S1. Introducción a las pruebas de sistemas

Sitio: Agencia de Habilidades para el Futuro
Curso: Metodología de Pruebas de Sistemas 2° D
Libro: S1. Introducción a las pruebas de sistemas

Imprimido por: Eduardo Moreno
Día: jueves, 14 de agosto de 2025, 22:59

Descripción

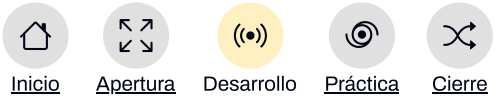


Tabla de contenidos

1. Comencemos con un código con errores

2. Introducción

3. Principios de testing
3.1. Principio 1: Definir
3.2. Principio 2: Evitar
3.3. Principio 3: Probar
3.4. Principio 4: Inspeccionar
3.5. Principio 5: Escribir
3.6. Principio 6: Examinar
3.7. Principio 7: No improvisar
3.8. Principio 8: Asumir
3.9. Principio 9: Encontrar
3.10. Principio 10: Ser creativos

4. Proceso de Testing

5. Enfoques y procesos claves de un software

6. Verificación y validación

7. Enfoques de pruebas
7.1. ¿En qué se diferencias los enfoques?

8. Un escenario posible para un ciclo de desarrollo y despliegue de software
8.1. Entorno de desarrollo
8.2. Entorno de prueba o testing
8.3. Entorno de pre-producción o staging
8.4. Entorno de producción

9. Niveles y tipos de pruebas

10. ¿Cuáles son los niveles de pruebas?
10.1. Pruebas unitarias
10.2. Pruebas de integración
10.3. Pruebas de sistemas
10.4. Pruebas de aceptación

11. Tipos de pruebas

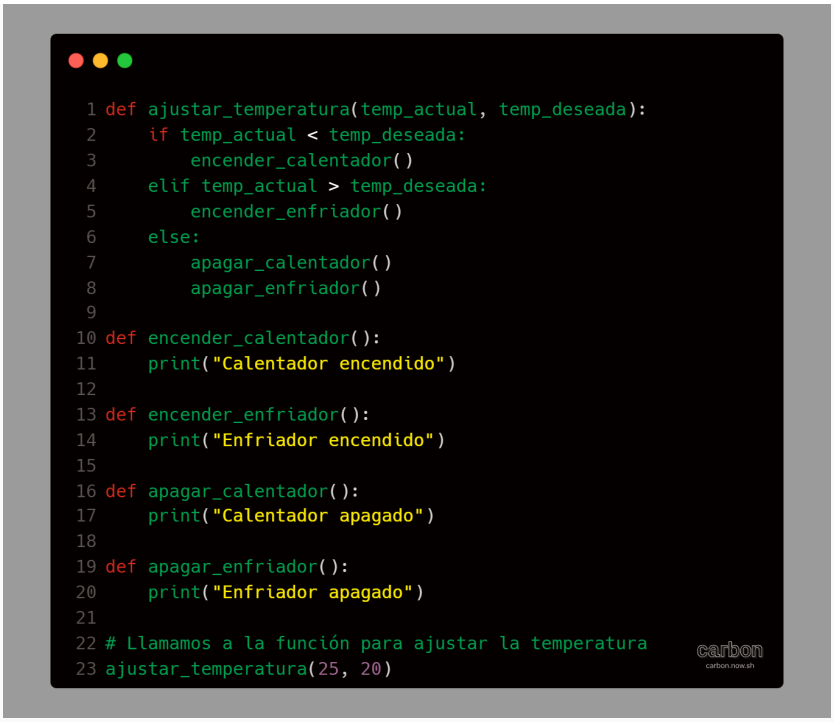
¡A encontrar un error en el código!

Antes de iniciar con los principios de *testing*, seguramente coincidimos que durante las materias anteriores, aprendiste a detectar errores y corregirlos. Por eso, comenzamos con un ejemplo simple de un código con errores en un dispositivo electrónico, en este caso, un termostato:

```
def ajustar_temperatura(temp_actual, temp_deseada):

    if temp_actual < temp_deseada:
        encender_calentador()
    elif temp_actual > temp_deseada:
        encender_enfriador()
    else:
        encender_calentador()
        encender_enfriador()

def encender_calentador():
print("Calentador encendido")

def encender_enfriador():
print("Enfriador encendido")

def apagar_calentador():
print("Calentador apagado")

def apagar_enfriador():
print("Enfriador apagado")

# Llamamos a la función para ajustar la temperatura
ajustar_temperatura(25, 20)
```

En este código, el objetivo es que el termostato ajuste la temperatura según la temperatura deseada. Sin embargo, hay un error en el flujo de control. Cuando la temperatura actual es igual a la temperatura deseada, el código sigue ejecutando las funciones encender_calentador() y encender_enfriador(), lo que no debería ocurrir cuando ya se alcanzó la temperatura deseada.

Para corregir este error, se debe modificar la función ajustar_temperatura() para que no encienda ni apague el calentador o el enfriador cuando la temperatura actual es igual a la deseada. Esto se puede lograr eliminando las llamadas a encender_calentador() y encender_enfriador() y apagando ambos en el caso else.



```python
def ajustar_temperatura(temp_actual, temp_deseada):
    if temp_actual < temp_deseada:
        encender_calentador()
    elif temp_actual > temp_deseada:
        encender_enfriador()
    else:
        apagar_calentador()
        apagar_enfriador()

def encender_calentador():
    print("Calentador encendido")

def encender_enfriador():
    print("Enfriador encendido")

def apagar_calentador():
    print("Calentador apagado")

def apagar_enfriador():
    print("Enfriador apagado")

# Llamamos a la función para ajustar la temperatura
ajustar_temperatura(25, 20)
```


En este código corregido, la función ajustar_temperatura() ahora apaga el calentador y el enfriador cuando la temperatura actual es igual a la temperatura deseada, lo que evita el comportamiento no deseado.

Introducción

El **testing de software** es un proceso, o una serie de procesos, diseñado para asegurarse que el código (que puede ser de computadora, celular, cámaras de foto, routers de internet, sensores y cualquier otro dispositivo electrónico) haga lo que debe hacer y que no haga nada que no debería hacer. El software debe ser **predecible y consistente**, sin esconder ninguna sorpresa para los usuarios.

En un mundo ideal queremos probar cada posible escenario o permutación de datos de un programa. En la mayoría de los casos no es posible debido a la cantidad de combinaciones posibles de entradas y salidas del software que queremos probar. Crear casos de prueba para cada uno de los posibles escenarios es poco práctico, requiere un tiempo muy elevado y necesita de muchos recursos económicos para que sea viable nuestro proyecto.

Cuando probamos un software no solo queremos "verificar que no haya errores" sino que deseamos agregar valor al producto. Agregar valor mediante las pruebas implica aumentar la fiabilidad del software. Aumentar la fiabilidad del software conlleva encontrar y eliminar los errores y defectos del software. Es decir que no probamos un software para ver que funcione, sino que lo hacemos asumiendo que el programa SI tiene errores y queremos encontrarlos.

En resumen, una definición completa del software sería **"El *testing* es el proceso de ejecutar un programa con la intención de encontrar errores".**

Cuando hacemos *testing* buscamos errores o defectos.

- Los **errores** son aquellos problemas con un artefacto de software que se encuentran en la etapa de su confección (por ejemplo, un error de especificación de requerimientos que se halla cuando se están especificando).
- Los **defectos** son errores de un artefacto de software que están en etapas posteriores a la de su confección (por ejemplo, un defecto del software que se encuentra en las etapas de *testing*).

Para encontrar errores y defectos utilizamos una herramienta llamada caso de prueba, que:

- es un conjunto de valores de entrada,
- son precondiciones de ejecución,
- son pasos de ejecución y
- son resultados esperados,
- están desarrollados para probar el software en ciertas condiciones específicas;
- sirven para establecer una ruta a seguir a la hora de probar el software y
- evaluar si se cumple un requerimiento en específico.

Principios de Testing

Se pueden establecer **10 principios** para las pruebas de software que deberían guiar esta disciplina en una organización o proyecto.

01

Principio 1

Una parte necesaria de un caso de prueba es la definición del resultado esperado.

02

Principio 2

Un programador debe evitar probar su propio programa.

03

Principio 3

Una organización de programación no debería probar sus propios programas.

04

Principio 4

Cualquier proceso de pruebas debe incluir una inspección de los resultados de cada prueba.

05

Principio 5

Los casos de prueba deben escribirse para condiciones de entrada que son inválidas e inesperadas, así como también para las válidas y esperadas.

06

Principio 6

Examinar un programa para ver si no hace lo que debe hacer es solo la mitad del trabajo, la otra mitad es ver que el programa no haga lo que no debe hacer.

07

Principio 7

Evita los casos de prueba improvisados.

08

Principio 8

No se deben planear procesos de pruebas asumiendo que no se encontraran errores.

09

Principio 9

La probabilidad de que existan más errores en una sección del programa es proporcional a la cantidad de errores ya encontrados en esa sección.

10

Principio 10

Probar es una tarea creativa e intelectualmente desafiante.

A continuación, estudiemos en detalle cada uno de estos principios.

Principio 1: Definir

Una parte necesaria de un caso de prueba es la definición del resultado esperado.

Este principio puede parecer obvio, pero ignorarlo es uno de los errores más frecuentes que se realizan en las empresas. Si el resultado esperado no se ha definido en un caso de prueba, entonces es muy probable que un resultado erróneo sea interpretado como correcto.

Para combatir este fenómeno, un caso de pruebas debe tener 2 componentes:

1. Una descripción detallada de la información de entrada del programa o función.
2. Una descripción precisa de los resultados esperados para el programa o función para esos datos específicos.

Para este principio podríamos considerar el siguiente **caso de prueba**:

> Debemos probar un programa de una calculadora y vamos a comenzar por la función suma. Esta función toma 2 valores, los suma entre ellos y retorna el resultado de la suma. Una descripción detallada de la entrada nos diría que el valor 1 es 5 y el valor 2 es 10. Una descripción precisa de los resultados esperados es 15. Por lo que probaremos la función con esas entradas y esperaremos que la respuesta sea un 15, para decir que funciona como corresponde. Otro ejemplo de este principio con el mismo programa, podría ser que el valor ingresado 1 es igual a 5 y el valor ingresado 2 es igual a “Hola”, aquí notamos que “Hola” no es valor numérico, por lo que no podemos realizar la suma.
>
> Esto quiere decir que nuestro resultado esperado es un mensaje que diga “Error. Ambos valores deben ser numéricos para poder sumarlos”. En caso de que nuestro programa no muestre ese mensaje especifico (porque no emite mensajes o muestra otro mensaje distinto) entonces podemos decir que falla la prueba y nuestro programa debe ser arreglado.

Les compartimos una representación de cómo podrías realizar pruebas de la función de suma.

```python
def suma(valor1, valor2):
    if isinstance(valor1, (int, float)) and isinstance(valor2, (int, float)):
        return valor1 + valor2
    else:
        return "Error. Ambos valores deben ser numéricos para poder sumarlos."

# Prueba 1: Valores numéricos válidos
resultado = suma(5, 10)
if resultado == 15:
    print("La prueba 1 pasó. El resultado es 15.")
else:
    print("La prueba 1 falló. El resultado es:", resultado)

# Prueba 2: Valor no numérico
resultado = suma(5, "Hola")
if resultado == "Error. Ambos valores deben ser numéricos para poder sumarlos.":
    print("La prueba 2 pasó. Se emitió el mensaje de error correctamente.")
else:
    print("La prueba 2 falló. El resultado es:", resultado)
```

Este código realiza dos pruebas: una con valores numéricos válidos y otra con un valor no numérico. Dependiendo de si el programa emite el resultado esperado o el mensaje de error correcto, puedes determinar

si la función de suma se comporta como se espera.

Principio 2: Evitar

Un programador debe evitar probar su propio programa.

Cualquier escritor/a sabe que es una mala idea intentar corregir sus propios escritos. Sabe qué es lo que debe decir, por lo que no reconoce cuándo, en realidad, dice otra cosa. Además, un desarrollador/escritor no quiere realmente encontrar errores en su propio trabajo. Lo mismo aplica para aquellas personas que escriben software. Otro problema es que cuando una persona que escribe un código con una mirada constructiva, va a tener problemas para mirar su código con un ojo destructivo para encontrar problemas.

Además de este problema asociado a la psicología, muchos de los defectos y problemas del software pueden ocurrir por un mal entendimiento del negocio o de los requerimientos. Si este es el caso, entonces el programador/a no podrá encontrar esos errores, incluso si quisiera.

Es por estas cuestiones que existe este principio:

Una persona que programa puede probar su propio código, pero no lo hará de una manera tan efectiva ni eficiente como alguien más. Sí es importante notar que una vez que los errores y defectos fueron encontrados, lo mejor es que el/la desarrollador/a que escribió el programa sea la persona indicada para arreglarlos.

Principio 3: Probar

Una organización de programación no debería probar sus propios programas.

Este principio es similar al anterior. Cualquier organización dedicada a la programación tiene los mismos problemas que una persona, tanto mentales como de conocimiento del negocio.

Además, se agrega el problema de los objetivos y métricas de las organizaciones de software; generalmente estos objetivos están ligados a tiempos de entrega y costos de los programas. Los tiempos y costos son simples y fáciles de medir, mientras que la calidad y fiabilidad de un software es algo extremadamente difícil de medir.

> Los objetivos de una organización no están orientados a la fiabilidad ni a la calidad del software, es probable que no hagan pruebas que realmente encuentren errores, ya que encontrar errores suele retrasar los tiempos de entrega y aumentar los costos de un proyecto.

Principio 4: Inspeccionar

Cualquier proceso de pruebas debe incluir una inspección de los resultados de cada prueba que implica:

- hacer un análisis de los resultados de las pruebas para revisar que efectivamente se esté testeando de forma precisa.
- ver que los defectos que se reportan sean realmente defectos,
- observar que se encuentra la mayor cantidad de defectos posibles.

Si se hacen muchas rondas de pruebas, es probable que las últimas encuentren defectos que no se encontraron en las pruebas originales.

Por ejemplo:

Imagina que estás probando una aplicación de comercio electrónico. Una de las pruebas consiste en verificar si los productos se agregan al carrito de compras correctamente. Después de ejecutar la prueba, inspeccionas los resultados y encuentras un error. La aplicación no muestra el mensaje de confirmación cuando se agrega un producto al carrito.

Este hallazgo es crucial, ya que sin el mensaje de confirmación, los usuarios pueden pensar que sus productos no se han agregado, lo que afecta la experiencia del usuario y las ventas. Identificar este problema durante la inspección de resultados te permite informar a los/as desarrolladores/as para que lo corrijan antes del lanzamiento de la aplicación, garantizando así una experiencia de usuario fluida.

Principio 5: Escribir

Los casos de prueba deben escribirse para condiciones de entrada que son inválidas e inesperadas, así como también para las válidas y esperadas.

Hay una tendencia natural cuando se prueba un programa a concentrarse en lo que es válido y las condiciones de entrada esperadas, y evitar probar las condiciones inválidas e inesperadas.

Por ejemplo, evitamos probar condiciones que darán mensajes de error.

- Si necesitamos que el usuario ingrese un número entre el 1 y el 10, entonces deberemos probar una entrada válida y esperada (un número del 1 al 10), pero también deberemos probar las entradas inválidas (un número negativo o un número mayor a 10).

Muchos de los defectos de los softwares se encuentran cuando los usuarios lo utilizan de formas que no son correctas y que, probablemente, no se nos hubieran ocurrido durante el desarrollo. Una persona experta en pruebas intenta buscar esas formas incorrectas y procura pensar como los usuarios del software.

Principio 6: Examinar

Examinar un programa para ver si no hace lo que debe hacer es sólo la mitad del trabajo, la otra mitad es ver que el programa no haga lo que no debe hacer.

En el principio 5 hablamos de probar todas las posibles entradas para verificar que nuestro software funciona correctamente; pero algo extremadamente importante, que muchos olvidan, es probar que el software no haga nada que no debe hacer.

Si el software no tiene errores para las funcionalidades que debe hacer, pero también tiene funcionalidades que no debería hacer, entonces no está funcionando como corresponde.

> Por ejemplo, un programa que ejecuta pagos de sueldos a empleados de una empresa que hace bien todos los pagos, pero que también genera cheques para empleados que no existen, no funciona como debe funcionar y tiene errores a arreglar. ¿Cuáles podrían los errores a arreglar?

- Generación de cheques para empleados inexistentes: Este es un error crítico, ya que se están generando pagos para empleados que no existen en el sistema. Debería implementarse una validación para asegurarse de que solo se realicen pagos a empleados registrados y activos en la empresa.
- Falta de verificación de la base de datos de empleados: El programa debería tener un mecanismo de verificación que compare la lista de empleados a pagar con la base de datos de empleados existentes. Esto garantizará que no se realicen pagos a empleados que no estén registrados.
- Falta de controles internos: El programa debe implementar controles internos efectivos para garantizar que los procesos de pago sean precisos y seguros. Esto incluye la revisión de datos y validación antes de realizar pagos.
- Informes y auditoría: Debe haber una funcionalidad de generación de informes y auditoría que permita rastrear y analizar los pagos realizados, identificar cualquier pago incorrecto o fraudulento y realizar un seguimiento de las transacciones.
- Seguridad de datos: Asegurarse de que los datos de los empleados y los detalles de los pagos estén protegidos adecuadamente para evitar posibles brechas de seguridad.

Corregir estos errores es fundamental para garantizar que el programa funcione como se espera y que los pagos de sueldos se realicen de manera precisa y segura.

Principio 7: No improvisar

Evitar los casos de prueba improvisados.

En ciertos tipos de empresas, es una práctica común sentarse a probar un software de forma improvisada e inventando casos de prueba a medida que se hacen las pruebas. El mayor problema con esta forma de trabajo es que, definir los casos de prueba, implica una gran inversión de tiempo y dinero; si no registramos esos casos, estamos desechando nuestra inversión. Si necesitamos volver a probar ese software, precisaremos definir nuevos casos de prueba, que incluso serán distintos a los primeros y pueden probar diferentes cosas.

Por lo general, re-definir los casos de prueba conlleva mucho esfuerzo, por lo que los nuevos casos de prueba no suelen ser tan exhaustivos como los primeros. Si en la corrección de otros errores se introducen nuevos errores, es muy probable que los nuevos casos de prueba no los identifiquen. Esto se debe a que las personas, generalmente, irán a probar las partes del software que fallaban, no las que funcionaban bien.

Guardar los casos de prueba que definimos y ejecutarlos en su completitud, luego de cambios al software, es lo que llamamos **pruebas de regresión**. En estas pruebas de regresión volvemos a ejecutar todos los casos de prueba, incluso si anteriormente no habían tenido errores.

Principio 8: Asumir

No se deben planear procesos de pruebas asumiendo que no se encontrarán errores.

Este es un error que se da cuando no se comprende la definición del testing, un proceso de ejecutar un programa con la intención de encontrar errores.

Es un error muy común para muchos directores de proyectos que esperan terminar su proyecto de software lo más rápido y barato posible, evitando pruebas que encuentren errores.

> Imaginá que una líder de proyecto está a cargo de desarrollar un software importante con un presupuesto ajustado y un plazo de entrega apretado. En un esfuerzo por completar el proyecto lo más rápido y económico posible, el director de proyecto decide omitir pruebas exhaustivas y evitar actividades de *testing* que puedan encontrar errores.
>
> En lugar de realizar pruebas rigurosas, el equipo de desarrollo lanza el software sin una validación adecuada. Como resultado, el software se lanza al mercado con múltiples errores y deficiencias no detectados, lo que genera una gran cantidad de quejas de los usuarios y problemas operativos.

Este ejemplo ilustra cómo la omisión o la reducción de pruebas en un proyecto de software puede ser un error costoso a largo plazo. A pesar de la intención inicial de ahorrar tiempo y dinero, la falta de pruebas adecuadas puede resultar en gastos adicionales para corregir errores, dañar la reputación de la empresa y afectar la satisfacción del cliente. Es un recordatorio de la importancia de no escatimar en el proceso de *testing* y de considerar las pruebas como una inversión en la calidad y la fiabilidad del software.

Principio 9: Encontrar

La probabilidad de que existan más errores en una sección del programa es proporcional a la cantidad de errores ya encontrados en esa sección.

Este fenómeno puede parecer un sinsentido, pero es una heurística que se cumple en la mayoría de los programas. Por lo general, cuando encontramos muchos errores o defectos en un software quiere decir que no está hecho con una gran calidad, por lo que es probable encontrar aún más.

Se puede decir que los errores o defectos vienen en grupos y que hay algunas secciones de los programas que tienden a tener más errores que otras. Teniendo esto en cuenta, podemos ver dónde centrarnos cuando estamos probando un software. Si encontramos muchos errores en una sección del software que estamos probando, entonces es un indicio de que debemos seguir probándolo para encontrar aún más errores.

Analicemos la sección en un software de administración de una clínica de salud con muchos errores podría ser la sección de "Programación de Citas".

Esta sección tiende a ser propensa a errores debido a su complejidad y a la gran cantidad de interacciones que involucra.

- Errores comunes en la sección de "Programación de Citas" podrían incluir:
- Doble reserva: El software permite programar dos citas en el mismo horario para un mismo médico, lo que resulta en conflictos de programación.
- Citas perdidas: Las citas programadas pueden desaparecer o no registrarse adecuadamente en el sistema, lo que lleva a pacientes sin cita y tiempo perdido.
- Incompatibilidad de horarios: El software no verifica la disponibilidad del médico o de la sala de consulta, lo que puede resultar en citas programadas en momentos en que no hay recursos disponibles.
- Falta de notificaciones: El software no envía recordatorios de citas a los pacientes, lo que puede resultar en ausencias y una menor eficiencia en la gestión de citas.
- Interfaz de usuario confusa: Una interfaz de usuario poco intuitiva puede llevar a errores de entrada de datos por parte del personal administrativo.

Estos errores pueden agruparse en la sección de "Programación de Citas" debido a su naturaleza compleja y a menudo interconectada. La falta de validación adecuada, la ausencia de notificaciones y problemas de interfaz de usuario contribuyen a la acumulación de errores en esta sección del software de administración de la clínica de salud.

En resumen

Corregir estos errores es esencial para garantizar un flujo de trabajo eficiente y una experiencia positiva para pacientes y personal administrativo.

Principio 10: Ser creativos

Probar es una tarea creativa e intelectualmente desafiante.

La creatividad necesaria para probar un programa es bastante mayor que la creatividad necesaria para diseñar ese programa.

Es imposible probar un software lo suficiente para garantizar la total ausencia de errores. A lo largo del curso estudiaremos metodologías para encontrar una cantidad numerosa de errores y para ayudarnos a que confecciones de casos de pruebas, pero todas estas metodologías requieren una gran cantidad de creatividad.

¿En qué momentos se necesita la creatividad? reflexionemos sobre algunas razones:

- Diseño de escenarios: los/las integrantes de los equipos deben ser creativos al diseñar una variedad de escenarios y casos de prueba que cubran diferentes aspectos del software, desde situaciones típicas hasta casos de borde y situaciones inusuales.
- Identificación de posibles problemas: Los/las integrantes de los equipos deben pensar de manera creativa para identificar posibles problemas, defectos o vulnerabilidades en el software. Esto implica considerar cómo los usuarios pueden interactuar con el software de maneras imprevistas.
- Creación de datos de prueba: En algunos casos, los/las integrantes de los equipos deben ser creativos al generar datos de prueba que reflejen situaciones reales o escenarios específicos que pueden ser difíciles de reproducir.
- Escritura de informes detallados: La creatividad se requiere al redactar informes detallados de pruebas que describan claramente los problemas encontrados, los pasos para reproducirlos y cualquier información adicional relevante.
- Exploración de funciones ocultas: Los/las integrantes de los equipos a menudo deben ser creativos al explorar funciones ocultas o poco documentadas del software, lo que puede revelar problemas no detectados por los usuarios típicos.
- Pruebas de usabilidad: La creatividad es esencial al evaluar la usabilidad del software, ya que implica ponerse en el lugar del usuario y pensar en cómo mejorar la experiencia.

En resumen

probar un software va más allá de simplemente seguir un conjunto de pasos predefinidos. Requiere un enfoque creativo para descubrir problemas, evaluar la usabilidad y garantizar que el software funcione de manera efectiva en una variedad de situaciones. La creatividad desempeña un papel fundamental en la identificación de problemas y la mejora de la calidad del software.

Enfoques y procesos claves de un software

Los procesos de comprobación de software y análisis del entorno son una parte fundamental de la práctica en el desarrollo de software, y contribuyen significativamente al éxito y la confiabilidad de los proyectos. Es por ello que estos procesos:

- Son actividades sistemáticas y planificadas que se realizan durante el ciclo de desarrollo de software.
- Tienen el objetivo de verificar y validar que el software cumple con los requisitos especificados y se comporta de manera esperada.
- La elección de las técnicas y herramientas de comprobación adecuadas depende del análisis del entorno al relevar las necesidades específicas del proyecto.

Verificación y validación

La **verificación** y **validación (V&V)** es el nombre que se da a los procesos de comprobación y análisis que aseguran que el software que se desarrolla está acorde a su especificación y cumple las necesidades de los clientes.

La V&V es un proceso de ciclo de vida completo. Inicia con las revisiones de los requerimientos y continúa con las revisiones del diseño y las inspecciones del código hasta la prueba del producto. Existen actividades de V&V en cada etapa del proceso de desarrollo del software.

La verificación y la validación no son lo mismo, sin embargo es muy común confundirlas. Conozcamos sus diferencias:

- **Verificación:** El proceso de verificación intenta comprobar que el software se está construyendo de acuerdo con su especificación. Se comprueba que el sistema cumple los **requerimientos funcionales y no funcionales** que se le han especificado. Básicamente, la verificación intenta responder la pregunta: ¿Estamos construyendo el producto correctamente?
- **Validación:** El proceso de validación intenta asegurar que el software cumple con las expectativas del client e . Va más allá de comprobar si el software está acorde con su especificación, para probar que el software hace lo que el usuario espera, a diferencia de lo que se ha especificado. Básicamente, la validación intenta responder la pregunta: ¿Estamos construyendo el producto correcto?

Es importante llevar a cabo la validación de los requerimientos del sistema de forma inicial. Es fácil cometer errores y omisiones durante la fase de análisis de requerimientos del sistema y, en tales casos, el software final no cumplirá las expectativas de los clientes.

> Si bien el análisis de requerimientos no es competencia de un Técnico Superior en Desarrollo de Software, en esta materia se verán contenidos que ayuden a validar esos requerimientos para que el Técnico Superior pueda leerlos y ayudar a quienes corresponde a encontrar errores pertinentes.

La validación de los requerimientos y la documentación es muy importante para lograr un software que satisfaga las necesidades de nuestros clientes. Sin embargo, en la realidad, la validación de los requerimientos no puede descubrir todos los problemas que presenta la aplicación. Algunos defectos en los requerimientos sólo pueden descubrirse cuando se ha terminado la implementación del sistema.

Enfoques de pruebas

Existen 2 enfoques distintos que nos ayudarán con la verificación y validación del software. Estos enfoques nos ayudarán a ver qué herramientas utilizaremos para encontrar los errores y defectos que queremos arreglar.

Enfoque estático

Con este enfoque buscamos errores o defectos sin la necesidad de ejecutar el sistema.

> Consiste en realizar un análisis y comprobación de las representaciones del sistema como el documento de requerimientos, los diagramas de diseño y el código fuente del programa. Se aplica a todas las etapas del proceso de desarrollo.

Este enfoque se complementa con algún tipo de análisis automático del texto fuente o de los documentos asociados. En el caso del código, podemos analizarlo con ayudas de herramientas como **Crucible**, que permite una revisión del código colaborativa y el inicio de conversaciones en porciones específicas del código para dejar comentarios o anotaciones puntuales. La herramienta de repositorios **GitHub** también contiene un analizador de código que permite hacer una comparación entre código anterior y código agregado recientemente.

Enfoque dinámico

Un entorno de trabajo es el contexto y conjunto de condiciones bajo los cuales se desarrolla, prueba y utiliza el software. Esto incluye desde el hardware a utilizar (cantidad de RAM, procesador, hardware específico como una máquina expendedora, etc.) hasta el software que ayuda a que nuestro programa funcione (sistemas operativos, navegadores web, paquetes de desarrollo, SDK, aplicaciones de monitoreo, otros programas que existan en la misma máquina al mismo momento) junto con la configuración que tienen ambos.

> Cuando nosotros programamos un software, lo hacemos en nuestras propias computadoras, que tendrán ciertas características que podrán ser diferentes a las características en las que un software se va a utilizar. Es decir que el entorno en el que desarrollamos la aplicación es distinto al entorno en el que se usará la aplicación.

Teniendo esto en cuenta lo anterior, podemos reconocer 4 entornos de trabajo diferentes que nos serán útiles a la hora de participar de cualquier proyecto de desarrollo. Cada uno de ellos son utilizados con un fin específico, y presentan ciertas ventajas sobre los otros en determinado momento del proceso de trabajo.

¿En qué se diferencias los enfoques?

El enfoque estático y el enfoque dinámico son dos metodologías diferentes utilizadas en el proceso de pruebas de software, y se diferencian principalmente en cuándo y cómo se realizan las pruebas:

Enfoque estático

- ¿Cuándo se aplica?: Se aplica durante las etapas iniciales del ciclo de desarrollo de software, antes de la ejecución del programa.
- Método principal: Se centra en la revisión y análisis de los artefactos del software, como documentos de especificación, diagramas de diseño, y código fuente. No implica la ejecución del software.
- Propósito: Identificar errores, inconsistencias, problemas de diseño y lógica en el software antes de que se ejecute, lo que ayuda a prevenir defectos en etapas posteriores del desarrollo.

Enfoque dinámico

- ¿Cuándo se aplica?: Se aplica durante la fase de ejecución del programa o aplicación.
- Método principal: Implica la ejecución real del software bajo condiciones de prueba específicas. Se utilizan casos de prueba y escenarios para evaluar su comportamiento y rendimiento.
- Propósito: Evaluar el funcionamiento real del software, descubrir defectos funcionales, identificar problemas de rendimiento y verificar si cumple con los requisitos especificados.

En resumen

el enfoque estático se enfoca en la revisión y análisis sin ejecutar el software, mientras que el enfoque dinámico se concentra en la ejecución real del software para evaluar su comportamiento.

Ambos enfoques son complementarios y se utilizan en diferentes etapas del proceso de desarrollo de software para garantizar la calidad y la fiabilidad del producto final.

Un escenario posible para un ciclo de desarrollo y despliegue de software

Antes de avanzar en la descripción de cada entorno para el desarrollo de un software, les compartimos características en en el contexto de un software de factura digital de la AFIP (Administración Federal de Ingresos Públicos de Argentina), en el que cada uno cumplen roles específicos en el ciclo de desarrollo y despliegue de software. A continuación, se describen las diferencias entre estos entornos:

1. **Entorno de desarrollo:**
    - Propósito: Es el entorno donde los desarrolladores escriben y prueban el código del software. Aquí se realizan cambios y mejoras en el software.
    - Características:
        - No está conectado al sistema en producción.
        - Se utiliza para la programación, depuración y pruebas iniciales.
        - Puede ser un ambiente local en las computadoras de los desarrolladores o un servidor de desarrollo.
        - No está destinado a uso público ni a operaciones reales.
2. **Entorno de prueba:**
    - Propósito: Se utiliza para probar el software en un ambiente controlado antes de implementarlo en producción. Aquí se validan las funcionalidades y se detectan errores.
    - Características:
        - Réplica del entorno de producción, pero aislado.
        - Se simulan escenarios de uso real.
        - Los datos de prueba se utilizan para probar la aplicación.
        - Se pueden realizar pruebas de rendimiento, seguridad y funcionalidad.
        - Los errores se identifican y corrigen antes de la implementación en producción.
3. **Entorno de pre-producción:**
    - Propósito: Es un entorno de transición entre el desarrollo y la producción. Se utilizan para realizar pruebas finales y asegurarse de que la aplicación esté lista para el entorno de producción.
    - Características:
        - Similar al entorno de producción, pero con datos y tráfico limitados.
        - Se realizan pruebas de integración, aceptación y validación.
        - Los usuarios finales o equipos de control de calidad pueden realizar pruebas.
        - Puede usarse para entrenamiento del personal.
4. **Entorno de producción:**
    - Propósito: Es el entorno en el que se ejecuta el software en operaciones reales y se utiliza por los usuarios finales.
    - Características:
        - Es el sistema en vivo que interactúa con los usuarios y procesa transacciones reales.
        - Debe ser altamente confiable, escalable y estar optimizado para el rendimiento.
        - Cualquier cambio o actualización debe pasar por los otros entornos antes de implementarse aquí.

En el contexto de la factura digital de AFIP, estos entornos son fundamentales para garantizar que el software cumpla con los requisitos legales y funcione de manera óptima antes de utilizarse en producción, donde se generan y envían facturas y documentos fiscales de manera oficial. Los entornos de prueba y pre-producción permiten identificar errores y garantizar la integridad y la conformidad con las regulaciones fiscales antes de operar en el entorno de producción.

Entorno de desarrollo

En el entorno de desarrollo se programa el software. Puede ser la computadora de la persona que programa el software o puede ser un servidor compartido por todos los/las desarrolladores accesible mediante una VPN (Virtual Private Network).

> Es el espacio de trabajo donde se desarrolla el código de nuestra aplicación y donde comprobaremos que nuestro código pueda ser compilado y ejecutado.

La ventaja de este entorno es que los/as desarrolladores/as:

- pueden instalar todos los paquetes que necesiten,
- cambiar la distribución de las carpetas a gusto,
- cualquier cambio que se realiza en un entorno de desarrollo quedará en el entorno de desarrollo hasta que el programador así lo decida,
- las computadoras de los/as desarrolladores/as suelen ser relativamente potentes con unas especificaciones relativamente altas, por lo que el programa puede ser compilado y probado rápidamente y sin demoras.

A continuación, describimos el entorno de prueba.

Entorno de prueba o *testing*

En el entorno de pruebas es donde el *tester* ejecuta las pruebas y busca errores de las funcionalidades que hayamos desarrollado.

En este entorno no tendremos instaladas ninguna de las librerías y aplicaciones extra que pueda haber en un entorno de desarrollo, sólo debería tener el software necesario para que el programa funcione.

> El movimiento del código entre el entorno de **desarrollo** y el entorno de **prueba** debe ser extremadamente frecuente, ya que mientras más rápido se pruebe algo, se podrán encontrar los errores, podremos arreglarlos y llevarlo al entorno de producción.

La principal ventaja de contar con un entorno de desarrollo:

- es la posibilidad de permitir a los administradores de la aplicación, a los clientes, u otros miembros del equipo,
- conocer el comportamiento de la aplicación, formación, y realizar pruebas y *testing* por el equipo de pruebas.

El servidor de pruebas, no solo permite tener un entorno en el cual otros miembros del equipo pueden probar la aplicación, también posibilita consultar el comportamiento de la aplicación, cuando esta requiere recibir respuestas de otro servidor.

> Por ejemplo, si nosotros quisiéramos hacer un software que permita facturar, deberemos conectarnos con el software de facturación de AFIP para generar la factura. Si quisiéramos probar que nuestro software funciona bien utilizando el software real de AFIP, podríamos generar muchas ventas que luego impactarían negativamente en nuestra situación impositiva ya que AFIP no podrá distinguir entre ventas reales y ventas de pruebas. Es por eso que AFIP tiene disponible un entorno de pruebas, para que nosotros podamos probar nuestro software y generar facturas falsas que no impactarían realmente en AFIP.

Entorno de pre-producción o *staging*

El entorno de pre-producción es el último entorno al que vamos a desplegar nuestra aplicación antes de que vaya a un entorno de producción. Este entorno tiene como objetivo emular el entorno de producción para probar el rendimiento de la aplicación, así como sus actualizaciones; queremos ver que funcione de forma eficiente y sin romper nada que ya estaba funcionando.

Si bien el objetivo de este entorno es ser lo más parecido a producción posible, no siempre tenemos los recursos disponibles para lograr este objetivo. A veces simplemente no es posible recrear un entorno de producción.

Un ejemplo de esto puede verse en el área de las telecomunicaciones, donde no tendremos satélites o cables telefónicos sólo para un entorno de pre-producción. Si queremos probar un software que maneja satélites, no podemos tener un “satélite de pre-producción” sólo para probar su software debido al elevado costo de manejar tal operación.

Entorno de producción

Este entorno tiene su nombre ya que decimos que es un entorno “productivo”, es decir que el software se utiliza para algo productivo y no solo para que se realicen pruebas. Es lo que llamamos “el mundo real”, donde finalmente ejecutamos la aplicación y nuestros usuarios finales tendrán acceso al programa.

> Llamamos “pasaje a producción” o “pasar a producción” cuando actualizamos el entorno de producción para que contenga los últimos cambios al software. Es importante notar que estos entornos son muy difíciles de replicar, por lo que muchas veces encontramos defectos y problemas que no esperábamos encontrar.

La dificultad de replicarlo está dada por la cantidad de tráfico, los usos no pensados que puedan darle los clientes al software, u otros programas instalados en la máquina del cliente que no son compatibles con nuestro software.

Continuemos con el escenario de una factura digital.

Supongamos que estamos tratando con un software de factura digital de AFIP (Administración Federal de Ingresos Públicos de Argentina) en un entorno de producción. Se llevarían a cabo una serie de actividades relacionadas con la operación en vivo del software o sistema, donde los usuarios finales interactúan con la aplicación para llevar a cabo sus tareas.

1. Generación y envío de facturas: En un entorno de producción, los contribuyentes utilizan el software para generar facturas electrónicas o comprobantes fiscales digitales de acuerdo con las regulaciones de la AFIP. Estas facturas se generan en tiempo real y se envían a los destinatarios, ya sea por correo electrónico o por medios electrónicos, según las normativas fiscales
2. Registro de transacciones: El software registra todas las transacciones fiscales, incluyendo la emisión de facturas, notas de crédito, notas de débito y otros comprobantes fiscales. Cada transacción se almacena en una base de datos segura para su posterior consulta y auditoría.
3. Procesamiento de impuestos: El sistema calcula automáticamente los impuestos correspondientes a cada factura emitida. Esto incluye impuestos como el IVA, ingresos brutos, percepciones, retenciones y otros tributos locales. Los impuestos calculados se informan a la AFIP para su posterior liquidación y pago.
4. Almacenamiento y respaldo de datos: En un entorno de producción, se mantiene una copia de seguridad de todos los datos fiscales y transacciones realizadas. Esto garantiza la integridad de la información y permite la recuperación de datos en caso de pérdida o fallos en el sistema.
5. Interacción con la AFIP: El software interactúa con los sistemas de la AFIP para enviar información relevante, como la facturación electrónica, los pagos de impuestos y otros reportes fiscales requeridos. Esta interacción asegura que la empresa cumpla con las regulaciones fiscales vigentes.
6. Soporte a usuarios: En el entorno de producción, es esencial proporcionar soporte técnico y asistencia a los usuarios en caso de problemas técnicos, dudas sobre el uso del software o consultas relacionadas con las transacciones fiscales. El soporte ayuda a garantizar un funcionamiento continuo del sistema.
7. Auditorías y cumplimiento: Se realizan auditorías internas y, en ocasiones, auditorías externas para verificar el cumplimiento de las regulaciones fiscales y la integridad de los registros fiscales. Esto es fundamental para garantizar la legalidad y la exactitud de las operaciones.

En resumen

Un entorno de producción es el escenario donde el software de facturación electrónica de AFIP se utiliza en operaciones en vivo para emitir facturas, gestionar impuestos y cumplir con las obligaciones fiscales. El funcionamiento correcto y eficiente en este entorno es esencial para las empresas y para garantizar el cumplimiento de las regulaciones fiscales.

Niveles y tipos de pruebas

Las pruebas desempeñan un papel fundamental en el desarrollo de software, contribuyendo a la calidad, confiabilidad y seguridad de las aplicaciones. Para comprender completamente cómo funcionan las pruebas, es importante explorar los diferentes niveles y tipos de pruebas disponibles. En este contexto, nos adentraremos en el mundo de las pruebas de software, y para empezar, plantearemos cuatro preguntas clave que nos ayudarán a entender mejor este aspecto esencial del proceso de desarrollo:

1. ¿Qué son los niveles de pruebas y por qué son importantes? Exploraremos cómo las pruebas se organizan en diferentes niveles, desde pruebas unitarias hasta pruebas de aceptación del usuario, y cómo cada nivel aborda aspectos específicos del software.
2. ¿Cuáles son los tipos de pruebas más comunes? Examinaremos una variedad de tipos de pruebas, como pruebas funcionales, pruebas de rendimiento, pruebas de seguridad y más. Cada tipo se enfoca en aspectos particulares del software.
3. ¿Cuándo se llevan a cabo las pruebas en el ciclo de desarrollo de software? Analizaremos en qué etapas del desarrollo se realizan las pruebas y por qué es crucial integrarlas a lo largo de todo el proceso.
4. ¿Cómo se seleccionan las pruebas adecuadas para un proyecto específico? Discutiremos cómo elegir las técnicas y herramientas de pruebas que se adapten a las necesidades particulares de un proyecto, teniendo en cuenta su entorno y requisitos.

Estas preguntas nos servirán de base para explorar en detalle los niveles y tipos de pruebas en el desarrollo de software, y cómo contribuyen al éxito y la calidad de las aplicaciones.

¿Cuáles son los niveles de prueba?

Conocer y comprender los diferentes niveles de pruebas en el desarrollo de software resulta fundamental para garantizar la calidad y fiabilidad de cualquier producto final. ¿Por qué? Pues, permite reducir riesgos y costos asociados con errores detectados tarde en el ciclo de desarrollo.
Por lo tanto, la comprensión de los niveles de prueba te permitirán una planificación adecuada de las mismas y una mejora continua en el proceso de desarrollo de software.

Cada nivel de prueba tiene un propósito específico y se lleva a cabo en una etapa particular del ciclo de desarrollo de software.

En esta oportunidad, nos detendremos en cuatro niveles:

- pruebas unitarias,
- pruebas de integración,
- pruebas de sistemas y,
- pruebas de aceptación.

Los niveles de pruebas están asociados al “qué” se prueba, pero considerando el sistema como un conjunto de partes. Cada nivel de pruebas se enfoca en un nivel de composición de partes del sistema.

Pruebas unitarias

Este nivel de pruebas se enfoca en componentes (métodos, funciones) de forma aislada. Es decir, la prueba se hace para métodos o funciones individuales.

> No requieren tener la totalidad del sistema terminado porque se prueba sólo el método bajo análisis. Si consideramos un auto, una prueba unitaria sería aquella que se enfoca en las ruedas.

Realicemos una caracterización general:

- son las pruebas más rápidas de ejecutar
- proveen retroalimentación más rápido, porque tan pronto se ejecuta la prueba se sabe si falló o no.
- son las más eficaces a la hora de detectar el componente exacto donde hay un error en el código.
- se realizan en el entorno de desarrollo por las mismas personas que escriben el código y pueden ser automatizadas.

Para entenderlo mejor podemos analizar este ejemplo. Si tenemos un método como el siguiente:

```
public int sumar(int a, int b) {
  return a + b;
}
```

Entonces una prueba unitaria declararía a con un valor y b con otro, y vería que se obtenga el resultado correcto.

Esto se haría de la siguiente forma:

```
int a = 5
int b = 10
@Test
assertEquals(15,sumar(a,b));
```

Esta prueba es lo que llamamos un **pseudocódigo**, es decir que no es exactamente código, sino que imita al código sin pertenecer a ningún lenguaje en particular.

En la prueba vemos que declaramos con números específicos a las variables a y b. Luego las enviamos por parámetro al método sumar() y comparamos que el resultado del método sea el que debería ser.

Pruebas de integración

Este nivel se enfoca en la combinación de diferentes componentes. Se centra en la interacción de los componentes del software y en buscar errores cuando unimos las partes que hemos desarrollado.

> Volviendo al ejemplo del auto, una prueba de integración sería para todo el sistema de transmisión que incluye las llantas, el embrague, la caja de cambios, el diferencial, etc.

Analicemos sus características:

- no son tan rápidas como las pruebas unitarias porque requiere construir una versión del sistema que incluye todos los componentes a probar.
- pueden ser realizadas por los/las desarrolladores/as que están escribiendo el código y también pueden ser automatizadas.

> Retomando el ejemplo del auto, si utilizamos el módulo del acelerador y sabemos que el auto está avanzando a una velocidad de 60 km/h, entonces cuando vamos al módulo del tablero de mando, la aguja del velocímetro deberá reflejar esos 60 km/h

Cualquier estrategia de prueba de versión o de integración debe ser incremental, para lo que existen dos esquemas principales:

Integración de arriba hacia abajo (top-down) donde se prueban primero los componentes de más alto nivel y se va avanzando hacia los de más bajo nivel. Esto significa que primero desarrollamos las interfaces entre subsistemas, las que retornan valores que son controlados, y posteriormente se reemplaza con el código real; esto permite probar el flujo completo en niveles superiores antes, y luego ir a niveles inferiores.

Integración de abajo hacia arriba (bottom-up) donde se prueban primero los componentes de más bajo nivel y se va avanzando hacia los de más alto nivel.

Los módulos o subsistemas críticos deben ser probados lo antes posible para encontrar errores críticos para el software rápidamente.

Entonces podemos decir que debemos ir conectando de a poco las partes del software que tenemos e ir probando una a una para encontrar errores de forma ordenada.

Pruebas de sistemas

En este nivel se prueba el sistema como un todo.

> En el ejemplo del auto, probaríamos el auto completo

Las pruebas de sistema se pueden hacer de forma granular, es decir, casos de uso/funcionalidades individuales, o combinando flujos completos de inicio a fin. Si seguimos los principios de las pruebas, este nivel debería hacerlo alguien diferente a quien desarrolló el software. Si bien estas pruebas pueden automatizarse, no es tan común hoy en día.

También se intenta determinar si el sistema en su globalidad opera satisfactoriamente. Con esto no nos referimos sólo a la funcionalidad sino también a cuestiones de recuperación de fallas, seguridad, pruebas de estrés y pruebas de performance.

> Como se mencionó anteriormente, el entorno de pre-producción, en el que se hacen las pruebas de sistema debe ser **lo más parecido posible** al entorno en el cual se usará el software en el día a día. Si los entornos son muy diferentes, es posible que se encuentren errores que en su uso normal no existirían, o aun peor, que no se encuentren errores que sí se encontraran en su entorno normal de uso.

Si bien se aspira a esta similitud, muchas veces no es posible anticipar las necesidades ni condiciones de uso reales.

> Es posible hacer un software de ventas online que considere que miles de personas accederán al sitio, pero si el negocio tiene un Hot Sale o descuento por fiestas, el trafico crecerá de una manera que no se puede anticipar y colapsara el sistema. Estos casos no necesitan ser tan extremos como el caso de un Hot Sale, muchas veces ocurrirá que no encontraremos defectos que si o si hallaremos en producción.

Pruebas de aceptación

Se prueba el sistema como un todo, pero en este caso en condiciones reales de uso.

> En el ejemplo del auto, sería conducir por las sierras de Córdoba, la ruta 40 o la avenida 9 de Julio en hora pico.

Estas pruebas:

- son las más costosas de realizar.
- es más difícil encontrar el lugar exacto del código donde se generó el error.
- son realizadas por el cliente o los usuarios del software para que determinen si el software se ajusta a sus necesidades.

El objetivo de las pruebas de aceptación es **establecer confianza en el sistema, las partes del sistema o las características específicas y no funcionales del sistema**. Encontrar defectos no es el foco principal en las pruebas de aceptación, cualquier error o defecto encontrado en esta etapa es extremadamente costoso.

Comprende tanto la prueba realizada por el usuario en ambiente de laboratorio (**pruebas alfa**), como la prueba en ambientes de trabajo reales (**pruebas beta**).

Algunos tipos de prueba

Se tienen muchos tipos de pruebas distintos que permiten asegurar una calidad mínima en el software que estaremos probando. Ninguno de ellos es independiente de otros ni mutuamente excluyentes; en un proceso de pruebas completo deberíamos utilizar varios tipos en combinación.

1. ***Tests* de operación:** Es el tipo más común. El sistema es probado en operación normal. Mide la confiabilidad del sistema y se pueden obtener mediciones estadísticas.
2. ***Tests* de *performance* o de capacidad:** El objetivo de esta prueba es medir la capacidad de procesamiento del sistema. Los valores obtenidos son comparados con los requeridos.
3. ***Tests* de escala completa:** Ejecutamos el sistema al máximo, todos los parámetros enfocan a valores máximos, todos los equipos conectados, usados por muchos usuarios ejecutando *use cases* simultáneamente. Si un sistema está hecho para ser usado por 1000 usuario simultáneos, entonces esta prueba busca probar qué ocurre cuando tenemos esos 1000 usuarios simultáneos.
4. ***Tests* de sobrecarga:** Cumple la función de determinar cómo se comporta el sistema cuando es sobrecargado. No se puede esperar que supere esta prueba, pero sí que no se venga abajo, que no ocurra una catástrofe. Cuántas veces se cayó el sistema es una medida interesante. Si tomamos el sistema de 1000 usuarios de la prueba de Escala Completa, este sistema prueba qué ocurre cuando tenemos 1001 usuarios en simultáneo, podemos realizar este tipo de prueba con tanta sobrecarga como queramos y necesitemos probar.
5. ***Tests* negativos:** El sistema es sistemática e intencionalmente usado de forma incorrecta. Este maltrato debe ser planeado para probar casos especiales y así asegurar la tolerancia a fallos.
6. ***Tests* basados en requerimientos:** Estos *tests* son los que pueden rastrearse, directamente desde la especificación de requerimientos.
7. ***Tests* ergonómicos:** Son muy importantes si el sistema será usado por gente inexperta. Se prueban cosas como:
    - Consistencia de la interfaz.
    - Consistencia entre las interfaces de los distintos *use cases*.
    - Si los menús son lógicos y legibles.
    - Si se entienden los mensajes de falla.
8. ***Tests* de documentación del usuario:** Con el estilo y características del anterior, se prueba la documentación del sistema.
9. ***Test* de aceptación:** Este test es ejecutado por la organización que solicita el sistema. El sistema es probado en un entorno real, usualmente llamado **Alfa**. Genera la aceptación o no del sistema. Cuando no hay un usuario que solicita el producto, se usan las pruebas **Beta**, que son encargadas a clientes selectos antes de liberar la versión.