Elizabeth León Guzmán, Ph.D.
Jonatan Gómez Perdomo, Ph. D.
Arles Rodríguez, Ph.D.
Camilo Cubides, Ph.D. (c)
Carlos Andres Sierra, M.Sc.

Historias de usuario y pruebas unitarias

Historias y pruebas

Para cualquier aclaración o información adicional puede escribir al correo soportemtic22_bog@unal.edu.co
o radicar solicitud en la mesa de ayuda https://educacioncontinuavirtual.unal.edu.co/soporte

Research Group on Data Mining
Grupo de Investigación en Minería de Datos – (Midas)
Research Group on Artificial Life
Grupo de Investigación en Vida Artificial – (Alife)
Computer and System Department
Engineering School
Universidad Nacional de Colombia

Agenda

Agenda

Metodologías Ágiles de Desarrollo I

Un equipo de desarrollo usa una metodología **ágil** cuando puede crear y/o adaptarse a cambios de manera rápida y eficiente. Es usual en proyectos pequeños (común en emprendimientos *startups*) que tienen alta posibilidad de cambios durante su desarrollo.

En las metodologías ágiles la documentación es simple y fácil de modificar. Es seguida por un equipo con buenas prácticas de desarrollo, y generalmente incluye: pruebas, integración y despliegue continuos (de ser posible, todo automatizado) realizadas en reuniones diarias.

MIDAS

Hechos QUE CONECTAN

UNIVERSIDAD NACIONAL DE COLOMBIA un

Metodologías Ágiles de Desarrollo II

En el desarrollo ágil se definen iniciativas o proyectos a desarrollar. En estas iniciativas, existen unos elementos básicos llamados **Hitos (Epics)** (cumplir una meta épica). Estos hitos son los objetivos del negocio a los cuales se les quiere agregar valor mediante el desarrollo en cuestión.

Los hitos son construidos y sostenidos constantemente por **Historias de Usuario**, las cuales se van actualizando a medida que surgen los objetivos que se buscan cumplir en pequeños periodos de tiempo.

Historias de usuario

Las **Historias de Usuario (User Stories)** son una representación simple e informal, desde la perspectiva del usuario final, de una funcionalidad del software a desarrollar.

La definición de la funcionalidad es escrita en lenguaje natural, de manera simple, casi como si fuera una conversación con el cliente.

La idea de trabajar las funcionalidades de esta manera es articular apropiadamente cada funcionalidad contra los objetivos del negocio, dandole valor a las necesidades e ideas del cliente o usuario final, generando un ambiente colaborativo, y con espacio a soluciones creativas.

Planeación basada en Historias de Usuario

En las metodologías ágiles es normal que existan ciclos de iteración cortos o *sprints* (término tomado de la técnica Scrum que es ampliamente utilizada en estos contextos). Estos ciclos regularmente son de dos semanas.

Al inicio de cada sprint, es común que se tomen las historias de usuario que no se han completado y se definan tareas (o sub-tareas) con una medida del esfuerzo para el cumpliento de las mismas. Luego, se determina cuales sub-tareas se incorporan al sprint.

Esto hace que, iteración a iteración, se esté aportando al negocio al hacer los ajustes requeridos a medida que el proyecto está avanzando.

Agenda

Diseño de Historias de Usuario

Para diseñar una historia de usuario se debe:

- Tener una condición clara de historia de usuario completada, que indique que se cumplieron todas las tareas asociadas a la misma.
- Solucionar las tareas asociadas a la historia de usuario de manera algorítmica (mediante una secuencia finita de pasos bien definidas).
- Considerar el rol del usuario final en el desarrollo, ya que con su retroalimentación se puede comprender mejor el resultado esperado.
- Definir la medida del esfuerzo de las tareas de manera concensuada entre todo el equipo, esto evita sesgos.

Requerimientos Vs. Historias de Usuario

En *Ingeniería de Software* el proceso para levantar requerimientos es bastante formal y detallado. Incluso, existe la división de *Ingeniería de Requerimientos*. Las historias de usuario son más informales, y sujetas a ser ajustadas o modificadas en cualquier momento del proyecto mientras no se hayan cumplido.

Una historia de usuario puede "equivaler" a varios requerimientos (específicados por cada tarea). Sin embargo, los requerimientos solo pueden definirse al inicio del proyecto mientras que las historias de usuario pueden aparecer en cualquier momento. Esto brinda mayor flexibilidad pero introduce riesgos al desarrollo si son usadas en exceso.

MIDAS

Hechos QUE CONECTAN

UNIVERSIDAD NACIONAL DE COLOMBIA un

Recomendaciones para el Diseño I

- Una forma sencilla para escribir historias de usuario es la siguiente:

  `\Como un [usuario], yo [quiero], [para]..."`

- Es bueno discutirla entre todo el equipo para construirla colaborativamente y refinarla hasta tener un acuerdo común, incluyendo al cliente.
- Siempre definir criterios de aceptación, es decir, qué cosas se deben validar como mínimo para considerar que ha sido completada.
- Deben ser pública para todo el equipo, de tal forma que todos conocen lo que espera el usuario, y siempre se tienen claros los objetivos del proyecto a desarrollar.

Recomendaciones para el Diseño II

Ejemplo

- Como un administrador, yo quiero entender el progreso de los agentes de ventas, para generar reportes de desempeño y fallas.
- Como Pepita, yo quiero organizar el inventario de la tienda virtual Unaleña, para tener más control de cuando debo reaprovisionar cierto producto.
- Como Pepito, yo quiero invitar personas al evento de la empresa, para atraer potenciales clientes a mi negocio.
- Como un vendedor, yo quiero tener el listado de clientes y compras de los últimos seis meses, para generar mejores programas de fidelización.

Terminología en Desarrollo de Software

- **Backlog**: Listado de las tareas que se plantearon en un proyecto.
- **Stakeholder**: Persona u organización interesado de alguna manera en el proyecto.
- **Product Owner**: Persona encargada del Backlog. Es quién redacta y prioriza las tareas según las necesidades del proyecto.
- **Developer**: Desarrollador de software, comúnmente con habilidades específicas en ciertas áreas del desarrollo.
- **Bug**: Se refiere a un error en el software. Normalmente se asocia a problemas inesperados.
- **QA Tester**: (Quality Assurance) Es quién se encarga de verificar y asegurar la calidad del software, siguiendo ciertos criterios de aceptabilidad.

Agenda

Agenda

Pruebas de Software

Definición

Una prueba de software es un segmento de código que ejecuta otra parte de código, usando unas entradas determinadas, y verificando que se obtengan las salidas esperadas (pruebas de estado) o una secuencia de pasos esperada (pruebas de comportamiento).

En esencia, las pruebas de software buscan asegurar la calidad del software, verificando en la medida de lo posible los atributos que definen la calidad: robustez, escalabilidad, confiabilidad, amigabilidad, entre otras.

Las pruebas de software se recomienda sean automatizadas, de tal manera que se puedan ejecutar fácilmente en cualquier instante del proyecto.

Ubicación en el Ciclo de Vida del Software

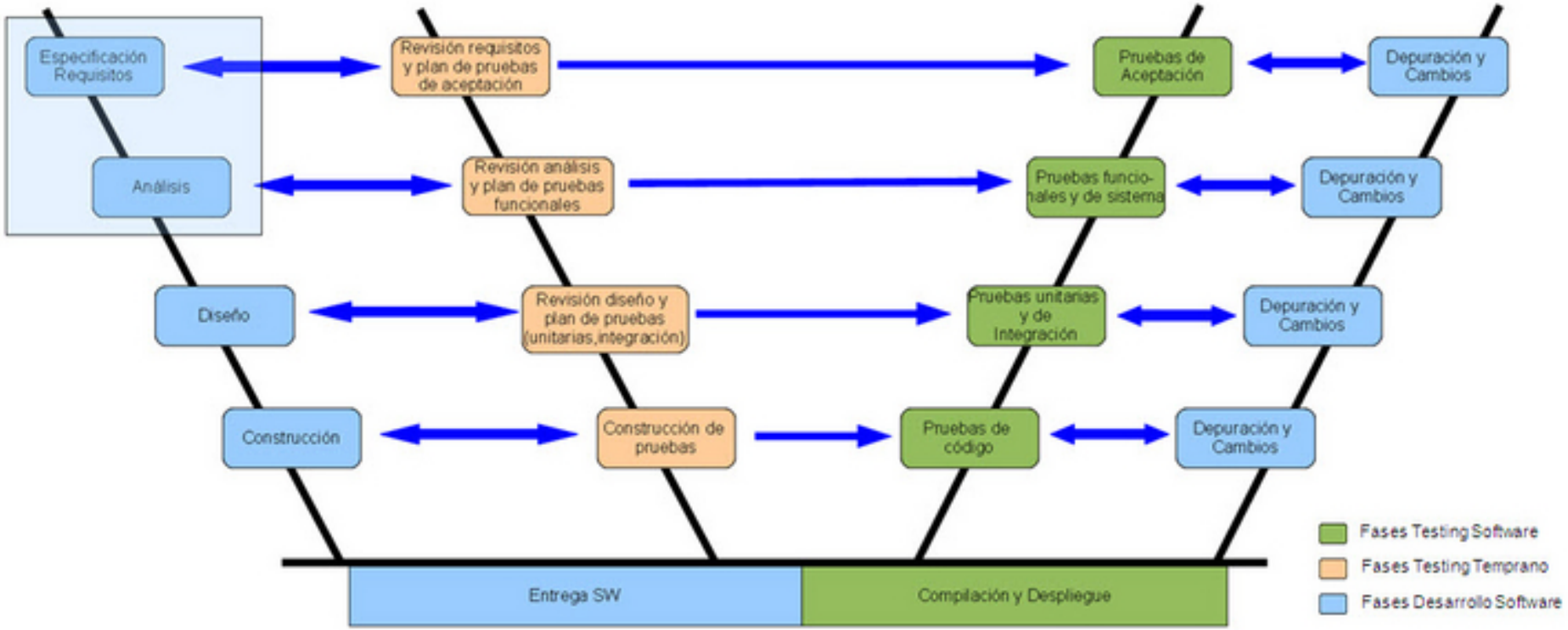

Figure: Imagen tomada de enlace

Tipos de Pruebas

- **Pruebas de Unidad**: Se encargan de probar casos específicos para componentes puntuales dentro del software.
- **Pruebas de Integración**: Permiten validar la comunicación entre el software y los componentes externos, como servidores o bases de datos.
- **Pruebas funcionales**: Se encargan de validar la lógica de negocio dentro de los componentes funcionales del software.
- **Pruebas de Usuario**: Pruebas directas con los usuarios finales, quienes validan que el software cumpla con sus espectativas.
- **Pruebas de Estrés**: Tienen como objetivo saturar la aplicación para conocer su capacidad máxima y el funcionamiento en niveles con alta demanda.

Buenas Prácticas en Pruebas

- Definir las historias de usuario de la forma más clara y exacta posible, evitando cualquier ambigüedad y especificando los criterios de aceptación de una forma rigurosa.
- Definir los posibles casos en los que se debe probar el software, intentando contemplar todos los escenarios que se puedan presentar. Para eso es bueno preguntarse: `¿Cuándo y cómo podría fallar el código?`
- No quedarse sólo con las pruebas evidentes. Se debe pensar en el peor de los casos, el caso más inesperado, en aquellos casos que ocurrirán poco y que serán la fuente de la mayoría de *bugs*.
- Se recomienda que un desarrollador haga unas primeras pruebas de verificación; sin embargo, debe ser un tester quien haga las pruebas de rigor.

Agenda

Pruebas de Unidad (I)

Las pruebas de unidad (o *Unit Tests*) se refieren a las pruebas que se realizan en componentes específicos dentro del desarrollo (normalmente funciones, métodos, o clases). Con ellas se comprueba que un fragmento de código funciona correctamente.

Esto se logra aislando un componente específico y sometiéndolo a varios **casos de prueba**, cada uno con valores distintos que contemplan una exploración de los posibles parámetros que le pueden llegar al componente mínimo de software.

Cada caso de prueba tiene asociada una **salida esperada**. Si el componente no responde como se esperaba, la prueba para ese caso particular falla, e indica que algo salió mal con ese parámetro.

Pruebas de Unidad (II)

Ejemplo

Para probar la función `Math.pow()` se pueden definir los siguientes casos de prueba:

Entrada	Salida
`Math.pow(2, 3)`	`8.0`
`Math.pow(3, 4)`	`81.0`
`Math.pow(0, 0)`	`1.0`
`Math.pow(0, "holi")`	`error`

Diseño de Pruebas de Unidad

Para diseñar pruebas de unidad se debe conocer con precisión qué hace el componente de software en cuestión. El diseño de pruebas de unidad olvida por completo cómo está implementado dicho componente; se centra únicamente en la **entrada** y **salida** de los datos.

Conociendo cómo debe comportarse el componente, se definirán los casos de prueba para cada posible escenario. Pueden surgir preguntas cómo:

- ¿Qué ocurre cuando la entrada es "obvia"?
- ¿Y si la entrada es nula?
- ¿Qué tal si se le pasan datos inesperados?
- ¿Y si intentamos "romper" el código forzando el error?

Casos a Tener en Cuenta

- Hacer pocas entradas "obvias". No será necesario probar tantas veces lo esperado.
- Fijarse en aquellos escenarios donde los valores son nulos, vacíos, o indefinidos.
- Contemplar casos tales como enviar cadenas, listas o mapas vacíos. En el caso de valores numéricos, casos como enviar ceros o valores negativos. Tenga en cuenta que *nulo* es distinto de *vacío*.
- Intentar forzar un error, por ejemplo en el caso de una implementación matemática forzar una indeterminación. ¿Qué se esperaría en esos casos?
- Enviar datos extremadamente grandes, casi al límite. ¿El programa responde para esos datos?

Pensar en los “malos” usuarios

No siempre los usuarios hacen buen uso de las herramientas de software, y esto es ampliamente conocido. Es importante pensar en cómo actuaría un mal usuario, bajo que situaciones puede actuar con el software, y esto que implicaciones puede tener.

No solo basta con intentar muchos casos de prueba, muchas veces se trata de ponerse en el lugar del usuario final, y que ese usuario final puede hacer cosas que para los programadores pueden parecer absolutamente improbables.

En este punto, es importante tener un buen manejo de errores derivados de malas acciones de los usuarios, y acá es donde conceptos como el manejo de excepciones cobra una alta relevancia.

Agenda

Introducción a JUnit

Definición

JUnit es un **Framework** que permite controlar la ejecución de Clases en Java para evaluar su comportamiento mediante pruebas de unidad.

Este framework usa anotaciones para identificar los métodos que especifican una prueba, haciendo fácil su uso y clara la implementación.

JUnit es un proyecto **Open Source**, almacenado en Github. Se puede encontrar el código fuente en este Repositorio.

Definir un test en JUnit - I

Para definir una prueba con JUnit, se recomienda seguir los siguientes pasos:

- Crear la clase `ClaseTest`. Esta será usada únicamente para realizar las pruebas a códigos de otras clases.
- Dentro de la clase para pruebas, se debe crear un método donde se definirá una prueba en específico. Encima del método usar la anotación `@Test`.

```
@Test
public void multiplicacionCerosDeberiaDevolverCero()
```

Definir un test en JUnit - II

- Usar un método *assert* dentro de la función de pruebas, que funciona para declarar que algo debe cumplirse.
- Hay varios tipos de assert, en el siguiente ejemplo se busca comparar que un resultado obtenido por un método sea igual a un resultado esperado.

```
MyClass tester = new MyClass();
double expectedValue = 0.0;
double actualValue   = 0.0;
double deltaResult   = 0.0;
assertEquals(expectedValue, actualValue, deltaResult);
```

Para ver más detalles de tipos de *assert* la siguiente documentación.

Convenciones en JUnit

- Usar el sufijo *Test* para el nombre de las clases que se usarán en pruebas. Ejemplo: `MyClassTest`.
- Como norma general, el nombre del método dentro del test debería ser lo más claro posible y describir al detalle lo que hace. Ejemplo: `multiplicacionDeCerosDeberiaDevolverCero()`.
- Usar la palabra "deberia" en los nombre de los métodos facilita saber qué sucede una vez el método es ejecutado.
- Otro enfoque útil para nombrar los metodos es describir la entrada, qué hace y qué debería devolver, así:
  `dado[ValoresEntrada]cuando[QueHace]deberiaDevolver[ValorEsperado]`

MIDAS

Hechos QUE CONECTAN

UNIVERSIDAD NACIONAL DE COLOMBIA un

Anotaciones para Métodos de Prueba

Anotación	Descripción
`@Test`	Ejecutado como un método de pruebas.
`@Before`	Ejecutado antes de cada test.
`@After`	Ejecutado después de cada test.
`@BeforeClass`	Ejecutado una sola vez antes de todos los test.
`@AfterClass`	Ejecutado una sola vez después de todos los test.
`@Ignore`	Marca el test como deshabilitado.

Comandos para Evaluar

JUnit proporciona métodos estáticos para probar ciertas condiciones usando la clase `Assert`.
A continuación se mencionan las más comunes:

- **fail([mensaje])**: Le permite al método fallar. Es usada comúnmente para comprobar si una parte del código está siendo alcanzada o que tenga un método que está fallando.
- **assertTrue([mensaje,] Condicion booleana)**: Afirma que la condición es `True`.
- **assertFalse([mensaje,] Condicion booleana)**: Afirma que la condición es `False`.
- **assertEquals([mensaje,] esperado, real [, delta])**: Afirma que el valor esperado es igual al valor real.
- **assertNull([mensaje,] object)**: Afirma que el objeto es nulo.

MIDAS

Hechos QUE CONECTAN

UNIVERSIDAD NACIONAL DE COLOMBIA un

Ejemplo con JUnit I

Genere una clase denominada `Calculadora`, la cual tendrá los siguientes métodos:

Ejemplo

```
public class Calculadora {

    public double suma(double num1, double num2) {
        return num1 + num2;
    }

    public double resta(double num1, double num2) {
        return num1 - num2;
    }

    public double multiplicacion(double num1, double num2) {
        return num1 * num2;
    }
```

Ejemplo con JUnit II

Ejemplo (continuación)

```
...
    public double division(double num1, double num2) {
        if (num2 != 0) {
            return num1 / num2;
        }
        else {
            return Double.NaN;
        }
    }

    public static void main(){}
}
```


Ejemplo con JUnit III

- NetBeans tiene una manera de generar UnitTest de manera automática a partir de una clase construida. En este caso, sobre las operaciones de la `Calculadora`.
- Vaya al menú `Tools`, y allí seleccione la opción `Create/Update Tools`. Allí coloque los valores por defecto, y esto generará un archivo de manera automática en la sección de `Test Packages`.
- En el menú `Run` vaya a la opción `Set Main Project` y seleccione su actual proyecto. Luego, en este mismo menú, puede seleccionar la opción `Test Project` para ejecutar todos los casos de prueba definidos.

Ejemplo con JUnit IV

Este es el ejemplo de un método auto-generado. Puede modificarlo a gusto, o usarlo como ejemplo para construir pruebas más específicas.

Ejemplo

```
@Test
public void testSuma() {
    System.out.println("suma");
    double num1 = 2.5;
    double num2 = 6.1;
    Calculadora instance = new Calculadora();
    double expResult = 8.6;
    double result = instance.suma(num1, num2);
    assertEquals("La suma no funciona bien.", expResult, result, 0.0);
}
```

En este enlace puede encontrar el código completo de este ejemplo.

Agenda

Problemas (I)

Problemas

1. Implemente la Clase `Calculadora`, que tendrá un método por cada operación matemática básica (suma, resta, multiplicación, división, módulo).
2. Luego cree la Clase `CalculadoraTest`, e implemente un método de Testing por cada método de la clase `Calculadora`.

Prepare casos de prueba para cada operación:

- ¿Qué pasa cuando la división es sobre cero?
- ¿Qué pasa en el módulo con cero?
- ¿Qué ocurre cuando los números son muy grandes?
- ¿Qué ocurre cuando los números son muy pequeños?

Problemas (II)

Problemas (continuación)

3. Implemente la clase `Persona`. Tendrá como atributos nombres, apellidos, nombreCompleto, edad, peso y estatura. El constructor recibirá nombres, apellidos, edad, peso y estatura y calculará el nombreCompleto como la concatenación de los nombres con los apellidos. Tendrá un único método `calcularIMC()` que retornará el IMC de la persona.
4. Cree la clase `PersonaTest` e implemente dos métodos de Testing, uno para evaluar el nombreCompleto y otro para evaluar el IMC. Defina los casos de prueba y corra los tests.

   ¿Pasaron todos los test a la primera? ¿Qué salió mal?