UT3 - DISEÑO Y REALIZACIÓN DE PRUEBAS – PROCEDIMIENTOS DE PRUEBAS Y TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA – PARTE I

RESULTADOS DE APRENDIZAJE ASOCIADOS
3.- Verifica el funcionamiento de programas diseñando y realizando pruebas.
CRITERIOS DE EVALUACIÓN
a) Se han identificado los diferentes tipos de pruebas.
b) Se han definido casos de prueba
c) Se han identificado las herramientas de depuración y prueba de aplicaciones ofrecidas por el entorno de desarrollo.
d) Se han utilizado herramientas de depuración para definir puntos de ruptura y seguimiento.
e) Se han utilizado las herramientas de depuración para examinar y modificar el comportamiento de un programa en tiempo de ejecución.
f) Se han efectuado pruebas unitarias de clases y funciones.
g) Se han implementado pruebas automáticas.
h) Se han documentado las incidencias detectadas.

UT3 - DISEÑO Y REALIZACIÓN DE PRUEBAS – PROCEDIMIENTOS DE PRUEBAS Y TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA – PARTE I

Índice de contenido

1.- Contenidos

Descubriremos en este módulo la aplicación de variadas técnicas en la creación de casos de prueba. Abordaremos la utilización de herramientas de depuración para establecer puntos de quiebre y analizar variables en tiempo real durante la ejecución de un programa. Asimismo, nos sumergiremos en el uso de ***JUnit*** como método para elaborar pruebas unitarias específicamente diseñadas para entornos Java.

2.- Introducción

La evaluación y comprobación de un producto de software antes de su implementación son aspectos fundamentales en las pruebas de software.

Su principal objetivo es someter a examen la aplicación construida, integrándose de manera inherente en diversas fases del ciclo de vida del software, en el marco de la ingeniería del software.

Siguiendo la perspectiva realista de ***Edsger Dijkstra***, quien afirmaba que realizar pruebas exhaustivas a un programa es prácticamente imposible debido a su elevado costo, nos enfrentamos a un desafío inherente en el proceso de desarrollo.

Investiga…

¿Quién fue *Edsger Dijkstra* y por qué tienen tanta importancia sus aseveraciones?

La ejecución de pruebas en un sistema implica una serie de etapas ordenadas:

- Planificación de pruebas
- Diseño y creación de casos de prueba
- Definición de procedimientos de prueba
- Ejecución de pruebas

- Registro de resultados obtenidos
- Documentación de errores identificados
- Depuración de errores
- Elaboración de informes sobre los resultados obtenidos

3.- Procedimientos de pruebas y técnicas de diseño de casos de prueba

Un **procedimiento de prueba** es la definición del objetivo que desea conseguirse con las pruebas, qué es lo que va a probarse y cómo.

El objetivo de las pruebas no siempre es detectar errores. Muchas veces lo que se persigue es que el sistema ofrezca:

- **Un rendimiento determinado**
- **Que la interfaz tenga una apariencia y cumpla unas características determinadas.**
- **Etc.**

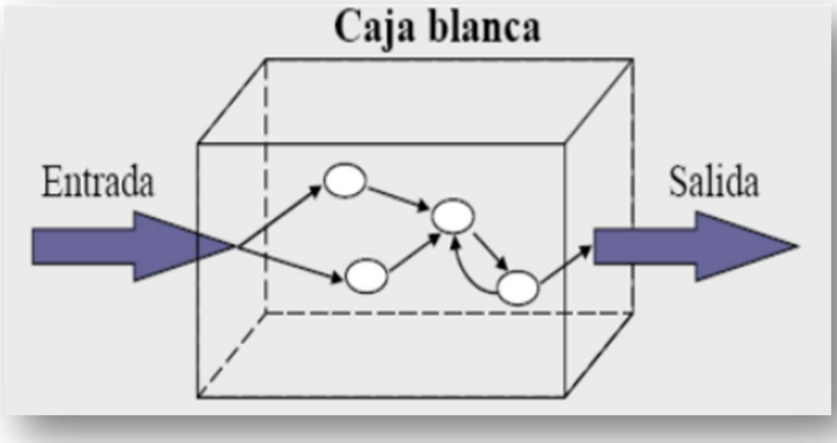
¿La ausencia de errores en las pruebas significa que el software ha superado esas pruebas? No tiene por qué, pues hay muchos parámetros en juego.

Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas o enfoques:

- **Prueba de caja blanca (White Box Testing)**
  - **Definición:** Esta técnica de prueba se centra en la estructura interna del código. El *tester* tiene acceso completo al código fuente y algoritmos implementados en el software.

- **Objetivo:** Evaluar la lógica interna del programa, la ejecución de rutas de código, la estructura de datos y asegurarse de que todas las instrucciones se ejecuten correctamente.

- **Métodos:** Se utilizan técnicas como pruebas de bucles, pruebas de camino, y pruebas de condiciones para garantizar la exhaustividad de la prueba.

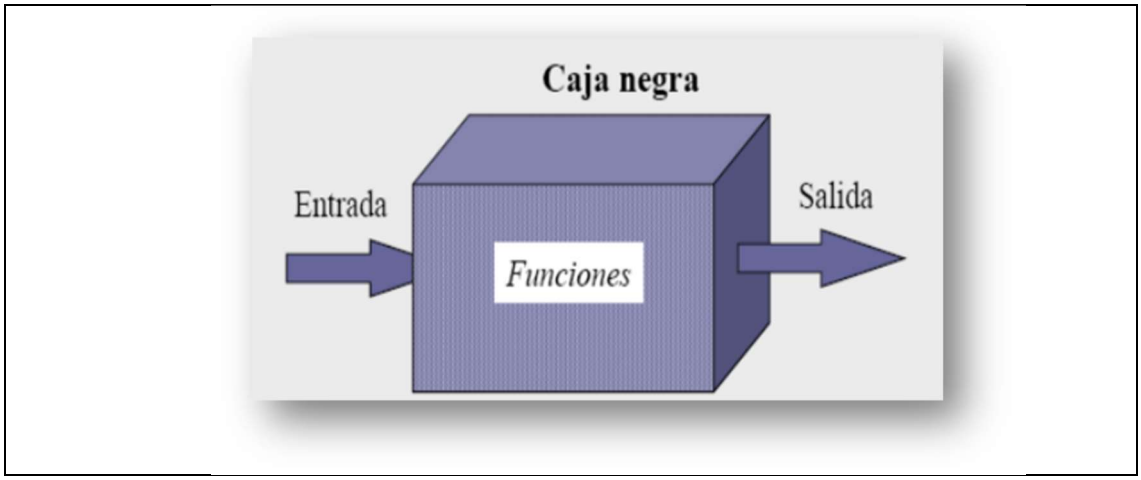


- **Prueba de caja negra (Black Box Testing)**

  - **Definición:** En contraste, la prueba de caja negra no requiere el conocimiento interno del código fuente. El *tester* evalúa la funcionalidad del software sin conocer la implementación interna.

  - **Objetivo:** Enfocada en la entrada y salida del sistema, se centra en probar las interfaces y la funcionalidad general del software.

  - **Métodos:** Los casos de prueba se diseñan con base en requisitos funcionales y especificaciones del software. Se prueba cómo reacciona el sistema ante diferentes entradas sin conocer cómo se procesan internamente.



No siempre tienen que ser los programadores los que hacen las pruebas, puede ser personal externo el que realice esta tarea.

Pensemos un poco…

¿Por qué piensas que puede ser buena idea que una empresa externa pruebe tu software? ¿Qué ventajas podría tener esto?

En los planes de pruebas (es un documento que detalla en profundidad las pruebas que se vayan a realizar), generalmente, se cubren los siguientes aspectos:

1. **Introducción**. Breve introducción del sistema describiendo objetivos, estrategia, etc.
2. **Módulos o partes del software por probar**. Se detallan cada una de esas partes o módulos.
3. **Características del software por probar**. Tanto individuales como conjuntos de ellas.
4. **Características del software que no ha de probarse**.
5. **Enfoque de las pruebas**. En el que se detallan, entre otros, las personas responsables, la planificación, la duración, etc.
6. **Criterios de validez o invalidez del software**. En este apartado, se registra cuando el software puede darse como válido o como inválido especificando claramente los criterios.

7. **Proceso de pruebas**. Se especificará el proceso y los procedimientos de las pruebas por ejecutar.
8. **Requerimientos del entorno**. Incluyendo niveles de seguridad, comunicaciones, necesidades de hardware y software, herramientas, etc.
9. **Homologación o aprobación del plan**. Este plan deberá estar firmado por los interesados o sus responsables.

3.1.- Tipos de pruebas: funcionales, estructurales y regresión

Existen muchas categorías de pruebas por realizar. A continuación, se repasan las más frecuentes:

Pruebas funcionales → buscan que los componentes software diseñados cumplan con la función con la que fueron diseñados y desarrollados.

Ejemplos:

Compras en Línea: • **Objetivo:** Confirmar que los usuarios pueden realizar compras con éxito. • **Pasos:** Seleccionar producto, realizar pago, verificar generación de compra y actualización de inventario.
Registro de Usuarios: • **Objetivo:** Verificar que los usuarios pueden registrarse correctamente. • **Pasos:** Introducir datos, confirmar el mensaje de registro exitoso, verificar el almacenamiento de datos.
Calculadora: • **Objetivo:** Asegurar que las operaciones matemáticas básicas funcionen correctamente. • **Pasos:** Ingresar números, realizar operaciones, verificar resultados precisos.

Pruebas no funcionales → son aquellas pruebas más técnicas que se realizan al sistema. Estas son de caja negra puesto que nunca se examina la lógica interna de la aplicación.

Ejemplos:

Prueba de Rendimiento: • **Objetivo:** Evaluar la capacidad del sistema para manejar carga y determinar su velocidad y escalabilidad. • **Ejemplo:** Realizar pruebas de carga para simular múltiples usuarios y verificar la respuesta del sistema bajo condiciones de alta demanda.
Prueba de Seguridad: • **Objetivo:** Identificar vulnerabilidades y evaluar la resistencia del sistema a posibles amenazas. • **Ejemplo:** Realizar pruebas de penetración para detectar posibles brechas de seguridad y evaluar la robustez de las medidas de protección.
Prueba de Compatibilidad: • **Objetivo:** Verificar que el software funciona correctamente en diferentes entornos y dispositivos. • **Ejemplo:** Probar la aplicación en diferentes navegadores web, sistemas operativos y dispositivos para garantizar la compatibilidad.

Pruebas estructurales → Examinan de forma más detallada la arquitectura de la aplicación.

Ejemplos:

Prueba de Integración: • **Objetivo:** Evaluar la interacción entre módulos o componentes del software. • **Ejemplo:** Verificar la comunicación correcta entre diferentes partes del código, probando la integración de módulos específicos.
Prueba de Dependencias: • **Objetivo:** Validar las dependencias externas y internas del software. • **Ejemplo:** Asegurarse de que las bibliotecas externas se estén utilizando correctamente y de que las dependencias internas estén actualizadas y funcionando.
Prueba de Condiciones: • **Objetivo:** Evaluar todas las condiciones lógicas en el código. • **Ejemplo:** Verificar que todas las combinaciones posibles de condiciones (if, switch) se prueben para asegurar que cada ruta lógica sea evaluada.
Prueba de Bucles: • **Objetivo:** Evaluar la correcta ejecución y salida de bucles. • **Ejemplo:** Probar bucles con entradas válidas e inválidas, así como casos límite, para asegurar que el comportamiento del bucle sea el esperado.

Pruebas de regresión o pruebas repetidas → No suele probarse lo que ya se ha probado, pero, en el caso de que el software haya sido modificado, generalmente se realiza este tipo de pruebas.

Este error intenta descubrir si existe algún error tras las modificaciones o si se encuentra algún problema que no se había descubierto previamente.

Sólo se realizarán estas pruebas en el caso de que haya una modificación de software.

3.2.- Pruebas de caja blanca

También se les conoce como ***pruebas estructurales o de caja de cristal.*** Se basan en el minucioso examen de los detalles procedimentales del código de la aplicación.

Mediante esta técnica se pueden obtener 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.

Existen algunas clases de pruebas de este tipo como, por ejemplo:

- Pruebas de cubrimiento
- Pruebas de condiciones
- Pruebas de bucles
- Etc.

3.1.1.- Pruebas de cubrimiento

En este tipo de pruebas, el objetivo es ejecutar, al menos una vez, todas las sentencias o líneas del programa.

En ocasiones, es imposible cubrir el 100% del código porque puede haber condiciones que nunca se cumplan:

```
if (a > 20 && a < 10) { …. }
```

Investiguemos un poco…

¿Qué es la complejidad ciclomática?

Para realizar las pruebas, habrá que generar el suficiente número de casos de prueba para poder cubrir los distintos caminos independientes del código. En cada condición, deberá cumplirse en un caso y en otro no.

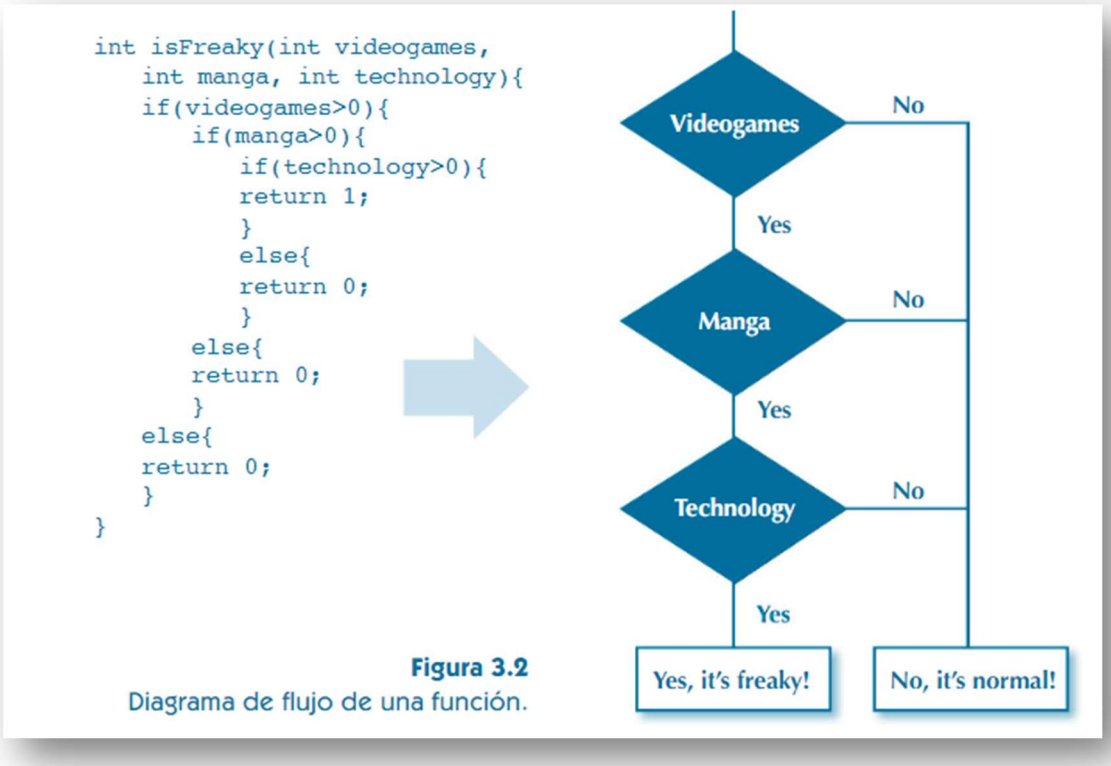


Figura 3.2
Diagrama de flujo de una función.

Para hacer la prueba de cubrimiento se debe:

- Crear casos de pruebas para ejecutar las líneas de código al menos una vez y, por lo tanto, los conjuntos de datos de pruebas:

{videogames=0, manga=0, technology=0}
{videogames=1, manga=0, technology=0}
{videogames=1, manga=1, technology=0}
{videogames=1, manga=1, technology=1}

En esta función no habría muchos caminos (o posibilidades), pero en un programa más extenso el número de combinaciones sería muy elevado).

¡Vamos a probarlo!

Tenemos el código de ejemplo (En JAVA) para hacerlo. Descárgalo del aula virtual y probemos todos los casos de uso.

3.1.2.- Pruebas de condiciones

En este caso, se necesitarán varios casos de prueba. En una condición, puede haber varias condiciones simples y habrá que generar un caso de pruebas por cada operando lógico o comparación.

La idea es que, en cada expresión, se cumpla un caso y en otro no.

Ejemplo:

```
if (videogames=1 && manga=1 && technology=1){ freaky = 1}
```

En el caso anterior, deberán comprobarse todas y cada una de las combinaciones de las tres variables anteriores. En esta ocasión, son variables, pero podrían ser otro tipo de expresiones más complejas.

Pero… ¿qué diferencia hay entre las pruebas de cubrimiento y las pruebas de condiciones?

Mientras que las pruebas de cubrimiento se centran en la ejecución del código y en garantizar que todas las partes hayan sido probadas, las pruebas de condiciones se centran en evaluar las diferentes condiciones lógicas del código para garantizar que todas las posibilidades sean consideradas durante las pruebas. Ambos enfoques son complementarios y se pueden utilizar de manera conjunta para lograr una cobertura más completa del código y una mayor confianza en la calidad del software.

3.1.3.- Pruebas de bucles

Los bucles son estructuras que se basan en la repetición, por lo tanto, la prueba de bucles se basará en la repetición de un número especial de veces.

- En el caso del ***bucle simple***, los casos de prueba deberían contemplar lo siguiente:
  a) Repetir el máximo, máximo-1 y máximo +1 veces el bucle para ver si el resultado del código es el esperado.
  b) Repetir el bucle cero y una vez.
  c) Repetir el bucle un número determinado de veces.

- En el caso de ***bucles anidados***, existirán bucles internos y externos. Sería bueno realizar la prueba de bucles simple para los bucles internos ejecutando el bucle externo un número determinado de veces y, luego, realizar la prueba contraria. El bucle interno se ejecuta un número determinado de veces y el externo se prueba con las pruebas anteriores de bucle simple.

En el aula dispones de ejemplos para ambos casos.

3.2.- Pruebas de caja negra

Entre las pruebas de caja negra (aquellas que simplemente prueban la interfaz sin tener en cuenta el código), pueden citarse las siguientes:

- Pruebas de clases de equivalencia de datos.
- Pruebas de valores límite
- Pruebas de interfaces

3.2.1.- Prueba de clases de equivalencia de datos

Imagina que se está probando una interfaz y debe generarse código de usuario y una clave.

El sistema dice que el **código de usuario** deberá tener mayúsculas y minúsculas, no puede tener caracteres que no sean alfabéticos y ha de

tener, al menos, 6 letras (máximo 12). Las **contraseñas** tendrán, al menos, 8 caracteres (máximo 10) y contendrán letras y números.

Para testear esta interfaz, lo que debe hacerse es establecer clases de equivalencia para cada uno de los campos. Tendrán que crearse clases válidas y clases inválidas para cada uno de los casos.

Ejemplo:

1.- *Usuario*
- Clases válidas -> "Pelegrino" y "Rocinante"
- Clases inválidas → "marrullero44", "nene", "Portaavionesgigante", "Z&aratrusta" y "Ventajoso12"

2.- *Contraseña*
- Clases válidas: "5Entrevías", "s8brino"
- Clases inválidas: "corta", "muyperoquemuylarguisiisisima" y "999999999"

En el aula dispones de ejemplos para ambos casos.

3.2.2.- Prueba de valores límite

El objetivo es generar valores que puedan probar si la interfaz y el programa funcionan correctamente.

Ejemplo:

Imagina que se accede a una página web de un banco para testearla y la interfaz, cuando va a transferirse una cantidad, comunica: "La cifra máxima que usted puede transferir hoy es de 10.000 euros".

Si quiere probarse dicha interfaz, el *tester* probaría, por ejemplo, valores fuera de rango como -100 o 20.000; también los valores límite como 0,1, 9.999, 10.000 y 10.001.

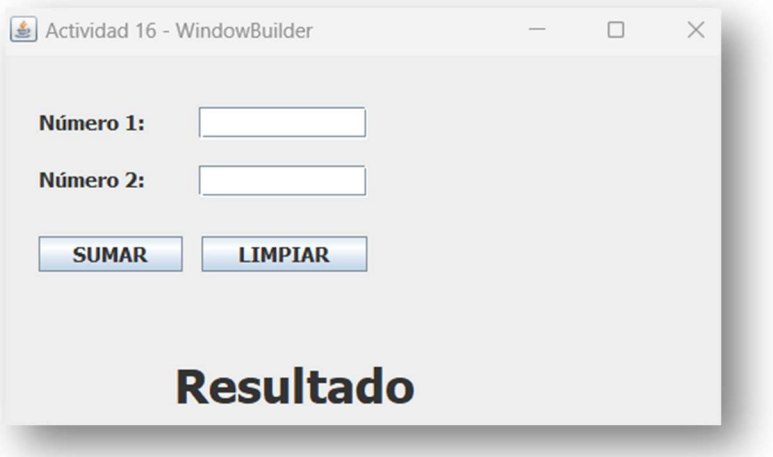
El objeto de esta prueba se halla en que, muchas veces, los programadores se equivocan al establecer los límites en la frontera (se equivocan y ponen < en lugar de <=, por ejemplo).

3.2.3.- Prueba de interfaces

Una interfaz de usuario o GUI (*Graphical User Interface)*, generalmente, se testea con una técnica que se denomina *prueba de interfaces.*

Generalmente, una interfaz es una serie de objetos con una serie de propiedades. Toda esta serie de objetos con sus propiedades en su conjunto formarán la interfaz.

Ejemplo de interfaz:



¿Cómo se testeará una interfaz como esta?

3.2.3.1- Cómo testear una interfaz

Una primera prueba puede consistir en seguir el manual de usuario. El *tester* deberá introducir datos (mejor datos reales que inventados) como si se tratase del propio usuario y comprobar que las salidas proporcionadas son las esperadas.

Si el software pasa esta prueba, entonces, podrá pasar a sufrir un testeo más serio utilizando casos de prueba.

ACTIVIDAD 3.1

En la UT2.- Instalación y uso de entornos de desarrollo (parte 2 de Windows) hicimos una interfaz donde se sumaban dos números.

Esta actividad consiste en construir otra interfaz similar, pero que, en lugar de sumar dos números, realice la división.

Añade diferentes casos de prueba para "testear" que la interfaz hace lo que debe.

3.2.3.2- Testear la usabilidad

Tiene por objeto evaluar si el producto va a resultar lo esperado por el usuario.

Hay que ver y trabajar con la interfaz desde el punto de vista del usuario. Además, en su testeo, deberían usarse datos reales.

Puede ser buena práctica observar cómo el usuario interactúa con el software y obteniendo *feedback* del mismo se podrán detectar aquellas características de este que lo hacen difícil y tedioso de utilizar.

Los tests de usabilidad deben estar integrados en el ciclo de vida de desarrollo del software.

Por ejemplo:

Muchas veces se echan de menos teclas rápidas o combinaciones de teclas, valores por defecto, autocompletar…

Interesante lectura:

https://interfaces.abrilcode.com/doku.php?id=bloque3:usabilidad

Para evaluar la usabilidad de una aplicación, es útil realizar preguntas que aborden diferentes aspectos de la experiencia del usuario. Aquí hay algunas preguntas clave que podrías hacerte para estudiar la usabilidad de una aplicación:

Facilidad de Uso:

- ¿Los usuarios encuentran fácilmente las funciones principales?
- ¿El diseño de la interfaz es intuitivo?
- ¿La navegación entre las pantallas es clara y sencilla?
- ¿Hay elementos visuales que indican la función de los controles?

Eficiencia:

- ¿Los usuarios pueden realizar tareas comunes de manera eficiente?
- ¿Hay atajos de teclado o funcionalidades que mejoren la velocidad de uso?
- ¿Se minimiza la cantidad de clics o acciones necesarios para completar una tarea?

Consistencia:

- ¿La interfaz mantiene una consistencia visual y de comportamiento en todas las secciones de la aplicación?
- ¿Se utilizan los mismos términos y conceptos en todo el sistema?

Retroalimentación y Confirmación:

- ¿La aplicación proporciona retroalimentación inmediata sobre las acciones del usuario?
- ¿Existen confirmaciones visuales o mensajes de error claros cuando se realiza una acción?

Flexibilidad y Control del Usuario:

- ¿Los usuarios tienen control sobre la aplicación?
- ¿Pueden personalizar la configuración según sus preferencias?
- ¿Hay opciones para deshacer acciones?

Legibilidad y Visibilidad:

- ¿La información en la interfaz es fácil de leer?
- ¿El contraste y el tamaño de texto son adecuados?
- ¿Los elementos importantes están resaltados de manera clara?

Manejo de Errores:

- ¿Cómo maneja la aplicación los errores y situaciones inesperadas?
- ¿Se proporcionan mensajes de error comprensibles y soluciones sugeridas?

Autocompletado y Sugerencias:

- ¿La aplicación ofrece autocompletado o sugerencias para facilitar la entrada de datos?
- ¿Las funciones predictivas son útiles y precisas?

Feedback de Usuarios:

- ¿Se recopila y considera el feedback de los usuarios?
- ¿Se realizan pruebas con usuarios reales para obtener perspectivas valiosas?

En el aula dispones de ejemplos para ambos casos.

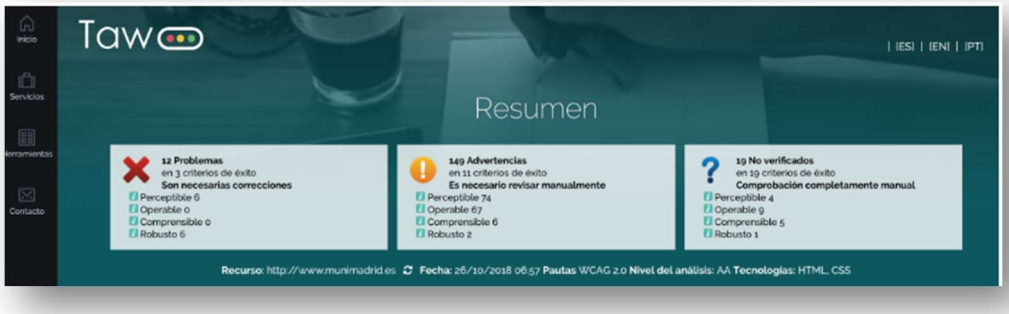
3.2.3.2- Testear la accesibilidad

Un software es accesible cuando el programa o aplicación se adecua a los usuarios con discapacidad. Éstos pueden realizar su trabajo de forma efectiva y sin ninguna contrariedad.

Hay que tener en cuenta que, en ocasiones, hay estándares y requerimientos preestablecidos de accesibilidad que el software ha de cumplir.

Ejemplos:

Test de accesibilidad Web (TAW) → https://www.tawdis.net/



Test accesibilidad móvil →

- **https://www.w3.org/WAI/standards-guidelines/mobile/es**
- **https://www.accesibles.org/herramienta-para-medir-la-accesibilidad-en-android-test-de-accesibilidad/**
- **https://es.slideshare.net/LisandraArmasguila/pruebas-de-accesibilidad-en-aplicaciones-mviles**

4.- Referencias bibliográficas

- Moreno Pérez, J.C. *Entornos de desarrollo.* Editorial Síntesis.
- Ramos Martín, A. & Ramos Martín, M.J. *Entornos de desarrollo*. Grupo editorial Garceta.