



Herramientas de Pruebas



Visual Studio

© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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, ...

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 definitiva; la prueba debería ser realizada por usuarios finales.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

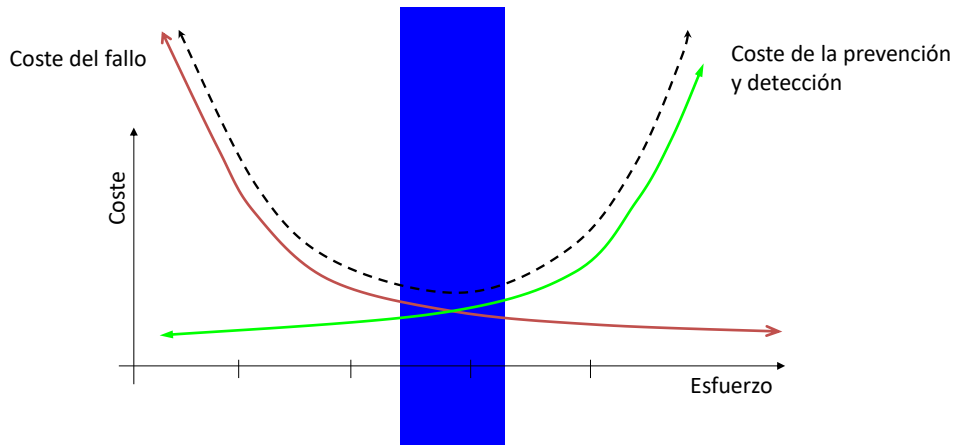
© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Las pruebas tienen un coste



© JMA 2020. All rights reserved

Pirámide de pruebas



© JMA 2020. All rights reserved

<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

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

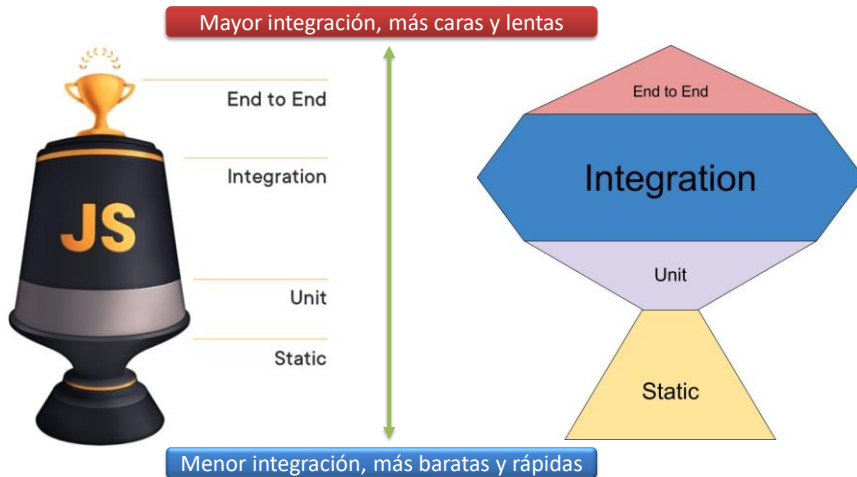
© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

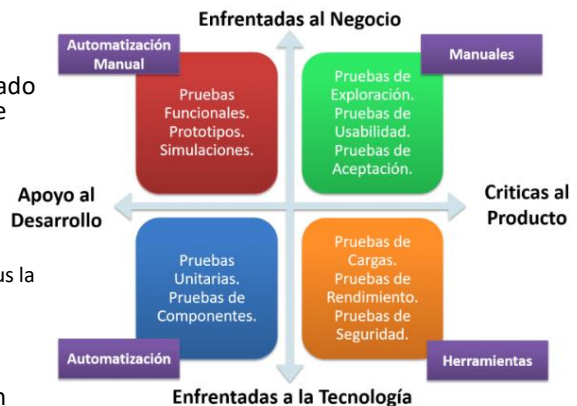
Testing Trophy



© JMA 2020. All rights reserved

Cuadrante de Pruebas

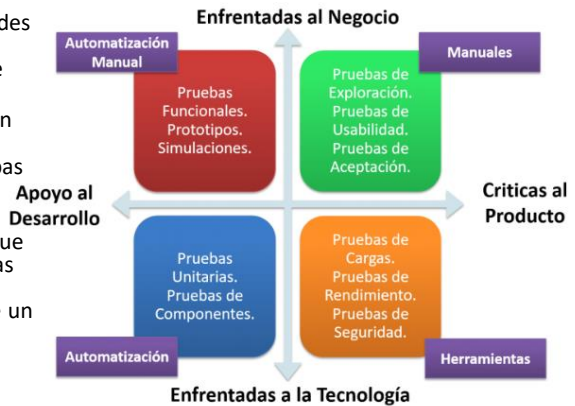
- El cuadrante de prueba divide todo el proceso de prueba en 4 partes. Esto hace que el proceso de prueba sea fácil de entender.
- El cuadrante de pruebas fue originalmente publicado por Brian Marick en 2003 como parte de una serie de artículos sobre Testeo Ágil y originalmente fue conocido como "Matriz 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.



© JMA 2020. All rights reserved

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.



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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, límites).
- 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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 enfrentarse 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.

© JMA 2020. All rights reserved

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 pruebas se diseñan para fallar si se cambia el comportamiento dependiente y deberían seguir pasando ante cambios no dependientes.

© JMA 2020. All rights reserved

Quality Assurance: Diseñar para probar

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

© JMA 2020. All rights reserved

Programar con interfaces

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

© JMA 2020. All rights reserved

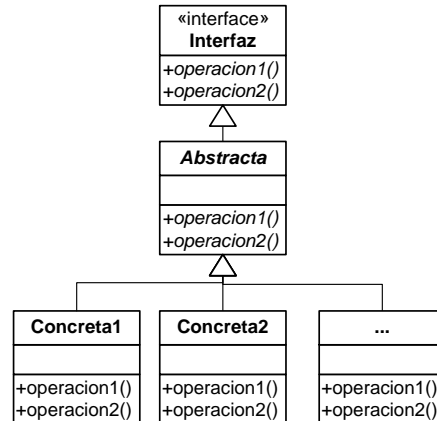
Doble Herencia

- **Problema:**

- Mantener las clases que implementan como internas del proyecto (internal o Friend), pero la interfaz pública.
- Organizar clases que tienen un comportamiento parecido para que sea consistente.

- **Solución:**

- Clase base es abstracta.
- La clase base puede heredar de mas de una interfaz.
- Una vez que están escritos los métodos, verifico si hay duplicación en las clases hijas.



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Simulación de objetos

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

© JMA 2020. All rights reserved

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.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

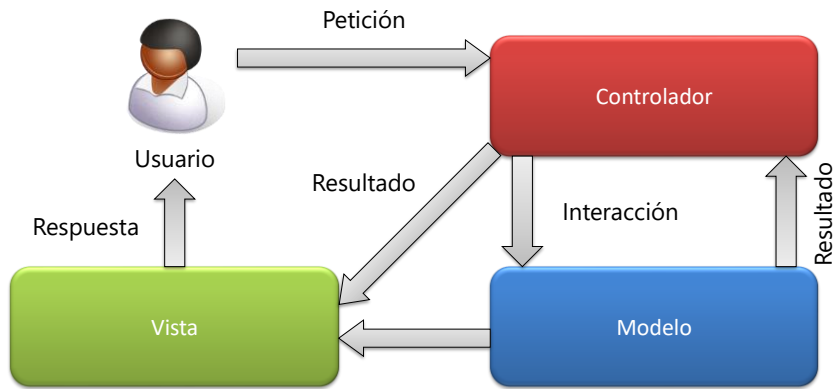
© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

El patrón MVC



© JMA 2020. All rights reserved

El patrón MVC



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



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

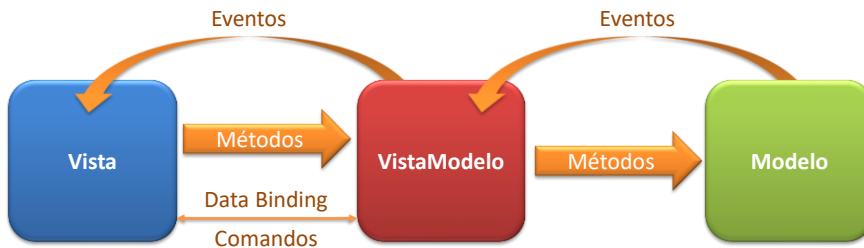


- **Intermediario** entre Modelo y Vista
- **Mapa acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

© JMA 2020. All rights reserved

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.



© JMA 2020. All rights reserved

Desarrollo Guiado por Pruebas (TDD)

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

© JMA 2020. All rights reserved

Ritmo TDD

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

© JMA 2020. All rights reserved

Beneficios de TDD

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

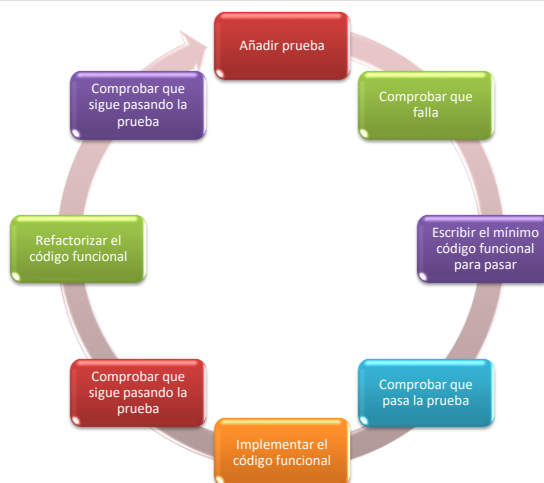
© JMA 2020. All rights reserved

Estrategia RED – GREEN

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

© JMA 2020. All rights reserved

Ritmo TDD



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 prerequisites 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.

© JMA 2020. All rights reserved

SpecFlow

<https://specflow.org/>

- Es una herramienta de código abierto para proyectos .NET que se creó con la misión de proporcionar un marco para trabajar con Especificación–Por–Ejemplo.
- Aspira a reducir la brecha de comunicación entre los que conocen el negocio de la aplicación y los desarrolladores mediante la unión de las especificaciones de negocio y ejemplos para la implementación subyacente. Se integra fácilmente en nuestros proyectos .Net, a diferencia de las otras opciones disponibles. Utiliza el lenguaje Gherkin (Given-When-Then) para la definición de escenarios de pruebas lo que facilita la creación de los mismos por parte de las personas del negocio.
- La extensión SpecFlow para Visual Studio proporciona una serie de características útiles, como el resaltado de sintaxis para archivos Gherkin (características). Esta extensión no es necesaria para usar SpecFlow, pero le recomendamos que la instale si está utilizando Visual Studio.

© JMA 2020. All rights reserved

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 software lo hace.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 generar datos de métricas de código que medir la complejidad y el mantenimiento del código administrado.

© JMA 2020. All rights reserved

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. 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.
- La cobertura de prueba tradicional (líneas, instrucciones, ramas, etc.) mide solo qué código ejecuta las pruebas. No comprueba que las pruebas son realmente capaces de detectar fallos en el código ejecutado, solo pueden identificar el código que no se ha probado. Los ejemplos más extremos del problema son las pruebas sin afirmaciones (poco comunes en la mayoría de los casos). Mucho más común es el código que solo se prueba parcialmente, cubrir todo los caminos no implica ejercitar todos 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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

AUTOMATIZACIÓN DE PRUEBAS

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Herramientas de automatización de pruebas



© JMA 2020. All rights reserved

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 entornos.
 - 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.

© JMA 2020. All rights reserved

HERRAMIENTAS DE PRUEBA DE VISUAL STUDIO

© JMA 2020. All rights reserved

Ecosistema

- Alrededor de las pruebas y el aseguramiento de la calidad existe todo un ecosistema de herramientas que se utilizan en una o más actividades de soporte de prueba, entre las que se encuentran:
 - Las herramientas que se utilizan directamente en las pruebas, como las herramientas de ejecución de pruebas, las herramientas de generación de datos de prueba y las herramientas de comparación de resultados.
 - Las herramientas que ayudan a gestionar el proceso de pruebas, como las que sirven para gestionar pruebas, resultados de pruebas, datos requisitos, incidencias, defectos ..., así como para elaborar informes y monitorizar la ejecución de pruebas.
 - Las herramientas que se utilizan en la fase de reconocimiento, como las herramientas de estrés y las herramientas que monitorizan y supervisan la actividad del sistema.
 - Cualquier otra herramienta que contribuya al proceso de pruebas sin ser específicas del mismo, como las hojas de cálculo, procesadores de texto, diagramadores, ...
- Las herramientas pueden clasificarse en base a distintos criterios, tales como el objetivo, comercial/código abierto, específicas/integradas, tecnología utilizada ...

© JMA 2020. All rights reserved

Objetivos

- Mejorar la eficiencia de las tareas de pruebas automatizando tareas repetitivas o dando soporte a las actividades de pruebas manuales, como la planificación, el diseño, la elaboración de informes y la monitorización de pruebas.
- Automatizar aquellas actividades que requieren muchos recursos si se hacen de forma manuales (como por ejemplo, las pruebas estáticas, pruebas de GUI).
- Automatizar aquellas actividades que no pueden ejecutarse de forma manual (como por ejemplo , pruebas de rendimiento a gran escala de aplicaciones cliente-servidor).
- Aumentar la fiabilidad de las pruebas (por ejemplo, automatizando las comparaciones de grandes ficheros de datos y simulando comportamientos).

© JMA 2020. All rights reserved

Ventajas

- Reducción del trabajo repetitivo (por ejemplo, la ejecución de pruebas de regresión, la reintroducción de los mismos datos de prueba y la comprobación contra estándares de codificación).
- Mayor consistencia y respetabilidad (por ejemplo las pruebas ejecutadas por una herramienta en el mismo orden y con la misma frecuencia, y pruebas derivadas de los requisitos).
- Evaluación de los objetivos (por ejemplo, medidas estáticas, cobertura).
- Facilidad de acceso a la información sobre las pruebas (por ejemplo, estadísticas y gráficos sobre el avance de las pruebas, la frecuencia de incidencias y el rendimiento).

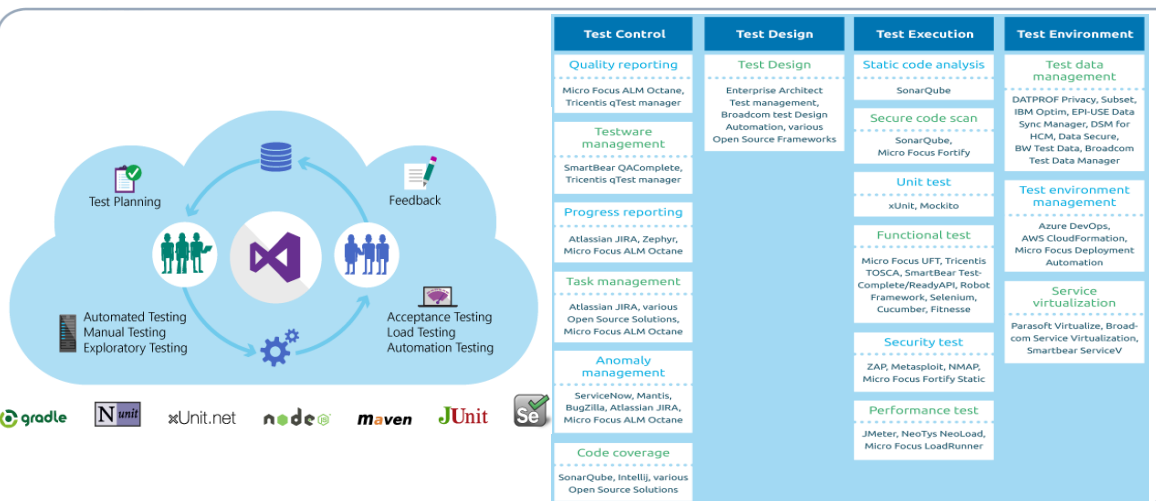
© JMA 2020. All rights reserved

Desventajas

- Expectativas poco realistas de la herramienta (incluyendo funcionalidad y facilidad de uso).
- Exceso de confianza en la herramienta (sustitución por el diseño de pruebas o uso de pruebas automatizadas cuando sería mejor llevar a cabo pruebas manuales).
- Subestimar:
 - La cantidad de tiempo, coste y esfuerzo necesario para la introducción inicial de una herramienta (incluyendo formación y experiencia externa).
 - El tiempo y el esfuerzo necesarios para conseguir ventajas significativas y constantes de la herramienta (incluyendo la necesidad de cambios en el proceso de pruebas y la mejora continua de la forma en la que se utiliza la herramienta).
 - El esfuerzo necesario para mantener los activos de prueba generados por la herramienta.
- Desprecio del control de versión de los activos de prueba en la herramienta.
- Desprecio de problemas de relaciones e interoperabilidad entre herramientas críticas tales como las herramientas de gestión de requisitos, herramientas de control de versiones, herramientas de gestión de incidencias, herramientas de seguimiento de defectos y herramientas procedentes de varios fabricantes.
- Coste de las herramientas comerciales o ausencia de garantías en las herramientas open source.
- Riesgo de que el fabricante de la herramienta cierre, retire la herramienta o venda la herramienta a otro proveedor.
- Mala respuesta del fabricante para soporte, actualizaciones y corrección de defectos.
- Imprevistos, tales como la incapacidad de soportar una nueva plataforma.

© JMA 2020. All rights reserved

Herramientas de prueba



© JMA 2020. All rights reserved

Marcos de pruebas .NET

- Unit Test Frameworks:
 - MSTest, NUnit, xUnit (<https://xunit.net/docs/comparisons>)
- Mocking Frameworks:
 - Microsoft Fakes, Moq, NSubstitute, Rhino Mocks, FakeItEasy, EntityFramework.Testing
- Code Coverage (Métrica de calidad del software: Valor cuantitativo que indica que cantidad del código ha sido ejercitada por un conjunto de casos de prueba):
 - .NET: Visual Studio, NCover, NCrunch, OpenCover
 - Java: Clover, EMMA, Cobertura
 - Ruby: RCov

© JMA 2020. All rights reserved

Herramientas de prueba de Visual Studio

- Las herramientas de prueba de Visual Studio permiten desarrollar y mantener altos estándares de excelencia de código.
- Las pruebas unitarias están disponibles en todas las ediciones de Visual Studio.
- Otras herramientas de pruebas, como Live Unit Testing, IntelliTest y Pruebas automatizadas de IU, solo están disponibles en la edición Visual Studio Enterprise.

© JMA 2020. All rights reserved

Visual Studio

- **Explorador de pruebas**
 - La ventana del Explorador de pruebas ayuda a los desarrolladores a crear, administrar y ejecutar pruebas unitarias. Puede usar el marco de pruebas unitarias de Microsoft (MSTest) o uno de los marcos de terceros y de código abierto.
- **Live Unit Testing (edición Enterprise)**
 - Live Unit Testing ejecuta automáticamente pruebas unitarias en segundo plano y muestra una representación gráfica de los resultados de la prueba y la cobertura de código en el editor de código de Visual Studio.
- **IntelliTest (edición Enterprise)**
 - IntelliTest genera automáticamente pruebas unitarias y datos de prueba para el código administrado. IntelliTest mejora la cobertura y reduce drásticamente el esfuerzo de crear y mantener pruebas unitarias para código nuevo o existente.

© JMA 2020. All rights reserved

Visual Studio

- **Cobertura de código**
 - El análisis de cobertura de código puede aplicarse al código administrado y no administrado (nativo).
 - La cobertura de código es una opción al ejecutar métodos de prueba mediante el Explorador de pruebas. La tabla de salida muestra el porcentaje de código que se ejecuta en cada ensamblado, clase y método. Además, el editor de código fuente muestra qué código se ha probado.
 - Utilizar cobertura de código para determinar la cantidad de código que se está probando
 - Pruebas unitarias, cobertura de código y análisis de clon de código con Visual Studio (Lab)
 - Personalizar el análisis de cobertura de código
- **Microsoft Fakes**
 - Microsoft Fakes ayuda a aislar el código que se está probando mediante la sustitución de otros elementos de la aplicación con código auxiliar (stub) o correcciones de compatibilidad (shim).

© JMA 2020. All rights reserved

Visual Studio

- Pruebas de interfaz de usuario con UI codificada (obsoleta VS2019) y Selenium
 - Las pruebas de UI codificada proporcionan una manera de crear pruebas completamente automatizadas con el fin de validar la funcionalidad y el comportamiento de la interfaz de usuario de la aplicación. Pueden automatizar las pruebas de la interfaz de usuario en varias tecnologías, incluidas las aplicaciones de UWP basadas en XAML, las aplicaciones del explorador y las de SharePoint.
 - Tanto si se elige las pruebas de IU codificadas más convenientes como las pruebas genéricas de interfaz de usuario basadas en exploradores con Selenium, Visual Studio proporciona todas las herramientas que necesita.
- Pruebas de carga (obsoleta VS2019)
 - La prueba de carga simula la carga en una aplicación de servidor mediante la ejecución de pruebas unitarias y pruebas de rendimiento web.

© JMA 2020. All rights reserved

Prueba genérica

- La prueba genérica es útil para probar un archivo ejecutable existente. Es el proceso de envolver el archivo ejecutable como una prueba genérica y luego ejecutarlo. Este tipo de prueba es muy útil cuando se prueba un componente de terceros sin el código fuente. Si el ejecutable requiere algún archivo adicional para la prueba, se puede agregar lo mismo como archivos de implementación a la prueba genérica. La prueba se puede ejecutar utilizando el Explorador de prueba o un comando de línea de comandos.
- Al usar Visual Studio, podemos recopilar los resultados de la prueba y también recopilar datos de cobertura de código. Podemos administrar y ejecutar las pruebas genéricas en Visual Studio al igual que otras pruebas. Permite integrar procesos de pruebas externos en el proceso de pruebas de Visual Studio. De hecho, el resultado del resultado de la prueba se puede publicar en Team Foundation Server para vincularlo con el código creado para las pruebas.

© JMA 2020. All rights reserved

Obsoletas

- La prueba automatizada de IU para pruebas funcionales controladas por la interfaz de usuario está en desuso. Visual Studio 2019 es la última versión en la que la prueba automatizada de IU estará disponible. Se recomienda usar Selenium, para probar aplicaciones web, Appium con WinAppDriver, para probar aplicaciones de escritorio y para UWP, y Xamarin.UITest para probar aplicaciones de iOS y Android mediante el marco de pruebas NUnit.
- La funcionalidad de pruebas de carga y rendimiento web está en desuso. Visual Studio 2019 es la última versión en la que las pruebas de carga y rendimiento web estarán disponibles.
- Microsoft Test Manager 2017 (que se envió con Microsoft Visual Studio 2017) es la última versión, se recomienda usar Azure Test Plans o Test hub in TFS (una solución de administración de prueba con todas las funciones) para todos los requisitos de administración de prueba.

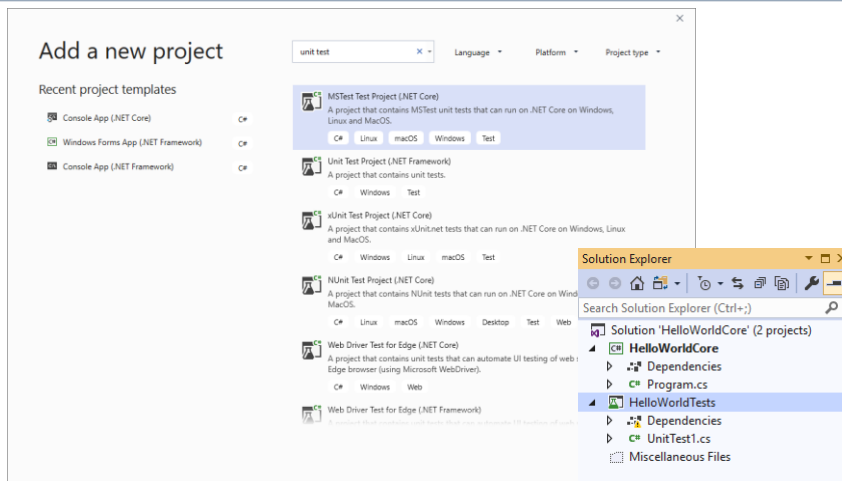
© JMA 2020. All rights reserved

Proyecto de prueba unitaria

- Las pruebas unitarias a menudo reflejan la estructura del código sometido a pruebas.
- Habitualmente, se crearía un proyecto de prueba unitaria para cada proyecto de código en el producto.
- El proyecto de prueba puede estar en la misma solución que el código fuente o puede estar en una solución independiente.
- En una solución se pueden tener tantos proyectos de prueba unitaria como sea necesario.
- Para crear un proyecto de prueba unitaria, en el menú Archivo > Nuevo > Proyecto, filtrar por tipo Prueba y seleccionar la plantilla de proyecto del marco de pruebas que se desea usar.
- En el proyecto de prueba unitaria, agregar la referencia al proyecto que se quiere probar haciendo clic con el botón derecho en Referencias o Dependencias y eligiendo Agregar referencia.
- En general, es más rápido generar el proyecto de prueba unitaria y los códigos auxiliares de pruebas unitarias a partir del código, haciendo clic con el botón derecho y seleccione *Crear pruebas unitarias* en el menú contextual.

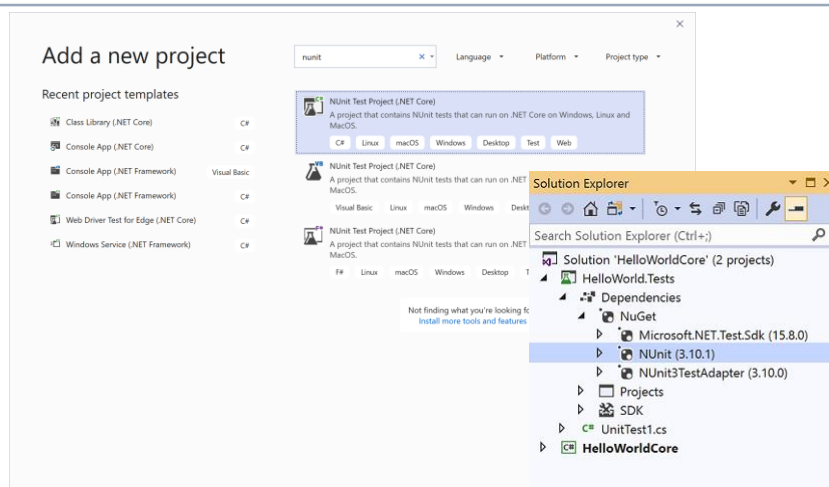
© JMA 2020. All rights reserved

Proyecto de prueba unitaria



© JMA 2020. All rights reserved

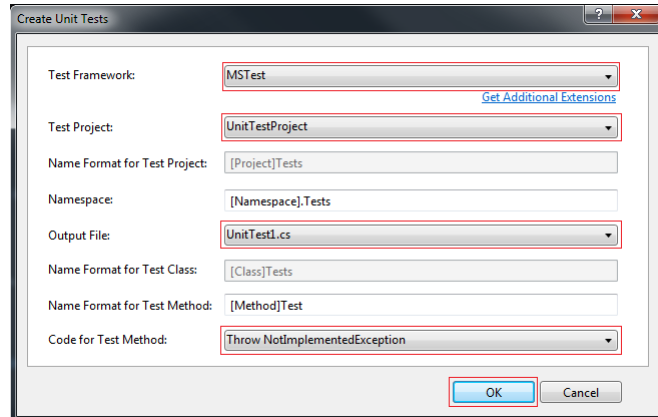
Marcos de pruebas de terceros



© JMA 2020. All rights reserved

Generar proyecto de prueba unitaria

- En la clase o método deseado hacer clic derecho → Crear pruebas unitarias



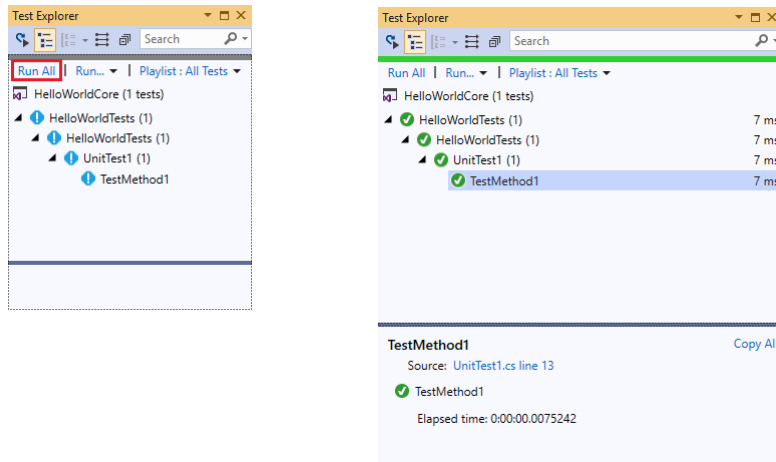
© JMA 2020. All rights reserved

Ejecutar pruebas unitarias

- Se puede usar el Explorador de pruebas para ejecutar pruebas unitarias en el marco de pruebas integrado (MSTest) o en marcos de pruebas de terceros.
- Se puede agrupar las pruebas en categorías, filtrar la lista de pruebas y crear, guardar y ejecutar listas de reproducción de pruebas.
- También se puede depurar las pruebas, analizar la cobertura de código y el rendimiento de la prueba.
- En Visual Studio Enterprise Edition, con Live Unit Testing se pueden ver los resultados en vivo de las pruebas unitarias.
- Para determinar qué proporción de código del proyecto se está probando realmente mediante pruebas codificadas como pruebas unitarias, se puede utilizar la característica de cobertura de código de Visual Studio. Para restringir con eficacia los errores, las pruebas deberían ensayar una proporción considerable del código.

© JMA 2020. All rights reserved

Ejecutar pruebas unitarias



© JMA 2020. All rights reserved

Live Unit Testing

- Live Unit Testing ejecuta las pruebas unitarias automáticamente y en tiempo real a medida que se realizan cambios de código. Esto permite refactorizar y cambiar código con mayor confianza. Live Unit Testing ejecuta automáticamente todas las pruebas afectadas mientras se edita el código para asegurarse de que los cambios no introducen regresiones.
- Live Unit Testing indica si las pruebas unitarias cubren adecuadamente el código. Muestra gráficamente la cobertura de código en tiempo real. Se puede ver de un vistazo en el editor de código cuántas pruebas cubren cada línea de código y qué líneas no cubren las pruebas unitarias.
- Si la solución incluye uno o varios proyectos de prueba unitaria, se puede habilitar con Probar → Live Unit Testing → Iniciar.
- Live Unit Testing solo está disponible en Visual Studio Enterprise Edition.

© JMA 2020. All rights reserved

Ejecutar pruebas desde la línea de comandos

- VSTest.Console.exe es la herramienta de línea de comandos para ejecutar pruebas. Puede especificar varias opciones en cualquier orden en la línea de comandos. Sustituye a la antigua utilidad MsTest que se conserva por compatibilidad con versiones anteriores.
- Es necesario abrir un “Símbolo del sistema para desarrolladores” para usar la herramienta de línea de comandos.
- La sintaxis para ejecutar VSTest.Console.exe es:
 - `vstest.console.exe [TestFileNames] [Options] [/?]`
- Para ejecutar uno o varios proyectos de pruebas (separar los nombres de archivos con espacios o utilizar comodines)
 - `vstest.console.exe myTestFile.dll myOtherTestFile.dll moreTest*.dll`
- Para ejecutar pruebas concretas o filtrarlas:
 - `vstest.console.exe mytestproject.dll /Tests:MétodoPrueba1,MétodoPrueba2`
 - `vstest.console.exe mytestproject.dll /TestCaseFilter:"Priority=1"`
- Para listar las pruebas sin ejecutarlas:
 - `vstest.console.exe mytestproject.dll /ListTests`

© JMA 2020. All rights reserved

dotnet test

- El comando `dotnet test` se usa para ejecutar pruebas unitarias en una solución determinada. El `dotnet test` comando compila la solución y ejecuta una aplicación host de prueba para cada proyecto de prueba de la solución mediante VSTest.
- Es necesario abrir un “Símbolo del sistema para desarrolladores” para usar la herramienta de línea de comandos.
- Para ejecutar las pruebas en el proyecto en el directorio actual:
 - `dotnet test`
- Para ejecutar las pruebas en el proyecto `test1`:
 - `dotnet test ~/projects/test1/test1.csproj`
- Para ejecutar pruebas concretas o filtrarlas:
 - `dotnet test --filter TestCategory=smoke`
 - `dotnet test --filter FullyQualifiedName\!~IntegrationTests`
- Para ejecutar las pruebas en el proyecto en el directorio actual y generar un archivo de cobertura de código (solo en Windows):
 - `dotnet test --collect "Code Coverage"`

© JMA 2020. All rights reserved

Configuración de pruebas unitarias

- En Visual Studio las pruebas unitarias se pueden configurar mediante un archivo .runsettings. Por ejemplo, se puede cambiar la versión de .NET en la que se ejecutan las pruebas, el directorio de los resultados de las pruebas, los datos recopilados durante una serie de pruebas o personalizar el análisis de cobertura de código.
- Los parámetros de ejecución son opcionales. Si no es necesaria una configuración especial, no se necesita un archivo .runsettings.
- Los archivos de parámetros de ejecución se pueden usar para configurar pruebas que se ejecuten desde la línea de comandos, en el IDE o en un flujo de trabajo de compilación mediante Azure Test Plans o Team Foundation Server (TFS).
 - IDE:
 - Prueba > Seleccionar archivo de configuración
 - Herramientas > Opciones > Probar > Detectar archivos runsettings automáticamente
 - Línea de comandos:
 - `vstest.console.exe mytestproject.dll /Settings:mytestproject.runsettings`

<https://docs.microsoft.com/es-es/visualstudio/test/configure-unit-tests-by-using-a-dot-runsettings-file>

© JMA 2020. All rights reserved

PRUEBAS: MSTest

© JMA 2020. All rights reserved

Introducción

- MSTest, Microsoft Testing Framework, es un marco de pruebas para aplicaciones .NET. Permite escribir y ejecutar pruebas y proporcionar conjuntos de pruebas con integración en Visual Studio y Exploradores de pruebas de Visual Studio Code, la CLI de .NET y muchas canalizaciones de CI. MSTest es un marco de pruebas totalmente compatible, de código abierto y multiplataforma que funciona con todos los destinos de .NET compatibles (.NET Framework, .NET Core, .NET, .NET, UWP, WinUI, etc.). El marco de trabajo MSTest admite pruebas unitarias en Visual Studio, que se encuentra en el espacio de nombres Microsoft.VisualStudio.TestTools.UnitTesting.
- El espacio de nombres se incluye en una instrucción using en la parte superior del archivo de prueba unitaria y contiene muchas clases para ayudar con las pruebas unitarias:
 - Atributos para definir y documentar los casos de prueba.
 - Atributos de inicialización y limpieza para ejecutar código antes o después de ejecutar las pruebas unitarias, a fin de asegurarse un estado inicial y final concretos.
 - Clases Assert que se pueden utilizar para comprobar las condiciones en las pruebas unitarias.
 - El atributo ExpectedException para comprobar si se inicia determinado tipo de excepción durante la ejecución de la prueba unitaria.
 - La clase TestContext que almacena la información que se proporciona a las pruebas unitarias, como la conexión de datos para las pruebas controladas por datos y la información necesaria para ejecutar las pruebas unitarias para los servicios Web ASP.NET.

© JMA 2020. All rights reserved

Casos de pruebas

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

© JMA 2020. All rights reserved

Clases y métodos de prueba

- Clase de prueba: cualquier clase con el atributo [TestClass]. Sin TestClassAttribute, los métodos de prueba se omiten.
- Método de prueba: cualquier método marcado con el atributo [TestMethod].
- Método del ciclo de vida: cualquier método marcado con [ClassInitialize], [ClassCleanup], [TestInitialize] o [TestCleanup].
- Los métodos de prueba y los métodos del ciclo de vida pueden declararse localmente dentro de la clase de prueba actual, heredarse de otra clase de prueba que esté en el mismo ensamblado, no deben ser privados ni abstractos o devolver un valor.
- Al crear pruebas unitarias, se incluye una variable denominada testContextInstance para cada clase de prueba. Las propiedades de la clase TestContext almacenan información referente a la prueba actual.

© JMA 2020. All rights reserved

Fixtures

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

© JMA 2020. All rights reserved

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, se crea una nueva instancia de cada clase de prueba antes de ejecutar cada método de prueba.
- Se pueden usar los atributos siguientes para incluir la inicialización y la limpieza mediante:
 - [ClassInitialize]: para ejecutar el código antes de hacer la primera prueba en la clase.
 - [ClassCleanup]: para ejecutar el código cuando todas las pruebas de una clase se hayan ejecutado.
 - [TestInitialize]: para ejecutar el código antes de hacer cada prueba.
 - [TestCleanup]: para ejecutar el código cuando cada prueba se haya ejecutado.
- Se crean métodos marcados con el atributo [ClassInitialize] o [TestInitialize] para preparar aspectos del entorno en el que se ejecutará la prueba unitaria. El propósito de esto es establecer un estado conocido para ejecutar la prueba unitaria (por ejemplo, para copiar, modificar o crear algunos archivos de datos que la prueba utilizará).
- Se crean métodos marcados con el atributo [ClassCleanup] o [TestCleanup] para devolver el entorno a un estado conocido después de que se haya ejecutado una prueba (por ejemplo, la eliminación de archivos de carpetas o el retorno de una base de datos a un estado conocido).

© JMA 2020. All rights reserved

Ciclo de vida de instancia de prueba

```
[ClassInitialize]
public static void ClassInitializeMethod(TestContext context) {
    System.Diagnostics.Debug.WriteLine("ClassInitialize - Se ejecuta UNA SOLA VEZ por clase de test. ANTES de ejecutar ningún test.");
}

[TestInitialize]
public void TestInitializeMethod() {
    System.Diagnostics.Debug.WriteLine("TestInitializeMethod - Se ejecuta ANTES de cada test.");
}

[TestCleanup]
public void TestCleanupMethod() {
    System.Diagnostics.Debug.WriteLine("TestCleanupMethod - Se ejecuta DESPUÉS de cada test.");
}

[ClassCleanup]
public static void ClassCleanupMethod() {
    System.Diagnostics.Debug.WriteLine("ClassCleanupMethod - Se ejecuta UNA SOLA VEZ por clase de test. DESPUÉS de ejecutar todos los test.");
}
```

© JMA 2020. All rights reserved

TestContext

- La clase `TestContext` se utiliza en las pruebas unitarias con varios propósitos. Éstos son sus usos más frecuentes:
 - En cualquier prueba unitaria, porque la clase `TestContext` almacena la información que se proporciona a las pruebas unitarias, como la ruta de acceso al directorio de implementación.
 - En pruebas unitarias que prueban servicios Web que se ejecutan en un servidor de desarrollo de ASP.NET. En este caso, `TestContext` almacena la dirección URL del servicio Web.
 - En pruebas unitarias de ASP.NET, para obtener acceso al objeto `Page`.
 - En las pruebas unitarias controladas por datos, se requiere la clase `TestContext` porque proporciona acceso a la fila de datos.
- Al ejecutar una prueba unitaria, se proporciona automáticamente una instancia concreta del tipo `TestContext` si la clase de prueba tiene definida una propiedad `TestContext`. El marco de trabajo de pruebas unitarias rellena automáticamente los miembros de `TestContext` para que se utilicen durante todas las pruebas. Esto significa que no tiene que crear instancias o derivar el tipo `TestContext` en el código.

```
public TestContext TestContext { get; set; }
```

© JMA 2020. All rights reserved

Propiedades de TestContext

- **TestName:** Obtiene el nombre de la prueba.
- **FullyQualifiedTestClassFullName:** Obtiene el nombre completo de la clase que contiene el método de prueba en ejecución actualmente.
- **CurrentTestOutcome:** Resultado de una prueba que se ha ejecutado (para usar en un método `TestCleanup`).
- **Properties:** Obtiene las propiedades de prueba (`[TestProperty("MyProperty1", "Big")]`).
- **RequestedPage:** Obtiene la página solicitada.
- **DataConnection:** Obtiene la conexión de datos actual para pruebas controladas por datos.
- **DataRow:** Obtiene la fila de datos actual cuando la prueba se utiliza para pruebas controladas por datos.
- **TestRunDirectory:** Obtiene el directorio de nivel superior para la ejecución de pruebas que contiene archivos implementados y archivos de resultados.
- **TestResultsDirectory:** Obtiene el directorio de los archivos de resultado de la prueba.
- **TestRunResultsDirectory:** Obtiene el directorio de nivel superior para los archivos de resultados de la ejecución de pruebas (normalmente contiene el subdirectorio de `ResultsDirectory`).
- **DeploymentDirectory:** Obtiene el directorio de los archivos implementados para la ejecución de pruebas (normalmente contiene el subdirectorio de `TestRunDirectory`).
- **ResultsDirectory:** Obtiene el directorio de nivel superior que contiene resultados de pruebas y directorios de resultados de pruebas para la ejecución de pruebas (suele ser un subdirectorio de `TestRunDirectory`).

© JMA 2020. All rights reserved

Métodos de prueba

- Los marcos ofrecen una manera (normalmente a través de instrucciones assert o atributos method) de indicar si el método de prueba se ha superado o no. Otros atributos identifican métodos de configuración opcionales.
- El patrón AAA (Arrange, Act, Assert) es una forma habitual de escribir pruebas unitarias para un método en pruebas.
 - La sección Arrange de un método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.
 - La sección Act invoca al método en pruebas con los parámetros organizados.
 - La sección Assert comprueba si la acción del método en pruebas se comporta de la forma prevista.

© JMA 2020. All rights reserved

Métodos de prueba

- Se utilizan las clases Assert para comprobar la funcionalidad específica. Un método de prueba unitaria utiliza el código de un método en el código de la aplicación, pero solo notifica la corrección del comportamiento del código si se incluyen instrucciones Assert.
- Al ejecutarse las pruebas se marcan como:
 - Passed (Correcta): Se ha superado la prueba.
 - Failed (Con error): Fallo, no se ha superado la prueba por excepciones o afirmaciones fallidas..
 - Timeout (Con error): Fallo, superó el tiempo de espera de ejecución.
 - Inconclusive (Omitida): La prueba se ha completado, pero no podemos decir si pasó o falló porque no esta completa.
 - Error (Con error): Hubo un error del sistema mientras intentábamos ejecutar una prueba.
 - Aborted: La prueba fue abortada por el usuario.
 - InProgress: La prueba se está ejecutando actualmente.
 - NotRunnable: La prueba no se puede ejecutar.

© JMA 2020. All rights reserved

Aserciones

- Las aserciones siguen el patrón esperado, obtenido y, opcionalmente, mensaje asociado al error.
- Si se encadenan varias aserciones consecutivas, la primera que no se cumpla detendrá la prueba como fallida y no evaluará el resto.
- El espacio de nombres `Microsoft.VisualStudio.TestTools.UnitTesting` proporciona varios tipos de clases `Assert`:
 - `Assert`: En el método de prueba, se puede llamar a la cantidad de métodos de la clase `Assert` que se desee, como `Assert.AreEqual()`. La clase `Assert` tiene numerosos métodos entre los que elegir y muchos de esos métodos tienen varias sobrecargas.
 - `CollectionAssert`: Se utiliza la clase `CollectionAssert` para comparar colecciones de objetos y para comprobar el estado de una o más colecciones.
 - `StringAssert`: Se utiliza la clase `StringAssert` para comparar cadenas. Esta clase contiene una variedad de métodos útiles como `StringAssert.Contains`, `StringAssert.Matches` y `StringAssert.StartsWith`.

© JMA 2020. All rights reserved

Aserciones: Clase Assert

- `AreEqual`: Comprueba si los valores especificados son iguales y genera una `AssertFailedException` si no son iguales.
- `AreNotEqual`: Comprueba si los valores especificados son distintos y genera una `AssertFailedException` si son iguales.
- `AreSame`: Comprueba si los objetos especificados se refieren al mismo objeto y genera una `AssertFailedException` si las dos entradas no se refieren al mismo objeto.
- `AreNotSame`: Comprueba si los objetos especificados se refieren a diferentes objetos y lanza una `AssertFailedException` si las dos entradas se refieren al mismo objeto.
- `IsInstanceOfType`: Comprueba si el objeto especificado es una instancia del tipo esperado y genera una `AssertFailedException` si el tipo esperado no está en la jerarquía de herencia del objeto.
- `IsNotInstanceOfType`: Comprueba si el objeto especificado no es una instancia del tipo incorrecto y genera una `AssertFailedException` si el tipo especificado está en la jerarquía de herencia del objeto.
- `IsTrue`: Comprueba si la condición especificada es verdadera y genera una `AssertFailedException` si la condición es falsa.
- `IsFalse`: Comprueba si la condición especificada es falsa y lanza una `AssertFailedException` si la condición es verdadera.
- `IsNull`: Comprueba si el objeto especificado es nulo y lanza una `AssertFailedException` si no lo es.
- `IsNotNull`: Comprueba si el objeto especificado no es nulo y genera una `AssertFailedException` si es nulo.

© JMA 2020. All rights reserved

Aserciones: Excepciones

- Todos los métodos Assert lanzan la excepción `AssertFailedException` cuando no se cumple la aserción, que marca la prueba como “Con error”.
- Se puede implementar la comprobación manualmente, en cuyo caso se utiliza el método `Assert.Fail()` para marcar la prueba como “Con error”.
- Cuando el código del método de pruebas todavía no está completo, se puede utilizar el método `Assert.Inconclusive()` para generar la excepción `AssertInconclusiveException` y marcar la prueba como “Omitida”, salvo que una aserción anterior haya marcado la prueba como “Con error”.
- `Assert.ThrowsException` comprueba si el código especificado por un delegado arroja una excepción exacta dada del tipo especificado (y no de un tipo derivado) y arroja `AssertFailedException` si el código no arroja una excepción o arroja una excepción de tipo diferente al especificado.
- El método de prueba se puede anotar con `[ExpectedExceptionBase]` o con `[ExpectedException]` para comprobar si el código del método arroja una excepción dada del tipo especificado, en caso de no producirse marca la prueba como “Con error”.

© JMA 2020. All rights reserved

Aserciones: Clase `StringAssert`

- `Contains`: Comprueba si la cadena especificada contiene la subcadena especificada y genera una `AssertFailedException` si la subcadena no se produce dentro de la cadena de prueba.
- `StartsWith`: Comprueba si la cadena especificada comienza con la subcadena especificada y genera una `AssertFailedException` si la cadena de prueba no comienza con la subcadena.
- `EndsWith`: Comprueba si la cadena especificada termina con la subcadena especificada y genera una `AssertFailedException` si la cadena de prueba no termina con la subcadena.
- `Matches`: Comprueba si la cadena especificada coincide con una expresión regular y genera una `AssertFailedException` si la cadena no coincide con la expresión.
- `DoesNotMatch`: Comprueba si la cadena especificada no coincide con una expresión regular y genera una `AssertFailedException` si la cadena coincide con la expresión.

© JMA 2020. All rights reserved

Aserciones: Clase CollectionAssert

- **AllItemsAreInstancesOfType**: Comprueba si todos los elementos de la colección especificada son instancias del tipo esperado y genera una `AssertFailedException` si el tipo esperado no está en la jerarquía de herencia de uno o más de los elementos.
- **AllItemsAreNotNull**: Comprueba si todos los elementos de la colección especificada no son nulos y genera una `AssertFailedException` si algún elemento es nulo.
- **AllItemsAreUnique**: Comprueba si todos los elementos de la colección especificada son únicos o no y arroja una `AssertFailedException` si dos elementos de la colección son iguales.
- **AreEqual**: Comprueba si las colecciones especificadas son iguales y genera una `AssertFailedException` si las dos colecciones no son iguales. La igualdad se define como tener los mismos elementos en el mismo orden y cantidad. Diferentes referencias al mismo valor se consideran iguales.
- **AreNotEqual**: Comprueba si las colecciones especificadas son desiguales y genera una `AssertFailedException` si son iguales.
- **AreEquivalent**: Comprueba si dos colecciones contienen los mismos elementos y genera una `AssertFailedException` si alguna colección contiene un elemento que no está en la otra colección.
- **AreNotEquivalent**: Comprueba si dos colecciones contienen los diferentes elementos y lanza una `AssertFailedException` si las dos colecciones contienen elementos idénticos sin importar el orden.
- **Contains**: Comprueba si la colección especificada contiene el elemento especificado y genera una `AssertFailedException` si el elemento no está en la colección.
- **DoesNotContain**: Comprueba si la colección especificada no contiene el elemento especificado y genera una `AssertFailedException` si el elemento está en la colección.
- **IsSubsetOf**: Comprueba si una colección es un subconjunto de otra colección y genera una `AssertFailedException` si algún elemento del subconjunto no está también en el superconjunto.
- **IsNotSubsetOf**: Comprueba si una colección no es un subconjunto de otra colección y genera una `AssertFailedException` si todos los elementos del subconjunto también están en el superconjunto.

© JMA 2020. All rights reserved

Documentar los resultados

- Todos los métodos `Assert` permiten un parámetro con la personalización del mensaje de error.
- Por defecto, al ejecutar las pruebas, se muestran los nombres de las clases y los métodos de pruebas.
- Se dispone de atributos específicos para personalizar las plataformas de ejecución de pruebas:
 - `DescriptionAttribute`
 - `OwnerAttribute`
 - `PriorityAttribute`
 - `DeploymentItemAttribute`
 - `WorkItemAttribute`
 - `CssIterationAttribute`
 - `CssProjectStructureAttribute`

© JMA 2020. All rights reserved

Control de ejecución

- Se puede usar `TimeoutAttribute` para establecer un tiempo de espera en un método de prueba individual:

```
[TestMethod]  
[Timeout(2000)] // Milliseconds  
public void My_Test() {
```
- Para establecer el tiempo de espera en el máximo permitido:

```
[TestMethod]  
[Timeout(TestTimeout.Infinite)] // Milliseconds  
public void My_Test () {
```
- Se puede usar `IgnoreAttribute` para omitir la ejecución de determinados métodos de prueba:

```
[TestMethod]  
[Ignore]  
public void My_Test () {
```

© JMA 2020. All rights reserved

Rasgos

- Si se planea ejecutar estas pruebas como parte del proceso de automatización de pruebas, se puede considerar la posibilidad de crear la prueba en otro proyecto de prueba (y establecer los rasgos de las pruebas unitarias para la prueba unitaria).
- Esto le permite incluir o excluir más fácilmente estas pruebas específicas como parte de una integración continua o de una canalización de implementación continua.
- Mediante el `TestCategoryAttribute` se pueden categorizar (rasgos) los métodos de prueba para filtrar las ejecuciones.

```
[TestMethod]  
[TestCategory("Funcional")]  
public void My_Test () {
```

© JMA 2020. All rights reserved

Listas de reproducción personalizadas

- Se puede crear y guardar una lista de pruebas que se desea ejecutar o ver como grupo.
- Cuando se seleccione una lista de reproducción, las pruebas de la lista aparecerá en una nueva pestaña del Explorador de pruebas.
- Se puede agregar una prueba a más de una lista de reproducción.
- Para crear una lista de reproducción, se elijen una o varias pruebas en el Explorador de pruebas. Haciendo clic con el botón derecho y eligiendo Agregar a lista de reproducción > Nueva lista de reproducción.
- Se puede hacer clic en el botón Guardar en la barra de herramientas de la ventana de la lista de reproducción, seleccionar un nombre y una ubicación para guardar la lista de reproducción (con la extensión .playlist) y abrir la lista de reproducción posteriormente para repetir las mismas pruebas.

© JMA 2020. All rights reserved

Prueba unitaria con parámetros

- Una prueba unitaria con parámetros (PUT) es la generalización sencilla de una prueba unitaria mediante el uso de parámetros, permite la refactorización de varios métodos de pruebas de un caso de prueba: contiene las instrucciones sobre el comportamiento del código para todo un conjunto de valores de entrada posibles (parámetros), en lugar de solamente un único escenario (argumentos).
- Expresa suposiciones de la entradas (pre condiciones), ejecuta una secuencia de acciones y realiza aserciones de propiedades que se deben mantener en el estado final (post condiciones); es decir, sirve como la especificación. Esta especificación no requiere ni introduce ningún artefacto o elemento nuevo.

```
void TestAddHelper(ArrayList list, object element) {  
    Assert.IsNotNull(list);  
    list.Add(element);  
    Assert.IsTrue(list.Contains(element));  
}  
[TestMethod()]  
void TestAdd() {  
    var arrange = new ArrayList();  
    TestAddHelper(arrange, "uno");  
}
```

© JMA 2020. All rights reserved

Pruebas unitarias para métodos genéricos

- Los métodos de prueba no pueden ser métodos genéricos, para crear pruebas unitarias para métodos genéricos es necesario utilizar una técnica similar a las PUT.
- Hay que crear dos métodos: un método genérico asistente y un método de prueba.
- El argumento de tipo del método genérico asistente debe cumplir todas las restricciones de tipo que el método genérico a probar.

```
public void AddTestHelper<T>() {  
    T val = default(T);  
    MyLinkedList<T> target = new MyLinkedList<T>(val);  
    target.add<T>(val);  
    Assert.AreEqual(1, target.SizeOfLinkedList());  
}  
[TestMethod()]  
public void AddTest() {  
    AddTestHelper<GenericParameterHelper>();  
}
```

© JMA 2020. All rights reserved

Probar métodos no públicos

- Los métodos de prueba para cualquier método privado, protegido o interno es una tarea más difícil que para los métodos públicos, puesto que son inaccesibles directamente a través de las instancias y requiere un mejor conocimiento de las complejidades de la reflexión.

```
var arrange = new Tipo();  
MethodInfo privado = arrange.GetType().GetMethod( "MyPrivateMethod",  
    BindingFlags.NonPublic | BindingFlags.Instance);  
var rsIt = privado.Invoke(arrange, new object[] { 2, 3 });  
Assert.IsNotNull(rsIt);
```

- Para cargar tipos internos no accesibles desde fuera del ensamblado a probar:

```
var arrange = typeof(UnTipoPublico).Assembly.CreateInstance(  
    "Name.Space.InternalType");
```

© JMA 2020. All rights reserved

Probar métodos no públicos

- Se puede generar una prueba para un método privado. Esta generación crea una clase de descriptor de acceso privado, que crea una instancia de un objeto de la clase PrivateObject.
- La clase PrivateObject es una clase contenedora que utiliza la reflexión como parte del proceso de descriptor de acceso privado.

```
var calculadora = new Calculadora();  
var privateObject = new PrivateObject(calculadora);  
var obj = privateObject.Invoke("toDouble", 0.1 + 0.2);  
Assert.AreEqual(0.3, calculadora.toDouble(0.1 + 0.2));
```
- La clase PrivateType es similar, pero se utiliza para llamar a métodos estáticos privados en lugar de llamar a métodos de instancia privados.

© JMA 2020. All rights reserved

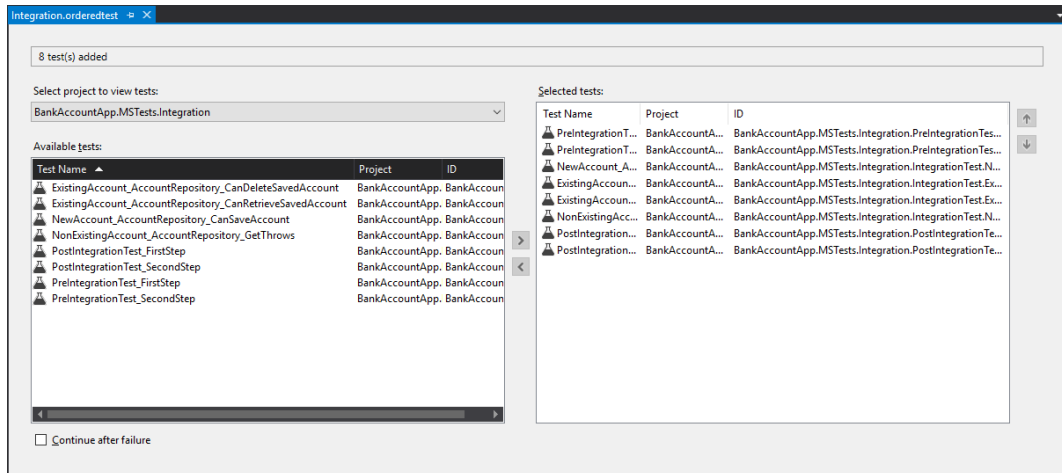
Pruebas ordenadas (Obsoletas)

MSTestv2 no es compatible con ordersTests.

- MSTest es el único marco de los tres marcos que tiene soporte incorporado para pruebas ordenadas.
- Las pruebas ordenadas se definen en un archivo .orderedtest, que es un archivo XML que contiene referencias a cada prueba. Este archivo se puede crear fácilmente desde el IDE de Visual Studio utilizando el editor visual, como se muestra a continuación.
- Puede especificar que una prueba ordenada se suspenda si una de las pruebas falla. Esto es útil si las pruebas tienen una dependencia funcional entre sí.
- Si las pruebas no tienen una dependencia funcional entre sí, pero una prueba no debe ejecutarse antes que la otra, puede dejar la opción deshabilitada.
- Las pruebas ordenadas aparecen en el "Explorador de pruebas" y se ejecutan como el resto de pruebas, pero se ven como un elemento único: no se muestran las pruebas 'secundarias' de las pruebas ordenadas.

© JMA 2020. All rights reserved

Pruebas ordenadas



© JMA 2020. All rights reserved

Pruebas genéricas

- Se pueden usar las pruebas genéricas para llamar a pruebas y programas externos. Después de hacer esto, el motor de la prueba trata la prueba genérica como cualquier otro tipo de prueba. Por ejemplo, puede ejecutar pruebas genéricas desde el Explorador de pruebas y obtener y publicar resultados de pruebas genéricas exactamente igual que lo hace con los demás tipos de pruebas.
- Se utiliza una prueba genérica para ajustar una prueba, un programa o una herramienta de otro fabricante ya existente que se comporta de la siguiente manera:
 - Puede ejecutarse desde una línea de comandos.
 - Devuelve un valor Sin errores o Con errores.
 - Opcionalmente, también devuelve resultados detallados de las pruebas “internas”, que son las pruebas que contiene.
- Visual Studio Enterprise trata las pruebas genéricas como a las otras pruebas y puede administrarlas y ejecutarlas mediante las mismas vistas, así como obtener y publicar sus resultados. Las pruebas genéricas son un modo sencillo de extensibilidad de Visual Studio.

© JMA 2020. All rights reserved

DATA DRIVEN TESTING

© JMA 2020. All rights reserved

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.
-

© JMA 2020. All rights reserved

Pruebas parametrizadas

- Los métodos de prueba pueden tener parámetros y hay disponibles varios atributos para indicar qué argumentos se suministran a las pruebas.
- Múltiples conjuntos de argumentos desencadenan la creación de múltiples pruebas. Todos los argumentos se crean en el punto de carga de las pruebas, por lo que resultados de los casos de prueba individuales se pueden consultar en los detalles del explorador de pruebas.
- Con el atributo [DataRow] se definen los datos insertados como argumentos en los parámetros de un método de prueba para un caso de prueba. El número de valores suministrados debe coincidir exactamente con el número de parámetros.
[TestMethod]
[DataRow(1, 6.2832)]
[DataRow(0.5, 3.1416)]
public void AreaDataTest(double radio, double expected) {
- Si no se supera uno de los caso de prueba se marca la prueba como no superada pero no detiene la ejecución del resto de los casos.

© JMA 2020. All rights reserved

Pruebas controladas por datos

- Mediante el marco de pruebas unitarias de Microsoft para código administrado, se puede configurar un método de prueba unitaria para recuperar los valores utilizados en el método de prueba de un origen de datos.
- El método se ejecuta para cada fila del origen de datos, lo que facilita probar una variedad de entrada mediante el uso de un único método.
- El origen de datos debe crearse antes de usarlo y vincularlo con el método de prueba y las propiedades. La fuente de datos puede tener diferentes formatos, como CSV, XML, Microsoft Access, Microsoft SQL Server Database o Oracle Database, o cualquier otra base de datos.
- Los ficheros origen de datos deben copiarse en el directorio de salida usando la ventana Propiedades o anotar el método con [DeploymentItem("ruta del fichero")].

© JMA 2020. All rights reserved

Método de prueba

- El atributo DataSource especifica la cadena de conexión para el origen de datos y el nombre de la tabla que se utiliza en el método de prueba. La información exacta de la cadena de conexión es diferente, dependiendo de qué tipo de origen de datos se está utilizando.

```
[TestMethod]
[DeploymentItem(@"..\..\data.csv")]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv",
"data#csv", DataAccessMethod.Sequential)]
public void My_Test() {
```

- Para obtener acceso a los datos de la tabla se utiliza el indizador DataRow del TestContext.
public TestContext TestContext { get; set; }
- DataRow es un objeto System.Data.DataRow, por lo que se recuperan valores de columna mediante los nombres de columna o índice. Dado que los valores se devuelven como objetos, es necesario convertirlos al tipo adecuado:
int x = Convert.ToInt32(TestContext.DataRow[0]);

© JMA 2020. All rights reserved

Prueba unitaria con parámetros

- Para el fichero CSV

```
radio,area
"0","0"
"0,5","3,1416"
"37","232,4779"
```

- Se realiza la prueba:

```
public TestContext TestContext { get; set; }

public void AreaTestHelper(double radio, double expected) {
    var arrange = new NuevoTipo();
    Assert.AreEqual(expected, Math.Round(arrange.Area(radio), 4));
}

[TestMethod]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
"|DataDirectory|\\AreaTest.csv", "AreaTest#csv", DataAccessMethod.Sequential)]
public void AreaTest() {
    AreaTestHelper(double.Parse(TestContext.DataRow[0].ToString()),
        double.Parse(TestContext.DataRow["area"].ToString()));
}
```

© JMA 2020. All rights reserved

Ejecutar la prueba y ver los resultados

- Cuando se ejecuta la prueba, se anima la barra de resultados de pruebas en la parte superior del explorador. Al final de la serie de pruebas, la barra será verde si todas las pruebas se completaron correctamente o roja si no alguna de las pruebas no lo hace. Un resumen de la ejecución de la prueba aparece en el panel de detalles de la parte inferior de la ventana Explorador de pruebas.
- Se produce un error en una prueba controlada por datos cuando ocurre un error en cualquiera de los métodos iterados con los datos de origen.
- Al elegir una prueba controlada por datos con errores en la ventana Explorador de pruebas, el panel de detalles muestra los resultados de cada iteración que se identifica mediante el índice de fila de datos.

© JMA 2020. All rights reserved

Tipos y atributos de origen de datos

- CSV
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential), DeploymentItem("data.csv"), TestMethod]
- Excel
DataSource("System.Data.Odbc", "Dsn=ExcelFiles;Driver={Microsoft Excel Driver (*.xls)};dbq=|DataDirectory|\\Data.xls;defaultdir=.;driverid=790;maxbufferize=2048;pagetimeout=5;readonly=true", "Sheet1\$", DataAccessMethod.Sequential), DeploymentItem("Sheet1.xls"), TestMethod]
- Caso de prueba de Team Foundation Server
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.TestCase", "http://vlm13261329:8080/tfs/DefaultCollection;Agile", "30", DataAccessMethod.Sequential), TestMethod]
- XML
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML", "|DataDirectory|\\data.xml", "Iterations", DataAccessMethod.Sequential), DeploymentItem("data.xml"), TestMethod]
- SQL Express
[DataSource("System.Data.SqlClient", "Data Source=\\sqllexpress;Initial Catalog=tempdb;Integrated Security=True", "Data", DataAccessMethod.Sequential), TestMethod]

© JMA 2020. All rights reserved

Configuración en app.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="microsoft.visualstudio.testtools" type="Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection,
      Microsoft.VisualStudio.TestPlatform.TestFramework.Extensions" />
  </configSections>
  <connectionStrings>
    <add name="MyExcelConn" connectionString="Dsn=Excel Files;dbq=data.xlsx;defaultdir=.;
      driverid=790;maxbufferSize=2048;pagetimeout=5" providerName="System.Data.Odbc" />
  </connectionStrings>
  <microsoft.visualstudio.testtools>
    <dataSources>
      <add name="MyExcelDataSource" connectionString="MyExcelConn" dataTableName="Sheet1$" dataAccessMethod="Sequential"/>
    </dataSources>
  </microsoft.visualstudio.testtools>
</configuration>

[DataSource("MyExcelDataSource")]
```

© JMA 2020. All rights reserved

<https://nunit.org/>

PRUEBAS: NUNIT

© JMA 2020. All rights reserved

Introducción

- NUnit es un marco de pruebas unitarias para todos los lenguajes .Net. Inicialmente portado desde JUnit, la versión de producción actual se ha reescrito por completo con muchas características nuevas y soporte para una amplia gama de plataformas .NET.
- Para obtener una copia de NUnit 3, puede usar varios enfoques de instalación.
 - Usa la plantilla de proyecto
 - Instalación completa de NUnit a través de NuGet.
 - Instalación de NUnitLite a través de NuGet.
 - Descarga de archivos Zip y / o MSI.
 - Enfoque combinado
- Hay disponible una serie de extensiones para Visual Studio:
 - NUnit VS Templates: Proporciona proyectos de Visual Studio y plantillas de elementos para NUnit 3 junto con fragmentos de código.
 - NUnit 3 Test Adapter: Adaptador NUnit 3 para ejecutar pruebas en Visual Studio.
 - Test Generator NUnit: Crea pruebas unitarias e Intellitests con NUnit 3.

© JMA 2020. All rights reserved

Casos de pruebas

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

© JMA 2020. All rights reserved

Clases y métodos de prueba

- Clase de prueba: cualquier clase con el atributo [TestFixture]. Sin TestFixtureAttribute, los métodos de prueba se omiten.
- Método de prueba: cualquier método marcado con el atributo [Test].
- Método del ciclo de vida: cualquier método marcado con [OneTimeSetUp], [OneTimeTearDown], [SetUp] o [TearDown].
- Los métodos de prueba y los métodos del ciclo de vida pueden declararse localmente dentro de la clase de prueba actual, heredarse de otra clase de prueba que esté en el mismo ensamblado, no deben ser privados ni abstractos o devolver un valor.
- Al crear pruebas unitarias, se incluye una variable denominada testContextInstance para cada clase de prueba. Las propiedades de la clase TestContext almacenan información referente a la prueba actual.

© JMA 2020. All rights reserved

Fixtures

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

© JMA 2020. All rights reserved

Ciclo de vida de instancia de prueba

- Se pueden usar los atributos siguientes para incluir la inicialización y la limpieza mediante:
 - [OneTimeSetUp]: para ejecutar el código antes de hacer la primera prueba en la clase.
 - [OneTimeTearDown]: para ejecutar el código cuando todas las pruebas de una clase se hayan ejecutado.
 - [SetUp]: para ejecutar el código antes de hacer cada prueba.
 - [TearDown]: para ejecutar el código cuando cada prueba se haya ejecutado.
- Se crean métodos marcados con el atributo [OneTimeSetUp] o [SetUp] para preparar aspectos del entorno en el que se ejecutará la prueba unitaria. El propósito de esto es establecer un estado conocido para ejecutar la prueba unitaria (por ejemplo, para copiar, modificar o crear algunos archivos de datos que la prueba utilizará).
- Se crean métodos marcados con el atributo [OneTimeTearDown] o [TearDown] para devolver el entorno a un estado conocido después de que se haya ejecutado una prueba (por ejemplo, la eliminación de archivos de carpetas o el retorno de una base de datos a un estado conocido).

© JMA 2020. All rights reserved

TestContext

- Cada prueba de NUnit se ejecuta en un contexto de ejecución, que incluye información sobre el entorno, así como la prueba en sí. La clase TestContext permite que las pruebas accedan a cierta información sobre el contexto de ejecución. La propiedad de clase CurrentContext da acceso al contexto actual.
- Las propiedades suministradas son:
 - Test: [ID, Name, FullName, MethodName, Properties]
 - Result.Outcome:
 - Status: Inconclusive | Skipped | Passed | Failed
 - Site: Test | SetUp | TearDown | Parent | Child
 - TestDirectory, WorkDirectory

© JMA 2020. All rights reserved

Métodos de prueba

- Los marcos ofrecen una manera (normalmente a través de instrucciones assert o atributos method) de indicar si el método de prueba se ha superado o no. Otros atributos identifican métodos de configuración opcionales.
- El patrón AAA (Arrange, Act, Assert) es una forma habitual de escribir pruebas unitarias para un método en pruebas.
 - La sección Arrange de un método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.
 - La sección Act invoca al método en pruebas con los parámetros organizados.
 - La sección Assert comprueba si la acción del método en pruebas se comporta de la forma prevista.

© JMA 2020. All rights reserved

Métodos de prueba

- Se utilizan las clases Assert para comprobar la funcionalidad específica. Un método de prueba unitaria utiliza el código de un método en el código de la aplicación, pero solo notifica la corrección del comportamiento del código si se incluyen instrucciones Assert. Al ejecutarse las pruebas se marcan como:
 - Success (Correcta): Se ha superado la prueba.
 - Inconclusive (Omitida): La prueba se ha completado, pero no podemos decir si pasó o falló porque no esta completa o no cumple las asunciones.
 - Failure (Con error): Fallo, no se ha superado la prueba por excepciones o afirmaciones fallidas.
 - Error (Con error): se produjo una excepción inesperada.
 - NotRunnable (Con error): La prueba no se puede ejecutar.
 - Cancelled (Omitida): La prueba fue abortada por el usuario.
 - Ignored (Omitida): La prueba esta marcada para que no se ejecute.
 - Skipped (Omitida): La prueba se omitió por alguna otra razón. .

© JMA 2020. All rights reserved

Aserciones

- Las aserciones son fundamentales para las pruebas unitarias, NUnit proporciona un amplio conjunto de aserciones como métodos estáticos de la clase Assert.
- Si falla una aserción, genera una excepción y se informa un error.
- NUnit presenta dos modelos de aserciones.
 - El modelo clásico usaba un método separado de la clase Assert para cada tipo aserción diferente (ya no se agregan nuevas características), siguen el patrón esperado, obtenido.
`Assert.AreEqual(4, 2+2);`
 - El modelo de restricción de aserciones utiliza el método `Assert.That` que toma objetos de restricción como argumento (es el mas ampliable y el recomendado).
`Assert.That(2+2, Is.EqualTo(4));`

© JMA 2020. All rights reserved

Modelo clásico de aserciones

- **Assert**
 - .True
 - .False
 - .Null
 - .NotNull
 - .Zero
 - .NotZero
 - .IsNaN
 - .IsEmpty
 - .IsNotEmpty
 - .AreEqual
 - .AreNotEqual
 - .AreSame
 - .AreNotSame
 - .Contains
 - .Greater
 - .GreaterOrEqual
 - .Less
 - .LessOrEqual
- **StringAssert**
 - .Contains
- **Positive**
 - .Negative
 - .InstanceOf
 - .IsNotInstanceOf
 - .IsAssignableFrom
 - .IsNotAssignableFrom
 - .Throws
 - .ThrowsAsync
 - .DoesNotThrow
 - .DoesNotThrowAsync
 - .Catch
 - .CatchAsync
 - .Pass
 - .Fail
 - .Ignore
 - .Inconclusive
- **CollectionAssert**
 - .AllItemsAreInstancesOfType
 - .AllItemsAreNotNull
 - .AllItemsAreUnique
 - .AreEqual
 - .AreEquivalent
 - .AreNotEqual
 - .AreNotEquivalent
 - .Contains
- **DoesNotContain**
 - .StartsWith
 - .DoesNotStartsWith
 - .EndsWith
 - .DoesNotEndWith
 - .AreEqualIgnoringCase
 - .AreNotEqualIgnoringCase
 - .IsMatch
 - .DoesNotMatch
- **DoesNotContain**
 - .IsSubsetOf
 - .IsNotSubsetOf
 - .IsEmpty
 - .IsNotEmpty
 - .IsOrdered
- **FileAssert**
 - .AreEqual
 - .AreNotEqual
 - .Exists
 - .DoesNotExist
- **DirectoryAssert**
 - .AreEqual
 - .AreNotEqual
 - .Exists
 - .DoesNotExist

© JMA 2020. All rights reserved

Modelo de restricción de aserciones

- El método `Assert.That()` admite el uso de restricciones como su segundo parámetro. Todas las restricciones están disponibles a través de las clases estáticas `Is`, `Has` y `Does`.
`Assert.That(actual, Is.EqualTo(expected));`
- Las restricciones se pueden combinar en expresiones fluidas utilizando los métodos incorporados `And`, `Or` y `With`. Las expresiones se pueden expandir convenientemente usando los muchos métodos en `ConstraintExpression`, como `AtMost` y `Contains`.
- Las afirmaciones grandes y fluidas se vuelven más difíciles de leer, pero cuando se combinan con clases que tienen buenas implementaciones de `ToString()`, pueden generar mensajes de error muy útiles.

```
[Test]
public void AdvancedConstraintsGiveUsefulErrorMessages() {
    Assert.That(actualCollection, Has
        .Count.EqualTo(4)
        .And.Exactly(1).Property("Age").GreaterThan(60)
        .And.Some.Property("Address").Null
        .And.No.Property("Age").LessThanOrEqualTo(17));
}
```

© JMA 2020. All rights reserved

Aserciones: Excepciones

- Todos los métodos `Assert` lanzan la excepción `AssertionException` cuando no se cumple la aserción, que marca la prueba como “Con error”.
- Se puede implementar la comprobación manualmente, en cuyo caso se utiliza el método `Assert.Fail()` para marcar la prueba como “Con error”.
- Cuando el código del método de pruebas todavía no está completo, se puede utilizar el método `Assert.Inconclusive()` para generar la excepción `InconclusiveException` y marcar la prueba como “Omitida”, salvo que una aserción anterior haya marcado la prueba como “Con error”.
- El método `Assert.Ignore` le brinda la capacidad de hacer que una prueba o suite sea ignorada dinámicamente en tiempo de ejecución, generando la excepción `IgnoreException` que marca la prueba como “Omitida”.
- El método `Assert.Pass` permite finalizar inmediatamente la prueba, grabándola como superada. Genera la excepción `SuccessException` que permite grabar un mensaje en el resultado de la prueba.

© JMA 2020. All rights reserved

Aserciones: Excepciones

- `Assert.ThrowsException` comprueba si el código especificado por un delegado arroja una excepción exacta dada del tipo especificado (y no de un tipo derivado) y arroja `AssertException` si el código no arroja una excepción o arroja una excepción de tipo diferente al especificado.
- `Assert.Catch` es similar a `Assert.Throws`, pero pasará por una excepción derivada de la especificada.
- `Assert.DoesNotThrow` verifica que el delegado proporcionado como argumento no arroje una excepción.
- El método de prueba se puede anotar con `[ExpectedException]` para comprobar si el código del método arroja una excepción dada del tipo especificado, en caso de no producirse marca la prueba como “Con error”.

© JMA 2020. All rights reserved

Agrupar aserciones

- Si falla una aserción, genera una excepción y se informa un error. Si una prueba contiene múltiples aserciones, no se ejecutará ninguna que siga a la que falló. Por esta razón, generalmente es mejor intentar una aserción por prueba.
- Pero a veces, es deseable continuar y acumular fallos adicionales para que todas puedan repararse a la vez. `Assert.Multiple` almacena los fallos encontrados en el bloque y los informa todos juntos al salir del bloque.

```
Assert.Multiple(() => {  
    Assert.AreEqual(5.2, result.RealPart, "Real part");  
    Assert.AreEqual(3.9, result.ImaginaryPart, "Imaginary part");  
});
```
- El bloque de aserción múltiple puede contener cualquier código arbitrario, no solo afirmaciones, incluso pueden estar anidados. La prueba finalizará de inmediato si se produce una excepción no controlada.

© JMA 2020. All rights reserved

Asunciones

- Las suposiciones tienen la intención de expresar el estado en el que debe estar una prueba para proporcionar un resultado significativo, las precondiciones.
- Son funcionalmente similares a las afirmaciones, sin embargo, una suposición no satisfecha producirá un resultado de prueba no concluyente (`InconclusiveException`), en lugar de fallida.
- La clase `Assume` dispone del método `That()` para fijar como restricciones las precondiciones que debe superar para continuar con la prueba:

```
Assume.That(myString, Is.EqualTo("Hello"));
```

© JMA 2020. All rights reserved

Advertencias

- A veces, especialmente en las pruebas de integración, es deseable dar un mensaje de advertencia pero continuar la ejecución de la prueba. `NUnit` lo admite esto con la clase `Warn` y el método `Assert.Warn`.

```
Warn.If(2+2 != 5);  
Warn.If(() => 2+2, Is.Not.EqualTo(5).After(3000));  
Assert.Warn("Warning message");
```
- Cada uno de los elementos anteriores fallaría. Sin embargo, la prueba continuará ejecutándose y los mensajes de advertencia solo se informarán al final de la prueba. Si la prueba falla posteriormente, se informarán las advertencias junto con el mensaje del fallo o mensajes en el caso de `Assert.Multiple`.

© JMA 2020. All rights reserved

Documentar los resultados

- Para la personalización del mensaje de error, los métodos Assert se pueden llamar sin un mensaje, con un mensaje de texto simple o con un mensaje y argumentos. En el último caso, el mensaje se formatea utilizando el texto y los argumentos proporcionados.
- Por defecto, al ejecutar las pruebas, se muestran los nombres de las clases y los métodos de pruebas.
- Se dispone de atributos específicos para personalizar las plataformas de ejecución de pruebas:
 - DescriptionAttribute
 - AuthorAttribute
 - PlatformAttribute

© JMA 2020. All rights reserved

Control de ejecución

- Se puede usar TimeoutAttribute para establecer un tiempo de espera en un método de prueba individual, se cancela la prueba:
`[Test, Timeout(2000)] // Milliseconds`
`public void My_Test() {`
- Se puede usar MaxTimeAttribute para establecer un tiempo de espera en un método de prueba individual, no cancela pero notifica prueba fallida:
`[Test, MaxTime(2000)] // Milliseconds`
`public void My_Test() {`
- Se puede usar IgnoreAttribute para omitir la ejecución de determinados métodos de prueba o TestFixture completas:
`[Test, Ignore("Razon por la que se omite")]`
`public void My_Test () {`

© JMA 2020. All rights reserved

Orden de ejecución de prueba

- Los casos de prueba individuales se ejecutan en el orden en que NUnit los descubre. Este orden no necesariamente sigue el orden léxico de los atributos y a menudo variará entre diferentes compiladores o diferentes versiones del CLR.
- El `OrderAttribute` puede ser colocado en un método de prueba o clase para especificar el orden en el que las pruebas se ejecutan. El orden se establece como un argumento `int` del atributo. Las pruebas con `[Order]` antes de cualquier prueba sin el atributo, en orden ascendente. Entre las pruebas con el mismo valor o sin el atributo, el orden de ejecución es indeterminado.

```
public class MyFixture {  
    [Test, Order(1)]  
    public void TestA() { ... }  
    [Test, Order(2)]  
    public void TestB() { ... }  
    [Test]  
    public void TestC() { ... }  
}
```

© JMA 2020. All rights reserved

Pruebas repetidas y reintentos

- `RepeatAttribute` se usa en un método de prueba para especificar que debe ejecutarse un número específico de veces. Si falla alguna repetición, las restantes no se ejecutan y se informa como prueba fallida.

```
[Test(), Repeat(5)]  
public void MyTest() {
```

- `RetryAttribute` se utiliza en un método de prueba para especificar que se debe volver a ejecutar si falla, hasta un número máximo de veces (el primer intento está incluido, `[Retry(1)]` no hace nada. Si una prueba genera una excepción inesperada, se devuelve un resultado de error y no se vuelve a intentar.

```
[Test(), Retry(3)]  
public void MyTest() {
```

© JMA 2020. All rights reserved

Rasgos

- Si se planea ejecutar estas pruebas como parte del proceso de automatización de pruebas, se puede considerar la posibilidad de crear la prueba en otro proyecto de prueba (y establecer los rasgos de las pruebas unitarias para la prueba unitaria).
- Esto le permite incluir o excluir más fácilmente estas pruebas específicas como parte de una integración continua o de una canalización de implementación continua.
- Mediante el `CategoryAttribute` se pueden categorizar (rasgos) los métodos de prueba para filtrar las ejecuciones.

```
[Test, Category("Funcional")]
public void My_Test () {
```
- También se puede crear un nuevo atributo que herede de `CategoryAttribute` y agrupar así los tests de la misma categoría.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class Critico : CategoryAttribute { }
```

© JMA 2020. All rights reserved

Prueba unitaria con parámetros

- Una prueba unitaria con parámetros (PUT) es la generalización sencilla de una prueba unitaria mediante el uso de parámetros, permite la refactorización de varios métodos de pruebas de un caso de prueba: contiene las instrucciones sobre el comportamiento del código para todo un conjunto de valores de entrada posibles (parámetros), en lugar de solamente un único escenario (argumentos).
- Expresa suposiciones de la entradas (pre condiciones), ejecuta una secuencia de acciones y realiza aserciones de propiedades que se deben mantener en el estado final (post condiciones); es decir, sirve como la especificación. Esta especificación no requiere ni introduce ningún artefacto o elemento nuevo.

```
void TestAddHelper(ArrayList list, object element) {
    Assert.IsNotNull(list);
    list.Add(element);
    Assert.IsTrue(list.Contains(element));
}
[Test()]
void TestAdd() {
    var arrange = new ArrayList();
    TestAddHelper(arrange, "uno");
}
```

© JMA 2020. All rights reserved

Pruebas parametrizadas

- NUnit admite pruebas parametrizadas. Los métodos de prueba pueden tener parámetros y hay disponibles varios atributos para indicar qué argumentos debe proporcionar NUnit a las pruebas.
- Múltiples conjuntos de argumentos desencadenan la creación de múltiples pruebas. Todos los argumentos se crean en el punto de carga de las pruebas, por lo que los casos de prueba individuales están disponibles para su visualización y selección en la interfaz gráfica de usuario, como pruebas individuales.
- Algunos atributos permiten especificar argumentos en línea, directamente en el atributo, mientras que otros usan un método, propiedad o campo separado para contener los argumentos.
- Algunos atributos se aplican al método de prueba completo y otros aplican a los parámetros del método de prueba. Los atributos que se aplican al método de prueba completo hacen innecesario el uso de [Test]. Hay disponibles atributos adicionales para establecer como se combinan los valores cuando se definen a nivel de parámetro.

© JMA 2020. All rights reserved

TestCaseAttribute

- TestCaseAttribute tiene el doble propósito de marcar un método con parámetros como método de prueba y proporcionar datos en línea que se utilizarán al invocar ese método:
[TestCase(1, 2)]
[TestCase(3, 4)]
[TestCase(5, 6, TestName = "Descripción del caso")]
public void Test(int num1, int num2) {
- Al utilizar el parámetro con nombre ExpectedResult, el conjunto de pruebas puede simplificarse aún más:
[TestCase(12, 2, ExpectedResult=6)]
[TestCase(12, 4, ExpectedResult=3)]
public int DivideTest(int n, int d) {

© JMA 2020. All rights reserved

TestCaseSourceAttribute

- TestCaseSourceAttribute se usa en un método de prueba parametrizado para identificar la fuente desde la cual se proporcionarán los argumentos requeridos. El atributo identifica adicionalmente el método como método de prueba. Los datos se mantienen separados de la prueba en sí y pueden ser utilizados por múltiples métodos de prueba.

```
public class MyTestClass
{
    static object[] DivideCases = {
        new object[] { 12, 3, 4 },
        new object[] { 12, 2, 6 },
        new object[] { 12, 4, 3 }
    };
    [TestCaseSource("DivideCases")]
    public void DivideTest(int n, int d, int rslt) {
```

© JMA 2020. All rights reserved

Valores de los argumentos

- El ValuesAttribute se utiliza para especificar un conjunto de valores a ser proporcionada por un parámetro individual de un método de ensayo con parámetros. Para booleanos y enumerados, si se pasan automáticamente todos los valores posibles si no se indican valores concretos.

```
[Test]
public void MyTest([Values(1, 2, 3)] int x, [Values("A", "B")] string s) {
```

- El RangeAttribute se utiliza para especificar un rango de valores a ser proporcionada por un parámetro individual de un método de ensayo con parámetros, indicando el valor inicial, el valor final y, opcionalmente, el delta.

```
[Test]
public void MyTest([Range(1, 10)] int x, [Range(0.2, 0.8, 0.2)] double d) {
```

© JMA 2020. All rights reserved

Valores de los argumentos

- El `RandomAttribute` se utiliza para especificar un conjunto de valores aleatorios, indicando el número de valores y, opcionalmente, el valor inicial y el valor final.

[Test]

```
public void MyTest([Random(10)] int x, [Random(0.2, 0.8, 10)] double d) {
```

- El `ValueSourceAttribute` se utiliza en parámetros individuales para identificar una fuente con nombre para los valores de argumento que se proporcionarán. La fuente puede ser un campo, una propiedad no indexada o un método sin argumentos, debe ser un miembro estático que devuelva un `IEnumerable` o un tipo que implemente `IEnumerable`.

```
static int[] MyCases = new int[] { 1, 3, 5, 7, 9 };
```

[Test()]

```
public void ValoresTest([ValueSource("MyCases")] int x) {
```

© JMA 2020. All rights reserved

Valores de los argumentos

- Dado que NUnit combina los datos proporcionados para cada parámetro en un conjunto de casos de prueba, se deben proporcionar datos todos los parámetros o para ninguno. No es necesario que coincidan la cantidad de valores.
- Por defecto, NUnit crea casos de prueba a partir de todas las combinaciones posibles de los puntos de datos proporcionados en los parámetros: el enfoque combinatorio (`CombinatorialAttribute`). Se puede modificar mediante atributos específicos en el propio método de prueba.
- El `PairwiseAttribute` se usa en una prueba para especificar que NUnit debería generar casos de prueba de tal manera que se usen todos los pares de valores posibles. Este es un enfoque bien conocido para combatir la explosión combinatoria de casos de prueba cuando están involucrados más de dos características (parámetros).
- El `SequentialAttribute` se usa en una prueba para especificar que NUnit debería generar casos de prueba seleccionando secuencialmente los elementos de datos individuales proporcionados para los parámetros de la prueba, sin generar combinaciones adicionales.

© JMA 2020. All rights reserved

<https://xunit.net/>

<https://github.com/xunit/samples.xunit>

PRUEBAS: XUNIT.NET

© JMA 2020. All rights reserved

Introducción

- NUnit fue inicialmente portado desde Junit, con las limitaciones que eso supuso. En 2007, sus autores (James Newkirk y Brad Wilson) vieron la necesidad de crear un nuevo marco de pruebas alineado más estrechamente con la plataforma .NET, que recogiera e impusiera algunos patrones muy claros de éxito (y limitar los de fracaso) y las nuevas tendencias en el mundo de las pruebas.
- Para obtener una copia de xUnit, puede usar varios enfoques de instalación.
 - Usa la plantilla de proyecto
 - Instalación completa de xUnit a través de NuGet.
- Hay disponible una serie de extensiones para Visual Studio:
 - xUnit Test Project Template: Proporciona proyectos de Visual Studio y plantillas de elementos para xUnit.
 - xUnit Test Code Snippets: Proporciona fragmentos de código para xUnit: xtestc fact dfact afact dafact theory dtheory attheory datheory xcdat xmdat.
 - xUnit.net.TestGenerator2022: Generador de pruebas xUnit.

© JMA 2020. All rights reserved

Casos de pruebas

- Los casos de prueba son clases que disponen de métodos para probar el comportamiento de una clase concreta. Así, por cada clase que se quiera probar se definirá su correspondiente clase de caso de prueba.
- Los casos de prueba se definen utilizando:
 - Anotaciones: Automatizan el proceso de definición, sondeo y ejecución de las pruebas.
 - Aserciones: Afirmaciones sobre lo que se esperaba y deben cumplirse para dar la prueba superada. Todas las aserciones del método deben cumplirse para superar la prueba. La primera aserción que no se cumpla detiene la ejecución del método y marca la prueba como fallida.
 - Clases de prueba: cualquier clase con métodos de prueba, no requiere un atributo específico.
 - Métodos de prueba: cualquier método marcado con el atributo [Fact] o [Theory].

© JMA 2020. All rights reserved

Montaje y desmontaje

- 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 prueba, podemos utilizar los llamados *fixtures*, que son elementos fijos que se crearán antes de ejecutar cada 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, se crea una nueva instancia de cada clase de prueba antes de ejecutar cada método de prueba.
- El equipo de xUnit.net considera que el montaje y el desmontaje de cada prueba genera un código de prueba difícil de seguir y de depurar, lo que a menudo hace que se ejecute código innecesario antes de que se ejecute cada prueba.

© JMA 2020. All rights reserved

Montaje y desmontaje

- xUnit.net crea una nueva instancia de la clase de prueba para cada prueba que se ejecuta por lo tanto, cualquier código que se coloque en el constructor de la clase de prueba se ejecuta para cada prueba. Esto hace que el constructor sea un lugar conveniente para colocar el código de configuración del contexto reusable cuando se desea compartir el código sin compartir instancias de objetos (es decir, obtiene una copia limpia de los objetos de contexto para cada prueba que se ejecuta). Para la limpieza del contexto, se agrega el interfaz `IDisposable` a la clase de prueba y en el método `Dispose()` se pone el código de limpieza. El método `Dispose` se invoca al terminar el método de prueba y antes de destruir la instancia.

```
public class StackTests : IDisposable {
    Stack<int> stack;

    public StackTests() {
        stack = new Stack<int>();
    }

    public void Dispose() {
        stack.Dispose();
    }
}
```

© JMA 2020. All rights reserved

IClassFixture

- A veces, la creación y limpieza del contexto de prueba puede ser muy costosa. Si hay que ejecutar el código de creación y limpieza durante cada prueba, podría realizar las pruebas más lentamente de lo deseable. Se puede usar la clase `IClassFixture` para compartir una sola instancia del objeto entre todas las pruebas en una clase de prueba. xUnit se asegurará de que la instancia de fixture se creará antes de que se ejecute cualquiera de las pruebas y, una vez todas las pruebas han terminado, limpiará el objeto fixture llamando `Dispose`, si está presente.

```
public class DatabaseFixture : IDisposable {
    public DatabaseFixture() {
        Db = new SqlConnection("MyConnectionString");
        // ... initialize data in the test database ...
    }

    public void Dispose() {
        // ... clean up test data from the database ...
    }

    public SqlConnection Db { get; private set; }
}

public class MyDatabaseTests : IClassFixture<DatabaseFixture> {
    DatabaseFixture fixture;
    public MyDatabaseTests(DatabaseFixture fixture) {
        this.fixture = fixture;
    }
}
```

© JMA 2020. All rights reserved

ICollectionFixture

- A veces querrá compartir un objeto fijo entre múltiples clases de prueba. Por ejemplo, es posible que se desee para inicializar una base de datos con un conjunto de datos de prueba, y luego dejar esa base de datos para su uso por múltiples clases de prueba. Se puede usar una `CollectionFixture` para compartir una sola instancia de objeto entre pruebas en varias clases de prueba. Se debe crear una clase `ICollectionFixture<>` vacía anotada con `[CollectionDefinition]` de definición de la colección, dándole un nombre único que identificará la colección de pruebas. Se añadirá la anotación `[Collection]` con el nombre establecido en `[CollectionDefinition]` a todas las clases de prueba que deban recibir el fixture compartido.

```
public class DatabaseFixture : IDisposable {  
    :  
}  
[CollectionDefinition("Database collection")]  
public class DatabaseCollection : ICollectionFixture<DatabaseFixture> {}  
[Collection("Database collection")]  
public class MyDatabaseTests {  
    DatabaseFixture fixture;  
    public MyDatabaseTests(DatabaseFixture fixture) {  
        this.fixture = fixture;  
    }  
    :  
    [Collection("Database collection")]  
    public class OtherDatabaseTests {
```

© JMA 2020. All rights reserved

El Hecho y la Teoría

- Los hechos son pruebas que siempre son ciertas. Prueban condiciones invariantes. Utilizamos `[Fact]` cuando tenemos algunos criterios que siempre deben cumplirse, independientemente de los datos. Los métodos de prueba no reciben parámetros por lo que el procedimiento de prueba coincide con el caso de prueba.
- Las teorías son pruebas que solo son ciertas para un conjunto particular de datos. `[Theory]`, por otro lado, depende del conjunto de parámetros y sus datos, la prueba pasará para un conjunto de datos y no para los demás. Tenemos la teoría que postula que solo en estos casos de prueba (conjunto de datos), superara la prueba. Los métodos de prueba reciben parámetros por lo que el procedimiento de prueba permite ejercitar múltiples casos de prueba. Es el equivalente a las pruebas parametrizadas o PUT de otros marcos de pruebas.

© JMA 2020. All rights reserved

Fact

- Los hechos son pruebas que siempre son ciertas. Son métodos anotados con [Fact] que no reciben parámetros y no devuelven valor (void) o devuelven una tarea (Task) en las pruebas asíncronas. La anotación Fact acepta:
 - DisplayName: Descripción del caso de prueba que sustituye al nombre del método en los resultados.
 - Skip: omite la ejecución con un literal que expresa el motivo.
 - Timeout: establece en milisegundos el tiempo de espera máximo para que termine el método.

```
[Fact(DisplayName = "Suma dos valores reales")]  
public void ShouldAddDoubleValues() {  
    var sut = new Calculator();  
  
    double result = sut.Add (0.1, 0.2);  
  
    Assert.Equal(0.3, result, precision: 1);  
}
```

© JMA 2020. All rights reserved

Theory

- Las teorías son pruebas que solo son ciertas para un conjunto particular de datos. Son métodos anotados con [Theory] que reciben parámetros y no devuelven valor (void) o devuelven una tarea (Task) en las pruebas asíncronas. La anotación Theory acepta: DisplayName, Skip y Timeout.
- Múltiples conjuntos de argumentos desencadenan la creación de múltiples pruebas. Todos los argumentos se crean en el punto de carga de las pruebas, por lo que resultados de los casos de prueba individuales se pueden consultar en los detalles del explorador de pruebas.
- Los métodos anotados con [Theory] esperan una o más instancias de [InlineData], [MemberData] o [ClassData] que proporcionen los valores para los argumentos del método. El método se ejecuta para cada conjunto de datos (caso de prueba) de forma independiente.
- Se pueden combinar diferentes orígenes [InlineData], [MemberData] y [ClassData] para la misma teoría pero se ignorarán los casos repetidos.

© JMA 2020. All rights reserved

Theory: InlineData

- Con los atributos [InlineData] se definen los datos insertados como argumentos en los parámetros de un método de prueba para un caso de prueba. El número de valores suministrados debe coincidir exactamente con el número y tipo de parámetros. A ser particulares de la teorías, no se pueden reutilizar en otras teorías.

```
[Theory(DisplayName = "Suma dos valores enteros")]
[InlineData(1, 2, 3)]
[InlineData(int.MinValue, -1, int.MaxValue)]
public void ShouldAddIntegerValues(int valueOne, int valueTwo, int expected) {
    var sut = new Calculator();
    double result = sut.Add (valueOne, valueTwo);
    Assert.Equal(expected, result);
}
```

© JMA 2020. All rights reserved

Theory: MemberData

- Con el atributo [MemberData] se establece el método estático o la propiedad estática IEnumerable<object[]> que proporciona los datos de prueba.

```
public static IEnumerable<object[]> GetTestCases() => new object[][] {
    new object[] { 1, 2, 3 },
    new object[] { int.MinValue, -1, int.MaxValue }
};

[Theory(DisplayName = "Suma dos valores enteros")]
[MemberData(nameof(GetTestCases))]
public void ShouldAddIntegerValues(int valueOne, int valueTwo, int expected) {
    var sut = new Calculator();

    double result = sut.Add (valueOne, valueTwo);

    Assert.Equal(expected, result);
}
```

© JMA 2020. All rights reserved

Theory: ClassData

- Con el atributo [ClassData] se establece la clase IEnumerable<object[]> cuyas instancias proporcionan los datos de prueba.

```
public class AddTestData : IEnumerable<object[]> {  
    public IEnumerator<object[]> GetEnumerator() {  
        yield return new object[] { 1, 2, 3 };  
        yield return new object[] { int.MinValue, -1, int.MaxValue };  
    }  
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();  
    //IEnumerator IEnumerable.GetEnumerator() => new  
        List<object[]>(File.ReadAllLines("foo.csv")).GetEnumerator();  
}  
  
[Theory(DisplayName = "Suma dos valores enteros")]  
[ClassData(typeof(AddTestData))]  
public void ShouldAddIntegerValues(int valueOne, int valueTwo, int expected) {
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Custom DataAttribute

```
public class CsvFileDataAttribute : DataAttribute {
    private readonly string _csvFileName;
    public CsvFileDataAttribute(string csvFileName) {
        _csvFileName = csvFileName;
    }
    public override IEnumerable<object[]> GetData(MethodInfo testMethod) {
        string[] csvLines = File.ReadAllLines(_csvFileName);
        var testCases = new List<object[]>();
        foreach (var csvLine in csvLines) {
            IEnumerable<int> values = csvLine.Split(',').Select(int.Parse);
            object[] testCase = values.Cast<object>().ToArray();
            testCases.Add(testCase);
        }
        return testCases;
    }
}

[Theory(DisplayName = "Suma dos valores enteros")]
[CsvFileData("testdata.csv")]
public void ShouldAddIntegerValues(int valueOne, int valueTwo, int expected) {
```

© JMA 2020. All rights reserved

Métodos de prueba

- Los marcos ofrecen una manera (normalmente a través de instrucciones assert) de indicar si el método de prueba se ha superado o no. Otros atributos identifican métodos de configuración opcionales.
- El patrón AAA (Arrange, Act, Assert) es una forma habitual de escribir pruebas unitarias para un método en pruebas.
 - La sección Arrange de un método de prueba unitaria inicializa objetos y establece el valor de los datos que se pasa al método en pruebas.
 - La sección Act invoca al método en pruebas con los parámetros organizados.
 - La sección Assert comprueba si la acción del método en pruebas se comporta de la forma prevista.

© JMA 2020. All rights reserved

Auto validar la prueba

- Se utilizan las clases Assert para comprobar la funcionalidad específica. Un método de prueba unitaria utiliza el código de un método en el código de la aplicación, pero solo notifica la corrección del comportamiento del código si se incluyen instrucciones Assert. Al ejecutarse las pruebas se marcan como:
 - Success (Correcta): Se ha superado la prueba.
 - Failure (Con error): Fallo, no se ha superado la prueba por excepciones o afirmaciones fallidas.
 - Error (Con error): se produjo una excepción inesperada.
 - Skipped (Omitida): La prueba se omitió por alguna otra razón. .

© JMA 2020. All rights reserved

Aserciones

- Las aserciones son fundamentales para las pruebas unitarias, xUnit proporciona un amplio conjunto de aserciones como métodos estáticos de la clase Assert.
- Si falla una aserción, se genera una excepción y se informa que falla la prueba. Para la personalización del mensaje de error, algunos métodos Assert admiten un ultimo argumento con un mensaje de texto.
- La mayoría de las aserciones son procedimientos que no devuelven valor (void). Algunas aserciones cuentan con una versión asíncrona, con el sufijo Async, que devuelven una tarea (Task). Si se propaga como devolución del método de prueba, se considera una prueba asíncrona.
- Se puede implementar la comprobación manualmente, en cuyo caso se utiliza el método Assert.Fail(msg) para marcar la prueba como “Fallida”. Al no ser una aserción específica es conveniente indicar el motivo.
- xUnit presenta un amplio juego de aserciones que se puede ampliar.

© JMA 2020. All rights reserved

Aserciones

- Booleanos
 - False
 - True
- Numéricos
 - *Equal*
 - NotEqual
 - InRange
 - NotInRange
- DateTime
 - *Equal*
- Cadenas
 - Empty
 - *Equal*
 - Contains
 - DoesNotContain
 - StartsWith
 - EndsWith
 - Matches
 - DoesNotMatch
- Colecciones y Conjuntos
 - All
 - AllAsync
 - Collection
 - Contains
 - DoesNotContain
 - *Equal*
 - NotEqual
 - Distinct
 - Empty
 - NotEmpty
 - Single
 - Subset
 - ProperSubset
- Excepciones
 - Throws
 - ThrowsAsync
 - ThrowsAny
 - ThrowsAnyAsync
- Eventos
 - Raises
 - RaisesAsync
 - RaisesAny
 - RaisesAnyAsync
 - PropertyChanged
 - PropertyChangedAsync
- Tipos y referencias
 - Null
 - NotNull
 - Same
 - NotSame
 - StrictEqual
 - NotStrictEqual
 - IsType
 - IsNotType
 - IsAssignableFrom
 - IsNotAssignableFrom

© JMA 2020. All rights reserved

Aserciones

```
Assert.Collection(fruits,
    item => Assert.IsType<Orange>(item),
    item => Assert.IsType<Apple>(item),
    item => Assert.IsType<Grape>(item),
    item => {
        Assert.IsType<Banana>(item);
        Assert.True(((Banana)item).Override);
    }
);

[Fact]
public void
ProfileShouldNotifyOfStatusMessageChanges() {
    Profile testProfile = new Profile();
    Assert.PropertyChanged(testProfile, "StatusMessage", () => testProfile.StatusMessage = "Hard at work");
}

[Fact]
public void
EmailerShouldRaiseEmailSentWhenSendingEmails() {
    string address = "test@test.com";
    string body = "this is a test";
    Emailer emailer = new Emailer();
    Assert.Raises<EmailSentEventArgs>(
        listener => emailer += listener,
        listener => emailer -= listener,
        () => {
            emailer.SendEmail(address, body);
        }
    );
}
```

© JMA 2020. All rights reserved

Aserciones: Excepciones

- Todos los métodos Assert lanzan una excepción personalizada heredera de `XunitException` cuando no se cumple la aserción, lo que marca la prueba como “Fallida”. Si se implementa la comprobación manualmente, se utiliza el método `Assert.Fail()` para lanzar la excepción que marca la prueba como “Fallida”.
- Si se produce una excepción no controlada, que no sea lanzada por Assert, se rompe la prueba y se marcar la prueba como “Con error”.
- El lanzamiento de excepciones es un aspecto esencial del desarrollo de software, ya que ayuda a administrar el flujo de control en una aplicación y manejar escenarios excepcionales. Por lo tanto, es necesario verificar que los casos inválidos lanzan las correspondientes excepciones sin romper la prueba.
- `Assert.Throws` comprueba si el código especificado por un delegado arroja una excepción exacta dada del tipo especificado (y no de un tipo derivado) y falla la prueba si el código no arroja una excepción o arroja una excepción de tipo diferente al especificado.
`Assert.Throws<DivideByZeroException>(() => calculadora.Divide(1, 0));`
- `Assert.ThrowsAny` es similar a `Assert.Throws`, pero pasará también con una excepción derivada de la especificada.
`Assert.ThrowsAny<ArithmeticException>(() => calculadora.Divide(1, 0));`

© JMA 2020. All rights reserved

Aserciones: Excepciones

- Es importante tener en cuenta que `Assert.Throws` y `Assert.ThrowsAny` solo afirman que se lanza una excepción del tipo especificado. Sin embargo, no comprueba el mensaje u otras propiedades de la excepción.
- Dado que el mismo tipo de excepción se puede lanzar en cualquier punto de la pila de llamadas y por diferentes motivos, a veces es necesario validar las propiedades de la excepción, por lo que `Assert.Throws` y `Assert.ThrowsAny` devuelven la excepción producida.
`var ex = Assert.Throws<DivideByZeroException>(() => calculadora.Divide(1, 0));
Assert.Equal("Intento de dividir por cero.", ex.Message);`
- Para separar la actuación de la afirmación, xUnit suministra `Record.Exception` para capturar la excepción sin necesidad de utilizar un `try/catch`.
`var ex = Record.Exception(() => {
 calculadora.Divide(1, 0);
});
Assert.NotNull(ex);
Assert.IsType<DivideByZeroException>(ex);
Assert.Equal("Intento de dividir por cero.", ex.Message);`

© JMA 2020. All rights reserved

Agrupar aserciones

- Si falla una aserción, genera una excepción y se informa un error. Si una prueba contiene múltiples aserciones, no se ejecutará ninguna que siga a la que falló. Por esta razón, generalmente es mejor intentar una aserción por prueba.
- Pero a veces, es deseable continuar y acumular fallos adicionales para que todas puedan repararse a la vez. `Assert.Multiple` almacena los fallos encontrados en el bloque y los informa todos juntos al salir de la aserción.

```
Assert.NotNull(actual);
Assert.Multiple(
    () => Assert.Equal(5.2, actual.RealPart, "Real part"),
    () => Assert.Equal(3.9, actual.ImaginaryPart, "Imaginary part")
);
```
- Las afirmaciones anidadas deben pasarse como acciones (delegados) a `Multiple` para que no fallen inmediatamente.

© JMA 2020. All rights reserved

ITestOutputHelper

- Para ayudar a depurar la prueba de falla (especialmente cuando se ejecutan en máquinas remotas sin acceso a un depurador), a menudo puede ser útil agregar salida de diagnóstico que está separada de los resultados de las pruebas que pasan o fallan.
- Las pruebas unitarias tienen acceso a la interfaz `ITestOutputHelper` que reemplaza el uso de `Console` y mecanismos similares. Para aprovechar esto, simplemente se agrega un argumento `ITestOutputHelper` al constructor para poder usarlo en las prueba unitarias, `Dispose`, ...

```
public class MyTestClass {
    private readonly ITestOutputHelper output;
    public MyTestClass(ITestOutputHelper output) {
        this.output = output;
    }
    [Fact]
    public void MyTest() {
        var temp = this.GetType().Name;
        output.WriteLine("This is output from {0}", temp);
    }
}
```

© JMA 2020. All rights reserved

Rasgos

- Si se planea ejecutar las pruebas como parte del proceso de automatización de pruebas, se puede considerar la posibilidad de crear varios proyectos de prueba o clasificar mediante rasgos las pruebas unitarias. Esto permite incluir o excluir más fácilmente las pruebas específicas como parte de una integración continua o de una canalización de implementación continua.
- Mediante el `TraitAttribute` se pueden agregar metadatos a las pruebas (rasgos) los ensamblados, clases y métodos de prueba para filtrar las ejecuciones. El primer parámetro es la clave de los metadatos, el segundo parámetro es el valor que acompaña a la clave.
`[Fact, Trait("Category", "functional"), Trait("Category", "smoke"), Trait("Type", "e2e")]`
`public void My_Test () {`
- Dado que las cadenas son susceptibles de errores tipográficos, es recomendable crear atributos personalizados que establezcan las cadena.

© JMA 2020. All rights reserved

Rasgos

```
public class CategoryDiscoverer : ITraitDiscoverer {
    public IEnumerable<KeyValuePair<string, string>> GetTraits(IAttributeInfo traitAttribute) {
        var name = (traitAttribute as ReflectionAttributeInfo).Attribute.GetType().Name.Replace("CategoryAttribute", "");
        var value = traitAttribute.GetConstructorArguments().FirstOrDefault()?.ToString() ?? name;
        yield return new KeyValuePair<string, string>("Category", value.ToLower());
    }
}

[TraitDiscoverer("Demos.CategoryDiscoverer", "Utilidades.xUnit.Tests")]
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]
public class CategoryAttribute : Attribute, ITraitAttribute {
    public CategoryAttribute(string category) { }
}

public class SmokeCategoryAttribute : CategoryAttribute { public SmokeCategoryAttribute() : base("smoke") { } }
public class FunctionalCategoryAttribute : CategoryAttribute { public FunctionalCategoryAttribute() : base("functional") { } }

[Fact, SmokeCategory, Category("functional")]
public void My_Test () {
```

© JMA 2020. All rights reserved

FRAMEWORKS COMPLEMENTARIOS

© JMA 2020. All rights reserved

AutoFixture

- [AutoFixture](#) es una biblioteca de código abierto para .NET diseñado para minimizar la fase 'Arrange' de las pruebas unitarias con el fin de maximizar la mantenibilidad. Su objetivo principal es permitir a los desarrolladores centrarse en lo que se está probando en lugar de cómo configurar el escenario de prueba, facilitando la creación de gráficos de objetos que contengan datos de prueba.
- Al escribir pruebas unitarias, normalmente necesita crear algunos objetos que representen el estado inicial de la prueba. A menudo, una API te obligará a especificar muchos más datos de los que realmente te importan (dummy's), por lo que con frecuencia terminas creando y manteniendo objetos que no tienen influencia en la prueba, simplemente para hacer que el código se compile.

```
var fixture = new Fixture();
var request = fixture.Create<int>();
var person = fixture.Create<Person>();
[Theory, AutoData]
public void IntroductoryTest(int request, Person person) { ... }
[Theory, AutoData]
public void IntroductoryTest(int request, Person person) { ... }
```
- Disponible para NUnit y xUnit:
 - dotnet add package AutoFixture AutoFixture.Xunit2 AutoFixture.NUnit3

© JMA 2020. All rights reserved

Fluent Assertions

- Fluent Assertions es un conjunto de métodos de extensión .NET que le permiten especificar de forma más natural el resultado esperado de una prueba unitaria de estilo TDD o BDD. Esto permite una sintaxis fluida mas intuitiva y simple.

```
string actual = "ABCDEFGHI";  
actual.Should().StartWith("AB").And.EndWith("HI").And.Contain("EF").And.HaveLength(9);
```
- Fluent Assertions admite muchos marcos de prueba unitarios diferentes. Simplemente hay que agregar una referencia a Fluent Assertions al correspondiente proyecto de prueba unitarios. Fluent Assertions encontrará automáticamente el ensamblado correspondiente al marco de prueba y lo usará para lanzar las excepciones específicas del marco.
 - dotnet add package FluentAssertions

© JMA 2020. All rights reserved

Pruebas de instantáneas

- La prueba de instantáneas es una poderosa técnica que permite comparar la salida del código con un valor almacenado previamente, también conocido como instantánea (snapshot) o, coloquialmente, una foto. De esta manera, se crea un bucle de retroalimentación rápida para detectar fácilmente cualquier cambio o regresión en el código sin tener que escribir afirmaciones complejas. Aunque cubren un amplio espectro, al depender del interfaz, son pruebas frágiles.
- Las pruebas de instantáneas generalmente almacenan una salida generada previamente como un archivo separado. Cuando se vuelve a ejecutar el caso de prueba, compara la nueva salida (local, no incluida en git) con el archivo almacenado (que se incluye en git). Si la nueva salida coincide con la salida existente, la prueba pasará, de lo contrario la prueba fallará. Cuando este último es el caso, puede actualizar la instantánea si se desea el cambio o investigar el cambio(s) si la salida es inesperada.
- Para las pruebas de instantáneas se puede utilizar la herramienta [Verify](#). Verify es una biblioteca simple que se puede usar como una herramienta independiente, o puede usar una de sus integraciones opt-in con marcos de prueba populares, como NUnit, xUnit y MSTest.

© JMA 2020. All rights reserved

Verify (MSTest)

- Para escribir una prueba de instantánea con Verify, se usa uno de los métodos Verify dentro de la prueba para devolver el modelo bajo prueba. De forma predeterminada, utiliza el nombre del método completamente calificado del caso de prueba para crear el archivo de salida existente (clase.metodo.verified.*) y el archivo recién generado (*.received.*). Cada vez que se ejecuta el caso de prueba, el archivo recibido se vuelve a crear y se compara con el archivo verificado.

[TestClass, **UsesVerify**] // Verify requires test classes to opt in to being processed by the Source Generator

```
public partial class ApprovalTest {  
    [TestMethod]  
    public Task ThirtyDays() {  
        var fakeOutput = new StringBuilder();  
        Console.SetOut(new StringWriter(fakeOutput));  
        Console.SetIn(new StringReader($"a{Environment.NewLine}"));  
        Program.Main(new string[] { "30" });  
        var output = fakeOutput.ToString();  
        return Verifier.Verify(output); // ApprovalTest.ThirtyDays.verified.txt  
    }  
}
```

Instalación:

- dotnet add package Verify.MSTest

© JMA 2020. All rights reserved

Verify (Nunit)

- Para escribir una prueba de instantánea con Verify, se usa uno de los métodos Verify dentro de la prueba para devolver el modelo bajo prueba. De forma predeterminada, utiliza el nombre del método completamente calificado del caso de prueba para crear el archivo de salida existente (clase.metodo.verified.*) y el archivo recién generado (*.received.*). Cada vez que se ejecuta el caso de prueba, el archivo recibido se vuelve a crear y se compara con el archivo verificado.

[TestFixture]

```
public class ApprovalTest {  
    [Test]  
    public Task ThirtyDays() {  
        var fakeOutput = new StringBuilder();  
        Console.SetOut(new StringWriter(fakeOutput));  
        Console.SetIn(new StringReader($"a{Environment.NewLine}"));  
        Program.Main(new string[] { "30" });  
        var output = fakeOutput.ToString();  
        return Verifier.Verify(output); // ApprovalTest.ThirtyDays.verified.txt  
    }  
}
```

Instalación:

- dotnet add package Verify.NUnit

© JMA 2020. All rights reserved

Verify (XUnit)

- Para escribir una prueba de instantánea con Verify, se usa uno de los métodos Verify dentro de la prueba para devolver el modelo bajo prueba.
- De forma predeterminada, utiliza el nombre del método completamente calificado del caso de prueba para crear el archivo de salida existente (clase.metodo.verified.*) y el archivo recién generado (*.received.*). Cada vez que se ejecuta el caso de prueba, el archivo recibido se vuelve a crear y se compara con el archivo verificado.

```
public class ApprovalTest {  
    [Fact]  
    public Task ThirtyDays() {  
        var fakeOutput = new StringBuilder();  
        Console.SetOut(new StringWriter(fakeOutput));  
        Console.SetIn(new StringReader($"a{Environment.NewLine}"));  
        Program.Main(new string[] { "30" });  
        var output = fakeOutput.ToString();  
        return Verifier.Verify(output); // ApprovalTest.ThirtyDays.verified.txt  
    }  
}
```

Instalación:

- `dotnet add package Verify.XunitV3`

© JMA 2020. All rights reserved

Visual Studio Enterprise

MICROSOFT FAKES

© JMA 2020. All rights reserved

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.
- Usar dobles de prueba (como los dobles en el cine) tiene ventajas:
 - Devuelven resultados determinísticos
 - Permiten crear o reproducir determinados estados
 - 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.
 - Memorizan los valores con los que se llama a cada uno de sus miembros
 - Permiten verificar si los valores esperados coinciden con los recibidos

© JMA 2020. All rights reserved

Simulación de objetos

- **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 preprogramado 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.

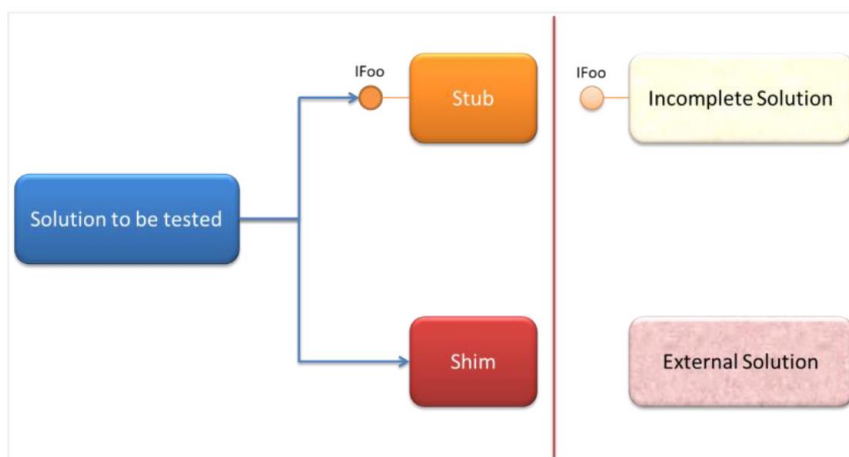
© JMA 2020. All rights reserved

Microsoft Fakes

- Microsoft Fakes es un framework de aislamiento que nos permite aislar el código a testear reemplazando otras partes de la aplicación con stubs o shims. Nos permite testear partes de nuestra solución incluso si otras partes de nuestra aplicación no han sido implementadas o aún no funcionan.
- Microsoft Fakes viene con dos sabores:
 - Stubs (código auxiliar): reemplaza a una clase por un pequeño sustituto que implementa la misma interfaz. Para utilizar código auxiliar, tiene que diseñar la aplicación para que cada componente dependa únicamente de interfaces y no de otros componentes.
 - Shims (corrección de compatibilidad): intercepta el código compilado de la aplicación en tiempo de ejecución para que, en lugar de realizar una llamada de método especificada, ejecute el código shim que proporciona la prueba. Las correcciones de compatibilidad (shims) se pueden usar para reemplazar las llamadas a ensamblados que no se pueden modificar, como los ensamblados .NET.

© JMA 2020. All rights reserved

Escenarios



© JMA 2020. All rights reserved

Stubs

- El código auxiliar es un fragmento de código que ocupa el lugar de otro componente durante las pruebas. La ventaja de utilizar código auxiliar es que devuelve resultados coherentes, haciendo que la prueba sea más fácil de escribir. Y se pueden ejecutar pruebas aun cuando los otros componentes no estén creados todavía.
- Los stubs son dobles de prueba que implementan parcialmente un interfaz explícito o un interfaz implícito extraído de una clase. Los miembros no implementados quedan sin definir y no deben ser usados.
- Para utilizar código auxiliar, hay que escribir el código que se desea probar de tal manera que no se mencionen clases en otro componente de la aplicación. Por "componente" se entiende una clase o clases que se desarrollan y se actualizan juntas. Las variables y los parámetros que se deben declarar con interfaces e instancias de otros componentes deben pasarse en o crearse mediante un generador.

© JMA 2020. All rights reserved

Shim

- Las correcciones de compatibilidad desvían las llamadas a métodos específicos hacia el código que se escribe como parte de la prueba haciendo que puedan devolver resultados coherentes en cada llamada. Esto hace que sea más fácil escribir pruebas.
- Los shim son sustitutos de las clases reales por lo que, al contrario que los Stubs, los Shims no requieren que el código a testear sea diseñado de una manera específica. Los Shims ofrecen la posibilidad de reemplazar las dependencias interceptando las llamadas a las dependencias en tiempo de ejecución desviándolas a un código especial con los valores apropiados para la prueba.
- Al utilizar Shims en un marco de pruebas unitarias, es necesario encapsular el código de prueba en un elemento ShimsContext para controlar la duración de las intercepciones o, de lo contrario, durarían hasta el cierre de AppDomain.

```
using(ShimsContext.Create()) {  
    // ...  
}
```

© JMA 2020. All rights reserved

Cuando elegir

Objetivo Consideración	Stub	Shim
Rendimiento	Mejor	Peor
Aplicabilidad	Inyección	Siempre
Métodos abstractos y virtuales	X	
Interfaces	X	
Tipos internos	X	X
Métodos estáticos		X
Tipos sellados		X
Métodos privados		x

© JMA 2020. All rights reserved

Agregar ensamblado de Fakes

- Para utilizar los Stubs y los Shims es necesario generar un ensamblado de Fakes por cada ensamblado que contenga las clases o interfaces a suplantar o sustituir. El ensamblado de Fakes contiene la implementación base de los Stubs y los Shims.
- En el Explorador de soluciones, en las Referencias del proyecto de prueba unitaria sobre el ensamblado deseado, hacer clic con el botón secundario y seleccionar Agregar ensamblado de Fakes.
- Automáticamente se añaden los ensamblados de soporte y se crea la carpeta Fakes que contiene los ficheros .fakes que controlan la generación de los ensamblados de Fakes.
- Al compilar se generan los ensamblados de Fakes con extensión .Fakes.dll en la carpeta oculta FakesAssemblies y, dichos ensamblados, se agregan automáticamente como referencias al proyecto de pruebas (en algunos casos es necesario agregar las referencias manualmente).

© JMA 2020. All rights reserved

Configurar la generación

- Los ficheros de configuración .fakes, en formato XML, disponen de las secciones <StubGeneration> y <ShimGeneration> que controlan para que tipos se generaran los Stubs y los Shims.
- Por defecto se generan para todos los tipos del ensamblado.
- Con <Clear /> se eliminan todos los tipos.
- Con <Add /> se añaden tipos concretos, espacios de nombres completos, ...
<Add FullName="System.IO.File" />
- Con <Remove /> se eliminan tipos concretos, espacios de nombres completos, ...
<Remove FullName="System.Security.Cryptography"/>
- Con Diagnostic="true" se habilita la depuración al generar el ensamblado de Fakes al compilar.

© JMA 2020. All rights reserved

Convenciones de nomenclatura

- Espacios de nombres
 - El sufijo .fakes se agrega al espacio de nombres (Por ejemplo, el espacio de nombres System.Fakes contiene los tipos de correcciones de compatibilidad del espacio de nombres System).
 - Global.Fakes contiene el tipo de correcciones de compatibilidad del espacio de nombres vacío.
- Nombres de tipo
 - Se agrega el prefijo Shim al nombre del tipo para generar el nombre del tipo de correcciones de compatibilidad (Por ejemplo, ShimExample es el tipo de correcciones de compatibilidad del tipo Example).
 - Se agrega el prefijo Stub al nombre del tipo para generar el nombre del tipo de stub (Por ejemplo, StubExample es el tipo de stub del tipo IExample).
- Argumentos de tipo y estructuras de tipo anidado
 - Se copian los argumentos de tipo genérico.
 - Se copia la estructura de tipo anidado para los tipos de correcciones de compatibilidad.

© JMA 2020. All rights reserved

Convenciones de nomenclatura

- Para métodos y propiedades se siguen una serie de convenciones: empiezan con el nombre salvo el constructor que se denomina Constructor.
- Si es una propiedad:
 - se anexa Get o Set según corresponda.
- Si el método:
 - si es genérico, se anexa Ofn, donde n es el número de argumentos de método genérico.
 - si tiene parámetros, se anexan los nombres de sus tipos en el orden de la firma.
 - Si los parámetros son por referencia o de salida, se anexa a su tipo Out o Ref.
 - Si los parámetros son arrays o genéricos, se anexa a su tipo Array o *Oftipo*, donde tipo es el tipo del genérico.

© JMA 2020. All rights reserved

Interfaces originales

```
namespace Demos {  
    public interface ILoadTextFile {  
        bool IsLoad { get; set; }  
        string[] Lines { get; set; }  
        int Size { get; }  
    }  
  
    public interface IProcessTextFile {  
        string Calculate(ILoadTextFile file);  
    }  
}
```

© JMA 2020. All rights reserved

Stubs generados

```
namespace Demos.Fakes {
    [StubClass(typeof(Demos.ILoadTextFile))]
    public class StubILoadTextFile : StubBase<Demos.ILoadTextFile>, Demos.ILoadTextFile {
        public FakesDelegates.Func<bool> IsLoadGet;
        public FakesDelegates.Action<bool> IsLoadSetBoolean;
        public FakesDelegates.Func<string[]> LinesGet;
        public FakesDelegates.Action<string[]> LinesSetStringArray;
        public FakesDelegates.Func<int> SizeGet;
        public StubILoadTextFile();
        public void AttachBackingFieldToIsLoad();
        public void AttachBackingFieldToLines();
    }

    [StubClass(typeof(Demos.IProcessTextFile))]
    public class StubIProcessTextFile : StubBase<Demos.IProcessTextFile>, Demos.IProcessTextFile {
        public FakesDelegates.Func<Demos.ILoadTextFile, string> CalculateILoadTextFile;
        public StubIProcessTextFile();
    }
}
```

© JMA 2020. All rights reserved

Clase Original

```
namespace Demos {
    public class LoadTextFile : ILoadTextFile {
        public string[] Lines { get; set; }
        public bool IsLoad { get; set; } = false;
        public int Size => Lines.Length;

        public LoadTextFile() {
            Lines = new string[] { };
        }
        public LoadTextFile(string fileName) {
            this.Lines = System.IO.File.ReadAllLines(fileName);
            IsLoad = true;
        }
    }
}
```

© JMA 2020. All rights reserved

Shim generado

```
namespace Demos.Fakes {  
    [ShimClass(typeof(Demos.LoadTextFile))]  
    public class ShimLoadTextFile : ShimBase<Demos.LoadTextFile> {  
        public static FakesDelegates.Action<Demos.LoadTextFile> Constructor { set; }  
        public static FakesDelegates.Action<Demos.LoadTextFile, string> ConstructorString { set; }  
        public FakesDelegates.Func<bool> IsLoadGet { set; }  
        public FakesDelegates.Action<bool> IsLoadSetBoolean { set; }  
        public FakesDelegates.Func<string[]> LinesGet { set; }  
        public FakesDelegates.Action<string[]> LinesSetStringArray { set; }  
        public FakesDelegates.Func<int> SizeGet { set; }  
        // ...  
    }  
}
```

© JMA 2020. All rights reserved

Método de prueba con Stubs

```
[TestMethod()]  
public void LoadTextFileStubTest() {  
    ILoadTextFile loadTextStub = new Demos.Fakes.StubILoadTextFile() {  
        SizeGet = () => 3  
    };  
    Assert.AreEqual(3, loadTextStub.Size);  
    Assert.IsNull(loadTextStub.Lines);  
  
    IProcessTextFile processTextStub = new Demos.Fakes.StubIProcessTextFile() {  
        CalculateLoadTextFile = (f) => f.Size > 0 ? "OK" : "KO"  
    };  
    Assert.AreEqual("OK", processTextStub.Calculate(loadTextStub));  
}
```

© JMA 2020. All rights reserved

Método de prueba con Shim

```
public void LoadTextFileShimTest() {  
    using (Microsoft.QualityTools.Testing.Fakes.ShimsContext.Create()) {  
        System.IO.Fakes.ShimFile.ReadAllLinesString =  
            s => new string[] { "cero", "uno", "dos" };  
        var arrage = new LoadTextFile("kk.file");  
        Assert.IsTrue(arrage.IsLoad);  
        Assert.AreEqual(3, arrage.Size);  
        Assert.AreEqual("dos", arrage.Lines[2]);  
        Assert.AreEqual("uno", arrage.Lines[1]);  
        Assert.AreEqual("cero", arrage.Lines[0]);  
    }  
}
```

© JMA 2020. All rights reserved

Espías

- En algunos casos se tiene que comprobar si se llama al componentes, cuantas veces, que parámetros se le han pasado, ... Se puede colocar una aserción en el código auxiliar o, mediante clausura, almacenar el valor y comprobarlo en el cuerpo principal de la prueba.
- Para llamar al método original mientras se está en el método de corrección de compatibilidad (shim) hay que encapsular la llamada al método original mediante ShimsContext.ExecuteWithoutShims().

```
[TestMethod()]  
public void LoadTextFileSpyTest() {  
    using (ShimsContext.Create()) {  
        int cont = 0;  
        var filename = @"D:\AreaTest.csv";  
        System.IO.Fakes.ShimFile.ReadAllLinesString = s => {  
            Assert.AreEqual(filename, s);  
            cont++;  
            string[] rslt = null;  
            ShimsContext.ExecuteWithoutShims(() => { rslt = File.ReadAllLines(s); });  
            return rslt;  
        };  
        var arrage = new LoadTextFile(filename);  
        Assert.AreEqual(1, cont);  
    }  
}
```

© JMA 2020. All rights reserved

<https://github.com/moq/moq4>

moq

© JMA 2020. All rights reserved

Introducción

- Moq (pronunciado "Mock-you" o simplemente "Mock") es la biblioteca para mocking más popular y amigable para .NET desarrollada desde cero para aprovechar al máximo los árboles de expresión .NET Linq y las expresiones lambda, lo que la convierte en una de las bibliotecas más productiva, con seguridad de tipos y fácil de refactorizar disponible.
- Admite dobles de pruebas tanto sobre interfaces como sobre clases.
- El API es extremadamente simple y directo, no requiere ningún conocimiento previo o experiencia con conceptos de dobles de pruebas.
- La instalación y descarga se realiza a través de NuGet.
 - Install-Package Moq

© JMA 2020. All rights reserved

Dobles de prueba

- Se puede usar Moq para crear dobles de pruebas sobre interfaces y clases existentes, pero hay algunos requisitos con las clases. Moq genera clases proxy para crear los dobles de pruebas de las clases (basado en el código de Castle DynamicProxy):
 - La clase no puede ser sellada.
 - Los métodos y propiedades simulados deben ser sobrescribibles (marcados con virtual).
 - No puede simular de los métodos estáticos (hay que usar el patrón adaptador para simular un método estático).
 - Si la clase no dispone de un constructor sin parámetros, al crear el mock hay que suministrar los argumentos del constructor deseado.
- Crear un mock:

```
var mock = new Mock<IFoo>();  
var mock = new Mock<MyClass>(ConstructorArgs);
```
- Para acceder al doble de prueba:

```
IFoo stub = mock.Object;
```

© JMA 2020. All rights reserved

Dobles de prueba

- Para hacer que el doble de prueba se comporte como un "simulacro verdadero", generando excepciones para cualquier cosa que no tenga la expectativa correspondiente:

```
var mock = new Mock<IFoo>(MockBehavior.Strict);
```
- El comportamiento predeterminado es simulacro "Loose", si no se suplanta un miembro no arroja excepciones y, para los interfaces, devuelve valores predeterminados (valor nulo correspondiente: null, 0, false, '0', ...) o matrices vacías, enumerables, etc.. En las clases se ejecuta el miembro real en caso de que no sea suplantado.
- Para añadir en modo estricto el resto de las propiedades no suplantadas:

```
mock.SetupAllProperties();
```
- Un doble de pruebas devolverá un nuevo doble de pruebas para cada miembro que no tenga expectativas y cuyo valor de retorno se puede suplantar.

© JMA 2020. All rights reserved

Suplantación

- El proceso de suplantación sobrescribe un método o propiedad de una clases o lo implementa en un interface con una versión que directamente establece la expectativa: un valor conocido.
- Para la suplantación de método se utilizan los métodos Setup siguiendo el patrón:
 - `mock.Setup(obj => obj.metodo(argumentos)).Returns(valor);`
- El valor devuelto puede ser constante o el resultado de ejecutar una expresión lambda.
`var count = 1;`
`mock.Setup(obj => obj.GetCount()).Returns(() => count);`
- Los argumentos pueden ser valores concretos, solo se activa la suplantación cuando se invoca con dichos valores, se pueden crear varias suplantaciones del mismo método con diferentes valor:
`mock.Setup(obj => obj.Get(1)).Returns("uno");`
`mock.Setup(obj => obj.Get(2)).Returns("dos");`

© JMA 2020. All rights reserved

Suplantación

- Para los parámetros por referencia hay que indicar la referencia de activación:
`var instance = new Bar();`
`mock.Setup(obj => obj.Submit(ref instance)).Returns(true);`
`var copy = instance; // var copy = new Bar();`
`Assert.IsTrue(mock.Object.Submit(ref copy));`
- Para parámetros de salida se debe suministrar una variable con el valor de retorno que se asignará a la referencia suministrada en la invocación:
`var outString = "ack";`
`mock.Setup(obj => obj.TryParse("ping", out outString)).Returns(true);`
`var myString = "";`
`Assert.IsTrue(mock.Object.TryParse("ping", out myString));`
`Assert.AreEqual("ack", myString);`

© JMA 2020. All rights reserved

Suplantación

- Con cualquier valor en los argumentos:
`mock.Setup(obj => obj.Get(It.IsAny<int>())).Returns(true);`
- Con cualquier valor en los argumentos que no sea nulo:
`mock.Setup(obj => obj.Do(It.IsNotNull<string>())).Returns("OK");`
- Con cualquier valor pasado por referencia en los argumentos:
`mock.Setup(obj => obj.Submit(ref It.Ref<Bar>.IsAny)).Returns(true);`
- Con valores en los argumentos dentro de un rango (incluyendo o excluyendo los extremos):
`mock.Setup(obj => obj.Add(It.IsInRange<int>(0, 10, Range.Inclusive))).Returns(true);`
- Con valores en los argumentos incluidos o excluidos en un conjunto de valores:
`mock.Setup(obj => obj.Get(It.IsIn<int>(4,5,6))).Returns("set");`
`mock.Setup(obj => obj.Get(It.IsNotIn<int>(4,5,6))).Returns("unset");`

© JMA 2020. All rights reserved

Suplantación

- Con valores en los argumentos que cumpla una expresión regular:
`mock.Setup(obj => obj.Do(It.IsRegex("[a-d]+"))).Returns("obj");`
- Con valores en los argumentos que cumpla condición expresada como una expresión lambda :
`mock.Setup(obj => obj.Add(It.Is<int>(i => i % 2 == 0))).Returns(true);`
- Para utilizar los valores de los argumentos en los valores devueltos:
`mock.Setup(x => x.Do(It.IsAny<string>())).Returns((string arg) => arg.ToLower());`
- Para devolver una excepción:
`mock.Setup(obj => obj.Do("reset")).Throws<InvalidOperationException>();`
`mock.Setup(obj => obj.Do("")).Throws(new ArgumentException("empty"));`
`Assert.ThrowsException<InvalidOperationException>(() =>`
`mock.Object.Do("reset"));`

© JMA 2020. All rights reserved

Suplantación

- Para devolver una secuencia de valores:

```
var mock = new Mock<IFoo>();
mock.SetupSequence(f => f.GetCount())
    .Returns(3) // will be returned on 1st invocation
    .Returns(2) // will be returned on 2nd invocation
    .Returns(1) // will be returned on 3rd invocation
    .Throws(new InvalidOperationException()); // will be thrown on 4th invocation
Assert.AreEqual(3, mock.Object.GetCount());
Assert.AreEqual(2, mock.Object.GetCount());
Assert.AreEqual(1, mock.Object.GetCount());
Assert.ThrowsException<InvalidOperationException>(() => mock.Object.GetCount());
```

© JMA 2020. All rights reserved

Verificación

- Moq suministra el método Verify para validar la interacción con los métodos de suplantación o espiar métodos no suplantados. Se comporta como una aserción: si falla la verificación, falla la prueba.
- Verificar que se ha invocado el método:
mock.Verify(foo => foo.Do("reset"), "This will print on failure");
- Verificar que se ha invocado el método un determinado número de veces:
mock.Verify(foo => foo.Do("reset"), Times.Exactly(3));
- Verificar que se ha invocado el método un rango de veces:
mock.Verify(foo => foo.Do("ping"), Times.Between(2, 5, Range.Inclusive));
- Verificar que se ha invocado el método al menos o como mucho un número de veces:
mock.Verify(foo => foo.Do("ping"), Times.AtLeastOnce());
mock.Verify(foo => foo.Do("ping"), Times.AtMost(5));
- Verificar que se no ha invocado el método:
mock.Verify(foo => foo.DoSomething("ping"), Times.Never());

© JMA 2020. All rights reserved

Verificación

- Al igual que en la suplantación, en la verificación se puede utilizar la clase `It` que permite la especificación de una condición coincidente para un argumento en una invocación de método, en lugar de un valor de argumento específico:
 - `Is<TValue>`: Coincide con cualquier valor que satisfaga el predicado dado.
 - `IsAny<TValue>`: Coincide con cualquier valor del tipo `TValue` dado.
 - `IsIn<TValue>`: Coincide con cualquier valor que esté presente en la secuencia especificada.
 - `IsInRange<TValue>`: Coincide con cualquier valor que esté en el rango especificado.
 - `IsNotIn<TValue>`: Coincide con cualquier valor que no se encuentre en la secuencia especificada.
 - `IsNotNull<TValue>`: Coincide con cualquier valor del tipo `TValue` dado, excepto nulo.
 - `IsRegex`: Coincide con un argumento de cadena si coincide con el patrón de expresión regular dado.

```
mock.Verify(obj => obj.Do(It.Is<string>(s => s.Length == 4))
```

© JMA 2020. All rights reserved

Verificación

- Para verificar que se han utilizado todas las suplantaciones:
`mock.VerifyAll();`
- Para marcar como verificable una suplantación y realizar una verificación conjunta:

```
var mock = new Mock<IFoo>();  
mock.Setup(obj => obj.Do("ping")).Returns("OK").Verifiable();  
mock.Setup(obj => obj.Do("pong")).Returns("KO");  
Assert.AreEqual("OK", mock.Object.Do("ping"));  
mock.Verify();
```

© JMA 2020. All rights reserved

Propiedades

- Para la suplantación de la propiedades se utiliza también los métodos Setup:
`mock.Setup(obj => obj.Name).Returns("value");`
- Para que la suplantación tenga "comportamiento de propiedad", guarde y recupere su valor:
`mock.SetupProperty(f => f.Name, "value");`
- Para fijar la expectativa de la asignación de una propiedad:
`mock.SetupSet(obj => obj.Name = "value");`
- Para verificar que se ha accedido a una propiedad:
`mock.VerifyGet(obj => obj.Name);`
- Para verificar que se ha asignado una propiedad:
`mock.VerifySet(obj => obj.Name);`
- Para verificar la asignación a la propiedad de un determinado valor o un rango de valores:
`mock.VerifySet(obj => obj.Name = "value");`
`mock.VerifySet(obj => obj.Value = It.IsInRange(1, 5, Range.Inclusive));`

© JMA 2020. All rights reserved

Eventos

- Para lanzar el evento desde el doble de prueba (+= null se ignora pero evita el error sintáctico):
`mock.Object.MyEvent += (object sender, EventArgs e) => str = "OK";`
`mock.Raise(m => m.MyEvent += null, new EventArgs());`
`Assert.AreEqual("OK", str);`
- Raise acepta mas argumentos si no se sigue el patrón EventHandler.
- Para activar la supervisión de la asignación y des asignación de controladores de eventos:
`mock.SetupAdd(m => m.MyEvent += It.IsAny<EventHandler>())`
`mock.SetupRemove(m => m.MyEvent -= It.IsAny<EventHandler>())`
- Para verificar que se ha asignado un controlador de eventos:
`mock.VerifyAdd(obj => obj.MyEvent += It.IsAny<EventHandler>());`
- Para verificar que se he quitado un controlador de eventos:
`mock.VerifyRemove(obj => obj.MyEvent -= It.IsAny<EventHandler>());`

© JMA 2020. All rights reserved

Callbacks

- Moq permite, mediante el método `Callbacks`, hacer un seguimiento de las llamadas antes y después de que se produzcan:

```
var mock = new Mock<IFoo>();
var calls = 0;
var callArgs = new List<string>();

mock.Setup(foo => foo.Do("ping"))
    .Callback(() => calls++)
    .Returns("OK")
    .Callback((string s) => callArgs.Add(s));
Assert.AreEqual("OK", mock.Object.Do("ping"));
Assert.AreEqual(1, calls);
Assert.AreEqual(1, callArgs.Count);
Assert.AreEqual("ping", callArgs[0]);
```

© JMA 2020. All rights reserved

LINQ to Mocks

- Moq es el único marco de simulación que permite especificar el comportamiento simulado a través de consultas de especificación declarativas.

```
var services = Mock.Of<IServiceProvider>{sp =>
    sp.GetService(typeof(IRepository)) == Mock.Of<IRepository>(r => r.IsAuthenticated == true) &&
    sp.GetService(typeof(IAuthentication)) == Mock.Of<IAuthentication>(a => a.AuthenticationType == "OAuth")};
```

- Múltiples suplantaciones en un solo doble:
`ControllerContext context = Mock.Of<ControllerContext>(ctx =>
 ctx.HttpContext.Request.IsAuthenticated == true &&
 ctx.HttpContext.Request.Url == new Uri("http://moqthis.com") &&
 ctx.HttpContext.Response.ContentType == "application/xml");`

- Múltiples suplantaciones y dobles:
`var context = Mock.Of<ControllerContext>(ctx =>
 ctx.HttpContext.Request.Url == new Uri("http://moqthis.me") &&
 ctx.HttpContext.Response.ContentType == "application/xml" &&
 // Especificación encadenada
 ctx.HttpContext.GetSection("server") == Mock.Of<ServerSection>(config =>
 config.Server.ServerUrl == new Uri("http://moqthis.com/api")));`

© JMA 2020. All rights reserved

MÉTRICAS PARA EL PROCESO DE PRUEBAS DE SOFTWARE

© JMA 2020. All rights reserved

Cobertura de código

- Para determinar qué proporción de código del proyecto se está probando realmente mediante pruebas codificadas como pruebas unitarias, se puede utilizar la característica de cobertura de código de Visual Studio. Para restringir con eficacia los errores, las pruebas deberían ensayar o “cubrir” una proporción considerable del código.
 - El análisis de cobertura de código puede aplicarse al código administrado (CLI) y no administrado (nativo).
 - La cobertura de código es una opción al ejecutar métodos de prueba mediante el Explorador de pruebas. La tabla de salida muestra el porcentaje de código que se ejecuta en cada ensamblado, clase y método. Además, el editor de código fuente muestra qué código se ha probado.
 - La característica de cobertura de código solo está disponible en la edición Visual Studio Enterprise.
-


© JMA 2020. All rights reserved

Cobertura de código

- La cobertura de código se cuenta en bloques.
- Un bloque es un fragmento de código con un punto de entrada y de salida exactamente.
- Si el flujo de control del programa pasa a través de un bloque durante una serie de pruebas, ese bloque se cuenta como cubierto.
- El número de veces que se utiliza el bloque no tiene ningún efecto en el resultado.
- También se pueden mostrar los resultados en líneas eligiendo Agregar o quitar columnas en el encabezado de tabla.
- Algunos usuarios prefieren un recuento de líneas porque los porcentajes corresponden más al tamaño de los fragmentos que aparece en el código fuente.
- Un bloque grande de cálculo contaría como un único bloque aunque ocupe muchas líneas.

© JMA 2020. All rights reserved

Cobertura de código

- La ventana de resultados de cobertura de código normalmente muestra el resultado de la ejecución más reciente. Los resultados variarán si se cambian los datos de prueba, o si se ejecutan solo algunas pruebas cada vez.
- La ventana de cobertura de código también se puede utilizar para ver los resultados anteriores o los resultados obtenidos en otros equipos.
- Después de que se hayan ejecutado las pruebas, para ver qué líneas se han ejecutado en el editor de código fuente, se marca el icono  “Mostrar colores en cobertura de código” en la ventana “Resultados de cobertura de código”. De forma predeterminada, el código que se incluye en las pruebas se resalta en color azul claro.

© JMA 2020. All rights reserved

Cobertura de código

- Puede que se desee excluir elementos concretos del código de las puntuaciones de cobertura, por ejemplo si el código se genera a partir de una plantilla de texto.
- Para excluir de la cobertura se agrega el atributo `[ExcludeFromCodeCoverage]` a cualquiera de los elementos de código: clase, struct, método, propiedad, establecedor o captador de propiedad, evento.
- Se puede tener más control sobre qué ensamblados y elementos están seleccionados para el análisis de cobertura de código escribiendo un archivo `.runsettings`.

© JMA 2020. All rights reserved

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 Visual Studio para generar datos de métricas de código que miden la complejidad y el mantenimiento del código administrado. Los datos de métricas de código pueden generarse para una solución completa o un proyecto único (opción “Calcular métricas de código”).

© JMA 2020. All rights reserved

Métricas de código

- Índice de mantenimiento

- Calcula un valor de índice entre 0 y 100 que representa su relativa facilidad de mantenimiento del código. Un valor alto significa mayor facilidad de mantenimiento. Las clasificaciones de colores, pueden utilizarse para identificar rápidamente los puntos conflictivos en el código. Una clasificación en verde entre 20 y 100 e indica que el código tiene buen mantenimiento. Una calificación amarilla es entre 10 y 19 e indica que el código es moderadamente fácil de mantener. Una clasificación de color rojo es una clasificación entre 0 y 9 e indica un mantenimiento baja.

- Complejidad ciclomática:

- Mide la complejidad del código estructural. Se crea, calculando el número de rutas de acceso de código diferente en el flujo del programa. Un programa que tiene un flujo de control complejo requiere más pruebas para lograr una buena cobertura de código y es mas difícil de mantener.

© JMA 2020. All rights reserved

Métricas de código

- Profundidad de herencia:

- Indica el número de clases diferentes que heredan de otra, hasta la clase base. La profundidad de la herencia es similar a la Unión de clases en que un cambio en una clase base puede afectar a cualquiera de sus clases heredadas. Cuanto mayor sea este número, más profunda será la herencia y mayor será la posibilidad de que las modificaciones de la clase base produzcan cambios importantes. En cuanto a la profundidad de la herencia, un valor bajo es bueno y un valor alto es mas arriesgado.

- Acoplamiento de clases:

- Mide el acoplamiento a las clases únicas a través de parámetros, variables locales, tipos de valor devuelto, llamadas a métodos, las creaciones de instancias genérica o de plantilla, clases base, implementaciones de interfaz, los campos definidos en los tipos externos y decoración de atributo. Un buen diseño de software dicta que se debe tener una alta cohesión y un bajo acoplamiento en los tipos y métodos. Un significativo acoplamiento indica un diseño que es difícil reutilizar y mantener debido a sus interdependencias en otros tipos.

© JMA 2020. All rights reserved

Métricas de código

- **Líneas de código fuente:**
 - Indica el número exacto de líneas de código fuente que se encuentran en el archivo de código fuente, incluidas las líneas en blanco. Esta métrica está disponible a partir de Visual Studio 2019, versión 16.4, y Microsoft.CodeAnalysis.Metrics (2.9.5).
- **Líneas de código ejecutable:**
 - Indica el número aproximado de operaciones o líneas de código ejecutable. Se trata de un recuento del número de operaciones en el código ejecutable. Esta métrica está disponible a partir de Visual Studio 2019, versión 16.4, y Microsoft.CodeAnalysis.Metrics (2.9.5). Normalmente, el valor es una coincidencia aproximada con la métrica anterior, líneas de código fuente, que es la métrica basada en instrucciones de MSIL utilizada en el modo heredado.

© JMA 2020. All rights reserved

Métricas de código

- **Métodos anónimos**
 - Un método anónimo es simplemente un método que no tiene nombre. Los métodos anónimos se utilizan con más frecuencia para pasar un bloque de código como parámetro delegado. Los resultados de las métricas de código para un método anónimo que se declara en un miembro, como un método o un descriptor de acceso, están asociados al miembro que declara el método. No se asocian con el miembro que llama al método.
- **Código generado**
 - Algunas herramientas de software y los compiladores generan código que se agrega a un proyecto y que el programador del proyecto no ve o no debe cambiar. Principalmente, las métricas del código omite el código generado cuando calcula los valores de métricas. Esto permite que los valores de las métricas reflejar lo que el desarrollador puede ver y cambiar. No se omite el código generado para Windows Forms, porque es código que el desarrollador puede ver y cambiar.

© JMA 2020. All rights reserved

Calidad de las pruebas

- Se insiste mucho en que la cobertura de test unitarios de los proyectos sea lo más alta posible, pero es evidente que cantidad (de test, en este caso) no siempre implica calidad, la calidad no se puede medir "al peso", y es la calidad lo que realmente importa.
- La cobertura de prueba tradicional (líneas, instrucciones, rama, etc.) mide solo qué código ejecuta las pruebas. No comprueba que las pruebas son realmente capaces de detectar fallos en el código ejecutado, solo pueden identificar el código que no se ha probado.
- Los ejemplos más extremos del problema son las pruebas sin afirmaciones (poco comunes en la mayoría de los casos). Mucho más común es el código que solo se prueba parcialmente, cubrir todo los caminos no implica ejercitar todos 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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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:
 - $\text{Mutación Puntuación} = 100 * D / (N - E)$
donde D es el número de mutantes muertos, N es el número de mutantes y E es el número de mutantes equivalentes

© JMA 2020. All rights reserved

Stryker.NET

- Stryker.NET permite realizar pruebas de mutación para proyectos .NET Core y .NET Framework. Stryker.NET es una herramienta de consola que se instala con NuGet y, para su ejecución, requiere el .net 8 RunTime, aunque los proyectos pueden estar en otras versiones.
- Para instalarlo globalmente:
 - `dotnet tool install -g dotnet-stryker`
- Para instalarlo localmente, en la carpeta del proyecto de pruebas:
 - `dotnet new tool-manifest`
 - `dotnet tool install dotnet-stryker`
- Matemos a algunos mutantes (en la carpeta del proyecto de pruebas):
 - `dotnet stryker`
 - `dotnet stryker -s "../solution.sln"` ← *si esta instalado globalmente*
- Una vez ejecutado, se emitirá un informe html que mostrará visualmente el proyecto y todas las mutaciones. Con la opción `--reporter` se emitirá en: "cleartext", "cleartexttree", "json", "markdown", ...

© JMA 2020. All rights reserved

Análisis estático Web

- HTML
 - <https://validator.w3.org/>
- CSS
 - <http://jigsaw.w3.org/css-validator/>
- WAI
 - <https://www.w3.org/WAI/ER/tools/>
- JavaScript
 - <http://jshint.com/>
 - <http://www.jshint.com/>
- TypeScript
 - <https://palantir.github.io/tslint/>
- Lighthouse en Chrome DevTools (panel "Audits")
 - <https://github.com/GoogleChrome/lighthouse>

© JMA 2020. All rights reserved

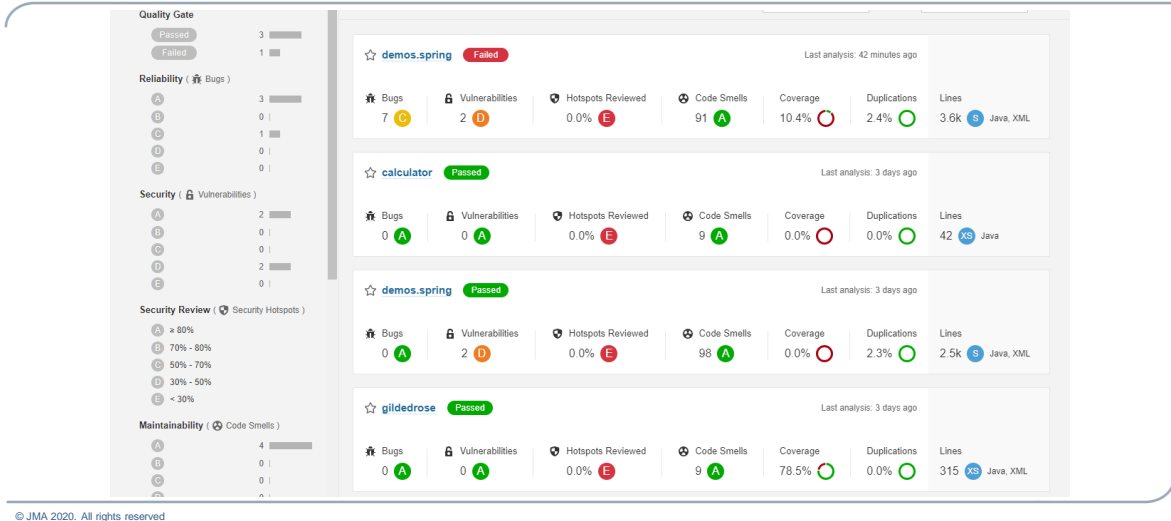
SonarQube

<http://www.sonarqube.org/>

- SonarQube (conocido anteriormente como Sonar) es una plataforma para la revisión y evaluación del código fuente.
- Es open source y realiza el análisis estático de código fuente integrando las mejores herramientas de medición de la calidad de código como Checkstyle, PMD o FindBugs, para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.
- Informa sobre código duplicado, estándares de codificación, pruebas unitarias, cobertura de código, complejidad ciclomática, posible errores, comentarios y diseño del software.
- Aunque pensado para Java, acepta mas de 20 lenguajes mediante extensiones.
- Se integra con Maven, Ant y herramientas de integración continua como Atlassian Bamboo, Jenkins y Hudson).

© JMA 2020. All rights reserved

Sonar



© JMA 2020. All rights reserved

BUENAS PRACTICAS

© JMA 2020. All rights reserved

Características de una buena prueba unitaria

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

© JMA 2020. All rights reserved

Asignar nombre a las pruebas

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

© JMA 2020. All rights reserved

Organizar el código de la prueba

- Prepara, actuar, afirmar es un patrón común al realizar pruebas unitarias. Como el propio nombre implica, consta de tres acciones principales:
 - Prepara los objetos, crearlos y configurarlos según sea necesario.
 - Actuar en un objeto.
 - Afirmar que algo es como se espera.
- Separa claramente en secciones lo que se está probando de los pasos preparación y verificación. Esta separación permite identificar donde está el problema en caso de producirse una excepción no controlada en el código de la prueba.
- Las secciones solo deben una vez como máximo y en el orden establecido.
- Minimiza la posibilidad de mezclar aserciones con el código para "actuar", solo aceptable en pruebas de flujos.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Sustituir literales por constantes

- Los valores literales pueden provocar confusión al lector de las pruebas. Si una cadena tiene un aspecto fuera de lo normal, puede preguntarse por qué se ha elegido un determinado valor para un parámetro o valor devuelto. Esto obliga a un vistazo más detallado a los detalles de implementación, en lugar de centrarse en la prueba. La asignación de literales a constantes permite dar nombre a los valores, aportando semántica.
- Evita la necesidad de que el lector de la prueba inspeccione el código de producción con el fin de averiguar lo que significa un valor, que hace el valor sea especial.
`Assert.IsTrue(rsIt.Length <= 10)`
- Muestra explícitamente lo que se intenta probar, en lugar de lo que se intenta lograr.
`const string VARCHAR_LEN = 10;`
`Assert.IsTrue(rsIt.Length <= VARCHAR_LEN)`

© JMA 2020. All rights reserved

Evitar varias aserciones

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

© JMA 2020. All rights reserved

Refactorizar código

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

© JMA 2020. All rights reserved

No validar métodos privados

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

© JMA 2020. All rights reserved

Aislar las pruebas

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

© JMA 2020. All rights reserved

Cubrir aspectos no evidentes

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

© JMA 2020. All rights reserved

Respetar los limites de las pruebas

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

© JMA 2020. All rights reserved

Visual Studio Enterprise

INTELLITEST

© JMA 2020. All rights reserved

PEX

- Pex (Program Exploration) es una herramienta desarrollada por Microsoft Research para automática y sistemáticamente producir el conjunto mínimo de entradas de prueba necesarios para ejecutar un número finito de rutas de acceso.
- Pex produce de manera automática un conjunto pequeño de casos de prueba con una amplia cobertura de código y aserción.
- Pex busca valores de entrada salida interesantes de los métodos, que pueden guardarse como un conjunto pequeño de pruebas con la máxima cobertura de código posible.
- Pex realiza un análisis sistemático, buscando para condiciones límite, excepciones y errores de aserción que puede depurar inmediatamente.
- Pex también permite pruebas parametrizada (PUT), una extensión de las pruebas unitarias que reduce los costos de mantenimiento de prueba y utiliza la ejecución dinámica simbólica para sondear a través del código de prueba para crear un conjunto de pruebas que tratan la mayoría de las ramas de ejecución.
- IntelliTest es la evolución de Pex, a partir de Visual Studio 2015.

© JMA 2020. All rights reserved

IntelliTest

- Las pruebas unitarias inteligentes, una característica de Visual Studio Enterprise, son un ayudante inteligente para el desarrollo de software, que ayuda a que los equipos de desarrollo encuentren errores pronto y reduzcan el coste de mantenimiento de las pruebas.
- Su motor usa análisis de código de caja blanca y resolución de restricciones para sintetizar los valores de entrada de prueba precisos y cubrir todas las rutas de código en el código que se somete a prueba, los conserva como un conjunto compacto de pruebas unitarias tradicionales con alta cobertura y evolucionar automáticamente el conjunto de pruebas a medida que evoluciona el código.
- IntelliTest explora el código .NET para generar datos de prueba y un conjunto de pruebas unitarias. Para cada instrucción en el código, se genera una entrada de prueba que ejecutará esa instrucción. Se lleva a cabo un análisis de caso para cada bifurcación condicional en el código. Con este análisis puede generar los datos de pruebas que deben usarse en una prueba unitaria parametrizada para cada método.
- Cuando se ejecuta Intelltest, se puede ver fácilmente qué pruebas son las que fallan y agregar cualquier código para corregirlas, seleccionar las pruebas generadas que quiere guardar en un proyecto de prueba para proporcionar un conjunto de regresión. Cuando cambie el código, se vuelve a ejecutar IntelliTest para mantener sincronizadas las pruebas generadas con los cambios de código.

© JMA 2020. All rights reserved

Características

- Genera automáticamente test
- Intenta conseguir la mayor cobertura de código con la menor cantidad de test.
- No cubre todos los caminos lógicos.
- Detecta valores limites aunque no todos
- Es necesario volver a ejecutar manualmente IntelliTest con cada cambio en nuestro código.
- Permite la personalización mediante partial class, factorías, forzar casos validos, ...
- Crea los casos de prueba, la verificación de las reglas semánticas hay que implementarlas manualmente con la personalización.
- Soporta MSTest y, mediante adaptadores, xUnit o NUnit.

© JMA 2020. All rights reserved

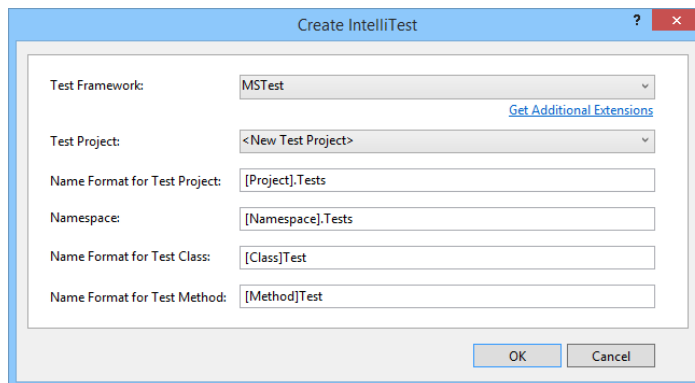
Limitaciones

- Presupone que el programa analizado es determinista. Si no lo es, IntelliTest lo recorrerá hasta que alcance un límite de exploración.
- No controla los programas multiproceso.
- No entiende el código nativo.
- Solo se admite en .NETframework de 32 bits basado en X86.
- Puede analizar los programas escritos en cualquier lenguaje de .NET pero Visual Studio solo admite C#.
- Usa un solucionador de restricciones automático para determinar los valores que son relevantes para la prueba y el programa sometido a prueba, no valida semántica.
- Solo disponible en Visual Studio Enterprise.

© JMA 2020. All rights reserved

Explorar el código

- En la clase o método deseado hacer clic con el botón secundario → IntelliTest → Crear IntelliTest.



© JMA 2020. All rights reserved

PexClass

- Genera una clase [PexClass] que contiene las exploraciones. La clase debe ser publica, no abstracta y tener el constructor sin parámetros.
- Las exploraciones son métodos de prueba unitaria parametrizada marcados con [PexMethod]:
 - debe ser un método de instancia
 - debe ser visible para la clase de prueba en la que se colocan las pruebas generadas según la cascada de configuración
 - puede tomar cualquier número de parámetros
 - puede ser genérica
 - puede contener pre condiciones y post condiciones

© JMA 2020. All rights reserved

Patrón AAAA

- Los métodos de prueba unitaria parametrizada siguen una variación del patrón AAA, el patrón Assume, Arrange, Act, Assert.
- Añade la fase previa Assume, donde se restringe las posibles entradas de prueba que genera y suministra IntelliTest, actuando como un filtro.
- Se suministra clases asistentes y atributos para facilitar la implementación de las pre condiciones (Assume) y las post condiciones (Assert).

© JMA 2020. All rights reserved

Clases del asistente

- IntelliTest proporciona un conjunto de clases del asistente estáticas que pueden usarse al crear pruebas unitarias parametrizadas:
 - PexAssume: se usa para definir hipótesis en las entradas, y es útil para filtrar las entradas no deseadas.
 - PexAssert: es una clase de aserción sencilla que se usa si su marco de pruebas no proporciona una.
 - PexChoose: un flujo de entradas de prueba adicionales a los parámetros que administra IntelliTest.
 - PexObserve: registra valores de salida concretos y, opcionalmente, los valida en el código generado.
 - PexSymbolicValue: utilidades para inspeccionar o modificar restricciones simbólicas en variables.

© JMA 2020. All rights reserved

PexAssumeNotNull

- Este atributo declara que el valor controlado no puede ser null. Puede adjuntarse a:
 - un parámetro de un método de prueba parametrizado

```
[PexMethod]
public void SomeTest([PexAssumeNotNull]IFoo foo, ...) {}
```
 - un campo

```
public class Foo {
    [PexAssumeNotNull]
    public object Bar;
}
```
 - un tipo

```
[PexAssumeNotNull]
public class Foo {}
```
- El atributo [PexAssumeUnderTest] indica Pex que sólo debe pasar los valores no nulos del tipo exacto especificado.

© JMA 2020. All rights reserved

PexUseType y PexAllowedException

- **PexUseType:** Este atributo le indica a IntelliTest que puede usar un tipo particular para crear instancias de interfaces o tipos base (abstractos).

```
[PexMethod]
[PexUseType(typeof(A))]
[PexUseType(typeof(B))]
public void MyTest(object testParameter) {
    ... // testParameter is A || testParameter is B
}
```
- **PexAllowedException:** Si este atributo está adjunto a PexMethod (o a PexClass, cambia la lógica de IntelliTest predeterminada que indica cuando se produce un error en las pruebas. La prueba no se considerará incorrecta, aunque genere la excepción especificada. Crea un caso de prueba específico buscando la excepción: [ExpectedException].

© JMA 2020. All rights reserved

PexMethod

```
[PexMethod]
void PUT([PexAssumeUnderTest] ArrayList target, [PexAssumeNotNull] object item, int p) {
    // assume
    PexAssume.IsTrue(p > 0);
    int i = PexChoose.ValueFrom<int>("i", new int[] { 1, 5, 10 });
    // arrange
    var count = target.Count;
    // act
    target.Add(item);
    // assert
    PexAssert.IsTrue(target.Count == count + 1);
    PexObserve.ValueForViewing<string>("count", target.Count.ToString());
    PexObserve.ValueForViewing<string>("result", $"Valor: {i * p}");
}
```

© JMA 2020. All rights reserved

Generar pruebas unitarias

- Una creado o revisados los `PexMethod` hay que generar las método de prueba [`TestMethod`]. La generación se realiza al 'Ejecutar IntelliTest', donde cada `PexMethod` se convierte en tantos `TestMethod` como juegos de argumentos determine IntelliTest que son necesarios.
- En la clase o método deseado hacer clic derecho con el botón secundario → IntelliTest → Ejecutar IntelliTest.

	lengths	result	Summary/Exception	Error Message
1	null		NullPointerException	Object refer...
2	{}		IndexOutOfRangeException	Index was out...
3	{0}		IndexOutOfRangeException	Index was out...
4	{0, 0}		IndexOutOfRangeException	Index was out...
5	{0, 0, 0}	Invalid		
6	{5, 538, 0}	Invalid		
7	{67, 0, 0}	Invalid		
8	{422, 536, 6...}	Scalene		
9	{528, 413, 5...}	Isosceles		
10	{2, 2, 3}	Isosceles		
11	{1, 512, 512}	Isosceles		
12	{512, 512, 5...}	Equilateral		

Details:
Stack trace:
System.NullReferenceException...
at Triangle.ClassifyBySideLengt...
at TriangleTest.ClassifyBySideL...

© JMA 2020. All rights reserved

Factorías

- Aunque IntelliTest genera instancias de entradas de prueba, algunas veces necesita ayuda para construir el objeto o se quiere personalizar las mismas.

```
public static partial class MiTipoFactory {  
    [PexFactoryMethod(typeof(MiTipo))]  
    public static MiTipo Create(int value_i, bool value_b) {  
        MiTipo miTipo = new MiTipo();  
        miTipo.prop1 = value_i;  
        miTipo.prop2 = value_b;  
        return miTipo;  
    }  
}
```

© JMA 2020. All rights reserved

Configuración en cascada

- El concepto de configuración en cascada significa que el usuario puede especificar opciones en el nivel Ensamblado (PexAssemblySettings), Corrección (PexClass) y Exploración (PexExplorationAttributeBase):
- Las opciones especificadas en el nivel Ensamblado (en el archivo PexAssemblyInfo.cs) afectan a todas las correcciones y a la exploración en ese ensamblado.
- Las opciones especificadas en el nivel Corrección afectan a todas las exploraciones de esa corrección.
- Las opciones secundarias tienen preferencia— si una opción se define en los niveles Ensamblado y Corrección, se usan las opciones de Corrección.

© JMA 2020. All rights reserved

Límites de exploración

- Límites de solución de restricciones
 - MaxConstraintSolverTime: el número de segundos que el solucionador de restricciones tiene para detectar entradas que provocarán una nueva y diferente ruta de ejecución que se va a seguir.
 - MaxConstraintSolverMemory: el tamaño en megabytes que puede usar el solucionador de restricciones para detectar entradas.
- Límites de la ruta de exploración
 - MaxBranches: el número máximo de ramas que se pueden tomar a lo largo de una sola ruta de ejecución.
 - MaxCalls: el número máximo de llamadas que pueden realizarse durante una sola ruta de ejecución.
 - MaxStack: el tamaño máximo de la pila en cualquier momento durante una sola ruta de ejecución, medido como el número de marcos de llamada activos.
 - MaxConditions: el número máximo de condiciones en las entradas que se pueden comprobar durante una sola ruta de acceso de ejecución.

© JMA 2020. All rights reserved

Límites de exploración

- Límites de exploración
 - MaxRuns: el número máximo de ejecuciones que se intentarán durante una exploración.
 - MaxRunsWithoutNewTests: el número máximo de ejecuciones consecutivas sin que se emita una nueva prueba.
 - MaxRunsWithUniquePaths: el número máximo de ejecuciones con rutas de ejecución únicas que se intentarán durante una exploración.
 - MaxExceptions: el número máximo de excepciones que pueden detectarse para una combinación de todas las rutas de ejecución detectadas.
- Configuración de la generación de código para el conjunto de pruebas
 - TestExcludePathBoundsExceeded: cuando se establezca en True, las rutas de ejecución que exceden cualquiera de los límites de ruta (MaxCalls, MaxBranches, MaxStack, MaxConditions) se ignoran.
 - TestEmissionFilter: indica en qué circunstancias IntelliTest debe emitir pruebas.
 - TestEmissionBranchHits: controla cuántas pruebas emite IntelliTest.

© JMA 2020. All rights reserved

PRUEBAS DE ACCESO A DATOS

© JMA 2020. All rights reserved

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 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.

© JMA 2020. All rights reserved

Prácticas recomendadas

- 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 ponga en un estado conocido antes de cada test.
- Para las pruebas de integración, usar múltiples conjuntos de datos pequeños en lugar de uno grande. Cada prueba necesita un conjunto de tablas y de registros, no necesariamente toda la base de datos. En cambio, las pruebas de sistema requieren entornos realistas.
- Inicializar los datos comunes sólo una vez para todos los tests. Si hay datos de sólo lectura, no tenemos por qué restaurarlos si nos aseguramos, con restricciones de permisos, que las pruebas no los modificaran.
- Descartar las transacciones una vez realizadas las verificaciones anula los cambios salvo excepciones como las identidades o las secuencias.
- La restauración de copias de seguridad es un mecanismo rápido y eficaz para llevar a la base de datos a un estado inicial pero requiere permisos administrativos.

© JMA 2020. All rights reserved

Pruebas de código que usa EF Core

- Para probar el código que accede a una base de datos, es necesario:
 - Usar dobles de prueba (moq) o algún otro mecanismo para evitar por completo el uso de una base de datos.
 - Ejecutar consultas y actualizaciones en el mismo sistema de base de datos que se usa en producción.
 - Ejecutar consultas y actualizaciones en algún otro sistema de base de datos más fácil de administrar.
- No todos los proveedores de bases de datos son iguales. Esto significa que, al cambiar de proveedor de base de datos, cambia el comportamiento de EF Core y puede que la aplicación no funcione correctamente, hay que tener en cuenta de forma explícita estas diferencias, aunque en muchos casos esto funciona, ya que hay un alto grado de homogeneidad entre bases de datos relacionales.
- La única manera de asegurarse de que se está probando lo que se ejecuta en producción es usar el mismo sistema de base de datos pero es lento y costoso.

© JMA 2020. All rights reserved

Ciclo de vida

- El ciclo de vida de las pruebas es el siguiente:
 - Eliminar el estado previo de la BD resultante de pruebas anteriores (en lugar de restaurarla tras cada test).
 - Cargar los datos necesarios para las pruebas de la BD (sólo los necesarios para cada test).
- El marco de pruebas creará una nueva instancia de clase de prueba para cada serie de pruebas. Esto significa que se puede instalar y configurar la base de datos en el constructor de prueba y que estará en un estado conocido para cada prueba. Esto funciona bien para las pruebas de base de datos en memoria de SQLite y EF, pero puede suponer una sobrecarga significativa con otros sistemas de base de datos, como SQL Server.
- Cuando se ejecuta cada prueba (Code First):
 - DbContextOptions se configuran para el proveedor en uso y se pasan al constructor de clase base
 - Estas opciones se almacenan en una propiedad y se usan en las pruebas para la creación de instancias de DbContext.
 - Se llama a un método de inicialización para crear e inicializar la base de datos
 - El método de inicialización garantiza que la base de datos está limpia al eliminarla y volver a crearla.
 - Algunas entidades de prueba conocidas se crean y se guardan en la base de datos.

© JMA 2020. All rights reserved

LocalDB

- Todos los principales sistemas de base de datos tienen alguna forma de "edición para desarrolladores" para las pruebas locales. SQL Server también tiene una característica denominada LocalDB. La principal ventaja de LocalDB es que inicia la instancia de base de datos a petición. Esto evita que haya un servicio de base de datos ejecutándose en el equipo aunque no se estén ejecutando pruebas.
- Pero LocalDB también plantea problemas:
 - No admite todo lo que SQL Server Developer Edition.
 - No está disponible en Linux.
 - Puede producir un retraso en la primera serie de pruebas cuando se inicia el servicio.

© JMA 2020. All rights reserved

SQLite

- La siguiente mejor opción es usar algo con funcionalidad similar. Esto suele significar otra base de datos relacional, para lo que SQLite es la opción obvia.
- SQLite es una buena opción porque:
 - Se ejecuta en proceso con la aplicación y, por tanto, tiene poca sobrecarga.
 - Usa archivos simples creados automáticamente para bases de datos, por lo que no requiere administración de bases de datos.
 - Tiene un modo en memoria que evita incluso la creación de archivos.
- Pero hay que tener en cuenta:
 - SQLite inevitablemente no admite todo lo que el sistema de base de datos de producción.
 - SQLite se comporta de forma diferente al sistema de base de datos de producción para algunas consultas.
- Por lo tanto, si se usa SQLite para algunas pruebas, hay que asegurarse de probar también en el sistema de base de datos real.

© JMA 2020. All rights reserved

Base de datos en memoria de EF Core

- EF Core incluye una base de datos en memoria que se usa para las pruebas internas del propio EF Core. Esta base de datos en general no es adecuada como sustituto para probar las aplicaciones que usan EF Core. De manera específica:
 - No es una base de datos relacional
 - No admite transacciones
 - No está optimizada para el rendimiento
- Nada de esto es muy importante a la hora de probar elementos internos de EF Core, ya que se usa específicamente donde la base de datos es irrelevante para la prueba. Por otro lado, estos aspectos tienden a ser muy importantes al probar una aplicación que usa EF Core.
- Los dobles de prueba se usan para las pruebas internas de EF Core. Pero nunca se intentan simular DbContext o IQueryable. Hacerlo es difícil, engorroso y delicado. No se debe hacer.
- En su lugar, se usa la base de datos en memoria de EF siempre que se realizan pruebas unitarias de algo que use DbContext. En este caso, el uso de la base de datos en memoria de EF es adecuado porque la prueba no depende del comportamiento de la base de datos. Pero no se debe hacer para probar consultas o actualizaciones reales de la base de datos.

© JMA 2020. All rights reserved

Base de datos en memoria de EF Core

```
public static DbContextOptions<AppDbContext> TestDbContextOptions() {  
    var serviceProvider = new ServiceCollection()  
        .AddEntityFrameworkInMemoryDatabase()  
        .BuildServiceProvider();  
  
    // Create a new options instance using an in-memory database and  
    // IServiceProvider that the context should resolve all of its  
    // services from.  
    var builder = new DbContextOptionsBuilder<AppDbContext>()  
        .UseInMemoryDatabase("InMemoryDb")  
        .UseInternalServiceProvider(serviceProvider);  
  
    return builder.Options;  
}  
  
using (var db = new AppDbContext(Utilities.TestDbContextOptions())) {
```

© JMA 2020. All rights reserved

Contenedores

- Hay disponibles imágenes de Docker listas para usar para la mayoría de bases de datos principales y pueden facilitar la instalación tanto en entornos de CI como en máquinas para desarrolladores.
- Se pueden configurar una o varias imágenes Docker con bases de datos configuradas para los diferentes escenarios de pruebas. Una vez descargada la imagen (push), la creación del contenedor es prácticamente inmediata, por lo que puede crearse y destruirse con cada ciclo de pruebas. Aunque esto requiere una inversión inicial en la configuración, una vez completada, se tiene un entorno de pruebas operativo y puede centrarse en cosas más importantes.
- Cuando se usa una base de datos en la nube, es aconsejable realizar las pruebas en una versión local de la base de datos, tanto para mejorar la velocidad como para reducir los costos. Por ejemplo, al usar SQL Azure en producción, puede realizar las pruebas en un servidor de SQL Server instalado localmente; los dos son muy similares (aunque sigue siendo aconsejable ejecutar las pruebas también en SQL Azure antes de pasar a producción). Si se usa Azure Cosmos DB, el emulador de Azure Cosmos DB es una herramienta útil tanto para desarrollar de forma local como para ejecutar pruebas.

© JMA 2020. All rights reserved

Patrones DDD

- Hay disponibles un amplio conjunto de patrones de arquitectónicos que aíslan la capa de acceso a datos y facilitan su sustitución por dobles de pruebas:
 - **Repositorio:** "media entre el dominio y las capas de asignación de datos mediante una interfaz similar a una colección para acceder a objetos de dominio". El objetivo del patrón de repositorio es aislar el código de las minucias del acceso a datos que es un rasgo necesario para la capacidad de prueba.
 - **Unidad de trabajo:** "mantendrá una lista de objetos afectados por una transacción comercial y coordina la escritura de los cambios y la resolución de problemas de simultaneidad". Es responsabilidad de la unidad de trabajo realizar un seguimiento de los cambios en los objetos que damos vida desde un repositorio y conservar los cambios que hayamos realizado en los objetos cuando le decimos a la unidad de trabajo que confirme los cambios. También es responsabilidad de la unidad de trabajo tomar los nuevos objetos que hemos agregado a todos los repositorios e insertar los objetos en una base de datos, y también la eliminación de la administración.
 - **Carga diferida:** lazy load describe "un objeto que no contiene todos los datos que necesita pero sabe cómo obtenerlo". La carga diferida transparente es una característica importante que debe tener al escribir código de negocio comprobable y trabajar con una base de datos relacional.

© JMA 2020. All rights reserved

PRUEBAS WEB

© JMA 2020. All rights reserved

Pruebas Unitarias Aisladas

- Las Pruebas Unitarias Aisladas examinan una instancia de una clase por sí misma aislada con respecto a su infraestructura, sin ninguna dependencia o de los valores inyectados.
- Un Controller, ApiController o middleware están respaldados por clases. Se crea una instancia de prueba de la clase con `new`, se le suministran los parámetros al constructor, sustituyendo las dependencias por dobles de pruebas, y se prueba la superficie de la instancia.
`var controller = new HomeController(mockRepo.Object);`
- Solo se comprueba el contenido de una única acción, no el comportamiento de sus dependencias o del marco en sí.
- Las pruebas unitarias son pequeñas, rápidas, consumen pocos recursos y concluyentes, verifican un comportamiento que ya no requiere se vuelva a comprobar. Una vez cubierto el máximo de casos posibles se pasa a las pruebas de integración y E2E.

© JMA 2020. All rights reserved

Probando los modelos

- El patrón MVC propugna una separación clara de los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones, esto permite la prueba aislada del código independientemente de la presentación.
- Los modelos, con la lógica de negocio y la capa de acceso a datos, contarán con una exhaustiva batería de pruebas unitarias y de integración.
- En las pruebas unitarias, todas las dependencias se sustituirán por dobles de prueba, ASP.NET facilita esta sustitución a través de la inyección de dependencias en los constructores.

© JMA 2020. All rights reserved

Probando los modelos

```
[TestClass()]
public class NIFAttributeTests {
    NifMock attr = new NifMock();

    [TestMethod()]
    [DataRow("12345678z")][DataRow("12345678Z")][DataRow("1234S")][DataRow(null)]
    public void NIFValidTest(string nif) {
        Assert.AreEqual(ValidationResult.Success, attr.IsValid(nif));
    }

    [TestMethod()]
    [DataRow("12345678")][DataRow("Z")][DataRow("1234J")][DataRow("")]
    public void NIFInvalidTest(string nif) {
        var rslt = attr.IsValid(nif);
        Assert.AreNotEqual(ValidationResult.Success, rslt);
        Assert.AreEqual("No es un NIF válido.", rslt.ErrorMessage);
    }
}
```

© JMA 2020. All rights reserved

Probando los controladores

- Los controladores, por definición, deberían contar con una lógica mínima: delegar en el modelo y generar el ActionResult a devolver por el método de acción (las vistas no llegan a representarse).
- En las pruebas unitarias, todas las dependencias de modelos y servicios se sustituirán por dobles de prueba, ASP.NET facilita esta sustitución a través de la inyección de dependencias en los constructores. La prueba unitaria del controlador evita escenarios como filtros, enrutamiento y enlace de modelos.
- Los ActionResult son objetos que permiten aserciones que verifiquen que los resultados son los esperados a través de las propiedades ViewData, Model, ModelState, ViewName, StatusCode, ...
- Los métodos de un ApiController devuelven los datos a serializar, que se pueden verificar mediante aserciones, o persisten datos, que se pueden verificar mediante dobles de pruebas.

© JMA 2020. All rights reserved

Probando los controladores

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessionsTest() {
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync()).ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

© JMA 2020. All rights reserved

JavaScript Test

- Extensiones:
 - Chutzpah Test Adapter for the Test Explorer
 - Chutzpah Test Runner Context Menu Extension
- Crear carpetas \scripts\app y \test
- Crear fichero \test\chutzpah.json

```
{
  "Framework": "jasmine",
  "References": [ {
    "Path": "../Scripts/app",
    "Include": "*.js",
    "Excludes": [ "*.spec.js" ]
  } ],
  "Tests": [ { "Path": "../test", "Includes": [ "*.spec.js" ] } ]
}
```
- Si fallan las dependencias, en Administrar paquetes NuGet: Añadir chutzpah y jasmine-js

© JMA 2020. All rights reserved

Probando las vistas

- Aseguremos que todo lo que llega a la vista esta testeado.
- Probamos la lógica de navegación.
- Probamos las validaciones.
- La vista se puede probar navegando el sitio manualmente o con alguna herramienta automatizada.
- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión.
- Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias.
- Las pruebas de integración y extremo a extremo (E2E: end to end) permiten encontrar estos problemas.

© JMA 2020. All rights reserved

Pruebas de integración

- Las pruebas de integración garantizan que los componentes de una aplicación funcionan correctamente en un nivel que incluye la infraestructura auxiliar de la aplicación. ASP.NET Core admite las pruebas de integración mediante un marco de pruebas unitarias con un host web de prueba y un servidor de pruebas en memoria. Debería separarse las pruebas unitarias de las pruebas de integración en proyectos diferentes.
- A diferencia de las pruebas unitarias, las pruebas de integración:
 - Usan los componentes reales que emplea la aplicación en producción.
 - Necesitan más código y procesamiento de datos.
 - Tardan más en ejecutarse.
- Para las pruebas de integración en ASP.NET Core se necesita lo siguiente:
 - Un proyecto de prueba, que se usa para contener y ejecutar las pruebas. El proyecto de prueba tiene una referencia al SUT (sistema bajo prueba).
 - El proyecto de prueba crea un host web de prueba para el SUT y usa un cliente de servidor de pruebas para controlar las solicitudes y las respuestas con el SUT.
 - Se usa un ejecutor de pruebas para ejecutar las pruebas y notificar los resultados de estas.

© JMA 2020. All rights reserved

Pruebas de integración

- Las pruebas de integración siguen una secuencia de eventos que incluye los pasos de prueba normales Arrange, Act y Assert:
 1. Se configura el host web del SUT.
 2. Se crea un cliente de servidor de pruebas para enviar solicitudes a la aplicación.
 3. Se ejecuta el paso de prueba Arrange: la aplicación de prueba prepara una solicitud.
 4. Se ejecuta el paso de prueba Act: el cliente envía la solicitud y recibe la respuesta.
 5. Se ejecuta el paso de prueba Assert: la respuesta real se valida como correcta o errónea en función de una respuesta esperada.
 6. El proceso continúa hasta que se ejecutan todas las pruebas.
 7. Se notifican los resultados de la prueba.
- Normalmente, el host web de prueba se configura de manera diferente al host web normal de la aplicación para las series de pruebas. Por ejemplo, puede usarse una base de datos diferente u otra configuración de aplicación para las pruebas.

© JMA 2020. All rights reserved

Pruebas de integración

- Los componentes de infraestructura, como el host web de prueba y el servidor de pruebas en memoria (TestServer), se proporcionan o se administran mediante el paquete Microsoft.AspNetCore.Mvc.Testing. El uso de este paquete simplifica la creación y ejecución de pruebas.
- El paquete Microsoft.AspNetCore.Mvc.Testing controla las siguientes tareas:
 - Copia el archivo de dependencias (.deps) del SUT en el directorio bin del proyecto de prueba.
 - Establece la raíz de contenido en la raíz de proyecto del SUT de modo que se puedan encontrar archivos estáticos y páginas o vistas cuando se ejecuten las pruebas.
 - Proporciona la clase WebApplicationFactory para simplificar el arranque del SUT con TestServer.

© JMA 2020. All rights reserved

Preparación

- El proyecto de prueba debe:
 - Hacer referencia al paquete Microsoft.AspNetCore.Mvc.Testing.
 - Especificar el SDK web en el archivo de proyecto (<Project Sdk="Microsoft.NET.Sdk.Web">).
- El proyecto sujeto a prueba debe:
 - Tener la clase Program pública mediante una declaración de clase parcial
public **partial** class Program {
 - O, de forma mas segura, exponer los tipos internos de la aplicación web solo al proyecto de prueba (.csproj):
<ItemGroup>
 <InternalsVisibleTo Include="MyTestProject" />
</ItemGroup>
- Las clases de prueba implementan la interfaz de accesorio de clase (IClassFixture) para indicar que la clase contiene pruebas y proporcionan instancias de objeto compartidas en las pruebas de la clase. WebApplicationFactory<TEntropyPoint> se usa para crear un elemento TestServer para pruebas de integración, donde TEntryPoint es la clase punto de entrada del SUT, normalmente Program.

© JMA 2020. All rights reserved

Clase de prueba

```
public class HomeControllerTests : IClassFixture<WebApplicationFactory<Program>> {
    private readonly WebApplicationFactory<Program> _factory;

    public HomeControllerTests(WebApplicationFactory<Program> factory) {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Home/Privacy")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url) {
        // Arrange
        var client = _factory.CreateClient();
        // Act
        var response = await client.GetAsync(url);
        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299 Assert.Equal(HttpStatusCode.OK, response.StatusCode);
        Assert.Equal("text/html; charset=utf-8", response.Content.Headers.ContentType.ToString());
    }
}
```

© JMA 2020. All rights reserved

Configuraciones

- La configuración de host web se puede crear independientemente de las clases de prueba al heredar de `WebApplicationFactory<TEntryPoint>` para crear una o más fábricas personalizadas con configuraciones de servicios, cambiando el `dbContext`, realizando una autenticación ficticia, ...
- Cuando se requiere configuración adicional del cliente en un método de prueba, `WithWebHostBuilder` crea una nueva instancia de `WebApplicationFactory` con un elemento `IWebHostBuilder` que se personaliza aún más mediante la configuración.
- Los servicios se pueden invalidar en una prueba con una llamada a `ConfigureTestServices` en el generador de hosts.

© JMA 2020. All rights reserved

PRUEBAS E2E

© JMA 2020. All rights reserved

Introducción

- A medida que las aplicaciones crecen en tamaño y complejidad, se vuelve imposible depender de pruebas manuales para verificar la corrección de las nuevas características, errores de captura y avisos de regresión. Las pruebas unitarias son la primera línea de defensa para la captura de errores, pero a veces las circunstancias requieran la integración entre componentes que no se pueden capturar en una prueba unitarias.
- 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 cliente, solo importa el interfaz (UI o API) y no como esté la aplicación 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 cliente) 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.

© JMA 2020. All rights reserved

Ventajas e inconvenientes

- Las pruebas de extremo a extremo han sido ampliamente adoptadas porque:
 - Ayuda a los equipos a expandir su cobertura de pruebas agregando casos de pruebas más detallados que otros métodos de prueba, como pruebas unitarias y funcionales.
 - Garantiza que la aplicación funcione correctamente ejecutando los casos de prueba en función del comportamiento del usuario final.
 - Ayuda a los equipos de lanzamiento a reducir el tiempo de paso a producción al permitirles automatizar las rutas críticas de los usuarios.
 - Reduce el costo general de creación y mantenimiento del software al disminuir el tiempo que lleva probar el software.
 - Ayuda a detectar errores de manera predecible y confiable, aunque no de forma concluyente.
- Los inconvenientes se presentan porque las pruebas de extremo a extremo:
 - Toman mucho tiempo, son frágiles y no son concluyentes (detectan el fallo pero no donde se produce).
 - Deben estar diseñadas para reproducir escenarios del mundo real.
 - Requiere una buena comprensión de los objetivos del usuario: Los usuarios no buscan características, buscan resolver sus problemas específicos.
- Las pruebas de extremo a extremo requieren un grupo multidisciplinario que incluye desarrolladores, probadores, administradores y usuarios.

© JMA 2020. All rights reserved

Estrategias de automatización

- **Record-and-replay (R&R):** Se orientan a la grabación de pruebas exploratorias sobre el GUI que luego se pueden reproducir automáticamente. La fase de grabación (recording) incluye aserciones de verificación y genera un script con las instrucciones a reproducir. El script puede estar en un lenguaje propio de la herramienta usada o puede tener instrucciones de un API para automatización. Tienen el beneficio que son fáciles de aprender, grabar y reproducir, sin embargo, pero son difíciles de mantener o personalizar al estar limitadas por la herramienta utilizada.
- **APIs de automatización:** Existen librerías y frameworks que permiten la automatización de pruebas de GUI mediante código, a través de APIs que permiten controlar externamente el GUI. Si bien se benefician de un mayor control y flexibilidad, tienen una curva de aprendizaje mucho mayor, deben ocuparse de los detalles, como los tiempos de espera, y dependen de un código que hay que escribir y mantener, mucho mas costoso que una grabación.

© JMA 2020. All rights reserved

Conceptos a tener en cuenta

- **Flujo funcional:**
 - Proceso en el que se define un comportamiento a partir de unas entradas, y un tratamiento de esas entradas por parte de todos los sistemas o componentes, hasta llegar a un resultado final.
- **Camino crítico:**
 - En un flujo funcional participan normalmente muchos sistemas o componentes. Sería un error pretender hacer la validación del todo, porque por una parte normalmente no hay tiempo material para hacerlo, y por otra, muchos de los sistemas no tienen relevancia. El camino crítico lo componen sólo los sistemas a validar en la prueba E2E.
- **Pruebas horizontales o verticales:**
 - Las horizontales cubren aspectos de toda la aplicación mientras que las pruebas verticales cubren un único aspecto.

© JMA 2020. All rights reserved

Proceso de Prueba

- Identifica con qué navegadores probar
- Elige el mejor lenguaje/entorno para ti y tu equipo.
- Configura entorno para que funcione con cada navegador que te interese.
- Descompón la aplicación web existente para identificar qué probar
- Escribe pruebas mantenibles y reutilizables que sean compatibles y ejecutables en todos los navegadores.
- Crea un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo de las herramientas de pruebas.

© JMA 2020. All rights reserved

Coded UI

- Las pruebas automatizadas de IU (CUIT) controlan la aplicación a través de su interfaz de usuario. Estas pruebas incluyen unas pruebas funcionales de los controles de la interfaz de usuario. Permiten comprobar si toda la aplicación, incluida la interfaz de usuario, funciona correctamente. Las pruebas automatizadas de IU son útiles donde haya validación u otra lógica en la interfaz de usuario, por ejemplo, en una página web. También se suelen usar para automatizar una prueba manual existente.
- Coded UI está en desuso, Visual Studio 2019 es la última versión en la que estará completamente disponible y en Visual Studio 2022 solo se pueden ejecutar las pruebas ya existentes.
- Como alternativas el equipos de Visual Studio:
 - Se recomienda usar Playwright para probar aplicaciones web
 - Appium con WinAppDriver para probar aplicaciones de escritorio y para UWP.
 - Xamarin.UITest para probar aplicaciones de iOS y Android con NUnit.

© JMA 2020. All rights reserved

Selenium

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web. Como principales herramientas Selenium cuenta con:
 - Selenium IDE: una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
 - Selenium Core: API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes como C#, Java, JavaScript, Ruby, Groovy, Perl, Php y Python.
 - WebDriver: interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
 - Selenium Grid: Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos, un conjunto de pruebas muy grande puede dividirse en varias maquinas remotas para una ejecución más rápida o si se necesitan repetir en múltiples entornos.

© JMA 2020. All rights reserved

Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas.
- Solo está disponible como una extensión de Firefox y Chrome.
- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese (JSON).
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
 - **Acciones:** son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
 - **Descriptores de acceso:** examinan el estado de la página y almacenan los resultados en variables.
 - **Aserciones:** son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

© JMA 2020. All rights reserved

Selenium IDE

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- Permite exportar los casos de prueba a Java, JavaScript, C#, Python y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

© JMA 2020. All rights reserved

Localizadores

- Localizar por Id:
 - id=loginForm
- Localizar por Name
 - name=username
- Localizar por el texto en los hipervínculos
 - link=Continue
- Localizar por CSS
 - css=input[name="username"]
- Localizar por XPath
 - xpath=//form[@id='loginForm']

© JMA 2020. All rights reserved

Variables

- Se puede usar variable en Selenium para almacenar constantes al principio de un script. Además, cuando se combina con un diseño de prueba controlado por datos, las variables de Selenium se pueden usar para almacenar valores pasados a la prueba desde la línea de comandos, desde otro programa o desde un archivo.
 - store target:valor value:varName
- Para acceder al valor de una variable:
 - \${userName}
- Hay métodos disponibles para recuperar información de la página y almacenarla en variables:
 - storeAttribute, storeText, storeValue, storeTitle, storeXPathCount

© JMA 2020. All rights reserved

Afirmar y Verificar

- Una "afirmación" hará fallar la prueba y abortará el caso de prueba actual, mientras que una "verificación" hará fallar la prueba pero continuará ejecutando el caso de prueba.
 - Tiene muy poco sentido para comprobar que el primer párrafo de la página sea el correcto si la prueba ya falló al comprobar que el navegador muestra la página esperada. Por otro lado, es posible que desee comprobar muchos atributos de una página sin abortar el caso de prueba al primer fallo, ya que esto permitirá revisar todos los fallos en la página y tomar la acción apropiada.
- Selenese permite múltiples formas de comprobar los elementos de la interfaz de usuario pero hay que decidir el métodos mas apropiado:
 - ¿Un elemento está presente en algún lugar de la página?
 - ¿El texto especificado está en algún lugar de la página?
 - ¿El texto especificado está en una ubicación específica en la página?
- Métodos:
 - `assert`, `assertAlert`, `assertChecked`, `assertNotChecked`, `assertConfirmation`, `assertEditable`, `assertNotEditable`, `assertElementPresent`, `assertElementNotPresent`, `assertPrompt`, `assertSelectedValue`, `assertNotSelectedValue`, `assertSelectedLabel`, `assertText`, `assertNotText`, `assertTitle`, `assertValue`
 - `verify`, `verifyChecked`, `verifyNotChecked`, `verifyEditable`, `verifyNotEditable`, `verifyElementPresent`, `verifyElementNotPresent`, `verifySelectedValue`, `verifyNotSelectedValue`, `verifyText`, `verifyNotText`, `verifyTitle`, `verifyValue`, `verifySelectedLabel`

© JMA 2020. All rights reserved

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
    System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}

@Before
public void setUp() throws Exception {
    driver = new ChromeDriver();
    baseUrl = "http://localhost/";
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@Test
public void testLoginOK() throws Exception {
    driver.get(baseUrl + "/login.php");
    driver.findElement(By.id("login")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("admin");
    driver.findElement(By.cssSelector("input[type='submit']")).click();
    try {
        assertEquals("", driver.findElement(By.cssSelector("img[title='Main Menu']")).getText());
    } catch (Error e) {
        verificationErrors.append(e.toString());
    }
}
```

Maven

```
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>3.13.0</version>
</dependency>
```

© JMA 2020. All rights reserved

Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (<https://nodejs.org>)
- Instalar CLI
 - npm install -g selenium-side-runner
 - npm install -g chromedriver edgedriver geckodriver
- Instalar los WebDriver:
 - Crear una carpeta y referenciarla en el PATH del sistema.
 - Descargar los drivers (<https://www.seleniumhq.org/download/>)
 - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
 - selenium-side-runner project.side project2.side *.side
- Para ejecutar en diferentes navegadores:
 - selenium-side-runner *.side -c "browserName=Chrome"
 - selenium-side-runner *.side -c "browserName=firefox"

© JMA 2020. All rights reserved

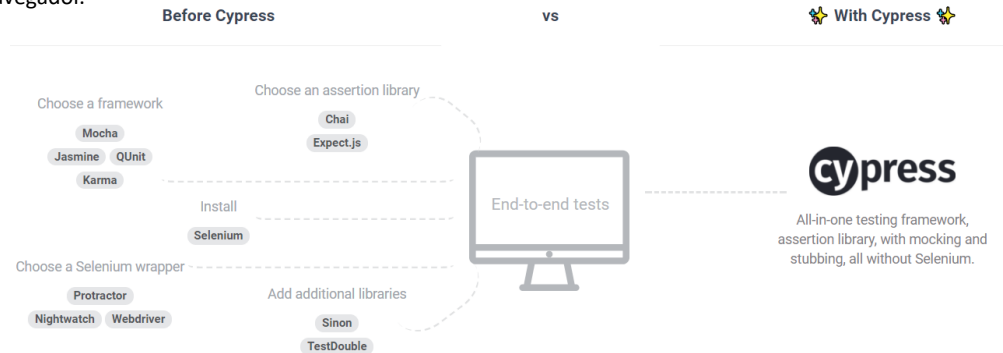
Cypress

- Cypress es una herramienta de prueba de front-end de próxima generación creada para la web moderna, que permite configurar, escribir, ejecutar y depurar pruebas.
- Aunque Cypress se compara con Selenium, es fundamental y arquitectónicamente diferente, controla los navegadores a través de JavaScript y las DevTools, por lo que no está limitado por las mismas restricciones que el Selenium. Esto permite escribir pruebas más rápidas, fáciles y confiables. Esta destinado a desarrolladores o ingenieros de control de calidad que crean aplicaciones web utilizando marcos de JavaScript modernos.
- Aunque Cypress puede probar cualquier cosa que se ejecute en un navegador, esta destinado a:
 - Pruebas de un extremo a extremo
 - Pruebas de integración
 - Pruebas unitarias (componentes)
- Cypress se ha construido para que las pruebas y el desarrollo se puedan realizar simultáneamente. Es suficientemente rápido para habilitar metodologías TDD, BDD o ATDD, donde los cambios en el código o en las pruebas se reflejan en tiempo real.

© JMA 2020. All rights reserved

Cypress

- Escribir pruebas e2e requiere muchas herramientas diferentes trabajando juntas. Cypress es un todo en uno, no es necesario instalar 10 herramientas y bibliotecas independientes para configurar el entorno de pruebas. Han tomado algunas de las mejores herramientas de su clase y las han hecho funcionar juntas sin problemas.
- Las pruebas de Cypress solo están escritas en JavaScript, el código de prueba se ejecuta dentro del propio navegador.



© JMA 2020. All rights reserved

Playwright

- [Playwright](#) es una biblioteca Node.js respaldada por Microsoft para automatizar navegadores con una sola API. Playwright está diseñado para habilitar la automatización web entre exploradores que es perenne, capaz, confiable y rápida.
- Cualquier navegador & Cualquier plataforma & One API
 - Cross-browser: Playwright es compatible con todos los motores de renderizado modernos, incluidos Chromium, WebKit y Firefox.
 - Cross-platform: Pruebas en Windows, Linux y macOS, localmente o en CI, sin cabeza o con cabeza.
 - Cross-language: Usa la API de Playwright en TypeScript, JavaScript, Python, .NET (MSTest o NUnit) o Java.
 - Prueba Web Móvil. Emulación móvil nativa de Google Chrome para Android y Mobile Safari. El mismo motor de renderizado funciona en el escritorio y en la nube.
- Potentes Herramientas
 - Codegen: Genera pruebas registrando sus acciones. Guárdelos en cualquier idioma.
 - Playwright inspector: Inspeccione la página, genere selectores, pase por la ejecución de la prueba, vea los puntos de clic, explore los registros de ejecución.
 - Trace Viewer: Capture toda la información para investigar el fallo de la prueba. La traza de Playwright contiene screencast de ejecución de prueba, instantáneas DOM en vivo, explorador de acción, fuente de prueba y muchos más.

© JMA 2020. All rights reserved

Appium

- [Appium](#) es una herramienta de automatización de código abierto para ejecutar scripts y probar aplicaciones nativas, aplicaciones web y aplicaciones híbridas sobre Android o iOS utilizando un webdriver basado en la [W3C WebDriver Specification](#) al estilo Selenium. Appium, originalmente en C#, se porto a Node.js en 2013, siendo actualmente su plataforma de ejecución.
- Appium es un proyecto de código abierto y un ecosistema relacionado software, diseñado para facilitar la automatización de UI de muchas plataformas de aplicaciones siempre que se disponga del correspondiente driver:
 - Móvil: iOS, Android, Tizen, ...
 - Navegador: Chrome, Firefox, Safari, ...
 - Escritorio: macOS, Windows, ...
 - TV: Roku, tvOS, Android TV, Samsung, LG WebOS, ...
- Appium Client admite que el código de automatización este escrito en diferentes lenguajes: C#, JavaScript, Java, Python, Ruby, ...

© JMA 2020. All rights reserved

Postman

<https://www.getpostman.com/>

- Postman surgió originariamente como una extensión para el navegador Google Chrome que permitía realizar peticiones API REST con métodos diferentes al GET. A día de hoy dispone de aplicaciones nativas para MAC, Windows y algunas producciones Linux.
- Está compuesto por diferentes herramientas y utilidades gratuitas (en la versión free) que permiten realizar tareas diferentes dentro del mundo API REST: creación de peticiones a APIs internas o de terceros, elaboración de tests para validar el comportamiento de APIs, posibilidad de crear entornos de trabajo diferentes (con variables globales y locales), y todo ello con la posibilidad de ser compartido con otros compañeros del equipo de manera gratuita (exportación de toda esta información mediante URL en formato JSON).
- Además, dispone de un modo cloud colaborativo (de pago) para que equipos de trabajo puedan desarrollar entre todos colecciones para APIs sincronizadas en la nube para una integración más inmediata y sincronizada.
- Quizás sea una de las herramientas más utilizadas para hacer testing exploratorio de API REST.

© JMA 2020. All rights reserved

SoapUI

<https://www.soapui.org>

- SoapUI es una herramienta para probar servicios web que pueden ser servicios web SOAP, servicios web RESTful u otros servicios basados en HTTP.
- SoapUI es una herramienta de código abierto y completamente gratuita con una versión comercial, ReadyAPI, que tiene una funcionalidad adicional para empresas con servicios web de misión crítica.
- SoapUI se considera el estándar de facto para las Pruebas de servicio API. Esto significa que hay mucho conocimiento en la red sobre la herramienta y blogs para obtener más información sobre el uso de SoapUI en la vida real.
- SoapUI permite realizar pruebas funcionales, pruebas de rendimiento, pruebas de interoperabilidad, pruebas de regresión y mucho más. Su objetivo es que la prueba sea bastante fácil de comenzar, por ejemplo, para crear una Prueba de carga, simplemente hay que hacer clic derecho en una prueba funcional y ejecutarla como una prueba de carga.
- SoapUI puede simular servicios web (mocking). Se puede grabar pruebas y usarlas más tarde.

© JMA 2020. All rights reserved

PRUEBAS DE RENDIMIENTO

© JMA 2020. All rights reserved

Pruebas de rendimiento



© JMA 2020. All rights reserved

Pruebas de carga, estrés y picos

- Pruebas de carga (load test): pruebas para determinar y validar la respuesta de la aplicación cuando es sometida a una carga de usuarios y/o transacciones que se espera en el ambiente de producción. Ejemplo: verificar la correcta respuesta de la aplicación ante el alta de 100 usuarios en forma simultánea. Se compara con el volumen esperado.
- Pruebas de estrés (stress test): pruebas para encontrar el volumen de datos o de tiempo en que la aplicación comienza a fallar o es incapaz de responder a las peticiones. Son pruebas de carga o rendimiento, pero superando los límites esperados en el ambiente de producción y/o determinados en las pruebas. Ejemplo: encontrar la cantidad de usuarios simultáneos, en que la aplicación deja de responder (cuelgue o time out) en forma correcta a todas las peticiones.
- Pruebas de picos (spike testing): es un sub tipo de prueba de estrés que mide el rendimiento del software bajo un «pico» significativo y repentino o una carga de trabajo creciente como la de los usuarios simulados. Indica si el software puede manejar ese aumento abrupto de la carga de trabajo de forma repetida y rápida.

© JMA 2020. All rights reserved

Pruebas de resistencia, volumen y escalabilidad

- Pruebas de resistencia (soak testing, endurance testing): evalúa el rendimiento del software durante un periodo prolongado bajo una carga de trabajo regular y fija, determina cuánto tiempo puede soportar el software una carga de trabajo constante para proporcionar sostenibilidad a largo plazo. Durante estas pruebas, los equipos de pruebas supervisan los KPI como las fugas de memoria, de proceso, etc. Las pruebas de resistencia también analizan los tiempos de respuesta y el rendimiento tras un uso prolongado para mostrar si estas métricas son consistentes.
- Pruebas de volumen (volume testing): comprueban la eficacia del software cuando se somete a grandes volúmenes de datos. Comprueba la pérdida de datos, el tiempo de respuesta del sistema, la fiabilidad del almacenamiento de datos, etc.
- Pruebas de escalabilidad (scalability testing): miden la eficacia del software a la hora de manejar una cantidad creciente de carga de trabajo, añadiendo volumen de datos o usuarios de forma gradual mientras se supervisa el rendimiento del software. La prueba informará sobre el comportamiento cuando aumenten o disminuyan los atributos de rendimiento del software.

© JMA 2020. All rights reserved

JMeter

- <http://jmeter.apache.org>
- JMeter es una herramienta de pruebas de carga para llevar acabo simulaciones sobre cualquier recurso de Software.
- Inicialmente diseñada para pruebas de estrés en aplicaciones web, hoy en día, su arquitectura ha evolucionado no sólo para llevar acabo pruebas en componentes habilitados en Internet (HTTP), sino también en Bases de Datos , programas en Perl, peticiones FTP y prácticamente cualquier otro medio.
- Además, posee la capacidad de realizar desde una solicitud sencilla hasta secuencias de peticiones que permiten diagnosticar el comportamiento de una aplicación en condiciones de producción.
- En este sentido, simula todas las funcionalidades de un navegador, o de cualquier otro cliente, siendo capaz de manipular resultados en determinada requisición y reutilizarlos para ser empleados en una nueva secuencia.

© JMA 2020. All rights reserved