Preguntas de examenes viejos

Codificación

1. Explique la programación Estructurada (1)

La programación estructurada tiene como objetivo simplificar la estructura de los programas para que sea facil de razonar sobre ellos.

Un programa tiene
- **estructura estática:** Orden en que las sentencias son escritas en el código.
- **estructura dinámica:** Orden en que las sentencias son ejecutadas

El objetivo de la programación estructurada es que estas sean iguales.

Los constructor de la programación estructurada son de una única entrada y única salida.
inicia en un solo punto del código y finaliza en otro.

De esta manera se logra que la estructura estatica y dinamica sean iguales

Esta **simplifica** el flujo de control, facilitando en consecuencia tanto la **comprensión** de los programas así como el **razonamiento** sobre estos.

2. Describa el proceso de codificación: Desarrollo dirigido por test (3)

El programador primero escribe los test y luego el código suficiente para pasarlos

Esto ayuda a asegurar una cobertura completa en cuanto a test en todo el código.

entonces el procedimiento para escribir codigo es

1 Escribimos un script con test que cubran toda la funcionalidad de las especificaciones
2 Escribimos un código para pasar estos tests
3 Corremos el test
4 si hay errores los arreglamos y volvemos al punto 3
5 si pasa los test pero el código necesita mejoras, las hacemos y volvemos al punto 3
6 Si hay especificaciones sin cubrir volvemos al punto 1
(dibujar el diagrama)

3. ¿Que es la estructura estática de un programa? ¿ Cuál es la estructura dinámica de un programa? ¿ Cual es el objetivo de la programación estructurada? (1)

Estructura estática: Es el orden en que las sentencias del código están escritas.
Estructura dinámica: Es el orden en que las sentencias del código son ejecutadas.

El objetivo de la programación estructurada es simplificar la estructura de los programas para que sea fácil razonar sobre ellos, esto se logra haciendo que la estructura estática sea igual que la dinámica.

4. Describa los procesos de codificación: incremental, desarrollo dirigido por test y programación de a pares (2)

Programación Incremental:
1 Consiste en escribir un código que implemente una funcionalidad,
2 luego escribimos un script para testar esta nueva funcionalidad,
3 lo corremos y si hay error lo arreglamos y volvemos al punto 2
4 si lo pasa miramos si todas las especificaciones están cubiertas, si no lo están volvemos al punto 1.
5 terminamos
(dibujar el diagrama)

Desarrollo dirigido por test:
El programador primero escribe el test y luego el código suficiente para pasar el test.

La cobertura de toda la funcionalidad depende del diseño del test y no la codificación.

1 Escribimos un script con test que cubran toda la funcionalidad de las especificaciones
2 Escribimos un código para pasar estos tests
3 Corremos el test
4 si hay errores los arreglamos y volvemos al punto 3
5 si pasa los test pero el código necesita mejoras, las hacemos y volvemos al punto 3
6 Si hay especificaciones sin cubrir volvemos al punto 1
(dibujar el diagrama)

programación de a pares:
El código se escribe por dos programadores en lugar de uno solo:

Conjuntamente, ambos programadores diseñan los algoritmos, estructuras de datos, estrategias.
Una persona tipea el código, la ora revisa activamente el código que se tipea
Se señalan los errores y conjuntamente formulan soluciones
Los roles se alterna periódicamente

5. ¿Qué es y para qué sirve la refactorización de código? Explique completamente los tipos más comunes y menciona al menos un ejemplo para cada uno de ellos (3)

La **refactorización** es la tarea que permite realizar cambios en un programa con el fin de simplificarlo y mejorar su comprensión, sin cambiar el comportamiento observacional de este.

El objetivo es reducir el acoplamiento, incrementar la cohesión, mejorar la respuesta al principio abierto-cerrado

El principal riesgo de realizar una mejora al código, es que se puede romper la funcionalidad existente. Por eso es que debemos refactorizar en pequeños pasos y disponer de un script para test automatizado para identificar errores más fácilmente y facilitar la verificación de que el comportamiento externo se preserva.

Refactorización, tipos comunes:

Mejoras de métodos

Objetivo: Separar en métodos cortos cuya signatura indique lo que el método hace

Se aplica en los siguientes casos:

- **Métodos demasiado largos**
  - Divide un método largo que realiza múltiples tareas en varios métodos más pequeños, cada uno enfocado en una tarea específica.
- **Extracción de Código a Nuevos Métodos**
  - Si encuentras una sección de código que podría funcionar como un método independiente, extráela y conviértela en un nuevo método.
- **Transformación de Variables Locales en Parámetros**
  - Cuando crees nuevos métodos extrayendo código de un método original y utilizar variables locales del método original, asegúrate de pasar esas variables como parámetros al nuevo método.
- **Definición de Variables en el Método Original**
  - Si una variable se declara en una parte del código pero se utiliza en múltiples lugares del método original, debería definirse en ese método en lugar de partes específicas.
- **Separación de Lógica de Cambio de Estado**
  - Si un método realiza cambios en el estado del objeto y también devuelve un valor, considera dividirlo en dos métodos separados: uno para cambiar el estado y otro para retornar el valor.

Agregar o eliminar parámetro de un método:

. Agregar solo si los parámetros existentes no proveen información necesaria.
. Eliminar si los parámetros se agregaron originalmente “por las dudas” pero no se utilizan

Ejemplo: Un método que añade un usuario a la base de datos además chequea que la contraseña sea correcta. Esta acción de verificación deberá realizarse en un método distinto con parámetros derivados del primero.

Mejoras de clases

Desplazamiento de métodos:

- Mover un método de una clase a otra
- Se realiza cuando el método actúa demasiado con los objetos de la otra clase
- Inicialmente puede ser conveniente dejar un metodo en la clase inicial que delegue al nuevo (debería tener a desaparecer)

Desplazamiento de atributos:

- Si un atributo se usa más en otra clase, moverlo a esta clase
- Mejora cohesión y acoplamiento

Extracción de clases:

- Si una clase agrupa múltiples conceptos, separa cada concepto en una clase distinta
- Mejora la cohesión.

Reemplazar valores de datos por objetos:

- Algunas veces, una colección de atributos se transforma en una entidad lógica.
- Separarlos con una clase y definir objetos para accederlos.

ejemplos:

Si queremos crear un carrito de supermercado y tenemos una clase Productos con un método que calcular el precio total, es mejor desplazar ese método a la clase Carrito_de_compras y que calcule el precio total de todo el carrito., ya que este método está más relacionado con la clase carrito

Mejoras de jerarquías

Reemplazar condicionales con polimorfismos

- Si el comportamiento depende de algún indicador de tipo, no se está explotando el poder de la OO
- Reemplazar tal análisis de casos a tras de una jerarquía de clases apropiada

Subir métodos / atributos:

- Los elementos comunes deben pertenecer a la superclase.
- Si la funcionalidad o atributo está duplicado en las subclases, pueden subirse a la supercla

Ejemplo: si tenemos un clase Animal y un método Sonido que dependiendo de un tipo “perro” o “gato” devuelve “guau” o “miau” entonces no estamos aprovechando que podemos aprovechar que podemos usar el polimorfismo

para crear una jerarquía de clases más efectiva. Dónde Animal se convierta en una clase base, y clases derivadas como Perro y Gato hereden de Animal y definan su propio método Sonido.

6. ¿Cuáles son las refactorizaciones más comunes? Mencione al menos un ejemplo de cada una de ellas. (2)

Es lo mismo, osea hay que saberlo de memoria a esto importantisimo

Modelos de desarrollo

7. Describa exhaustivamente el proceso de desarrollo iterativo. (1)

El desarrollo iterativo consiste en desarrollar y entregar software **completo en sí mismo** incrementalmente.

El testing es dado en cada incremento, lo que facilita el mismo. Ya que es más fácil que testear todo el código completo.

Además contamos con feedback entre cada una de las iteraciones y puede utilizarse para los posteriores.

Modelo con mejora iterativa:
pasos:

1. implementación simple para un subconjunto del problema completo
2. Crear **lista de control del proyecto (LCP)** que contiene (en orden) las tareas que se deben realizar para lograr la implementación final. Cada una debe ser lo suficientemente simple como para comprenderla completamente.
3. Cada iteración consiste en eliminar la siguiente tarea de la lista haciendo **diseño, implementación y análisis** del sistema parcial, y actualizar la LCP.
4. Se repite hasta vaciar la lista.

Aplicación:

- Efectivo en desarrollo de productos donde los desarrolladores mismos proveen la especificación y los usuarios proveen feedback en cada release.
- Efectivo donde las empresas requieren respuestas rápidas, no se puede arriesgar el “todo o nada”

Beneficios:

- Pagos y entregas incrementales, feedback para mejorar lo desarrollado.

Inconvenientes:

- La arquitectura y el diseño pueden no ser óptimos;

- La revisión del trabajo hecho puede incrementarse;
- El costo total puede ser mayor

Aplicaciones

- Cuando el tiempo de respuesta es importante;
- Cuando no se puede tomar el riesgo de proyectos largos;
- Cuando no se conocen todos los requerimientos.

8. Describa exhaustivamente el proceso de desarrollo prototipado. (2)

Se construye un prototipo que permita **comprender los requerimientos**
Permite que el cliente tenga una idea de lo que sería el SW y así conseguir mejor feedback de él.

La etapa de análisis de requerimientos es reemplazada por una mini-cascada para crear el prototipo.

Esta incluye

. Análisis de Requerimientos
. Diseño
. Codificación
. Test

Para crear el prototipo

. Se modifican y crean nuevos requerimientos en base al prototipo y el feedback
. Diseño
. Codificación
. Test

para el proyecto final

Desarrollo del prototipo:

- **Solo incluye** las características claves que necesitan mejor comprensión
- **El cliente** provee feedback importante para la mejora de comprensión de los requerimientos.
- **Luego del feedback** el prototipo se modifica y se repite el proceso hasta que los costos y el tiempo superen los beneficios de este proceso
- **Teniendo en cuenta el feedback** los requerimientos iniciales se modifican para producir la especificación final de los requerimientos.
- **Finalmente** el prototipo se descarta

El costo del prototipo debe ser bajo:

La calidad no importa, solo poder desarrollar el prototipo rápidamente.
Omitir manejo de excepciones, recuperación, estándares.
Reducir testing

Ventajas:

Mayor estabilidad en los requerimientos
Los requerimientos se congelan más tarde

La experiencia en la construcción del prototipo ayuda al desarrollo principal

Desventajas:
Potencial impacto en costo tiempo

Aplicación:
Cuando los requerimientos son difíciles de determinar y la confianza en ellos es baja

Bue las demás preguntas sobre esto es explicar estos 4 modelos
Cascada, Prototipo, Iterativo, Timeboxing

Procesos de software

9. ¿Cual es la principal motivación para tener un proceso de desarrollo de software separado en fases? ¿ En qué consiste el enfoque ETVX? Describa brevemente las actividades básicas fundamentales de los proceso de desarrollo de software (1) pajon

Es necesario por la estrategia divide y vencerás: descomponer el problema en pequeñas partes permite que el proyecto sea manejable y comprensible más fácilmente

Cada parte ataca distintas partes del problema lo que permite validar continuamente el proyecto

Enfoque ETVX:
Cada fase sigue el enfoque ETVX (Entry-Task-Verification-Exit)

Criterio de entrada: Que condiciones deben cumplirse para iniciar la fase
Tarea: Lo que debe realizar esa fase
Verificación: Las inspecciones que deben realizarse a la salida de la fase
Criterio de salida: Cuando puede considerarse que la fase fue realizada exitosamente.

Además, cada fase produce información para la administración del proceso

Las fases son:
- **Análisis de requerimientos y especificaciones:**
  - Comprender precisamente el problema
  - **Salida:** especificación de los requerimientos del software
- **Análisis y Diseño:**
  - Involucra tres tareas:
    - **Diseño arquitectónico:** Componentes y correctores
    - **Diseño de alto nivel:** establece los módulos y estructuras de datos

Diseño detallado: lógica de los módulos

Salida: Documentos correspondientes

Codificación:

Implementar el diseño con código simple y fácil de comprender

Salida: el código

Testing:

Identificar la mayoría de los defectos y eliminarlos

Salida: Plan de test conjuntamente con los resultados, y el código final testeado

Instalación y entrega:

Entrega del producto final al cliente.

10. Describa cuales son las diferentes fases del proceso de administración del proyecto

La administración del proyecto: Se enfoca en el planeamiento y control del proceso de desarrollo con el fin de cumplir los objetivos.

Fases:

Planeamiento: Se realiza antes de comenzar el proyecto

Tareas claves:

Estimación de costos y tiempos
Seleccionar personal
Planear el seguimiento
…

Seguimiento y control: Acompaña al proceso de desarrollo

Tareas claves:

Seguir y observar parámetros claves como costo, tiempos, riesgo, asi como los factores que los afectan
Tomar acciones correctiva si es necesario
Las métricas proveen la información del proceso de desarrollo necesaria para el seguimiento.

Análisis de terminación:

Se realiza al finalizar el proceso de desarrollo.
El propósito fundamental es analizar el desempeño del proceso e identificar las lecciones aprendidas.
En procesos iterativos el análisis de terminación se realiza al finalizar cada iteración y se usa para mejorar en iteraciones siguientes.

Testing

11. ¿Qué es y para qué sirve el oráculo en test? ¿Qué son y para qué sirven los criterios de selección en tests? ¿Cuáles son las diferencias entre testing de caja negra y blanca? (3)

Oráculo:
Para verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos conocer el comportamiento correcto para ese caso i.e necesitamos un **oráculo de test.**

Para realizar el test tenemos que tener muy claro los casos a testear y su oráculo de test.
Al correr el test el resultado dado por el script y el resultado del oráculo se deben comparar con el fin de que sean iguales.

Idealmente pretendemos que el oráculo siempre de el resultado correcto.
El oráculo creado por un humano usa las especificaciones para predecir el “comportamiento correcto”.
En algunos casos el oráculo puede generarse automáticamente dependiendo de las especificaciones.

Criterio de selección:
El criterio de selección específica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o especificaciones. (no entra igual)

Existen dos enfoques para diseñar casos de test:
Caja negra y Caja blanca

Caja Negra:

No se utiliza el funcionamiento interno del software, se trata como una caja negra al cual le damos una entrada y nos entrega una salida esperada.

El comportamiento **está** especificado:
Para diseñar los casos de test, se utiliza el comportamiento esperado del sistema.
Se seleccionan casos de test sólo a partir de la especificación.

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

Caja Blanca:

El testing de caja blanca se enfoca en el código, el fin es ejecutar distintas estructuras del programa con el fin de cubrir errores.

Los casos de test se derivan a partir del código

12. ¿Qué es el testing de caja negra? Enuncie dos criterios dentro de esa categoría (3)

Idem con el anterior.

Criterios

Particionado por clase de equivalencia:
Particionar los datos de entrada en clases de equivalencia para las cuales se especifican distintos comportamientos.
Osea: la especificación requiere el mismo comportamiento en todos los elementos de una misma clase.

Dividir los datos de salida en clases de equivalencia. (Normalmente son eq. válidas y eq. inválidas) (comportamiento esperado del sistema)

Teniendo estas clases de salida generamos casos de test para cada una eligiendo apropiadamente las clases de entradas.
Dos métodos:
Estos deben cubrir tantas clases de entrada como sea posible,
O dar un caso de test que cubra a lo sumo una clase válida para cada entrada

Análisis de valores límites:
Particionar los datos de entrada en clases de equivalencia para las cuales se especifican distintos comportamientos.
Osea: la especificación requiere el mismo comportamiento en todos los elementos de una misma clase.

Dividir los datos de salida en clases de equivalencia. (Normalmente son eq. válidas y eq. inválidas). (comportamiento esperado del sistema)

Luego tomamos un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencia **entrada o salida.**

Para cada clase de equivalencia:
Elegir valores en los límites de la clase, los que están justo fuera y dentro de los límites

Con este conjunto más un valor normal se pueden producir casos de test que generen salidas esperadas sobre ellos.

Si tenemos varias variables con casos límites en cada una:
Dos formas:

1. Ejecutar todas las combinaciones posibles de las distintas variables.
    - Si tengo n variables y cada una tiene m casos límites entonces tengo m^n
2. Seleccionar los casos límites para una variable y mantener las otras en casos normales y considerar el caso de todo normal

- Si tengo n variables y cada una tiene m casos límites entonces tengo mn + 1

13. Explique el criterio de cobertura de sentencia , dando un ejemplo no mencionado en el teórico. (2)

Criterio de **Cobertura de sentencia:**
Este criterio está basado en el criterio de flujo de control, que es un criterio para seleccionar el conjunto de casos de test en testing estructural (Caja Blanca)

El criterio basado en flujo de control considera al programa como un grafo de flujo de control.
Los nodos representan bloques de código
Una artista (i, j), representa una posible transferencia de control del nodo i a j

Suponiendo la existencia de un nodo inicial y final, un camino es una secuencia del nodo inicial al nodo final

Entonces el criterio de cobertura de sentencia:
Cada sentencia se ejecuta al menos una vez durante el testing.
Osea el conjunto de caminos ejecutados durante el testing debe incluir todos los nodos

Limitación: Si el error está en la falta de sentencias, puede pasarse por desapercibido el error.

además no es posible garantizar 100% de cobertura debido a que puede haber nodos inalcanzables.

14. Explique y dé un ejemplo de testing de caja negra de análisis de valores límites (1)

idem explicación que 12

ejemplo:

Si tenemos datos de entrada en clases de equivalencia de la forma que 1<= n <= 6 y 3 <= m <=4

Los casos límite de n en esta clase son 0.9 y 6.1 fuera del rango y 1.1 y 5.9 dentro del rango.
y
Los casos límite de m en esta clase son 2.9 y 4.1 fuera del rango y 3.1 y 3.9 dentro del rango.

Y supongamos que los datos de salida en clases de equivalencia son válidas para n y m dentro del rango y invalido para n y m fuera del rango

por lo que n y m tiene 4 casos límites más un valor normal.

Por lo que la cantidad de test a realizar es:
método 1: $5^2 = 25$ tests

método 2: 5*2 + 1 = 11 tests

15. Explique y dé un ejemplo de testing de caja negra de particionado por clases de equivalencia (1)

Particionado por clases de equivalencia:

Es un método de selección para los casos de test en testing de caja negra

Se divide el espacio de entrada en clases de equivalencia para las cuales se especifican distintos comportamientos para cada una.

También se divide los datos de salida en clases de equivalencia con el valor esperado. Estos son los distintos comportamientos de cada clase de equivalencia de el espacio de entrada.

Luego elegimos los casos de test teniendo en cuenta estas equivalencias.
Método 1: Cada test cubre tantas clases de entrada como sea posible
Método 2: Cada test cubre a lo sumo una clase válida para cada entrada

Además de los casos de test separados por cada clase inválida

Ejemplo:
Si tenemos las variables n y m con n 6 datos de entrada que devuelven una equivalencia válida y 4 clase de equivalencia que devuelven una equivalencia invalida.
m 1 clase de equivalencia de entrada que devuelve una equivalencia válida y 1 clase de equivalencia que devuelve una equivalencia invalida.

Entonces con el método 1:
tenemos casó con n que cumple las 6 clases de entrada que devuelven una equivalencia válida y m que cumple con la clase de entrada que la valida y luego un test por cada clase inválida en n y m

En total tenemos 1 + 5 = 6 tests

Método 2: un test por cada clase válida de n (también clase valida de m) y un test por cada clase inválida en n y m

En total tenemos 6+5 = 12 tests

Práctico

Parte práctica

Ej. 6. Considere el siguiente fragmento de código:

```
1. int ordenado, i, j, aux;
2. int main()
3. {
4.     printf("Lista original:\n");
5.     for(i=0;i<4;i++){
6.         (i<3)?printf("%d,",lista[i]):printf("%d.",lista[i]);
7.     }
8.
9.     for(i=0; i<4;i++){
10.         for(j=0;j<3;j++){
11.             if(lista[j]>lista[j+1]){
12.                 aux= lista[j];
13.                 lista[j]= lista[j+1];
14.                 lista[j+1]= aux;
15.             }
16.         }
17.     }
18.
19.     printf("\nLista ordenada:\n");
20.     for(i=0;i<4;i++){
21.         (i<3)?printf("%d,",lista[i]):printf("%d.",lista[i]);
22.     }
23.     return 0;
24. }
```

(a) Construya el grafo de definición-uso etiquetando apropiadamente los conjuntos: def., uso-c y uso-p. Usar como referencia los número de líneas dados.

(b) Describa el criterios de coberturas de todas las definiciones. Construya un conjunto de casos de test para este código que cumpla con dicho criterio para la variable: lista[0].

Cajeta blanca: