Técnicas de Pruebas en Java

INTRODUCCIÓN A LAS TÉCNICAS DE PRUEBAS

Introducción

- El software generado en la fase de implementación no puede ser "entregado" al cliente para que lo utilice, sin practicarle antes una serie de pruebas.
- La fase de pruebas tienen como objetivo encontrar defectos en el sistema final debido a la omisión o mala interpretación de alguna parte del análisis o el diseño. Los defectos deberán entonces detectarse y corregirse en esta fase del proyecto.
- En ocasiones los defectos pueden deberse a errores en la implementación de código (errores propios del lenguaje o sistema de implementación), aunque en esta etapa es posible realizar una efectiva detección de los mismos, estos deben ser detectados y corregidos en la fase de implementación.
- La prueba puede ser llevada a cabo durante la implementación, para verificar que el software se comporta como su diseñador pretendía, y después de que la implementación esté completa.

Introducción

- Esta fase tardía de prueba comprueba la conformidad con los requerimientos y asegura la fiabilidad del sistema.
- Las distintas clases de prueba utilizan **clases de datos de prueba** diferentes:
  - La **prueba estadística**.
  - La **prueba de defectos**.
  - La **prueba de regresión**.

Prueba estadística

- La **prueba estadística** se puede utilizar para probar el rendimiento del programa y su confiabilidad.
- Las pruebas se diseñan para reflejar la frecuencia de entradas reales de usuario.
- Después de ejecutar las pruebas, se puede hacer una estimación de la confiabilidad operacional del sistema.
- El rendimiento del programa se puede juzgar midiendo la ejecución de las pruebas estadísticas.

Prueba de defectos

- La **prueba de defectos** intenta incluir áreas donde el programa no está de acuerdo con sus especificaciones.
- Las pruebas se diseñan para revelar la presencia de defectos en el sistema.
- Cuando se han encontrado defectos en un programa, éstos deben ser localizados y eliminados. A este proceso se le denomina **depuración**.
- La prueba de defectos y la depuración son consideradas a veces como parte del mismo proceso. En realidad, son muy diferentes, puesto que la prueba establece la existencia de errores, mientras que la depuración se refiere a la localización y corrección de estos errores.

Error, defecto o fallo

- En el área del aseguramiento de la calidad del software, debemos tener claros los conceptos de Error, Defecto y Fallo.
- En muchos casos se utilizan indistintamente pero representan conceptos diferentes:
  - Error: Es una acción humana, una idea equivocada de algo, que produce un resultado incorrecto. El error es una equivocación por parte del desarrollador o del analista.
  - Defecto: Es una imperfección de un componente causado por un error. El defecto se encuentra en algún componente del sistema. El analista de pruebas es quien debe encontrar el defecto ya que es el encargado de elaborar y ejecutar los casos de prueba.
  - Fallo: Es la manifestación visible de un defecto. Es decir que si un defecto es encontrado durante la ejecución de una aplicación entonces va a producir un fallo.
- Un error puede generar uno o más defectos y un defecto puede causar un fallo.

Prueba de regresión

- Durante la depuración se debe generar hipótesis sobre el comportamiento observable del programa para probar entonces estas hipótesis esperando provocar un fallo y encontrar el defecto que causó la anomalía en la salida.
- Después de descubrir un error en el programa, debe corregirse y volver a probar el sistema.
- A esta forma de prueba se le denomina **prueba de regresión**.
- La prueba de regresión se utiliza para comprobar que los cambios hechos a un programa no han generado nuevos fallos en el sistema.

Conflicto de intereses

- Aparte de esto, en cualquier proyecto software existe un **conflicto de intereses** inherente que aparece cuando comienza la prueba:
  - Desde un punto de vista psicológico, el análisis, diseño y codificación del software son tareas **constructivas**.
    - El ingeniero de software crea algo de lo que está orgulloso, y se enfrenta a cualquiera que intente sacarle defectos.
    - La prueba, entonces, puede aparecer como un intento de “romper" lo que ha construido el ingeniero de software.
  - Desde el punto de vista del constructor, la prueba se puede considerar (psicológicamente) **destructiva**.
    - Por tanto, el constructor anda con cuidado, diseñando y ejecutando pruebas que demuestren que el programa funciona, en lugar de detectar errores.
    - Desgraciadamente los errores seguirán estando, y si el ingeniero de software no los encuentra, lo hará el cliente.

Desarrollador como probador

- A menudo, existen ciertos **malentendidos** que se pueden deducir equivocadamente de la anterior discusión:
  1. El desarrollador del software no debe entrar en el proceso de prueba.
  2. El software debe ser “puesto a salvo” de extraños que puedan probarlo de forma despiadada.
  3. Los encargados de la prueba sólo aparecen en el proyecto cuando comienzan las etapas de prueba.
- Cada una de estas frases es incorrecta.
- El **desarrollador** siempre es responsable de probar las unidades individuales (módulos) del programa, asegurándose de que cada una lleva a cabo la función para la que fue diseñada.
- En muchos casos, también se encargará de la prueba de integración de todos los elementos en la estructura total del sistema.

Grupo independiente de prueba

- Sólo una vez que la arquitectura del software esté completa, entra en juego un **grupo independiente de prueba**, que debe eliminar los problemas inherentes asociados con el hecho de permitir al constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría presente.
- En cualquier caso, el desarrollador y el grupo independiente deben trabajar estrechamente a lo largo del proyecto de software para asegurar que se realizan pruebas exhaustivas.
- Mientras se dirige la prueba, el desarrollador debe estar disponible para corregir los errores que se van descubriendo.

Principios fundamentales

- Hay 6 principios fundamentales respecto a las metodologías de pruebas que deben quedar claros desde el primer momento aunque volveremos a ellos continuamente:
  - Las pruebas exhaustivas no son viables
  - Ejecución de pruebas bajo diferentes condiciones
  - El proceso de pruebas no puede demostrar la ausencia de defectos
  - Las pruebas no garantizan ni mejoran la calidad del software
  - Las pruebas tienen un coste
  - Inicio temprano de pruebas

Las pruebas exhaustivas no son viables

- Es imposible, inviable, crear casos de prueba que cubran todas las posibles combinaciones de entrada y salida que pueden llegar a tener las funcionalidades (salvo que sean triviales).
- Por otro lado, en proyectos cuyo número de casos de uso o historias de usuario desarrollados sea considerable, se requeriría de una inversión muy alta en cuanto a recursos y tiempo necesarios para cubrir con pruebas todas las funcionalidades del sistema.
- Por lo tanto es conveniente realizar un análisis de riesgos de todas las funcionalidades y determinar en este punto cuales serán objeto de prueba y cuales no, creando pruebas que cubran el mayor número de casos de prueba posibles.

Ejecución de pruebas bajo diferentes condiciones

- El plan de pruebas determina la condiciones y el número de ciclos de prueba que se ejecutarán sobre las funcionalidades del negocio.
- Por cada ciclo de prueba, se generan diferentes tipos de condiciones, basados principalmente en la variabilidad de los datos de entrada y en los conjuntos de datos utilizados.
- No es conveniente, ejecutar en cada ciclo, los casos de prueba basados en los mismos datos del ciclo anterior, dado que con mucha probabilidad, se obtendrán los mismos resultados.
- Ejecutar ciclos bajo diferentes tipos de condiciones, permitirá identificar posibles fallos en el sistema que antes no se detectaron y no son fácilmente reproducibles.

El proceso no puede demostrar la ausencia de defectos

- Independientemente de la rigurosidad con la que se haya planeado el proceso de pruebas de un producto, nunca será posible garantizar al ejecutar este proceso, la ausencia total de defectos (es inviable una cobertura del 100%).
- Una prueba se considera un éxito si detecta un error. Si no detecta un error no significa que no haya error, significa que no se ha detectado.
- Un proceso de pruebas riguroso puede garantizar una reducción significativa de los posibles fallos y/o defectos del software, pero nunca podrá garantizar que el software no fallará en producción.

Las pruebas no garantizan ni mejoran la calidad del software

- Las pruebas ayudan a **mejorar la percepción** de la calidad permitiendo la eliminación de los defectos detectados.
- La calidad del software viene determinada por las metodologías y buenas practicas empleadas en el desarrollo del mismo.
- Las pruebas **permiten medir la calidad** del software, lo que permite, a su vez, mejorar los procesos de desarrollo que son los que conllevan la mejora de la calidad y permiten garantizar un nivel determinado de calidad.

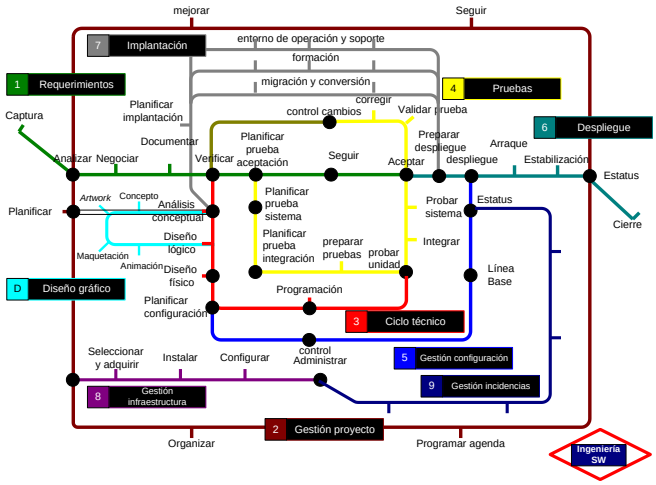
Las pruebas tienen un coste

- Aunque exige dedicar esfuerzo (coste para las empresas) para crear y mantener los test, los beneficios obtenidos son mayores que la inversión realizada.
- La creciente inclusión del software como un elemento más de muchos sistemas productivos y la importancia de los "costes" asociados a un fallo del mismo están motivando la creación de pruebas minuciosas y bien planificadas.
- No es raro que una organización de desarrollo de software gaste el 40 por 100 del esfuerzo total de un proyecto en la prueba.
- En casos extremos, la prueba del software para actividades críticas (por ejemplo, control de tráfico aéreo, o control de reactores nucleares) puede costar ¡de 3 a 5 veces más que el resto de los pasos de la ingeniería del software juntos!
- El coste de hacer las pruebas es siempre inferior al coste de no hacer las pruebas (deuda técnica).

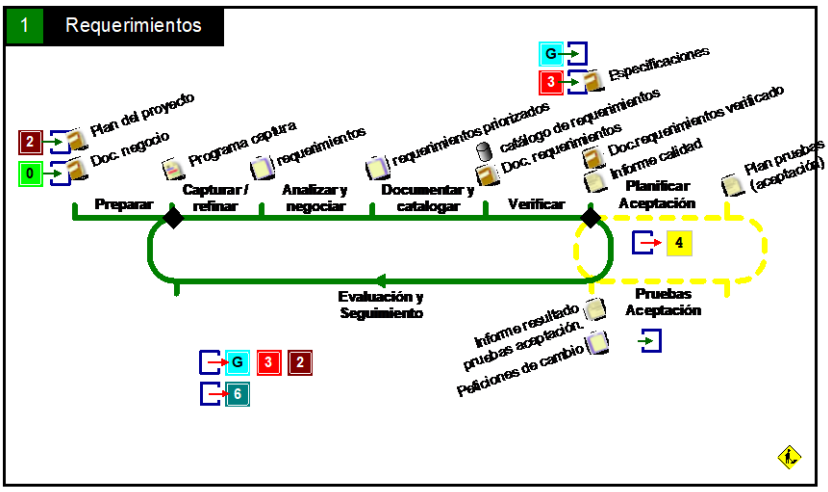
Inicio temprano de pruebas

- Si bien la fase de pruebas es la última del ciclo de vida, las actividades del proceso de pruebas deben ser incorporadas desde la fase de especificación, incluso antes de que se ejecutan las etapas de análisis y diseño.
- De esta forma los documentos de especificación y de diseño deben ser sometidos a revisiones y validaciones, lo que ayudará a detectar problemas en la lógica del negocio mucho antes de que se escriba una sola línea de código.
- Cuanto mas temprano se detecte un defecto, ya sea sobre los entregables de especificación, diseño o sobre el producto, menor impacto tendrá en el desarrollo y menor será el costo de dar solución a dichos defectos.

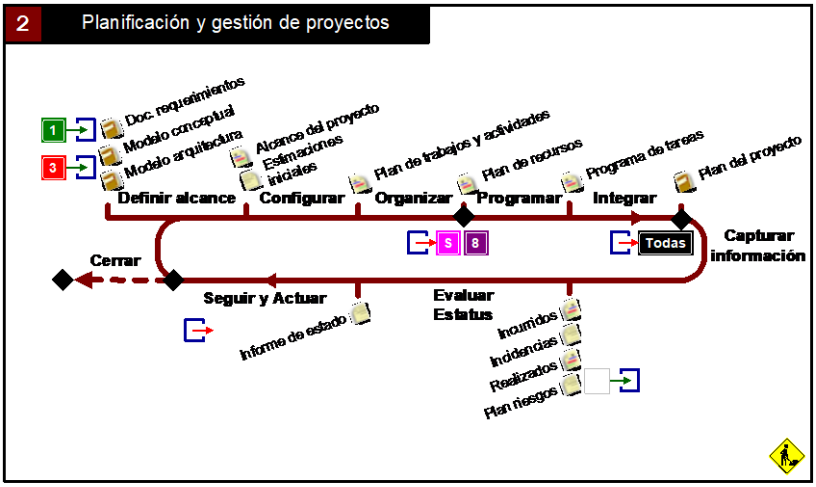
Integración en el ciclo de vida
Símil del mapa del Metro



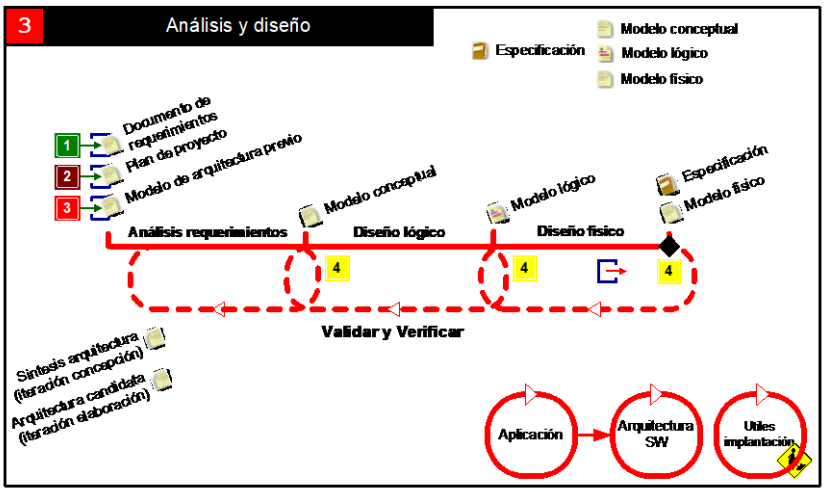
Línea 1: Requerimientos



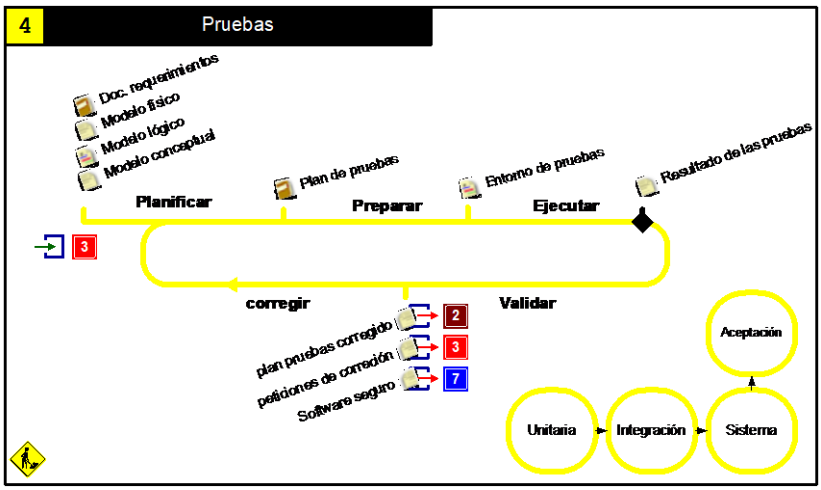
Línea 2: Planificación y gestión



Línea 3: Análisis y diseño



Línea 4: Pruebas

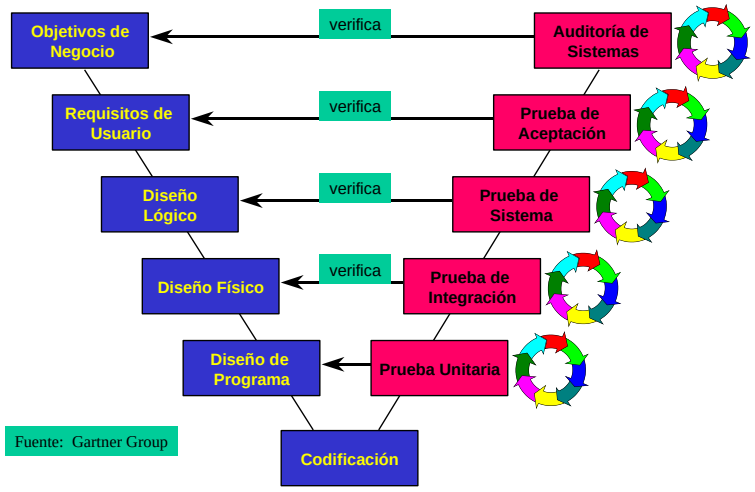


EL PROCESO DE PRUEBAS

El proceso

- Excepto para programas pequeños, los sistemas no deberían probarse como un único elemento indivisible.
- Los sistemas grandes se construyen a partir de subsistemas que se construyen a partir de módulos, compuestos de funciones y procedimientos.
- El proceso de prueba debería trabajar por etapas, llevando a cabo la prueba de forma incremental a la vez que se realiza la implementación del sistema, siguiendo el modelo en V.

Proceso de pruebas: Ciclo en V



Ciclo en V

- El modelo en V establece una simetría entre las fases de desarrollo y las pruebas.
- Las principales consideraciones se basan en la inclusión de las actividades de planificación y ejecución de pruebas como parte del proyecto de desarrollo.
- Inicialmente, la ingeniería del sistema define el papel del software y conduce al análisis de los requisitos del software, donde se establece el campo de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Al movernos hacia abajo, llegamos al diseño y, por último, a la codificación, el vértice de la V.
- Para desarrollar las pruebas seguimos el camino ascendente por la otra rama de la V.
- Partiendo de los elementos más básicos, probamos que funcionan como deben (lo que hacen, lo hacen bien). Los combinamos y probamos que siguen funcionando como deben. Para terminar probamos que hacen lo que deben (que hacen todo lo que tienen que hacer).

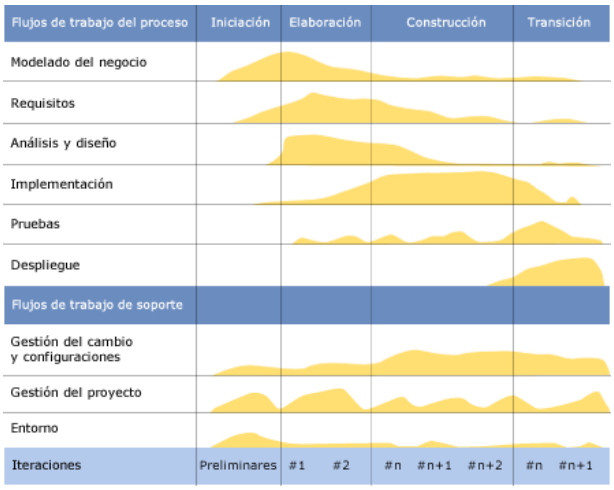
Desarrollo iterativo e incremental

- El desarrollo incremental implica establecer requisitos, diseñar, construir y probar un sistema en fragmentos, lo que significa que las prestaciones del software crecen de forma incremental. El tamaño de estos incrementos de prestaciones varía, ya que algunos métodos tienen piezas más grandes y otros más pequeñas. Los incrementos de prestaciones pueden ser tan pequeños como un simple cambio en una pantalla de la interfaz de usuario o una nueva opción en una consulta.
- El desarrollo iterativo se produce cuando se especifican, diseñan, construyen y prueban conjuntamente grupos de prestaciones en una serie de ciclos, a menudo, de una duración fija. Las iteraciones pueden implicar cambios en las prestaciones desarrolladas en iteraciones anteriores, junto con cambios en el alcance del proyecto. Cada iteración proporciona software operativo, que es un subconjunto creciente del conjunto general de prestaciones hasta que se entrega el software final o se detiene el desarrollo.

Desarrollo iterativo e incremental

- Rational Unified Process (RUP):
  - Cada iteración tiende a ser relativamente larga (por ejemplo, de dos a tres meses), y los incrementos de las prestaciones son proporcionalmente grandes, como por ejemplo dos o tres grupos de prestaciones relacionadas.
- Scrum:
  - Cada iteración tiende a ser relativamente corta (por ejemplo, horas, días o unas pocas semanas), y los incrementos de las prestaciones son proporcionalmente pequeños, como unas pocas mejoras y/o dos o tres prestaciones nuevas.
- Kanban:
  - Implementado con o sin iteraciones de longitud fija, que puede ofrecer una sola mejora o prestación una vez finalizada, o puede agrupar prestaciones para liberarlas de una sola vez.
- Espiral (o prototipado):
  - Implica la creación de incrementos experimentales, algunos de los cuales pueden ser reelaborados en profundidad o incluso abandonados en trabajos de desarrollo posteriores.

RUP



Clasificación de Pruebas

- Las actividades de las pruebas pueden centrarse en comprobar el sistema en base a un objetivo o motivo especifico:
  - Una función a realizar por el software.
  - Una característica no funcional como el rendimiento o la fiabilidad.
  - La estructura o arquitectura del sistema o el software.
  - Los cambios para confirmar que se han solucionado los defectos o localizar los no intencionados.
- Las pruebas se pueden clasificar como:
  - Pruebas funcionales
  - Pruebas no funcionales
  - Pruebas estructurales
  - Pruebas de mantenimiento

Niveles de pruebas

- **Pruebas Unitarias o de Componentes**: verifican la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- **Pruebas de Integración**: verifican el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- **Pruebas de Regresión**: verifican que los cambios sobre un componente de un sistema de información no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- **Pruebas del Sistema**: ejercitan profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- **Pruebas de Aceptación**: validan que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.

Pruebas Unitarias

- Las pruebas unitarias tienen como objetivo verificar la funcionalidad y estructura de cada componente individualmente, una vez que ha sido codificado.
- Con las pruebas unitarias verificas el diseño de los programas, vigilando que no se producen errores y que el resultado de los programas es el esperado.
- Estas pruebas las efectúa normalmente la misma persona que codifica o modifica el componente y que, también normalmente, genera un juego de ensayo para probar y depurar las condiciones de prueba.
- Las pruebas unitarias constituyen la prueba inicial de un sistema y las demás pruebas deben apoyarse sobre ellas.

Pruebas Unitarias

- Existen dos **enfoques** principales para el diseño de casos de prueba:
  - **Enfoque estructural o de caja blanca**. Se verifica la estructura interna del componente con independencia de la funcionalidad establecida para el mismo.

    Por tanto, no se comprueba la corrección de los resultados, sólo si éstos se producen. Ejemplos de este tipo de pruebas pueden ser ejecutar todas las instrucciones del programa, localizar código no usado, comprobar los caminos lógicos del programa, etc.
  - **Enfoque funcional o de caja negra**. Se comprueba el correcto funcionamiento de los componentes del sistema de información, analizando las entradas y salidas y verificando que el resultado es el esperado. Se consideran exclusivamente las entradas y salidas del sistema sin preocuparse por la estructura interna del mismo.
- El enfoque que suele adoptarse para una prueba unitaria está claramente orientado al diseño de casos de caja blanca, aunque se complemente con caja negra.

Pruebas Unitarias

- Los **pasos necesarios** para llevar a cabo las pruebas unitarias son los siguientes:
  - **Ejecutar todos los casos de prueba** asociados a cada verificación establecida en el plan de pruebas, registrando su resultado. Los casos de prueba deben contemplar tanto las condiciones válidas y esperadas como las inválidas e inesperadas.
  - **Corregir los errores o defectos encontrados y repetir las pruebas que los detectaron**. Si se considera necesario, debido a su implicación o importancia, se repetirán otros casos de prueba ya realizados con anterioridad.

Pruebas Unitarias

- La prueba unitaria se da por finalizada cuando se hayan realizado todas las verificaciones establecidas y no se encuentre ningún defecto, o bien se determine su suspensión.
- Al finalizar las pruebas, obtienes las **métricas de calidad del componente** y las contrastas con las existentes antes de la modificación:
  - Número ciclomático.
  - Cobertura de código.
  - Porcentaje de comentarios.
  - Defectos hallados contra especificaciones o estándares.
  - Rendimientos.

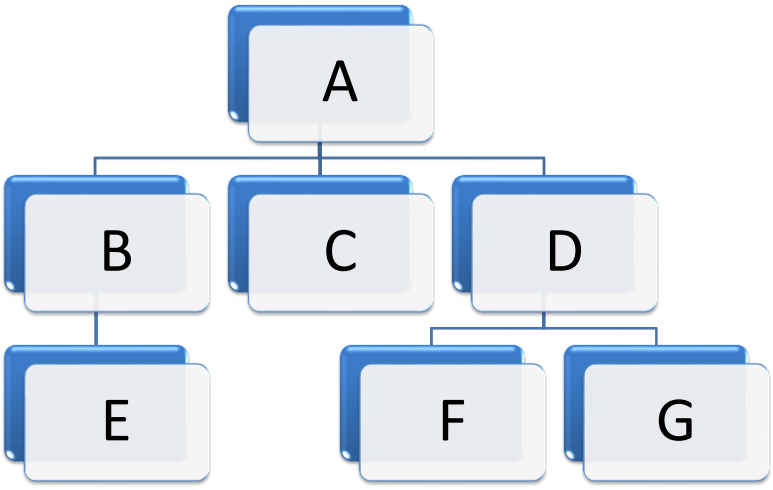
Pruebas de Integración

- Las pruebas de integración te permiten verificar el correcto ensamblaje entre los distintos componentes una vez que han sido probados unitariamente, con el fin de comprobar que interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida y se ajustan a los requisitos no funcionales especificados en las verificaciones correspondientes.
- Se trata de probar los caminos lógicos del flujo de los datos y mensajes a través de un conjunto de componentes relacionados que definen una cierta funcionalidad.
- En las pruebas de integración examinas las interfaces entre grupos de componentes o subsistemas para asegurar que son llamados cuando es necesario y que los datos o mensajes que se transmiten son los requeridos.
- Debido a que en las pruebas unitarias es necesario crear módulos auxiliares que simulen las acciones de los componentes invocados por el que se está probando, y a que se han de crear componentes "conductores" para establecer las precondiciones necesarias, llamar al componente objeto de la prueba y examinar los resultados de la prueba, a menudo se combinan los tipos de prueba unitarias y de integración.

Pruebas de Integración

- Los **tipos fundamentales de integración** son los siguientes:
  - **Integración incremental:** combinas el siguiente componente que debes probar con el conjunto de componentes que ya están probados y vas incrementando progresivamente el número de componentes a probar.
  - **Integración no incremental:** pruebas cada componente por separado y, luego, los integras todos de una vez realizando las pruebas pertinentes. Este tipo de integración se denomina también Big-Bang (gran explosión).
- Con el tipo de prueba incremental lo más probable es que los problemas te surjan al incorporar un nuevo componente o un grupo de componentes al previamente probado. Los problemas serán debidos a este último o a las interfaces entre éste y los otros componentes.

Pruebas de Integración



Estrategias de integración

- **De arriba a abajo (top-down):** el primer componente que se desarrolla y prueba es el primero de la jerarquía (A).
  - Los componentes de nivel más bajo se sustituyen por componentes auxiliares o resguardos, para simular a los componentes invocados. En este caso no son necesarios componentes conductores.
  - Una de las ventajas de aplicar esta estrategia es que las interfaces entre los distintos componentes se prueban en una fase temprana y con frecuencia.
- **De abajo a arriba (bottom-up):** en este caso se crean primero los componentes de más bajo nivel (E, F, G) y se crean componentes conductores para simular a los componentes que los llaman.
  - A continuación se desarrollan los componentes de más alto nivel (B, C, D) y se prueban. Por último dichos componentes se combinan con el que los llama (A). Los componentes auxiliares son necesarios en raras ocasiones.
  - Este tipo de enfoque permite un desarrollo más en paralelo que el enfoque de arriba a abajo, pero presenta mayores dificultades a la hora de planificar y de gestionar.
- **Estrategias combinadas:** A menudo es útil aplicar las estrategias anteriores conjuntamente. De este modo, se desarrollan partes del sistema con un enfoque "top-down", mientras que los componentes más críticos en el nivel más bajo se desarrollan siguiendo un enfoque "bottom-up".
  - En este caso es necesaria una planificación cuidadosa y coordinada de modo que los componentes individuales se "encuentren" en el centro.

Pruebas de Regresión

- El objetivo de las pruebas de regresión es eliminar el efecto onda, es decir, comprobar que los cambios sobre un componente de un sistema de información, no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error como para realizar una mejora.
- No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.
- Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

Pruebas de Regresión

- Las pruebas de regresión **pueden incluir**:
  - La repetición de los casos de pruebas que se han realizado anteriormente y están directamente relacionados con la parte del sistema modificada.
  - La revisión de los procedimientos manuales preparados antes del cambio, para asegurar que permanecen correctamente.
  - La obtención impresa del diccionario de datos de forma que se compruebe que los elementos de datos que han sufrido algún cambio son correctos.
- El **responsable** de realizar las pruebas de regresión será el equipo de desarrollo junto al técnico de mantenimiento, quién a su vez, será responsable de especificar el plan de pruebas de regresión y de evaluar los resultados de dichas pruebas.

Pruebas del Sistema

- Las pruebas del sistema tienen como objetivo ejercitar profundamente el sistema comprobando la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.
- Son pruebas de integración del sistema de información completo, y permiten probar el sistema en su conjunto y con otros sistemas con los que se relaciona para verificar que las especificaciones funcionales y técnicas se cumplen. Dan una visión muy similar a su comportamiento en el entorno de producción.
- Una vez que se han probado los componentes individuales y se han integrado, se prueba el sistema de forma global. En esta etapa pueden distinguirse diferentes tipos de pruebas, cada uno con un objetivo claramente diferenciado.

Pruebas del Sistema

- **Pruebas funcionales**: dirigidas a asegurar que el sistema de información realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.
- **Pruebas de humo**: son un conjunto de pruebas aplicadas a cada nueva versión, su objetivo es validar que las funcionalidades básicas de la versión se cumplen según lo especificado. Impiden la ejecución el plan de pruebas si detectan grandes inestabilidades o si elementos clave faltan o son defectuosos.
- **Pruebas de comunicaciones**: determinan que las interfaces entre los componentes del sistema funcionan adecuadamente, tanto a través de dispositivos remotos, como locales. Asimismo, se han de probar las interfaces hombre-máquina.
- **Pruebas de rendimiento**: consisten en determinar que los tiempos de respuesta están dentro de los intervalos establecidos en las especificaciones del sistema.
- **Pruebas de volumen**: consisten en examinar el funcionamiento del sistema cuando está trabajando con grandes volúmenes de datos, simulando las cargas de trabajo esperadas.
- **Pruebas de sobrecarga o estrés**: consisten en comprobar el funcionamiento del sistema en el umbral límite de los recursos, sometiéndole a cargas masivas. El objetivo es establecer los puntos extremos en los cuales el sistema empieza a operar por debajo de los requisitos establecidos.

Pruebas del Sistema

- **Pruebas de disponibilidad de datos**: consisten en demostrar que el sistema puede recuperarse ante fallos, tanto de equipo físico como lógico, sin comprometer la integridad de los datos.
- **Pruebas de configuración**: consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **Pruebas de usabilidad**: consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.
- **Pruebas extremo a extremo (e2e)**: consisten en interactuar con la aplicación como un usuario regular lo haría, cliente-servidor, y evaluando las respuestas para el comportamiento esperado.
- **Pruebas de operación**: consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
- **Pruebas de entorno**: consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.
- **Pruebas de seguridad**: consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

Pruebas de Aceptación

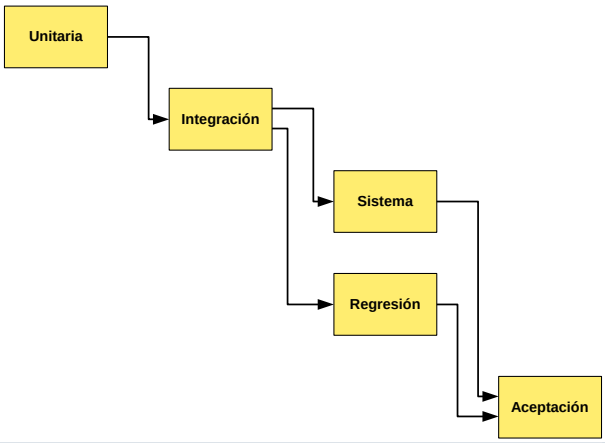
- El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema, que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.
- Las pruebas de aceptación son preparadas por el usuario del sistema y el equipo de desarrollo, aunque la ejecución y aprobación final corresponde al usuario.
- Estas pruebas van dirigidas a comprobar que el sistema cumple los requisitos de funcionamiento esperado recogidos en el catálogo de requisitos y en los criterios de aceptación del sistema de información, y conseguir la aceptación final del sistema por parte del usuario.

Pruebas de Aceptación

- Previamente a la realización de las pruebas, el responsable de usuarios revisa los criterios de aceptación que se especificaron previamente en el plan de pruebas del sistema y dirige las pruebas de aceptación final.
- La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba asociados.
- Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta, a su vez, los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.
- La formalidad de estas pruebas dependerá en mayor o menor medida de cada organización, y vendrá dada por la criticidad del sistema, el número de usuarios implicados en las mismas y el tiempo del que se disponga para llevarlas cabo, entre otros.

Niveles de pruebas y orden de ejecución.

- De tal forma que la secuencia de pruebas es:



Prueba exploratoria

- Incluso los esfuerzos de automatización de pruebas más diligentes no son perfectos. A veces se pierden ciertos casos extremos en sus pruebas automatizadas. A veces es casi imposible detectar un error en particular escribiendo una prueba unitaria. Ciertos problemas de calidad ni siquiera se hacen evidentes en las pruebas automatizadas (como en el diseño o la usabilidad).
- Las pruebas exploratorias es un enfoque de prueba manual que enfatiza la libertad y creatividad del probador para detectar problemas de calidad en un sistema en ejecución.
  - Simplemente tomate un tiempo en un horario regular, arremángate e intenta romper la aplicación.
  - Usa una mentalidad destructiva y encuentra formas de provocar problemas y errores en la aplicación.
  - Ten en cuenta los errores, los problemas de diseño, los tiempos de respuesta lentos, los mensajes de error faltantes o engañosos y, en general, todo lo que pueda molestarte como usuario de una aplicación.
  - Documenta todo lo que encuentre para más adelante.
- La buena noticia es que se puede automatizar la mayoría de los hallazgos con pruebas automatizadas. Escribir pruebas automatizadas para los errores que se detectan asegura que no habrá regresiones a ese error en el futuro. Además, ayuda a reducir la causa raíz de ese problema durante la corrección de errores.

Pruebas de mutaciones

- Los pruebas de mutaciones son las pruebas de las pruebas unitarias y el objetivo es tener una idea de la calidad de las pruebas en cuanto a fiabilidad.
- Su funcionamiento es relativamente sencillo: la herramienta que se utilice debe generar pequeños cambios en el código fuente. A estos pequeños cambios se les conoce como mutaciones y crean mutantes.
- Una vez creados los mutantes, se lanzan todos los tests:
  - Si los test unitarios fallan, es que han sido capaces de detectar ese cambio de código. En este caso el mutante se considera eliminado.
  - Si, por el contrario, los test unitarios pasan, el mutante sobrevive y la fiabilidad (y calidad) de los tests unitarios queda en entredicho.
- Los test de mutaciones presentan informes del porcentaje de mutantes detectados: cuanto más se acerque este porcentaje al 100%, mayor será la calidad de nuestros test unitarios.

Pirámide de pruebas

Aprender con pruebas unitarias

- La incorporación de código de tercero es complicado, hay que aprenderlo primero e integrarlo después. Hacer las dos cosa a la vez es el doble de complicado.
- Utilizar pruebas unitarias en el proceso de aprendizaje (*pruebas de aprendizaje* según Jim Newkirk) aporta importantes ventajas:
  - La inmediatez de las pruebas unitarias y sus entornos.
  - Realizar “pruebas de concepto” para comprobar si el comportamiento se corresponde con lo que hemos entendido, permitiéndonos clarificarlo.
  - Experimentar para encontrar los mejores escenarios de integración.
  - Permite saber si un fallo es nuestro, de la librería o del uso inadecuado de la librería.

Pruebas de aprendizaje

- Las pruebas de aprendizaje no suponen un coste adicional, es parte del coste de aprendizaje que, en todo caso, lo minoran.
- Es mas, las pruebas de aprendizaje son rentables. Ante la aparición de nuevas versiones del código ajeno, ejecutar la batería de pruebas de aprendizaje valida el impacto de la adopción de la nueva versión: detecta cambios relevantes, efectos negativos en las integraciones, ...
- Las pruebas de aprendizaje no sustituyen al conjunto de pruebas que respaldan los limites establecidos.

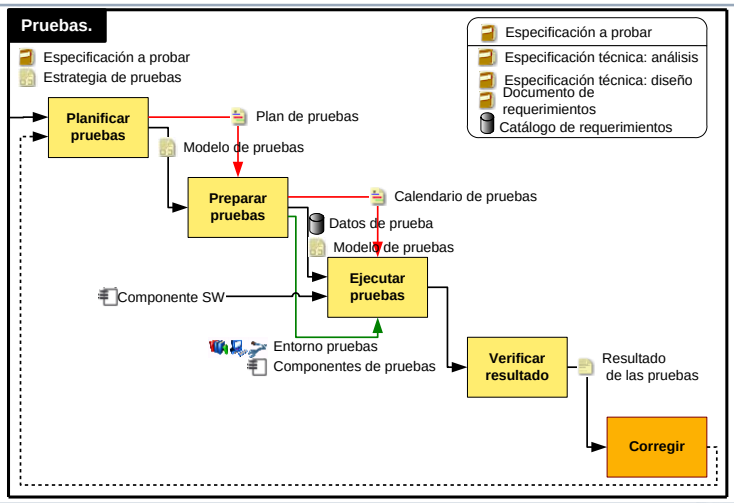
Principios F.I.R.S.T.

- El principio FIRST fue definido por Robert Cecil Martin en su libro Clean Code. Este autor, entre otras muchas cosas, es conocido por ser uno de los escritores del Agile Manifesto, escrito hace más de 15 años y que a día de hoy se sigue teniendo muy en cuenta a la hora de desarrollar software.
  - Fast: Los tests deben ser rápidos, del orden de milisegundos, hay cientos de tests en un proyecto.
  - Isolated/Independent (Aislado/Independiente). Los tests no deben depender del entorno ni de ejecuciones de tests anteriores.
  - Repeatable. Los tests deben ser repetibles y ante la misma entrada de datos, los mismos resultados.
  - Self-Validating. Los tests tienen que ser autovalidados, es decir, NO debe de existir la intervención humana en la validación
  - Thorough and Timely (Completo y oportuno). Los tests deben de cubrir el escenario propuesto, no el 100% del código, y se han de realizar en el momento oportuno

Metodología

- El primer paso que debes dar es la **planificación de la prueba**, pasando a continuación a su **diseño**.
- Para diseñar la prueba debes identificar y diseñar los casos de prueba de integración, los casos de prueba de sistema, los casos de prueba de regresión y los procedimientos de prueba.
- Una vez concluido el diseño, debes **implementar los componentes de pruebas necesarios**.
- Por último realizas las **pruebas de integración** y la **prueba de sistema**.
- Concluyes el proceso con la **evaluación de la prueba**.

Metodología



Diseñar la prueba

- Para diseñar la prueba empiezas por identificar y describir los casos de prueba de cada componente.
- La selección de las técnicas de pruebas depende factores adicionales como pueden ser requisitos contractuales o normativos, documentación disponible, tiempo, presupuesto, conocimientos, experiencia, …
- Cuando dispongas de los casos de prueba, identificas y estructuras los procedimientos de prueba describiendo cómo ejecutar los casos de prueba.

Patrones

- Los casos de prueba se pueden estructurar siguiendo diferentes patrones:
  - ARRANGE-ACT-ASSERT: Preparar, Actuar, Afirmar
  - GIVEN-WHEN-THEN: Dado, Cuando, Entonces
  - BUILD-OPERATE-CHECK: Generar, Operar, Comprobar
- Aunque con diferencias conceptuales, todos dividen el proceso en tres fases:
  - Una fase inicial donde montar el escenario de pruebas que hace que el resultado sea predecible.
  - Una fase intermedia donde se realizan las acciones que son el objetivo de la prueba.
  - Una fase final donde se comparan los resultados con el escenario previsto. Pueden tomar la forma de:
    - Aserción: Es una afirmación sobre el resultado que puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

Preparación mínima

- La sección de preparación, con la entrada del caso de prueba, debe ser lo más sencilla posible, lo imprescindible para comprobar el comportamiento que se está probando.
- Las pruebas se hacen más resistentes a los cambios futuros en el código base y más cercano al comportamiento de prueba que a la implementación.
- Las pruebas que incluyen más información de la necesaria tienen una mayor posibilidad de incorporar errores en la prueba y pueden hacer confusa su intención. Al escribir pruebas, el usuario quiere centrarse en el comportamiento. El establecimiento de propiedades adicionales en los modelos o el empleo de valores distintos de cero cuando no es necesario solo resta de lo que se quiere probar.

Actuación mínima

- Al escribir las pruebas hay que evitar introducir condiciones lógicas como if, switch, while, for, etc.
- Minimiza la posibilidad de incorporar un error a las pruebas.
- El foco está en el resultado final, en lugar de en los detalles de implementación.
- Al incorporar lógica al conjunto de pruebas, aumenta considerablemente la posibilidad de agregar un error. Cuando se produce un error en una prueba, se quiere saber realmente que algo va mal con el código probado y no en el código que prueba. En caso contrario, restan confianza y las pruebas en las que no se confía no aportan ningún valor.
- El objetivo de la prueba debe ser único, si la lógica en la prueba parece inevitable, denota que el objetivo es múltiple y hay que considerar la posibilidad de dividirla en dos o más pruebas diferentes.

Comprobación mínima

- Al escribir las pruebas, hay que intentar comprobar una única cosa, es decir, incluir solo una aserción por prueba.
  - Si se produce un error en una aserción, no se evalúan las aserciones posteriores.
  - Garantiza que no se estén declarando varios casos en las pruebas.
  - Proporciona la imagen exacta de por qué se producen errores en las pruebas.
- Al incorporar varias aserciones en un caso de prueba, no se garantiza que se ejecuten todas. Es un todas o ninguna, se sabe por cual fallo pero no si el resto también falla o es correcto, proporcionando la imagen parcial.
- Una excepción común a esta regla es cuando la validación cubre varios aspectos. En este caso, suele ser aceptable que haya varias aserciones para asegurarse de que el resultado está en el estado que se espera que esté.
- Los enfoques comunes para usar solo una aserción incluyen:
  - Crear una prueba independiente para cada aserción.
  - Usar pruebas con parámetros.

Diseñar para probar

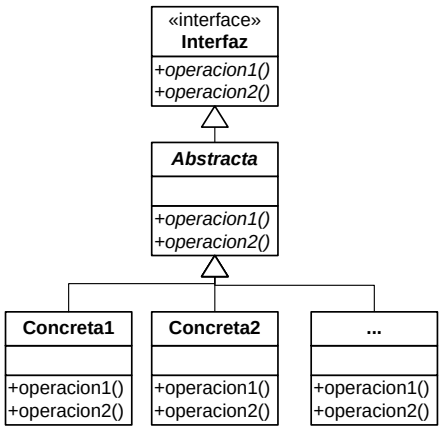
- Patrones:
  - Doble herencia
  - Inversión de Control
  - Inyección de Dependencias
  - Modelo Vista Controlador (MVC)
  - Model View ViewModel (MVVM)
- Metodologías:
  - Desarrollo Guiado por Pruebas (TDD)
  - Desarrollo Dirigido por Comportamiento (BDD)

Programar con interfaces

- La herencia de clase define la implementación de una clase a partir de otra (excepto métodos abstractos). La implementación de interfaz define como se llamara el método o propiedad, pudiendo escribir distinto código en clases no relacionadas.
- Reutilizar la implementación de la clase base es la mitad de la historia.
- Programar para las interfaz, no para la herencia. Favorecer la composición antes que la herencia.
- Ventajas:
  - Reducción de dependencias.
  - El cliente desconoce la implementación.
  - La vinculación se realiza en tiempo de ejecución.
  - Da consistencia (contrato).
- Desventaja:
  - Indireccionamiento.

Doble Herencia

- Problema:
  - Mantener las clases que implementan como internas del proyecto (internal o Friend), pero la interfaz pública.
  - Organizar clases que tienen un comportamiento parecido para que sea consistente.
- Solución:
  - Clase base es abstracta.
  - La clase base puede heredar de mas de una interfaz.
  - Una vez que están escritos los métodos, verifico si hay duplicación en las clases hijas.



Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto con unas técnicas de programación:
  - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
  - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. Técnicas de implementación:
  - Service Locator: es un componente (contenedor) que contiene referencias a los servicios y encapsula la lógica que los localiza dichos servicios.
  - Inyección de dependencias.

Inyección de Dependencias

- Las dependencias son expresadas en términos de interfaces en lugar de clases concretas y se resuelven dinámicamente en tiempo de ejecución.
- La Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java o .NET, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del Reflection y crear una instancia de dicha clase inyectándole sus dependencias.

Simulación de objetos

- Las dependencias con sistemas externos afectan a la complejidad de la estrategia de pruebas, ya que es necesario contar con sustitutos de estos servicios externos durante el desarrollo. Ejemplos típicos de estas dependencias son Servicios Web, Sistemas de envío de correo, Fuentes de Datos o simplemente dispositivos hardware.
- Estos sustitutos, muchas veces son exactamente iguales que el servicio original, pero en otro entorno o son simuladores que exponen el mismo interfaz pero realmente no realizan las mismas tareas que el sistema real, o las realizan contra un entorno controlado.
- Para poder emplear la técnica de simulación de objetos se debe diseñar el código a probar de forma que sea posible trabajar con los objetos reales o con los objetos simulados:
  - Doble herencia
  - IoC: Inversión de Control (Inversion Of Control)
  - DI: Inyección de Dependencias (Dependency Injection)
  - Objetos Mock

Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

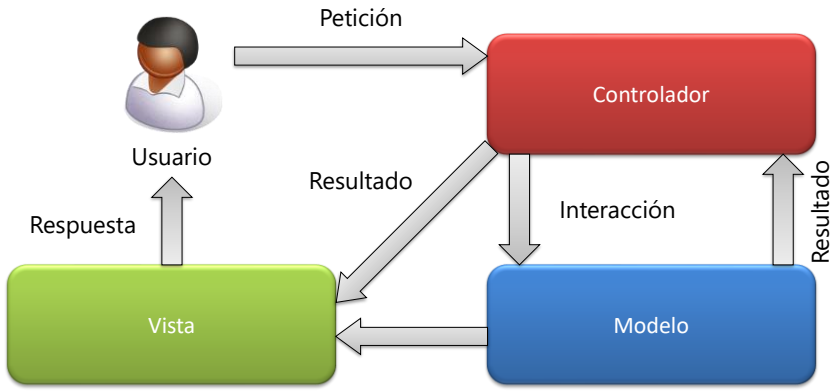
Dobles de prueba

- La forma de establecer los valores esperados y “memorizar” el valor con el que se ha llamado al simulador para posteriormente verificarlo se ha generalizado dando lugar a un marco de trabajo que permite definir objetos simulados sin necesidad de crear explícitamente el código que verifica cada uno de los valores.
- Los dobles de prueba son objetos que siempre realizan las mismas tareas:
  - Implementan un interfaz dado
  - Permiten establecer los valores esperados (tanto de entrada como de salida)
  - Permiten establecer el comportamiento (para lanzar excepciones en casos concretos)
  - Memorizan los valores con los que se llama a cada uno de sus miembros
  - Permiten verificar si los valores esperados coinciden con los recibidos

Dobles de prueba

- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: Objeto que tiene una implementación que realmente funciona pero, por lo general, usa una simplificación que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: Objeto que proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas (**spy**).
- **Mock**: Objeto pre programado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

El patrón MVC



El patrón MVC

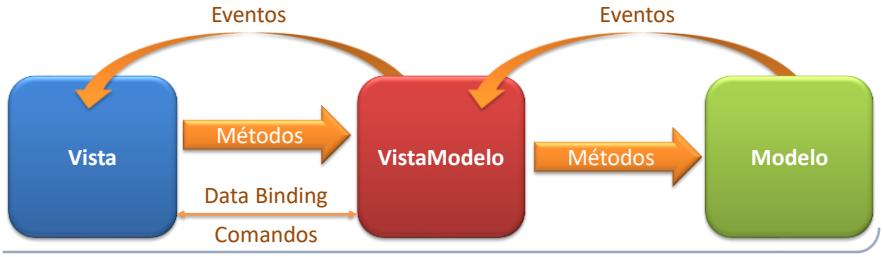
M Modelo
- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

V Vista
- **Interfaz** de usuario
- Incluye elementos de **interacción**

C Controlador
- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



¿Cuáles son los beneficios del patrón MVVM?

- Separación de vista / presentación.
- Permite las pruebas unitarias: como la lógica de presentación está separada de la vista, podemos realizar pruebas unitarias sobre la VistaModelo.
- Mejora la reutilización de código.
- Soporte para manejar datos en tiempo de diseño.
- Múltiples vistas: la VistaModelo puede ser presentada en múltiples vistas, dependiendo del rol del usuario por ejemplo.

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus reglas básicas:
  - No añadir código sin escribir antes una prueba que falle
  - Eliminar el Código Duplicado empleando Refactorización

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
  1. Escribir una prueba que demuestre la necesidad de escribir código.
  2. Escribir el mínimo código para que el código de pruebas compile
  3. Implementar exclusivamente la funcionalidad demandada por las pruebas
  4. Mejorar el código (Refactoring) sin añadir funcionalidad
  5. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y que se puede probar automáticamente.

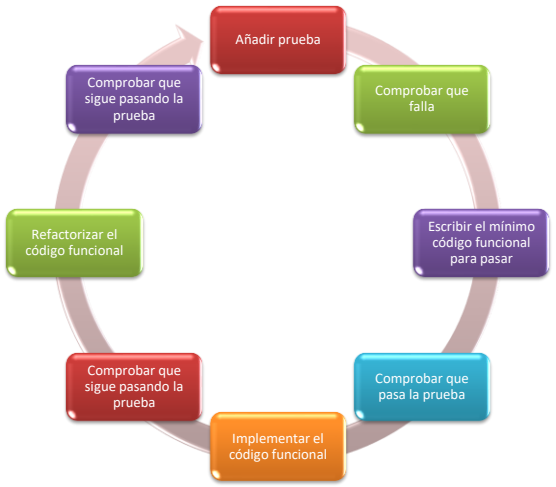
Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el código:
  - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
  - Protegen ante errores de regresión (rollbacks a versiones anteriores).
  - Dan confianza.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

Estrategia RED – GREEN

- Se recomienda una estrategia de test unitarios conocida como RED (fallo) – GREEN (éxito), es especialmente útil en equipos de desarrollo ágil.
- Una vez que entendamos la lógica y la intención de un test unitario, hay que seguir estos pasos:
  - Escribe el código del test (**Stub**) para que compile (pase de **RED** a **GEEN)**
    - Inicialmente la compilación fallará **RED** debido a que falta código
    - Implementa sólo el código necesario para que compile **GREEN** (aún no hay implementación real).
  - Escribe el código del test para que se **ejecute** (pase de **RED** a **GREEN)**
    - Inicialmente el test fallará **RED** ya que no existe funcionalidad.
    - Implementa la funcionalidad que va a probar el test hasta que se ejecute adecuadamente **GREEN.**
  - **Refactoriza** el test y el código una vez que este todo **GREEN** y la solución vaya evolucionando.

Ritmo TDD



Refactorizar el código en pruebas

- Una refactorización es un cambio que está pensado para que el código se ejecute mejor o para que sea más fácil de comprender.
- No está pensado para alterar el comportamiento del código y, por tanto, no se cambian las pruebas.
- Se recomienda realizar los pasos de refactorización independientemente de los pasos que amplían la funcionalidad.
- Mantener las pruebas sin cambios aporta la confianza de no haber introducido errores accidentalmente durante la refactorización.

Desarrollo Dirigido por Comportamiento (BDD)

- El Desarrollo Dirigido por Comportamiento (Behaviour Driver Development) es una evolución de TDD (Test Driven Development o Desarrollo Dirigido por Pruebas), el concepto de BDD fue inicialmente introducido por Dan North como respuesta a los problemas que surgían al enseñar TDD.
- En BDD también vamos a escribir las pruebas antes de escribir el código fuente, pero en lugar de pruebas unitarias, lo que haremos será escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Tras escribir las pruebas escribimos el código fuente de la funcionalidad que haga que estas pruebas pasen correctamente. Después refactorizamos el código fuente.
- Partiremos de historias de usuario, siguiendo el modelo “Como [rol] quiero [característica] para [los beneficios]”. A partir de aquí, en lugar de describir en 'lenguaje natural' lo que tiene que hacer esa nueva funcionalidad, vamos a usar un lenguaje ubicuo (un lenguaje semiformal que es compartido tanto por desarrolladores como personal no técnico) que nos va a permitir describir todas nuestras funcionalidades de una única forma.

BDD

- Para empezar a hacer BDD sólo nos hace falta conocer 5 palabras, con las que construiremos sentencias con las que vamos a describir las funcionalidades:
  - Feature (característica): Indica el nombre de la funcionalidad que vamos a probar. Debe ser un título claro y explícito. Incluimos aquí una descripción en forma de historia de usuario: “Como [rol] quiero [característica] para [los beneficios]”. Sobre esta descripción empezaremos a construir nuestros escenarios de prueba.
  - Scenario: Describe cada escenario que vamos a probar.
  - Given (dado): Provee el contexto para el escenario en que se va a ejecutar el test, tales como el punto donde se ejecuta el test, o prerequisitos en los datos. Incluye los pasos necesarios para poner al sistema en el estado que se desea probar.
  - When (cuando): Especifica el conjunto de acciones que lanzan el test. La interacción del usuario que acciona la funcionalidad que deseamos testear.
  - Then (entonces): Especifica el resultado esperado en el test. Observamos los cambios en el sistema y vemos si son los deseados.

Desarrollo Dirigido por Tests de Aceptación (ATDD)

- El Desarrollo Dirigido por Test de Aceptación (ATDD), técnica conocida también como Story Test-Driven Development (STDD), es una variación del TDD pero a un nivel diferente.
- Las pruebas de aceptación o de cliente son el criterio escrito de que un sistema cumple con el funcionamiento esperado y los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto. Es el punto de partida del desarrollo en cada iteración.
- ATDD/STDD es una forma de afrontar la implementación de una manera totalmente distinta a las metodologías tradicionales. Cambia el punto de partida, la forma de recoger y formalizar las especificaciones, sustituye los requisitos escritos en lenguaje natural (nuestro idioma) por historias de usuario con ejemplos concretos ejecutables de como el usuario utilizara el sistema, que en realidad son casos de prueba. Los ejemplos ejecutables surgen del consenso entre los distintos miembros del equipo y el usuario final.
- La lista de ejemplos (pruebas) de cada historia, se escribe en una reunión que incluye a dueños de producto, usuarios finales, desarrolladores y responsables de calidad. Todo el equipo debe entender qué es lo que hay que hacer y por qué, para concretar el modo en que se certifica que el sotfware lo hace.

ATDD

- El algoritmo o ritmo es el mismo de tres pasos que en el TDD practicado exclusivamente por desarrolladores pero a un nivel superior.
- En ATDD hay dos prácticas claves:
  - Antes de implementar (fundamental lo de antes de implementar) una necesidad, requisito, historia de usuario, etc., los miembros del equipo colaboran para crear escenarios, ejemplos, de cómo se comportará dicha necesidad.
  - Después, el equipo convierte esos escenarios en pruebas de aceptación automatizadas. Estas pruebas de aceptación típicamente se automatizan usando Selenium o similares, “frameworks” como Cucumber, etc.

Data Driven Testing (DDT)

- Se basa en la creación de tests para ejecutarse en simultáneo con sus conjuntos de datos relacionados en un framework. El framework provee una lógica de test reusable para reducir el mantenimiento y mejorar la cobertura de test. La entrada y salida (del criterio de test) pueden ser resguardados en uno o más lugares del almacenamiento central o bases de datos, el formato real y la organización de los datos serán específicos para cada caso.
- Todo lo que tiene potencial de cambiar (también llamado "variabilidad," e incluye elementos como el entorno, puntos de salida, datos de test, ubicaciones, etc) está separado de la lógica del test (scripts) y movido a un 'recurso externo'. Esto puede ser configuración o conjunto de datos de test. La lógica ejecutada en el script está dictada por los valores.
- Los datos incluyen variables usadas tanto para la entrada como la verificación de la salida. En casos avanzados (y maduros) los entornos de automatización pueden ser obtenidos desde algún sistema usando los datos reales o un "sniffer", el framework DDT por lo tanto ejecuta pruebas sobre la base de lo obtenido produciendo una herramienta de test automáticos para regresión.

TÉCNICAS

Introducción

- Durante las fases anteriores de definición y de desarrollo, has intentado construir el software partiendo de un concepto abstracto y llegando a una implementación tangible.
- A continuación llega la prueba, debes crear una serie de casos de prueba que intenten demoler el software que ha sido construido. De hecho, la prueba es uno de los pasos de la ingeniería del software que, por lo menos psicológicamente, se puede ver como destructivo en lugar de constructivo.
- Para comenzar vamos a establecer una serie de reglas que sirven como objetivos de prueba:
  1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
  2. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
  3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Introducción

- Los objetivos anteriores suponen un cambio dramático de punto de vista. Nos quitan la idea que, normalmente, tenemos de que una prueba tiene éxito si no descubre errores.
- El objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.
- Si la prueba se lleva a cabo con éxito (de acuerdo con el objetivo anteriormente establecido), descubrirá errores en el software.
- Como ventaja secundaria, la prueba demuestra hasta qué punto las funciones del software parecen funcionar de acuerdo con las especificaciones y parecen alcanzarse los requisitos de rendimiento. Además, los datos que se van recogiendo a medida que se lleva a cabo la prueba proporcionan una buena indicación de la fiabilidad del software y, de alguna manera, indican la calidad del software como un todo.
- Sin embargo, hay una cosa que no puede hacer la prueba: ”**La prueba no puede asegurar la ausencia de defectos, sólo puede demostrar que existen defectos en el software**”. Es importante tener en mente esta frase pesimista mientras se lleva a cabo la prueba.

Casos de prueba

- El diseño de pruebas para el software puede requerir tanto esfuerzo como el propio diseño inicial del producto.
- Recordando el objetivo de la prueba, debes diseñar pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y de tiempo.
- Un **caso de prueba** especifica una forma de probar el sistema, incluyendo la entrada con la que se ha de probar, los resultados que se esperan obtener y las condiciones bajo las que ha de probarse.

C.O.R.R.E.C.T?

- Conformance.
  - ¿Se muestran los valores de la forma esperada?
- Ordering.
  - ¿Se presentan los valores apropiadamente ordenados/desordenados?
- Range.
  - ¿Los valores generados están dentro del rango esperado [max, min]?
- Reference.
  - ¿Se refiere el código a algo externo que existe?
- Existence.
  - ¿Existe el valor (es no nulo), está presente en un array, etc?
- Cardinality.
  - ¿Se obtiene el número deseado de elementos o valores?
- Time.
  - ¿Ocurren las cosas cuando deben?

Casos de prueba

- Cualquier producto de ingeniería puede ser probado de una de estas dos formas:
  - **Conociendo la función específica** para la que fue diseñado el producto, se pueden llevar a cabo pruebas que demuestren que cada función es completamente operativa.
  - **Conociendo el funcionamiento del producto**, se pueden desarrollar pruebas que aseguren que "todas las piezas encajan"; o sea, que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.
- La primera forma, se denomina **prueba de caja negra** y la segunda **prueba de caja blanca**.

Casos de prueba

- La **prueba de caja negra** se refiere a las pruebas que se llevan a cabo sobre la interfaz del software.
- O sea, los casos de prueba pretenden demostrar que las funciones del software son operativas, que la entrada se acepta de forma adecuada y que se produce una salida correcta, así como que la integridad de la información externa se mantiene.
- Una prueba de caja negra examina algunos aspectos del modelo fundamental del sistema sin tener demasiado en cuenta la estructura lógica interna del software, siendo las **más adecuadas para probar los casos de uso del "modelo de casos de uso"**. Las pruebas de caja negra pueden ser ejecutadas por personal sin experiencia en ingeniería de software.

Casos de prueba

- La **prueba de caja blanca** del software se basa en el minucioso examen de los detalles procedimentales.
- Se comprueban los caminos lógicos del software proponiendo casos de prueba que ejerciten conjuntos específicos de condiciones y/o bucles. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el esperado o afirmado.
- A primera vista parecería que una prueba de caja blanca muy profunda nos llevaría a tener "programas 100% correctos". Todo lo que tienes que hacer es definir todos los caminos lógicos, desarrollar casos de prueba que los ejerciten y evaluar los resultados; es decir, generar casos de prueba que ejerciten exhaustivamente la lógica del programa.

Casos de prueba

- Desgraciadamente, la prueba exhaustiva presenta problemas. Incluso para pequeños programas, el número de caminos lógicos posibles puede ser enorme. La prueba exhaustiva es imposible para los grandes sistemas software. **Las pruebas de caja blanca están especialmente indicadas para validar las realizaciones de casos de uso del "modelo de diseño"**.
- La prueba de caja blanca, sin embargo, no se debe desechar como impracticable.
- Se puede elegir y ejercitar una serie de caminos lógicos importantes. Se pueden comprobar las estructuras de datos más importantes para ver su validez. Se pueden combinar los atributos de la prueba de caja blanca así con los de caja negra, para llegar a un método que valide la interfaz del software y asegure selectivamente que el funcionamiento interno del software es correcto.

Casos de prueba

- Las pruebas de caja blanca y caja negra enfocan las pruebas desde el punto de vista de la ejecución.
- Existe otro punto de vista, el del usuario, donde las pruebas deben ir enfocadas a buscar los errores producidos en **la interfaz con el usuario**.
- Al contrario de las pruebas que exigen la ejecución de software, las pruebas dinámicas, **las pruebas estáticas** se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Formalmente, los casos de prueba escritos consisten principalmente en tres secciones:
  - Identificación
  - Definición
  - Resultados

Identificación

- **Identificador**: Es un identificador único para futuras referencias, por ejemplo, mientras se describe un defecto encontrado.
- **Nombre**: El caso de prueba debe tener un título entendible por las personas, para facilitar la comprensión del propósito del caso de prueba y su campo de aplicación.
- **Propósito**: Contiene una breve descripción del propósito de la prueba y la funcionalidad probada.
- **Creador**: Es el nombre del analista o diseñador de pruebas, quien ha desarrollado pruebas o es responsable de su desarrollo.
- **Versión**: Número de la definición actual del caso de prueba.
- **Referencias**: Identificadores de requerimientos, casos de uso o de especificaciones funcionales que están cubiertos por el caso de prueba.
- **Dependencias**: Orden de ejecución de casos de pruebas, razón de las dependencias.

Definición

- **Precondiciones**: situación previa a la ejecución de pruebas o características de un objeto de pruebas antes de llevar a la práctica (ejecución) un caso de prueba.
- **Valores de entrada**: descripción de los datos de entrada de un objeto de pruebas.
- **Resultados esperados**: datos de salida que se espera que produzca un objeto de prueba.
- **Poscondiciones**: características de un objeto de prueba tras la ejecución de pruebas, descripción de su situación tras la ejecución de las pruebas.
- **Requisitos**: características del objeto de pruebas que el caso de prueba debe evaluar. Contiene información acerca de la configuración del hardware o software en el cuál se ejecutará el caso de prueba.
- **Acciones**: Pasos a realizar para completar la prueba.

Resultados

- **Salida obtenida**: Contiene una breve descripción de lo que el analista encuentra después de que los pasos de prueba se hayan completado.
- **Resultado**: Indica el resultado cualitativo de la ejecución del caso de prueba, a menudo con un Correcto/Fallido.
- **Severidad**: Indica el impacto del defecto en el sistema: Critico, Grave, Normal, Menor.
- **Evidencia**: En los casos que se aplica, contiene información donde se evidencia la salida obtenida y como reproducirla. Puede ir acompañado por un enlace, un pantallazo (screenshot), una consulta a base de datos, el contenido de un fichero de trazas …
- **Seguimiento**: Si un caso de prueba falla, frecuentemente la referencia al defecto implicado se debe enumerar. Contiene el código correlativo del defecto, a menudo corresponde al código del sistema de seguimiento de defectos que se esté usando.
- **Estado**: Indica si el caso de prueba está: No iniciado, En curso o Terminado.

Pruebas de Caja Negra

- La prueba de la caja negra es un método de diseño de casos de prueba que se centra en los requerimientos funcionales del software para descubrir los casos de prueba.
- Los errores que intentan encontrar son:
  - Funciones incorrectas o ausentes.
  - Errores de interfaz.
  - Errores de estructuras de datos.
  - Errores de rendimiento.
  - Errores de inicialización y de terminación.
- Van orientadas al dominio de información de los datos, es decir, al conjunto de diferentes valores de cada variable: de entrada / salida, de interfaz, variables de programa...

Técnica de la Partición Equivalente

- Trata de obtener un pequeño subconjunto de todas las posibles entradas con la mayor probabilidad de encontrar todos los errores.
- Todo caso de prueba bien seleccionado debe cumplir:
  - Reducir, en un coeficiente mayor que uno, el número de casos de prueba adicionales que se deben realizar para alcanzar una prueba razonable. Es decir, debe incluir tantas condiciones de entrada distintas como sea posible.
  - Cubrir un gran conjunto de posibles casos de prueba. Es decir, nos dice algo sobre la presencia o ausencia de un error asociado solamente con la prueba en particular que se encuentra disponible.
- Debemos intentar dividir el dominio de entrada de un programa en un número finito de clases de equivalencia tal, que se pueda asumir de forma razonable (aunque nunca estaremos totalmente seguros) que probar un valor representativo de cada clase es equivalente a probar algún otro valor.

Clases de equivalencia

- Es decir, si un caso de prueba en una clase de equivalencia detecta un error, todos los demás posibles casos de prueba en la clase de equivalencia deberán detectar el mismo error. Y también, si un caso de prueba no detecta un error, podríamos pensar que ningún otro caso de prueba en esa clase de equivalencia encontraría ese error.
- La **estrategia** a seguir se realiza en dos etapas:
  1. Identificar cada una de las clases de equivalencia.
  2. Definir los casos de prueba.
- Las **clases de equivalencia** las identificas dividiendo en dos o más grupos cada condición de entrada (normalmente ésta es una frase en la especificación). Se identifican siempre dos tipos de Clases de Equivalencia:
  1. Clases de equivalencia válidas.
  2. Clases de equivalencia inválidas (entradas erróneas).

Directrices para definir las clases de equivalencia

- Si una condición de entrada especifica un rango de valores (p. e. entre 1 y 999) se define una clase de equivalencia válida (1 <= valor <= 999) y dos inválidas (valor < 1 y valor > 999).
- Si una condición de entrada especifica una situación "debe ser" (por ejemplo: "el primer carácter del N.I.F. debe ser una letra"), se define una clase de equivalencia válida (es una letra) y otra inválida (no es una letra).
- Si una condición de entrada especifica un conjunto de valores de entrada y sabemos que cada valor va a tratarse de forma diferente en el programa, se define una clase de equivalencia válida para cada valor y una clase de equivalencia inválida (por ejemplo, "los colores pueden ser: ROJO, AMARILLO y VERDE", una clase inválida incluye "AZUL").
- Si una condición de entrada especifica el número de valores (ejemplo: una persona puede tener desde 1 hasta 3 nombres), identificar una clase de equivalencia válida y 2 inválidas (sin nombre o hay más de 3).
- Si tenemos una razón para creer que los elementos de una clase de equivalencia no se tratan de la misma manera en un programa, dividiremos la clase de equivalencia en clases más pequeñas.

Tabla de equivalencias

- Según vas identificando las condiciones y las clases de equivalencia, **rellenas la siguiente tabla:**

Condiciones de Entrada	Clases de equivalencia validas	Clases de equivalencia invalidas
La duración es	55	0 1278
El semáforo esta	“ROJO” ”AMARILLO” “VERDE"	“AZUL”
El NIF es	“12345678Z”	“Z12345678”

Definir los casos de prueba

- Para **definir los casos de prueba** das los siguientes **pasos**:
  1. Asignas un número único a cada clase de equivalencia.
  2. Escribes casos de prueba hasta que sean cubiertas todas las clases de equivalencia válidas, intentando cubrir en cada caso tantas clases de equivalencia como sea posible.
  3. Escribes casos de prueba hasta que sean cubiertas todas las clases de equivalencia inválidas cubriendo en cada caso una y sólo una clase de equivalencia aún no cubierta.
- La razón por la que las clases de equivalencia inválidas se cubren de forma individual se debe a que al detectar una entrada errónea seguramente no se chequea la siguiente entrada.

Ejemplo

- Tomemos como ejemplo de un módulo lee 3 valores enteros. Los 3 valores son la longitud de los 3 lados de un triángulo. El módulo devuelve un valor diciendo si el triángulo es escaleno, isósceles o equilátero.
- Aplicando el método:

Condiciones de Entrada	Clases de equivalencia validas	Clases de equivalencia invalidas
El número de valores es	(1) 3	(2) 2 (3) 4
Los valores son	(4) enteros	(5) cualquier otro dato
Valores mayores de 0	(6) todos >0	(7) alguno <=0

Casos de prueba

- Los casos de prueba resultantes son:

Valores de entrada: 3, 5, 4	(engloba a todas las clases válidas: 1, 4, 6)
Valores de entrada: 2, 5	(clase 2: 2 datos)
Valores de entrada: 2, 5, 8, 9	(clase 3: 4 datos)
Valores de entrada: 2, -5, 'C'	(clase 5: dato no entero)
Valores de entrada: 2, -2, 5	(clase 7: algún dato <=0)

- No siempre que se meten 3 enteros se puede formar un triángulo (se debe cumplir que a + b > c). Este error no le va a detectar la partición equivalente.

Análisis de Valores Límite

- Las condiciones límites son aquellas situaciones que se dan cuando se introducen valores que están justo en el límite de las clases de equivalencia de entrada y de salida. Está comprobado que los errores se acumulan en torno a los límites del dominio de un dato, en mayor medida que para valores intermedios.
- Esta técnica conduce a la elección de casos de prueba en los bordes de la clase correspondiente.
- Las diferencias con el método de la partición equivalente estriban en que no se selecciona un elemento representativo de una clase de equivalencia sino uno o más elementos tal que se prueben los extremos de esa clase de equivalencia, prestando atención no sólo a las condiciones de entrada (espacio de entradas) sino también al espacio de resultados.
- La **ventaja** de este método es que la probabilidad de detectar errores es mayor que con el método de la partición equivalente.

Estrategia

- La **estrategia** será:
  - Si una condición de entrada es un rango comprendido entre dos valores definidos (máximo y mínimo), definirás casos de prueba con ambos valores, y también para valores justo por encima y justo por debajo del máximo y mínimo respectivamente.
  - Si una condición de entrada es un número de valores, crearás casos de prueba para el valor máximo y mínimo (sí es que se conocen), y para los valores por encima y por debajo de ambos.
  - Debes usar ambas reglas para cada condición de salida. Es importante que examines los límites del resultado pues no siempre los límites de entrada coinciden con los límites de salida.

Datos de salida

- Esta técnica también la empleas para los **datos de salida**:
  - Si la entrada y/o salida es un archivo o conjunto ordenado (fichero secuencial, tabla...), tienes que centrar la atención en el primer y último elemento del conjunto.
  - Si la estructura de datos interna tienen límites preestablecidos (una tabla de x elementos), debes asegurarte de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.
- Esta técnica es uno de los mejores métodos para derivar casos de prueba, si se utiliza correctamente. A veces la determinación de valores límite es muy delicada y requiere una minuciosa elaboración.

Ejemplo

- Vamos a aplicar el método de análisis de valores límite al caso del triángulo ya resuelto mediante partición equivalente.
- Además de reflejar en la tabla las condiciones de entrada, debemos considerar las condiciones de salida.
- En segundo lugar, cuando tomemos los casos de prueba debemos recordar que elegiremos los valores extremos de las clases de equivalencia obtenidas.

Tabla las condiciones de entrada

Condiciones de Entrada	Clases de equivalencia válidas	Clases de equivalencia inválidas
El número de valores es	(1) 3	(2) 2 (3) 4
Los valores son	(4) enteros	(5) cualquier otro dato
Valores mayores de 0	(6) todos >0	(7) alguno <=0
Número de resultados	(8) 1	(9) <1 (10) >1
Tipo salida	(11) cadena	(12) cualquier otro
Relación entre longitudes	(13) a+b>c	(14) a+b<=c
Cuando a=b=c, devuelve	(15) Equilátero	(16) cualquier otro
Cuando dos lados son iguales, devuelve	(17) Isósceles	(18) cualquier otro
Cuando los tres son distintos, devuelve	(19) Escaleno	(20) cualquier otro

Casos de prueba resultantes

Caso	Valores de entrada	Valores de salida	Clases de equivalencia contempladas
1	1, 1, 1	Equilátero	1, 4, 6, 8, 11, 13, 15 (solo faltan 17 y 19)
2	1, 2, 2	Isósceles	1, 4, 6, 8, 11, 13, 17
3	2, 3, 4	Escaleno	1, 4, 6, 8, 11, 13, 19
4	1, 1	?	2
5	1, 1, 1, 1	?	3
6	2, 5, 'C'	?	5
7	2, 2, 5	No triángulo	7
8	1, 1, 1	(salida vacía)	9
9	1, 1, 1	Equil. + Isósc.	10
10	1, 1, 1	1	12
11	1, 2, 3	No triángulo	14
12	1, 1, 1	Isósceles	16
13	1, 2, 2	Escaleno	18
14	2, 3, 4	Equilátero	20

Observaciones

- Observa que no es posible incluir todas las clases de equivalencia válidas en un solo caso de prueba.
- No obstante intentaremos incluir todas las clases de equivalencia válidas en el mínimo número de casos de prueba (y cada clase de equivalencia inválida supondrá 1 o más casos de prueba separados).
- Así mismo, fíjate en que se han elegido los valores 1, 1 y 1 por ser los valores válidos más cercanos al límite.

Pruebas de Caja Blanca

- La prueba de caja blanca es una técnica de diseño de casos de prueba que usa la estructura de control del diseño para descubrir los casos de prueba.
- Mediante los métodos de prueba de caja blanca, puedes obtener casos de prueba que:
  - Garanticen que se ejercitan por lo menos una vez todos los caminos independientes de cada módulo.
  - Ejerciten todas las decisiones lógicas en sus vertientes verdadera y falsa.
  - Ejecuten todos los bucles en sus límites y con sus límites operacionales.
  - Ejerciten las estructuras internas de datos para asegurar su validez.
- Las pruebas de caja blanca **no garantizan** que:
  - El programa cumpla con su especificación.
  - El programa esté completo. Un programa puede estar incompleto por falta de estructuras obviadas por el programador al generar el código.
  - No detectan problemas de dominio de datos.

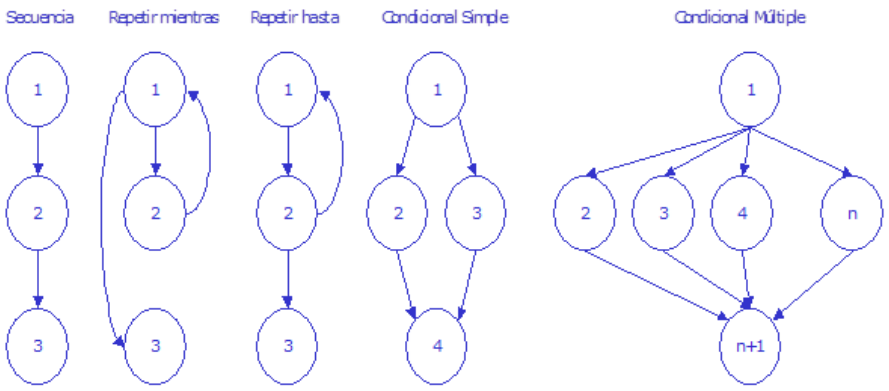
Pruebas de Caja Blanca

- Las pruebas de caja blanca se pueden aplicar a cualquier nivel donde se conozca la estructura:
  - Pruebas unitarias: la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
  - Pruebas de integración: la estructura de llamadas, secuencias o colaboración, o la transición de estados.
  - Pruebas de sistema: la estructura puede ser el árbol de navegación, los procesos de negocio o la estructura del sitio web.
- La pruebas de caja blanca permiten establecer la métrica de cobertura de código que indica el porcentaje de instrucciones que han participado en la prueba, o coloquialmente, que se han probado.

Técnica del Camino Básico

- El método del camino básico te permite obtener una medida de la complejidad lógica de un diseño y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución.
- Los casos de prueba derivados del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa. Los pasos que debes dar son:
  1. Numeras las sentencias del diseño o código, que aparecerán en los correspondientes nodos del grafo de flujo.
  2. Para numerar se escogen: alternativas, repetitivas, secuencias resultado de alternativas, conjuntos de sentencias útiles.
  3. Dibujas el grafo de flujo, partiendo del diseño o del código del procedimiento.
  4. Obtienes la complejidad ciclomática del grafo de flujo.
  5. Fijas un conjunto básico de caminos linealmente independientes.
  6. Determinar los casos de prueba que permitan la ejecución de cada camino del conjunto anterior.
- Para construir el **grafo de flujo**, hay que transformar las construcciones del diseño o de la programación en los subgrafos de flujo adecuados.

Sub grafos de flujo



Complejidad Ciclomática

- El círculo, denominado **nodo**, representa una o más sentencias. Las flechas, denominadas **aristas**, representan los flujos de control. Las áreas delimitadas por aristas se denominan **regiones**. El área más externa dentro del grafo también es una región.
- El número de caminos independientes del conjunto básico de un programa o una realización de caso de uso del diseño se puede calcular a partir de la **Complejidad Ciclomática** basada en la teoría de grafos, y nos da el número mínimo de casos de prueba que debemos diseñar para asegurar que se ejecuta cada sentencia al menos una vez:

Complejidad Ciclomática = Número de aristas - Número nodos + 2

Prueba de Bucles

- El método se centra en la validez de las construcciones de bucles. Debes intentar descubrir errores de inicialización, errores de indexación o de incremento y errores en los límites de los bucles.
- Este tipo especial de pruebas es complementario a la prueba de los caminos básicos.
- Podemos definir **tres tipos de bucles**:
  - Simples
  - Anidados
  - Concatenados

Prueba de Bucles

- A los **bucles simples** se les debe aplicar el siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucle:
  1. Saltar totalmente el bucle.
  2. Hacer n-1 pasos por el bucle.
  3. Hacer n pasos por el bucle.
  4. Hacer n+1 pasos por el bucle.
- Si extendiéramos el enfoque de prueba de los bucles simples a los **bucles anidados**, el número de posibles pruebas crecería geométricamente a medida que aumentara el nivel de anidamiento. Esto nos llevaría a un número impracticable de pruebas.

Prueba de Bucles

- Un enfoque para **reducir el número de pruebas** en este caso es:
  1. Comenzar en el bucle más interno. Disponer todos los demás bucles en sus valores mínimos.
  2. Llevar a cabo las pruebas de bucles simples con el bucle más interno mientras se mantienen los bucles externos con los valores mínimos para sus parámetros de iteración (por ejemplo contadores de bucle). Añadir otras pruebas para los valores fuera de rango o para valores excluidos.
  3. Progresar hacia fuera, llevando a cabo las pruebas para el siguiente bucle, pero manteniendo todos los demás bucles externos en sus valores mínimos y los demás bucles anidados con sus valores típicos.
  4. Continuar hasta que se hayan probado todos los bucles.
- Los **bucles concatenados** se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras que cada uno de los bucles sea independiente del resto. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.

Análisis de Decisiones

- Cuando un programa contiene alternativas dentro de la estructura de diseño, nos obliga a generar datos de prueba que aseguren que se ejecuta cada rama lógica del programa.
- Las alternativas pueden tener una o más condiciones, la estrategia para cada caso es diferente.
- Este análisis es complementario a la prueba de los caminos básicos.
- Cuando la alternativa tiene una sola condición el criterio de pruebas mínimo es generar datos de prueba que cubran:
  - Todos los valores de cada decisión al menos una vez.
  - Llamadas a cada punto de entrada, al menos una vez.
- Para programas con decisiones de tipo multicondición, el criterio a aplicar es el de establecer un número de casos de prueba suficiente para cubrir:
  - Todas las combinaciones posibles de todos los valores de cada condición.
  - Llamadas a cada punto de entrada, al menos una vez.

Tablas de decisión

- Las tablas de decisión representan relaciones lógicas entre las condiciones (entradas) y las acciones (salidas). Los casos de prueba son derivados sistemáticamente considerando cada combinación posible de condiciones y de acciones.
- Son una buena manera de capturar los requerimientos del sistema que contienen condiciones lógicas.
- Pueden ser utilizadas para registrar reglas de negocio complejas de un sistema.
  - Se analiza la especificación, y las condiciones y las acciones del sistema se identifican.
  - Las condiciones y las acciones de la entrada se indican de una manera tal que puedan ser verdaderas o falsas (booleano).
  - Cada columna de la tabla corresponde a una regla de negocio que define una combinación única de las condiciones que dan lugar a la ejecución de las acciones asociadas a esa regla.
  - La idea es tener por lo menos una prueba por columna, que implica cubrir todas las combinaciones para accionar condiciones.

Tablas de decisión

C1	Tarjeta Seguridad Social	SI	SI	SI	SI	NO	NO	NO	NO
C2	Tarjeta Sociedad Medica	SI	SI	NO	NO	SI	SI	NO	NO
C3	Tarjeta Fidelidad	SI	NO	SI	NO	SI	NO	SI	NO
A1	Sin Pago	X	X	X	X	X			
A2	Pago franquicia						X		X
A3	Pago servicios							X	X

- La fortaleza de las tablas de decisión es crear combinaciones de condiciones que pueden no ejercitarse de otra manera.
- Puede ser aplicada cuando la acción del software depende de varias decisiones lógicas.

Combinando las técnicas de caja blanca

- En el siguiente ejemplo vamos a estudiar un módulo, donde combinaremos las diferentes técnicas de caja blanca.
- El módulo lee como máximo 100 números de una tabla y calcula su media siempre y cuando los valores de los números estén entre 3 y 10, ignorándose el resto.
- Si no existe ningún número válido para la media, el módulo devuelve –1 como valor de la media.
- El final de la tabla se marca con el valor –1.

Pseudocódigo del módulo

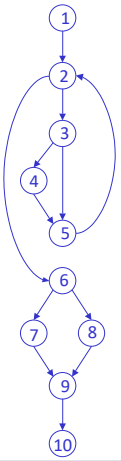
```
inicio calcular_media (entrada: tabla, salida: media)
    contador = 0
    suma = 0
    indice = 0
    mientras ((tabla[indice] <> -1) y (indice < 100))
        si ((tabla[indice] >= 3) y (tabla[indice] <= 10))
                suma = suma + tabla[indice]
                contador = contador + 1
        fin si
        indice = indice + 1
    fin mientras
    si (contador > 0)
        media = suma / contador
    si_no
        media = -1
    fin si
fin calcular_media
```

Generar el grafo

- Empezamos identificando y numerando los nodos.

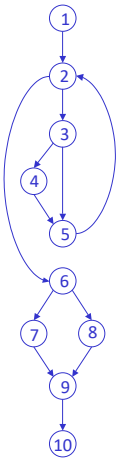
```
inicio calcular_media (entrada: tabla, salida: media)
    contador = 0                                              1
    suma = 0
    indice = 0
    mientras ((tabla[indice] <> -1) y (indice < 100))         2
        si ((tabla[indice] >= 3) y (tabla[indice] <= 10))     3
            suma = suma + tabla[indice]                       4
            contador = contador + 1
        fin si                                                5
        indice = indice + 1
    fin mientras
    si (contador > 0)                                         6
        media = suma / contador                               7
    si_no                                                     8
        media = -1
    fin si                                                    9
fin calcular_media                                            10
```

Dibujamos el grafo



Complejidad Ciclomática y caminos independientes

- Contamos las aristas y los nodos para calcular la Complejidad Ciclomática.
- **Aplicamos la fórmula**:
  Complejidad Ciclomática = 12 aristas - 10 nodos + 2 = 4
- Obtenemos que tenemos 4 caminos y **pasamos a fijar el conjunto básico de caminos linealmente independientes:**
  CB1: 1 → 2 → 6 → 7 → 9 → 10
  CB2: 1 → 2 → 6 → 8 → 9 → 10
  CB3: 1 → 2 → 3 → 5 → 2 → 6 → (...resto del CB1 ó CB2...) → 8 → 9 → 10
  CB4: 1 → 2 → 3 → 4 → 5 → 2 → 6 → (...resto del CB1 ó CB2...) → 7 → 9 → 10



Pruebas de bucles

- **Identificamos las pruebas de bucles**:

  B1: saltar el bucle; la tabla vacía, la primera posición con valor –1

  B2: ejecutarlo 99 veces; la tabla con 99 números, la posición 99 con valor -1

  B3: ejecutarlo 100 veces; la tabla con 100 números, la posición 100 con valor –1

  B4: ejecutarlo 101 veces; la tabla con 101 números, la posición 101 con valor –1

Prueba de las condiciones

Nodo 2		R1	R2	R3	R4
C1	tabla[indice] <> -1	V	V	F	F
C2	indice < 100	V	F	V	F
A1	Permanencia en bucle	X			

Nodo 3		R5	R6	R7
C3	tabla[indice] >= 3	V	V	F
C4	tabla[indice] <= 10	V	F	-
A2	Acumular y contar	X		

Nodo 6		R8	R9
C5	contador > 0	V	F
A3	Calcular media	X	
A4	Devolver -1		X

Condiciones

- R1: tabla[indice] = -1
- R2: tabla[indice] <> -1
- R3: tabla[indice] >= 3
- R4: tabla[indice] <= 10
- R5: tabla[indice] < 3
- R6: tabla[indice] > 10
- R7: indice < 100
- R8: indice >= 100
- R9: contador = 0
- R10: contador > 0

Casos de prueba

- Una vez tengamos fijados los caminos, los bucles y las reglas, determinamos los casos de prueba que permitan la ejecución de cada camino del conjunto anterior.
- Caso de prueba 1:
  - Entrada:
    - La tabla vacía:

0	1	2	...
-1			

  - Salida:
    - Media = -1
  - Objetivo:
    - **CB2**, **B1**, **R1**, **R9**

Casos de prueba

- **Caso de prueba 2:**
  - Entrada:
    - La tabla con 99 números y todos están entre 3 y 10

0	1	2	3	4	5	...	98	99	100	101
2	11	13	-2	0	1	1	1	1	-1	

  - Salida:
    - Media = 5
  - Objetivo:
    - **CB1**, **CB4**, **B2**, R1, **R2**, R4, **R6**, **R7, R10**

Casos de prueba

- **Caso de prueba 3:**
  - Entrada:
    - La tabla con 100 números y ninguno está entre 3 y 10

0	1	2	3	4	5	...	98	99	100	101
2	11	13	-2	0	1	1	1	1	-1	

  - Salida:
    - Media = -1
  - Objetivo:
    - **CB3**, **B3**, R1, **R4**, **R5**, **R7**, R10

Casos de prueba

- **Caso de prueba 4:**
  - Entrada:
    - La tabla con 101 números surtidos

0	1	2	3	4	5	...	98	99	100	101
3	7	5	5	1	11	11	10	21	1	-1

  - Salida:
    - Media = 6
  - Objetivo:
    - **B4**, R1, **R2**, R5, R6, R8, R10

Cobertura

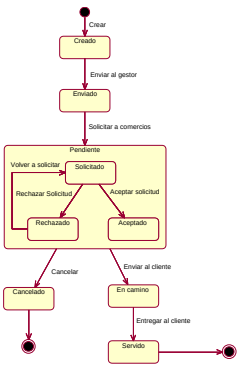
- Las pruebas de sentencia ejercitan las sentencias ejecutables en el código. La cobertura se mide como el número de sentencias ejecutadas por las pruebas dividido por el número total de sentencias ejecutables en el objeto de prueba, normalmente expresado como un porcentaje.
- La pruebas de decisión ejercitan las condiciones en el código y ejercitan el código que se ejecuta basado en los resultados de la decisión. Para ello, los casos de prueba siguen los flujos de control que se producen desde un punto de decisión (por ejemplo, para una declaración IF, uno para el resultado verdadero y otro para el resultado falso; para una declaración CASE, se necesitarían casos de prueba para todos los resultados posibles, incluido el resultado por defecto). La cobertura se mide como el número de resultados de decisión ejecutados por las pruebas dividido por el número total de resultados de decisión en el objeto de prueba, normalmente expresado como un porcentaje.

Cobertura

- Cuando se logra una cobertura del 100% de sentencia, se asegura de que todas las sentencias ejecutables del código se han probado al menos una vez, pero no asegura que se haya probado toda la lógica de decisión. Cuando se alcanza el 100% de cobertura de decisión, se ejecutan todos los resultados de decisión, lo que incluye probar el resultado verdadero y también el resultado falso, incluso cuando no hay una sentencia falsa explícita (por ejemplo, en el caso de una sentencia IF sin un ELSE en el código).
- La cobertura de sentencia ayuda a encontrar defectos en el código que no fueron practicados por otras pruebas. La cobertura de decisión ayuda a encontrar defectos en el código donde otras pruebas no han tenido ambos resultados, verdadero y falso.
- Lograr una cobertura del 100% de decisión garantiza una cobertura del 100% de sentencia (pero no al revés).

Técnica de Transición de Estado

- Algunos elementos del sistema pueden tener estado y dichos estados pueden estar tipados: conjunto de valores predefinidos. El cambio de un valor del estado a otro valor de estado se denomina transición de estado y es desencadenado por una acción o evento. No todas las transiciones son posibles: combinaciones de valor inicial con valor final.
- Los casos de pruebas pueden ser diseñados para cubrir una secuencia de estados típica, para practicar todos los estados, para practicar cada transición, para practicar secuencias específicas de transiciones, o para probar transiciones inválidas.
- Los diagramas de estado representan todos los posibles estados de un elemento del sistema y sus correspondientes transiciones de estado. Una tabla de transición de estado muestra todas las transiciones válidas y las transiciones potencialmente inválidas entre estados, así como los eventos, las condiciones de guarda y las acciones resultantes para las transiciones válidas.



Pruebas de interfaces de usuario

- A diferencia de los casos anteriores, las pruebas de interfaces de usuario se basan en la experiencia para descubrir los casos de prueba.
- La técnica se basa en una serie de cuestionarios que contienen las situaciones más habituales de error. La identificación de los casos de prueba la realizas recorriendo cada una de las cuestiones de cada lista, estudiando si es o no es aplicable. En caso de ser aplicable necesitas diseñar un caso de prueba.
- Las listas que te proponemos son más o menos estándar. No son cerradas. Debes irlas completando sobre la base de tu experiencia.

Ventanas

- ❑ ¿Se abrirán las ventanas basándose en órdenes basadas en el teclado o en un menú?
- ❑ ¿Se puede ajustar el tamaño, mover y desplegar la ventana?
- ❑ ¿Está todo el contenido de la información dentro de la ventana accesible adecuadamente con el ratón, teclas de función, flechas de dirección y teclado?
- ❑ ¿Se genera adecuadamente cuando se sobrescribe y se vuelve a abrir?
- ❑ ¿Están operativas todas las funciones relacionadas con la ventana?
- ❑ ¿Están disponibles y desplegados apropiadamente en la ventana todos los menús emergentes, barras de herramientas, barras deslizantes, cuadros de diálogo, botones, iconos y otros controles importantes?
- ❑ Cuándo se despliegan varias ventanas, ¿se representa adecuadamente el nombre de cada ventana?
- ❑ ¿Está resaltada adecuadamente la ventana activa?
- ❑ Si se utiliza multitarea, ¿están actualizadas todas las ventanas en los momentos adecuados?
- ❑ ¿Causan las selecciones múltiples o incorrectas del ratón dentro de la ventana efectos secundarios inesperados?
- ❑ ¿Están de acuerdo con las especificaciones los indicadores de audio y/o de color de la ventana o como consecuencia de operaciones de la ventana?
- ❑ ¿Se cierra adecuadamente la ventana?

Menús emergentes y operaciones con el ratón

- ¿Se muestra la barra de menú apropiada en el contexto apropiado?
- ¿Despliega la barra de menú de la aplicación características relacionadas con el sistema (por ejemplo: la pantalla de un reloj)?
- ¿Funcionan adecuadamente las operaciones de despliegue?
- ¿Funcionan adecuadamente los menús de escape, paletas y barras de herramientas?
- ¿Están listadas adecuadamente todas las funciones del menú y subfunciones emergentes?
- ¿Son todas las funciones del menú accesibles con el ratón?
- ¿Es correcto el tipo, tamaño y formato de texto?
- ¿Es posible invocar todas las funciones del menú usando su orden alternativa de texto?

Menús emergentes y operaciones con el ratón (cont.)

- ¿Están resaltadas las funciones del menú (o difuminadas) dependiendo del contexto de las operaciones actuales de la ventana?
- ¿Se ejecutan todas las funciones de cada menú como se anunciaba?
- ¿Son suficientemente claros los nombres de las funciones del menú?
- ¿Hay ayuda disponible para cada elemento del menú y es sensible al contexto?
- ¿Se reconocen apropiadamente las operaciones del ratón a lo largo de todo el contexto interactivo?
- Si se necesitan múltiples clic, ¿están apropiadamente reconocidos en el contexto?
- Si el ratón tiene varios botones, ¿son reconocidos apropiadamente en el contexto?
- ¿Cambian adecuadamente el cursor, el indicador de procesamiento y el puntero al invocar diferentes operaciones?

Entrada de datos

- ¿Se repiten y son introducidos adecuadamente los datos alfanuméricos en el sistema?
- ¿Funcionan adecuadamente los modos gráficos de entrada de datos (por ejemplo: una barra deslizante)?
- ¿Se reconocen adecuadamente los datos no válidos?
- ¿Son inteligibles los mensajes de entrada de datos?

Documentación y ayuda

- ¿Describe con exactitud la documentación cómo conseguir cada modo de empleo?
- ¿Es exacta la descripción de cada secuencia de interacción?
- ¿Son exactos los ejemplos?
- ¿Son consistentes con el programa real la terminología, las descripciones del menú y las respuestas del sistema?
- ¿Es relativamente fácil localizar ayuda en la documentación?
- ¿Se pueden solucionar problemas fácilmente con la documentación?
- ¿Son exactos y completos la tabla de contenido y el índice?
- ¿Facilita el diseño del documento (distribución, tipos de letra, indentación, grafos) la comprensión y rápida asimilación de la información?
- ¿Están descritos con gran detalle los mensajes de error para el usuario en el documento?
- Si se utilizan enlaces de hipertexto, ¿son exactos y completos?

Páginas Web

- La navegación
- Las interacciones con la página y sus elementos
- El rellenado de formularios y sus validaciones
- El arrastrar y soltar, si procede
- La estética, presencia y visualización de elementos esenciales, con especial atención al Responsive Design.
- Moverse entre ventanas y marcos (aunque están prohibidos por la WAI)
- Uso de cookies, Local Storage, Session Storage, Service Worker, …

Técnicas estáticas

- Al contrario que las pruebas dinámicas, que exigen la ejecución de software, las técnicas de pruebas estáticas se basan en el examen manual (revisiones) y en el análisis automatizado (análisis estático) del código o de cualquier otra documentación del proyecto sin ejecutar el código.
- Las revisiones constituyen una forma de probar los productos de trabajo del software (incluyendo el código) y pueden realizarse antes de ejecutar las pruebas dinámicas. Los defectos detectados durante las revisiones al principio del ciclo (por ejemplo, los defectos encontrados en los requisitos) a menudo son mucho más baratos de eliminar que los detectados durante las pruebas realizadas ejecutando el código.
- Una revisión podría hacerse íntegramente como una actividad manual, pero también existen herramientas de soporte.

Revisión

- La principal actividad manual consiste en examinar un producto de trabajo y hacer comentarios al respecto. Cualquier producto de trabajo de software puede ser objeto de una revisión como por ejemplo:
  - Las especificaciones de requisitos,
  - Las especificaciones de diseño,
  - El código,
  - Los planes de pruebas,
  - Las especificaciones de pruebas,
  - Los casos de pruebas,
  - Los guiones de pruebas,
  - Las guías de usuario o las páginas web.
- Los beneficios de las revisiones incluyen la detección y corrección temprana de defectos, el desarrollo de mejoras de productividad, la reducción de los tiempos de desarrollo, el ahorro de tiempo y dinero invertido en pruebas, el menor coste de vida, menos defectos y comunicación mejorada. Las revisiones pueden encontrar omisiones , por ejemplo, en requisitos, que no suelen encontrarse en pruebas dinámicas.

Revisión

- Las revisiones, el análisis estático y las pruebas dinámicas tienen el mismo objetivo: Identificar defectos.
  - Se trata de procesos complementarios.
  - Las distintas técnicas pueden encontrar distintos tipos de defectos de una manera eficiente y efectiva.
  - En comparación con las pruebas dinámicas, las técnicas estáticas localizan las causas o los defectos más que los propios fallos.
- Entre los defectos típicos que resultan más fáciles de localizar en revisiones que en las pruebas dinámicas se incluyen:
  - Desviaciones de los estándares,
  - Defectos de requisitos,
  - Defectos de diseño,
  - Mantenibilidad insuficiente y
  - Especificaciones de interfaz incorrectas.

Tipos de revisiones

- **Revisión Informal**: Es una forma barata de obtener beneficios, sin un proceso formal, cuyos resultados pueden estar documentados. Su utilidad varía en función de los revisores.
- **Revisión guiada**: Reunión guiada por el autor del artefacto, cuyo objetivo principal es aprender, entender y encontrar defectos. En la práctica puede variar desde bastante informal hasta muy formal, siendo opcional la elaboración de un informe de revisión en el que se incluya la lista de conclusiones.
- **Revisión técnica**: Es un proceso formal, documentado y definido para la detección de defectos cuyos objetivos principales son: debatir, tomar decisiones, evaluar alternativas, encontrar defectos, resolver problemas técnicos y comprobar la conformidad con las especificaciones, los planes, la normativa y los estándares. El informe de revisión debe incluir la lista de conclusiones, el veredicto de si el artefacto cumple los requisitos y, si procede, recomendaciones en base a las conclusiones. Requiere preparación previa a la reunión.
- **Inspección**: Dirigida por un especialista, es un proceso formal basado en normas y listas de comprobación cuyo objetivo principal es identificar defectos. Incluye una recopilación de métricas, criterios de entrada y salida especificados para la aceptación del producto de software y un proceso de seguimiento formal. El informe de revisión debe incluir la lista de conclusiones. Requiere preparación previa a la reunión.

Actividades de una revisión

- Una revisión formal típica incluye las siguientes actividades principales:
  1.- **Planificar**:
    - Definir los criterios de revisión.
    - Seleccionar al personal.
    - Asignar funciones.
  2.- **Definir los criterios de entrada y salida** (para tipos de revisión más formales).
    - Seleccionar qué partes de los documentos deben revisarse.
  3.- **Inicio**:
    - Repartir los documentos.
    - Explicar los objetivos, procesos y documentos a los participantes.
  4.- **Comprobar los criterios de entrada** (para tipos de revisión más formales).
  5.- **Preparación individual**.
    - Prepararse para la reunión de revisión repasando los documentos.

Actividades de una revisión

6.- **Prestar especial atención a posibles defectos, preguntas y comentarios**.

7.- **Examen/evaluación/registro de los resultados** (reunión de revisión):

- Debatir o registrar, mediante resultados documentados o actas (para tipos de revisión más formales).
- Prestar especial atención a los defectos, hacer recomendaciones sobre cómo manejar los defectos, tomar decisiones al respecto.

8.- **Examinar/evaluar y registrar** durante reuniones físicas o de seguimiento de los grupos comunicaciones electrónicas.

9.- **Adaptar**:

- Corregir los defectos detectados (normalmente a cargo del autor).
- Registrar el estado actualizado de los defectos (en revisiones formales).

10.- **Hacer un seguimiento**:

- Comprobar que los defectos han sido tratados.
- Recopilar métricas.

11.- **Comprobar los criterios de salida** (para tipos de revisión más formales).

Análisis estático con herramientas

- El objetivo del análisis estático es detectar defectos en el código fuente y en los modelos de software.
- El análisis estático se realiza sin que la herramienta llegue a ejecutar el software objeto de la revisión, como ocurre en las pruebas dinámicas, centrándose mas en como está escrito el código que en como se ejecuta el código.
- El análisis estático permite identificar defectos difíciles de encontrar mediante pruebas dinámicas.
- Al igual que sucede con las revisiones, el análisis estático encuentra defectos en lugar de fallos.
- Las herramientas de análisis estático analizan el código del programa (por ejemplo, el flujo de control y flujo de datos) y las salidas generadas (tales como HTML o XML).
- Algunos de los posibles aspectos que pueden ser comprobados con análisis estático:
  - Reglas, estándares de programación y buenas practicas.
  - Diseño de un programa (análisis de flujo de control).
  - Uso de datos (análisis del flujo de datos).
  - Complejidad de la estructura de un programa (métricas, por ejemplo valor ciclomático).

Valor del análisis estático

- La detección temprana de defectos antes de la ejecución de las pruebas.
- Advertencia temprana sobre aspectos sospechosos del código o del diseño mediante el cálculo de métricas, tales como una medición de alta complejidad.
- Identificación de defectos que no se encuentran fácilmente mediante pruebas dinámicas.
- Detectar dependencias e inconsistencias en modelos de software, como enlaces.
- Mantenibilidad mejorada del código y del diseño.
- Prevención de defectos, si se aprende la lección en la fase de desarrollo.

Defectos habitualmente detectados

- Referenciar una variable con un valor indefinido.
- Interfaces inconsistentes entre módulos y componentes.
- Variables que no se utilizan o que se declaran de forma incorrecta.
- Código inaccesible (muerto).
- Ausencia de lógica o lógica errónea (posibles bucles infinitos).
- Construcciones demasiado complicadas.
- Infracciones de los estándares de programación.
- Vulnerabilidad de seguridad.
- Infracciones de sintaxis del código y modelos de software.

Ejecución

- Las herramientas de análisis estático generalmente las utilizan los desarrolladores (cotejar con las reglas predefinidas o estándares de programación) antes y durante las pruebas unitarias y de integración, o durante la comprobación del código.
- Las herramientas de análisis estático pueden producir un gran número de mensajes de advertencias que deben ser bien gestionados para permitir el uso más efectivo de la herramienta.
- Los compiladores pueden constituir un soporte para los análisis estáticos, incluyendo el cálculo de métricas.
- El Compilador detecta errores sintácticos en el código fuente de un programa, crea datos de referencia del programa (por ejemplo lista de referencia cruzada, llamada jerárquica, tabla de símbolos), comprueba la consistencia entre los tipos de variables y detecta variables no declaradas y código inaccesible (código muerto).
- El Analizador trata aspectos adicionales tales como: Convenciones y estándares, Métricas de complejidad y Acoplamiento de objetos.

Técnicas basadas en la experiencia

- Hasta ahora hemos visto técnicas formales, basadas en pruebas sistemáticas, pero existe un conjunto de pruebas basadas en la experiencia que derivan de la habilidad e intuición del probador y de su experiencia con aplicaciones y tecnologías similares.
- Los casos de prueba están basados en la intuición y experiencia:
  - ¿Dónde se han acumulado errores en el pasado?
  - ¿Dónde falla normalmente el software?
- Se trata de un enfoque especialmente útil en los casos en los que las especificaciones son escasas o inadecuadas y existe una importante presión temporal.
- Pueden complementar a las técnicas sistemáticas e identificar pruebas especiales que no pueden capturarse fácilmente mediante técnicas formales, aunque el grado de efectividad esta directamente relacionado con la experiencia del probador.
- Asimismo, puede servir como comprobación del proceso de pruebas, para ayudar a garantizar que los defectos más graves han sido efectivamente detectados.

Diseño de casos de prueba.

- Las pruebas basadas en la experiencia también se denomina pruebas intuitivas (intuitive testing) e incluyen predicciones de errores (error guessing - pruebas orientadas a puntos débiles) y pruebas exploratorias (pruebas iterativas basadas en el conocimiento adquiridos respecto del sistema).
- **Intuición:** ¿Dónde se pueden esconder los defectos? La intuición caracteriza a un buen probador.
- **Experiencia:** ¿Qué **defectos** han sido **detectados** en el pasado y dónde?
- **Conocimiento / percepción** - ¿Dónde se **esperan defectos** especificos?
  - Se incluyen **detalles** específicos **del proyecto**.
  - ¿Dónde se producirán defectos como consecuencia de la **presión por los plazos** y la **complejidad**?
  - ¿Están involucrados programadores sin experiencia?

Diseño intuitivo de casos de prueba

- Las posibles fuentes del diseño intuitivo de casos de prueba son:
  - Resultados de pruebas y experiencia práctica con sistemas similares.
    - Posiblemente con el sistema actual a sustituir por el nuevo sistema u otro sistema con funcionalidad similar.
  - Experiencia de usuario.
    - Aplicar la experiencia con el sistema como usuario del mismo.
  - Enfoque del despliegue.
    - ¿Qué partes del sistema serán utilizados con mayor frecuencia?
  - Problemas de desarrollo.
    - ¿Hay algún punto débil como resultado de dificultades en el proceso de desarrollo?

Predicción de errores

- La fuente de la predicción de errores es la lista de defectos comunes que has ido elaborándote en la prueba de otros sistemas anteriores.
- Los pasos a dar para la creación de los casos de prueba serian:
  - Comprobar lista de defectos.
    - Enumerar posibles errores.
    - Ponderar factores dependientes del riesgo y probabilidad de ocurrencia.
  - Diseño de caso de prueba.
    - Creación de casos de prueba dirigidos a producir los defectos de la lista.
    - Aumentar la prioridad a aquellos casos de prueba considerando el valor de su riesgo.
  - Actualizar la lista de defectos durante las pruebas.
    - Es un procedimiento iterativo incremental.
    - Es útil para cuando se repita el procedimiento en futuros proyectos.

Pruebas exploratorias

- Es un procedimiento de diseño de casos de prueba especialmente apropiado cuando la información base se encuentra poco estructurada o cuando el tiempo disponible para pruebas es escaso.
- Procedimiento:
  - Revisar las partes constituyentes (individuales/identificables) del objeto de prueba.
  - Ejecutar un número reducido de casos de prueba exclusivamente sobre aquellas partes que deben ser probadas, aplicando predicción de errores ("errores guessing").
  - Analizar los resultados, desarrollar un modelo preliminar ("rough model") de cómo funciona el objeto de prueba.
  - Por iteración:
    - Diseñar nuevos casos de prueba aplicando el conocimiento adquirido recientemente.
    - Por lo tanto concentrándose en las áreas relevantes y explorando características adicionales del objeto de prueba.
- Es recomendable utilizar herramientas de captura, que son útiles para registrar las actividades de pruebas, y seleccionar objetos pequeños y/o concentrándose en aspectos particulares del objeto de pruebas, una iteración unitaria no debería llevar más de un par de horas.
- Los resultados de una iteración constituyen la base de información para la siguiente iteración, un conocimiento que permite la elección apropiada del métodos de diseño de casos de prueba.

Selección de la técnica de pruebas

- La selección de la técnica de pruebas a utilizar depende de una serie de factores entre los que se incluyen el tipo de sistema, los estándares normativos, los requisitos contractuales o de cliente, el nivel de riesgo, el tipo de riesgo, el objetivo de la prueba, la documentación disponible, el conocimiento de los probadores, el tiempo y el presupuesto, el ciclo de vida de desarrollo, los modelos de caso de uso y la experiencia previa en los tipos de defectos detectados.
- Algunas técnicas resultan más aplicables a ciertas situaciones y niveles de prueba, mientras que otras son aplicables a todos los niveles de pruebas.
- Para crear los casos de prueba, tendrás generalmente que aplicar una combinación de técnicas de pruebas, entre las que se incluyen técnicas guiadas por procesos, reglas y datos, con vistas a garantizar la correcta cobertura del objeto que se está probando.

http://junit.org/junit5/

PRUEBAS: JUNIT

Introducción

- JUnit es un entorno de pruebas opensource que nos permiten probar nuestras aplicaciones Java y permite la creación de los componentes de prueba automatiza que implementan uno o varios casos de prueba.
- JUnit 5 requiere Java 8 (o superior) en tiempo de ejecución. Sin embargo, aún puede probar el código que se ha compilado con versiones anteriores del JDK.
- A diferencia de las versiones anteriores de JUnit, JUnit 5 está compuesto por varios módulos diferentes de tres subproyectos diferentes:.
  - La plataforma JUnit: sirve como base para lanzar marcos de prueba en la JVM.
  - JUnit Jupiter: es la combinación del nuevo modelo de programación y el modelo de extensión para escribir pruebas y extensiones en JUnit 5.
  - JUnit Vintage: proporciona una función TestEngine para ejecutar pruebas basadas en JUnit 3 y JUnit 4 en la plataforma.

Casos de pruebas

- Los casos de prueba son clases que disponen de métodos para probar el comportamiento de una clase concreta. Así, para cada clase que quisiéramos probar definiríamos su correspondiente clase de caso de prueba.
- Los casos de prueba se definen utilizando:
  - Anotaciones: Automatizan el proceso de definición, sondeo y ejecución de las pruebas.
  - Aserciones: Afirmaciones sobre lo que se esperaba y deben cumplirse para dar la prueba superada. Todas las aserciones del método deben cumplirse para superar la prueba. La primera aserción que no se cumpla detiene la ejecución del método y marca la prueba como fallida.
  - Asunciones: Afirmaciones que deben cumplirse para continuar con el método de prueba, en caso de no cumplirse se salta la ejecución del método y lo marca como tal.

Clases y métodos de prueba

- Clase de prueba: cualquier clase de nivel superior, clase miembro static o clase @Nested que contenga al menos un método de prueba, no deben ser abstractas y deben tener un solo constructor. El anidamiento de clases de pruebas @Nested permiten organizar las pruebas a diferentes niveles y conjuntos.
- Método de prueba: cualquier método de instancia anotado con @Test, @RepeatedTest, @ParameterizedTest, @TestFactory o @TestTemplate.
- Método del ciclo de vida: cualquier método anotado con @BeforeAll, @AfterAll, @BeforeEach o @AfterEach.
- Los métodos de prueba y los métodos del ciclo de vida pueden declararse localmente dentro de la clase de prueba actual, heredarse de superclases o de interfaces, no deben ser privados ni abstractos o devolver un valor.

Documentar los resultados

- Por defecto, al ejecutar las pruebas, se muestran los nombres de las clases y los métodos de pruebas.
- Se puede personalizar los nombre mostrados mediante la anotación @DisplayName:

```
@DisplayName("A special test case")
class DisplayNameDemo {
    @Test
    @DisplayName("Custom test name")
    void testWithDisplayName() {    }
```

- Se puede anotar la clase con @DisplayNameGeneration para utilizar un generador de nombres personalizados transformar los nombres de los métodos a mostrar.

```
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
class Display_Name_Demo {
      @Test
      void if_it_is_zero() { }
```

Ciclo de vida de instancia de prueba

- Para permitir que los métodos de prueba individuales se ejecuten de forma aislada y evitar efectos secundarios inesperados debido al estado de instancia de prueba mutable, JUnit crea una nueva instancia de cada clase de prueba antes de ejecutar cada método de prueba (consulte Clases y métodos de prueba). Este ciclo de vida de instancia de prueba "por método" es el comportamiento predeterminado en JUnit Jupiter y es análogo a todas las versiones anteriores de JUnit.
- Los métodos anotados con @BeforeEach y @AfterEach se ejecutan antes y después de cada método de prueba.
- Anotando la clase de prueba con @TestInstance(Lifecycle.PER_CLASS) se pueden ejecutar todos los métodos de prueba en la misma instancia de prueba. Si los métodos de prueba dependen del estado almacenado en atributos de instancia, es posible que se deba restablecer ese estado en métodos @BeforeEach o @AfterEach.
- Los métodos de clase anotados con @BeforeAll y los @AfterAll se ejecutan al crear la instancia y al destruir la instancia, solo tienen sentido en el ciclo de vida de instancia "por clase".

Fixtures

- Es probable que en varias de las pruebas implementadas se utilicen los mismos datos de entrada o de salida esperada, o que se requieran los mismos recursos.
- Para evitar tener código repetido en los diferentes métodos de *test*, podemos utilizar los llamados *fixtures*, que son elementos fijos que se crearán antes de ejecutar cada prueba.

Ciclo de vida de instancia de prueba

- **@BeforeAll** public **static** void method()

  Este método es ejecutado una vez antes de ejecutar todos los test. Se usa para ejecutar actividades intensivas como conectar a una base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.
- **@BeforeEach** public void method()

  Este método es ejecutado antes de cada test. Se usa para preparar el entorno de test (p.ej., leer datos de entrada, inicializar la clase).
- **@AfterEach** public void method()

  Este método es ejecutado después de cada test. Se usa para limpiar el entorno de test (p.ej., borrar datos temporales, restaurar valores por defecto). Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas.
- **@AfterAll** public **static** void method()

  Este método es ejecutado una vez después que todos los tests hayan terminado. Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.

Ejecución de pruebas

- Las pruebas se pueden ejecutar desde:
  - IDE: IntelliJ IDEA , Eclipse , NetBeans y Visual Studio Code
  - Herramientas de compilación: Gradle, Maven y Ant (y en los servidores de automatización que soporte alguna de ellas)
  - Lanzador de consola: aplicación Java de línea de comandos que permite iniciar JUnit Platform desde la consola.
- Al ejecutarse las pruebas se marcan como:
  - Successful: Se ha superado la prueba.
  - Failed: Fallo, no se ha superado la prueba.
  - Aborted (Skipped): Se ha cancelado la prueba con una asunción.
  - Disabled (Skipped): No se ha ejecutado la prueba.
  - Error: Excepción en la ejecución del método de prueba, no se ha ejecutado la prueba.

Aserciones

- assertTrue(boolean condition, [message]): Verifica que la condición booleana sea true.
- assertFalse(boolean condition, [message]): Verifica que la condición booleana sea false.
- assertNull(object, [message]): Verifica que el objeto sea nulo.
- assertNotNull(object, [message]): Verifica que el objeto no sea nulo.
- assertEquals(expected, actual, [message]): Verifica que los dos valores son iguales.
  - assertEquals(expected, actual, tolerance, [message]): Verifica que los valores float o double coincidan. La tolerancia es el número de decimales que deben ser iguales.
- assertArrayEquals (expected, actual, [message]): Verifica que el contenido de dos arrays son iguales.
- assertIterableEquals(expected, actual, [message]): Verifica (profunda) que el contenido de dos iterables son iguales.
- assertLinesMatch(expectedLines, actualLines, [message]): Verifica (flexible) que el contenido de dos listas de cadenas son iguales.

Aserciones

- assertNotEquals(expected, actual, [message]): Verifica que los dos valores NO son iguales.
- assertSame(expected, actual, [message]): Verifica que las dos variables referencien al mismo objeto.
- assertNotSame(expected, actual, [message]): Verifica que las dos variables no referencien al mismo objeto.
- assertThrows(exceptionType, executable, [message]): Verifica que se genera una determinada excepción.
- assertDoesNotThrow(executable, [message]): Verifica que no lanza ninguna excepción.
- assertTimeout(timeout, executable , [message]): Verifica que la ejecución no exceda el timeout dado.
- assertAll(title, asserts): Verifica que se cumplan todas las aserciones de una colección.
- fail([message]): Hace que el método falle. Debe ser usado para probar que cierta parte del código no es alcanzable para que el test devuelva fallo hasta que se implemente el método de prueba.

Aserciones

- Aserciones de comparación:

```
assertEquals(2, calculator.add(1, 1));
assertTrue(Double.isInfinite(calculator.divide(1, 0)) , "División por cero");
assertNotNull(lst.get(1));
```

- Verificaciones de excepciones:

```
Exception forValidateMessageException =
      assertThrows(ArithmeticException.class, () -> divide(1.0, 0.0));
assertDoesNotThrow(() -> lst.remove(1));
```

- Verificación SLA:

```
String actualResult = assertTimeout(Duration.ofMillis(90), () -> {
   Thread.sleep(50);
   return "OK";
 });
assertEquals("OK", actualResult);
```

- Fallar directamente:

```
fail("Not yet implemented");
```


Hamcrest

- Si queremos aserciones todavía más avanzadas, se recomienda usar librerías específicas, como por ejemplo Hamcrest: http://hamcrest.org
- Hamcrest es un marco para escribir objetos de coincidencia que permite que las restricciones se definan de forma declarativa. Hay una serie de situaciones en las que los comparadores son invariables, como la validación de la interfaz de usuario o el filtrado de datos, pero es en el área de redacción de pruebas flexibles donde los comparadores son los más utilizados.

```
import static org.hamcrest.MatcherAssert.assertThat;

assertThat(calculator.subtract(4, 1), not(is(equalTo(2))));
assertThat(Arrays.asList("foo", "bar", "baz"), hasItem(startsWith("b") ,
      endsWith("z")));
assertThat(person, has(
      property("firstName", equalTo("Pepito")),
      property("lastName", equalTo("Grillo")),
      property("age", greaterThan(18))));
```

AssertJ

- Otra alternativa para aserciones todavía más avanzadas es AssertJ: https://assertj.github.io/doc/
- AssertJ es una biblioteca de Java que proporciona una api fluido con un amplio conjunto de aserciones y mensajes de error realmente útiles, que mejora la legibilidad del código de prueba y está diseñado para ser súper fácil de usar dentro de su IDE favorito. AssertJ es un proyecto de código abierto que ha surgido a partir del desaparecido Fest Assert. También, dispone de un generador automático de comprobaciones para los atributos de las clases, lo que añade más semántica a nuestras clases de prueba.

```
import static org.assertj.core.api.Assertions.*;

assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```


Agrupar aserciones

- La primera aserción que no se cumpla detiene la ejecución del método y marca la prueba como fallida.
- Si un método de prueba cuenta con varias aserciones, el fallo de una de ellas impedirá la evaluación de las posteriores por lo que no se sabrá si fallan o no, lo cual puede ser un inconveniente.
- Para solucionarlo se dispone de assertAll: ejecuta todas las aserciones contenidas e informa del resultado, en caso de que alguna falle assertAll falla.

```
@Test
  void groupedAssertions() {
    assertAll("person",
      () -> assertEquals("Jane", person.getFirstName()),
      () -> assertEquals("Doe", person.getLastName())
    );
  }
```

Agrupar pruebas

- Las pruebas anidadas, clases de prueba anidadas dentro de otra clase de prueba y anotadas con @Nested, permiten expresar la relación entre varios grupos de pruebas.

```
class EntityTest {
        @Test
        void testCanCreate() { }
        @Nested
        class DataRulesTest {
                @Test
                void testProperty1() { }
        }
        @Nested
        class BusinessRulesTest {
                @Nested
                class PersistenceTest {

                }
        }
}
```


Pruebas repetidas

- Se puede repetir una prueba un número específico de veces con la anotación @RepeatedTest. Cada iteración se ejecuta en un ciclo de vida completo.
- Se puede configurar el nombre de salida con:
  - {displayName}: nombre del método que se ejecuta
  - {currentRepetition}: recuento actual de repeticiones
  - {totalRepetitions}: número total de repeticiones
- Se puede inyectar RepetitionInfo para tener acceso a la información de la iteración.

```
@RepeatedTest(value = 5, name = "{displayName}
        {currentRepetition}/{totalRepetitions}")
void repeatedTest(RepetitionInfo repetitionInfo) {
    assertEquals(5, repetitionInfo.getTotalRepetitions());
}
```

Pruebas parametrizadas

- Las pruebas parametrizadas permiten ejecutar una prueba varias veces con diferentes argumentos. Se declaran como los métodos @Test normales, pero usan la anotación @ParameterizedTest y aceptan parámetros. Además, se debe declarar al menos una fuente que proporcionará los argumentos para cada invocación y utilizar los parámetros en el método de prueba.
- Las fuentes disponibles son:
  - @ValueSource: un array de valores String, int, long, o double
  - @EnumSource: valores de una enumeración (java.lang.Enum)
  - @CsvSource: valores separados por coma, en formato CSV (comma-separated values)
  - @CsvFileSource: valores en formato CSV en un fichero localizado en el classpath
  - @MethodSource: un método estático de la clase que proporciona un Stream de valores
  - @ArgumentsSource: una clase que proporciona los valores e implementa el interfaz ArgumentsProvider

Pruebas parametrizadas

- Se puede configurar el nombre de salida con:
  - {displayName}: nombre del método que se ejecuta
  - {index}: índice de invocación actual (basado en 1)
  - {arguments}: lista completa de argumentos separados por comas
  - {argumentsWithNames}: lista completa de argumentos separados por comas con nombres de parámetros
  - {0} {1}: argumentos individuales

```
@ParameterizedTest(name = "{index} => ''{0}'' –> {1}")
@CsvSource({ "uno,3", "dos,3", "'tres, cuatro',12" })
void testWithCsvSource(String str, int len) {
```

@ValueSource

- Es la fuente más simples posibles, permite especificar una única colección de valores literales y solo puede usarse para proporcionar un único argumento por invocación de prueba parametrizada.
- Los tipos compatibles son: short, byte, int, long, float, double, char, boolean, String, Class.

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
   assertTrue(argument > 0 && argument < 4);
}
```


Fuentes nulas y vacías

- Con el fin de comprobar los casos de límite y verificar el comportamiento adecuado del código cuando se suministra una entrada incorrecta, puede ser útil suministrar valores null y vacíos en las pruebas con parámetros.
- Las siguientes anotaciones lo permiten:
  - @NullSource: proporciona un único argumento null para el @ParameterizedTest, no se puede usar para un parámetro que tiene un tipo primitivo.
  - @EmptySource: proporciona un único argumento vacío al método, solo para parámetros de los tipos: String, List, Set, Map, las matrices primitivas (por ejemplo, int[], char[][], etc.), matrices de objetos (por ejemplo, String[], Integer[][], etc.). Los subtipos de los tipos admitidos no son compatibles.
  - @NullAndEmptySource: anotación compuesta que combina la funcionalidad de @NullSource y @EmptySource.

```
@ParameterizedTest
@NullSource
@ValueSource(strings = { "uno", "dos", "tres" } )
void testOrdinales(String candidate) {
```

@EnumSource

- Si la lista de valores es una enumeración se anota con @EnumSource:

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit unit) {
```

- En caso de no indicar el tipo, se utiliza el tipo declarado del primer parámetro del método. La anotación proporciona un atributo names opcional que permite: listar los valores (como cadenas) de la enumeración incluidos o excluidos o el patrón inclusión o exclusión. El atributo opcional mode determina el uso de names: INCLUDE (por defecto), EXCLUDE, MATCH_ALL, MATCH_ANY.

```
@ParameterizedTest
@EnumSource(mode = EXCLUDE, names = { "DAYS", "HOURS" })
@ParameterizedTest
void testWithEnumSource(TimeUnit unit) {
```


@CsvSource

- La anotación @CsvSource permite expresar la lista de argumentos como valores separados por comas (cadenas literales con los valores separados).
- El delimitador predeterminado es la coma, pero se puede usar otro carácter configurando el atributo delimiter.
- @CsvSource usa el apostrofe (') como delimitador de cita literal (ignora los delimitadores de valor). Un valor entre apostrofes vacío ('') se interpreta como cadena vacía. Un valor vacío (,,) se interpreta como referencia null, aunque con el atributo nullValues se puede establecer el literal que se interpretará como null (nullValues = "NIL").

```
@ParameterizedTest
@CsvSource({ "uno,3", "dos,3", "'tres, cuatro',12" })
void testWithCsvSource(String str, int len) {
```

- Con la anotación @CsvFileSource se puede usar archivos CSV desde el classpath. Cada línea de un archivo CSV es una invocación de la prueba parametrizada (salvo que comience con # que se interpretará como un comentario y se ignorará). Con el atributo numLinesToSkip se pueden saltar las líneas de cabecera.

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String str, int len) {
```

@MethodSource

- La lista de valores se puede generar mediante un método factoría de la clase de prueba o de clases externas.
- Los métodos factoría deben ser a static y debe devolver una secuencia que se pueda convertir en un Stream. Cada elemento de la secuencia es un juego de argumentos para cada iteración de la prueba, el elemento será a su vez una secuencia si el método de prueba requiere múltiples parámetros.
- El método factoría se asocia con la anotación @MethodSource donde se indica como cadena el nombre de la factoría (el nombre es opcional si coincide con el del método de prueba), si es un método externo de static se proporciona su nombre completo: com.example.clase#método.

```
private static Stream<Arguments> paramProvider() {
   return Stream.of(Arguments.of("uno", 3), Arguments.of("dos", 3), Arguments.of("tres", 4));
}
@ParameterizedTest
@MethodSource("paramProvider")
void testWithMethodSource(String str, int len) {
```

- Arguments es una abstracción que proporciona acceso a una matriz de objetos que se utilizarán para invocar un método @ParameterizedTest.

@ArgumentsSource

- Como alternativa a los métodos factoría se pueden utilizar clases proveedoras de argumentos. Mediante la anotación @ArgumentsSource se asociara el método de prueba. La clase proveedora debe declararse como una clase de nivel superior o como una clase anidada static. Debe implementar la interfaz ArgumentsProvider con el método provideArguments, similar a los métodos factoría.

```
static class CustomArgumentProvider implements ArgumentsProvider {
   @Override
   public Stream<? extends Arguments> provideArguments(ExtensionContext
context) throws Exception {
      return Stream.of(Arguments.of("uno", 2), Arguments.of("dos", 3),
Arguments.of("tres", 4));
   }
}
@ParameterizedTest
@ArgumentsSource(CustomArgumentProvider.class)
void testWithArgumentsSource(String str, int len) {
```

Plantillas de prueba

- Las pruebas parametrizadas nos permiten ejecutar un solo método de prueba varias veces con diferentes parámetros.
- Hay escenarios de prueba donde necesitamos ejecutar nuestro método de prueba no solo con diferentes parámetros, sino también bajo un contexto de invocación diferente cada vez:
  - usando diferentes parámetros
  - preparar la instancia de la clase de prueba de manera diferente, es decir, inyectar diferentes dependencias en la instancia de prueba
  - ejecutar la prueba en diferentes condiciones, como habilitar/deshabilitar un subconjunto de invocaciones si el entorno es "QA"
  - ejecutándose con un comportamiento de devolución de llamada de ciclo de vida diferente; tal vez queramos configurar y eliminar una base de datos antes y después de un subconjunto de invocaciones
- Los métodos de la plantilla de prueba, anotados con @TestTemplate, no son métodos normales, se ejecutarán múltiples veces dependiendo de los valores devueltos por el proveedor de contexto.

Plantillas de prueba

- Las pruebas repetidas y las pruebas parametrizadas son especializaciones integradas de las plantillas de prueba.
- Un método @TestTemplate solo puede ejecutarse si está registrada al menos una extensión TestTemplateInvocationContextProvider. La ejecución del proveedor suministra contextos con los que se invocan todos los métodos de plantilla como si fueran métodos @Test regulares con soporte completo para las mismas devoluciones de llamada y extensiones del ciclo de vida.

```
final List<String> fruits = Arrays.asList("apple", "banana", "lemon");

@TestTemplate
@ExtendWith(MyTestTemplateInvocationContextProvider.class)
void testTemplate(String fruit) {
   assertTrue(fruits.contains(fruit));
}
```

Plantillas de prueba

```
public class MyTestTemplateInvocationContextProvider implements TestTemplateInvocationContextProvider {
    @Override
    public boolean supportsTestTemplate(ExtensionContext context) { return true; }
    @Override
    public Stream<TestTemplateInvocationContext> provideTestTemplateInvocationContexts(ExtensionContext context) {
        return Stream.of(invocationContext("apple"), invocationContext("banana"));
    }
    private TestTemplateInvocationContext invocationContext(String parameter) {
        return new TestTemplateInvocationContext() {
            @Override
            public String getDisplayName(int invocationIndex) { return parameter; }
            @Override
            public List<Extension> getAdditionalExtensions() {
                return Collections.singletonList(new ParameterResolver() {
                    @Override
                    public boolean supportsParameter(ParameterContext parameterContext,
                            ExtensionContext extensionContext) {
                        return parameterContext.getParameter().getType().equals(String.class);
                    }
                    @Override
                    public Object resolveParameter(ParameterContext parameterContext,
                            ExtensionContext extensionContext) {
                        return parameter;
                    }
                });
            }
        };
    }
}
```


Pruebas Dinámicas

- Las pruebas dinámicas permiten crear, en tiempo de ejecución, un número variable de casos de prueba. Cada uno de ellos se lanza independientemente, por lo que ni su ejecución ni el resultado dependen del resto. El uso más habitual es generar batería de pruebas personalizada sobre cada uno de los estados de entrada de un conjunto (dato o conjunto de datos).
- Para implementar los pruebas dinámicas disponemos de la clase DynamicTest, que define un caso de prueba: el nombre que se mostrará en el árbol al ejecutarlo y la instancia de la interfaz Executable con el propio código que se ejecutará (método de prueba).
- Un método anotado con @TestFactory devuelve un conjunto iterable de DynamicTest (este conjunto es cualquier instancia de Iterator, Iterable o Stream).
- Al ejecutar se crearán todos los casos de prueba dinámicos devueltos por la factoría y los tratará como si se tratasen de métodos regulares anotados con @Test pero los métodos @BeforeEach y @AfterEach se ejecutan para el método @TestFactory pero no para cada prueba dinámica.

Pruebas Dinámicas

```
@TestFactory
Collection<DynamicTest> testFactory() {
   ArrayList<DynamicTest> testBattery = new ArrayList<DynamicTest>();
   DynamicTest testKO = dynamicTest("Should fail", () -> assertTrue(false));
   DynamicTest testOK = dynamicTest("Should pass", () -> assertTrue(true));
   int state = 1; // entity.getState();
   boolean rslt = true; // entity.isEnabled();
   if (rslt)
      testBattery.add(dynamicTest("Enabled", () -> assertTrue(true)));
   else
      testBattery.add(dynamicTest("Disabled", () -> assertTrue(true)));
   switch (state) {
   case 1:
      testBattery.add(testOK);
      break;
   case 2:
      testBattery.add(testKO);
      break;
   case 3:
      testBattery.add(testOK);
      testBattery.add(testKO);
      break;
   }
   return testBattery;
}
```


Desactivar pruebas

- Se pueden deshabilitar clases de prueba completas o métodos de prueba individuales mediante anotaciones.
- @Disabled: Deshabilita incondicionalmente todos los métodos de prueba de una clase o métodos individuales.
- @EnabledOnOs y @DisabledOnOs: Habilita o deshabilita una prueba para un determinado sistema operativo.

  ```
  @EnabledOnOs({ OS.LINUX, OS.MAC })
  ```
- @EnabledOnJre, @EnabledForJreRange, @DisabledOnJre y @DisabledForJreRange: Habilita o deshabilita una prueba para versiones particulares o un rango de versiones del Java Runtime Environment (JRE).

  ```
  @EnabledOnJre(JAVA_8), @EnabledForJreRange(min = JAVA_9, max = JAVA_11)
  ```
- @EnabledIfSystemProperty y @DisabledIfSystemProperty: Habilita o deshabilita una prueba en función del valor de una propiedad del JVM.

  ```
  @EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
  ```
- @EnabledIfEnvironmentVariable y @DisabledIfEnvironmentVariable: Habilita o deshabilita una prueba en función del valor de variable de entorno.

  ```
  @EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
  ```

Asunciones

- La clase org.junit.jupiter.api.Assumptions es una colección de métodos de utilidad que permite la ejecución de pruebas condicionales basadas en suposiciones.
- En contraste directo con afirmaciones fallidas, las asunciones ignoran la prueba (skipped) cuando fallan.
- Las asunciones se usan generalmente cuando no tiene sentido continuar la ejecución de un método de prueba dado cuando depende de algo que no existe en el entorno de ejecución actual.
- Las asunciones disponibles son:
  - assumeTrue(boolean assumption, [message]): Continua si la condición booleana es true.
  - assumeFalse(boolean assumption, [message]): Continua si la condición booleana es false.
  - assumingThat(boolean assumption, executable): Ejecuta solo si se cumple la condición.

Ejecución condicional

- Las clases y métodos de prueba se pueden etiquetar mediante la anotación @Tag. Las etiquetas se pueden usar más tarde para incluir y/o excluir (filtrar) del descubrimiento y la ejecución de la prueba.

```
@Tag("Model")
class EntidadTest {
```

- Se recomienda crear anotaciones propias para las etiquetas:

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("smoke")
public @interface Smoke { }
```

En Eclipse: Run configurations → Test → Include and exclude tags

Orden de ejecución de prueba

- Por defecto, los métodos de prueba se ordenarán usando un algoritmo que es determinista pero intencionalmente no obvio.
- Para controlar el orden en que se ejecutan los métodos de prueba, hay que anotar la clase con @TestMethodOrder:
  - MethodOrderer.Alphanumeric: ordena los métodos de prueba alfanuméricamente en función de sus nombres y listas de parámetros formales.
  - MethodOrderer.OrderAnnotation: ordena los métodos de prueba numéricamente según los valores especificados a través de la anotación @Order.

```
@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {
   @Test
   @Order(1)
   void nullValues() { }
```


Tiempos de espera

- La anotación @Timeout permite declarar que una prueba, factoría de pruebas, plantilla de prueba o un método de ciclo de vida deberían dar error si su tiempo de ejecución excede una duración determinada.
- La unidad de tiempo para la duración predeterminada es segundos pero es configurable.

```
@Test
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
void failsIfExecutionTimeExceeds100Milliseconds() {
      try {
         Thread.sleep(150);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
}
```

- Las aserciones de timeout solo miden el tiempo de la propia aserción, en caso de excederlo dan la prueba como fallida.

Modelo de extensión

- El nuevo modelo de extensión de JUnit 5, la Extension API, permite ampliar el modelo de programación con funcionalidades personalizadas.
- Gracias al modelo de extensiones, frameworks externos pueden proporcionar integración con JUnit 5 de una manera sencilla.
- Hay 3 formas de usar una extensión:
  - Declarativamente, usando la anotación @ExtendWith (se puede usar a nivel de clase o de método)

```
@ExtendWith({ DatabaseExtension.class, WebServerExtension.class })
class MyFirstTests {
```

  - Programáticamente, usando la anotación @RegisterExtension (anotando campos en las clases de prueba)

```
@RegisterExtension
static WebServerExtension server = WebServerExtension.builder()
    .enableSecurity(false)
    .build();
```

  - Automáticamente, usando el mecanismo de carga de servicios de Java a través de la clase java.util.ServiceLoader

Inyección de dependencia

- La inyección de dependencia en JUnit Jupiter permite que los constructores y métodos de prueba tengan parámetros. Estos parámetros deben ser resueltos dinámicamente en tiempo de ejecución por un resolutor de parámetros registrado.
- ParameterResolveres es la extensión para proporcionar resolutores de parámetros que se pueden registrar mediante anotaciones @ExtendWith. Aquí es donde el modelo de programación de Júpiter se encuentra con su modelo de extensión.
- JUnit Jupiter proporciona algunos solucionadores integrados que se registran automáticamente:
  - TestInfo para acceder a información sobre la prueba que se está ejecutando actualmente.
  - TestReporter para publicar en la consola datos adicionales sobre la ejecución de prueba actual.

```
@Test @Smoke @DisplayName("Cotilla")
void cotilla(TestInfo testInfo, TestReporter testReporter) {
        assertEquals("Cotilla", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("smoke"));
        for(String tag: testInfo.getTags()) testReporter.publishEntry(tag);
}
```

Dobles de prueba

- La regla de oro de las pruebas unitarias, es que una unidad (unit) tiene que ser testeada sin utilizar ninguna de sus dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

Extensiones

- DBUnit: para realizar pruebas que comprueben el correcto funcionamiento de las operaciones de acceso y manejo de datos que se encuentran dentro de las bases de datos.
- Mockito: es uno de los frameworks de Mock más utilizados en la plataforma Java.
- JMockit: es uno de los frameworks mas completos.
- EasyMock: es un framework basado en el patrón record-replay-verify.
- PowerMock: es un framework que extiende tanto EasyMock como Mockito complementándolos

https://site.mockito.org/

MOCKITO

Introducción

- Mockito es un marco de simulación de Java que tiene como objetivo proporcionar la capacidad de escribir con claridad una prueba de unidad legible utilizando un API simple. Se diferencia de otros marcos de simulacros al dejar el patrón de esperar-ejecutar-verificar que la mayoría de los otros marcos utilizan.
- En su lugar, solo conoce una forma de simular las clases e interfaces (no final) y permite verificar y apilar basándose en comparadores de argumentos flexibles.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
```

Extensión Junit 5

- Para poder utilizar la extensión se requiere la dependencia:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
</dependency>
```

- Para registran la extensión:

```
@ExtendWith(MockitoExtension.class)
class MyTest {
```

- Las sentencias de importar son:

```
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;
```

- Para minimiza el código repetitivo y hacer la clase de prueba más legible dispone de anotaciones como alternativa al código imperativo.

Dobles de prueba

- Mockito provee de dos mecanismos para crear dobles de prueba.
  - Mock: Stub que solo proporciona respuestas predefinidas, sin responder a cualquier otra cosa fuera de aquello para lo que ha sido programado. Toman como referencia una clase o interface existente.

```
@Mock private Calculator calculator;  // declarativo

Calculator calculator = mock(Calculator.class); // imperativo
```

  - Spy: Objetos real al que se le suplantan determinados métodos.

```
@Spy private Calculator calculator;

Calculator calculator = spy(Calculator.class);
```

Suplantación

- El proceso de creación de los métodos de suplantación sigue los siguientes patrones:
  - Devolución de un valor:
    ```
    when(mockedList.get(0)).thenReturn("first");
    doReturn("first").when(mockedList).get(0);
    doNothing().when(mockedList).clear(); // Procedimientos
    ```
  - Devolución de multiples valores (iteradores):
    ```
    when(mockedList.get(0)).thenReturn(10).thenReturn(20).thenReturn(30);
    ```
  - Devolución de una excepción:
    ```
    when(mockedList.get(1)).thenThrow(new RuntimeException());
    doThrow(new RuntimeException()).when(mockedList).clear();
    ```
  - Invocación del método real.
    ```
    when(mockedList.get(0)).thenCallRealMethod();
    doCallRealMethod().when(mockedList).clear();
    ```
- Es obligatorio utilizar en la prueba todos los métodos de suplantación o se genera una excepción.

Argumentos

- Mockito verifica los valores de los argumentos siguiendo el estilo natural de Java: mediante el uso de un método equals(). A veces, cuando se requiere una flexibilidad adicional, se pueden usar los emparejadores de argumentos.
  ```
  when(mockedList.get(anyInt())).thenReturn("element");
  ```
- Dispone de una amplia colección de matchers como emparejadores:
  - any, is, that, eq, some, notNull, …
- Si se está utilizando emparejadores de argumentos, todos los argumentos deben ser proporcionados por emparejadores.

Preparar valores

- La mayoría de las pruebas requieren que los valores devueltos sean constantes.
- Para aquellas que requieran la preparación de los valores devueltos se dispone de la interfaz genérica Answer:

```
when(mock.someMethod(anyString())).thenAnswer(
    new Answer() {
        public Object answer(InvocationOnMock invocation) {
            Object[] args = invocation.getArguments();
            Object mock = invocation.getMock();
            // …
            return rslt;
        }
});
```

- También está disponible la versión:

```
doAnswer(Answer).when(mockedList).someMethod(anyString());
```


Verificación

- Mockito suministra el método verify para validar la interacción con los métodos de suplantación o espiar métodos no suplantados. Se comporta como una aserción: si falla la verificación, falla la prueba.
- Verificar que se ha invocado el método:

```
verify(mockedList, description("This will print on failure")).get(0);
```

- Verificar que se ha invocado el método un determinado número de veces:

```
verify(mockedList, times(3)).get(0);
```

- Verificar que se ha invocado el método al menos o como mucho un número de veces:

```
verify(mockedList, atLeast(2)).get(0);
verify(mockedList, atMost(5)).get(0);
```

- Verificar que se no ha invocado el método:

```
verify(mockedList, never()).get(0);
```

Verificación

- Verificación en orden:

```
InOrder inOrder = inOrder(mockedList);
inOrder.verify(mockedList).get(0);
inOrder.verify(mockedList).get(1);
```

- Verificar que no se produjeron mas invocaciones que las explícitamente verificadas:

```
verifyNoMoreInteractions(mockedList);
```

- Verificar que no se produjeron mas invocaciones a ninguno de los métodos:

```
verifyZeroInteractions(mockedList);
```

- Tiempos de espera antes de la verificación:

```
verify(mock, timeout(100)).someMethod();
verify(mock, timeout(100).times(2)).someMethod();
verify(mock, timeout(100).atLeast(2)).someMethod();
```


Verificando argumentos

- A veces, cuando se requiere una flexibilidad adicional para verificar los argumentos, se necesita capturar los parámetros con qué parámetros fueron invocadas utilizando ArgumentCaptor y @Captor: permiten extraer los argumentos en le método de prueba y realizar aseveraciones en ellos.
- Para extraer los parámetros se utiliza el método capture en el verify del método que se quiere capturar.

```
ArgumentCaptor<List<String>> captor = ArgumentCaptor.forClass(List.class);
List mockedList = mock(List.class);
mockedList.addAll(Arrays.asList("uno", "dos", "tres"));

verify(mockedList).addAll(captor.capture());

assertEquals(3, captor.getValue().size());
assertEquals("dos", captor.getValue().get(1));
```

- Los capturadores se pueden declarar mediante anotaciones:

```
@Captor
ArgumentCaptor<List<String>> captor
```

Inyección de dependencias

- Utilizando la anotación @InjectMocks, se pueden inyectar los mocks y spy que requieren los objetos con dependencias.
- Mockito intentará resolver la inyección de dependencia en el siguiente orden:
  - Inyección basada en el constructor: los simulacros se inyectan en el constructor a través de los argumentos (si no se pueden encontrar alguno de los argumentos, se pasan nulos). Si un objeto es creado con éxito a través del constructor, entonces no se aplicarán otras estrategias.
  - Inyección basada en Setter: los mock son inyectados por los tipos de las propiedades. Si hay varias propiedades del mismo tipo, inyectará en aquellas que los nombres de las propiedades y los nombres simulados coincidirán.
  - Inyección directa en el campo: igual que la inyección basada en Setter.
- Hay que tener en cuenta que no se informa de ningún error en caso de que alguna de las estrategias mencionadas fallara.

Inyección de dependencias

```
public class ArticleManagerTest extends SampleBaseTestCase {
    @Mock private ArticleCalculator calculator;
    @Mock(name = "database") private ArticleDatabase dbMock;
    @Spy private UserProvider userProvider = new ConsumerUserProvider();

    @InjectMocks
    private ArticleManager manager;

    @Test public void shouldDoSomething() {
       manager.initiateArticle();
       verify(database).addListener(any(ArticleListener.class));
    }
  }

public class ArticleManager {
    private ArticleCalculator calculator;
    ArticleManager(ArticleCalculator calculator, ArticleDatabase database) {
       // parameterized constructor
    }
    void setDatabase(ArticleDatabase database) { }
```

http://dbunit.sourceforge.net/

PRUEBAS: DBUNIT

Introducción

- El correcto acceso a datos es fundamental en cualquier aplicación. La complejidad de algunos modelos de datos crea la necesidad de pruebas sistemáticas de la capa de datos. Por un lado se necesita probar que la capa de acceso a datos genera el estado correcto en la base de datos (BD). Por otro lado también se necesita probar que ante determinado estado de la BD el código Java se comporta de la manera esperada.
- Sistematizar estas pruebas requiere una manera sencilla de reestablecer el estado de la base de datos. De otra manera surgirían problemas cada vez que un test falle y deje la BD en un estado inconsistente para los siguientes tests.
- DbUnit es un framework de código abierto que fue creado por Manuel Laflamme. Está basado en JUnit, de hecho sus clases extienden el comportamiento de las clases de JUnit. Eso permite la total integración con el resto de pruebas en JUnit.

Características

- DbUnit nos permite solucionar los problemas que surgen si el último caso de prueba ha dejado la base de datos inconsistente.
- Nos proporciona un mecanismo basado en XML para cargar los datos en la BD y para exportarlos desde la BD.
- DbUnit puede trabajar con conjuntos de datos grandes.
- Permite verificar que el contenido de la base de datos es igual a determinado conjunto de datos esperado, a nivel de fichero, a nivel de consulta o bien a nivel de tabla.
- Proporciona una forma de aislar los experimentos en distintos casos de prueba individuales, uno por cada operación.

Ciclo de vida

- El ciclo de vida de las pruebas con DbUnit es el siguiente:
  1. Eliminar el estado previo de la BD resultante de pruebas anteriores (en lugar de restaurarla tras cada test).
  2. Cargar los datos necesarios para las pruebas de la BD (sólo los necesarios para cada test).
  3. Utilizar los métodos de la librería DbUnit en las aserciones para realizar el test.

Prácticas recomendadas

- La documentación de DbUnit establece una serie de pautas como prácticas recomendadas o prácticas adecuadas (best practices).
  - Usar una instancia de la BD por cada desarrollador. Así se evitan interferencias entre ellos.
  - Programar las pruebas de tal manera que no haya que restaurar el estado de la base de datos tras el test. No pasa nada si la base de datos se queda en un estado diferente tras el test. Dejarlo puede ayudar para encontrar el fallo de determinada prueba. Lo importante es que la base de datos se pone en un estado conocido antes de cada test.
  - Usar múltiples conjuntos de datos pequeños en lugar de uno grande. Cada prueba necesita un conjunto de tablas y de registros, no necesariamente toda la base de datos.
  - Inicializar los datos comunes sólo una vez para todos los tests. Si hay datos que sólo son de lectura, no tenemos por qué tocarlos si nos aseguramos de que las pruebas no los modifican.

Clases e interfaces

- DBTestCase hereda de la clase TestCase de JUnit y proporciona métodos para inicializar y restaurar la BD antes y después de cada test. Utiliza la interfaz IDatabaseTester para conectar con la BD. A partir de la versión 2.2 de DbUnit se ha pasado a la alternativa de utilizar directamente el IDatabaseTester.
- IDatabaseTester devuelve conexiones a la base de datos, del tipo IDatabaseConnection. La implementación que nosotros vamos a utilizar es la de JDBC, JdbcDatabaseTester. Tiene métodos onSetUp(), setSetUpOperation(op), onTearDown(), setTearDownOperation(op), getConnection(), entre otros.
- IDatabaseConnection es la interfaz a la conexión con la base de datos y cuenta con métodos para crear un conjunto de datos createDataSet(), crear una lista concreta de tablas, createDataSet(listaTablas), crear una tabla extraída de la base de datos, createTable(tabla), crear una tabla con el resultado de una query sobre la BD con createQueryTable(tabla, sql), y otros métodos como getConnection(), getConfig(), getRow().

Clases e interfaces

- La interfaz IDataSet representa una colección de tablas y se utiliza para situar la BD en un estado determinado, así como para comparar el estado actual de la BD con el estado esperado. Dos implementaciones son QueryDataSet y FlatXmlDataSet. Esta última implementación sirve para importar y exportar conjuntos de datos a XML en un formato como el siguiente:

```
<?xml version='1.0' encoding='UTF-8'?>
    <dataset>
        <TEST_TABLE COL0="row 0 col 0" COL1="row 0 col 1" COL2="row 0 col
        2"/>
        <TEST_TABLE COL1="row 1 col 1"/>
        <SECOND_TABLE COL0="row 0 col 0" COL1="row 0 col 1" />
        <EMPTY_TABLE/>
</dataset>
```

- La clase Assertion que define los métodos estáticos para realizar las comparaciones: assertEquals(IDataSet, IDataSet) y assertEquals(ITable, ITable)

Configurar

- Referencias
  - DBunit
  - JUnit
  - Jakarta Commons IO: utilidades de entrada y salida
  - Slf4j: frontend para frameworks de logging
- Necesitaremos una carpeta de recursos resources donde guardar los recursos XML.
- En la carpeta de recursos se crearan ficheros independientes para cada prueba, tanto para la inicialización de la BD, como para el estado esperado de la BD tras cada prueba:
  - db-init.xml
  - db-expected.xml

Test

```
public class MiJDBCDAOTest {
   private JDBCDAO dao;
   private IDatabaseTester databaseTester;

@Before
public void setUp() throws Exception {
   //Obtener instancia del DAO que testeamos
   dao = new JDBCPeliculaDAO();
   //Acceder a la base de datos
   databaseTester = new JdbcDatabaseTester("com.mysql.jdbc.Driver",
      "jdbc:mysql://localhost/databasename", "username", "password");
   //Inicializar el dataset en la BD
   FlatXmlDataSetBuilder builder = new FlatXmlDataSetBuilder();
   IDataSet dataSet = builder.build(this.getClass().getResourceAsStream("/db-init.xml"));
   databaseTester.setDataSet(dataSet);
   //Llamar a la operación por defecto setUpOperation
   databaseTester.onSetup();
}
@After
public void tearDown() throws Exception {
   databaseTester.onTearDown();
}
```


Test

```
@Test
public void Case1Test() throws Exception {
   //Realizar la operación con el JDBC DAO
   ...
   // Conectar a la base de datos MySQL
   IDatabaseConnection connection = databaseTester.getConnection();
   DatabaseConfig dbconfig = connection.getConfig();
   dbconfig.setProperty(
         "http://www.dbunit.org/properties/datatypeFactory",
         new MySqlDataTypeFactory());
   // Crear el DataSet de la base de datos MySQL
   QueryDataSet partialDataSet = new QueryDataSet(connection);
   partialDataSet.addTable("tabla1");
   // Escribir el DataSet en XML para después compararlo con el esperado
   File outputFile = new File("db-output.xml");
   FlatXmlDataSet.write(partialDataSet, new FileOutputStream(outputFile));
   // Obtener los datos esperados del XML
   URL url = IDatabaseTester.class.getResource("/db-expected1.xml");
   Assert.assertNotNull(url);
   File inputFile = new File(url.getPath());
   // Comparar los ficheros XML
   Assert.assertEquals(FileUtils.readFileToString(inputFile, "UTF8"), FileUtils.readFileToString(outputFile, "UTF8"));
}
```

GenerateData

- GenerateData es una herramienta para la generación automatizada de juegos de datos.
- Ofrece ya una serie de datos precargados en BBDD y un conjunto de tipos de datos bastante amplio, así como la posibilidad de generar tipos genéricos.
- Podemos elegir en que formato se desea la salida de entre los siguientes:
  - CSV
  - Excel
  - HTML
  - JSON
  - LDIF
  - Lenguajes de programación (JavaScript, Perl, PHP, Ruby, C#)
  - SQL (MySQL, Postgres, SQLite, Oracle, SQL Server)
  - XML
- Online: http://www.generatedata.com/?lang=es
- Instalación (PHP): http://benkeen.github.io/generatedata/

Mockaroo

- https://www.mockaroo.com/
- Mockaroo permite descargar rápida y fácilmente grandes cantidades de datos realistas de prueba generados aleatoriamente en función de sus propias especificaciones que luego puede cargar directamente en su entorno de prueba utilizando formatos CSV, JSON, XML, Excel, SQL, … No se requiere programación y es gratuita (para generar datos de 1.000 en 1.000).
- Los datos realistas son variados y contendrán caracteres que pueden no funcionar bien con nuestro código, como apóstrofes o caracteres unicode de otros idiomas. Las pruebas con datos realistas harán que la aplicación sea más robusta porque detectará errores en escenarios de producción.
- El proceso es sencillo ya que solo hay que ir añadiendo nombres de campos y escoger su tipo. Por defecto nos ofrece mas de 140 tipos de datos diferentes que van desde nombre y apellidos (pudiendo escoger estilo, género, etc...) hasta ISBNs, ubicaciones geográficas o datos encriptados simulados.
- Además es posible hacer que los datos sigan una distribución Normal o de Poisson, secuencias, que cumplan una expresión regular o incluso que fuercen cadenas complicadas con caracteres extraños y cosas así. Tenemos la posibilidad de crear fórmulas propias para generarlos, teniendo en cuenta otros campos, condicionales, etc... Es altamente flexible.

MÉTRICAS PARA EL PROCESO DE PRUEBAS DE SOFTWARE

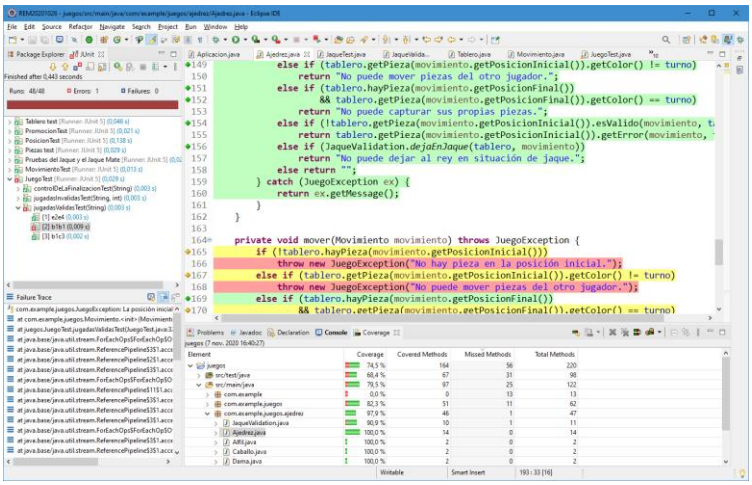
Métricas de código

- La mayor complejidad de las aplicaciones de software moderno también aumenta la dificultad de hacer que el código confiable y fácil de mantener.
- Las métricas de código son un conjunto de medidas de software que proporcionan a los programadores una mejor visión del código que están desarrollando.
- Aprovechando las ventajas de las métricas del código, los desarrolladores pueden entender qué tipos o métodos deberían rehacerse o más pruebas.
- Los equipos de desarrollo pueden identificar los posibles riesgos, comprender el estado actual de un proyecto y realizar un seguimiento del progreso durante el desarrollo de software.
- Los desarrolladores pueden usar el IDE para generar datos de métricas de código que medir la complejidad y el mantenimiento del código administrado.

Cobertura de código

- La herramienta de cobertura de código Java EclEmma (basa en la biblioteca de cobertura de código JaCoCo), integrada en Eclipse, lleva el análisis de cobertura de código directamente al entorno de trabajo:
  - Descripción general de la cobertura: La vista Cobertura enumera los resúmenes de cobertura para los proyectos Java, lo que permite profundizar en el nivel de método.
  - Resaltado de fuente: El resultado de una sesión de cobertura también es directamente visible en los editores de fuente de Java. Un código de color personalizable resalta las líneas totalmente o parcialmente cubiertas y las no cubiertas. Esto funciona tanto para código fuente propio como para fuentes adjuntas a bibliotecas externas instrumentadas.
  - Importación y exportación: los datos de cobertura se pueden exportar en formato HTML, XML o CSV o como archivos de datos de ejecución de JaCoCo (*.exec para su importación).
- Para el análisis para la cobertura de la prueba admite:
  - Diferentes contadores.
  - Varias sesiones de cobertura.
  - Fusionar sesiones.

Cobertura de código



Contadores de cobertura

- El informe utiliza un conjunto de contadores diferentes para calcular métricas de cobertura. Todos estos contadores se derivan de la información contenida en archivos de clase Java que básicamente son instrucciones de código de bytes de Java e información de depuración opcionalmente incrustada en archivos de clase.
- Este enfoque permite una instrumentación y un análisis eficientes sobre la marcha de las aplicaciones, incluso cuando no se dispone de código fuente.
- En la mayoría de los casos, la información recopilada se puede asignar al código fuente y visualizar hasta la granularidad de nivel de línea.
- De todos modos, existen limitaciones para este enfoque. Los archivos de clase deben compilarse con información de depuración para calcular la cobertura de nivel de línea y proporcionar un resaltado de fuente. No todas las construcciones del lenguaje Java se pueden compilar directamente en el código de bytes correspondiente. En tales casos, el compilador de Java crea los llamados código sintético que a veces da como resultado resultados inesperados de cobertura de código.

Contadores de cobertura

- Instrucciones (Cobertura C0) - Instructions counters
  - Los recuentos de unidades más pequeñas son instrucciones de código de un solo byte de Java. La cobertura de instrucciones proporciona información sobre la cantidad de código que se ha ejecutado o se ha perdido. Esta métrica es completamente independiente del formato de origen y siempre está disponible, incluso en ausencia de información de depuración en los archivos de clase.
- Bifurcaciones (Cobertura C1) - Branches counters
  - También calcula la cobertura de bifurcaciones para todos los if y switch. Esta métrica cuenta el número total de tales ramas en un método y determina el número de ramas ejecutadas o perdidas. La cobertura de bifurcaciones siempre está disponible, incluso en ausencia de información de depuración en los archivos de clase. El manejo de excepciones no se considera ramas en el contexto de esta definición de contador.
    - Sin cobertura: No se han ejecutado ramas en la línea (diamante rojo)
    - Cobertura parcial: Solo se ha ejecutado una parte de los ramales de la línea (diamante amarillo)
    - Cobertura total: se han ejecutado todas las ramas de la línea (diamante verde)

Contadores de cobertura

- Líneas - Lines counters
  - Para todos los archivos de clase que se han compilado con información de depuración, se puede calcular la información de cobertura para líneas individuales. Una línea fuente se considera ejecutada cuando se ha ejecutado al menos una instrucción asignada a esta línea.
    - Sin cobertura: no se ha ejecutado ninguna instrucción en la línea (fondo rojo)
    - Cobertura parcial: solo se ha ejecutado una parte de la instrucción en la línea (fondo amarillo)
    - Cobertura total: se han ejecutado todas las instrucciones de la línea (fondo verde)
- Métodos - Methods counters
  - Cada método no abstracto contiene al menos una instrucción. Un método se considera ejecutado cuando se ha ejecutado al menos una instrucción. Como funciona a nivel de código de bytes, también los constructores y los inicializadores estáticos se cuentan como métodos. Es posible que algunos de estos métodos no tengan una correspondencia directa en el código fuente de Java, como constructores o inicializadores implícitos.
- Clases - Type counters
  - Una clase (o interfaz con código) se considera ejecutada cuando se ha ejecutado al menos uno de sus métodos (incluidos constructores e inicializadores).

Contadores de cobertura

- Complejidad ciclomática
  - También se calcula la complejidad ciclomática para cada método no abstracto y se resume para clases, paquetes y grupos. La complejidad ciclomática es el número mínimo de rutas pasar por todas las instrucciones. Por lo tanto, el valor de complejidad puede servir como una indicación del número de casos de prueba unitarios para cubrir por completo el código.
  - La definición formal de la complejidad ciclomática se basa en la representación del gráfico de flujo de control de un método como un grafo dirigido:
    - Complejidad = [número de aristas] – [número de nodos]+ 2
  - El contador calcula la complejidad ciclomática de un método con la ecuación equivalente:
    - Complejidad = [número de ramas] – [número de puntos de decisión] + 1
  - Según el estado de cobertura de cada bifurcación, JaCoCo también calcula la complejidad cubierta y perdida para cada método. La complejidad perdida nuevamente es una indicación de la cantidad de casos de prueba que faltan para cubrir completamente un módulo. Dado que no se considera el manejo de excepciones como bifurcaciones, los bloques try / catch de las ramas tampoco aumentarán la complejidad.

Calidad de las pruebas

- Se insiste mucho en que la cobertura de test unitarios de los proyectos sea lo más alta posible, pero es evidente que cantidad (de test, en este caso) no siempre implica calidad, la calidad no se puede medir "al peso", y es la calidad lo que realmente importa.
- La cobertura de prueba tradicional (líneas, instrucciones, rama, etc.) mide solo qué código ejecuta las pruebas. No comprueba que las pruebas son realmente capaces de detectar fallos en el código ejecutado, solo pueden identificar el código que no se ha probado.
- Los ejemplos más extremos del problema son las pruebas sin afirmaciones (poco comunes en la mayoría de los casos). Mucho más común es el código que solo se prueba parcialmente, cubrir todo los caminos no implica ejercitar tolos las clases de equivalencia y valores límite.
- La calidad de las pruebas también debe ser puesta a prueba: No serviría de tener una cobertura del 100% en test unitarios, si no son capaces de detectar y prevenir problemas en el código.
- La herramienta que testea los test unitarios son los test de mutaciones: Es un test de los test.

Pruebas de mutaciones

- Los pruebas de mutaciones son las pruebas de las pruebas unitarias y el objetivo es tener una idea de la calidad de las pruebas en cuanto a fiabilidad.
- Su funcionamiento es relativamente sencillo: la herramienta que se utilice debe generar pequeños cambios en el código fuente. A estos pequeños cambios se les conoce como mutaciones y crean mutantes.
- Una vez creados los mutantes, se lanzan todos los tests:
  - Si los test unitarios fallan, es que han sido capaces de detectar ese cambio de código. A esto se le llama matar al mutante.
  - Si, por el contrario, los test unitarios pasan, el mutante sobrevive y la fiabilidad (y calidad) de los tests unitarios queda en entredicho.
- Los test de mutaciones presentan informes del porcentaje de mutantes detectados: cuanto más se acerque este porcentaje al 100%, mayor será la calidad de nuestros test unitarios.

Pruebas de mutaciones

- Los mutantes cuyo comportamiento es siempre exactamente igual al del programa original se los llama mutantes funcionalmente equivalentes o, simplemente, mutantes equivalentes, y representan "ruido" que dificulta el análisis de los resultados.
- Para poder matar a un mutante:
  - La sentencia mutada debe estar cubierta por un caso de prueba.
  - Entre la entrada y la salida debe crearse un estado intermedio erróneo.
  - El estado incorrecto debe propagarse hasta la salida.
- La puntuación de mutación para un conjunto de casos de prueba es el porcentaje de mutantes no equivalentes muertos por los datos de prueba:
  - Mutación Puntuación = 100 * D / (N - E)

  donde D es el número de mutantes muertos, N es el número de mutantes y E es el número de mutantes equivalentes

Pitest

- http://pitest.org/
- Pitest es un sistema de prueba de mutación de última generación que proporciona una cobertura de prueba estándar de oro para Java y jvm. Es rápido, escalable y se integra con herramientas modernas de prueba y construcción.
- PIT introduce mutaciones en el código (bitcode) y ejecuta las pruebas unitarias contra versiones modificadas automáticamente del código de la aplicación.
- Cuando el código de la aplicación cambia, debería producir resultados diferentes y hacer que las pruebas unitarias fallen. Si una prueba unitaria no falla en esta situación, puede indicar un problema con el conjunto de pruebas.
- La calidad de las pruebas se puede medir a partir del porcentaje de mutaciones muertas.

Pitest

- PIT se puede ejecutar con ant, maven, gradle y otros, o integrarlo en Eclipse, IntelliJ, …
- La mayoría de los sistemas de prueba de mutaciones para Java son lentos, difíciles de usar y están escritos para satisfacer las necesidades de la investigación académica en lugar de los equipos de desarrollo reales. PIT es rápido: puede analizar en minutos lo que tomaría días en otros sistemas.
- Los informes producidos por PIT están en un formato HTML fácil de leer que combina la cobertura de línea y la información de cobertura de mutación.

BUENAS PRACTICAS

Características de una buena prueba unitaria

- Rápida. No es infrecuente que los proyectos maduros tengan miles de pruebas unitarias. Las pruebas unitarias deberían tardar muy poco tiempo en ejecutarse. Milisegundos.
- Aislada. Las pruebas unitarias son independientes, se pueden ejecutar de forma aislada y no tienen ninguna dependencia en ningún factor externo, como sistemas de archivos o bases de datos.
- Reiterativa. La ejecución de una prueba unitaria debe ser coherente con sus resultados, es decir, devolver siempre el mismo resultado si no cambia nada entre ejecuciones.
- Autocomprobada. La prueba debe ser capaz de detectar automáticamente si el resultado ha sido correcto o incorrecto sin necesidad de intervención humana.
- Oportuna. Una prueba unitaria no debe tardar un tiempo desproporcionado en escribirse en comparación con el código que se va a probar. Si observa que la prueba del código tarda mucho en comparación con su escritura, considere un diseño más fácil de probar.

Asignar nombre a las pruebas

- El nombre de la prueba debe constar de tres partes:
  - Nombre del método que se va a probar.
  - Escenario en el que se está probando.
  - Comportamiento esperado al invocar al escenario.
- Los estándares de nomenclatura son importantes porque expresan de forma explícita la intención de la prueba.

Organizar el código de la prueba

- Prepara, actuar, afirmar es un patrón común al realizar pruebas unitarias. Como el propio nombre implica, consta de tres acciones principales:
  - Prepara los objetos, crearlos y configurarlos según sea necesario.
  - Actuar en un objeto.
  - Afirmar que algo es como se espera.
- Separa claramente en secciones lo que se está probando de los pasos preparación y verificación.
- Las secciones solo deben aparecer una vez como máximo y en el orden establecido.
- Minimiza la posibilidad de mezclar aserciones con el código para "actuar".

Preparación mínima

- La sección de preparación, con la entrada del caso de prueba, debe ser lo más sencilla posible, lo imprescindible para comprobar el comportamiento que se está probando.
- Las pruebas se hacen más resistentes a los cambios futuros en el código base y más cercano al comportamiento de prueba que a la implementación.
- Las pruebas que incluyen más información de la necesaria tienen una mayor posibilidad de incorporar errores en la prueba y pueden hacer confusa su intención. Al escribir pruebas, el usuario quiere centrarse en el comportamiento. El establecimiento de propiedades adicionales en los modelos o el empleo de valores distintos de cero cuando no es necesario solo resta de lo que se quiere probar.

Actuación mínima

- Al escribir las pruebas hay que evitar introducir condiciones lógicas como if, switch, while, for, etc.
- Minimiza la posibilidad de incorporar un error a las pruebas.
- El foco está en el resultado final, en lugar de en los detalles de implementación.
- Al incorporar lógica al conjunto de pruebas, aumenta considerablemente la posibilidad de agregar un error. Cuando se produce un error en una prueba, se quiere saber realmente que algo va mal con el código probado y no en el código que prueba. En caso contrario, restan confianza y las pruebas en las que no se confía no aportan ningún valor.
- El objetivo de la prueba debe ser único, si la lógica en la prueba parece inevitable, denota que el objetivo es múltiple y hay que considerar la posibilidad de dividirla en dos o más pruebas diferentes.

Sustituir literales por constantes

- La asignación de literales a constantes permite dar nombre a los valores, aportando semántica.
- Evita la necesidad de que el lector de la prueba inspeccione el código de producción con el fin de averiguar lo que significa un valor, que hace el valor sea especial.

```
Assert.IsTrue(rslt.Length <= 10)
```

- Muestra explícitamente lo que se intenta probar, en lugar de lo que se intenta lograr.

```
const string VARCHAR_LEN = 10;
Assert.IsTrue(rslt.Length <= VARCHAR_LEN)
```

- Las valores literales pueden provocar confusión al lector de las pruebas. Si una cadena tiene un aspecto fuera de lo normal, puede preguntarse por qué se ha elegido un determinado valor para un parámetro o valor devuelto. Esto obliga a un vistazo más detallado a los detalles de implementación, en lugar de centrarse en la prueba.

Evitar varias aserciones

- Al escribir las pruebas, hay que intentar incluir solo una aserción por prueba. Los enfoques comunes para usar solo una aserción incluyen:
  - Crear una prueba independiente para cada aserción.
  - Usar pruebas con parámetros.
- Si se produce un error en una aserción, no se evalúan las aserciones posteriores.
- Garantiza que no se estén declarando varios casos en las pruebas.
- Proporciona la imagen exacta de por qué se producen errores en las pruebas.
- Al incorporar varias aserciones en un caso de prueba, no se garantiza que se ejecuten todas. Es un todas o ninguna, se sabe por cual fallo pero no si el resto también falla o es correcto, proporcionando la imagen parcial.
- Una excepción común a esta regla es cuando la validación cubre varios aspectos. En este caso, suele ser aceptable que haya varias aserciones para asegurarse de que el resultado está en el estado que se espera que esté.

Refactorizar código

- La refactorización del código de prueba favorece le reutilización y la legibilidad, simplifican las pruebas.
- Salvo que todos los métodos de prueba usen los mismos requisitos, si se necesita un objeto o un estado similar para las pruebas, es preferible usar métodos auxiliares a los métodos de instalación y desmontaje (si existen):
  - Menos confusión al leer las pruebas, puesto que todo el código es visible desde dentro de cada prueba.
  - Menor posibilidad de configurar mas o menos de lo necesario para la prueba.
  - Menor posibilidad de compartir el estado entre las pruebas, lo que crea dependencias no deseadas entre ellas.
- Cada prueba normalmente tendrá requisitos diferentes para funcionar y ejecutarse. Los métodos de instalación y desmontaje son únicos, pero se pueden crear tantos métodos auxiliares como escenarios reutilizables se necesiten.

No validar métodos privados

- En la mayoría de los casos, no debería haber necesidad de probar un método privado.
- Los métodos privados son un detalle de implementación.
- Se puede considerar de esta forma: los métodos privados nunca existen de forma aislada. En algún momento, va a haber un método público que llame al método privado como parte de su implementación. Lo que debería importar es el resultado final del método público que llama al privado.

Aislar las pruebas

- Las dependencias externas afectan a la complejidad de la estrategia de pruebas, hay que aislar a las pruebas de las dependencias externas, sustituyendo las dependencias por dobles de prueba, salvo que se este probando específicamente dichas dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
  - Devuelven resultados determinísticos
  - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
  - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
  - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
  - Permiten incluir atributos o métodos exclusivamente para el testeo.

Cubrir aspectos no evidentes

- Las pruebas no deben cubrir solo los casos evidentes, los correctos, sino que deben ampliarse a los casos incorrectos.
- Un juego de pruebas debe ejercitar la resiliencia: la capacidad de resistir los errores y la recuperación ante los mismos.
- En los cálculos no hay que comprobar solamente si realiza correctamente el calculo, también hay que verificar que es el calculo que se debe realizar.
- Los dominios de los datos determinan la validez de los mismos y fijan la calidad de la información, dichos dominios deben ser ejercitados profundamente.

Respetar los limites de las pruebas

- La pruebas unitarias ejercitan profundamente los componentes de formar aislada centrándose en la funcionalidad, los cálculos, las reglas de dominio y semánticas de los datos. Opcionalmente la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
- Las pruebas de integración se basan en componentes ya probados (unitaria o integración) o en dobles de pruebas y se centran en la estructura de llamadas, secuencias o colaboración, así como en la transición de estados.
- Hay muchos tipos de pruebas de sistema y cada uno pone el foco en un aspecto muy concreto, cada prueba solo debe cubrir un solo aspecto. Las pruebas funcionales del sistema son las pruebas de integración de todo el sistema centrándose en compleción de la funcionalidades y los procesos de negocio, su estructura, disponibilidad y accesibilidad.