Técnicas de Pruebas en Java

INTRODUCCIÓN

Definiciones de Calidad

- Real Academia de la Lengua Española: Propiedad o conjunto de propiedades inherentes a una cosa que permiten apreciarla como igual, mejor o peor que las restantes de su especie.
- ISO 9000: Grado en el que un conjunto de características inherentes cumple con los requisitos.
- Scrum: La capacidad del producto terminado o de las entregas para cumplir con los criterios de aceptación y lograr el valor comercial esperado por el cliente.
- Producto: La calidad es un conjunto de propiedades inherentes a un objeto que le confieren capacidad para satisfacer necesidades implícitas o explícitas.
- Producción: La calidad puede definirse como la conformidad relativa con las especificaciones, al grado en que un producto cumple las especificaciones del diseño.
- Cliente: La calidad significa aportar valor al cliente, esto es, ofrecer unas condiciones de uso del producto o servicio superiores a las que el cliente espera recibir y a un precio accesible.
- La calidad es el resultado de la interacción de la dimensión objetiva (lo que se ofrece) y la dimensión subjetiva (lo que el cliente quiere).

Concepto de Calidad

- La calidad es relativa:
  - Es la propiedad inherente de cualquier cosa que permite que la misma sea valorada, positiva o negativamente, con respecto a cualquier otra de su misma especie.
  - Debe definirse en un contexto: producto o servicio, interna o externa, productor o consumidor, …
- Para conseguir un elevado grado de calidad en el producto o servicio hay que tener en cuenta tres aspectos importantes (dimensiones básicas de la calidad):
  - Dimensión técnica: engloba los aspectos científicos y tecnológicos que afectan al producto o servicio.
  - Dimensión humana: son las relaciones entre clientes y empresas.
  - Dimensión económica: intenta minimizar costos tanto para el cliente como para la empresa.
- Para ello se debe establecer la cantidad justa y deseada de producto que hay que fabricar y ofrecer, el precio exacto del producto y su rápida distribución, el soporte y la sostenibilidad del mismo, …
- Los parámetros de la calificación de la calidad son:
  - Calidad de diseño: es el grado en el que un producto o servicio se ve reflejado en su diseño.
  - Calidad de conformidad: es el grado de fidelidad con el que es reproducido un producto o servicio respecto a su diseño.
  - Calidad de uso: es el grado en el que el producto ha de ser fácil de usar, seguro, fiable, etc.
- El cliente es el nuevo objetivo, las nuevas teorías sitúan al cliente como parte activa de la calificación de la calidad de un producto, intentando crear un estándar en base al punto subjetivo de un cliente. La calidad de un producto no se va a determinar solamente por parámetros duramente objetivos sino incluyendo las opiniones de un cliente que usa determinado producto o servicio.

Alcanzar la calidad

- Para alcanzar la calidad de software, se sugiere tener en cuenta los siguientes aspectos:
  - La calidad se gestiona desde el inicio, no es el resultado de la magia
  - Los requisitos de calidad tienen en cuenta las exigencias del usuario final.
  - Los procesos del desarrollo del software deben estar interrelacionados y conectados con los procesos de aseguramiento y de control de calidad para así tener una mayor certeza de poder cumplir con las exigencias del usuario.
  - La calidad es un sistema que incluye procesos antes, durante y después de la fabricación de las piezas de software que al final se integran en un programa y/o sistema de aplicación.
- La calidad se debe verificar y validar usando diferentes métricas en el ciclo de desarrollo del proyecto y durante el ciclo de vida del producto.
- Las pruebas del software miden el grado de calidad que el producto tiene en cada momento.

Gestión de calidad

- La gestión de calidad es el conjunto de actividades y procesos que se llevan a cabo en una organización con el objetivo de garantizar y mejorar la calidad de los productos o servicios que ofrece.
- Consiste en planificar, coordinar, controlar y evaluar todas las etapas y aspectos relacionados con la calidad, desde el diseño y desarrollo hasta la producción, distribución y atención al cliente.
- La gestión de calidad implica establecer estándares y normas de calidad, implementar sistemas y procesos para cumplir con esos estándares, realizar el control y seguimiento de la calidad, y tomar medidas correctivas y preventivas para asegurar que se cumplan los requisitos de calidad establecidos.
- También implica la capacitación y participación de los empleados en la mejora continua de los procesos y en la satisfacción del cliente.

Gestión de la calidad

La gestión de la calidad (QM Quality Management) es el concepto más amplio que incluye planificación y estrategia. Considera la cadena de valor de un proyecto, proceso o producto de forma completa. Es un concepto global, dentro del cual se incluyen otros conceptos anidados:

- QA (Quality Assurance) o Aseguramiento de la Calidad: Se centra en proporcionar confianza en que se cumplirán los requisitos de calidad. Basada en metodologías y buenas practicas, se enfoca de manera proactiva en los procesos y sistemas.
- QC (Quality Control) o Control de la Calidad: Se centra en el cumplimiento de los requisitos de calidad. Se enfoca de manera reactiva en las partes del sistema y los productos.
- Testing o Pruebas: Es el proceso de detección de errores en un sistema o producto. Ayuda a reducir riesgos e incrementar la confianza.

Pruebas

- Las pruebas consisten en actividades de verificación, validación y exploración que brindan información sobre la calidad y los riesgos relacionados, para establecer el nivel de confianza de que el objeto de prueba podrá entregar el valor comercial buscado en la base de la prueba.
- La verificación es la confirmación mediante examen y mediante la provisión de evidencia objetiva de que se han cumplido los requisitos especificados. Responde a la pregunta: ¿Estamos construyendo correctamente el sistema de TI?
- La validación es la confirmación mediante examen y mediante la provisión de evidencia objetiva de que se han cumplido las demandas para un uso específico previsto. Responde a la pregunta: ¿Estamos construyendo el sistema de TI adecuado?
- La exploración es la actividad de investigar y establecer la calidad y los riesgos del uso de un sistema de TI mediante examen, indagación y análisis. Responde a la pregunta: ¿Cómo se podría (mal)utilizar el sistema de TI?
- Las pruebas deben proporcionar diferentes niveles de información:
  - Detallado: para conocer la calidad de los objetos nuevos y modificados que entregaron e investigar anomalías en el sistema de TI y tomar medidas correctivas cuando sea necesario.
  - Intermedio: para realizar un seguimiento del estado y el progreso.
  - General: para respaldar las decisiones de seguir o no.

Criterios de evaluación

- Las pruebas consisten en evaluar la calidad basándose en criterios. En las pruebas utilizamos criterios para determinar si el objeto de prueba cumple con las expectativas sobre calidad y riesgos.
- Los **criterios de entrada** son los criterios que un objeto (por ejemplo, un documento base de prueba o un objeto de prueba) debe satisfacer para estar listo para usarse en una actividad específica.
- Los **criterios de salida** son los criterios que un objeto (por ejemplo, un documento base de prueba o un objeto de prueba) debe satisfacer para estar listo al final de una actividad o etapa específica del proyecto.
- Los **criterios de aceptación** son los criterios que debe cumplir un objeto de prueba para ser aceptado por un usuario, cliente u otra parte interesada.
- Los **criterios de finalización** son los criterios que un equipo debe satisfacer para haber completado una (grupo de) actividad(es).

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. Es una equivocación por parte del desarrollador o 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. 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.

V & V

- Validación: ¿Estamos construyendo el sistema correcto?
  - Proceso de evaluación de un sistema o componente durante o al final del proceso de desarrollo para comprobar si se satisfacen los requisitos especificados (IEEE Std610.12-1990)
- Verificación: ¿Estamos construyendo correctamente el sistema?
  - Proceso de evaluar un sistema o componente para determinar si los productos obtenidos en una determinada fase de desarrollo satisfacen las condiciones impuestas al comienzo de dicha fase (IEEE Std610.12-1990)

Depuración y Pruebas

- Cuando se han encontrado fallos 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 fallos, mientras que la depuración se refiere a la localización los defectos/errores que se han manifestado en los fallos y corrección de los mismos.
- El proceso de depuración suele requerir los siguientes pasos:
  - Identificación de errores
  - Análisis de errores
  - Corrección y validación

Testing vs QA

- El aseguramiento de calidad o QA (Quality Assurance), esta orientado al proceso de obtener un software de calidad, que viene determinado por las metodologías y buenas practicas empleadas en el desarrollo del mismo, y se inicia incluso antes que el propio proyecto.
- Las pruebas son parte del proceso de QA, validan y verifican la corrección del producto obtenido, destinadas a revelar los errores inherentes a cualquier actividad humana que ni las mejores practicas o metodologías pueden evitar.

Principios fundamentales

- Hay 7 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
  - El proceso de pruebas no puede demostrar la ausencia de defectos
  - La mayoría de defectos relevantes suelen concentrarse en partes muy concretas.
  - Las pruebas se deben ejecutar bajo diferentes condiciones
  - Las pruebas no garantizan ni mejoran la calidad del software
  - Las pruebas tienen un coste
  - El inicio temprano de pruebas ahorran tiempo y dinero

Las pruebas exhaustivas no son viables

- Es inviable (tiempo, coste o recursos) o imposible (cardinalidad) 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.

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% de los casos de uso).
- Una prueba se debe considerar un éxito si detecta un error. Si se supera la prueba, no detecta un error, no significa que no haya error, significa que no se ha detectado. Una prueba fallida es concluyente, una prueba superada no lo es.
- Un proceso de pruebas riguroso puede garantizar una reducción significativa de los posibles fallos y/o defectos del software, sobre todo en las áreas criticas, pero nunca podrá garantizar que no fallará en producción (la falacia de la ausencia de errores).
- Las pruebas reducen la probabilidad de la presencia de defectos que permanezcan sin ser detectados. La ausencia de fallos no demuestran la corrección de un producto software

Agrupación de defectos

- Generalmente, hay ciertos módulos y funcionalidades que son más proclives a presentar incidencias (de mayor prioridad) en comparación al resto de las partes que conforman un producto.
- Encuentra un defecto y encontrarás más defectos cerca. Los defectos aparecen agrupados como hongos o cucarachas, vale la pena investigar un mismo módulo donde se ha detectado un defecto: fallo → defecto → error → defectos → fallos.
- Este es el Principio de Pareto aplicado a las pruebas de software, donde el 80% de los problemas se encuentran en el 20% del código. Existen múltiples razones (complejidad, especificidad, novedad, organizativas, …) que explican esta concentración.
- El Principio de Pareto también se puede aplicar a la funcionalidad, el 80% de los usuarios solo usa el 20% de la funcionalidad.
- La coincidencias de ambos 20% puede llevar a una situación critica.

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, una combinación única, 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. Cuando se ejecutan los mismos casos de pruebas una y otra vez y sin ningún cambio, eventualmente estos dejarán de encontrar defectos nuevos (paradoja del pesticida). La paradoja del pesticida solo resulta beneficiosa en las pruebas de regresión.
- Ejecutar ciclos bajo diferentes tipos de condiciones, permitirá identificar posibles fallos en el sistema que antes no se detectaron y no son fácilmente reproducibles.

Las pruebas no garantizan ni mejoran la calidad del software

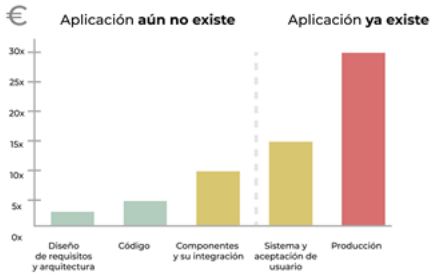
- Las pruebas ayudan a **mejorar la percepción** de la calidad permitiendo la eliminación de los defectos detectados. Una prueba fallida no beneficia directamente al usuario. ¡El valor solo se crea cuando se soluciona el error!
- La calidad del software viene determinada por las metodologías y buenas practicas empleadas en el desarrollo del mismo. Actualmente hay metodologías que ponen el foco en las pruebas.
- 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 dependen de su contexto: la estrategia y el tipo de pruebas serán seleccionados en función del sistema y a los entornos que se pretenden verificar.

Las pruebas tienen un coste

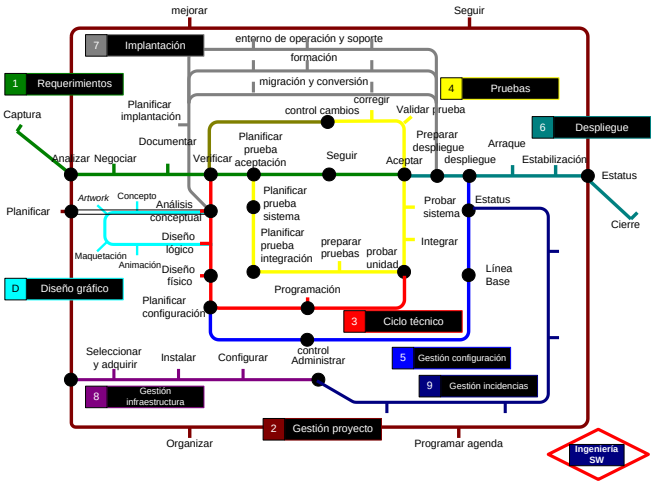
- Aunque exige dedicar esfuerzo para crear y mantener los test (coste para las empresas), los beneficios obtenidos son mayores que la inversión realizada.
- El proceso de prueba implica más que sólo la ejecución de la prueba.
- La creciente inclusión del software como un elemento más de muchos sistemas 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, hay vidas o mucho dinero en juego, como control de tráfico aéreo o 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 muy 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 (pruebas estáticas), 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



Conflicto de intereses

- 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 enfrentará 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 (Happy Path), en lugar de detectar errores.
    - Desgraciadamente los errores seguirán estando, y si el ingeniero de software no los encuentra, lo hará el cliente o el usuario.

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.
- El grupo independiente de prueba está formado por especialista en técnicas de pruebas que basan los casos de prueba en las técnicas, la intuición y la experiencia: ¿Qué casos tienen mas probabilidad de encontrar fallos?¿Dónde se han acumulado errores en el pasado?¿Dónde falla normalmente el software?
- 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.

Roles

- **Jefe de prueba**: lidera el equipo de pruebas. Su responsabilidad es poder informar sobre la calidad del software en cualquier momento de las pruebas. Es responsable de la creación del Plan de Pruebas y Sumario de Pruebas de Evaluación, así como , administración de recursos y solución de problemas.
- **Analistas y diseñadores de pruebas**: analiza los requisitos y especificaciones para identificar y definir los casos de prueba a partir de ellos. Por lo tanto, los diseñadores de pruebas deben poder comprender el contexto empresarial y sentirse cómodos con la metodología. Son necesarios en todos los ciclos de vida del proyecto de prueba.
- **Ingenieros de pruebas**: automatiza las pruebas funcionales o configura las no funcionales. El puesto requiere un alto conocimiento técnico, por ejemplo, experiencia con el desarrollo de software, y debe conocer las herramientas de automatización.

Roles

- **Probadores o Tester**: ejecuta los casos de prueba y evalúa los resultados. Deben documentarse las diferencias con el resultado esperado.
- **Consultor de pruebas**: Rol externo con amplia experiencia que pueden ayudar en la implantación de las pruebas en una organización o en la revisión de proyectos de prueba.

- Existen roles adicionales como especialistas en pruebas funcionales y en no funcionales, expertos en datos de prueba, probadores que participan en la prueba de aceptación y desarrolladores de software.

Variedad de pruebas

- Las pruebas cubren una gran cantidad de escenarios por lo que tienen una gran variedad de clasificaciones en función de la metodología, enfoque, objetivos, criterios, …
  - Tipo: Estáticas y Dinámicas
  - Información: Pruebas de defectos y Pruebas estadísticas
  - Nivel: Unitarias, Integración, Sistema, Aceptación
  - Proceso: Pruebas manuales o automatizadas
  - Características de calidad: Pruebas funcionales y no funcionales (rendimiento, estructurales, de mantenimiento, seguridad, …)
  - Enfoque: Basadas en la especificación, en la estructura o en la experiencia.
  - Estrategias: Pruebas de humo, progresión, regresión, aprendizaje, exploratorias, …

Pruebas estáticas

- Las pruebas estáticas, también conocidas como revisión, son pruebas que examinan productos (como especificaciones de requisitos, manuales o código fuente) sin que se ejecuten programas.
- Esto se puede hacer sin que el objeto de prueba se esté ejecutando, por ejemplo, revisando requisitos o documentos de diseño, código fuente, manuales de usuario, planes de proyecto y prueba y casos de prueba.
- En los modelos de entrega de TI de alto rendimiento, como Scrum y DevOps, una parte importante de las actividades de prueba estáticas se realiza durante el refinamiento de las historias de los usuarios, por lo que no se implementa como una actividad separada, pero aún requiere un enfoque específico. Además, se realiza un análisis estático del código para cumplir con los estándares de codificación.
- Las pruebas estáticas estarán respaldadas por herramientas tanto como sea posible. De hecho, hoy en día existen herramientas que no sólo hacen análisis estático sino que también son capaces de refactorizar el código basándose en patrones de refactorización estándar. Aún así, en general, las pruebas estáticas son una combinación de inteligencia humana y poder de herramientas.

Pruebas dinámicas

- Las pruebas dinámicas son pruebas mediante la ejecución del objeto de prueba, es decir, la ejecución de una aplicación.
- Cuando ejecutamos una prueba, en otras palabras, ejecutamos el objeto de la prueba (que puede variar desde un único módulo de programa hasta un proceso de negocio completo de extremo a extremo en múltiples organizaciones), eso es lo que llamamos prueba dinámica.
- Las pruebas dinámicas se pueden realizar manualmente, pero para muchas variedades de pruebas, se recomienda encarecidamente el soporte de herramientas de automatización de pruebas.

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 que se manifiestan en forma de fallos.
- Los fallos demuestran la existencia de defectos que han sido ocasionados por errores.
- Cuando se han encontrado defectos en un programa, éstos deben ser localizados y eliminados. A este proceso se le denomina **depuración**.

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 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.

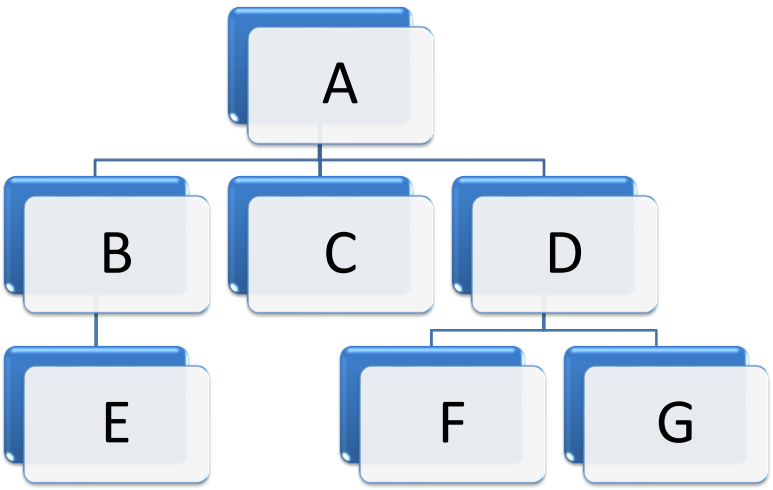
Pruebas de Integración

- 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 dobles de pueba, 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 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 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 configuración:** consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **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.
- 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.

Pruebas alfa y beta

- Las pruebas alfa y beta son la alternativa a las pruebas de aceptación en el software comercial de distribución masiva (COTS). Las pruebas alfa y beta suelen ser utilizadas por los desarrolladores de COTS que desean obtener retroalimentación de los usuarios, clientes y/u operadores existentes o potenciales antes de que el producto de software sea puesto en el mercado. Uno de los objetivos de las pruebas alfa y beta es generar la confianza de que se pueden utilizar el sistema en condiciones normales y cotidianas, con la mínima dificultad, coste y riesgo.
- Las pruebas alfa son pruebas de software realizadas en las fases iniciales del proyecto cuando el sistema está en desarrollo y cuyo objetivo es asegurar que lo que estamos desarrollando es probablemente correcto y útil para el cliente. Si las pruebas alfa devuelven buenos comentarios positivos entonces podríamos seguir por esa vía. Si devolvieran resultados muy negativos tendríamos que replantear el problema para adaptarse mejor a los requerimientos.
- Las pruebas beta son las pruebas de software que se realizan cuando el sistema está completado y, en teoría, es correcto, antes de pasar a producción. Dado que no es viable realizar pruebas exhaustivas, siempre habrá fallos que no han sido descubiertos por los desarrolladores ni por el equipo de pruebas. Las pruebas beta son pruebas para localizar esos problemas no detectados y poder corregirlos antes de liberar la versión definitivas; la prueba debería ser realizada por usuarios finales.

Pruebas de Regresión

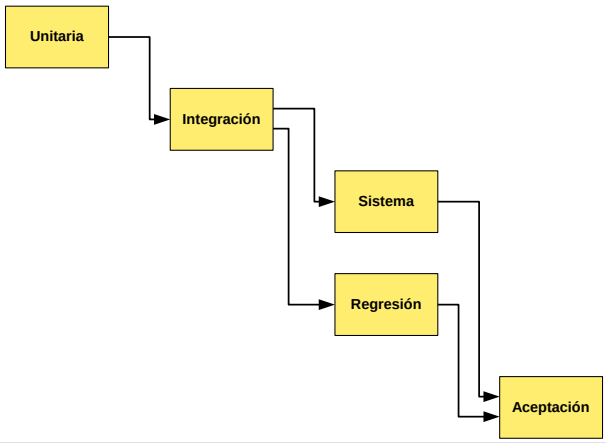
- El objetivo de las pruebas de regresión es eliminar el efecto onda, es decir, comprobar que los cambios sobre un componente de un sistema de información, no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- Las pruebas de regresión se deben llevar 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 (también conocidas como **pruebas de progresión o confirmación**), sino que también es necesario controlar que las modificaciones no produzcan efectos secundarios 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.

Niveles de pruebas y orden de ejecución.

- De tal forma que la secuencia de pruebas es:



Smoke Testing y Sanity Testing

- Un smoke test (prueba de humo) es una prueba superficial realizada para asegurar de que una nueva distribución puede realizar la funcionalidad básica sin encontrar errores críticos. El objetivo principal de un smoke test, subconjunto de pruebas rápidas y automatizadas, es determinar si la compilación del software desplegado es suficientemente estable antes de realizar pruebas más exhaustivas y costosas.
- Un sanity test (prueba de cordura o sanidad) es una prueba superficial realizada para asegurar para comprobar si un cambio o nueva funcionalidad de una nueva distribución funciona correctamente. Un sanity test es un tipo de pruebas de regresión más enfocado y específico para validar los componentes mejorados en lugar del producto completo.
- Las pruebas de sanidad se ejecutan después de haber superado las pruebas de humo y antes de acometer las pruebas de regresión.
- Las pruebas de humo y sanidad son parte de las estrategias de pruebas y ahorran mucho tiempo y esfuerzo porque evitan que los equipos de control de calidad pierdan tiempo en pruebas más profundas antes de asegurarse de que las funciones básicas de la compilación del software funcionan como deberían.

Pruebas de extremo a extremo (e2e)

- Algunas pruebas deben tener una vista de pájaro de alto nivel de la aplicación. Simulan a un usuario interactuando con la aplicación: navegando a una dirección, leyendo texto, haciendo clic en un enlace o botón, llenando un formulario, moviendo el mouse o escribiendo en el teclado. Estas pruebas generan las expectativas sobre lo que el usuario ve y lee en el navegador.
- Desde la perspectiva del usuario, no importa que la aplicación como esté implementada. Los detalles técnicos como la estructura interna de su código no son relevantes. No hay distinción entre front-end y back-end, entre partes del código. Se prueba la experiencia completa.
- Estas pruebas se denominan pruebas de extremo a extremo (E2E) ya que integran todas las partes de la aplicación desde un extremo (el usuario) hasta el otro extremo (los rincones más oscuros del back-end). Las pruebas de extremo a extremo también forman la parte automatizada de las pruebas de aceptación, ya que indican si la aplicación funciona para el usuario.

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 limitaciones de la automatización de pruebas son:
  - No todas las pruebas manuales se pueden automatizar y no son un sustituto de las pruebas exploratorias.
  - La automatización sólo puede comprobar resultados interpretables por la máquina.
  - La automatización sólo puede comprobar los resultados reales que pueden ser verificados por un oráculo de prueba automatizado.
- Las pruebas exploratorias es un enfoque de prueba manual que enfatiza la libertad y creatividad de la persona que prueba 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.

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.

Cobertura

- La cobertura es la relación entre lo que se puede probar y lo que realmente se prueba con el equipo de prueba.
- La cobertura es un concepto útil que ayuda a los evaluadores con preguntas complejas, como:
  - Dado que es imposible probarlo todo y, por lo tanto, nos vemos obligados a probar solo un subconjunto de todas las posibilidades, ¿cuál es el mejor subconjunto?
  - ¿Cuál es la diferencia entre "pruebas elementales" y "pruebas exhaustivas"? ¿Qué significa concretamente para los casos de prueba que necesitamos para esto?
- La cobertura tiene mucho que ver con el deseo de encontrar la mayor cantidad de defectos posibles con el menor número de casos de prueba posibles.
- En lugar de simplemente probar "cualquier" subconjunto de opciones, nuestro objetivo es compilar un conjunto de casos de prueba que creen el mayor cambio posible para encontrar los defectos que existen.

Tipo e índice de cobertura

- El tipo de cobertura es la forma en que se expresa la cobertura de las situaciones de prueba deducibles de la base de la prueba. Indica qué tipo de posibilidades se distinguen y están involucradas en las situaciones de prueba.
- El índice de cobertura es el porcentaje de situaciones de prueba, según lo definido por el tipo de cobertura, que cubre la prueba. Indica cuántas de estas posibilidades se prueban y generalmente se expresa como porcentaje. Muestra qué parte de todas las alternativas realmente se han probado. El número (porcentaje) solo tiene significado cuando se asocia con el tipo de cobertura.
- No podemos hablar de la cobertura, sino de muchas formas o tipos de cobertura. Los diferentes tipos de cobertura tienden a complementarse en lugar de reemplazarse entre sí, NO PUEDEN compararse en términos de "X es mejor que Y", las pruebas más exhaustivas debe realizarse seleccionando varios tipos de cobertura. La elección de los tipos de cobertura depende de dónde se ubican los mayores riesgos y que tipos se adaptan mejor a dichas zona.
- Los tipos de cobertura se pueden dividir en cuatro grupos de cobertura básicos, las técnicas de diseño de pruebas permiten encontrar los casos de pruebas de un grupo concreto.

Grupos de tipos cobertura

- **Proceso**: Los procesos se pueden identificar en varios niveles. Existen algoritmos de control de flujos, procesos de negocio. Se pueden utilizar tipos de cobertura como caminos, cobertura de declaraciones y transiciones de estado para probar estos procesos.
- **Condiciones**: En casi todos los sistemas existen puntos de decisión que consisten en condiciones en las que el comportamiento del sistema puede ir en diferentes direcciones, dependiendo del resultado de dicho punto de decisión. Las variaciones de estas condiciones y sus resultados se pueden probar utilizando tipos de cobertura como cobertura de decisión, cobertura de decisión/condición modificada y cobertura de condición múltiple.
- **Datos**: Los datos se crean y se eliminan cuando finalizan. En el medio, los datos se utilizan consultándolos o actualizándolos. Se puede probar este ciclo de vida de los datos, pero también combinaciones de datos de entrada, así como los atributos (dominios) de los datos de entrada o de salida. Algunos tipos de cobertura aquí son valores límite, CRUD, flujos de datos y sintaxis.
- **Apariencia**: Cómo opera un sistema, cómo se desempeña, cuál debe ser su apariencia, a menudo se describe en requisitos no funcionales. Dentro de este grupo encontramos tipos de cobertura como perfiles operativos, de carga y presentación.

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 hacer 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 generar datos de métricas de código con que medir la complejidad y el mantenimiento del código administrado.
- Se insiste mucho en que la cobertura de código 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. El criterio de la cobertura de código se basa en la suposición que a mayor cantidad de código ejecutada por las pruebas, menor la probabilidad de que el código presente defectos.

Calidad de las pruebas

- La cobertura del código a menudo se considera la métrica estándar de oro para comprender la calidad de las pruebas, y eso es algo desafortunado.
- Un problema aún más insidioso con la cobertura del código es que, al igual que otras métricas, rápidamente se convierte en un objetivo en sí mismo.
- La cobertura de prueba tradicional (líneas, instrucciones, ramas, etc.) solo mide la proporción del código qué participa en las pruebas. No comprueba si 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 todas 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 la calidad de 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. 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.

Cantidad correcta de pruebas

- Hay un feroz debate que gira en torno a la cantidad correcta de pruebas. Muy pocas pruebas son un problema: las funciones no se especifican correctamente, los errores pasan desapercibidos, ocurren regresiones. Pero demasiadas pruebas consumen tiempo de desarrollo y recursos, no generan ganancias adicionales y ralentizan el desarrollo a largo plazo.
- No es cuestión de hacer muchas pruebas, de hecho, hay que hacer las imprescindibles pero seleccionando buenas pruebas: las que mayor probabilidad tengan de detectar un error y cubran el mayor numero de escenarios.
- Las pruebas difieren en su valor y calidad. Algunas pruebas son más significativas que otras. Los recursos son limitados.
- Esto significa que la calidad de las pruebas es más importante que su cantidad.

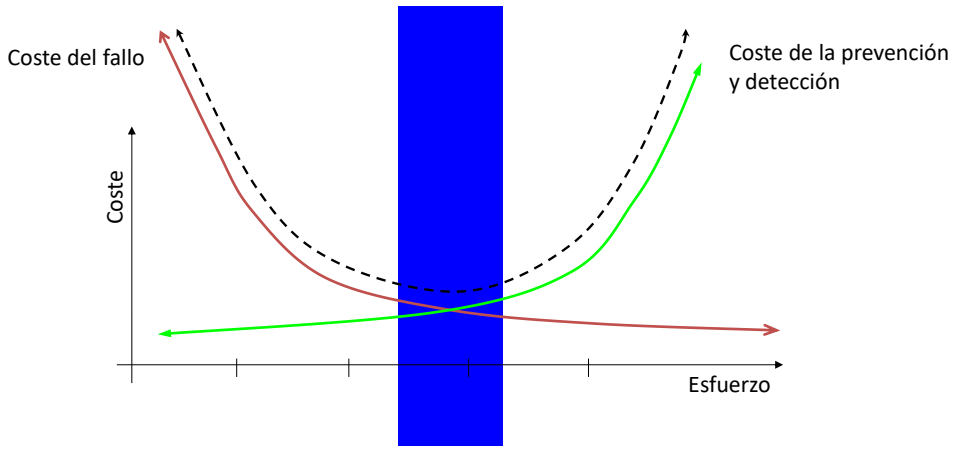
Automatización de pruebas

- Las pruebas exploratorias (manuales) son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización.
- Las pruebas funcionales de usuario final son caras de ejecutar, requieren abrir un navegador e interactuar con el. Además, normalmente requieren que una infraestructura considerable este disponible para estas ejecutarse de manera efectiva. Es una buena regla preguntarse siempre si lo que se quiere probar se puede hacer usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- JavaScript al ser interpretado directamente por el navegador (no requiere compilación) posibilita que hasta los errores sintácticos lleguen a ejecución. Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.

Cantidad correcta de pruebas

- Hay un feroz debate que gira en torno a la cantidad correcta de pruebas. Muy pocas pruebas son un problema: las funciones no se especifican correctamente, los errores pasan desapercibidos, ocurren regresiones. Pero demasiadas pruebas consumen tiempo de desarrollo y recursos, no generan ganancias adicionales y ralentizan el desarrollo a largo plazo.
- No es cuestión de hacer muchas pruebas, de hecho, hay que hacer las imprescindibles pero seleccionando buenas pruebas: las que mayor probabilidad tengan de detectar un error y cubran el mayor numero de escenarios.
- Las pruebas difieren en su valor y calidad. Algunas pruebas son más significativas que otras. Los recursos son limitados.
- Esto significa que la calidad de las pruebas es más importante que su cantidad.

Las pruebas tienen un coste



Pirámide de pruebas

https://martinfowler.com/bliki/TestPyramid.html

Comparativa

Nivel	Extremo a extremo	Integración	Unitarias
Cobertura	completa	grande	pequeña
Rendimiento	lenta	rápida	muy rápida
Fiabilidad	menos confiable	confiable	mas fiable
Aislar fallos	complicado	justo	fácil
Coste	muy alto	mediano	muy bajo
Simula el usuario real	sí	no	no

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 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.

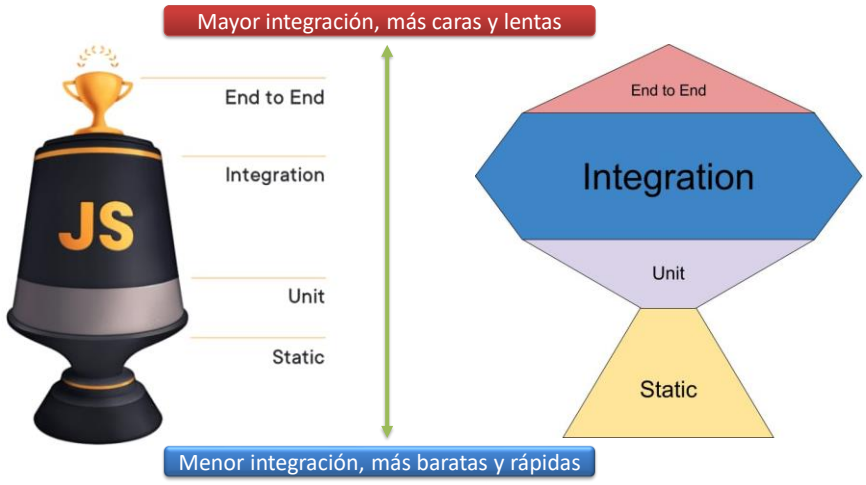
Ejecución del análisis estático

- 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.

El Trofeo de Pruebas

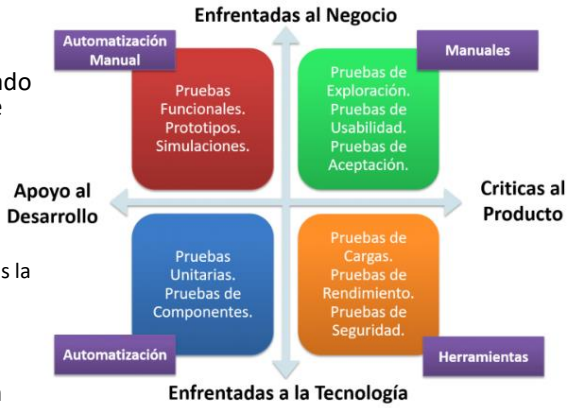
- Testing Trophy es un método de pruebas propuesto por Kent C. Dodds *para aplicaciones web*. Se trata de escribir suficientes pruebas, no muchas, pero si las pruebas correctas: proporciona la mejor combinación de velocidad, costo y confiabilidad.
- Se superpondrán las siguientes técnicas:
  - Usar un sistema de captura de errores de tipo, estilo y de formato utilizando linters, formateadores de errores y verificadores de tipo (ESLint, SonarQube, …).
  - Escribir pruebas unitarias efectivas que apunten solo al comportamiento crítico y la funcionalidad de la aplicación.
  - Desarrollar pruebas de integración para auditar la aplicación de manera integral y asegurarse de que todo funcione correctamente en armonía.
  - Crear pruebas funcionales de extremo a extremo (e2e) para pruebas de interacción automatizadas de las rutas críticas y los flujos de trabajo más utilizados por los usuarios.

Testing Trophy



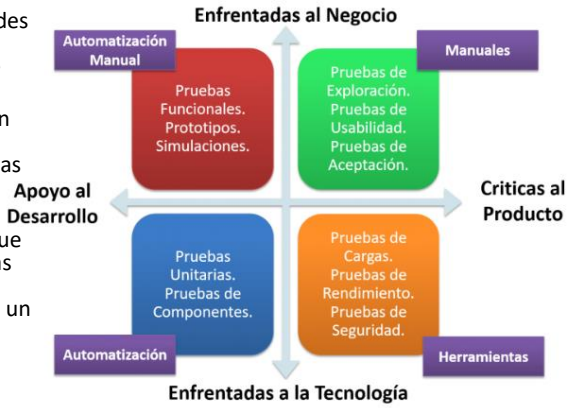
Cuadrante de Pruebas

- El cuadrante de prueba divide todo el proceso de prueba en 4 partes. Esto hace que el proceso de prueba sea fácil de entender.
- El cuadrante de pruebas fue originalmente publicado por Brian Marick en 2003 como parte de una serie de artículos sobre Testeo Ágil y originalmente fue conocido como “Matriz de pruebas de Marick” (Marick Test Matrix”).
- Hay cuatro cuadrantes de prueba que son el resultado de dos ejes.
  - El eje vertical muestra la perspectiva tecnológica versus la perspectiva empresarial.
  - El eje horizontal trata sobre la orientación durante la creación de un producto versus la evaluación del producto una vez que está listo.
- Hay que asegurarse de que se organicen y realicen pruebas para los cuatro cuadrantes.



Cuadrante de Pruebas

- El primer cuadrante (**abajo a la izquierda**) es donde la atención se centra en la tecnología y donde las actividades de prueba guían al equipo en la creación del producto. Suele implicar TDD para las unidades y la integración de las unidades. Las prueba deberían automatizarse.
- El segundo cuadrante (**arriba a la izquierda**) se centra en el negocio y también se centra en guiar al equipo. Esto implica, por ejemplo, el desarrollo impulsado por pruebas de aceptación (ATDD), donde se prueba un proceso de negocio, preferiblemente automatizadas.
- El tercer cuadrante (**arriba a la derecha**) tiene un enfoque empresarial en la evaluación del producto. Estas pruebas se ejecutan principalmente de forma manual, ya sea siguiendo escenarios de prueba preparados o mediante un enfoque exploratorio.
- El cuarto cuadrante (**abajo a la derecha**) se centra en la tecnología y evalúa los aspectos no funcionales de un producto que sólo pueden validarse una vez que el producto esté listo. Las pruebas de rendimiento dinámico son un buen ejemplo de ello.



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.

Terminología

- 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. La **base de prueba** es el punto de partida del caso de prueba (requisito, criterio de aceptación, …) y el **objeto de prueba** es el destinatario de la prueba (componente, sistema, documento, …).
- Un **procedimiento de prueba** (o caso de prueba lógico) especifica cómo realizar uno o varios casos de prueba. El procedimiento documenta los pasos que deben darse para cada uno de los casos de prueba. Los procedimientos de prueba pueden reutilizarse para varios casos de prueba similares. Así mismo, un caso de prueba puede estar incluido en varios procedimientos de prueba.
- Un **componente de prueba** (o caso de prueba físico) automatiza uno o varios procedimientos de prueba o partes de éstos. Los componentes de prueba se diseñan e implementan de forma específica para proporcionar las entradas, controlar la ejecución e informar de la salida de los elementos a probar.
- Una **suite de pruebas** es un conjunto organizado de pruebas que se ejecutan de manera automatizada para verificar un aspecto la funcionalidad y calidad de un sistema, aplicación o componente.
- Un **ejecutor de pruebas** (o test runner) es la herramienta encargada de descubrir y ejecutar los componentes de pruebas y genera un informe para exportar e integrar el resultado de las prueba.

Características de una buena prueba unitaria

- 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

Las pruebas deben

- Tener un objetivo único y un propósito claro.
- Centrarse en el comportamiento antes que en los detalles de la implementación.
- Empezar por los casos válidos (Happy Path) antes de pasar a los casos inválidos (extremos, limites).
- Tener nombres descriptivos y un código limpio, son parte de la documentación.
- Seguir el patrón AAA o una de sus variaciones.
- Ser breves, simples y eficientes.
- Ser deterministas y repetibles.
- Ser independientes entre si (inicializar el entorno), aisladas (dobles de pruebas), no tener efectos secundarios, no influir en el observado.

Casos de prueba de mala calidad

- Sin aserciones
- No comprueban el resultado completo esperado
- No utilizan las aserciones mas especificas
- Cubren múltiples escenarios
- Complejos, con mucho código, no dejan claro que están probando y son frágiles
- Su código apesta, difíciles de entender y mantener
- Dependen de otros casos de pruebas
- Indeterministas, unas veces fallan y otras no, sin cambiar el código fuente ni la prueba
- Cruzan limites, no respetan los limites de las pruebas

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
- 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.

Características de las pruebas valiosa

- **Las pruebas formalizan y documentan los requisitos.**
  - Un conjunto de pruebas es una descripción formal, legible por humanos y máquinas, de cómo debe comportarse el código. Ayuda a los desarrolladores, en la creación, a comprender los requisitos que deben implementar. Ayuda a los desarrolladores, en el mantenimiento, a comprender los desafíos a que tuvieron que enfrentase los creadores.
  - *Una prueba valiosa describe claramente cómo debe comportarse el código de implementación.* La prueba utiliza un lenguaje adecuado para hablar con los desarrolladores y transmitir los requisitos. La prueba enumera los casos conocidos con los que tiene que lidiar la implementación.
- **Las pruebas aseguran que el código implemente los requisitos y no muestre fallos.**
  - Las pruebas aprovechan cada parte del código para encontrar fallos.
  - *Una prueba valiosa cubre los escenarios importantes*: tanto las entradas correctas como las incorrectas, los casos esperados y los casos excepcionales.

Características de las pruebas valiosa

- **Las pruebas ahorran tiempo y dinero.**
  - Las pruebas intentan cortar los problemas de software de raíz. Las pruebas previenen errores antes de que causen un daño real, cuando todavía son manejables y están bajo control.
  - *Una prueba valiosa es rentable.* La prueba previene errores que, en última instancia, podrían inutilizar la aplicación. La prueba es barata de escribir en comparación con el daño potencial que previene.

- **Las pruebas hacen que el cambio sea seguro al evitar las regresiones.**
  - Las pruebas no solo verifican que la implementación actual cumpla con los requisitos. También verifican que el código aún funcione como se esperaba después de los cambios. Con las pruebas automatizadas adecuadas, es menos probable que se rompa accidentalmente. La implementación de nuevas funciones y la refactorización de código es más segura.
  - *Una prueba valiosa falla cuando se cambia o elimina el código esencial.* Las prueba se diseñan para fallar si se cambia el comportamiento dependiente y deberían seguir pasando ante cambios no dependientes.

Automatización de pruebas

- Las pruebas exploratorias (manuales) son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización.
- Las pruebas funcionales de usuario final son caras de ejecutar que, por ejemplo, requieren abrir un navegador e interactuar con el. Además, normalmente requieren que una infraestructura considerable este disponible para estas ejecutarse de manera efectiva. Es una buena regla preguntarse siempre si lo que se quiere probar se puede hacer usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- Los analizadores de código son herramientas de fuerza bruta que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.

QUALITY ASSURANCE: DISEÑAR PARA PROBAR

Aseguramiento de la calidad

- El aseguramiento de la calidad son todas aquellas actividades y los procesos que se realizan para asegurar que los productos y servicios de un proyecto posean el nivel de calidad requerido, está orientado al proceso y se centra en el desarrollo del producto o servicio.
- El aseguramiento de la calidad del software (SQA) es un conjunto de prácticas, metodologías y actividades adoptadas en el proceso de desarrollo con el objetivo de garantizar que el software producido cumpla con altos estándares de calidad. El QA en el contexto del desarrollo de software es esencial para reducir defectos, mejorar la confiabilidad del software, garantizar la seguridad y satisfacer al cliente. Sus principales características son:
  - Preventivo: es un enfoque preventivo para la gestión de la calidad. Se centra en la definición de procesos, estándares y procedimientos para prevenir defectos y problemas de calidad desde el inicio.
  - Actividades de planificación: implica la elaboración de planes de calidad que establecen objetivos, estrategias, recursos y procesos para el control de calidad durante todo el ciclo de vida del proyecto.
  - Orientado a procesos: el enfoque principal está en la calidad de los procesos utilizados para desarrollar el software, se centra en asegurar que los procesos estén bien definidos, bien gestionados y que se sigan.
  - Prevención de defectos: el objetivo principal es prevenir defectos antes de que se produzcan y mejorar constantemente los procesos para minimizar la probabilidad de errores en el software.
  - Involucración continua: está involucrado en todo el ciclo de vida del proyecto, desde la planificación hasta la entrega, para garantizar que se cumplan los estándares de calidad en cada fase.

Diseñar para probar

- 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 y tenerlas en cuenta en las etapas de análisis y diseño con facilitadores de las pruebas.
- Fundamentos de diseño
  - Programar para las interfaces, no para la herencia.
  - Favorecer la composición antes que la herencia.
- Patrones:
  - Delegación
  - Doble herencia
  - Inversión de Control e Inyección de Dependencias
  - Segregación IU/Código: Modelo Vista Controlador (MVC), Model View ViewModel (MVVM)
- Metodologías (Test-first development):
  - Desarrollo Guiado por Pruebas (TDD)
  - Desarrollo Dirigido por Comportamiento (BDD)
  - Desarrollo Dirigido por Tests de Aceptación (ATDD)

Simulación de objetos

- Las dependencias introducen ruido en las pruebas: ¿el fallo se está produciendo en el objeto de la prueba o en una de sus dependencias?
- 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) o
  - 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 probada 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

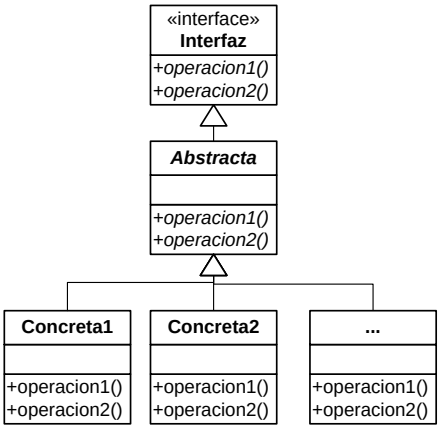
- **Fixture**: Es el término se utiliza para hablar de los datos de contexto de las pruebas, aquellos que se necesitan para construir el escenario que requiere la prueba.
- **Dummy**: Objeto que se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar 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.

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.
- 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:
  - Organizar clases que tienen un comportamiento parecido para que sean intercambiables y consistentes (polimorfismo).
  - Mantener las clases que implementan el interfaz como internas del proyecto (internal o Friend), pero la interfaz pública.
- Solución:
  - Crear un interfaz paralelo a la clase base
  - Clase base es abstracta y sus herederos implementaciones concretas.
  - La clase base puede heredar de mas de una interfaz.
  - Cualquier clase puede implementar el interfaz independientemente de su jerarquía.



Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.
- En lugar de crear una dependencia en sí misma, una parte de la aplicación simplemente declara la dependencia. La tediosa tarea de crear y proporcionar la dependencia se delega a un inyector que se encuentra en la parte superior.
- Esta división del trabajo desacopla una parte de la aplicación de sus dependencias: una parte que no necesita saber cómo configurar una dependencia, y mucho menos las dependencias de la dependencia, etc.

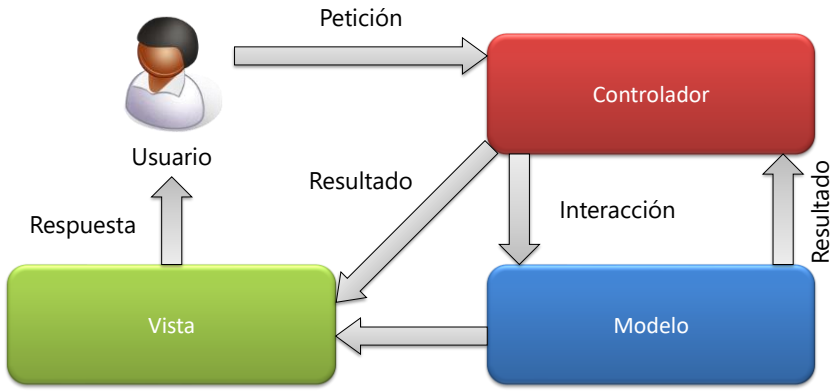
Inversión de Control

- 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.

El patrón MVC



El patrón MVC

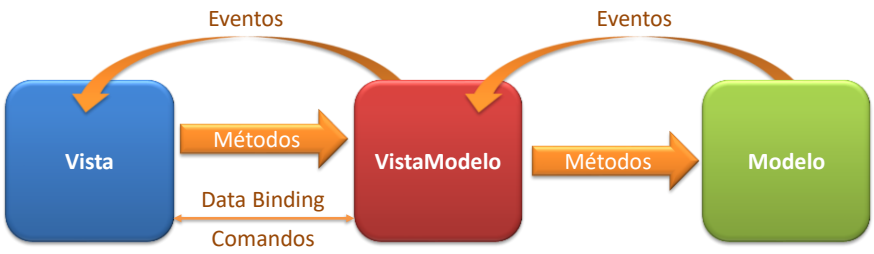
- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**

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

- **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.



Test-First Development (TFD)

- La prueba gana peso hasta convertirse en "ciudadano de primera clase", es un enfoque que consiste empezar por la prueba para una característica determinada, para luego diseñarla y desarrollarla en un solo paso.
- Al escribir la prueba primero, estas pueden centrarse en los resultados esperados en lugar de en los detalles de implementación, facilitan el diseño de una solución clara y eficaz y, una vez que se completa el desarrollo, se ejecutan para ver si pasan o no.
- Como evolución del Test-first development (XP) las principales metodologías son:
  - Desarrollo Guiado por Pruebas (TDD)
  - Desarrollo Dirigido por Comportamiento (BDD)
  - Desarrollo Dirigido por Tests de Aceptación (ATDD)

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 y refactorizar después de implementar 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 3 leyes:
  - No escribirás código de producción sin antes escribir un test que falle.
  - No escribirás más de un test unitario suficiente para fallar (y no compilar es fallar).
  - No escribirás más código del necesario para hacer pasar el test.

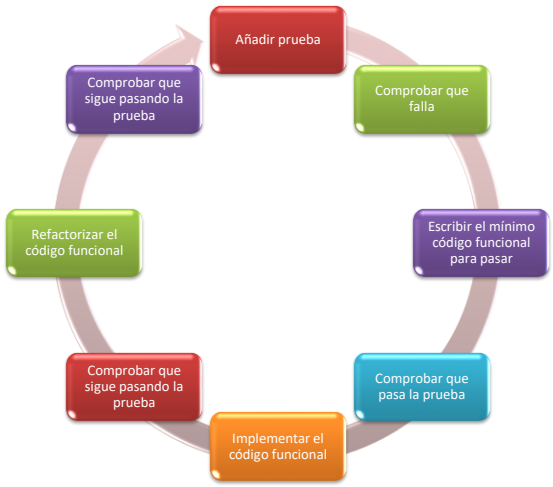
Escribir la prueba

- La escritura de las pruebas sigue varios patrones identificados y detallados por Beck. En particular se destaca que las pruebas:
  - Son escritas por el propio desarrollador en el mismo lenguaje de programación que la aplicación, y su ejecución se automatiza. Esto último es primordial para que obtenga un retorno inmediato, indicándose que el ritmo de cambio entre prueba-código-prueba esta pautado por intervalos de no más de diez minutos. La automatización también permite aplicar, en todo momento, la batería de pruebas, implicando pruebas de regresión inmediatas y continuas.
  - Las pruebas se deben escribir pensando en primer lugar en probar las operaciones que se deben implementar.
  - Deben ser aisladas, de forma que puedan correrse independientemente de otras, no importando el orden. Este aislamiento determinará que las soluciones de código cohesivas y bajo acoplamiento, con componentes ortogonales.
  - Deben de escribirse antes que el código que se desea probar.

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. Escoger un requisito.
  2. Escribir una prueba que demuestre la necesidad de escribir código (assert first).
  3. Escribir el mínimo código para que el código de pruebas compile (probar la prueba)
  4. Fijar la prueba
  5. Implementar exclusivamente la funcionalidad demandada por las pruebas
  6. Ejecutar todas la prueba.
  7. Mejorar el código (refactorización) sin añadir funcionalidad
  8. Actualizar la lista de requisitos (añadir los nuevos requisitos que hayan aparecido)
  9. 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 probado automáticamente.

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.

Estrategia RED – GREEN – REFACTOR

- 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 fuente 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.

Beneficios de TDD

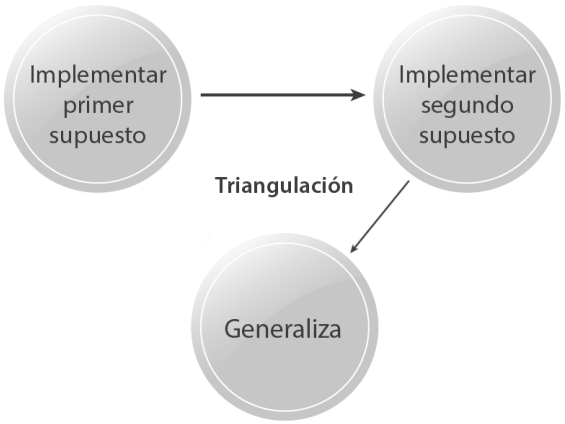
- Facilita desarrollar ciñéndose a los requisitos.
  - Ayuda a encontrar inconsistencias en los requisitos
  - Reduce el número de errores y bugs ya que éstos se detectan antes incluso de crearlos.
- Facilita el mantenimiento del código:
  - Protege ante cambios, los errores que surgen al aplicar un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

Inside-Out vs Outside-In

- ¿La mejor manera de empezar es por los detalles de lo que está construyendo, y dejar que la arquitectura vaya emergiendo con un enfoque de adentro hacia afuera (Inside Out)? ¿O bien, empezar por algo grande y dejar que los detalles vayan revelando a medida que se van usando desde afuera hacia adentro (Outside In)?.
- La escuela clásica (Inside Out/Bottom Up/Chicago School/Aproximación Clásica) toma su nombre por representar el concepto original definido en libros como «Test-driven Development By Example» de Kent Beck, y que se distingue por enfatizar la práctica de la triangulación (Inside Out).
- La escuela de Londres (Outside In TDD/Top Down/Mockist Approach) toma su nombre debido a que tiene su base, principalmente, en el libro «Growing Object Oriented Software Guide By Test» de Stephen Freeman y Nat Pryce, enfatizando los roles, responsabilidades e interacciones (Outside In).

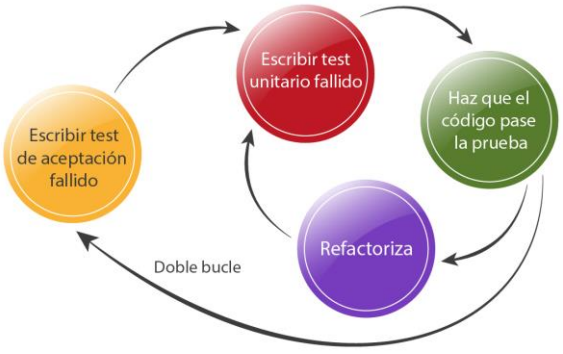
Técnica de la triangulación

- Una vez que tenemos el test fallando, tenemos tres estrategias posibles para pasar a verde:
  - Fake It: crear un fake que devuelva una constante.
  - Obvious Implementation: implementamos la solución más simple posible si es obvia.
  - Triangulation
- Triangular, o la técnica de la triangulación, es el paso natural que sigue a la técnica de la implementación falsa:
  - Escoger el caso más simple que debe resolver el algoritmo.
  - Aplicar Red-Green-Refactor.
  - Repetir los pasos anteriores cubriendo las diferentes casuísticas.
- El concepto se basa en obtener dos o más casos similares antes de proceder con la implementación de una solución genérica.



Técnica del doble bucle

- La escuela de Londres toma un enfoque distinto, centrándose en verificar que el comportamiento de los objetos es el esperado. Dado que este es el objetivo final, verificar las correctas interacciones entre objetos, y no el estado en sí mismo de los objetos, ahorrándonos todo el trabajo con los objetos reales (creación y mantenimiento) sustituyéndolos por dobles de prueba. Podremos empezar por la funcionalidad final que se necesita, implementando poco a poco toda la estructura que dé soporte a dicha funcionalidad, Outside-in, desde el exterior hacia dentro.
- Mediante el uso de la técnica del doble bucle:
  - Comenzaremos por un test de aceptación que falle, lo que nos guiará al bucle interno.
  - En el bucle interno, que representa la metodología de trabajo TDD («Red-Green-Refactor»), implementaremos la lógica de nuestra solución.
  - Realizaremos las iteraciones necesarias de este bucle interno hasta conseguir pasar el test de aceptación.



Aproximaciones

Escuela Clásica

La aproximación de TDD De adentro hacia afuera (Inside Out) favorece a los desarrolladores a los que les gusta ir construyendo pieza a pieza. Puede ser más sencillo para empezar, siguiendo un enfoque metódico para identificar las entidades, y trabajando duro para llevar a cabo el comportamiento interno.

- Cuando se quiera verificar el estado de los objetos.
- Cuando las colaboraciones entre objetos son sencillas.
- Si no se quiere acoplar la implementación a las pruebas.
- Si se prefiere no pensar en la implementación mientras se escriben las pruebas.

Escuela de Londres

La aproximación de TDD De fuera hacia Adentro (Outside In) el desarrollador empieza a construir todo el sistema, y lo descompone en componentes más pequeños cuando se presentan oportunidades de refactorización. Esta ruta es más exploratorio, y es ideal en las situaciones donde hay una idea general del objetivo, pero los detalles finales de la implementación están menos claros.

- Cuando se quiera verificar el comportamiento de los objetos.
- Cuando las colaboraciones entre objetos sean complejas.
- Si no importa acoplar la implementación a las pruebas.
- Si se prefiere pensar en la implementación mientras se escriben las pruebas.

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.

Ejemplo

```
# language: es
Característica: Suma de dos números
  Como matemático novato
  Yo quiero obtener la suma de dos cifras
  Para aprender a sumar

  Escenario: Sumar dos números positivos
  Dado que estoy en la aplicación
  Cuando pongo los números 1 y 3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser 4

  Escenario: Sumar dos números negativos
  Dado que estoy en la aplicación
  Cuando pongo los números -1 y -3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser -4

  Escenario: Sumar un numero positivo y uno negativo
  Dado que estoy en la aplicación
  Cuando pongo los números -2 y 3
  Y solicito el resultado del cálculo
  Entonces el resultado debe ser 1
```

```
public class CalculadoraStepDefinitions {
    Calculadora calc;
    int op1, op2, rslt;

    @Dado("que estoy en la aplicación")
    public void que_estoy_en_la_aplicación() {
        calc = new Calculadora();
    }
    @Cuando("pongo los números {int} y {int}")
    public void pongo_los_números_y(Integer a, Integer b) {
        op1 = a;
        op2 = b;
    }
    @Cuando("solicito el resultado del cálculo")
    public void solicito_el_resultado_del_cálculo() {
        rslt = calc.suma(op1, op2);
    }
    @Entonces("el resultado debe ser {int}")
    public void el_resultado_debe_ser(Integer r) {
        assertEquals(r, rslt);
    }
}
```


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.

Keyword Driven Development (KDD)

- Keyword Driven Development o KDD es un proceso de desarrollo software basado en KDT, que se impulsa por la palabra clave almacenada en una base de datos y en la automatización a la que hace referencia.
- Keyword Driven Testing o KDT es un framework de automatización de pruebas, donde se utiliza un formato de tabla para definir palabras claves o palabras de acción para cada función que nos gustaría utilizar. La ISO/IEC/IEEE 29119-5 define KDT como una forma de describir los casos de prueba mediante el uso de un conjunto predefinido de palabras clave, en un lenguaje de uso común. Estas palabras clave son nombres que están asociados con un conjunto de acciones que se requieren para realizar los pasos específico en un caso de prueba. Mediante el uso de palabras clave para describir las acciones de prueba en lugar de lenguaje natural, los casos de prueba pueden ser más fáciles de entender, mantener y automatizar.
- La definición de las keywords se produce de una forma independiente a las acciones asociadas, lo que provoca la separación del diseño de la prueba respecto a su ejecución. La reusabilidad de las keywords constituye otra de las potencialidades.

Componentes de Keyword Driven Testing

- Almacén de palabras clave (Keywords): Las palabras que creamos las guardaremos en un determinado formato para que posteriormente sean accedidas por los scripts.
- Almacén de datos: Contiene los datos necesarios para las pruebas y se almacenan en un determinado formato similar al seleccionado para las palabras clave.
- Repositorio de objetos: Repositorio de las referencias necesarias a los objetos UI sobre los que se actuará. Pueden estar referenciadas por id, XPath, etc. En este punto es importante señalar que durante la creación del frontal de la aplicación es conveniente que los desarrolladores codifiquen acorde a los estándares e identifiquen los objetos UI de forma adecuada.
- Biblioteca de funciones: Contiene la codificación de los flujos de negocio para los distintos elementos UI.
- Scripts de pruebas: Son el conjunto de funciones que se llaman desde la biblioteca de funciones para ejecutar el código asociado a una palabra clave.

AUTOMATIZACIÓN DE PRUEBAS

Automatización de la Prueba

- La automatización de la prueba permite ejecutar muchos casos de prueba de forma consistente y repetida en las diferentes versiones del sistema sujeto a prueba (SSP) y/o entornos. Pero la automatización de pruebas es más que un mecanismo para ejecutar un juego de pruebas sin interacción humana. Implica un proceso de diseño de productos de prueba, entre los que se incluyen:
  - Software.
  - Documentación.
  - Casos de prueba.
  - Entornos de prueba
  - Datos de prueba
- Una Solución de Automatización Pruebas (SAP) debe permitir:
  - Implementar casos de prueba automatizados.
  - Monitorizar y controlar la ejecución de pruebas automatizadas.
  - Interpretar, informar y registrar los resultados de pruebas automatizadas.

Objetivos de la automatización de pruebas

- Mejorar la eficiencia de la prueba.
- Aportar una cobertura de funciones más amplia.
- Reducir el coste total de la prueba.
- Realizar pruebas que los probadores manuales no pueden.
- Acortar el período de ejecución de la prueba.
- Aumentar la frecuencia de la prueba y reducir el tiempo necesario para los ciclos de prueba.

Ventajas y Desventajas de la automatización

Ventajas

- Se pueden realizar más pruebas por compilación.
- La posibilidad de crear pruebas que no se pueden realizar manualmente (pruebas en tiempo real, remotas, en paralelo).
- Las pruebas pueden ser más complejas.
- Las pruebas se ejecutan más rápido.
- Las pruebas están menos sujetas a errores del operador.
- Uso más eficaz y eficiente de los recursos de pruebas
- Información de retorno más rápida sobre la calidad del software.
- Mejora de la fiabilidad del sistema (por ejemplo, repetibilidad y consistencia).
- Mejora de la consistencia de las pruebas.

Desventajas

- Requiere tecnologías adicionales.
- Existencia de costes adicionales.
- Inversión inicial para el establecimiento de la SAP.
- Requiere un mantenimiento continuo de la SAP.
- El equipo necesita tener competencia en desarrollo y automatización.
- Puede distraer la atención respecto a los objetivos de la prueba, por ejemplo, centrándose en la automatización de casos de prueba a expensas del objetivo de las pruebas.
- Las pruebas pueden volverse más complejas.
- La automatización puede introducir errores adicionales.

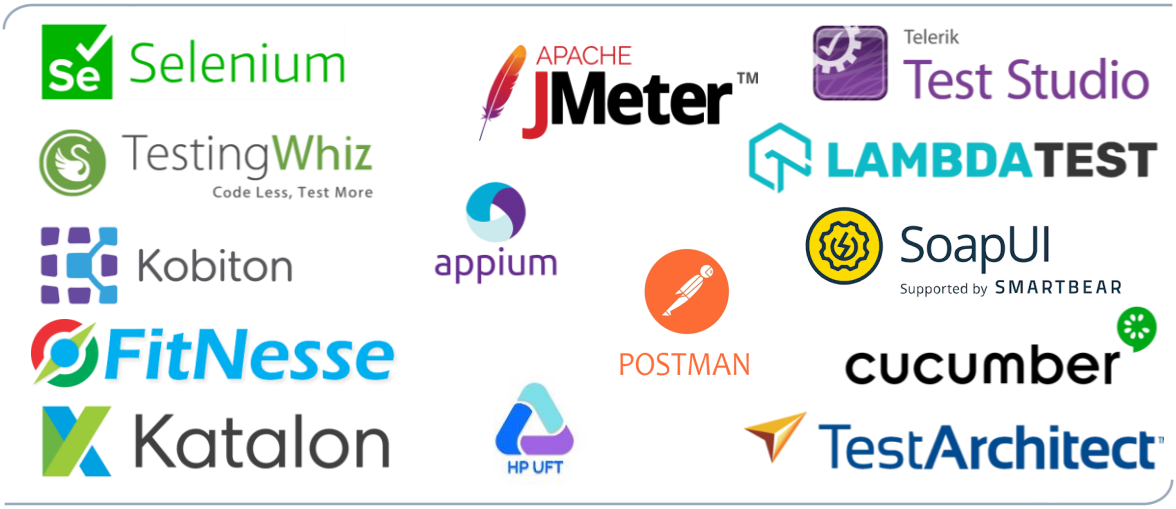
Limitaciones de la automatización de pruebas

- No todas las pruebas manuales se pueden automatizar.
- La automatización sólo puede comprobar resultados predecibles e interpretables por la máquina.
- La automatización sólo puede comprobar los resultados reales que pueden ser verificados por un oráculo de prueba automatizado.
- No es un sustituto de las pruebas exploratorias.

Técnicas para la automatización de pruebas

- Las herramientas para la automatización de pruebas funcionales aplican técnicas diferentes para la creación y ejecución de pruebas:
  - Técnica de **Programación con Frameworks**: Los marcos de pruebas unitarias (unitarias, integración, sistema) simplifican la creación y ejecución de los procedimientos de pruebas en el mismo lenguaje que el objeto de la prueba. La ejecución y cobertura de las pruebas se pueden integrar en los entornos de desarrollo.
    - Unit Test Frameworks: JUnit, MSTest, NUnit, xUnit, Jest, PHPUnit, …
    - Mocking Frameworks: Mockito, Microsoft Fakes, Moq, NSubstitute, Rhino Mocks, …
    - Code Coverage: Jacoco, Clover, NCover, RCov, …
  - Técnica de **registro/reproducción** (Record & Playback): Esta técnica consiste en grabar una ejecución de la prueba realizada en la interfaz de la aplicación y su reproducción posterior.
  - Técnica de **Programación de Scripts Estructurados**: Difiere de la técnica de registro/reproducción en la introducción de una librería de scripts reutilizables, que realizan secuencias de instrucciones que se requieren comúnmente en una serie de pruebas. Los script de pruebas se crean en un lenguaje de scripting común.
  - Técnica de **Data-Driven**: Esta es una técnica que se enfoca en la separación de los datos de prueba de los scripts, y los almacena en archivos separados. Por lo tanto, los scripts sólo contendrá los procedimientos de prueba y las acciones para la aplicación.
  - Técnica de **Keyword-Driven**: Se basa en la recuperación de los procedimientos de prueba desde los scripts, quedando sólo los datos de prueba y acciones específicas de prueba, que se identifican por palabras clave.

Herramientas de automatización de pruebas



Entornos de pruebas

- Un enfoque de varios entornos permite compilar, probar y liberar código con mayor velocidad y frecuencia para que la implementación sea lo más sencilla posible. Permite quitar la sobrecarga manual y el riesgo de una versión manual y, en su lugar, automatizar el desarrollo con un proceso de varias fases destinado a diferentes entorno.
  - Desarrollo: es donde se desarrollan los cambios en el software.
  - Prueba: permite que los evaluadores humanos o las pruebas automatizadas prueben el código nuevo y actualizado. Los desarrolladores deben aceptar código y configuraciones nuevos mediante la realización de pruebas unitarias en el entorno de desarrollo antes de permitir que esos elementos entren en uno o varios entornos de prueba.
  - Ensayo/preproducción: donde se realizan pruebas finales inmediatamente antes de la implementación en producción, debe reflejar un entorno de producción real con la mayor precisión posible.
  - UAT: Las pruebas de aceptación de usuario (UAT) permiten a los usuarios finales o a los clientes realizar pruebas para comprobar o aceptar el sistema de software antes de que una aplicación de software pueda pasar a su entorno de producción.
  - Producción: es el entorno con el que interactúan directamente los usuarios.

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.

Maven

```
<properties>
	<maven.compiler.release>17</maven.compiler.release>
	<junit.jupiter.version>5.10.2</junit.jupiter.version>
	<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
	<dependency>
		<groupId>org.junit.jupiter</groupId>
		<artifactId>junit-jupiter</artifactId>
		<version>${junit.jupiter.version}</version>
		<scope>test</scope>
	</dependency>
</dependencies>
```


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 casos 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 como 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

- Las pruebas son una parte importante parte de la documentación: los casos de uso. 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 y transformar los nombres de los métodos a mostrar.

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

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, escenarios de entrada que permiten prever las salidas, que se crearán antes de ejecutar cada prueba.
- Se pueden crear en métodos de la instancia y ser invocados directamente por cada caso de prueba o delegar su invocación en el ciclo de vida de la pruebas.

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.
- 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". Deben ser métodos de clase (static).

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 por una asunción.
  - Disabled (Skipped): No se ha ejecutado la prueba.
  - Error: Excepción en el código 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 y, 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);
}
```


@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 y @CsvFileSource

- 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 static y deben 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 al 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) {
```


Fuentes nulas y vacías

- Con el fin de comprobar los casos 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 al @ParameterizedTest un único argumento null, 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) {
```

Agregación de argumentos

- De forma predeterminada, cada argumento proporcionado a un método @ParameterizedTest corresponde a un único parámetro de método. En consecuencia, las fuentes de argumentos que se espera que proporcionen una gran cantidad de argumentos pueden generar firmas de métodos grandes.
- En tales casos, se puede utilizar un ArgumentsAccessor en lugar de varios parámetros. Con esta API, se puede acceder a los argumentos proporcionados a través de un único argumento pasado al método de prueba. Además, se puede recuperar el índice de la invocación de prueba actual con ArgumentsAccessor.getInvocationIndex().

```
@ParameterizedTest
@CsvSource({ "1,uno", "2,dos", "3,tres" })
void testArgumentsAccessor(ArgumentsAccessor args) {
    Elemento<Integer, String> ele = new Elemento<Integer, String>(args.getInteger(0),
            args.getString(1));
```


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;
}
```

Asunciones

- Como alternativa a las anotaciones @EnabledOn y @DisabledOn, la clase org.junit.jupiter.api.Assumptions ofrece una colección de métodos de utilidad que permiten la ejecución de pruebas condicionales basadas en suposiciones.
- En contraste directo con afirmaciones fallidas, las asunciones ignoran la prueba (skipped) cuando no se cumplen.
- Las asunciones se usan generalmente cuando no tiene sentido continuar la ejecución de un método de prueba dado que 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 las pruebas.

```
@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

Desactivar pruebas y ejecución condicional

- 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 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 en función del valor de una variable de entorno.

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

Condiciones personalizadas

- Se puede habilitar o deshabilitar un contenedor o prueba según un método de condición configurado mediante las anotaciones @EnabledIfy @DisabledIf. El método de condición debe tener un tipo de retorno boolean, sin parámetros o un parámetros de tipo ExtensionContextar.

```
boolean customCondition() {
   return true;
}
@Test
@EnabledIf("customCondition")
void enabled() {
   // ...
}
@Test
@DisabledIf("customCondition")
void disabled() {
   // ...
}
```

Orden de ejecución de prueba

- Cada caso de prueba debe ser independiente del resto de los casos, por lo que deben poderse ejecutar en cualquier orden (desordenados). 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.DisplayName, MethodOrderer.MethodName, ~~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>
```

https://jar-download.com/artifacts/org.mockito

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 solo suplanta con los valores literales definidos y 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 mayor flexibilidad que los valores literales, 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 suplantados 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 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 notifica 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) { }
```

BDDMockito

- El término BDD fue acuñado por primera vez por Dan North, en 2006. BDD fomenta la escritura de pruebas en un lenguaje natural legible por humanos que se centra en el comportamiento de la aplicación.
- Define una forma claramente estructurada de escribir pruebas siguiendo tres secciones (similares Preparar, Actuar, Afirmar):
  - Given: dadas algunas condiciones previas (Arrange)
  - When: cuando ocurre una acción (Act)
  - Then: luego verifique la salida (Assert)
- La biblioteca Mockito ahora trae una clase BDDMockito que presenta un API compatible con BDD. Esta API permite adoptar un enfoque más amigable con BDD preparando las pruebas con given() y haciendo afirmaciones usando then().
- Las pruebas se vuelven más legibles con la importación estática:

  import static org.mockito.BDDMockito.*;

BDDMockito

- BDDMockito proporciona alias BDD para varios métodos de Mockito: given(), en lugar when(), y then(), en lugar de verify().

```
given(phoneBookRepository.contains(momContactName)) // fijo
  .willReturn(false);
given(phoneBookRepository.getPhoneNumberByContactName(momContactName)) // dinámico
  .will((InvocationOnMock invocation) ->
      invocation.getArgument(0).equals(momContactName) ? momPhoneNumber : null);
willThrow(new RuntimeException()) // excepción
  .given(phoneBookRepository)
  .insert(any(String.class), eq(tooLongPhoneNumber));

try {
    phoneBookService.register(xContactName, tooLongPhoneNumber);
    fail("Should throw exception");
} catch (RuntimeException ex) { }

then(phoneBookRepository)
  .should(never())
  .insert(momContactName, tooLongPhoneNumber);
```

- Se debe intercambiar give y will* para los métodos que no tiene valor de retorno.

Servidores simulados (Mock Servers)

- Los retrasos en el front-end o back-end dificultan que los equipos dependientes completen su trabajo de manera eficiente. Los servidores simulados pueden aliviar esos retrasos en el proceso de desarrollo.
- Esto permite implementar y probar clientes mucho más rápido que si se hubiera tenido que esperar a que se desarrollara la solución real.
- Antes de enviar una solicitud real, se puede crear un mock server para simular cada punto final y su respuesta correspondiente, esto permite obtener un resultado rápido y determinista, incluso off-line.
- Las herramientas de pruebas e2e permiten la creación de mock server:
  - SoapUI (https://www.soapui.org)
  - Postman (https://www.getpostman.com/)

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 filas, 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.

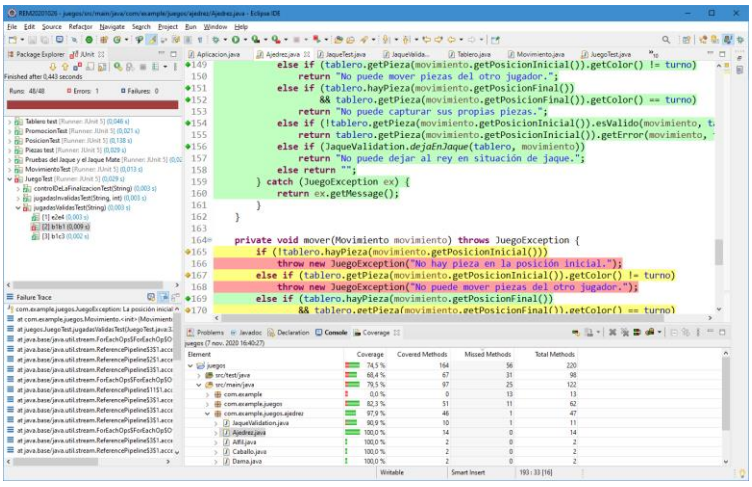
Cobertura de código

- La cobertura de código, también conocida como cobertura de pruebas, mide la proporción del código fuente validada por las pruebas automatizadas.
- Las herramientas de cobertura de código utilizan una serie de criterios para medir la cobertura, como el número de líneas de código, métodos o funciones, ramas y condiciones. Se puede utilizar una herramienta de cobertura de código para identificar las partes de la base de código que no están cubiertas actualmente por las pruebas automatizadas.
- La supervisión de las métricas de cobertura del código ayuda a garantizar que se mantiene un nivel suficiente de pruebas automatizadas. Si la cobertura comienza a disminuir, podría ser una señal de que no se está tratando las pruebas automatizadas como un elemento esencial de la escritura de nuevo código.
- Sin embargo, aunque la cobertura de código indica qué parte del código está cubierta por pruebas, no le indica la eficacia de esas pruebas ni si abordan todos los modos de fallo.

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 partes del código se ejecuta en 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

- 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 automáticamente ejecuta las pruebas unitarias contra las versiones modificadas 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.

http://pitest.org/

Pitest

- PIT se puede ejecutar con ant, maven, gradle y otros, o integrarlo en Eclipse (Pitclipse), 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, se pueden establecer exclusiones para optimizar el proceso (código generado, test de integración, …).
- 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.

Maven

```
<build>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>1.15.3</version>
            <dependencies>
                <dependency>
                    <groupId>org.pitest</groupId>
                    <artifactId>pitest-junit5-plugin</artifactId>
                    <version>1.2.1</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>

mvn test-compile org.pitest:pitest-maven:mutationCoverage
```

SPRING

Introducción

- La creación y el mantenimiento de pruebas para aplicaciones Spring presenta a los desarrolladores un conjunto único de desafíos, que incluyen los siguientes:
  - El marco de Spring debe inicializarse y configurarse
  - La aplicación suele tener dependencias de biblioteca de 3ª partes (almacenamiento persistente, servicios externos, etc.)
  - Las aplicaciones a menudo usan funciones integradas de Spring para sesiones, seguridad, mensajería, etc. Estas pueden ser difíciles de configurar para los desarrolladores que son nuevos en las pruebas de Spring.
  - Las dependencias de la aplicación (es decir, beans) deben configurarse adecuadamente
- Spring Framework proporciona un soporte de primera clase para las pruebas de integración en el módulo spring-test.
- El soporte de pruebas de integración de Spring tiene los siguientes objetivos principales:
  - Para administrar el almacenamiento en caché del contenedor Spring IoC entre la ejecución de la prueba.
  - Proporcionar inyección de dependencia personalizada en las prueba.
  - Proporcionar una gestión de transacciones adecuada a las pruebas de integración.
  - Proporcionar clases base específicas de Spring que ayuden a los desarrolladores a escribir pruebas de integración.

Inyección de dependencias

- Cuando el marco TestContext carga el contexto de la aplicación, opcionalmente se puede configurar instancias de sus clases de prueba a través de Dependency Injection.
- Esto proporciona un mecanismo conveniente para configurar casos de prueba utilizando beans preconfigurados del contexto de la aplicación.
- Uno de los grandes beneficios aquí es que se puede reutilizar contextos de aplicación en varios escenarios de prueba (por ejemplo, para configurar gráficos de objetos administrados por Spring, proxies transaccionales, fuentes de datos, etc.), evitando así la necesidad de duplicar la configuración compleja de dispositivos de prueba para casos de prueba individuales.
- Se debe crear un fichero xml o clase de configuración y asociarla a la clase de prueba mediante la anotación @ContextConfiguration o una clase interna anotada con @TestConfiguration.

@ContextConfiguration

```
public static class IoCTestConfig {
	@Bean
	Servicio getServicio() {
		// return new ServicioImpl();
		return new ServicioMockImpl();
	}
}

@Nested
@ContextConfiguration(classes = IoCTestConfig.class)
class IoCTest {
	@Autowired
	Servicio srv;

	@Test
	void contextLoads() {
		assertEquals("Soy el doble de pruebas", srv.getName());
	}
}
```

@TestConfiguration

```
@Nested
class IoCTest {

    @TestConfiguration
    public static class IoCTestConfig {
        @Bean
        Servicio getServicio() {
            return new ServicioMockImpl();
        }
    }

    @Autowired
    Servicio srv;

    @Test
    void contextLoads() {
        assertEquals("Soy el doble de pruebas", srv.getName());
    }
}
```


Sprint Boot

- Spring Boot proporciona una serie de utilidades y anotaciones para ayudarlo a probar su aplicación.
- El "Starter" spring-boot-starter-test (en test scope) contiene las siguientes bibliotecas:
  - JUnit 5 : El estándar de facto para pruebas unitarias de aplicaciones Java.
  - Spring Test y Spring Boot Test: compatibilidad con pruebas de integración y utilidades para aplicaciones Spring Boot.
  - AssertJ: una biblioteca de aserciones fluyentes.
  - Hamcrest: una biblioteca de objetos coincidentes (también conocidos como restricciones o predicados).
  - Mockito: marco de referencia para dobles de prueba simulación en Java.
  - JSONassert: una biblioteca de aserciones para JSON.
  - JsonPath: XPath para JSON.

Sprint Boot

- Una aplicación Spring Boot es un ApplicationContext de Spring, por lo que no se debe hacer nada especial para probarlo más allá de lo que normalmente haría con un contexto Spring vainilla.
- Spring Boot proporciona la anotación @SpringBootTest, que se puede utilizar como alternativa a la anotación estándar @ContextConfiguration cuando se necesiten las funciones de Spring Boot. De forma predeterminada, @SpringBootTest no iniciará un servidor. Puede usar el atributo webEnvironment de @SpringBootTest para refinar aún más cómo se ejecutan sus pruebas:
  - MOCK(Predeterminado): carga una web ApplicationContext y proporciona un entorno web simulado. Los servidores integrados no se inician cuando se usa esta anotación. Si un entorno web no está disponible en el classpath, este modo recurre de forma transparente a la creación de un entorno no web normal ApplicationContext. Para pruebas basadas en simulacros de su aplicación web, se puede utilizar con @AutoConfigureMockMvc o @AutoConfigureWebTestClient.
  - RANDOM_PORT: Carga un WebServerApplicationContext y proporciona un entorno web real. Los servidores integrados se inician y escuchan en un puerto aleatorio.
  - DEFINED_PORT: Carga un WebServerApplicationContext y proporciona un entorno web real. Los servidores integrados se inician y escuchan en un puerto definido (en application.properties) o en el puerto predeterminado de 8080.
  - NONE: Carga un ApplicationContext pero no proporciona ningún entorno web (simulado o de otro tipo).

Autoconfiguraciones

- @AutoConfigureTestDatabase: Por lo general tendremos una Base de Datos embebida o en memoria, esta anotación, sin embargo, nos permitirá realizar nuestros test contra una base de datos real.
- @AutoconfigureMockMvc: Nos va a permitir añadir un MockMvc a nuestro ApplicationContext, de esta manera podremos realizar peticiones HTTP contra nuestro controlador.
- @AutoConfigureWebTestClient: Nos permite verificar los endpoint del servidor, agrega WebTestClient al contexto.
- @DataMongoTest: para probar aplicaciones MongoDB, que configura un MongoDB incrustado en memoria si el controlador está en las dependencias, configura un MongoTemplate, busca @Document y configura los repositorios.
- @DataRedisTest: facilita la prueba de aplicaciones de Redis. Busca clases de @RedisHash y configura repositorios Spring Data Redis de forma predeterminada.

Archivos de propiedades

- Al igual que hacemos en nuestra aplicación Spring Boot haciendo uso de diferentes perfiles a través de nuestras propiedades, en nuestros test integrados podemos también cargar propiedades específicas (como application-test.yml) con la anotación @ActiveProfiles.

```
@SpringBootTest
@ActiveProfiles("test")
class MyApplicationTests {
```

- Otra manera de poder cargar propiedades en nuestro ApplicationContext, es haciendo uso de @TestPropertySource, con lo que podemos cargar un fichero personalizado (como src/test/resources/test.properties) para nuestros test.

```
@TestPropertySource(locations = "/test.properties")
```

- ActiveProfiles cargar un archivo de propiedades entero, pero en muchas ocasiones, solo necesitamos sobre escribir alguna propiedad. La anotación @SpringBootTest permite establecer el valor dichas propiedades:

```
@SpringBootTest(properties = {"perfil.valor=Valor para pruebas"})
class MyApplicationTests {
```


@DataJpaTest

- Podemos usar anotaciones @DataJpaTest para que las pruebas se centren solo en probar los componentes JPA. @DataJpaTest deshabilitará la configuración automática completa del contexto de la aplicación y, en su lugar, aplicará solo la configuración relevante para los componentes y las pruebas de JPA.
- @DataJpaTest proporciona una configuración estándar necesaria para probar la capa de persistencia:
  - configurando una base de datos en memoria si está disponible en el classpath (H2). Se puede anular con @AutoConfigureTestDatabase(replace = Replace.NONE)
  - configurando Hibernate, Spring Data y DataSource
  - realizando un @EntityScan
  - activando el registro de SQL (spring.jpa.show-sql=true)

@Sql

- La anotación @Sql proporciona una forma declarativa a nivel de clase y método de inicializar y rellenar un esquema de prueba. Con executionPhase se establece la fase: BEFORE_TEST_CLASS, BEFORE_TEST_METHOD, AFTER_TEST_METHOD o AFTER_TEST_CLASS.

```
@Sql(scripts = {"/employees_schema.sql", "/employees_data1.sql"}, executionPhase = BEFORE_TEST_CLASS)
public class SpringBootInitialLoadIntegrationTest {
    @Test
    @Sql(scripts = {"/employees_data2.sql"}, executionPhase = BEFORE_TEST_METHOD)
    public void testLoadDataForTestCase() {
```

- La anotación @SqlConfig, en la clase o en una @Sql, permite configurar la forma en que se analiza y ejecutan los scripts SQL: codificación, comentarios, modelo de errores, transacciones, …
- @Sql permite el uso de anotaciones repetidas para diferentes configuraciones.

TestEntityManager

- Para llevar a cabo operaciones de base de datos, necesitamos algunos registros que ya estén en nuestra base de datos. Para configurar estos datos, podemos inyectar un bean TestEntityManager, una alternativa al JPA EntityManager estándar, que proporciona los métodos comúnmente utilizados al escribir pruebas.
- De forma predeterminada, las pruebas JPA de datos son transaccionales que se anulan al final de cada prueba (auto roll back). Para deshabilitar este comportamiento:

```
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class TestAppData {
```

- La anotación @AutoConfigureTestEntityManager permite utilizar TestEntityManager sin la anotación @DataJpaTest.

TestEntityManager

- Para agregar una base de datos en memoria H2 al classpath:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

- Para utilizar el TestEntityManager:

```
@DataJpaTest
class ActorRepositoryTest {
    @Autowired
    private TestEntityManager em;
    @BeforeEach
    void setUp() throws Exception {
        em.persist(new Actor(0, "Pepito", "Grillo"));
        em.persist(new Actor(0, "Carmelo", "Coton"));
        em.persist(new Actor(0, "Capitan", "Tan"));
```


TestEntityManager

```
@DataJpaTest
class ActorRepositoryTest {
	@Autowired
	private TestEntityManager em;
	@Autowired
	ActorRepository dao;
	@Test
	void testGetAll_isEmpty() {
		var rslt = dao.findAll();
		assertThat(rslt).isEmpty();  }

	@Nested
	class Con_datos {
		@Autowired
		private TestEntityManager em;

		@BeforeEach
		void setUp() throws Exception {
			em.persist(new Actor(0, "Pepito", "Grillo"));
			em.persist(new Actor(0, "Carmelo", "Coton"));
			em.persist(new Actor(0, "Capitan", "Tan"));
		}
		@Test
		void testGetAll_isNotEmpty() {
			var rslt = dao.findAll();
			assertThat(rslt.size()).isEqualTo(3);
			assertThat(dao.findTop10ByFirstNameStartingWithOrderByLastNameDesc("C").size()).isEqualTo(2);
			assertThat(dao.findByActorIdGreaterThan(2).size()).isEqualTo(1);
		}

		@Test
		void testGetOne() {
			var item = dao.findById(dao.findAll().get(0).getActorId());
			assertThat(item.isPresent()).isTrue();
			assertThat(item.get()).asString().isEqualTo("Actor [actorId=4, firstName=Pepito, lastName=Grillo, lastUpdate=null]");
		}

	}
```

@MockBean

- La anotación @MockBean permite agregar objetos simulados al contexto de la aplicación Spring. El doble de prueba reemplazará cualquier bean existente del mismo tipo en el contexto de la aplicación. Si no se defino uno del mismo tipo, se agrega uno nuevo. Se puede usar como una anotación de nivel de clase o en campos de clases @Configuration. Es útil en pruebas unitarias y de integración en las que es necesario aislarlas de las dependencias.
- La anotación @Mock solo se usa en clases de prueba para crear un doble de prueba de una clase o interfaz que no participa en la inyección de dependencias.
- La infraestructura de dobles de pruebas, también para @MockBean, la suministra Mockito.

@MockBean

```
@ComponentScan(basePackages = {"com.example"})
class ActorServiceTest {
	@MockBean
	ActorRepository dao;

	@Autowired
	ActorServiceImpl srv;

	@Test
	void testGetAll_isNotEmpty() {
		List<Actor> lista = new ArrayList<>(
			Arrays.asList(new Actor(1, "Pepito", "Grillo"),
				new Actor(2, "Carmelo", "Coton"),
				new Actor(3, "Capitan", "Tan")));

		when(dao.findAll()).thenReturn(lista);
		var rslt = srv.getAll();
		assertThat(rslt.size()).isEqualTo(3);
	}
```

MockMvc

- Se pueden escribir pruebas unitarias sencillas para Spring MVC instanciando un controlador, inyectándole dependencias y llamando a sus métodos. Sin embargo, dichas pruebas no verifican las asignaciones de solicitudes, el enlace de datos, la conversión de mensajes, la conversión de tipos, la validación y tampoco involucran ninguno de los métodos de soporte @InitBinder, @ModelAttribute o @ExceptionHandler.
- MockMvc proporciona soporte para probar aplicaciones Spring MVC. Realiza el manejo completo de solicitudes de Spring MVC, pero a través de objetos de solicitud y respuesta simulados en lugar de un servidor en ejecución. MockMvc se puede usar solo para realizar solicitudes y verificar respuestas.
- También se puede usar a través de @WebTestClient donde MockMvc está conectado como el servidor para manejar las solicitudes. La ventaja de @WebTestClient es la opción de trabajar con objetos de nivel superior en lugar de datos sin procesar, así como la capacidad de cambiar a pruebas HTTP completas de extremo a extremo en un servidor en vivo y usar la misma API de prueba.

MockMvc

- Cuando se use MockMvc directamente para realizar solicitudes, son convenientes las siguientes importaciones estáticas:

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
```

- Se puede configurar MockMvc para probar un controlador específico:

```
MockMvc mockMvc;
@BeforeEach
void setup() {
   this.mockMvc = MockMvcBuilders.standaloneSetup(new DemoController()).build();
}
```

- MockMvc se puede autoconfigurar con la anotación @WebMvcTest.

@WebMvcTest

- @WebMvcTest configura automáticamente la infraestructura Spring MVC y limita los beans escaneados a @Controller, @ControllerAdvice, @JsonComponent, Converter, GenericConverter, HandlerInterceptor, HandlerMethodArgumentResolver, Filter, WebMvcConfigurer y WebMvcRegistrations. No se escanean los @Component, @Service, @Repository y @ConfigurationProperties, aunque estos últimos se pueden incluir con @EnableConfigurationProperties.
- A menudo, @WebMvcTest se limita a un solo controlador y se usa en combinación @MockBean para proporcionar implementaciones simuladas para los colaboradores necesarios. @WebMvcTest también configura automáticamente MockMvc.

```
@WebMvcTest(DemoController.class)
class DemoControllerTests {
   @Autowired
   private MockMvc mockMvc;
   @MockBean
   private DemoService srv;
```


Solicitudes MockMvc

- MockMvc dispone de un api fluyente para realizar solicitudes y verificar respuestas.

```
mockMvc.perform(…).andExpect(…)...andDo(print());
```

- Para realizar solicitudes que utilicen cualquier método HTTP:
  - get(), post(), put(), delete(), multipart(), …
  - .queryParam().header().accept().contentType().content()

```
mockMvc.perform(get("/api/v1/actores/{id}", 1)
    .queryParam("mode", "edit"))
mockMvc.perform(put("/api/v1/actores/{id}", 1)
    .contentType(MediaType.APPLICATION_JSON)
    .content(objectMapper.writeValueAsString(ele)))
```

Expectativas MockMvc

- Se puede definir múltiples expectativas agregando .andExpectAll(...) o encadenando .andExpect() después de realizar una solicitud.
- MockMvcResultMatchers.* proporciona una serie de expectativas, algunas de las cuales están anidadas con expectativas más detalladas.
- Las expectativas se dividen en afirmaciones que verifican las propiedades de la respuesta (por ejemplo, el estado de la respuesta, los encabezados y el contenido) y otras que permiten inspeccionar aspectos específicos de Spring MVC, como qué método de controlador procesó la solicitud, si se generó y manejó una excepción, cuál es el contenido del modelo, qué vista se seleccionó, qué atributos flash se agregaron, etc.

Expectativas MockMvc

- Se puede definir múltiples expectativas agregando .andExpectAll(...) o encadenando .andExpect() después de realizar una solicitud.
- MockMvcResultMatchers.* proporciona una serie de expectativas, algunas de las cuales están anidadas con expectativas más detalladas.
- Las expectativas se dividen en afirmaciones que verifican las propiedades de la respuesta (por ejemplo, el estado de la respuesta, los encabezados y el contenido) y otras que permiten inspeccionar aspectos específicos de Spring MVC, como qué método de controlador procesó la solicitud, si se generó y manejó una excepción, cuál es el contenido del modelo, qué vista se seleccionó, qué atributos flash se agregaron, etc.

Expectativas MockMvc

- status(): Acceso a aserciones de estado de respuesta.
- cookie(): Acceso a aserciones de cookies de respuesta.
- header(): Acceso a aserciones de encabezado de respuesta.
- content(): Acceso a aserciones del cuerpo de respuesta.
- jsonPath(): Acceso a las aserciones del cuerpo de la respuesta mediante una expresión JsonPath para inspeccionar un subconjunto específico del cuerpo.
- xpath(): Acceso a las aserciones del cuerpo de la respuesta mediante una expresión XPath para inspeccionar un subconjunto específico del cuerpo.
- redirectedUrl(): Afirma que la solicitud se redirigió a la URL dada.
- forwardedUrl(): Afirma que la solicitud se reenvió a la URL dada.
- view(): Acceso a aserciones en la vista seleccionada.
- model(): Acceso a aserciones relacionadas con el modelo.
- handler(): Acceso a aserciones para el controlador que manejó la solicitud.
- request(): Acceso a aserciones relacionadas con la solicitud.

Expectativas MockMvc

```
mockMvc.perform(…)
        .andExpectAll(
                status().isOk(),
                content().contentType("application/json"),
                jsonPath("$.size()").value(3)
        );
mockMvc.perform(…)
        .andExpect(status().isCreated())
        .andExpect(header().string("Location", url + "/1"));
mockMvc.perform(…)
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("$.error").value("Not found"));
mockMvc.perform(…)
        .andExpect(content().contentType(MediaType.APPLICATION_XML))
        .andExpect(xpath("/person/ns:link[@rel='self']/@href", ns)
                .string("http://localhost:8080/people"));
```

Salidas de MockMvc

- Muchas veces, al escribir pruebas, es útil volcar los resultados de la solicitud realizada.

  mockMvc…andDo(print())
- Siempre que el procesamiento de solicitudes no provoque una excepción no controlada, el método print() imprime todos los datos de resultados disponibles en System.out. Si se desea que se registren los datos de resultados en lugar de imprimirlos, se puede invocar el método log(), que registra los datos de resultados como un solo mensaje DEBUG en la categoría de registro org.springframework.test.web.servlet.result.
- En algunos casos, es posible que se desee obtener acceso directo al resultado y verificar algo que no se puede verificar de otra manera. Esto se puede lograr agregando .andReturn() después de todas las demás expectativas:

  MvcResult mvcResult = mockMvc.perform(…).andExpect(…).andReturn();

MockMvc

```
@WebMvcTest(DemosResource.class)
class DemosResourceTest {
	@MockBean
	ActorRepository dao;

	@Autowired
	private DemosResource srv;

	@Autowired
	private MockMvc mockMvc;

	@Test
	void testGetInicio() throws Exception {
		assertEquals("{saludo=Hola mundo}", srv.getInicio().toString());
		mockMvc.perform(get("/")).andExpect(status().isOk())
			.andExpect(content().contentType(MediaType.APPLICATION_JSON))
			.andExpect(jsonPath("$.saludo").value("Hola mundo"));
	}

	@Test
	void testGetActores() throws Exception {
		List<ActorDTO> lista = new ArrayList<>(
		    Arrays.asList(new ActorDTO(1, "Pepito", "Grillo"),
				new ActorDTO(2, "Carmelo", "Coton"),
				new ActorDTO(3, "Capitan", "Tan")));

		when(dao.findByActorIdIsNotNull(ActorDTO.class)).thenReturn(lista);
		mockMvc.perform(get("/actores")).andExpect(status().isOk())
			.andExpect(content().contentType(MediaType.APPLICATION_JSON))
			.andExpect(jsonPath("$.size()").value(3))
			.andExpect(jsonPath("$[0].nombre").value("Pepito"))
			.andExpect(jsonPath("$[2].nombre").value("Capitan"))
			.andDo(print());
	}
```

TestRestTemplate

- TestRestTemplate es una alternativa conveniente a RestTemplate que es útil en las pruebas de integración. Se puede obtener una plantilla estándar o una que envíe autenticación HTTP básica (con un nombre de usuario y contraseña). La plantilla es tolerante a errores, se comporta de una manera amigable con las pruebas al no generar excepciones en los errores 4xx y 5xx, dichos errores se deben detectar a través del código de estado del ResponseEntity devuelto.
- Se puede crear una instancia TestRestTemplate directamente en las pruebas de integración:

```
private final TestRestTemplate template = new TestRestTemplate();
```

- Si se usa la anotación @SpringBootTest con RANDOM_PORT o DEFINED_PORT, se puede inyectar una configuración completa TestRestTemplate y comenzar a usarla.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MySpringBootTests {
   @Autowired
   private TestRestTemplate template;
```


MockRestServiceServer

- MockRestServiceServer es el punto de entrada principal para las pruebas REST del lado del cliente. Se utiliza para pruebas que implican el uso directo o indirecto de RestTemplate, código que usa internamente el RestTemplate. Proporciona una forma de configurar las solicitudes esperadas que se realizarán a través de RestTemplate y las respuestas simuladas para enviar de vuelta, eliminando así la necesidad de un servidor real. La idea es declarar las solicitudes esperadas y proporcionar respuestas "stub" para concentrarse en probar el código de forma aislada (es decir, sin ejecutar un servidor).

```
import static org.springframework.test.web.client.match.MockRestRequestMatchers.*;
import static org.springframework.test.web.client.response.MockRestResponseCreators.*;

RestTemplate restTemplate = new RestTemplate();
MockRestServiceServer server = MockRestServiceServer.bindTo(restTemplate).build();
server.expect(ExpectedCount.manyTimes(),
requestTo("/fake/42")).andExpect(method(HttpMethod.GET)).andRespond(
      withSuccess("{ \"id\" : \"42\", \"nombre\" : \"pepito\"}", MediaType.APPLICATION_JSON));

ActorDTO obj = restTemplate.getForObject("/fake/{id}", ActorDTO.class, 42);
```

@RestClientTest

- Se puede usar la anotación @RestClientTest para probar clientes REST, con código que usa internamente el RestTemplate. De forma predeterminada, configura automáticamente la compatibilidad con Jackson, GSON y Jsonb, configura un RestTemplateBuild y añade soporte para MockRestServiceServer.

```
//@AutoConfigureWebClient(registerRestTemplate=true).
@RestClientTest(RemoteService.class)
class MyRestClientTests {
   @Autowired
   private RemoteService service;
   @Autowired
   private MockRestServiceServer server;
   @Test
   void getDetailsWhenResultIsSuccessShouldReturnDetails() {
      this.server.expect(requestTo("/greet/details"))
          .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN));
      String greeting = this.service.callRestService();
      assertThat(greeting).isEqualTo("hello");
   }
}
```


@JsonTest

- Para probar que la serialización y deserialización JSON de objetos funciona como se esperaba, se puede usar la anotación @JsonTest. @JsonTest configura automáticamente el asignador JSON compatible disponible, que puede ser una de las siguientes bibliotecas:
  - Jackson ObjectMapper, cualquier @JsonComponent y cualquier ModuleJackson
  - Gson
  - Jsonb

@JsonTest

```
@JsonTest
public class MyJsonTests {
   @Autowired
   private JacksonTester<ActorDTO> json;

   @Test
   void serialize() throws Exception {
      var details = new ActorDTO(1, "Pepito", "Grillo");
      assertThat(this.json.write(details)).hasJsonPathNumberValue("@.id");
      assertThat(this.json.write(details)).extractingJsonPathStringValue("@.nombre") .isEqualTo(details.getFirstName());
   }

   @Test
   void deserialize() throws Exception {
      String content = "{\"id\":1,\"nombre\":\"Pepito\",\"apellidos\":\"Grillo\"}";
      assertThat(this.json.parse(content)).isEqualTo(new ActorDTO(1, "Pepito", "Grillo"));
      assertThat(this.json.parseObject(content).getFirstName()).isEqualTo("Pepito");
   }
}
```