2025

Entorno de Desarrollo

TEMA 2 Y 3

OLIVER BITICA – 1°DAM

Contenido

Leyenda de Colores

- Amarillo: Para conceptos fundamentales, definiciones y puntos clave.
- Verde: Para ventajas, objetivos, propósitos y características positivas.
- Azul: Para tipos, clasificaciones, componentes, estructuras y ejemplos.
- Rojo/Salmón: Para problemas, inconvenientes, limitaciones o advertencias.
- Gris: Para tecnologías específicas, nombres propios, estándares o secciones explícitamente marcadas como "Contenido Prioritario".

Guía Profesional de Herramientas y Prácticas de Calidad en el Desarrollo de Software

1. El Entorno Profesional del Desarrollador: Los IDEs

En el desarrollo de software moderno, la eficiencia y la calidad no son objetivos deseables, sino requisitos fundamentales. La elección de las herramientas adecuadas es una decisión estratégica que impacta directamente en la productividad del equipo, la mantenibilidad del código y la velocidad de entrega. En el corazón del arsenal de un desarrollador se encuentra el Entorno de Desarrollo Integrado (IDE), una plataforma unificada que consolida todas las herramientas necesarias para transformar una idea en una aplicación funcional, robusta y de alta calidad.

1.1. ¿Qué es un Entorno de Desarrollo Integrado (IDE)?

Un **Entorno de Desarrollo Integrado (IDE) es una aplicación de software que proporciona un conjunto completo de herramientas para facilitar el ciclo de vida del desarrollo de software**. Si bien es técnicamente posible construir un programa utilizando componentes aislados, como un editor de texto simple y un compilador ejecutado desde la línea de comandos, este enfoque es fragmentado e ineficiente. El IDE supera estas limitaciones al integrar todas las funcionalidades esenciales en una única interfaz cohesiva, convirtiéndose en el estándar de facto en el ámbito profesional por su capacidad para agilizar los flujos de trabajo y reducir la fricción en el proceso de creación.

1.2. Anatomía de un IDE: Componentes Esenciales y su Impacto

Un IDE moderno es mucho más que un simple editor de código. Es un ecosistema de herramientas diseñadas para trabajar en sinergia, cada una contribuyendo a

mejorar la productividad del desarrollador y la calidad final del software. Sus componentes fundamentales incluyen:

- **Editor de texto**: Un editor de código avanzado que ofrece funcionalidades críticas como el resaltado de sintaxis, que colorea el código según su estructura para mejorar la legibilidad y facilitar la detección inmediata de errores tipográficos o sintácticos.
- **Compilador o Intérprete**: Integra la herramienta necesaria para traducir el código fuente a un formato ejecutable. Dependiendo del lenguaje, puede ser un compilador (que traduce todo el código a lenguaje máquina antes de la ejecución) o un intérprete (que ejecuta el código línea por línea).
- **Depurador (Debugger)**: Una de las herramientas más potentes para garantizar la calidad. Permite ejecutar el programa de forma controlada, paso a paso, inspeccionar el valor de las variables en tiempo real y establecer puntos de interrupción para analizar el estado de la aplicación y diagnosticar errores lógicos complejos.
- **Gestor de proyectos y ficheros**: Proporciona una estructura organizada para administrar todos los archivos, directorios, paquetes y dependencias de un proyecto, simplificando la navegación y la gestión de bases de código complejas.
- **Constructor de interfaces gráficas (GUI Builder)**: Herramientas visuales de "arrastrar y soltar" que permiten diseñar y construir interfaces de usuario (ventanas, botones, formularios) de manera rápida e intuitiva, acelerando significativamente el desarrollo del front-end.
- **Herramientas de ayuda**: Un conjunto de funcionalidades que aumentan la productividad, como el autocompletado de código (sugiere código mientras se escribe), la generación automática de documentación y la automatización de tareas repetitivas como la compilación y el empaquetado.
- **Integración con herramientas externas**: Capacidad de conectarse de forma nativa con otros sistemas cruciales para el desarrollo, como gestores de bases de datos, terminales de comandos, servidores web y, fundamentalmente, sistemas de control de versiones como Git.
- **Plugins o Extensiones**: Un ecosistema de complementos que permite personalizar y extender las capacidades del IDE, adaptándolo a las necesidades específicas de un lenguaje, un framework o un flujo de trabajo particular.

1.3. El Ecosistema de IDEs: Opciones Libres vs. Propietarias

El mercado de los IDEs se divide principalmente en dos categorías según su modelo de licencia: **libres** y **propietarios**. Los IDEs de licencia libre son de uso público, no requieren el pago de una licencia y su código fuente suele estar abierto a la comunidad. Por otro lado, los IDEs propietarios exigen el pago de una licencia para su uso y están controlados por una entidad comercial.

Es importante notar que algunas compañías, como JetBrains, operan con un modelo **freemium**: ofrecen versiones "Community" gratuitas y muy potentes de sus IDEs, mientras reservan funcionalidades avanzadas para sus ediciones "Ultimate" de pago.

IDEs Libres

IDE	Lenguajes soportados	URL
Eclipse	Java, C/C++, Python, PHP, Ruby, JavaScript, Go, Groovy, Perl	https://eclipseide.org/
JetBrains	Java, Kotlin, Python, JavaScript, C++, C#, PHP, SQL, Go, Ruby, Rust, Swift	https://www.jetbrains.com/
NetBeans	Java, PHP, C/C++, JavaScript, Groovy	https://netbeans.apache.org
CodeLite	C/C++, PHP, JavaScript	https://codelite.org/
Microsoft Visual Studio Code	JavaScript, TypeScript (nativo). Extensiones para: C/C++, C#, Python, Go, PHP, Java, Rust	https://code.visualstudio.com/
JDeveloper	Java, SQL, PHP, tecnologías Oracle	https://www.oracle.com/application-development

IDEs Propietarios

IDE	Lenguajes soportados	URL
JetBrains (IntelliJ, PyCharm, etc)	Java, Kotlin, Python, JavaScript, C++, C#, PHP, SQL, Go, Ruby, Rust, Swift	https://www.jetbrains.com/
C++ Builder	C++	https://www.embarcadero.com/
Microsoft Visual Studio	C#, .NET, C++, JavaScript, Python, PHP, Java, Go	https://visualstudio.microsoft.com/es/
Xcode	Swift, Objective-C (requiere pago por publicación en la App Store)	https://developer.apple.com/xcode/

La elección entre un IDE libre y uno propietario tiene implicaciones significativas. Las opciones libres, impulsadas por comunidades activas como VS Code o Eclipse, son ideales para startups, desarrolladores individuales y proyectos de código abierto.

Por otro lado, las soluciones propietarias como Visual Studio Enterprise o las versiones Ultimate de JetBrains se justifican en entornos corporativos donde el soporte técnico garantizado, las herramientas de análisis de rendimiento avanzadas y las integraciones de nivel empresarial son críticas para la misión.

La selección de un IDE adecuado es el primer paso para establecer un entorno de desarrollo profesional, pero para asegurar la fiabilidad del producto final, es imprescindible complementar estas herramientas con un proceso riguroso de garantía de calidad a través de las pruebas.

2. Garantía de Calidad: Fundamentos y Estrategias de Prusas de Software

La creación de software es un proceso complejo propenso a errores humanos en cada una de sus fases, desde la especificación de requisitos hasta la codificación. Por esta razón, las pruebas de software no son una fase opcional, sino una disciplina de ingeniería indispensable dentro del ciclo de vida del desarrollo. Su objetivo trasciende la simple detección de errores; es un pilar fundamental para la verificación del correcto funcionamiento técnico, la validación de que el producto satisface las necesidades del usuario y la garantía de que el software se puede mantener y evolucionar con confianza a lo largo del tiempo.

2.1. Objetivos Clave de las Pruebas: Verificación y Validación

El proceso de pruebas se guía por dos objetivos primordiales que, aunque relacionados, abordan preguntas distintas sobre la calidad del software:

- **Verificación**: Responde a la pregunta: "¿Se está construyendo el sistema correctamente?". Este proceso se centra en asegurar que el software cumple con las condiciones y especificaciones técnicas definidas en cada fase de su desarrollo. Es una evaluación interna que comprueba si los componentes funcionan según lo diseñado.
- **Validación**: Responde a la pregunta: "¿Se está construyendo el sistema correcto?". Este proceso evalúa el producto final para determinar si satisface los requisitos funcionales y cumple con las expectativas y necesidades del usuario. Es una evaluación externa que asegura que el software aporta el valor esperado.

En conjunto, la verificación y la validación garantizan que el producto no solo está libre de defectos técnicos, sino que también es la solución adecuada para el problema que se propuso resolver.

2.2. Características de un Proceso de Pruebas Efectivo

Para que el esfuerzo de pruebas sea verdaderamente eficaz, los casos de prueba deben diseñarse con rigor profesional. Un buen conjunto de pruebas es

automatizable, para poder ejecutarse sin intervención manual; **completo**, cubriendo la mayor cantidad de código posible; **repetible**, para poder ejecutarse múltiples veces con resultados consistentes; **independiente**, de modo que la ejecución de una prueba no afecte a las demás; y **profesional**, tratado con la misma seriedad que el código de producción en términos de calidad y documentación.

Implementar un proceso de pruebas que cumpla con estas características no es un mero ejercicio técnico; desbloquea ventajas estratégicas cruciales para la agilidad y la sostenibilidad del proyecto:

- Fomentan el cambio: Un conjunto sólido de pruebas actúa como una red de seguridad, permitiendo a los desarrolladores refactorizar y mejorar el código con la confianza de que cualquier regresión o efecto secundario no deseado será detectado.
- Simplifican la integración: Al verificar que las distintas partes del código funcionan correctamente en conjunto, las pruebas de integración aumentan la seguridad y reducen los riesgos al combinar módulos.
- Documentan el código: Los casos de prueba sirven como ejemplos prácticos y ejecutables de cómo se debe utilizar una función o un componente, complementando la documentación tradicional.
- Separan la interfaz de la implementación: Las pruebas que validan la interfaz pública de un componente permiten modificar su lógica interna sin temor a romper la funcionalidad, siempre que el comportamiento externo se mantenga.

2.3. Metodologías Estratégicas de Prueba

El diseño de casos de prueba se puede abordar desde diferentes perspectivas estratégicas, cada una con un enfoque particular:

- **Pruebas de Caja Negra (Funcionales)**: Este enfoque trata el software como una "caja negra", centrándose exclusivamente en su interfaz externa y su comportamiento. El probador no necesita conocer la implementación interna; su objetivo es verificar que para una entrada de datos específica, el sistema produce la salida correcta esperada.
- **Pruebas de Caja Blanca (Estructurales)**: En contraste, esta metodología se basa en un conocimiento profundo del código fuente. El objetivo es analizar y probar la estructura lógica interna del software, buscando caminos de ejecución incorrectos, código no utilizado o bucles ineficientes.
- **Enfoque Aleatorio**: Utiliza modelos de las posibles entradas que el programa puede recibir para generar automáticamente un gran volumen de casos de prueba, simulando un uso diverso e impredecible del sistema.

2.4. Tipos de Pruebas a lo Largo del Ciclo de Vida del Desarrollo

Las pruebas se aplican en diferentes niveles de granularidad y en distintas fases del ciclo de vida del desarrollo para garantizar una cobertura completa:

- **Pruebas Unitarias**: Son el nivel más bajo de prueba y se centran en verificar el correcto funcionamiento de las porciones más pequeñas y aisladas de código, como una única función o un método de una clase.
- **Pruebas de Integración**: Una vez que las unidades han sido probadas, estas pruebas comprueban que los diferentes módulos o componentes de software funcionan correctamente cuando se combinan e interactúan entre sí.
- **Pruebas de Validación**: Se realizan para asegurar que el sistema en su conjunto cumple con los requisitos de software definidos. Generalmente, involucran activamente al cliente y se basan en casos de prueba de caja negra para simular el uso real.
- **Pruebas de Regresión**: Se ejecutan después de realizar una modificación en el código (ya sea para corregir un error o añadir una nueva funcionalidad) para asegurar que los cambios no han introducido nuevos defectos en partes del sistema que antes funcionaban correctamente.
- **Pruebas del Sistema**: Verifican el funcionamiento del software completo e integrado en su entorno de producción, incluyendo su interacción con el hardware, el sistema operativo y otros sistemas externos.

Para garantizar la calidad desde la base, es fundamental profundizar en las metodologías que examinan el corazón mismo del software: las pruebas de caja blanca.

3. Análisis Profundo: La Técnica de Pruebas de Caja Blanca

Las pruebas de caja blanca representan un enfoque de ingeniería riguroso para la garantía de calidad, centrado en la estructura interna del código. A diferencia de las pruebas de caja negra, que validan el "qué" hace el software, las pruebas de caja blanca verifican el "cómo" lo hace. Su objetivo principal es asegurar una cobertura exhaustiva de la lógica del programa, garantizando que todas las instrucciones se ejecuten, todas las decisiones se evalúen y todos los caminos lógicos se recorran al menos una vez.

3.1. Criterios de Cobertura de Código

Las pruebas de caja blanca a menudo se conocen como pruebas de cobertura de código, ya que su eficacia se mide por el porcentaje de código que es ejecutado por los casos de prueba. Existen varios criterios para medir esta cobertura, cada uno con un nivel de exigencia mayor:

- **Cobertura de Sentencias**: El criterio más básico. Busca asegurar que cada instrucción o línea de código ejecutable se ejecute al menos una vez.
- **Cobertura de Decisiones**: Va un paso más allá, exigiendo que cada resultado posible de una estructura de control (como un if o un while) se pruebe. Esto significa que cada rama de la decisión (tanto el resultado verdadero como el falso) debe ser recorrida.
- **Cobertura de Condiciones**: Descompone las decisiones complejas en sus elementos individuales. Busca que cada subcondición dentro de una expresión lógica se evalúe tanto como verdadero y falso.
- **Cobertura de Camino**: El criterio más exhaustivo. Su objetivo es probar cada posible camino de ejecución a través de una función, incluyendo la verificación de bucles en tres escenarios clave: sin entrar al bucle, con una sola iteración y con al menos dos iteraciones.

3.2. Diseño de Pruebas Basado en la Complejidad Ciclomática

Para diseñar sistemáticamente un conjunto de pruebas que garantice una cobertura adecuada, se puede utilizar una técnica formal basada en la teoría de grafos y la complejidad ciclomática. El proceso consta de los siguientes pasos:

1. **Creación del grafo de flujo de control**: Se representa visualmente la lógica del código como un grafo, donde los nodos son bloques de código y las aristas representan los posibles flujos de ejecución entre ellos.
2. **Cálculo de la complejidad ciclomática**: Se calcula una métrica que determina el número de caminos linealmente independientes a través del código. Este valor establece el número mínimo de pruebas necesarias para cubrir todas las decisiones lógicas. Se puede calcular de tres maneras:
    - Nº de arcos (las flechas que conectan los bloques de código) - Nº de nodos (los bloques de código) + 2
    - Nº de regiones cerradas del grafo + 1
    - Nº de nodos de condición + 1
3. **Determinación del número de caminos de prueba**: El valor de la complejidad ciclomática indica directamente el número de caminos de prueba que deben diseñarse para lograr una cobertura de decisiones completa.
4. **Definición de casos de prueba para cada camino**: Para cada camino identificado en el grafo, se define un caso de prueba específico, detallando los valores de entrada necesarios para forzar la ejecución por ese camino y el resultado esperado.
5. **Ejecución y comparación de resultados**: Finalmente, se ejecutan las pruebas automatizadas y se comparan los resultados obtenidos con los resultados esperados para validar la correcta implementación de la lógica.

3.3. Caso Práctico: Análisis de Cobertura para la Serie de Fibonacci

Este enfoque técnico se puede ilustrar con el análisis de un programa que calcula la serie de Fibonacci. Tras dibujar el grafo de flujo de control a partir del código fuente, se aplicó el cálculo de la complejidad ciclomática.

El análisis determinó que la complejidad ciclomática era de 6. Este valor indicó la necesidad de identificar y probar 6 caminos de prueba distintos para asegurar una cobertura lógica exhaustiva del programa. En consecuencia, se diseñaron 6 casos de prueba específicos, cada uno destinado a recorrer uno de estos caminos.

La siguiente tabla detalla los 6 casos de prueba diseñados para validar cada camino lógico del algoritmo:

Caso de prueba	Entrada 'salir'	Entrada 'cantidad'	Resultado esperado
Camino 1	"S"	No aplica	Fin del programa
Camino 2	"s"	No aplica	Fin del programa
Camino 3	Distinto de "S" y "s"	4	No muestra resultado
Camino 4	Distinto de "S" y "s"	1	Imprime en pantalla "0"
Camino 5	Distinto de "S" y "s"	2	Imprime en pantalla "0 1"
Camino 6	Distinto de "S" y "s"	3	Imprime en pantalla "0 1 1"

El Camino 3 es un excelente ejemplo de una prueba de caso límite o edge case. El resultado "No muestra resultado" sugiere que el caso de prueba está diseñado para validar un camino lógico específico donde el programa, bajo ciertas condiciones de entrada (cantidad = 4), podría no cumplir con un requisito para entrar en el bucle principal de procesamiento de la serie y, por lo tanto, terminar sin generar una salida. Este tipo de prueba es crucial para asegurar que el programa se comporta de manera predecible incluso con entradas inesperadas o en los límites de su lógica.

La aplicación de técnicas rigurosas como esta es fundamental, pero para garantizar su consistencia y repetibilidad a lo largo de un proyecto y entre diferentes equipos, es crucial formalizar y estandarizar el proceso de pruebas.

4. Formalización del Proceso: Documentación y Estándares de la Industria

Para que las pruebas de software asciendan de ser una actividad ad-hoc a una disciplina de ingeniería predecible y rigurosa, es imprescindible formalizar el proceso. La documentación sistemática y la adhesión a estándares reconocidos por la industria transforman las pruebas en una fase auditable, repetible y gestionable del ciclo de vida del desarrollo, garantizando la calidad de manera consistente.

4.1. Artefactos de Documentación Esenciales

Al igual que el diseño de la arquitectura o la codificación, la fase de pruebas debe generar un conjunto de documentos clave (artefactos) que guíen y registren el esfuerzo realizado:

- **Plan de pruebas**: Documento estratégico que describe el alcance, el enfoque, los recursos y la planificación general de las actividades de prueba.
- **Diseño de pruebas**: Especificación detallada de las pruebas que se realizarán para cada bloque o funcionalidad del sistema, describiendo las características a probar.
- **Casos de prueba**: Definición concreta de las pruebas, tanto de caja negra como de caja blanca, con las entradas, condiciones previas y resultados esperados.
- **Procedimientos de prueba**: Instrucciones paso a paso sobre cómo ejecutar los casos de prueba, incluyendo la configuración del entorno y los criterios de éxito o fracaso.
- **Registro de pruebas**: Un histórico detallado de las pruebas ejecutadas, incluyendo quién las ejecutó, cuándo y cuáles fueron los resultados obtenidos.
- **Informes de incidencias**: Documentos formales que describen los defectos encontrados durante las pruebas, proporcionando información detallada para que los desarrolladores puedan reproducir y corregir el error.

4.2. Adhesión a Estándares de Pruebas de Software

Para guiar y estandarizar la creación de esta documentación y la ejecución del proceso de pruebas, existen diversas normativas y estándares desarrollados por organizaciones profesionales. Seguir estos estándares asegura que el proceso de pruebas es completo, consistente y alineado con las mejores prácticas de la industria. Entre los más relevantes se encuentran:

- **Métrica v3**: Una metodología promovida por la administración pública en España para la planificación, desarrollo y mantenimiento de sistemas de información.

- **BSI (British Standards Institution)**: Organización que publica estándares sobre diversas áreas de la ingeniería, incluyendo el testing de software.
- **Estándares IEEE**: El Institute of Electrical and Electronics Engineers ha publicado estándares clave como el **IEEE 829** (para la documentación de pruebas de software) y el **IEEE 1008** (para las pruebas unitarias).
- **ISO/IEC 29119**: Una norma internacional más reciente que busca unificar los estándares existentes de pruebas de software, proporcionando un marco de trabajo coherente y reconocido a nivel mundial.

5. Conclusión: Hacia un Desarrollo de Software Integral y de Alta Calidad

En definitiva, la construcción de software robusto, mantenible y que aporte un valor real al usuario final es el resultado de un enfoque integral que abarca tanto las herramientas como los procesos. La adopción de un **Entorno de Desarrollo Integrado (IDE)** potente y bien configurado sienta las bases para la productividad y la eficiencia del desarrollador. Sin embargo, las herramientas por sí solas no son suficientes. Es la implementación de un proceso de pruebas riguroso, estructurado y formalizado —desde las pruebas unitarias hasta las de sistema, y aplicando técnicas tanto de caja negra como de caja blanca— lo que verdaderamente garantiza la calidad del producto final. La combinación sinérgica de un entorno de desarrollo profesional y una estrategia de calidad bien definida es la piedra angular para entregar software que no solo funcione, sino que inspire confianza.