IES SERRA PERENXISA

GENERALITAT VALENCIANA
Conselleria de Educación, Cultura, Universidades y Empleo

FPCV
Formació Professional Comunitat Valenciana

Unidad 01 - Fundamentos del desarrollo de software y prueba de aplicaciones

Puesta en Producción Segura - CE Seguridad Informática
Juan José Martínez Zaballos - IES SERRA PERENXISA
Revisión : 2025 - 09

CETI 25//26

OBJETIVOS DE LA UNIDAD

- Comparar diferentes lenguajes de programación de acuerdo a sus características principales.
- Describir los diferentes modelos de ejecución de software.
- Reconocer lo elementos básicos del código fuente, dándoles significado.
- Ejecutar diferentes tipos de pruebas de software.
- Evaluar los lenguajes de programación de acuerdo con la infraestructura de seguridad que proporcionan.

https://www.itdigitalsecurity.es/endpoint/2025/09/el-85-de-las-empresas-espanolas-carece-de-personal-cualificado-en-materia-de-ciberseguridad

ÍNDICE

Duración aprox : 20 horas / 10 clases

INTRODUCCIÓN AL DESARROLLO SEGURO

Los fallos de seguridad en software cuestan mucho dinero y reputación.

Ejemplo: en 2017, Equifax perdió los datos de 147 millones de personas por no parchear una librería de Java → coste estimado: más de 1.400 millones de $.

- Más del 80% de los ataques explotan vulnerabilidades conocidas en software (fuente: OWASP, ENISA).
- Arreglar un fallo en producción es hasta **30 veces más caro** que hacerlo en la fase de desarrollo (datos de IBM Systems Sciences Institute).
- **Un solo error de programación** (ej. inyección SQL) puede exponer miles de registros sensibles.

La vulnerabilidades de software han cobrado muchas víctimas en los últimos años.

Las empresas cada vez exigen a los equipos de desarrollo una alta eficiencia durante todo el ciclo del software, para garantizar un software de calidad y sobre todo, seguro.

INTRODUCCIÓN AL DESARROLLO SEGURO

¿Que es el desarrollo seguro?
El desarrollo seguro es una necesidad en el diseño y desarrollo de software. La idea detrás del diseño y desarrollo seguro de aplicaciones es tener en cuenta la seguridad desde el minuto cero del ciclo de vida del software.

¿Que es el ciclo de vida del software?
El ciclo de vida es el conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o reemplazado (muere).

INTRODUCCIÓN AL DESARROLLO SEGURO

METODOLOGÍAS ÁGILES DE DESARROLLO

Scrum, Kanban, XP, FDD, DSDM, etc.

- Enfocadas en velocidad y flexibilidad
- Iteraciones cortas (sprints)
- Priorizan entregas rápidas y funcionales
- Mejoran la comunicación y la adaptación

Pregunta clave: *"¿Qué quiere el cliente ahora?"*

MARCOS DE DESARROLLO SEGURO

OWASP SAMM, Microsoft SDL, NIST SSDF

- Enfocados en seguridad y calidad
- Seguridad integrada en cada fase
- Priorizan gestión de riesgos y cumplimiento
- Mejoran la confianza, resiliencia y cumplimiento legal

Pregunta clave: *"¿Cómo aseguramos que lo que hacemos es seguro?"*

"Las metodologías ágiles organizan el desarrollo.
Los marcos de desarrollo seguro integran la seguridad en ese proceso."

INTRODUCCIÓN AL DESARROLLO SEGURO

Riesgo real : paquetes “npm” comprometidos

Junio 2024 — Miles de paquetes npm fueron comprometidos, publicando versiones maliciosas.

Resultado: robo de credenciales de desarrolladores y posible infiltración en entornos de producción.

Fuente: *The Hacker News*

"Con un simple npm install, miles de empresas quedaron expuestas."

¿Qué es npm y por qué importa la seguridad?

- npm: Node Package Manager, gestor de librerías para proyectos Node.js.
- npm install: descarga todas las dependencias declaradas en package.json.
- Riesgo: si una librería está comprometida, puede ejecutar código malicioso en tu entorno de desarrollo o producción.
- Reflexión clave: ***"Tu software es tan seguro como la librería más insegura que uses."***

EJERCICIO PRÁCTICO

Paso 0 – Preparación. Instala Node.js

En Ubuntu:

```
sudo apt update && sudo apt install -y nodejs npm
```

En Windows: descarga desde 👉 https://nodejs.org

Comprueba que está bien instalado:

```
node -v
npm -v
```

Paso 1 – Creamos un proyecto vacío

```
mkdir demo-npm
cd demo-npm
npm init -y
```

Esto genera un archivo **`package.json`** con metadatos básicos del proyecto. Es como la ficha técnica del proyecto de SW.

(Si alguien manipula este archivo, por ejemplo añade una dependencia maliciosa tenemos una vulnerabilidad importante)

Node.js sirve para ejecutar código JavaScript del lado del servidor, permitiendo crear aplicaciones web rápidas y escalables, como servidores web, APIs, microservicios y aplicaciones en tiempo real (chats, juegos), gracias a su modelo de E/S sin bloqueo y su eficiente manejo de múltiples conexiones. Su uso del mismo lenguaje en el front-end y back-end simplifica el desarrollo full stack.

EJERCICIO PRÁCTICO

Paso 2 – Instala una dependencia conocida

```
npm install express
```

- Verás que se crea la carpeta **`node_modules/`** (donde están todas las librerías).

Mira dentro de la carpeta: `node_modules`
Destacable : 👉 Sorprenderá la cantidad de dependencias instaladas por **una sola librería**.

Paso 3 – Muestra dependencias transitivas

```
npm list --depth=0
npm list --depth=1
```

Esto enseña cómo `express` a su vez depende de muchas otras librerías → es la **cadena de suministro**.

`npm audit` (usa bases públicas)

`npm view express` (muestra metadatos del paquete)

NOTA: ¿Qué es Express?

Express es un **framework web minimalista para Node.js**. Su objetivo principal es **facilitar la creación de aplicaciones web y APIs**.

- Gestionar **peticiones y respuestas** (GET, POST, etc.).
- Usar **middlewares** (bloques de código que procesan la petición, p. ej. autenticación, logging, parsing de JSON).
- Levantar un **servidor web** en pocas líneas de código.

EJERCICIO PRÁCTICO

Paso 4 – Desde el ángulo de seguridad

“Cada vez que hacemos `npm install`, descargamos código de internet que **no hemos auditado**. Si un atacante compromete un paquete de esta cadena, puede ejecutar código malicioso en nuestro entorno de desarrollo o producción”.

👉 Opcional : busca en el código instalado algo sensible:

```
Get-ChildItem -Recurse node_modules | Select-String "process.env"
```

o

```
grep -R "process.env" node_modules/express | head
```

Esto muestra cómo un paquete puede acceder a **variables de entorno** (donde a menudo guardamos contraseñas o tokens).

1. FUNDAMENTOS DE PROGRAMACIÓN

PROGRAMA:

Un programa es una secuencia de instrucciones que permiten controlar el comportamiento físico o lógico de un sistema informático.

LENGUAJES DE PROGRAMACIÓN:

Es un conjunto de instrucciones y reglas (lógica) que forman un lenguaje formal y proporcionan la capacidad de escribir órdenes que controlan el comportamiento de un ordenador y sus programas.

Elementos que lo componen:

Sintaxis:
Estructura gramatical del lenguaje. “Que lo que escribes sea gramaticalmente correcto”.

Semántica:
Se refiere al significado del código. Un programa puede ser sintácticamente correcto pero semánticamente incorrecto. “Compila, pero no hace lo que quiero”.

1. FUNDAMENTOS DE PROGRAMACIÓN

```python
# Programa sencillo en Python
nombre = input("Introduce tu nombre: ")
print(f"Hola, {nombre}!")
```

- Sintaxis simple, no se declaran tipos.
- Interpreta directamente línea por línea.

```java
// Programa sencillo en Java
import java.util.Scanner;

public class Saludo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduce tu nombre: ");
        String nombre = sc.nextLine();
        System.out.println("Hola, " + nombre + "!");
        sc.close();
    }
}
```

- Java requiere **clase** y método `main`.
- Es **fuertemente tipado**: debes declarar los tipos de las variables (`String`).
- Compila antes de ejecutar.

1. FUNDAMENTOS DE PROGRAMACIÓN

```
// Programa sencillo en C++
#include <iostream>
#include <string>
using namespace std;

int main() {
    string nombre;
    cout << "Introduce tu nombre: ";
    cin >> nombre;
    cout << "Hola, " << nombre << "!" << endl;
    return 0;
}
```

- C++ también es **tipado estático**, pero más cercano al hardware.
- Uso de `cin`/`cout` para entrada/salida.
- Necesitas incluir **librerías** (`#include <iostream>` y `<string>`).

1. FUNDAMENTOS DE PROGRAMACIÓN

TIPOS DE DATOS:

Diferentes tipos de valores que se pueden manejar en un programa.

Primitivos: int, float, double, char, bool.

Compuestos: Arrays, listas, tuplas, diccionarios, conjuntos

Definidos por el usuario: estructuras, clases.

VARIABLES:

Contenedores que almacenan valores y están asociados a un tipo de dato. Los lenguajes de programación utilizan variables para manipular datos y conservar información.

1. FUNDAMENTOS DE PROGRAMACIÓN

OPERADORES:

Símbolos que indican qué operaciones se realizarán sobre los valores y las variables.

Aritméticos: + - * / %

Relacionales : == , != , <, >, >=, <=

Lógicos : &&, !. (Dependen de la sintaxis, buscar ejemplos)

Asignación: =, +=, -=, *=, /= %=

1. FUNDAMENTOS DE PROGRAMACIÓN

SENTENCIAS DE CONTROL DE FLUJO:

Expresiones que permiten modificar el flujo de ejecución del programa dependiendo de ciertas **condiciones**.

If – else:

Se indica una condición que al evaluarse devuelve cierto o falso y en función del resultado ejecuta uno u otro bloque.

Switch:

Permite evaluar una expresión y ejecutar más de dos posibles opciones en función de la expresión.

```
if (condición) then
        bloque de instrucciones si cierto
else
        bloque de instrucciones si falso
endif
```

```
Switch (expresión):
Case X:
    Bloque instrucciones
Break
Case Y:
    Bloque instrucciones
Break
Default:
     Bloque instrucciones
endswitch
```

1. FUNDAMENTOS DE PROGRAMACIÓN

SENTENCIAS DE CONTROL DE FLUJO:

BUCLES

While

Se ejecuta el bloque contenido en la estructura mientras se cumpla la condición lógica. Puede darse el caso de no ejecutarse ninguna vez.

```
While (condición):
    bloque de instrucciones.
endwhile
```

Do-While

La evaluación de la condición si ejecuta al final del bloque de instrucciones por lo que se ejecuta como mínimo una vez.

```
Do
    bloque de instrucciones.
while (condición):
```

1. FUNDAMENTOS DE PROGRAMACIÓN

SENTENCIAS DE CONTROL DE FLUJO:

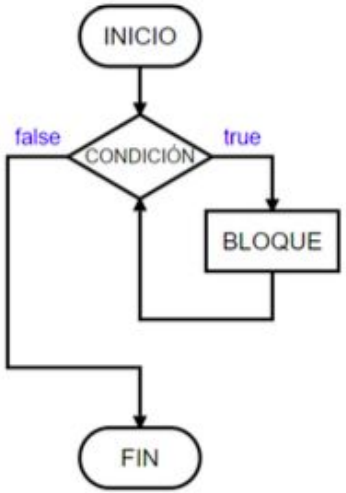
BUCLES:

For:

En este caso, además de la condición, posee un bloque de instrucciones que se ejecutan al inicio, el bloque en el que se encuentra la condición a evaluar en cada iteración y por último un bloque de instrucciones que se ejecuta en cada iteración del bucle.

SALTOS: break, continue, return, goto (desuso)

```
For (bloque incialización; bloque condición; bloque iteracción):
    bloque instrucciones.
endFor
```



1. FUNDAMENTOS DE PROGRAMACIÓN

FUNCIONES:

Fragmento de código identificado con un nombre, que puede ser reutilizado utilizando ese nombre, que devuelve un valor de un tipo de dato, que recibe una serie de argumentos de un determinado tipo de llamados parámetros en cada una de las llamadas.

```
int funcion_multiplicar(int a, int b){
    return a*b;
    }
int valor1=5, valor2=2;
int resultado=funcion_multiplicar(valor1,valor2);
```

1. FUNDAMENTOS DE PROGRAMACIÓN

ESTRUCTURAS DE DATOS:

Formas organizadas para almacenar y gestionar colecciones de datos

Arrays, Listas enlazadas, Pilas y colas, Árboles, Grafos, Tablas hash

MANEJO DE ERRORES

Técnicas para manejar situaciones inesperadas.

try, catch, finally

ENTRADA/SALIDA

Interacciones con el mundo exterior

```
# función para imprimir “Hola Mundo”
def hola_mundo():
        mensaje = “Hola Mundo”  #declara variable y asignación
        print(mensaje)  #llamada a la función y salida
hola_mundo()        # invocación de la función
```

COMENTARIOS:

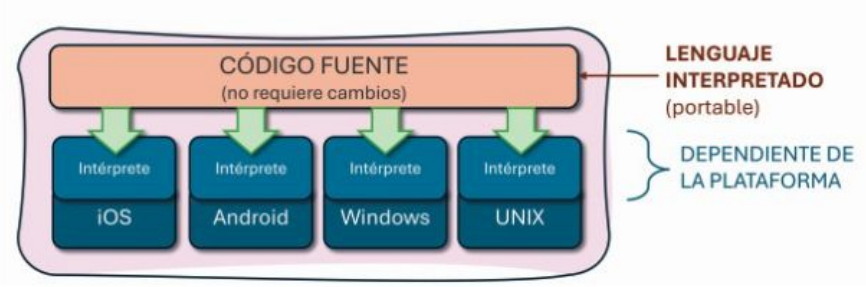
Notas escritas por el programador para explicar qué hace el código. # // /*

2. LENGUAJES INTERPRETADOS vs COMPILADOS

LENGUAJE INTERPRETADO

El lenguaje interpretado es aquel cuyo código fuente es traducido por un intérprete a un lenguaje entendible para la máquina.

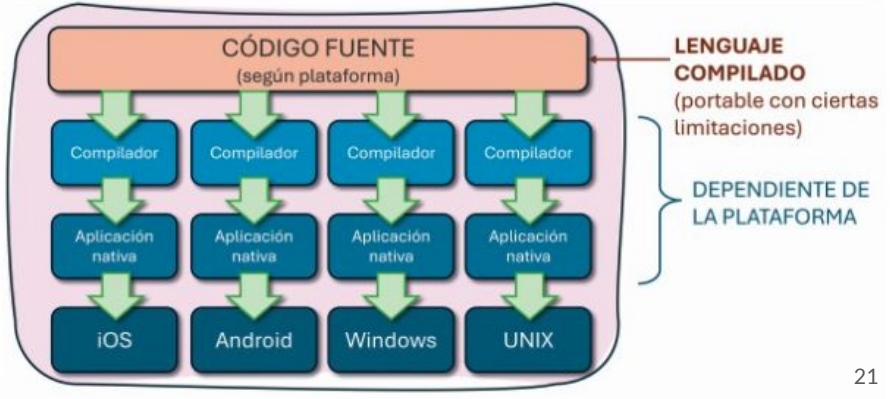
Es convertido a lenguaje máquina a medida que es ejecutado.



LENGUAJE COMPILADO

El lenguaje compilado es aquel cuyo código fuente es traducido por un compilador a un archivo ejecutable.

El lenguaje compilado requiere un paso adicional antes de ser ejecutado.



2. LENGUAJES INTERPRETADOS vs COMPILADOS

LENGUAJE INTERPRETADO

Ciclo de desarrollo más rápido

Son multiplataforma

Mayor facilidad para probarlos

—----------

La máquina debe tener instalado el intérprete para poder ejecutar código.

Menor velocidad a la hora de ejecución.

El código fuente es público.

LENGUAJE COMPILADO

Mucho más rápido a la hora de ejecutarse.

El código fuente es inaccesible.

Preparados para ejecutarse.

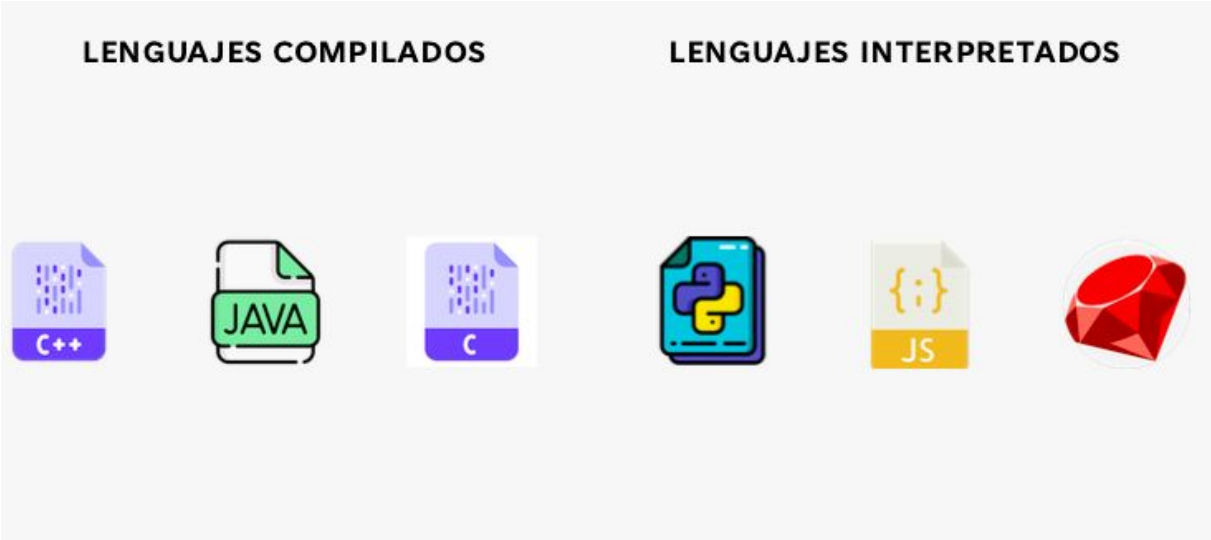
—----------

Los ejecutables no son compatibles con todos los sistemas operativos.

Poca flexibilidad.

Se requiere un paso extra para la ejecución.

2. LENGUAJES INTERPRETADOS vs COMPILADOS



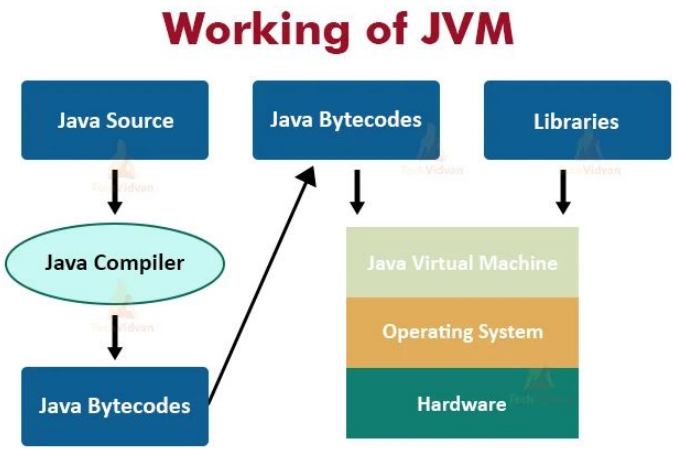
Ruby

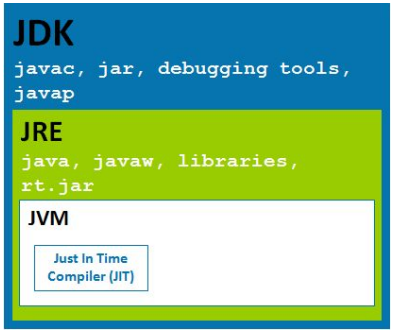
2. LENGUAJES INTERPRETADOS vs COMPILADOS

LENGUAJES HÍBRIDOS : JAVA

JAVA es un lenguaje compilado, pero tiene una compilación particular, ya que es compilado a un lenguaje intermedio llamado bytecode, que posteriormente es interpretado.

JAVA se creó como un lenguaje compilado que se pudiera ejecutar en cualquier sistema operativo y procesador sin necesidad de crear varios ejecutables, es por eso, que para ejecutar código JAVA debes instalar JRE(Java Runtime Enviroment) y para compilar código necesitas JDK (Java Development Kit).





3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

“Conjunto de líneas de texto escritas en un lenguaje de programación que contienen la lógica de la aplicación.”

ESTRUCTURA DEL CÓDIGO FUENTE:

- VARIABLES
- COMPARACIONES
- BUCLES
- COMANDOS
- COMENTARIOS

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

Para desarrollar un código fuente solo nos hace falta un **editor de texto simple**, como, **Bloc de notas (Windows), Gedit (Ubuntu)**, etc. Pero si además de poder editar texto plano simple, incluyen funcionalidades relacionadas con la programación se denominan **editores de código**.

– Resalte de sintaxis
– Autocompletado
– Resalte de llaves
– Plegado de código
– Edición multilínea

```
const Paciente = {
  constructor () {
    this.nombre = 'Pepe';
    this.apellidos = 'Pepito';
    this.fechaDeNacimiento = '23/09/2000';
    this.altura = '187';


    return {
      nombre: this.nombre,
      apellidos: this.apellidos,
      fechaDeNacimiento: this.fechaDeNacimiento,
      altura: this.altura,

    }
  },

  /**
   * Setters.
   */
  modificarNombre(newName) {
    if (!newName) {
      return "Debes añadir tu nuevo nombre";
    } else {
      this.nombre = newName;
      return `Tu nuevo nombre es ${this.nombre}`;
    }
```

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

Ejemplos de editores de código:

Sublime text

Atom

Visual Studio Code

Notepad++

https://blog.infranetworking.com/top-5-mejores-editores-de-codigo/

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

Los proyectos de mayor complejidad deben desarrollarse con entornos de desarrollo (IDE). Dichos entornos engloban diversas herramientas para acelerar el flujo de trabajo.

Generalmente, un IDE cuenta con las siguientes características:

- Editor de código fuente.
- Depurador.
- Automatización de compilación.

Ejemplos de entornos de desarrollo **IDE:**

IntelliJ

Eclipse

Visual Studio

Netbeans

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

PRÁCTICA 1 : ENTORNOS DE DESARROLLO

Ejemplos de entornos de desarrollo **IDE**:

IntelliJ

Eclipse

Visual Studio

Netbeans

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

VENTAJAS DE USAR UN IDE:

- La curva de aprendizaje es muy baja.
- Proporciona mayor agilidad y optimización para usuarios inexpertos en el uso de la consola.
- Avisos de errores en pantalla.
- Visualización de la estructura del proyecto de manera gráfica.
- Navegador interno.
- Formato de código.

3. CÓDIGO FUENTE Y ENTORNOS DE DESARROLLO

Visual Studio (VS)

- Es un **IDE (Entorno de Desarrollo Integrado)** muy completo.
- Está pensado para proyectos grandes, especialmente en **.NET, C#, C++, ASP.NET, Xamarin, Azure**, etc.
- Incluye **compilador, depurador, diseñador de interfaces gráficas, gestión de bases de datos**, integración con Git, pruebas unitarias, etc.
- Es más pesado y requiere más recursos.
- Versión gratuita: **Visual Studio Community**.

👉 Ejemplo de uso: una empresa que desarrolla una aplicación completa en C# con interfaz gráfica y conexión a SQL Server usaría Visual Studio.

Visual Studio Code (VS Code)

- Es un **editor de código ligero y multiplataforma** (Windows, Linux, macOS).
- No trae compilador propio: depende de que instales **extensiones** o el compilador/lenguaje aparte.
- Muy flexible gracias a su **marketplace de extensiones** (sirve para Python, Java, C++, JavaScript, PHP, etc.).
- Arranca rápido, consume menos recursos.
- Muy usado por desarrolladores web, DevOps, administración de sistemas, etc.

👉 Ejemplo de uso: un programador que hace una API en **Node.js** o scripts en **Python** seguramente use VS Code.

4. ELEMENTOS DE UN PROGRAMA

Los elementos principales de un programa informático clásico se dividen en varios grupos dependiendo del paradigma, del lenguaje y de la finalidad. Bloques comunes:

- **Bloque de declaraciones:** incluye la declaración y normalmente la instanciación de todos los objetos y elementos a procesar como constantes o variables.
- **Bloque de instrucciones:** Acciones sobre los elementos definidos en el bloque de declaración que permiten lograr el objetivo del programa. (Entrada, proceso y salida)
- **Espacio de nombre o paquete:** Indica el ámbito o alcance del código de forma uqe se puedan encapsular para su posterior uso.
- **Bloque de uso de elementos externos.** Indica las clases o funciones externas que se van a utilizar en el programa, denominados librerías o paquetes dependiendo del lenguaje.
- **Bloque de definición del fichero/clase:** Se incluyen comentarios como el autor, tipo de licencia, uso o función del código o clase

5. EJECUCIÓN DE SOFTWARE

Para solucionar el problema de carga de ejecutables de gran tamaño y el uso de memoria y dado que muchas de las funcionalidades en los diferentes programas son comunes se desarrollaron las **librerías dinámicas.**

Son fragmentos de código binario que se cargan en memoria y que diferentes programas pueden utilizar.

Problemas con las distintas versiones y la necesidad de utilizar versiones concretas. **(Infierno de DLL)**

Otro problema es el borrado de una libreria al eliminar un programa impidiendo que el resto pueda utilizarla.

5. EJECUCIÓN DE SOFTWARE

Las librerías dinámicas, en especial las DLL, son un punto de entrada de virus y malware al inyectar código mediante librerías para interceptar llamadas del sistema y poder modificarlas.

TECNICA DE MALWARE: **Inyección de DLL**

5. EJECUCIÓN DE SOFTWARE

La inyección de DLL es una técnica en Windows por la cual un atacante consigue que un proceso legítimo cargue y ejecute código arbitrario contenido en una DLL (biblioteca dinámica). El código malicioso corre dentro del contexto del proceso víctima, con las mismas capacidades que ese proceso, lo que permite evadir detección, robar información o actuar con privilegios del proceso.

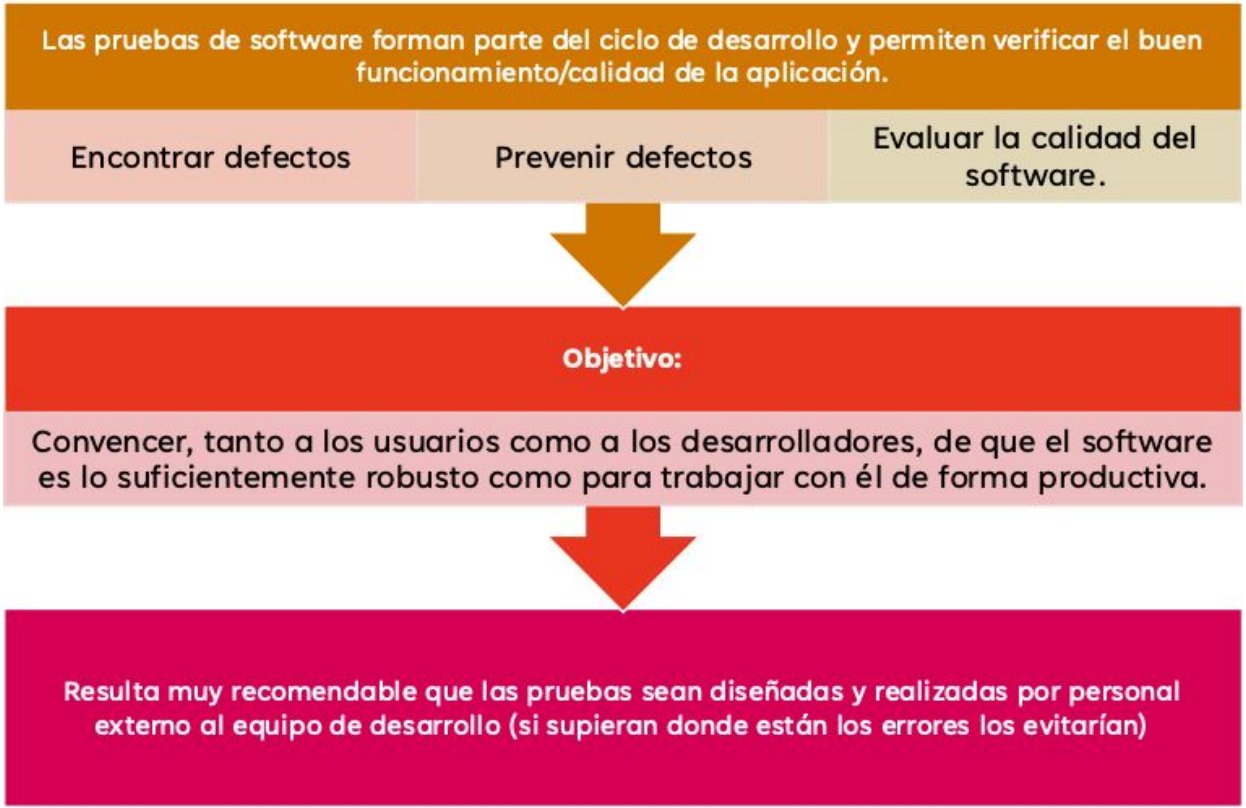
Idea de mini-demo segura (no intrusiva)

1. **Mostrar Process Explorer**: ejecutar una aplicación inocua (por ejemplo, un editor) y ver las DLL que carga. Explicar por qué aparecen. Microsoft Learn

2. **Revisar Autoruns** para ver entradas de inicio legítimas vs entradas sospechosas (sin modificar nada). Microsoft Learn

3. **Analizar un log de Sysmon** (preparado) que muestre un evento de CreateRemoteThread y discutir por qué sería sospechoso. *(Usar ejemplos ya conocidos o logs sanitizados — no ejecutar exploits.)* Mitre ATT&CK

6. PRUEBAS


Las pruebas de software forman parte del ciclo de desarrollo y permiten verificar el buen funcionamiento/calidad de la aplicación.
Encontrar defectos
Prevenir defectos
Evaluar la calidad del software.
Objetivo:
Convencer, tanto a los usuarios como a los desarrolladores, de que el software es lo suficientemente robusto como para trabajar con él de forma productiva.
Resulta muy recomendable que las pruebas sean diseñadas y realizadas por personal externo al equipo de desarrollo (si supieran donde están los errores los evitarían)

6. PRUEBAS

Razones principales para “externalizar” las pruebas:

- Aliviar a los equipos de desarrollo y centrarse en el desarrollo del software.
- Aumento de la satisfacción de los empleados, ya que las pruebas, a menudo desagradables para el equipo de desarrollo, son llevas a cabo por personal especializado.
- Pruebas eficientes realizadas por expertos certificados.
- Prueba utilizando un amplio y siempre actualizado conjunto de instrumentos de inspección.
- La garantía externa de calidad a través de pruebas previene la ceguera operativa.
- Las pruebas calificadas y los informes de errores dan como resultado ciclos de publicación.

6. PRUEBAS

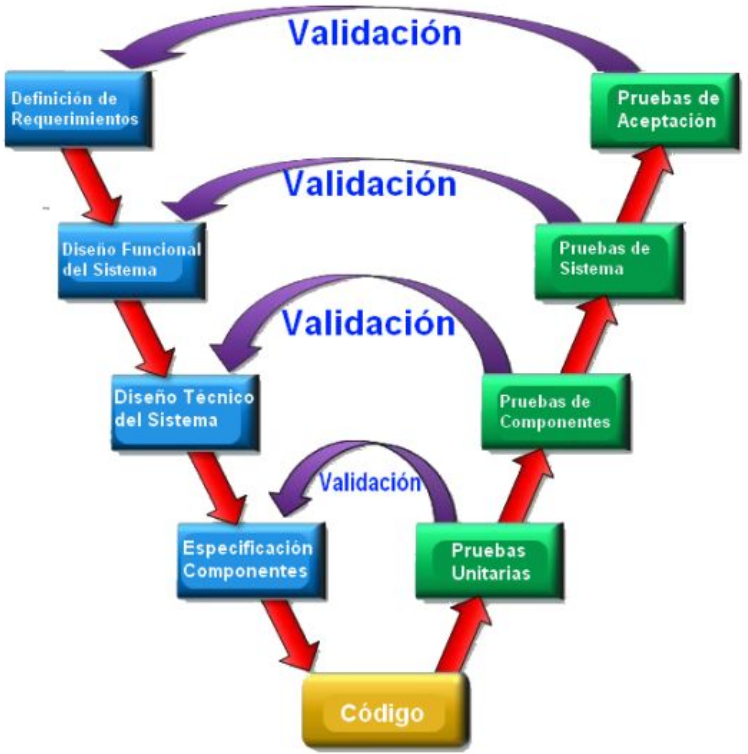
Verificación	Validación
La verificación es el conjunto de actividades que aseguran que el software implementa correctamente una función específica. Determina si un flujo de trabajo se ha llevado a cabo correctamente. ¿Estamos construyendo correctamente el producto? ¿El software cumple sus requisitos?	La validación es un proceso más general, debe asegurar que el sistema software satisface las expectativas del cliente. Proceso de evaluación intensa que se lleva a cabo justo antes de entregar el producto al cliente. ¿Estamos construyendo el producto correcto? ¿Hace el software lo que el usuario realmente requiere?

6. PRUEBAS

ETAPAS O NIVELES DE PRUEBA:

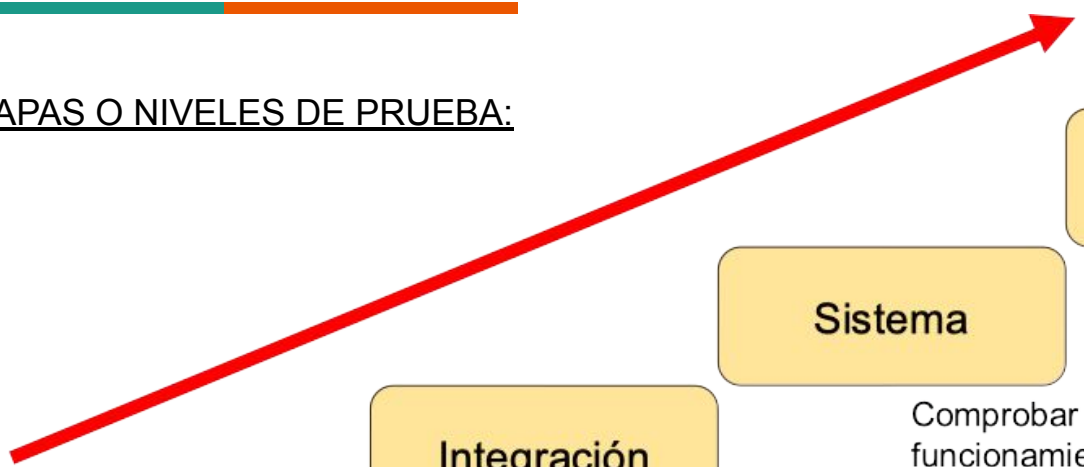
El **modelo en V** plantea que en todo este proceso se debe llevar a cabo la verificación y validación.

Cada nivel de Desarrollo se verifica con el nivel anterior, es decir, se comprueba si los requisitos y definiciones de niveles previous han sido implementados de forma correcta.



6. PRUEBAS

ETAPAS O NIVELES DE PRUEBA:



Unidades

Se centran en probar un único componente de forma aislada

Integración

Permiten probar las comunicaciones de un grupo de componentes relacionados.

Sistema

Comprobar el correcto funcionamiento del sistema completo.

Aceptación

Ayudar al cliente a decidir el paso a producción.

6. TIPOS DE PRUEBAS

Funcionales	No funcionales
Encaminadas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en sus especificaciones durante la fase de análisis de requisitos del software. Las pruebas funcionales tienen por objeto comprobar el correcto funcionamiento de las diferentes funcionalidades del sistema.	Encaminadas a controlar la calidad de los sistemas implementados. Asegurar que todo funciona bien y en qué circunstancias podría fallar. Nos permiten conocer qué riesgos corre el producto y nos dicen si tiene un mal desempeño o un bajo rendimiento en los entornos de producción. Su fin principal es obtener información del sistema.

6. TIPOS DE PRUEBAS

Funcionales	No funcionales
• Pruebas exploratorias • Pruebas de regresión (cada vez que se realiza un cambio de funcionalidad) • Pruebas de compatibilidad de entorno. • Pruebas libres • Pruebas de humo (pruebas rápidas sobre las primeras versiones sin entrar en detalle). • Pruebas de mono (navegar por los distintos caminos del software a lo loco sin criterio teórico) • Pruebas de sanidad (similares a las de regresión)	• Pruebas de rendimiento (de carga, de estrés, de estabilidad) • Recuperación o vuelta atrás (el sistema tendrá la capacidad de recuperarse o ponerse en funcionamiento en caso de caída) • Pruebas de instalación • Pruebas estructurales (caja blanca) • Pruebas de configuración (hardware y software) • Pruebas de usabilidad (experiencia del usuario)

6. TIPOS DE PRUEBAS

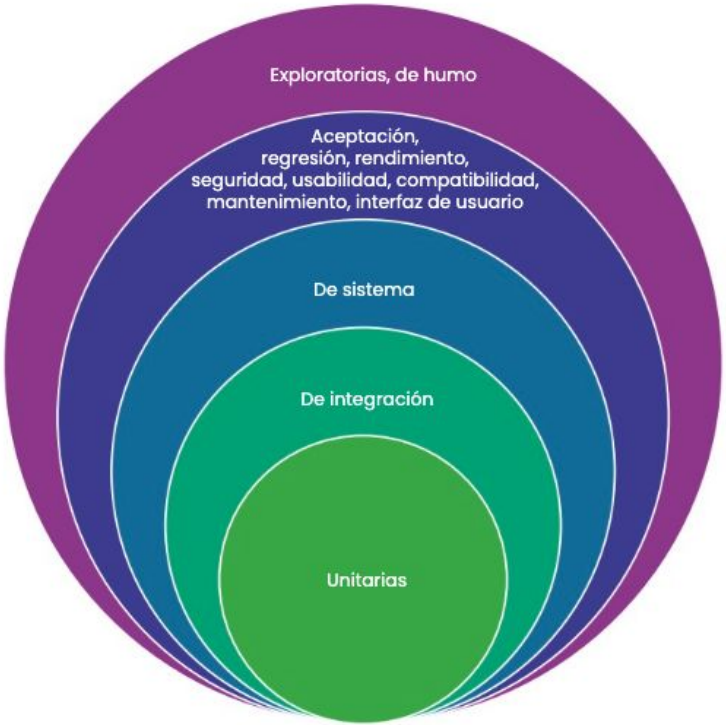


Figura 1.41. Resumen de tipos de pruebas de software.

6. TÉCNICAS DE PRUEBAS

Dentro de cada nivel de prueba, se pueden establecer diferentes tipos de pruebas que establecen que técnica usar:

Enfoque funcional o de **caja negra (basados en especificación)**	Enfoque estructural o de **caja blanca (basados en código)**	Enfoque aleatorio
•Pruebas de comportamiento. •Se centran en estudiar la especificación de las funciones, la entrada y salida para derivar los casos.	•Se basan en el análisis de la estructura del componente o sistema.	•Consiste en utilizar modelos estadísticos que representan las posibles entradas al programa para crear los casos de prueba.

Técnicas de Prueba:

PRUEBAS DE CAJA NEGRA

PRUEBAS DE CAJA NEGRA:

Estas pruebas permiten obtener un conjunto de condiciones de entrada que ejerciten completamente **todos los requisitos funcionales de un programa**.

1. Analizar la especificación.
2. Seleccionar un conjunto de comportamientos según algún criterio.
3. Obtener un conjunto de casos de prueba que recorren estos comportamientos.

Técnicas de Prueba:

PRUEBAS DE CAJA NEGRA

Métodos para la elaboración de pruebas de caja negra

- **Método de clases o particiones equivalentes**
  Las clases de equivalencia (condiciones, particiones) de entrada pueden ser validas o invalidas.
  Por ejemplo, si tenemos una aplicación donde se puede insertar el mes de un año:
  - Clase valida: 1 a 12.
  - Clase inválida: Valor menor a 1 o superior a 12.

- **Método de Análisis de Valores Límite**
  - Prueba la habilidad del programa para manejar datos que se encuentran en los límites aceptables.
- **Método de pruebas de comparación**
  - Partiendo de las mismas especificaciones se desarrollan diversas versiones independientes de una aplicación.
- **Método de la conjetura de errores**
  - A partir de una lista que contiene los posibles errores de la aplicación se generan los casos de prueba.
- **Método de los Grafos de Causa – Efecto**
  - Permite representar sin ambigüedad las relaciones entre las condiciones lógicas y sus respectivas acciones.

Técnicas de Prueba:

PRUEBAS DE CAJA NEGRA

Ejemplo para identificar clases de equivalencia:

Especificación: Método en el que, dados como entradas:

1. un carácter X introducido por el usuario,
2. un número N entre 5 y 10, y
3. el valor “rojo” o “azul”,

devuelve (salida) una cadena de N caracteres X de color rojo o (N-1) caracteres de color azul, o bien el mensaje “ERROR: repite entrada” si el usuario proporciona un valor de N < 5 ó N >10 o no númerico, o caracter no alfanumerico, o un color diferente a “rojo” o “azul” .

Técnicas de Prueba:

PRUEBAS DE CAJA NEGRA

Ejemplo para identificar clases de equivalencia:

- Entrada 1 (E1) (carácter X): puede ser cualquier carácter
  - Clase válida: V1
- Entrada 2 (E2)(número N): un valor comprendido entre 5 y 10
  - Clase válida: V2: valores entre 5 y 10 ($5 \leq N \leq 10$)
  - Clases inválidas: N1: valores menores que 5 ($N < 5$), y N2: valores mayores que 10 ($N > 10$)
- Entrada 3: uno de los valores: “rojo”, “azul”
  - Clases válidas: V3: “rojo”, V4: “azul”
- Salida (cadena de N caracteres):
  - Clase válida: S1: Cadena de N caracteres de color rojo
  - Clase válida: S2: Cadena de (N-1) caracteres de color azul
  - Clase inválida: NS1: “ERROR: repite entrada”

Técnicas de Prueba:

PRUEBAS DE CAJA NEGRA

+

Ejemplo para identificar clases de equivalencia:

Clases	Datos de entrada			Resultado esperado
	E1	E2	E3	
V1-V2-V3-S1	‘a’	6	“azul”	“aaaaa”
V1-V2-V4-S2	‘b’	7	“rojo”	“bbbbbbb”
V1-N1-V4-NS1	‘c’	1	“azul”	“ERROR: repite entrada”
V1-N2-V4-NS1	‘d’	50	“azul”	“ERROR: repite entrada”

- Cubre **todas las clases de equivalencia** (C1–C7) con al menos un test cada una.
- Asegura **pruebas de frontera** para N = 4,5,10,11.
- Añade tests para tipos erróneos (string/float) y entradas múltiples para X.

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

Este método se centra en cómo diseñar los casos de prueba atendiendo al comportamiento interno y la estructura del programa. Se examina la lógica interna del programa sin considerar los aspectos de rendimiento.

Se pueden obtener casos de prueba que:

- Garanticen que se ejerciten por lo menos una vez todos los caminos independientes de cada módulo, programa o método.
- Ejerciten todas las decisiones lógicas en las vertientes verdadera y falsa.
- Ejecuten todos los bucles en sus límites operacionales.
- Ejerciten las estructuras internas de datos para asegurar su validez.

Técnicas de Prueba:

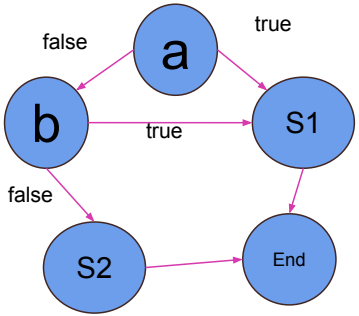
PRUEBAS DE CAJA BLANCA

+

Para las pruebas de caja blanca utilizaremos un CFG (Control Flow Testing). Un CFG no es más que una representación gráfica de una unidad de programa (un método). Para ello usaremos un grafo dirigido donde:

- Cada nodo representa una o más sentencias secuenciales o una única condición.
- Las aristas representan el flujo de ejecución entre dos conjuntos de sentencias.

```
If (a || b){
        S1;
}
else{
        S2;
}
```



Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

+

Nodo:

- Cada círculo dibujado en el grafo de flujo, el cual representa una o más secuencias procedimentales. Un solo nodo puede corresponder a una secuencia de instrucciones o una sentencia de decisión.

Nodo Predicado:

- Cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR…) se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado

Regiones:

- Son las áreas delimitadas por las aristas y los nodos. También se incluye el área exterior del grafo, contando como una región más.

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

EJERCICIO:

Obtén el grafo de flujo siguiente:

```
If (a && b){
        S1;
}
else{
        S2;
}
```

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

Método del camino básico

Es una técnica de Caja Blanca que permite obtener una medida de la complejidad lógica de un diseño y usar esta medida como guía para la definición de un conjunto básico.

Los pasos que se siguen para aplicar esta técnica son:

1. Dibujar el grafo de flujo a partir del código.
2. Calcular la complejidad ciclomática (CC).
3. Obtener los caminos independientes del grafo.
4. Preparar los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

+

Complejidad ciclomática

Se trata de una métrica que proporciona una medida de la complejidad lógica de un componente software. Se calcula como:

CC = número de arcos - números de nodos + 2

El valor del CC indica el máximo número de caminos independientes del grafo.

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

+

Complejidad ciclomática

Existen otras formas de calcular la complejidad ciclomática:

- CC = número de arcos - números de nodos + 2
- CC = número de regiones
- CC = número de condiciones + 1

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

🔹 ¿Se usa realmente la complejidad ciclomática?

- **Auditorías de calidad de software** → Métricas como *SonarQube*, *Coverity* o *Fortify* calculan automáticamente la complejidad ciclomática para detectar funciones demasiado complejas.
- **Buenas prácticas de desarrollo** → La regla práctica es:
  - 1–10: bajo riesgo, código fácil de probar.
  - 10–20: moderado, puede ser difícil de mantener.
  - 20–50: alto riesgo, necesita refactorización.
  - +50: casi inmanejable.

⚠️ Es decir, **no lo calculas a mano en tu día a día**; lo hacen las herramientas y te dicen:
👉 “Esta función tiene complejidad 25, deberías dividirla en funciones más simples”.

Técnicas de Prueba:

PRUEBAS DE CAJA BLANCA

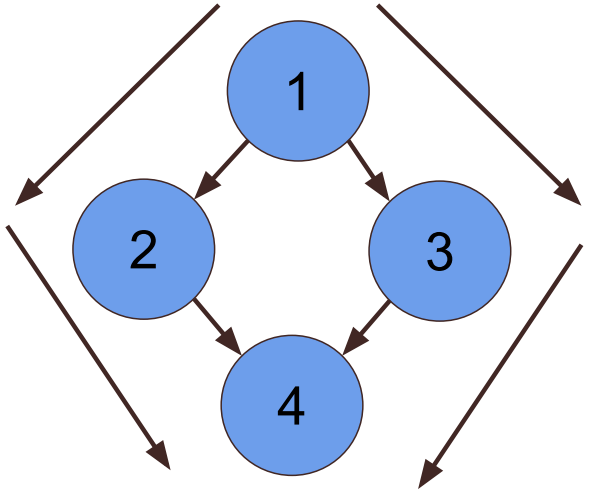
+

Caminos independientes

Buscamos como máximo tantos caminos independientes como valor obtenido de CC.

Ejemplo:

Camino 1-2-4



Camino 1-3-4


Práctica 2:
Grafo de flujo

7. Seguridad de los lenguajes de programación en sus entornos de ejecución (“sandboxes”).

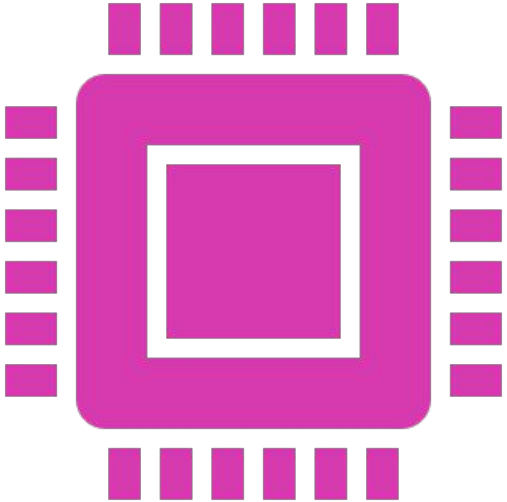
¿Que es una sandbox?

Una sandbox o caja de arena es un entorno de prueba aislado donde los usuarios pueden ejecutar programas o archivos sin afectar a la aplicación, sistema o plataforma donde se ejecutan.

Los profesionales en ciberseguridad usan las sandbox para probar software potencialmente peligroso.

RESUMEN

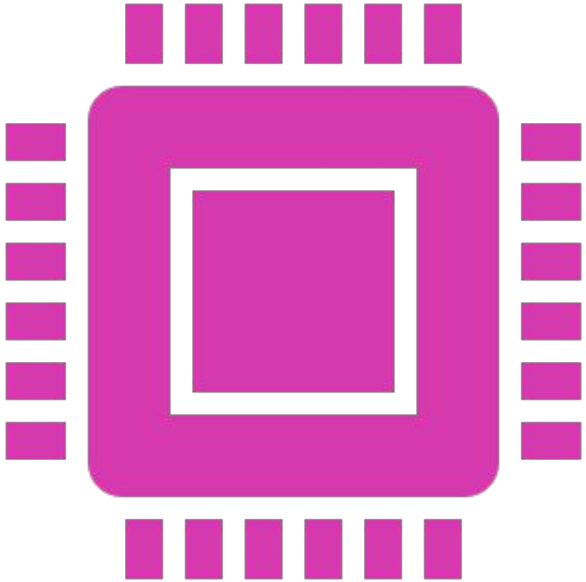
- Los sistemas software son elementos complejos que han de cumplir estrictamente los requisitos para los que han sido diseñados. La creación de sistemas totalmente seguros y previsibles es uno de los objetivos que se persigue en la industria informática y a la que cada día se destinan más recursos, desarrollándose nuevas metodologías, librerías y herramientas con este objetivo.
- Las pruebas de software forman parte del ciclo de desarrollo y permiten verificar el buen funcionamiento/calidad de la aplicación.
- La finalidad primordial del proceso de pruebas es detectar la presencia de posibles defectos del software cuando antes para poder corregirlos.
- Un banco de pruebas de software es un conjunto integrado de herramientas para soportar el proceso de pruebas. La automatización de pruebas consiste en el uso de herramientas para controlar su ejecución y la comparación de resultados.



Práctica 3: Pruebas unitarias en Visual Studio Code

CASO PRÁCTICO 3: PRÁCTICAS UNITARIAS VISUAL STUDIO CODE

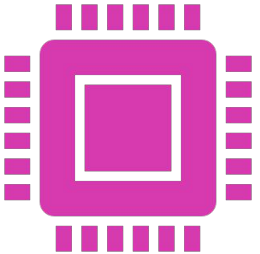
En esta práctica se estudiará cómo implementar y ejecutar de forma automatizada casos de prueba unitarias y de integración. Para ello, hará uso de:



- Visual Studio Code (IDE)
- Node.js (Entorno de ejecución de Javascript)
- El framework de pruebas mocha.js
- La librería chai.js

https://www.youtube.com/watch?v=g3MislTPMMc

CASO PRÁCTICO 3: PRÁCTICAS UNITARIAS VISUAL STUDIO CODE



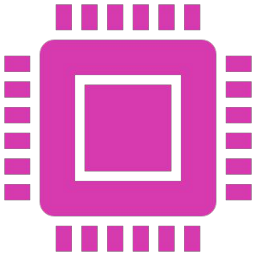
1. Editor y entorno

- **VS Code** → tu editor principal, donde escribes y guardas el código.
- **Node.js** → el motor que ejecuta JavaScript fuera del navegador (necesario para correr tu código y las pruebas).
- **npm** (Node Package Manager) → gestor de dependencias de Node. Lo usas para instalar librerías como Mocha y Chai.

2. Dependencias para testing

- **Mocha** → el **framework de testing**.
  - Define la estructura de las pruebas (`describe`, `it`).
  - Se encarga de ejecutar todos los ficheros en la carpeta `test/`.
- **Chai** → la **librería de aserciones**.
  - Ofrece métodos (`expect`, `should`, `assert`) para comparar resultados esperados con los reales.
  - Ejemplo: `expect(resultado).to.equal(5);`.

CASO PRÁCTICO 3: PRÁCTICAS UNITARIAS VISUAL STUDIO CODE



```
mi-proyecto/
│
├─ index.js          → tu código principal (funciones, lógica, etc.)
├─ package.json      → configuración del proyecto y dependencias
├─ node_modules/     → librerías instaladas con npm (Mocha, Chai, etc.)
└─ test/
   └─ test.js        → tus pruebas unitarias
```

VS Code → donde desarrollas.

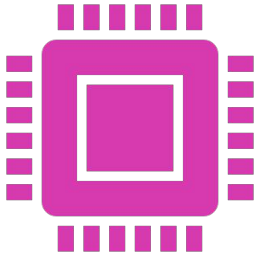
Node.js → ejecuta tanto tu código como Mocha.

npm → gestiona las librerías que necesitas.

Mocha → organiza y corre las pruebas.

Chai → compara resultados esperados vs. obtenidos.

CASO PRÁCTICO 3: PRÁCTICAS UNITARIAS VISUAL STUDIO CODE



Flujo de trabajo

1. Inicializas proyecto:
   `npm init -y` (genera `package.json`)

2. Instalas dependencias de pruebas:
   `npm install --save-dev mocha chai`

3. Añades en `package.json` un script para ejecutar las pruebas:

```
"scripts": {
     "test": "mocha"
     }
```

4. Escribes tu código en `index.js`.

```
export function sumar(a, b) {
return a + b;
}
```

5. Escribes las pruebas en `test/test.js`.

```
import { expect } from 'chai';
import { sumar } from '../index.js';

describe('Pruebas de sumar', () => {
  it('2+3 = 5', () => {
    expect(sumar(2,3)).to.equal(5);
  });
});
```

6. Ejecutas pruebas desde la raíz del proyecto:

```
npm test
```

Referencias

- ¿Qué es el ciclo de CI/CD?

  https://www.paloaltonetworks.es/cyberpedia/what-is-the-ci-cd-pipeline-and-ci-cd-security