el método *walkthough* involucran un equipo de personas leyendo o inspeccionando visualmente un programa. Con ambos métodos, los/as participantes deben realizar trabajo previo para las reuniones. El punto cúlmine de ambos métodos es un "encuentro de mentes" en una reunión donde estén todos los involucrados. El objetivo de estas reuniones es encontrar errores, pero no encontrar las soluciones. Estos métodos sirven para probar y no para depurar. Además son ampliamente usadas, y es debido a los principios de *testing* analizados en la semana 1.

En un *walkthough* y una inspección

- un grupo de desarrolladores (entre 3 y 4 es un número óptimo) hacen la revisión.
- solo uno de los participantes es el autor del programa, por lo que la mayoría de los participantes de esta forma de *testing* no deben haber estado involucrados con el desarrollo.
- son mejoras a los procesos de pruebas de escritorios, ya que en ese método es un/a programador/a leyendo su propio código antes de probarlo.

¿En qué se diferencian estos métodos son las pruebas de caja negra?

Cuando se encuentra un error se localiza precisamente en el código, a diferencia de las pruebas de caja negra donde sólo recibimos un resultado inesperado. Además, este proceso frecuentemente expone errores en masa, permitiendo que estos errores se corrijan también en masa después. En cambio, en las técnicas computarizadas o de caja negra solo exponen un síntoma del error. Estos errores son entonces detectados y corregidos de a uno.

Los métodos de pruebas estáticas humanas generalmente son efectivos encontrando entre un 30 a un 70 por ciento de los errores de programación o de lógica hechos típicamente por los desarrolladores. Sin embargo, no son efectivos detectando errores de diseño de alto nivel, como los errores que se hacen en el análisis de requerimientos del proceso.

Una crítica frecuente a estos métodos es que sólo encuentran los errores "fáciles" que son triviales de hallar con métodos automatizados; mientras que los errores más difíciles, oscuros y complicados no pueden ser encontrados por métodos manuales.

Es importante notar que, en general, los dos tipos de técnicas (manuales o automatizadas) tienen sus tipos de errores favorecidos, es decir que los métodos manuales son mejores para encontrar algunos tipos de errores y los métodos automáticos son mejores para otros tipos de errores.

Lo que se intenta explicar, es que los métodos manuales y los métodos automáticos son complementarios. La eficiencia de detección de errores será menor si falta alguno de los dos métodos.

Finalmente, a pesar que estos procesos manuales son invaluables para probar nuevos programas, son de igual o mayor valor al probar modificaciones a programas ya existentes . Modificar un programa existente es un proceso más dado a errores que escribir un programa nuevo totalmente desde cero. Esto quiere decir que las modificaciones a programas deben ser sujetas a estos procesos de una forma más rigurosa, así como también utilizar prueba de regresión.

Inspecciones de código

Una inspección de código es un conjunto de procedimientos y técnicas de detección de errores para leer y realizar en grupo. La mayoría de las discusiones alrededor de esta técnica se enfocan en los procedimientos, formularios a llenar y otras formalidades.

Esta semana nosotros nos enfocaremos en la detección de errores luego de una breve introducción al procedimiento general.

> Es importante notar que las inspecciones de código son un método formal, es decir que tiene varias formalidades involucradas y su resultado será un documento con todos los errores encontrados.

Equipo para la inspección del código

Un equipo de inspección normalmente participantes 4 personas que tienen roles y tareas específicas.

Primer miembro del equipo: Moderador/a
En este contexto sería equivalente al/la ingeniero/a de control de calidad. Debe ser un/a programador/a competente, pero no debe ser el/la escritor/a del programa a analizar ni a estar familiarizado con los detalles del programa. Las tareas del/la moderador/a incluyen:

- **Distribuir** los materiales para la sesión de inspección.
- **Planificar** la fecha de la sesión de inspección.
- **Liderar** la sesión de inspección.
- **Registrar** todos los errores encontrados.
- **Asegurarse** que los errores encontrados sean corregidos.

2do miembro del equipo: Programador/a

Es quien trabajó en el código a inspeccionar. Algunas de las tareas incluyen:

- Preparar el código
- Explicar el código
- Tomar notas

3er miembro del equipo: Diseñador/a del programa (si es que es una persona diferente al/la programador/a). Desempeña un papel crítico en la inspección de código al garantizar que el código se adhiera al diseño original.

- Revisar el diseño original
- Comprobar la consistencia del diseño
- Colaborar en mejoras de diseño

4to miembro del equipo: Especialista de pruebas que debería ser experto/a en *testing* de software y estar familiarizado/a con los errores de programación más comunes.

- Identificar errores comunes
- Revisar los informes de pruebas
- Realizar pruebas adicionales
- Evaluar la estructura del código
- Comprobar la documentación
- Generar informes claros

Discutiremos cuáles son los errores más comunes más adelante.

Sesiones de una reunión de inspección

Planificación de la inspección de código:

Varios días antes de la sesión de inspección, el moderador distribuye las especificaciones de diseño del programa a todos los participantes. Cada miembro del equipo debe revisar y comprender el material antes del encuentro.

Actividades durante la sesión:

El moderador guía la sesión, asegurando que se mantenga enfocada en la búsqueda de errores y no en su corrección. Cualquier error identificado se corregirá posteriormente por el programador.

- **Narración del programa:** El programador presenta el código, explicando su lógica línea por línea. Los participantes pueden plantear preguntas y dudas durante esta narración. El objetivo es identificar posibles errores y aclararlos en tiempo real. A menudo, el propio programador identifica la mayoría de los errores durante esta etapa.
- **Revisión con un checklist:** El programa es sometido a una revisión utilizando un checklist que aborda errores comunes de programación. Esta lista será discutida en detalle en futuras sesiones.

Después de la sesión:

Una vez finalizada la inspección, el programador recibe una lista de errores detectados. Si se encuentran muchos errores o correcciones complejas, se puede programar otra sesión de inspección para verificar las correcciones.

La lista de errores es analizada y categorizada según el checklist, lo que ayuda a refinar futuras inspecciones.

Duración de la sesión:

La inspección requiere un ambiente sin interrupciones externas. La duración óptima de una sesión es de 90 a 120 minutos, ya que es mentalmente exigente. Las sesiones más largas suelen ser menos productivas debido a la fatiga. La velocidad de revisión es de aproximadamente 150 líneas por hora.

Para programas extensos, se pueden realizar múltiples sesiones, cada una tratando módulos relacionados entre sí.

Actitud constructiva en las reuniones

Es importante notar que para que este proceso de inspección sea efectivo, el equipo debe adoptar la actitud apropiada.

Actitud defensiva: Si el/la desarrollador/a ve la inspección como un ataque personal y adopta una posición defensiva, el proceso no será efectivo.

Actitud proactiva: Es cuando el equipo tiene en cuenta que el objetivo de la inspección es encontrar errores en el código, para mejorar la calidad del código.

Es por este motivo que muchas personas sugieren que los resultados de la inspección sean **confidenciales** y solo sean compartidas entre los/as participantes de la sesión. Si los/as jefes/as de proyectos y/o organizaciones utilizan estos resultados para otra cosa (como por ejemplo asumir que un/a desarrollador/a es incompetente o ineficiente) el proceso pierde el objetivo y nadie querrá ser participe.

Fortalecer al equipo

El proceso de inspecciones tiene varios **beneficios** secundarios además del objetivo principal de encontrar errores.

En primer lugar, **e/la programador/a suele recibir retroalimentación muy positiva** respecto a su estilo de programación, elección de algoritmos y técnicas de programación. Los/as otros/as participantes van mimetizando sus estilos de trabajo al ser expuestos a los algoritmos de otros programadores y sus errores.

En segundo lugar, el proceso de inspección es una buena forma de identificar de manera temprana las secciones del programa más propensas a errores; ayudando a enfocar la atención de forma directa en estas secciones durante el análisis con métodos automáticos, siguiendo el principio 9 del proceso de *testing* que vimos la semana 1.

En resumen

reforzar un enfoque de equipo al proyecto inspeccionado en particular, así como también a todos los proyectos que involucran a los/as participantes de la inspección en general. Así mismo, favorece para que las relaciones entre los integrantes del equipo evolucionen a no tener rivalidades ni enemistades. De esta forma, se fomenta la cooperatividad para llevar al proceso de desarrollo de software con más eficiente y confiabilidad.

Checklist de errores para las inspecciones

Una parte importante del proceso de inspecciones es utilizar el *checklist* para examinar el programa buscando errores comunes.

Pero..., algunos *checklist* pueden incluir elementos que no se enfocan en la detección de errores específicos en el código. Un estilo de *checklist* con elementos incorrectos enfocados en cuestiones de estilo y sin abordar errores podrían ser:

- ¿Los comentarios son precisos y significativos?
- ¿Están los bloques de código alineados adecuadamente?
- ¿El código cumple con los requerimientos y especificaciones de diseño?
- ¿Las variables tienen nombres descriptivos?
- ¿Se siguen las convenciones de estilo de codificación?

Estos elementos no están diseñados para identificar errores específicos, sino más bien para abordar cuestiones de estilo y buenas prácticas de codificación.

> Un checklist efectivo debe centrarse en aspectos como la verificación de límites, la validación de entradas, el manejo de excepciones y otros problemas que pueden causar errores en el software.

El *checklist* que presentamos en esta sección está dividido en 8 categorías y fue compilado por **Glenford Myers** (autor del libro “El arte de probar software”) a lo largo de muchos años del estudio de errores de software. Es mayormente independiente del lenguaje, lo que quiere decir que la mayoría de los errores pueden ocurrir en cualquier lenguaje de programación.

Errores de referencia de variables/datos

- ¿Una variable referenciada tiene un valor no inicializado o sin asignar? Este problema es el error de programación más frecuente que ocurre en una amplia gama de circunstancias. Para cada referencia a un ítem de datos (Variable, array, elemento, campo, etc.) debemos intentar “probar” informalmente que ese ítem tiene un valor asignado en ese punto.

  Por ejemplo, si tenemos el vector de estudiantes “e” y vamos a utilizarlo para mostrar la información de los estudiantes, el vector “e” debe tener al menos un estudiante cuando se ejecute el comando “e.show()”.

- Para cada referencia a un array, ¿Cada valor que llamamos está dentro de los límites definidos por su correspondiente dimensión?

  Es decir, si tenemos un array de 5 posiciones debemos asegurarnos que en ningún momento se intente usar la posición 6. Además, si tenemos un vector unidimensional no debemos buscar la posición [5][6] (Como si lo haríamos en una matriz).

- Para cada referencia a un array, ¿Todas las referencias utilizan valores enteros? Esto no es necesariamente un error en todos los lenguajes, pero en general, es una práctica peligrosa trabajar con referencias no enteras a vectores.

  Esto quiere decir que debemos verificar que en ningún lado se llame a la posición [3.5] del vector.

- Para todas las referencias a través de un puntero o variables referenciales, ¿La porción de memoria referenciada tiene información alocada en el momento de usarla? Esto se conoce como el problema de la “referencia colgada”. Ocurre en situaciones donde la vida de un puntero es mayor a la vida de la memoria referenciada.

  Una instancia de esto ocurre cuando un puntero referencia a una variable local dentro de un procedimiento, el valor del puntero es asignado a una variable de salida o una variable global, el procedimiento tiene un retorno (liberando la variable local) y luego el programa intenta usar el valor del puntero. Debemos intentar comprobar que el valor referenciado existe.

- ¿El valor de una variable tiene un tipo de valor distinto al que espera el compilador?
- Si tenemos un puntero o variables referenciales, ¿La posición de memoria tiene los atributos e información que el compilador espera?
- Si una estructura es referencia en múltiples procedimientos o subrutinas, ¿Esa estructura está definida de la misma forma en cada procedimiento?
- Para lenguajes orientados a objetos, ¿Se cumplen todos los requerimientos que tiene la herencia en la clase hijo?

Errores de declaración de datos

- **¿Todas las variables han sido explícitamente declaradas?** No hacerlo no es necesariamente un error, pero es una fuente normal de problemas. Por ejemplo, si una subrutina del programa recibe un parámetro que es un array y no define de forma explícita que el parámetro es un array, una referencia al array puede ser interpretada como una llamada a una función, llevando a que la máquina intente ejecutar el array como un programa. Además, si la variable no está explícitamente declarada en un procedimiento o un bloque, se asume que la variable es compartida con el resto del programa.

- Si los atributos de una clase no están explícitamente declarados, **¿Se entiende cuales son los atributos por defecto?**

  Por ejemplo, en **Java** los atributos y sus valores por defecto suelen ser una sorpresa cuando no están bien declarados.

- Cuando una variable es inicializada en una sentencia declarativa, **¿Está bien inicializada?** En muchos lenguajes, la inicialización de *arrays* y *strings* es algo un poco complicado, y por lo tanto propenso a errores.

- **¿A cada variable le hemos asignado la longitud y tipo correcto?**

- **¿Es la inicialización de la variable consistente con el tipo de memoria?** Por ejemplo, si a una variable en una subrutina de **Fortran** le hace falta ser re-inicializada cada vez que se llame a la subrutina, debe ser inicializada con una asignación y no con la sentencia DATA.

- **¿Hay varias variables con nombres similares (Por ejemplo, ruta y rutas)?** Esto no es necesariamente un error, pero debe ser visto como una advertencia. Estos nombres pueden ser confundidos muy fácilmente, por lo que deben ser doblemente revisados en el resto del programa. Otra cuestión a considerar es que nombres tan similares pueden ser considerados como mala práctica y ser reportados como errores o mejoras.

Errores de cálculo

- **¿Hay algún cálculo usando variables con tipos de datos inconsistentes?** Esto quiere decir que debemos verificar que no estemos sumando un valor numérico a un valor no numérico, por ejemplo.
- **¿Hay algún tipo de cálculo mixto?** Un ejemplo de esto es cuando estamos trabajando con variables de coma flotante (*Float*) y variables enteras. Algo como esto no necesariamente causa un error, pero debe ser explorado con cuidado para asegurar que se sigan todas las reglas de conversión del lenguaje. Consideremos el siguiente código en Java mostrando los errores de redondeo que pueden ocurrir cuando trabajamos con enteros:

```
int x = 1;

int y = 2;

int z = 0;

z = x/y;

System.out.println ("z = " + z);

OUTPUT

Z = 0
```

- **¿Hay algún tipo de cálculo usando variables que tienen el mismo tipo de dato, pero de diferentes longitudes?** Esto puede ocurrir cuando hacemos cálculos entre *arrays* o entre números de tipo *float* con distinta cantidad de decimales.
- **¿Es el tipo de dato de una variable destino de una asignación más pequeño que el tipo de dato de la expresión correspondiente?** Por ejemplo, en Java el tipo de dato int utiliza menos bytes de memoria que el tipo de dato long, por lo que puede guardar números más pequeños. Una operación con números de tipo *long* cuyo resultado se guarde en una variable de tipo int no necesariamente será un error, pero puede ocurrir que se desborde la variable resultado si el número es demasiado grande.
- **¿Es posible un *overflow* de alguna variable como resultado de algún cálculo?** Eso quiere decir que el resultado puede parecer válido pero algún resultado puede ser muy grande o pequeño para el tipo de dato. Esto es similar al ítem anterior, pero esto puede ocurrir incluso si los tipos de datos se corresponden. Una variable de tipo *int* puede almacenar como número máximo el 2147483647, si a este número le sumamos 1 la variable no tendría como valor 2147483648, sino -2147483648. Esto es debido a un *overflow*.
- **¿Es posible que el divisor de una división sea cero?** Una división por cero lanzará una excepción que causará una falla mortal en el código, debemos reportar estos casos para que se controlen y validen.
- **Cuando aplique, ¿es posible que el valor de una variable se vaya fuera de un rango significativo?** Por ejemplo, si estamos calculando probabilidades debemos revisar que el valor sea mayor a 0 y menor a 1 siempre, si existe la posibilidad que eso no ocurra debemos marcarlo como algo a arreglar.
- Para expresiones que contienen más de un operador, **¿Estas correctas las asunciones sobre el orden de evaluación y precedencia de los operadores?**
- **¿Hay algún uso invalido de la aritmética de enteros, particularmente de divisiones?** Por ejemplo, si i es una variable de tipo *int*. La expresión `2*i/2==i` depende de si i es un número par o impar y de si se evalúa primero la multiplicación o la división.

Errores de comparación

- ¿Hay alguna comparación entre variables que tienen diferentes tipos de datos como por ejemplo un *string* con una fecha o número?
- ¿Hay alguna comparación mixta o comparaciones entre variables de diferente largo? Si las hay, es importante asegurarnos que las reglas de conversión han sido bien entendidas.
- ¿Son correctos los operadores de comparación? Los programadores frecuentemente confunden las relaciones de “es más grande que”, “ es menor que”, “es al menos”, “es como máximo”, “ diferente a”, “menor o igual”, etc. Especialmente cuando hablamos de comparaciones entre fechas.
- ¿Cada expresión booleana expresa lo que debe expresar? Los programadores frecuentemente cometen errores con las expresiones lógicas que involucran los comandos *or* y *and*.
- ¿Son los operandos de un operador booleano efectivamente booleanos? Este representa un error frecuente entre los programadores. Algunos ejemplos típicos de errores se ilustran a continuación:

  - Si queremos determinar si i está entre 2 y 10 la expresión “2<i<10 ” no es correcta, deberíamos usar la expresión “ (2<i)&&(i<10) ”.
  - Si queremos determinar si i es mayor a x o y entonces la expresión “i>x||y ” no es correcto. Debe usarse la expresión “ (i>X)||(i>Y) ”.
  - Si queremos comparar que 3 números sean iguales entre sí, la expresión “(A==B==C) ” no es correcta. Debemos usar la expresión “(A==B)&&(A==C)”
  - Si queremos probar la relación matemática “x>y>z ” entonces la expresión correcta es “(x>y)&&(y>z)”.

- Para expresiones que contienen más de un operador booleano, ¿Las asunciones sobre el orden de evaluación y precedencia de operadores es correcta? Esto quiere decir, si vemos una expresión como “if ((a==2)&&(b==2) || (c==3))”, ¿se entiende sin dudas si el operador && o el operador || se evalúa primero?
- ¿La forma como el compilador evalúa las expresiones booleanas afecta el programa? Por ejemplo, la decisión “if (x==0 && (x/y) > z) ” puede ser aceptable para los compiladores que hacen cortocircuito si la primera condición de un AND es falsa, pero puede causar una división por cero con otros compiladores.

Errores de control de flujo

- Si el programa contiene una rama con muchos caminos (como una sentencia GOTO o SWITCH), ¿Es posible que la variable de índice exceda el número de posibilidades de la rama? Por ejemplo, si tenemos la sentencia "GOTO(200,300,400),i", ¿la variable i siempre tendrá algunos de esos 3 valores o puede tener un valor diferente?
- ¿Todos los ciclos terminan en algún momento o puede generarse un bucle infinito? Es importante revisar con algún tipo de prueba informal si todos los ciclos terminan alguna vez.
- ¿El programa, módulo o subrutina podrá terminar en algún momento? En el caso del programa puede que eso no sea necesario, pero para todas las subrutinas debe haber un final alcanzable.
- ¿Es posible que, debido a las condiciones de entrada al mismo, un ciclo nunca se ejecute? Por ejemplo, si tuviéramos algunos de los siguientes ciclos:

```
for (i = x ; i <= z; i++) {
...
}
While (NOTFOUND) {
...
}
```

¿Qué pasa si x es mayor a z o si NOTFOUND es inicialmente falsa? ¿Es correcto que no se ejecuten estos bucles?

- Para un bucle controlado por iteraciones y condiciones booleanas (Por ejemplo, una búsqueda) ¿Cuáles son las consecuencias de que el bucle no obtenga su resultado esperado? Por ejemplo, consideremos el siguiente código:

```
DO i=1 to TABLESIZE WHILE (NOTFOUND)
```

¿En este caso, qué ocurre si la variable NOTFOUND nunca se vuelve falsa? ¿El bucle seguiría para siempre o tenemos otra condición de corte? ¿Es siquiera posible que NOTGOUND se mantenga en false para siempre?

- ¿Hay algún tipo de error dado por una iteración de más o de menos? Este es un error muy común en lenguajes que comienzan sus bucles en cero. Si se hace un bucle de 10 iteraciones, por lo general se olvida contar el 0 como un número. Por ejemplo, si quisiéramos crear código en Java para un bucle for que itere 10 veces, el siguiente código sería erróneo ya que haría 11 iteraciones:

```
for (int i = 0 ; i <= 10 ; i++) {
System.out.println(i);
}
```

La versión correcta que itere solo 10 veces sería

```
for (int i = 0 ; i < 10 ; i++) {
System.out.println(i);
}
```

- ¿Hay un corchete o llave de cierre para corchete o llave abierta previamente?
- ¿Hay alguna decisión no exhaustiva? Es decir, si para un input se espera los valores 1, 2 o 3, ¿el software debe asumir que el valor es 3 si no es 1 o 2? ¿Es esa asunción válida?

Errores de interfaces

- ¿El número de parámetros recibido por el módulo o función es igual al número de parámetros enviados cada vez que se llama a ese módulo o función? ¿El orden de esos parámetros es correcto?
- ¿El tipo y tamaño de cada parámetro recibido coincide con el tipo y tamaño de cada correspondiente argumento?
- ¿Las unidades de medida de cada parámetro coinciden con las unidades de medida correspondiente al parámetro esperado? Por ejemplo, si tengo una función que espera metros y yo ingreso kilómetros, o espera grados y yo ingreso radianes.
- ¿El número de parámetros enviados por este módulo o función a otro módulo o función coincide con el número esperado para ese módulo o función?
- ¿El tipo y tamaño de cada parámetro enviado coincide con el tipo y tamaño de cada correspondiente argumento?
- Si se invoca a funciones propias de lenguaje, ¿el número, tipo y orden de los parámetros está correcto?
- Si un módulo o clase tiene muchos puntos de entrada, ¿Se referencia algún parámetro que no está asociado con el punto actual de entrada? Un error como este puede verse en el siguiente código de PL/1:

```
A:      PROCEDURE (W,X);
W=X+1;
REUTRN
B:      ENTRY (Y;Z);
Y=X+Z;
END;
```

- ¿Una subrutina altera un parámetro que debería ser solo una variable de entrada?
- Si hay presente alguna variable global, ¿Tiene la misma definición en todos los módulos que la referencian?
- ¿Se envían constantes por parámetro?

Errores de entrada/salida

- Si se declara de forma explícita un archivo a importar, ¿Sus atributos son correctos?
- ¿Son correctos los atributos de la sentencia OPEN de un archivo?
- ¿La especificación de formato en los requerimientos está de acuerdo con la sentencia de I/O? ¿La especificación del formato de archivo en el código se coindice con la sentencia de I/O? Distintos lenguajes pueden tener distintas sentencias para abrir distintos tipos de archivos.
- ¿Hay suficiente memoria alocada y disponible para contener el archivo que nuestro programa quiere leer? ¿Se han abierto archivos antes que este?
- ¿Se han cerrado todos los archivos que se han abierto una vez finalizado su uso?
- ¿Se detectan y manejan correctamente las condiciones de fin de archivo?
- ¿Se manejan correctamente los errores y excepciones de I/O?
- ¿Hay errores ortográficos o gramaticales en el texto que imprime o muestra el programa?
- ¿El programa maneja correctamente los errores del tipo "*File not Found*"?

Otros errores

- Si el compilador produce una referencia cruzada de identificadores, debemos examinarlo para ver si hay variables que no se referencian nunca o se referencian sólo una vez.
- Si el compilador produce un listado de atributos de una variable, debemos revisarlos para ver que no haya ningún atributo por defecto inesperado asignado.
- Si el programa compila exitosamente, pero la computadora produce uno o más mensajes de tipo "*warning*" o "*informational*", debemos revisar cada uno con mucho cuidado. Las "*warnings*" o advertencias son indicaciones que el compilador sospecha que estamos haciendo algo de calidad cuestionable. Los mensajes de tipo "*informational*" o informacionales pueden listar variables no declaradas, no usadas o lenguaje usado que impide la optimización del código.
- ¿Es el programa o módulo lo suficientemente robusto? Es decir, ¿revisa cualquier entrada para confirmar la validez de los datos?
- ¿Falta alguna función del programa?

Método 2: *Walkthroughs*

El segundo método de prueba estática humana y que está orientada al código se denomina: *walkthroughs.* Tiene mucho en común con el otro método de prueba que denominamos inspecciones, como ya mencionamos esta semana, pero los procedimientos son ligeramente diferentes y se emplea una técnica diferente de detección de errores.

Como en las inspecciones, los *walkthrough* consisten en una reunión ininterrumpida de entre una a dos horas de duración.

El equipo de *walkthrough* consiste entre 3 a 5 personas y cada uno/a tiene su propio rol.

- El/la moderador/a: cumple un rol similar al proceso de inspección.
- Secretario/a : anota todos los errores encontrados.
- *Tester:* examina y analiza el código bajo revisión para identificar errores.

Por supuesto que el/la programador/a es uno de los participantes, pero algunas sugerencias para participar son:

- **Un programador altamente experimentado.**
- Un experto en el lenguaje de desarrollo.
- Un nuevo programador, para dar una mirada fresca e imparcial.
- La persona que eventualmente tendrá que mantener o arreglar el programa.
- Alguien de un proyecto diferente.
- Alguien del mismo equipo de desarrollo que el programador original.

Sesiones de un reunión de Walkthroughs

En la mayoría de los *walkthrough* , se descubren más errores a través del proceso de cuestionamiento del programador que a través de la ejecución mental de los casos de prueba.

Planificación de la inspección de código

Los participantes reciben el material con varios días de anticipación, lo que les da tiempo para leer y re-leer el programa.

Actividades durante la sesión

En lugar de limitarse a leer el programa o seguir una lista de verificación de errores, los participantes en un *walkthrough* se sumergen en la experiencia de "actuar como computadoras".

El tester asignado llega a la reunión con un conjunto de casos de prueba impresos en papel. Estos casos de prueba contienen entradas representativas y las salidas esperadas para el programa o módulo que se va a analizar. Durante la sesión, cada caso de prueba se ejecuta mentalmente, es decir, los participantes siguen el flujo de lógica del programa en sus mentes.

Mantienen un seguimiento del estado del programa, incluido el valor de las variables, utilizando papel o pizarrón.

Características clave de los casos de prueba:

1. Deben ser relativamente simples para que todos los participantes puedan seguir el proceso mental.
2. Deben ser relativamente escasos, ya que los humanos ejecutan programas significativamente más lentos que las máquinas.
3. Sirven como puntos de partida para analizar y cuestionar la lógica del programa y las suposiciones del programador.

Similar a las inspecciones, la actitud de los participantes es crucial. Los comentarios y observaciones deben centrarse en el programa y no en el programador. Los errores se consideran desafíos inherentes a la programación de software y no debilidades personales. La crítica debe ser dirigida hacia el error, pero la consideración debe ofrecerse a la persona.

En los walkthrough, es importante tener un proceso de seguimiento similar para garantizar que los errores identificados se aborden de manera adecuada.

En el *walkthrough* debemos tener un proceso de seguimiento similar al descrito por las inspecciones. Observamos en esta técnica efectos secundarios muy similares a los de las inspecciones también: **identificación de secciones propensas a errores, educación en errores, educación en técnicas de programación y educación en estilo de programación** .

Actitudes en las sesiones

Al igual que en las inspecciones, la actitud de los participantes es crítica.

Los comentarios deben estar dirigidos al programa y no al programador. En otras palabras, los errores no se consideran como debilidades de la persona que los cometió, se ven como algo inherente a la dificultad de programar software. Hay que ser duros con el error, pero suaves con la persona.

En el walkthrough debemos tener un proceso de seguimiento similar al descrito por las inspecciones. Observamos en esta técnica efectos secundarios muy similares a los de las inspecciones también:

- identificación de secciones propensas a errores,
- educación en errores,
- educación en técnicas de programación y
- educación en estilo de programación.

¡Una reunión en tensión!

Compartimos un breve relato de una reunión walkthrough donde la actitud de las personas son negativas. Al finalizar leé con atención las preguntas que permiten analizar el relato y pensar en situaciones de mejora para el equipo.

Un equipo en tensiones!

En una reunión de walkthrough, el ambiente estaba cargado de tensión y desconfianza. Los participantes, en lugar de colaborar para mejorar el código, parecían estar en una competencia por señalar defectos y errores.

El programador, visiblemente molesto, se mostraba defensivo ante cada comentario y crítica. En lugar de escuchar atentamente, se esforzaba por justificar cada decisión que había tomado en la codificación. Cada vez que se señalaba un problema, se percibía una actitud negativa, como si fuera un ataque personal.

El tester, por su parte, parecía disfrutar destacando los errores del código y, en lugar de proponer soluciones constructivas, se limitaba a señalar defectos con un tono crítico.

A medida que avanzaba la reunión, la tensión aumentaba. En lugar de buscar maneras de mejorar el código, los participantes se enfrascaban en discusiones y debates infructuosos. La falta de una actitud positiva y constructiva estaba socavando el propósito del walkthrough , que era identificar y abordar los problemas de manera colaborativa.

Esta reunión sirvió como un claro ejemplo de cómo una actitud negativa y defensiva puede obstaculizar el proceso de mejora del código y resaltar la importancia de fomentar un ambiente de trabajo colaborativo y constructivo en las sesiones de walkthrough .

Para analizar el relato de la reunión walkthrough con **actitud negativa**, podríamos hacer las siguientes preguntas:

- ¿Cuáles fueron los principales problemas de actitud y comportamiento detectados en la reunión?
- ¿Cómo influyeron estas actitudes negativas en la dinámica de la reunión y en el objetivo de mejora del código?
- ¿Qué efectos tuvo la defensividad del programador y la crítica constante del tester en la colaboración del equipo?
- ¿Se lograron identificar y abordar de manera efectiva los problemas del código en esta reunión?.
- ¿Cómo podrían haberse transformado esas actitudes negativas en una reunión positiva y constructiva?

Para transformar una reunión de walkthrough en una **experiencia más positiva**, podríamos considerar las siguientes preguntas:

- ¿Cómo podemos fomentar una atmósfera de colaboración y apoyo mutuo durante la reunión?
- ¿Cuál es la mejor manera de dar y recibir retroalimentación constructiva en lugar de crítica destructiva?
-¿Cómo podemos enfocarnos en identificar soluciones y mejoras en lugar de simplemente señalar errores?
- ¿Qué técnicas de comunicación pueden ayudar a mantener un tono positivo y respetuoso en la discusión?
- ¿Cuál es el papel del moderador en asegurarse de que la reunión se mantenga en un tono positivo y constructivo?

Estas preguntas pueden servir como guía para transformar una reunión de walkthrough en una experiencia más efectiva y colaborativa.

Una reunión constructiva

El ambiente en la reunión fue de colaboración, y todos estaban comprometidos en encontrar soluciones y aumentar la calidad del código. Las críticas se dirigieron al programa y sus aspectos técnicos, en lugar de atacar al programador.

Al final de la reunión, se identificaron áreas de mejora, pero el equipo se fue con una sensación de logro y un plan claro para realizar modificaciones beneficiosas en el código. La actitud positiva y la voluntad de aprender y colaborar contribuyeron a una reunión de walkthrough exitosa y efectiva.

En una reunión de walkthrough, los participantes se reunieron con una actitud positiva y constructiva. El programador presentó su código y, en lugar de ser defensivo, lo hizo con la disposición de aprender y mejorar. El tester y otros miembros del equipo abordaron la revisión del código con respeto y apertura, buscando identificar áreas de mejora en lugar de señalar errores de manera negativa.

Durante la sesión, los participantes se centraron en comprender la lógica y el diseño del programa. El tester presentó casos de prueba simples y representativos que ayudaron a ilustrar el funcionamiento del código. A medida que avanzaba la revisión, surgieron discusiones constructivas sobre posibles mejoras y alternativas.

El ambiente en la reunión fue de colaboración, y todos estaban comprometidos en encontrar soluciones y aumentar la calidad del código. Las críticas se dirigieron al programa y sus aspectos técnicos, en lugar de atacar al programador.

Al final de la reunión, se identificaron áreas de mejora, pero el equipo se fue con una sensación de logro y un plan claro para realizar modificaciones beneficiosas en el código. La actitud positiva y la voluntad de aprender y colaborar contribuyeron a una reunión de walkthrough exitosa y efectiva.

Para analizar el relato de la reunión walkthrough positiva:

- ¿Cómo se describió la actitud general de los participantes en la reunión?.
- ¿Cuál fue el enfoque principal de la revisión del código?
-¿Qué tipo de preguntas y comentarios se realizaron durante la sesión?
- ¿Cómo se abordaron los casos de prueba presentados por el tester?
- ¿Cómo se manejaron las discusiones sobre mejoras y alternativas?
- ¿Cuál fue el resultado general de la reunión y el estado de ánimo al final?

Para transformar el relato en una reunión walkthrough negativa, podrías plantear preguntas como:

- ¿Cómo podríamos cambiar la actitud de los participantes para que sea más negativa y defensiva?
- ¿Qué tipo de comentarios o críticas podrían utilizarse para desalentar la colaboración?
- ¿Qué actitudes o comportamientos podrían generar conflictos en lugar de discusiones constructivas?
- ¿Cómo podríamos enfocar la revisión del código de manera que resalte los errores en lugar de las áreas de mejora?
- ¿Qué estrategias podrían utilizarse para desalentar la disposición a aprender y mejorar?
- ¿Cómo podríamos crear un ambiente de desánimo en lugar de logro al final de la reunión?

Método 1 y 2: Sus diferencias

Las principales diferencias entre walkthrough e inspección de código son:

Nivel	Walkthrough	Inspección de código
Proceso	Revisan el código de manera más informal y se centran en entender y discutir el funcionamiento del programa. Se "leen" o "recorren" partes del código, y se enfatiza la comprensión.	Se sigue un proceso más formalizado y estructurado, con roles definidos, pasos específicos y un enfoque en la detección de errores.
Participantes	Los participantes suelen ser menos formales, y el programador que escribió el código puede estar presente para explicarlo. La actitud es más colaborativa.	En la inspección de código, se requiere un equipo que incluye a un moderador, el autor del código, un revisor principal y otros revisores. La revisión se lleva a cabo bajo la guía del moderador.
Objetivo principal	Entender el código, revisar el diseño y la lógica, y promover la colaboración entre los miembros del equipo.	Detectar errores, defectos y problemas en el código. La revisión se enfoca en identificar posibles mejoras y correcciones.
Técnica de revisión	Los participantes suelen "jugar a ser computadoras", ejecutando casos de prueba mentalmente y siguiendo el flujo del programa. La discusión puede ser más abierta y orientada hacia la comprensión.	Los revisores siguen un checklist de errores específico y se centran en encontrar problemas siguiendo una metodología detallada.
Comentarios y retroalimentación	Los comentarios pueden ser más informales y pueden incluir preguntas, sugerencias y discusiones sobre diseño. La retroalimentación es menos enfocada en la detección de errores.	Los comentarios se centran en errores y defectos específicos, y la retroalimentación es más estructurada y dirigida a la corrección de problemas.
Documentación	Puede haber menos énfasis en la documentación detallada, ya que el enfoque es la comprensión y la discusión.	Se espera que haya documentación detallada, como un checklist de errores, para guiar la revisión y documentar los problemas encontrados.

Para sintetizar podemos afirmar que tanto el walkthrough como la inspección de código son técnicas de revisión de código, pero difieren en su formalidad, estructura, objetivos y enfoques. La elección entre uno u otro depende de los objetivos de la revisión y la cultura de desarrollo de software de la organización.

Método 3: Pruebas de escritorio

El método, también humano denominado prueba de escritorio coincide con los métodos de **inspección o de *walkthrough*** por la búsqueda de detectar errores, pero el método 3 es realizado por una sola persona y no por un equipo. Además, es una de las prácticas mas antiguas de análisis de código.

El procedimiento es:

- Una persona lee el programa y lo somete a una revisión individual, utilizando una lista de posibles errores y realizando pruebas mentales. Esto nos recuerda a la actividad de pruebas de escritorio que realizamos en la semana 1.

Ventajas	Desventajas
• tienen los mejores resultados cuando la revisión la hace alguien diferente a la persona que escribió el código. Por ejemplo, 2 programadores podrían cambiar de programas y hacer pruebas de escritorio en el código del otro	• Para la mayoría de las personas, las pruebas de escritorio son poco productivas • Es un proceso completamente indisciplinado. • Va en contra del principio de testing 2, donde una persona que escribe el código no debería ser la misma que lo prueba. • Es que 4 pares de ojos ven más que solo 1, por lo que nos estamos perdiendo varios puntos de vista potencialmente importantes.

En resumen, las pruebas de escritorio **son mejores que no hacer nada para revisar el código, pero muchísimo menos efectivas** que el método de inspecciones o de *walkthrough*.

Lectura de código: evaluación entre-pares

El último proceso de revisión manual que veremos no está asociado exactamente a encontrar errores. Sin embargo, lo incluimos porque está relacionado a la idea de la lectura del código y puede resultar útil para medir que tan bien o mal está nuestro código. El propósito de este proceso es el de permitir que los programadores auto-evalúen sus habilidades de programación.

El objetivo de esta técnica es que los programadores tengan una forma de auto-evaluación.

Es una técnica que evalúa de forma anónima los programas en términos de:

- calidad,
- mantenibilidad,
- extensión,
- usabilidad y;
- claridad.

¿Cómo es el proceso de la técnica?

El proceso de revisión se lleva a cabo en varias etapas y participan personas que asumen algunos de los siguientes roles: administrador del proceso, desarrolladores, revisores. Así mismo, se usa un formulario para realizar las calificaciones.

Etapa 1: Selección de participantes y programas

Comienza con la selección de un administrador del proceso, quien a su vez elige entre 6 y 20 participantes (con un mínimo de 6 para garantizar el anonimato). Los participantes deben tener habilidades y antecedentes similares, como todos siendo desarrolladores de Python, por ejemplo.

Cada participante selecciona 2 de sus programas para su revisión. Uno de los programas elegidos debe representar su mejor trabajo, mientras que el otro programa debe ser lo que considera como su trabajo de menor calidad.

Etapa 2: Distribución de los programas

Una vez que el administrador tiene todos los programas, los distribuye al azar entre los participantes. Cada revisor recibe 4 programas para evaluar, de los cuales 2 son los "mejores" y 2 son los "peores". Sin embargo, los revisores no saben cuál es cuál, ya que un programa puede ser revisado por más de una persona.

Etapa 3: Revisión con el formulario de evaluación.

Cada revisor dedica 30 minutos a evaluar cada programa y luego completa un formulario de evaluación.

Etapa 4: Calificación de los programas

Una vez que se han revisado todos los programas, cada participante califica la calidad relativa de los cuatro. El formulario de evaluación solicita a los revisores que respondan en una escala del 1 al 10 (donde 1 significa "Definitivamente sí" y 10 significa "Definitivamente no") una serie de preguntas, que pueden ser adicionales según lo determine el administrador.

Las posibles preguntas serían.

- ¿El programa fue fácil de comprender?
- ¿El diseño de alto nivel era visible y razonable?
- ¿El diseño de bajo nivel era visible y razonable?
- ¿Sería fácil para ti modificar este programa?
- ¿Estarías orgulloso si hubieras escrito este programa?

También se le piden comentarios generales y sugerencias de mejora al revisor.

Etapa 5: Recepción de la evaluación y calificación

Luego de la revisión, cada participante recibe la evaluación anónima para cada uno de sus 2 programas. También se les entrega un resumen estadístico, mostrando de forma detallada y general como quedaron sus programas con respecto a todos los programas entregados. Además de esto se les da un análisis de cómo las calificaciones que ellos dieron se comparan con las de otros revisores del mismo programa.