Historias de Usuario y Pruebas Unitarias

Programación Orientada a Objetos

Jonatan Goméz Perdomo, Ph.D.
jgomezpe@unal.edu.co

Carlos Andrés Sierra, M.Sc.
casierrav@unal.edu.co

Grupo de investigación en vida artificial – Research Group on Artificial Life – (Alife)
Departamento de Ingeniería de Sistemas e Industrial
Facultad de Ingeniería
Universidad Nacional de Colombia

Agenda

Agenda

Metodologías Ágiles de Desarrollo

El concepto de ágil se refiere a la habilidad de un equipo tanto de crear software como de adaptarse a cambios de manera rápida y eficiente. Surge en proyectos pequeños, con alta posibilidad de cambios durante el desarrollo, como es el caso de las *startups*.

Hoy en día, no solo pequeños equipos de trabajo las usan, pero suele ser la recomendación. Lo interesante de las metodologías ágiles es que la documentación es simple, fácil de modificar, y se pretende mantener un equipo con buenas prácticas de desarrollo, en particular: reuniones diarias, y pruebas, integración y despliegue continuos (de ser posible, todo automatizado).

Historias de usuario

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

La idea de trabajar las funcionalidades de esta manera es articular apropiadamente cada funcionalidad contra los objetivos del negocio, lo que conlleva a darle valor al cliente o usuario final.

Entre la filosofía de **User Stories** está el que la definición a realizar debe estar escrita en lenguaje natural, de manera simple, casi como si fuera una conversación con el cliente.

Historias de Usuario en el Desarrollo Ágil

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

Estos hitos son construidos y sostenidos constantemente por **User Stories**, las cuales se van actualizando a medida que surgen los objetivos que se buscan cumplir en pequeños periodos de tiempo. Entre las ventajas que dan las **User Stories** es que hay bastante enfoque en el cliente, se genera un ambiente bastante colaborativo, se da espacio a soluciones creativas, y se logran pequeñas victorias dando tracción a los procesos.

Planeación basada en Historias de Usuario

En las metodologías ágiles es normal que existan ciclos de iteración cortos, actualmente se les llama *sprints* (debido a que Scrum es ampliamente utilizado en estos contextos), los cuales regularmente son dos semanas.

Cada inicio de sprint, es común que se tomen los **User Stories** que no se han completado, se definan tareas (o sub-tareas), y se haga una medida del esfuerzo de las mismas. Luego, se determina cuales sub-tareas se incorporan al sprint.

Esto hace que iteración a iteración, siempre se esté aportando al negocio pero haciendo los ajustes correspondientes a medida que el proyecto está avanzando.

Agenda

Diseño de User Stories

Para diseñar **User Stories**, se debe tener en cuenta lo siguiente:

- Debe ser muy clara la condición con la cual se considera que el **User Story** ha sido completada, lo que incluye cumplir con todas las tareas.
- Las tareas se convierten en un paso a paso el cual si se logra se puede dar por cumplida la tarea. En este sentido, es importante tener muy bien definidas las tareas asociadas al **User Story**.
- Se debe tener en cuenta el usuario final, su rol en el proceso, ya que hacia este será orientada la **User Story**. La retroalimentación del usuario, en las propias palabras de este, permite comprender mejor el resultado esperado.
- La medida del esfuerzo para las tareas y para completar la **User Story** debe ser concensuada entre todo el equipo, esto evita sesgos.

Requerimientos Vs. User Stories

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

Una **User Story** podría tener el equivalente a varios requerimientos, que estarían específicadas en las tareas. Sin embargo, no se abordan desde la misma manera, puesto que los requerimientos se definen al comienzo del proyecto, y la idea es definirlos todos desde el inicio; en el caso de las **User Stories**, puede aparecer una nueva en cualquier momento, brindando más flexibilidad (que también puede ser peligrosa en exceso).

Recomendaciones para el Diseño

- Una forma sencilla para escribir las **User Stories** es la siguiente:

  `"Como un [usuario], yo [quiero], [para]…"`

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

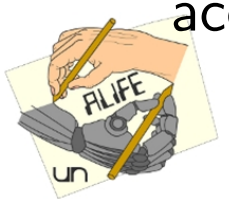
Recomendaciones para el Diseño

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

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

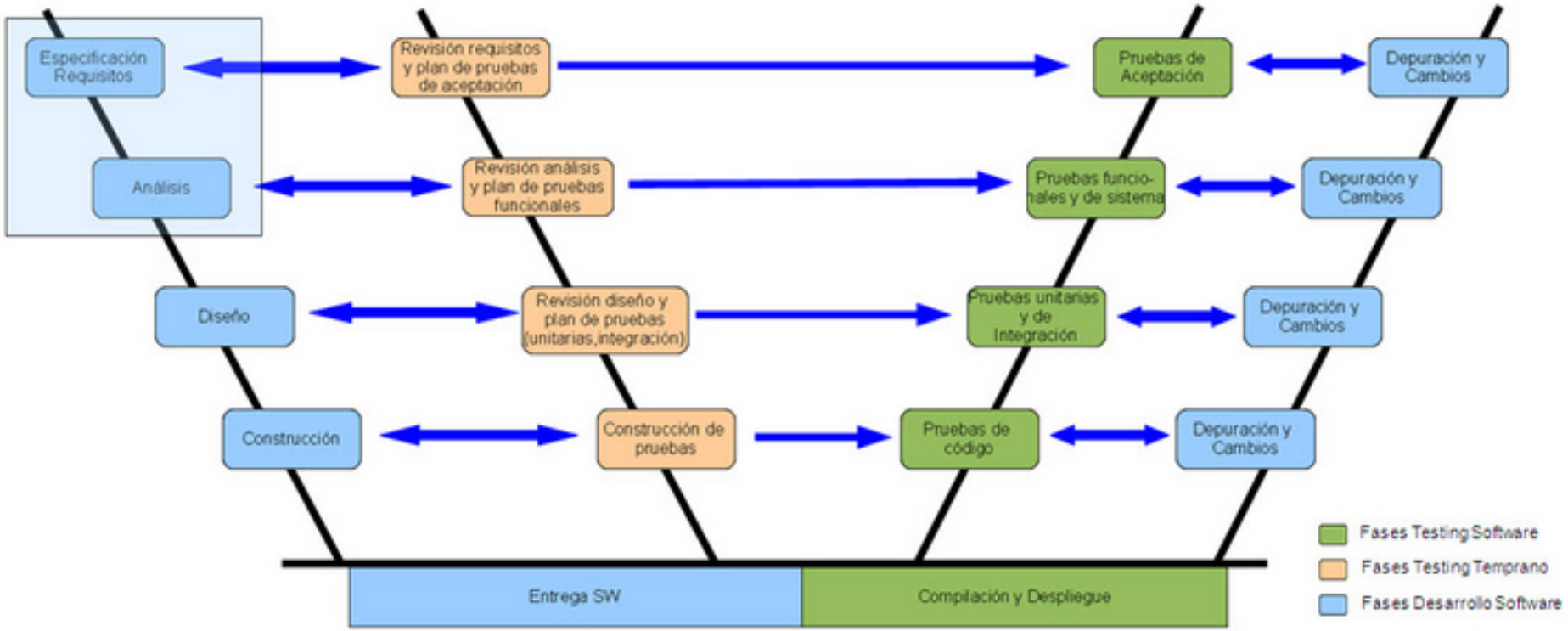


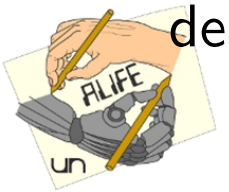
Figure: Imagen tomada desde este link

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 **User Stories** 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 parámetror.

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
`Math.pow(3, 4)`	81
`Math.pow(0, 0)`	1
`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

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 MyTests. Esta será usada únicamente para realizar los Tests de otras clases.
- Dentro de la clase, crear un método donde se definirá un Test en específico. Encima de la clase usar la anotación `@Test`

```
@Test
public void multiplicacionDeCerosDeberiaDevolverCero()
```

Definir un test en JUnit (II)

- Usar un método *assert* dentro de la función de pruebas, que funciona para declarar que algo debe cumplirse.
- Usar mensajes dentro de los *assert* para facilitar la identificación de las pruebas que fallaron.

```
MyClass tester = new MyClass();
assertEquals(0, tester.multiply(10, 0), "Must be 0");
```

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]`

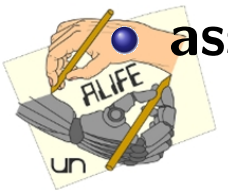
Anotaciones para Métodos de Prueba

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([message,] Esperado, Real)**: Afirma que el valor esperado es igual al valor real.
- **assertNull([Mensaje], object)**: Afirma que el objeto es nulo.



Ejemplos de JUnit

```
@Test
public void multiplicacionCerosDeberiaDevolverCero() {
MiCalculadora tester = new MiCalculadora();
assertEquals(0, tester.multiplicar(0, 0));
}
```

Un tutorial completo de JUnit con el ejemplo anterior está disponible aquí.
https://www.vogella.com/tutorials/JUnit/article.html

Agenda

Ejercicio (I)

Ejercicio

Implemente la Clase `Calculadora`, que tendrá un método por cada operación matemática básica (suma, resta, multiplicación, división, módulo).

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?

Ejercicio (II)

Ejercicio

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.

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?