Resumen 3 Diseño y realización de pruebas.

Objetivos:

Gestionar entornos de desarrollo adaptando su configuración en cada caso para permitir el desarrollo y despliegue de aplicaciones.
Desarrollar interfaces gráficos de usuario interactivos y con la usabilidad adecuada, empleando componentes visuales estándar o implementando componentes visuales específicos.
Realizar planes de pruebas verificando el funcionamiento de los componentes software desarrollados, según las especificaciones.
Desplegar y distribuir aplicaciones en distintos ámbitos de implantación verificando su comportamiento y realizando las modificaciones necesarias.

1 Introducción. Consiste en verificar (comprueba el funcionamiento correcto del software desde un punto de vista técnico) y validar (comprueba si los resultados de usuario se cumplen) un producto software antes de su puesta en marcha.

Un sistema puede pasar la validación, sin embargo, no pasa la verificación. Cumple con la especificación del usuario, con lo que él quería, cubre sus necesidades pero internamente puede adolecer de graves detalles como un precario diseño en la base de datos o bien usa un excesivo e innecesario número de líneas de código.

Se integra dentro de las diferentes fases del ciclo de vida del software dentro de la ingeniería de software.
La ejecución de pruebas de un sistema involucra una serie de etapas:

Planificación de pruebas.
Diseño y construcción de los casos de prueba.
Definición de los procedimientos de prueba.
Ejecución de las pruebas.
Registro de resultados obtenidos.
Registro de errores encontrados.
Depuración de los errores.
Informe de los resultados obtenidos.

2 Técnicas de diseño de casos de prueba.

Casos de prueba: conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para conseguir un objetivo particular o condición de prueba.

Se utilizan dos técnicas o enfoques: prueba de caja blanca (verifican la estructura interna del programa) y prueba de caja negra (validan los requisitos funcionales sin fijarse en el funcionamiento interno del programa).

1.

Pruebas de caja blanca (estructurales) se obtienen casos de prueba que:
Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
Ejecuten todas las sentencias al menos una vez.

Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
Ejecuten todos los bucles en sus límites.
Utilicen todas las estructuras de datos internas para asegurar su validez.

1. Pruebas de caja negra (funcionales o de comportamiento) se intenta encontrar errores de las siguientes categorías:
   Funcionalidades incorrectas o ausente

   Errores de interfaz.

   Errores en estructuras de datos o en accesos a bases de datos externas.

   Errores de rendimiento.

   Errores de inicialización y finalización.

3 Estrategias de prueba del software.

- 3.1 Prueba de unidad. Se prueba cada unidad del código, con el objetivo de eliminar errores en la interfaz y en la lógica interna. Pueden ser de interfaz, de estructuras de datos locales, de condiciones límites, de caminos independientes o de camino de manejo de errores.

- 3.2 Prueba de integración. Se prueba la interacción de los distintos módulos.

- 3.3 Prueba de validación. Se prueba que el software funcione según el documento ERS de Especificaciones de Requisitos de Software, de la fase de análisis. Hacemos pruebas de caja negra, que pueden ser Alfa, si se hacen en el lugar del desarrollo, o bien Beta si se hacen en el lugar del cliente.

- 3.4 Prueba del sistema. Se prueba el sistema en profundidad. Pueden ser pruebas de recuperación, de seguridad, o de resistencia.

4 Documentación de la prueba. Plan de pruebas, Especificaciones de prueba, Informes de pruebas.

5 Pruebas de código. Consiste en ejecutar parte del código con el objetivo de encontrar errores.

-5.1 Prueba del camino básico. Mide la complejidad lógica (Complejidad ciclomática) del diseño procedimental, y establece los distintos caminos básico de ejecución. Un camino básico de ejecución es cada uno de las posibles combinaciones de flujos.

Ejemplo: Programa que lee 10 números de teclado y muestra cuántos de los números son pares y cuántos son impares. Para comprobar si el número es par o impar utilizamos el operador % de Java, que devuelve 0 si es par. La estructura principal es un WHILE y dentro tiene un IF. Tienes a la izquierda el diagrama de flujo y a la derecha el grafo de flujo.

Camino	Nodos	Caso de Prueba	Resultado esperado
1	1-2-9	C=10	Visualizar el número de pares y el de impares
2	1-2-3-4-5-6-8-2-9	C=1, N=5	Contar números impares
3	1-2-3-4-5-7-8-2-9	C=1, N=4	Contar números pares

La complejidad ciclomática V(G) en este ejemplo es 3 porque hay 3 caminos básicos de ejecución.

-5.2 Partición o clases de equivalencia. Prueba de caja negra, que divide los valores de entrada en 2 o más clases de equivalencia. Cómo mínimo habrá 2 clases: la válida, y la no válida, y se definen el número de clases según la siguiente tabla:

Condiciones de entrada	de clases válidas	de clases no
1 Rango	npla los valores del	valor por encima del rango valor por del rango
2 Valor fico	npla dicho valor	valor por valor por
3 Miembro de un conjunto	clase por cada ros conjunto	Un valor que no pertenece al conjunto

4 Lógica	clase que cumpla condición	Una clase que no cumpla la condición

-5.3 Análisis de valores límite. Los errores tienden a producirse en los extremos de los valores de entrada. Es complementaria a la partición, y se escogen clases de equivalencia, por encima y por debajo de los valores límites de entrada y se exploran condiciones de salidas límites. Aquí tienes algunos ejemplos:

a) Si una condición de entrada especifica un rango de valores enteros entre 1 y 10, se crean casos de prueba para 1, 10, 0 y 11.

b) Si una condición de entrada especifica un número de valores específicos (ejemplo de 2 a 10 valores), se crean casos de prueba para los 2, 10, 1 y 11 valores de entradas.

c) Si la condición de salida es de descuento comprendido entre el 10 % y el 50 %, generar casos de prueba del 10%, 50%, 9,99% y 50,01%.

d) Si la condición de salida es una tabla de 1 a 10 valores específicos, diseñar casos de prueba que generen 0, 1, 10 u 11 valores.

e) Si las estructuras de datos internas tienen límites (array de 100 elementos), diseñar casos de prueba que use la estructura en sus límites, primer y último elemento.

6 Herramientas de depuración. El proceso de depuración comienza con la ejecución de un caso de prueba

tenemos 2 opciones, si se identifican las causas, se corrigen y si no, debemos de sospechar de causas posibles, que se confirmarán o no con nuevos casos de prueba.

Al desarrollar programas cometemos 2 tipos de errores, los de compilación y los lógicos. Los primeros los detecta el IDE al ir codificando, en cambio los segundos pueden devolver resultados inesperados y erroneos, que son conocidos como “bugs”.Para resolver los errores lógicos contamos con el depurador (debugger) del IDE.

- 6.1 Puntos de ruptura. Linea de nuestro código donde queremos que el código se detenga, para ver los valores de las variables en ese momento. Una vez detenido el código, tenemos varias opciones.

También se pueden crear puntos de ruptura condicionales, que se detendran si la variable tiene un valor concreto.

- 6.2 Examinadores de variables. Desde la vista “Inspección” y desde la pestaña “Variables”, podemos inspeccionar las variables en el punto donde se ha detenido el programa

7 Pruebas unitarias Junit. Junit es un framework para realizar pruebas unitarias automatizadas. Un framework son un conjunto de librerías, que te permitirán reutilizarlas y reducir el código que se tendría que desarrollar para un requerimiento dado. Está integrado en Eclipse. Las pruebas unitarias se realizan sobre una clase para probar su comportamiento de modo aislado independientemente del resto de clases de la aplicación, aunque a veces es necesario que esta clase dependa de otras.

- 7.1 Creación de una clase de prueba. En Eclipse, se crea la clase de prueba, añadiendo al pograma la “Junit Test Case”, se seleccionan los métodos que queremos probar, se añaden las librerías necesarias, y se guarda en un paquete diferente a la clase a probar. En la Práctica 3b lo harás con Eclipse.

  7.2 Preparación y ejecución. Los métodos de tipo void de Junit para hacer las comprobaciones son:

Un fallo es una comprobación que no se cumple, mientras que un error es una excepción durante la ejecución del código.

7.3 Tipo de anotaciones. Junit dispone de una serie de anotaciones que permiten ejecutar código antes y después de las pruebas:

@Before: si anotamos un método con esta etiqueta, el código será ejecutado antes de cualquier método de prueba. Este método se puede utilizar para inicializar datos. Por ejemplo en una aplicación de acceso a una BD, se preparan los datos de inicialización.

@After: el código será ejecutado después de todos los métodos de prueba. Se puede utilizar para limpiar datos..

@BeforeClass: solo puede haber un método con esta etiqueta y es invocado una vez al principio del lanzamiento de todas las pruebas. Se utilizan para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan mucho.

@AfterClass: solo puede haber un método con esta etiqueta y es invocado una vez al cuando finalicen todas las pruebas.
7.4 Pruebas parametrizadas. Son pruebas repetidas varias veces, cambiando los distintos valores de entrada. Para hacerlo tenemos que:

a) añadir la etiqueta @RunWith(Parameterized.class) a la clase de prueba. Se declara un atributo por cada uno de los parámetros de la prueba y un constructor con tantos argumentos como parámetros en cada prueba.

b) definir un método anotado con la etiqueta @Parameters, que será el encargado de devolver la lista de valores a probar. Por ejemplo si queremos probar la división exacta de 2 números, la lista de valores sería {num1, num2, resultado}, osea {10,2,5}.

7. Suite de pruebas.JUnit dispone de “Test Suites”, que consiste en una combinación de varias clases de prueba para que se ejecuten una tras otra
8. 8 Calidad del sotware. La calidad del software, trata los conceptos, los métodos, las técnicas, los procedimientos y los estándares necesarios para producir productos y procesos software de alta calidad. Para ello se usan herramientas que permiten medir, mediante métricas, los siguientes aspectos del código:

   1Reglas de codificación

   2Diseño y Arquitectura

   3Código duplicado

   4Complejidad del código

   5Pruebas unitarias

   6Errores potenciales

   7Comentarios en el código