

MÓDULO PROFESIONAL

ENTORNOS DE DESARROLLO

UD3: INTRODUCCIÓN A
LAS PRUEBAS DE
SOFTWARE Y
DEPURACIÓN

CONTENIDO

- Diseño y realización de pruebas
 - Prueba
 - Casos de prueba
 - Error, defecto y fallo
- Tipos de defectos
- Pruebas en el ciclo de vida del desarrollo del software
 - Diferencia entre verificación y validación
 - Pruebas unitarias
 - Pruebas de integración
 - Pruebas de sistema
 - Pruebas de aceptación
- Técnicas de diseño de casos de prueba
 - Caja blanca
 - Caja negra
- Ejemplos de casos de prueba y procedimientos de prueba
- Depuración de código fuente
- Frameworks para pruebas unitarias: JUnit 5
- Herramientas para la realización de pruebas
- Calidad del software

DISEÑO Y REALIZACIÓN DE PRUEBAS

Las **pruebas** son necesarias en la fabricación de cualquier producto industrial y, de forma análoga, en el desarrollo de proyectos informáticos.

¿Quién pondría a la venta una aspiradora sin estar seguro que aspira correctamente, o una radio digital sin haber comprobado que pueda sintonizar los canales?

DISEÑO Y REALIZACIÓN DE PRUEBAS

Una aplicación informática no puede llegar a las manos de un usuario final con **errores**, y menos si estos son suficientemente visibles y claros como para haber sido detectados por los desarrolladores.

Se daría una situación de falta de profesionalidad y disminuye la confianza por parte de los usuarios, que podría mermar oportunidades futuras.

DISEÑO Y REALIZACIÓN DE PRUEBAS

¿Cuándo hay que llevar a cabo las pruebas?

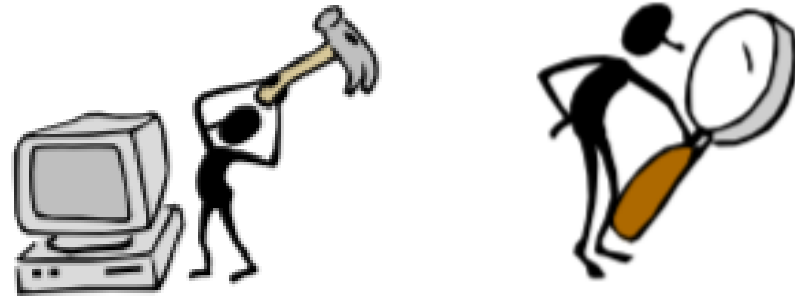
¿Qué hay que probar?

Todas las fases establecidas en el desarrollo del software son importantes. La falta o mala ejecución de alguna de ellas puede provocar que el resto del proyecto arrastre uno o varios errores que serán determinantes para su éxito. **Cuanto antes se detecte un error, menos costoso será de solucionar.**

PRUEBA. DEFINICIÓN

La prueba (testing) es el proceso de ejecutar un programa con la intención de encontrar errores.

[Glenford J. Myers]



PRUEBA. DEFINICIÓN

IEEE

IEEE es el acrónimo para Institute of Electrical and Electronics Engineers, una asociación profesional sin ánimo de lucro que determina la mayor parte de los estándares en las Ingenierías.

En la literatura existen otras definiciones. Por ejemplo, de acuerdo a la IEEE [IEEE90] el concepto de prueba se define como

*“El proceso de operar un sistema o un componente bajo unas **condiciones específicas**, observando o registrando los **resultados obtenidos** con el fin de realizar a posteriori una **evaluación** de ciertos aspectos clave”*

PRUEBA. DEFINICIÓN

Descubrir un **error es el éxito** de
una prueba

CASOS DE PRUEBA.

DEFINICIÓN

Con el fin de desarrollar las pruebas de una manera óptima, los **casos de prueba** cobran especial importancia.

La IEEE [IEEE90] define caso de prueba como:

*“Un **conjunto de entradas, condiciones de ejecución y resultados esperados** que son desarrollados con el fin de cumplir un determinado objetivo.”*

PRUEBAS

La prueba exhaustiva del software es **impracticable** (no se pueden probar todas las posibilidades de su funcionamiento ni siquiera en programas sencillos).

PRUEBAS

El programa se prueba ejecutando los casos de prueba posibles dado que por lo general es física, económica o técnicamente imposible ejecutarlo para todos los valores de entrada posibles, de aquí la frase de Dijkstra:

“Las pruebas del software pueden usarse para demostrar la existencia de errores, nunca su ausencia”

PRUEBAS

El principal objetivo de las pruebas es el de encontrar **errores** en el programa, tantos como sea posible.

Entorno al concepto de error existen distintos términos que poseen ciertos aspectos en común y a menudo inducen a confusión. Por ello, el estándar IEEE [IEEE90] establece tres conceptos:

- Error (error)
- Defecto (bug)
- Fallo (failure)

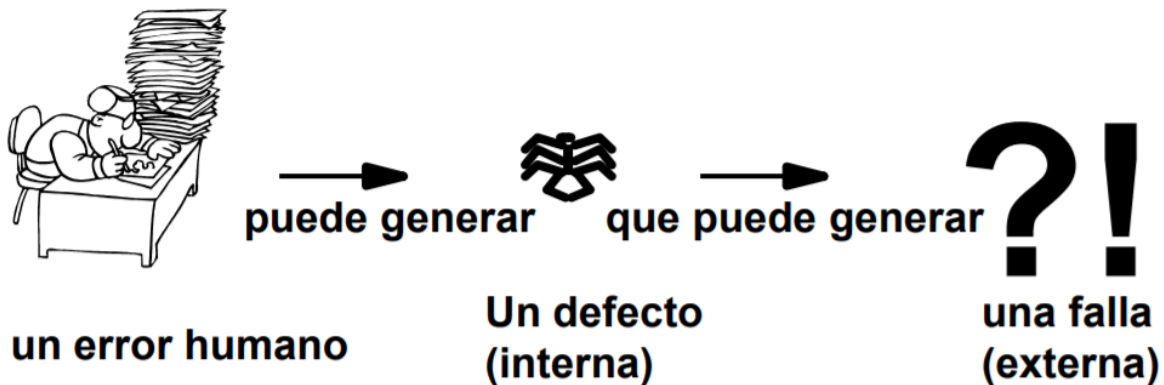
ERROR, DEFECTO, FALLO

- **Error (error):** Es una acción humana que produce un resultado incorrecto. Es una equivocación cometida por el desarrollador. Algunos ejemplos son: un error de código, una malinterpretación de un requerimiento o de la funcionalidad de un método.
- **Defecto (bug):** El defecto en un componente o sistema es la causa por el cual el sistema o componente no logra llevar a cabo su función específica. Un defecto potencialmente origina un fallo.
- **Fallo (failure):** Es la manifestación del defecto, ya sea física o funcional, que se produce al ejecutar un programa el cual es incapaz de funcionar correctamente.

RELACIÓN DEFECTO, FALLO, ERROR

Tenemos entonces la siguiente relación:

El **error** cometido inyecta un **defecto** que puede provocar un **fallo** en el sistema.



No todos los defectos corresponden a un fallo. Por ejemplo: si un trozo de código defectuoso nunca se ejecuta, el defecto nunca provocará el fallo del código.

RELACIÓN DEFECTO, FALLO, ERROR



ERROR

del programador



DEFECTO

en el software



FALLO

depende del sistema

ACTIVIDAD BUGS

1. Busca información sobre bugs: uno célebre en la historia y dos que sean más actuales.
 - Bug célebre: Escribe sobre sus consecuencias, costes, contexto...
 - Bug corriente: Escribe cómo detectarlos, cómo evitarlos...
- Encontrarás mucha más información en páginas en inglés.

TIPOS DE DEFECTOS

- En algoritmos
- De sintaxis
- De precisión o cálculo
- De documentación
- De estrés o sobrecarga
- De capacidad o desbordamiento
- De sincronización o coordinación
- De capacidad de procesamiento o desempeño
- De recuperación
- De estándares y/o procedimientos
- Relativos al hardware o software del sistema

TIPOS DE DEFECTOS

En algoritmos

- Condiciones equivocadas.
- No inicializar variables.
- No evaluar una condición particular.
- Comparar variables de tipos no adecuados.

De sintaxis

- Confundir un 0 por una O.
- Los compiladores detectan la mayoría.

TIPOS DE DEFECTOS

De precisión o de cálculo

- Fórmulas no implementadas correctamente.
- No entender el orden correcto de las operaciones.
- Falta de precisión como un truncamiento no esperado.

De documentación

- La documentación no es consistente con lo que hace el software. Ejemplo: El manual de usuario tiene un ejemplo que no funciona en el sistema.

TIPOS DE DEFECTOS

De estrés o sobrecarga

- Exceder el tamaño máximo de un área de almacenamiento intermedio.
- Ejemplos:
 - El sistema funciona bien con 100 usuarios pero no con 110.
 - Sistema que funciona bien al principio del día y se va degradando paulatinamente el desempeño hasta no ser funcional al caer la tarde. Defecto: había tareas que no liberaban memoria.

TIPOS DE DEFECTOS

De capacidad o desbordamiento.

- Más de lo que el sistema puede manejar.
- Ejemplos
 - El sistema funciona bien con importes < 1000000
 - Año 2000. sistema trata bien fechas hasta el 31/12/99

TIPOS DE DEFECTOS

De sincronización o coordinación

- No cumplir con un requerimiento de tiempo o frecuencia.
- Ejemplo: comunicación entre procesos con fallos.

De capacidad de procesamiento o desempeño

- No terminar el trabajo en el tiempo requerido.
- Tiempo de respuesta inadecuado.

De recuperación

- No poder volver a un estado normal después de un fallo.

TIPOS DE DEFECTOS

De estándares y/o procedimientos

- No cumplir con la definición de estándares y/o procedimientos.

Relativos al hardware o software del sistema

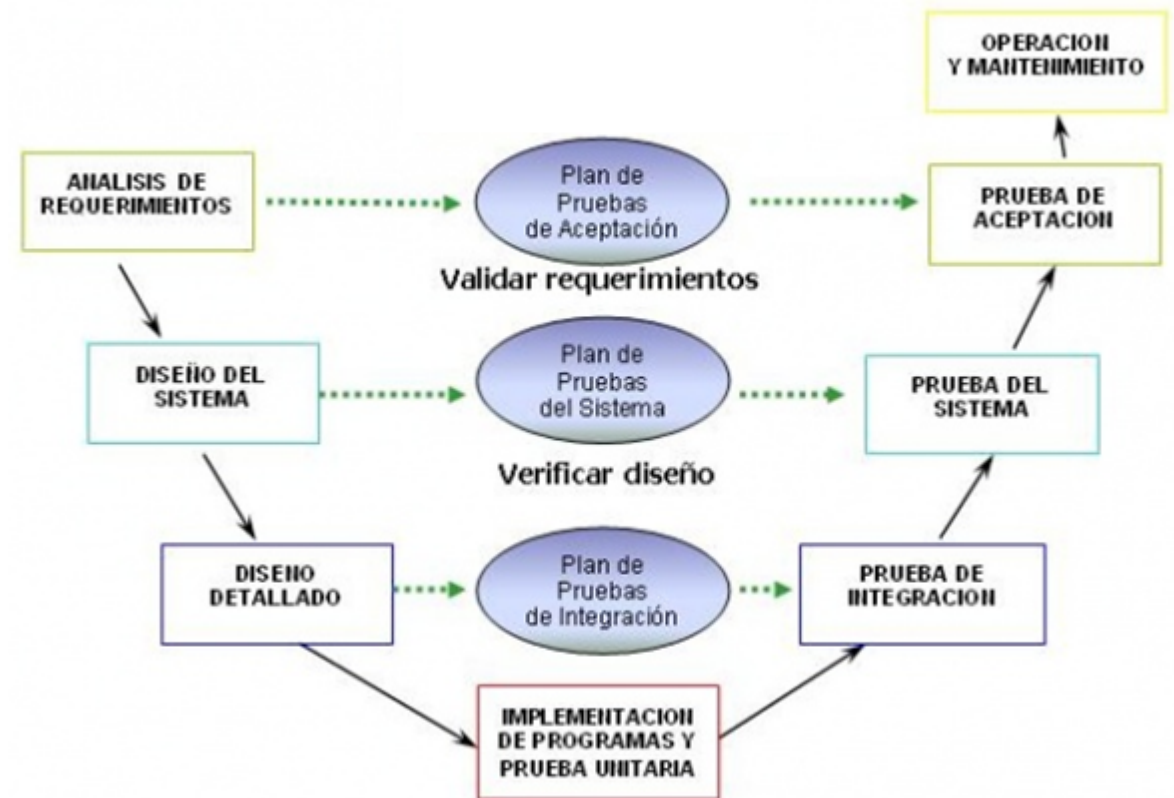
- Incompatibilidad entre componentes.

¿CÓMO ENCAJAN LAS PRUEBAS EN EL CICLO DE VIDA DE DESARROLLO DEL SOFTWARE?



En cada una de las fases del ciclo de vida de un proyecto, será necesario que el trabajo llevado a cabo sea **validado y verificado**.

Existe una correspondencia entre cada nivel de prueba y el trabajo realizado en cada etapa del desarrollo:



LAS PRUEBAS EN EL CICLO DE VIDA DE DESARROLLO DE SOFTWARE

¿Es lo mismo **verificar** que **validar**?

LAS PRUEBAS EN EL CICLO DE VIDA DE DESARROLLO DE SOFTWARE

La verificación y la validación, dentro del ámbito de desarrollo de software, no son la misma cosa, aunque es muy fácil confundirlas:

Verificación: *¿El software está de acuerdo con su especificación?* El papel de la verificación comprende comprobar que el software cumple los requisitos funcionales y no funcionales de su especificación.

Validación: *¿El software cumple las expectativas del cliente?* La validación es un proceso más general. Se debe asegurar que el software hace lo que el usuario espera.

LAS PRUEBAS EN EL CICLO DE VIDA DE DESARROLLO DE SOFTWARE

Existen muchos **tipos de pruebas** que deben cubrir las especificaciones de un proyecto informático a través de los procedimientos y de los casos de prueba.

Se presenta un **resumen** de estos tipos de pruebas (más adelante las veremos en detalle)

Pruebas unitarias: una prueba unitaria es la manera de comprobar el correcto funcionamiento de un componente de código (una parte de un programa). Esto nos permite asegurar que todos los componentes del sistema desarrollado funcionen correctamente por separado. Detectan errores en los datos, lógica y algoritmos. Participan los **programadores**.

Pruebas de integración: Una vez que se hayan probado que los componentes funcionan correctamente en forma aislada se procede a probar cómo funcionan integrados con otros. El objetivo es detectar errores de interfaces y relaciones entre componentes. Participan **programadores**.

LAS PRUEBAS EN EL CICLO DE VIDA DE DESARROLLO DE SOFTWARE

Pruebas de sistema: Esta prueba reproduce el sistema del propio cliente y tiene como objetivo verificar que se han integrado adecuadamente todos los elementos del sistema (hardware y software) y que realizan las funciones adecuadas. Permiten probar el sistema en su conjunto y con otros sistemas con los que se relaciona para verificar que las especificaciones funcionales y técnicas se cumplen. Participan **probadores y analistas.**

Pruebas de aceptación: El propósito es confirmar que el sistema está terminado, que desarrolla puntualmente las necesidades del cliente y que es aceptado por los usuarios finales. Participan **probadores, analistas y cliente.**

PLANIFICACIÓN DE LAS PRUEBAS

A la vez que se avanza en el desarrollo del software se van **planificando las pruebas** que se harán en cada fase del proyecto.

Esta planificación se concretará en un **plan de pruebas** que se aplicará a cada producto desarrollado.

Cuando se detectan errores en un producto se debe volver a la fase anterior para depurarlo y corregirlo.

PLANIFICACIÓN DE LAS PRUEBAS

Es importante tener presente que, cuanto **antes** se detecte un error, **más fácil** será contrarrestar y solucionar este error.

El coste de la resolución de un problema crece exponencialmente a medida que avanzan las fases del proyecto en las que se detecte.

TÉCNICAS DE DISEÑO DE CASOS DE PRUEBAS

Clasificación clásica de técnicas de prueba:

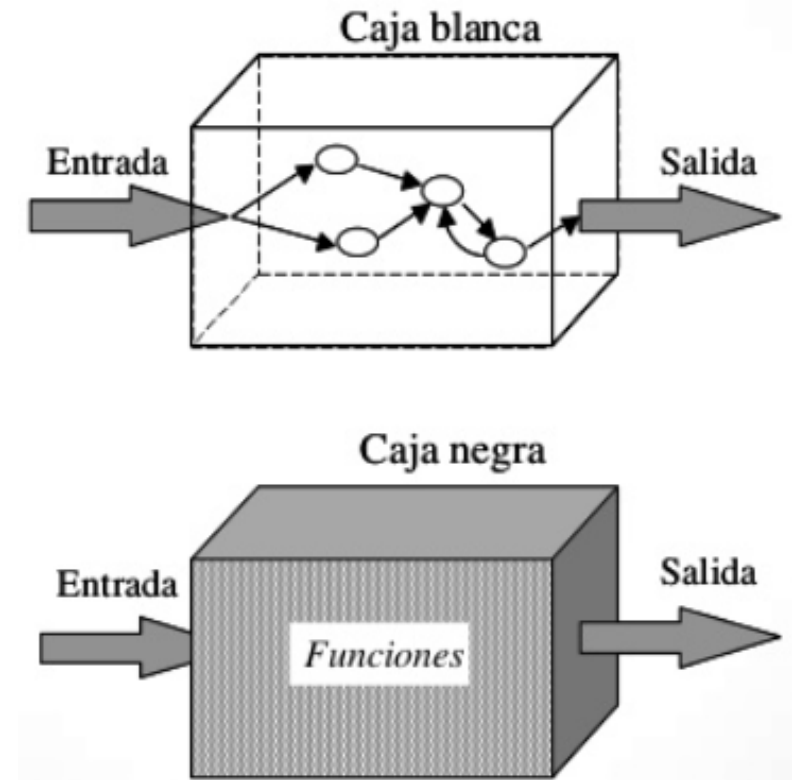
- **Pruebas estructurales o de caja blanca:** Cuando se diseña la prueba a partir del conocimiento de la estructura interna del código. Estas pruebas se centran en la estructura interna del programa, analizando los caminos de ejecución.
- **Pruebas funcionales o de caja negra:** Su objetivo es validar que el código cumple la funcionalidad definida para el programa sin el conocimiento de su estructura interna.

TÉCNICAS DE DISEÑO DE CASOS DE PRUEBAS

Clasificación clásica de **técnicas de diseño de casos de prueba**:

Las **pruebas de caja blanca** permitirán recorrer todos los posibles caminos del código y ver qué sucede en cada caso posible. Se probará qué ocurre con las condiciones y los bucles que se ejecutan. Las pruebas se llevarán a cabo con datos que garanticen que han tenido lugar todas las combinaciones posibles. Para decidir qué valores deberán tomar estos datos es necesario saber cómo se ha desarrollado el código, buscando que no quede ningún rincón sin revisar.

Las técnicas de **pruebas de caja negra** pretenden encontrar errores en funciones incorrectas o ausentes, errores de interfaz, errores de rendimiento, inicialización y finalización. Se centra en las funciones y en sus entradas y salidas.



TÉCNICAS DE DISEÑO DE CASOS DE PRUEBAS

Un **buen caso de prueba** es aquel que tiene una probabilidad muy alta de descubrir un nuevo error.

Para **diseñar** los casos de prueba, vamos a estudiar las **dos técnicas** antes mencionadas, las cuales se verán en detalle en las dos unidades didácticas que siguen.

CASOS DE PRUEBA

A partir del **plan de pruebas** se deberá especificar las **partes** de código a tratar, en qué **orden** habrá que hacer las pruebas, **quien** las hará y mucha información más. Ahora sólo falta entrar en detalle, especificando el **caso de prueba** para cada una de las pruebas a realizar.

Un **caso de prueba** define cómo se llevarán a cabo las pruebas, especificando, entre otros: el tipo de pruebas, las entradas de las pruebas, los resultados esperados o las condiciones bajo las que se tendrán que desarrollar.

CASOS DE PRUEBA

Los **casos de pruebas** tienen un objetivo muy marcado: **identificar los errores** que hay en el software para que éstos no lleguen al usuario final.

Estos errores pueden encontrarse como defectos en la interfaz de usuario, en la ejecución de estructuras de datos o un determinado requisito funcional.

A la hora de **diseñar los casos de prueba**, **no sólo se debe validar que la aplicación hace lo que se espera ante entradas correctas**, sino que también se debe validar que tenga un **comportamiento estable ante entradas no esperadas**, informando del error.

CASOS DE PRUEBA

Los casos de prueba siguen el siguiente ciclo de vida:

- **Definición** de los casos de prueba.
- **Creación** de los casos de prueba.
- **Selección** de los valores para las pruebas.
- **Ejecución** de los casos de prueba.
- Comparación de los **resultados obtenidos** con los **resultados esperados**.

Cada caso de prueba deberá ser independiente de los demás, tendrá un comienzo y un final muy marcado y deberá almacenar toda la información referente a su definición, creación, ejecución y validación final.

CASOS DE PRUEBA

A continuación se indican algunas informaciones que debería contemplar cualquier caso de prueba:

- Identificador del caso de prueba.
- Módulo o función a probar.
- Descripción del caso de prueba detallado.
- Entorno que se deberá cumplir antes de la ejecución del caso de prueba.
- Datos necesarios para el caso, especificando sus valores.
- Tareas que ejecutará el plan de pruebas y su secuencia.

CASOS DE PRUEBA

- Resultado esperado.
- Resultado obtenido.
- Observaciones o comentarios después de la ejecución.
- Responsable del caso de prueba.
- Fecha de ejecución.
- Estado (finalizado, pendiente, en proceso).

[illegible]

CASOS DE PRUEBA

Todo caso de prueba está asociado, como mínimo, a un procedimiento de prueba.

Los **procedimientos de prueba** especifican cómo se podrán llevar a cabo los casos de prueba o parte de estos de forma independiente o de forma conjunta, estableciendo las relaciones entre ellos y el orden en que deberán atender.

DEFINICIÓN Y EJEMPLO DE CASO DE PRUEBA

CASO DE PRUEBA: Es un conjunto de entradas de prueba, condiciones de ejecución y resultados esperados. Tiene un objetivo concreto (probar algo). Ejemplo

- **Caso de prueba:** CP1 para el **caso de uso** (módulo): “Entrada al Sistema”
- **Entrada:** usuario “hacker” contraseña “ejemplo”

Condiciones de ejecución: no existe en la tabla “Usuarios” definida por las columnas (Usuario, Contraseña, Intentos) el registro (“hacker”, “ejemplo”, x) pero sí un registro (“hacker”, “hola”, x)

- **Resultado esperado:** no deja entrar y cambia el registro (“hacker”, “hola”, x+1).
- **Objetivo del caso de prueba:** comprobar que no deja entrar a un usuario existente.

Usuario

Contraseña

¿Desea cambiar la contraseña?
Me olvide la contraseña

Entrar

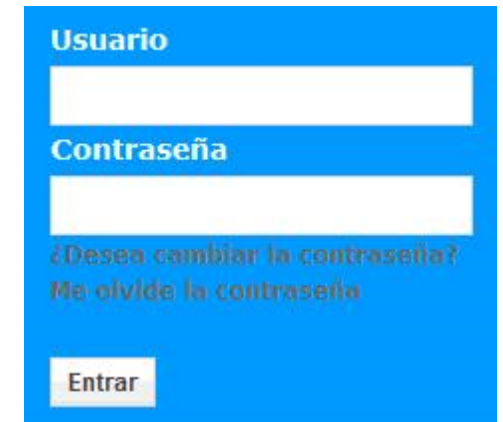
Usuario	Contraseña	Intentos
hacker	hola	1
gabriela	mundo	2
jesús	adiós	0

DEFINICIÓN Y EJEMPLO DE PROCEDIMIENTO DE PRUEBA

PROCEDIMIENTO DE PRUEBA: Pasos que hay que llevar a cabo para probar uno (o varios) casos de prueba. ¿Cómo probar el caso de prueba y verificar si ha tenido éxito?

Ejemplo: Procedimiento de prueba para Caso de Prueba CP1

1. Ejecutar el programa EntradaSistema. Comprobar que en la BD existe el registro ("hacker","hola",x)
2. Escribir "hacker" en la interfaz gráfica (en el campo de texto etiquetado como "Usuario". Escribir "ejemplo" en la interfaz gráfica (en el campo de texto etiquetado como "Contraseña")
3. Pulsa el botón "Entrar"
4. Comprobar que no deja entrar al sistema y que en la BD el registro a cambiando a ("hacker", "hola", x+1)



Usuario

Contraseña

¿Desea cambiar la contraseña?

Me olvide la contraseña

Entrar

Usuario	Contraseña	Intentos
hacker	hola	1
gabriela	mundo	2
jesús	adiós	0

DEPURACIÓN DE CÓDIGO FUENTE

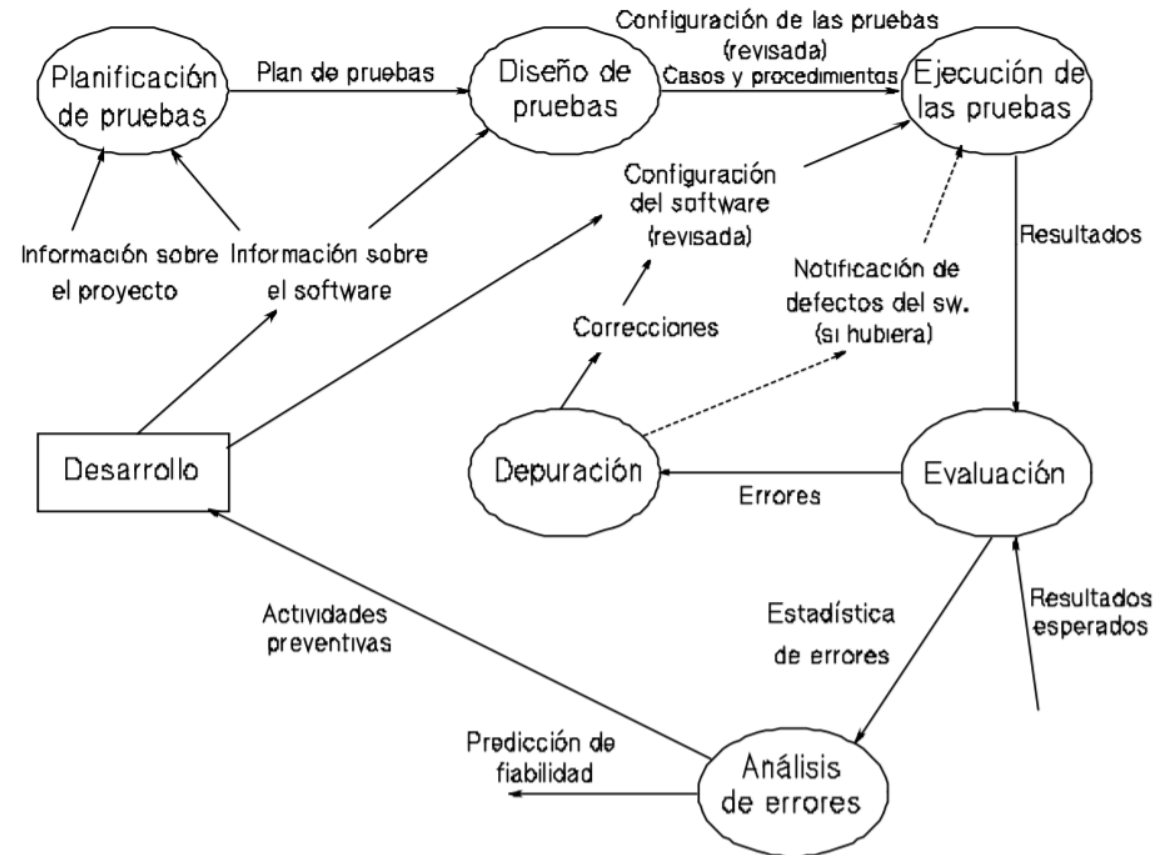
Una técnica muy importante para la ejecución de las pruebas y, en general, para los programadores, es la **depuración del código fuente**.

Los **depuradores** (debuggers) son aplicaciones o herramientas que permiten la ejecución controlada de un programa o un código, siguiendo las instrucciones ejecutadas observando los estados intermedios de las variables y los datos implicados para facilitar la localización de los defectos (bugs).

DEPURACIÓN DE CÓDIGO FUENTE

La depuración ocurre como consecuencia de una prueba.

En el proceso de depuración se evalúan los resultados debido a que se detecta una discrepancia entre los resultados esperados y los observados.



DEPURACIÓN

Los procedimientos que están vinculados a la depuración del código fuente son:

- 1º Identificar la casuística para poder reproducir el error.
- 2º Diagnosticar el problema.
- 3º Solucionar el error atacando el problema.
- 4º Verificar la corrección del error y comprobar que no se han introducido nuevos errores en el resto del software (volver a probar).

DEPURACIÓN

La depuración del código es una herramienta muy útil si se sabe localizar la parte del código fuente donde se encuentra un determinado error.

Si nuestro código es muy complejo y/o el error es difícil de reproducir, también será complicado encontrar una solución. En estos casos tendremos que hacer uso de la depuración acotando el error dentro del código hasta localizarlo.

También hay que tener en cuenta que muchas veces una **modificación en el código para solucionar un problema puede generar otro error que esté relacionado** o que no tenga nada que ver. Habrá que volver a confirmar la correcta ejecución del código una vez finalizada la corrección.

DEPURACIÓN

La depuración de código permite la creación de un **punto de ruptura (breakpoint)** en el código fuente hasta el cual el software será ejecutado de forma directa.

Cuando la ejecución haya llegado a este **punto de ruptura**, se podrá **avanzar en la ejecución del código paso a paso** hasta encontrar el error que se busca.

DEPURACIÓN

Otra forma de llevar a cabo la depuración de código es ir atrás, desde el punto del error, hasta encontrar el causante.

Esta táctica se utiliza en casos más complejos y no es tan habitual, pero será muy útil en el caso de no tener detectada la ubicación del error, que puede encontrarse oculto en alguna parte del código fuente.

En estos casos, a partir del lugar donde se genera el error, se llevará a cabo un recorrido hacia atrás, yendo paso a paso en sentido inverso.

FRAMEWORKS PARA PRUEBAS UNITARIAS

Las **pruebas unitarias** se ocupan de probar componentes individuales de un software para asegurarse de que se ejecute según sus especificaciones. Estos componentes individuales, en la programación orientada a objetos suelen ser métodos.

Existen frameworks para realizar pruebas unitarias prácticamente en cualquier lenguaje de programación. En el caso de **Java** podemos incluso elegir entre **varias opciones**.

Antes de la llegada de estos frameworks, los programadores probaban sus programas imprimiendo en la consola o en un archivo de seguimiento expresiones de prueba. Este enfoque no es satisfactorio porque requiere el criterio humano para analizar los resultados producidos.

JUNIT

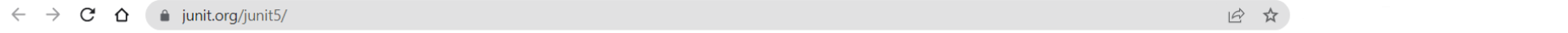


Uno de estos frameworks es **JUnit**, puede que el más consolidado para realizar pruebas unitarias en Java.

JUnit no está incluido en el JDK de Java, pero está incluido por defecto en los principales IDEs como NetBeans y Eclipse.


Usaremos la versión más reciente de JUnit (versión 5), siendo [esta](#) la página principal del framework.

JUNIT




 JUnit 4

The 5th major version of the programmer-friendly
testing framework for Java and the JVM

 User Guide

 Javadoc

 Code & Issues

 Q & A

 Support JUnit

Dentro de la página principal podemos acceder a varios enlaces de gran utilidad, como la [guía de usuario](#), la API con la [documentación oficial](#) y el repositorio de [GitHub](#) con algunos ejemplos básicos.

ESCRIBIENDO CASOS DE PRUEBA. PRUEBAS UNITARIAS. LAS MEJORES PRÁCTICAS.

¿Cómo probar un programa para hacerlo correctamente?

Hay dos técnicas: prueba de **caja blanca** y prueba de **caja negra**.

- Las pruebas de caja blanca inspeccionan los códigos del programa y prueban la lógica del programa.
- Las pruebas de caja negra no inspeccionan los códigos del programa, sino que miran la entrada-salida y tratan el programa como un recuadro negro.

Veremos estas técnicas en las dos próximas unidades