

+Instituto Superior de Formación Técnica Nº 151



Carrera: Analista de Sistemas

2 Año. Algoritmos y Estructuras de Datos II

Trabajo Práctico Nº 7	Unidad 7
Modalidad: Semi-Presencial	Estratégica Didáctica: Trabajo Grupal.
Metodología de Desarrollo: acordar	Metodología de Corrección: acordar docente
Carácter de Trabajo: Obligatorio – con Nota	Fecha Entrega: A confirmar por el Docente.

Unidad 7 - Trabajo Práctico - Testing & CMake & GTest

Desarrollo Teórico

Testing

1. Que entiende por Proceso de Testing?. ¿Que Busca?
2. Dar ejemplos de Pruebas de Caja Negra y Blanca.
3. Describir los Distintos niveles de Pruebas y dar ejemplos de estos niveles.
4. Describir (brevemente) los distintos Tipos de Testing.
5. Relacionar los Principios de Pruebas y las Directrices.
6. ¿Que es un Plan de Pruebas? Dar un ejemplo.

CMake

7. Para que sirve CMake
8. v
9. Que es el CMakeLists.Txt
10. Describir las Instrucciones de Tunnelización y Compilación para generar Exe's.

GTest

11. Que es un Framework de Prueba?
12. Que es una Prueba Unitaria en GTest?
13. Que “Macros” tenemos y podemos usar en GTest?
14. Dar un ejemplo de Equal y Assert.

Desarrollo Práctico “**Drone IBag151**”

Problema) Una empresa de Reparto de Mercadería no encarga realizar una Aplicación que permita manejar un Dron, tenemos una interfaz que controla y captura automáticamente los datos x consola y la manda al Drone, por lo que cada “Texto” que se Imprima en Consola se “Envia” al Drone y lo ejecuta, por este motivo, necesitamos que el **Drone “IBag151”** pueda, Despegar, Aterrizar, Elevar, Bajar, girar derecha, girar Izquierda, Descargar el Paquete, sacar foto de la Recepción y notificar la Entrega, acelerar y frenar.

Se pide:

1. Crear en POO el Drone y su Interfaz que Implemente las actividades antes descritas.
2. Realizar en DrawIO el Diagrama de Clases Correspondiente.
3. Implementar los Conceptos Vistos en las Unidades anteriores como “Directrices de Buen Diseño”
4. Se Deberán hacer pruebas Unitarias por cada actividad para probar que funcione correctamente, las mismas serán de forma Incremental, debiendo en lo posible escribir la prueba Primero y el Código después a fin de que pase la Prueba.
- 5 Implementar lo Anterior utilizando: VSCode, CMake, GTest, C++ 11 (o Superior), SOLID, GRASP, Unit Test, UML, Code Clean, Clean Architecture y Refactoring,

Nota: El presente Trabajo se Continuará en la Unidad de TDD (Test Driven Development).

La Cátedra.

Desarrollo Teórico

Testing

1. Que entiende por Proceso de Testing?. ¿Que Busca?

El testing de software, que surgió alrededor de 1960 durante la crisis del desarrollo de software, es una disciplina de la ingeniería de software. Se utiliza para identificar y corregir defectos en el software, mejorando así su calidad. A diferencia de lo que algunos pueden pensar, el testing no es una actividad que se realiza únicamente al final del desarrollo de una aplicación. De hecho, debe llevarse a cabo durante todo el proceso de desarrollo, convirtiéndose en una metodología esencial para validar cada etapa del desarrollo de una aplicación.

El objetivo principal del testing (lo que busca) es detectar fallas en el software, pero también sirve para aumentar la confianza en la calidad del software, proporcionar información útil a los programadores y descubrir oportunidades de mejora.

En resumen, el testing de software es una práctica crucial en el desarrollo de software que ayuda a garantizar que el producto final funcione como se espera y cumpla con las necesidades del usuario.

2. Dar ejemplos de Pruebas de Caja Negra y Blanca.

Las pruebas de caja negra, que pueden conocerse también como pruebas funcionales o black box testing, son un tipo de pruebas de software en los que importa verificar la entrada o salida de datos del sistema, sin que me importe la estructura interna del software probado. Las de caja blanca (también conocidas como pruebas de caja de cristal o pruebas estructurales), son pruebas que se centran en la estructura interna de un software, con énfasis en el que código fuente que lo compone.

Ejemplos de prueba de caja negra:

- **Técnica de partición equivalente:** esta técnica divide el dominio de entrada de un programa en clases de equivalencia de datos desde las cuales se pueden seleccionar casos de prueba. Por ejemplo, si una función acepta un número entre 1 y 100, puedo probar con un número en el rango, un número por debajo del rango y un número por encima del rango.
- **Prueba de envío de correo electrónico al registrarse una transacción:** se realiza para comprobar si el sistema envía correctamente correos electrónicos cuando se hacen algunos determinados tipos de transacciones o acciones.
- **Pruebas de valores límite:** se introducen valores que estén en el límite del rango de valores que el sistema puede procesar.

Ejemplos de pruebas de caja blanca:

- **Técnica de cobertura de sentencia:** es una técnica cuyo principal objetivo es asegurarse de que todas las sentencias del código se ejecutaron, al menos una vez durante las pruebas. La prueba de cobertura de sentencia es importante porque ayuda a identificar las sentencias que no fueron ejecutadas. Las sentencias no ejecutadas pueden ser un signo de código muerto o de secciones de código que no se utilizan. También puede ser un signo de que algunas funcionalidades del programa no se probaron adecuadamente.
- **Técnica de cobertura de decisión:** esta técnica se utiliza para asegurar que todas las posibles opciones en una estructura de control fueron probadas.
- **Técnicas de camino de datos:** este tipo de pruebas se realizan para verificar que las variables en el software se definen, utilizan y manipulan correctamente. Por ejemplo, si tengo una función que utiliza variables para almacenar y manipular datos, me interesa probar que diferentes entradas y condiciones me aseguran que las variables se manejan correctamente.

3. Describir los Distintos niveles de Pruebas y dar ejemplos de estos niveles.

Los cuatro niveles de pruebas son, la prueba unitaria, la de integración, la del sistema y la de aceptación. Las pruebas unitarias son pruebas que se ejecutan en el nivel más bajo del código y se centran en probar piezas individuales de una aplicación, su propósito es validar que cada unidad del software se comporte de la manera que fue diseñada. Las de integración tratan de probar las conexiones entre los módulos de un sistema de información, lo que se trata es de encontrar las fallas entre las unidades acopladas. Las pruebas de sistema suben un nivel de abstracción, con ellas se evalúa el cumplimiento de los requisitos especificados en los requerimientos. Estos pueden ser de varios tipos, incluyendo requerimientos funcionales, no funcionales, de sistema, de usuario, etc. Finalmente, el nivel más alto de abstracción de pruebas, son las pruebas de aceptación, también conocidas como las User Acceptance Testing(UAT). El objetivo principal de las pruebas de aceptación es determinar si el software a entregar cumple con las necesidades y/o requerimientos de las empresas y sus usuarios.

Ejemplos de elementos probados en estos niveles:

- Módulos, funciones, clases, estructuras, sentencias de control se prueban en las pruebas unitarias.
- Los mismos elementos que mencione en el anterior apartado se prueban en las pruebas de integración, la diferencia es que se prueba sus niveles de cohesión, acoplamiento y la efectividad de la interacción entre ellos.
- En el nivel de pruebas de sistemas se pueden probar aplicaciones completas.
- Por ejemplo, si estoy desarrollando una aplicación de comercio electrónico, puedo realizar pruebas de aceptación para verificar si los usuarios pueden realizar pedidos, buscar productos, realizar pagos, etc., de manera eficiente y efectiva. La idea es ver si los requerimientos están alcanzados.

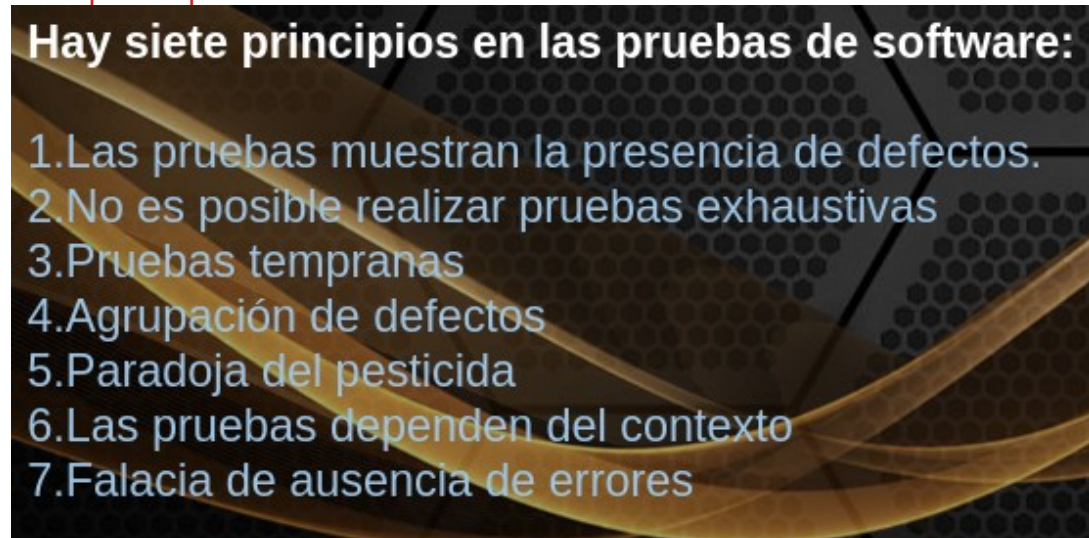
4. Describir (brevemente) los distintos Tipos de Testing.

Las pruebas unitarias se enfocan en probar piezas individuales de una aplicación para validar su comportamiento. Las pruebas de integración buscan encontrar fallas entre las unidades acopladas. Las pruebas de regresión verifican que los cambios no afecten negativamente las funciones existentes. Las pruebas de humo son pruebas superficiales para asegurarse de que las funciones críticas funcionan correctamente. Las pruebas alfa y beta son pruebas de aceptación realizadas antes de entregar un producto al cliente.

Una prueba de sistema es una prueba de caja negra que se enfoca en la entrada y salida de datos. Las pruebas de estrés llevan a la aplicación al peor escenario posible. Las pruebas de rendimiento evalúan el rendimiento del sistema ante una carga dada. Finalmente, las pruebas orientadas a objetos verifican y validan un software escrito bajo el paradigma de la orientación a objetos.

5. Relacionar los Principios de Pruebas y las Directrices.

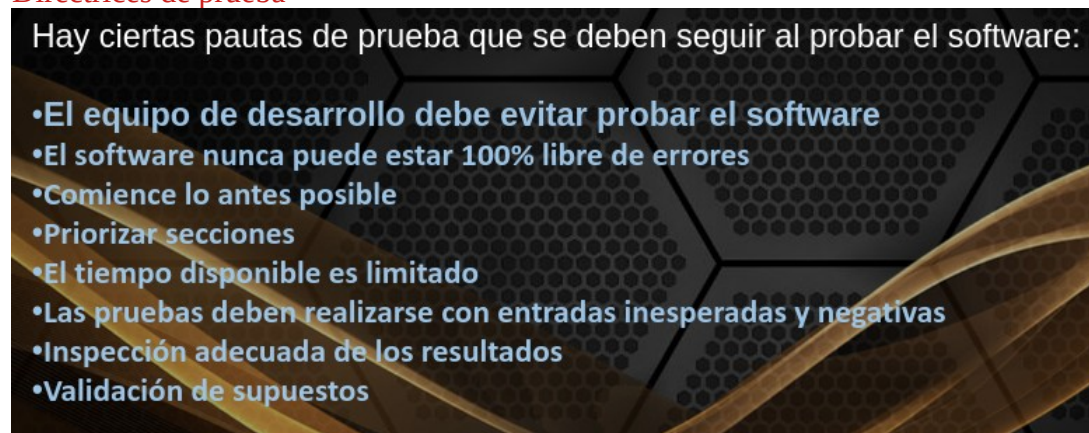
Principios de pruebas



Hay siete principios en las pruebas de software:

1. Las pruebas muestran la presencia de defectos.
2. No es posible realizar pruebas exhaustivas
3. Pruebas tempranas
4. Agrupación de defectos
5. Paradoja del pesticida
6. Las pruebas dependen del contexto
7. Falacia de ausencia de errores

Directrices de prueba



Hay ciertas pautas de prueba que se deben seguir al probar el software:

- El equipo de desarrollo debe evitar probar el software
- El software nunca puede estar 100% libre de errores
- Comience lo antes posible
- Priorizar secciones
- El tiempo disponible es limitado
- Las pruebas deben realizarse con entradas inesperadas y negativas
- Inspección adecuada de los resultados
- Validación de supuestos

Tanto las directrices de prueba como los siete principios fijan normas para el testing. Da la impresión de que los principios son más bien un enfoque teórico; es

decir, plantean un amplio marco general de ideas y conceptos que deben tenerse en cuenta cuando se realizan pruebas de software. En cambio en las directrices apuntan a un enfoque más práctico de las pruebas de software, yendo a detalles más concretos y específicos de las mismas.

En resumen los principios apuntan a un marco teórico de las pruebas de software, mientras que las directrices apuntan a un ámbito pragmático y realista de las mismas.

6. ¿Que es un Plan de Pruebas? Dar un ejemplo.

Un plan de pruebas es un documento integral que recopila toda la información crucial de nuestro proyecto de pruebas. Comienza detallando la estrategia a seguir y los tipos de pruebas a realizar. Luego, establece los criterios de aceptación, reanudación y suspensión, proporcionando una clara dirección para el proceso de pruebas.

Para prepararse para cualquier eventualidad, el plan también identifica posibles riesgos y establece planes de contingencia. Además, se detallan aspectos operativos como la metodología de pruebas, los roles y responsabilidades del equipo de pruebas, el alcance del proyecto, el plan de trabajo, la preparación de los datos y los entornos de prueba.

En resumen un plan de pruebas es documento que recopila todas las estrategias y procedimientos de prueba para un proyecto de Software.

Método para realizar un plan de pruebas en 5 pasos

1. Analizar los requerimientos de desarrollo de software
2. Identificar las funcionalidades nuevas a probar
3. Identificar las funcionalidades de sistemas existentes que deben probarse
4. Definir la estrategia de pruebas
5. Definir os criterios de inicio, aceptación y suspensión de pruebas

Siguiendo este plan, aca un esbozo de lo podría ser un plan de pruebas:

1. **Creo el nombre o identificador del plan de pruebas.** Por ejemplo, “Plan de pruebas integral Control de desarrollo para el desarrollo de la aplicación Pepito versión 1.0.1”
2. **Analizo los requerimientos de desarrollo de software.** Ejemplo:

Se analizan los requerimientos solicitado por el cliente xxxx xxxx para el desarrollo de la aplicación pepito y se discutió el día 00/00/2023 con el equipo de testing todo lo referente al desarrollo de las pruebas.

3. Identifico las nuevas funcionalidades a probar:

Se probarán en la aplicación Pepito ver 1.0.1 las siguientes funcionalidades:

- Conexión con base de datos tipo mongo db.
- Nuevas funcionalidades referidas a plug in de Chatgpt y Bard
- Se probará el nuevo soporte para las Chromebook
- Nueva interfaz gráfica que se parece a Material de Android

4. Identifico las funcionalidades existentes a probar

- Se probará el módulo existente de órdenes de compra
- Se harán testings exhaustivos sobre el módulo de Proveedores
- Se pondrá énfasis especial en el módulo Clientes que está siendo motivo constante de quejas de los usuarios
- Se realizarán tests de rendimiento de la aplicación en los nuevos modelos con el procesador M3 de la familia de productos Apple, de los que hay considerables quejas por la mala optimización del Hardware.

5. Defino la estrategia de prueba: Se realizarán pruebas de caja negra, pruebas de caja blanca, pruebas de integración, pruebas de sistema y pruebas de aceptación del usuario.

6. Defino criterios de inicio, aceptación y suspensión de pruebas. Ejemplo:

El día 11/11/2023 a las 12:30 horas, el equipo de testing yyyy de la división integral de nuestra empresa, comenzará con el “Plan de pruebas integral Control de desarrollo para el desarrollo de la aplicación Pepito versión 1.0.1”. El testing se hará en el servidor de pruebas de la empresa, durará tres días y si es superado con éxito, la aplicación se distribuirá la nueva versión para ser instalada por nuestros clientes. El criterio de éxito de las pruebas es que el porcentaje de error de las pruebas no sea mayor al 1%, siendo el 1% de fallos el límite tolerable de error en el proceso exhaustivo de testing a realizar. Se suspenderán las pruebas el día 14/11/2023 a las 12:30, si estas son exitosas. Si las pruebas muestran un porcentaje mayor al 5%, se suspende la realización de las pruebas y el software vuelve al equipo de desarrollo para su revisión.

CMake

7. ¿Para qué sirve CMake?

CMake es una herramienta que automatiza el proceso de compilación en proyectos desarrollados en C++. Surgió para satisfacer una necesidad en el mundo de la programación: compilar de manera eficiente y correcta en múltiples plataformas. Anteriormente, compilar un proyecto multiplataforma era muy difícil debido a las diferencias entre las versiones de los diversos compiladores de C++ disponibles y las dificultades que surgían al intentar que diferentes arquitecturas de computadoras respondieran bien a un mismo proyecto.

Cmake también resuelve otro problema de la compilación, que es del proceso de compilación; porque en un proyecto de software, especialmente los grandes, hay muchos archivos de código fuente que deben ser compilados y enlazados juntos para crear el programa final. Hacer esto manualmente puede ser un proceso tedioso y propenso a errores.

En resumen, Cmake sirve para compilar de manera simple y automatizada y para asegurarme que mi compilación sea multiplataforma.

8. Que resuelve?

Antes de la aparición de CMake en el año 2001, existían varias herramientas de compilación que se utilizaban para compilar proyectos multiplataforma, como Make, Autotools y MSVC. Cada una de estas herramientas tiene sus propias fortalezas y debilidades, y ninguna de ellas es perfecta para todos los casos de uso. CMake fue desarrollado para abordar algunas de las limitaciones de estas herramientas y proporcionar una solución de compilación más flexible y fácil de usar. Cmake resuelve principalmente dos problemas:

- 1.) El problema de compilar un proyecto de software y que esa compilación funcione en diferentes dispositivos, sistemas operativos, entornos de desarrollo, diversas arquitecturas de ordenadores, etc.
- 2.) El problema del inmenso aburrimiento y complejidad que traía compilar a mano un proyecto con muchos archivos.

9. ¿Qué es el CmakeLists.Txt?

CMakeLists.txt es un archivo de texto que contiene las instrucciones para compilar, ensamblar y vincular un proyecto de software. Es utilizado por el sistema de construcción CMake para generar el código necesario para construir un proyecto de software.

Cuando creo un nuevo proyecto, CMake genera el archivo CMakeLists.txt automáticamente y lo coloca en el directorio raíz del proyecto. Este archivo define lo que debe hacer el sistema de construcción en ese directorio específico. Las acciones típicas incluyen:

- Crear una biblioteca o un ejecutable a partir de algunos de los archivos de origen en este directorio.
- Agregar una ruta de archivo a la ruta de inclusión utilizada durante la compilación.
- Definir las variables que el sistema de construcción utilizará en este directorio y en sus subdirectorios.
- Generar un archivo, basado en la configuración de construcción específica.
- Localizar una biblioteca que se encuentra en algún lugar del árbol de origen.

10. Describir las Instrucciones de Tunelización y Compilación para generar Exe's.

El proceso para la tunelización y compilación para generar un ejecutable en Cmake, se puede definir así:

1. Defino los objetivos de construcción: en el archivo CmakeLists.txt, defino usando las sentencias `add_executable()` y `add_library()`, los objetivos de construcción que quiero crear
2. Defino las dependencias de los objetivos de construcción: en este segundo paso, debo definir las dependencias de cada objetivo. Se hace con el comando `target_link_libraries()`.
3. Configuro las opciones de compilación: Configuro las opciones de compilación con los comandos `set()` y `target_compile_options()`.
4. Genero el código de compilación: en el próximo paso genero el código de compilación usando el comando `cmake` en la terminal.
5. Compilo los objetivos: Una vez que genere el código de compilación uso el comando `make` para terminar el proceso.

GTest

11. Que es un Framework de Prueba?

Un framework de prueba es un conjunto de herramientas, bibliotecas y directrices que se utilizan para automatizar las pruebas de software. Los frameworks de prueba proporcionan una estructura para organizar y ejecutar pruebas, lo que puede ayudar a mejorar la eficiencia y la eficacia del proceso de pruebas. Algunos ejemplos de frameworks de prueba son:

1. **JUnit**: framework de pruebas unitarias para el lenguaje de programación Java.
2. **Selenium**: framework para realizar pruebas en aplicaciones web a través de una variedad de navegadores y plataformas.
3. **TestNG**: este framework de pruebas esta inspirado en JUnit y NUnit, pero además introduce algunas funcionalidades nuevas que hacen que sea más potente y fácil de usar, como por ejemplo anotaciones, ejecución paralela de pruebas, entre otros.
4. **Mocha**: framework de pruebas de JavaScript que se ejecuta en Node.js y en el navegador, lo que lo hace flexible y fácil de usar con bibliotecas de aserciones.
5. **PyTest**: framework de pruebas para el lenguaje de programación Python. Es muy flexible y se puede utilizar para todo tipo de pruebas, desde unitarias hasta funcionales.
6. **Google Test**: también conocido como gtest, es una biblioteca de pruebas unitarias para el lenguaje de programación C++. Fue lanzada bajo la licencia BSD de 3 cláusulas y se basa en la arquitectura xUnit

12. Que es una Prueba Unitaria en Gtest?

Un prueba unitaria es una prueba que se utiliza para verificar el correcto funcionamiento de una unidad de código, que puede ser una función, un método o una clase. Una prueba unitaria en Gtest es una prueba unitaria que se utilice en ese framework para determinar el correcto funcionamiento de un modulo de un sistema basado en el lenguaje de programación C++.

13. Que “Macros” tenemos y podemos usar en Gtest?

Algunos de los macros más comunes para usar en Gtest son:

1. **TEST(TestSuiteName, TestName)**: Define una prueba individual llamada TestName en la suite de pruebas TestSuiteName1.
2. **TEST_F(TestFixtureName, TestName)**: Define una prueba individual llamada TestName que utiliza la clase de fixture de prueba TestFixtureName.
3. **TEST_P(TestFixtureName, TestName)**: Define una prueba individual parametrizada llamada TestName que utiliza la clase de fixture de prueba TestFixtureName1.
4. **INSTANTIATE_TEST_SUITE_P**(InstantiationName, TestSuiteName, param_generator): Instancia la suite de pruebas parametrizadas TestSuiteName (definida con TEST_P)1.
5. **EXPECT_ y ASSERT_**: Estas macros vienen en pares y se utilizan para verificar el comportamiento del código. Por ejemplo, EXPECT_TRUE(my_condition) verificará si my_condition es verdadero. Si no lo es, generará un fallo no fatal y permitirá que la función actual continúe ejecutándose. Por otro lado, ASSERT_TRUE(my_condition) generará un fallo fatal y abortará la función actual si my_condition no es verdadero.
6. **SUCCEED()**: Genera un éxito.
7. **FAIL()**: Genera un fallo fatal.
8. **ADD_FAILURE()**: Genera un fallo no fatal.
9. **ADD_FAILURE_AT(file_path, line_number)**: Genera un fallo no fatal en el archivo y la línea especificados.
10. **EXPECT_THAT(value, matcher)** y **ASSERT_THAT(value, matcher)**: Verifican que el valor coincida con el matcher.

14. Dar un ejemplo de Equal y Assert.

En el contexto de las pruebas y la ingeniería de software, assert es una declaración que permite verificar si una determinada condición es verdadera o falsa. Si la condición es verdadera, el programa continúa su ejecución normalmente. Sin embargo, si la condición es falsa, el programa normalmente se detiene y genera algún tipo de error o excepción. En el contexto más específico de las pruebas unitarias assert se utiliza para verificar si el resultado de una función o método coincide con el resultado esperado. Si el resultado coincide con el resultado esperado, la prueba pasa. Si no coincide, la prueba falla.

```
#include <gtest/gtest.h>

// Función que quiero probar
int Suma(int a, int b)
{
    return a + b;
}

//Defino la prueba
TEST(TestSuma, TesteaSumaPositivos)
{
    EXPECT_EQ(Suma(1, 2), 3);
    ASSERT_EQ(Suma(5, 7), 12);
}
```

```
int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```