



# Herramientas de Pruebas

Visual Studio

© JMA 2019. All rights reserved

---

## INTRODUCCIÓN A LAS TÉCNICAS DE PRUEBAS

---

© JMA 2019. All rights reserved

## Introducción

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

© JMA 2019. All rights reserved

## Introducción

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

© JMA 2019. All rights reserved

## Prueba de defectos

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

© JMA 2019. All rights reserved

## Prueba estadística

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

© JMA 2019. All rights reserved

## Prueba de regresión

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

© JMA 2019. All rights reserved

## Conflicto de intereses

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

© JMA 2019. All rights reserved

## Desarrollador como probador

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

© JMA 2019. All rights reserved

## Grupo independiente de prueba

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

© JMA 2019. All rights reserved

## Principios fundamentales

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

© JMA 2019. All rights reserved

## Las pruebas exhaustivas no son viables

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

© JMA 2019. All rights reserved

## Ejecución de pruebas bajo diferentes condiciones

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

© JMA 2019. All rights reserved

## El proceso no puede demostrar la ausencia de defectos

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

© JMA 2019. All rights reserved

## Las pruebas no garantizan ni mejoran la calidad del software

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

© JMA 2019. All rights reserved

## Las pruebas tienen un coste

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

© JMA 2019. All rights reserved

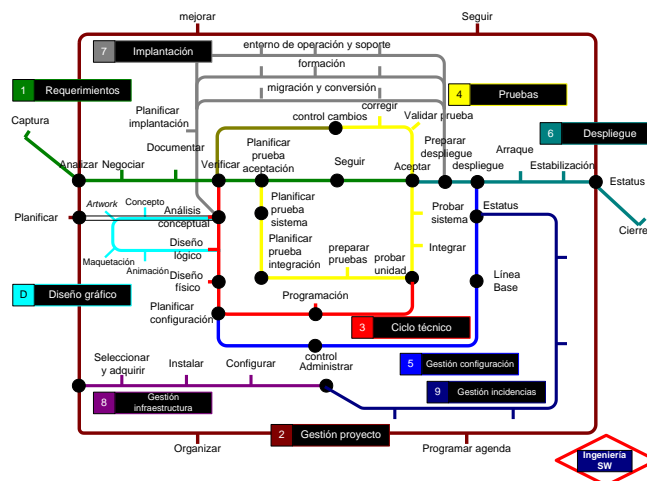


## Inicio temprano de pruebas

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

© JMA 2019. All rights reserved

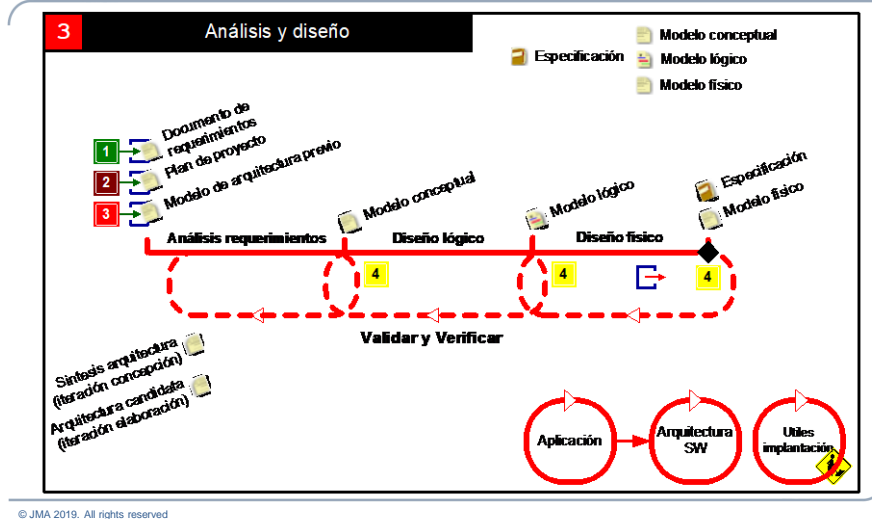
## Integración en el ciclo de vida Símil del mapa del Metro



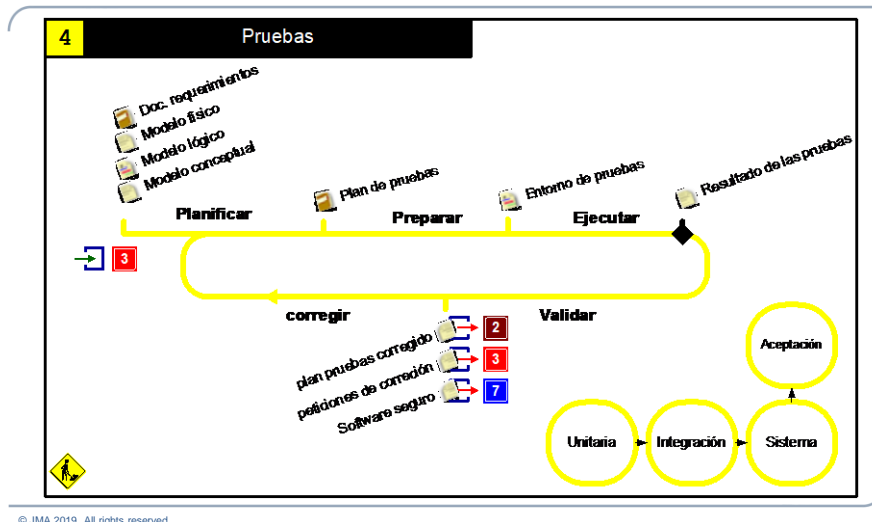
© JMA 2019. All rights reserved



## Línea 3: Análisis y diseño



## Línea 4: Pruebas



---

## EL PROCESO DE PRUEBAS

---

© JMA 2019. All rights reserved

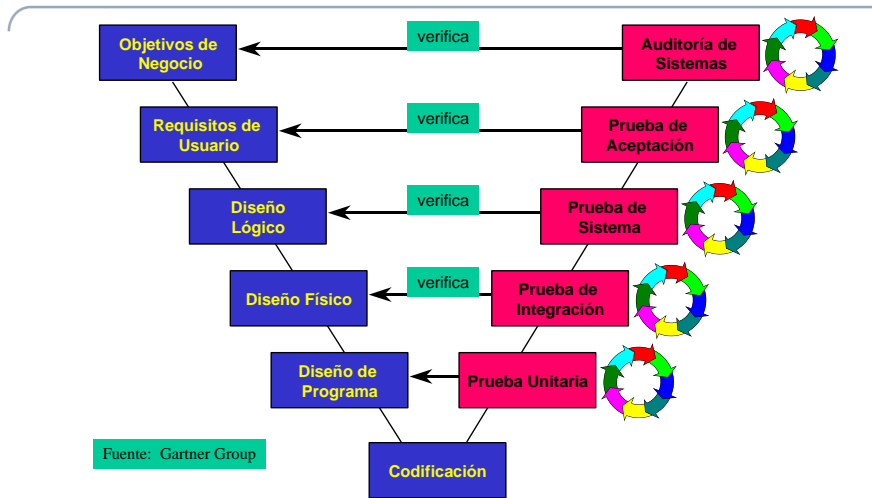
### El proceso

---

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

© JMA 2019. All rights reserved

## Proceso de pruebas: Ciclo en V



© JMA 2019. All rights reserved

## Ciclo en V

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

© JMA 2019. All rights reserved

## Niveles de pruebas

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

© JMA 2019. All rights reserved

## Tipos de Pruebas

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

© JMA 2019. All rights reserved

## Pruebas Unitarias

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

© JMA 2019. All rights reserved

## Pruebas Unitarias

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

© JMA 2019. All rights reserved

## Pruebas Unitarias

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

© JMA 2019. All rights reserved

## Pruebas Unitarias

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

© JMA 2019. All rights reserved



## Pruebas de Integración

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

© JMA 2019. All rights reserved

## Pruebas de Integración

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

© JMA 2019. All rights reserved

## Pruebas del Sistema

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

© JMA 2019. 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, sometiénolo 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 2019. 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 configuración:** consisten en comprobar todos y cada uno de los dispositivos, en sus propiedades mínimo y máximo posibles.
- **Pruebas de usabilidad:** consisten en comprobar la adaptabilidad del sistema a las necesidades de los usuarios, tanto para asegurar que se acomoda a su modo habitual de trabajo, como para determinar las facilidades que aporta al introducir datos en el sistema y obtener los resultados.
- **Pruebas extremo a extremo (e2e):** consisten en interactuar con la aplicación como un usuario regular lo haría, cliente-servidor, y evaluando las respuestas para el comportamiento esperado.
- **Pruebas de operación:** consisten en comprobar la correcta implementación de los procedimientos de operación, incluyendo la planificación y control de trabajos, arranque y re-arranque del sistema, etc.
- **Pruebas de entorno:** consisten en verificar las interacciones del sistema con otros sistemas dentro del mismo entorno.
- **Pruebas de seguridad:** consisten en verificar los mecanismos de control de acceso al sistema para evitar alteraciones indebidas en los datos.

© JMA 2019. All rights reserved

## Pruebas de Aceptación

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

© JMA 2019. All rights reserved

## Pruebas de Aceptación

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

© JMA 2019. 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, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.
- Normalmente, este tipo de pruebas implica la repetición de las pruebas que ya se han realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

© JMA 2019. All rights reserved

## Pruebas de Regresión

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

© JMA 2019. All rights reserved

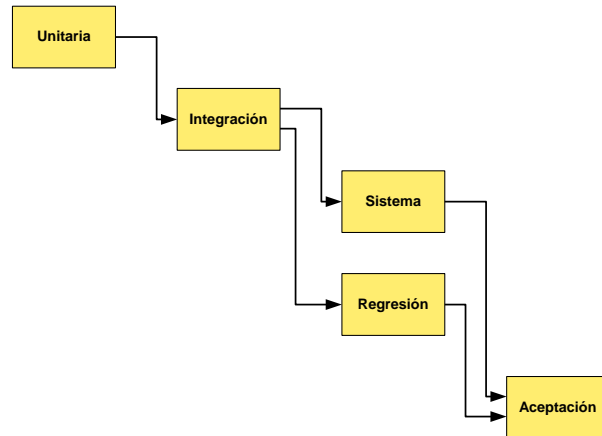
## Prueba exploratoria

- 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 sus pruebas automatizadas (como en el diseño o la usabilidad).
- Las pruebas exploratorias es un enfoque de prueba manual que enfatiza la libertad y creatividad del probador para detectar problemas de calidad en un sistema en ejecución. Simplemente tome un tiempo en un horario regular, arremangue e intente romper su aplicación. Use una mentalidad destructiva y encuentre formas de provocar problemas y errores en su aplicación. Documente todo lo que encuentre para más adelante. Tenga cuidado con los errores, los problemas de diseño, los tiempos de respuesta lentos, los mensajes de error faltantes o engañosos y todo lo que pueda molestarlo como usuario de su software.
- La buena noticia es que se puede automatizar la mayoría de sus hallazgos con pruebas automatizadas. Escribir pruebas automatizadas para los errores que se detectan asegura que no habrá regresiones de ese error en el futuro. Además, ayuda a reducir la causa raíz de ese problema durante la corrección de errores.

© JMA 2019. All rights reserved

## Niveles de pruebas y orden de ejecución.

- De tal forma que la secuencia de pruebas es:



© JMA 2019. All rights reserved

## Pirámide de pruebas



© JMA 2019. All rights reserved

## Principios F.I.R.S.T.

- El principio FIRST fue definido por Robert Cecil Martin en su libro Clean Code. Este autor, entre otras muchas cosas, es conocido por ser uno de los escritores del Agile Manifesto, escrito hace más de 15 años y que a día de hoy se sigue teniendo muy en cuenta a la hora de desarrollar software.
  - Fast: Los tests deben ser rápidos, del orden de milisegundos, hay cientos de tests en un proyecto.
  - Isolated/Independent (Aislado/Independiente): Los tests no deben depender del entorno ni de ejecuciones de tests anteriores.
  - Repeatable: Los tests deben ser repetibles y ante la misma entrada de datos, producen 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 2019. All rights reserved

## METODOLOGÍA

© JMA 2019. All rights reserved

## Introducción

- Durante las fases anteriores a la fase de pruebas se han dedicado a construir el sistema, partiendo de las especificaciones han realizado el análisis, el diseño y la implementación.
- En la fase de pruebas te dedicas a destruir el sistema. El objetivo es identificar el mayor número de posibles causas de fallo y probar si se producen errores.
- En la fase de pruebas **estudiaremos toda la documentación anteriormente generada**: el modelo de casos de uso con el documento de requisitos adicionales, el modelo de análisis, el modelo de diseño, el modelo de implementación y el documento de descripción de arquitectura.
- La calidad de dicha documentación es fundamental para la fase de pruebas, incidiendo directamente en la calidad del proceso de pruebas.

© JMA 2019. All rights reserved

## Introducción

- Una baja calidad en la documentación o la ausencia de algunos documentos degrada la validez del proceso de pruebas llegando incluso a imposibilitar dicho proceso.
- El **modelo de pruebas** describe cómo se prueban los componentes ejecutables del modelo de implementación con pruebas de integración y de sistema, así como aspectos específicos como puede ser la consistencia de la interfaz de usuario, si el manual del usuario cumple su cometido, y otros.
- A la fase de prueba se llega después de completar la fase de implementación, es cometido de dicha fase haber realizado las pruebas de unidad.

© JMA 2019. All rights reserved

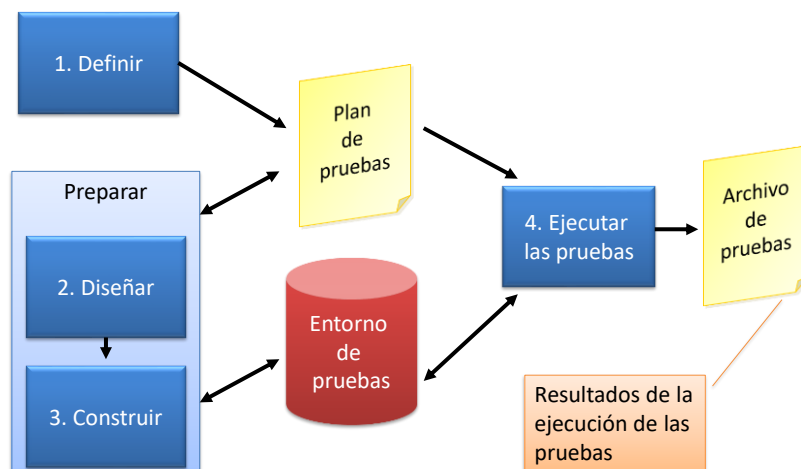


## Elementos

- Como vimos en su momento, la base de la fase de pruebas son los **casos de prueba**. Recuerda que 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.
- Un **procedimiento de prueba** especifica cómo realizar uno o varios casos de prueba o partes de éstos.
- En el procedimiento documentas 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.

© JMA 2019. All rights reserved

## Procedimiento de pruebas



© JMA 2019. All rights reserved

## Elementos

- Un **componente de prueba** 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.
- Los componentes de prueba son necesarios puesto que la mayoría de las veces el elemento que estas probando no dispone de un interfaz de usuario que te permita la comunicación directa con él.

© JMA 2019. All rights reserved

## Elementos

- Un **defecto** es una anomalía del sistema, como por ejemplo un síntoma de un fallo software o un problema descubierto en una revisión. Los defectos deben estar documentados indicando síntomas, posibles causas y consecuencias.
- Existen **herramientas específicas de prueba de software** que permiten establecer los procedimientos de pruebas, ejecutar los procedimientos y evaluar los resultados guardando un registro de los mismos. Si utilizas dichas herramientas, en muchos casos, evitarás la necesidad de desarrollar componentes de prueba.
- Por lo tanto, el modelo de prueba está compuesto de casos de prueba, procedimientos de prueba y componentes de prueba. El modelo se organiza mediante los planes de prueba.

© JMA 2019. All rights reserved

## Plan de pruebas

- La prueba de sistemas es cara.
- La creciente inclusión del software como un elemento más de muchos sistemas productivos y la importancia de los "costes" asociados a un fallo del mismo están motivando la creación de pruebas minuciosas y bien planificadas.
- No es raro que una organización de desarrollo de software gaste el 40 por 100 del esfuerzo total de un proyecto en la prueba.
- En casos extremos, la prueba del software para actividades críticas (por ejemplo, control de tráfico aéreo, o control de reactores nucleares) puede costar ¡de 3 a 5 veces más que el resto de los pasos de la ingeniería del software juntos!

© JMA 2019. All rights reserved

## Plan de pruebas

- Se necesita una planificación cuidadosa para obtener lo máximo del proceso de prueba y controlar los costes.
- El propósito del plan de pruebas es explicitar el alcance, enfoque, recursos requeridos, calendario, responsables y manejo de riesgos de un proceso de pruebas.
- Puede haber un plan global que explicita el énfasis a realizar sobre los distintos tipos de pruebas (verificación, integración).

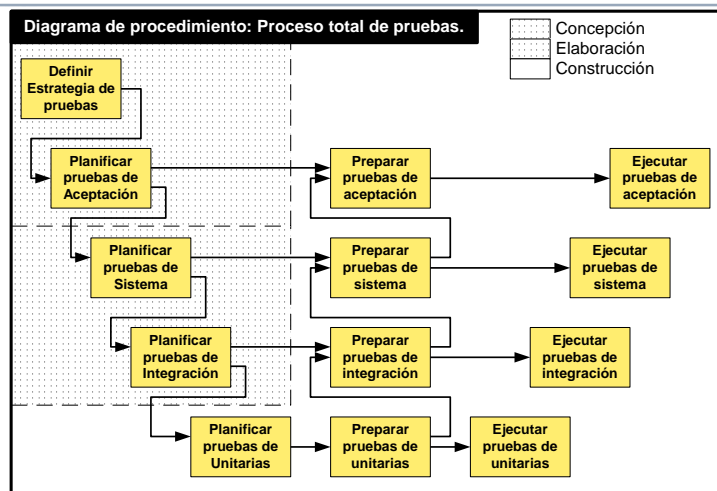
© JMA 2019. All rights reserved

## Características del plan de pruebas

- Este plan debería contemplar cantidades significativas de eventualidades, de forma que errores en el diseño o la implementación se puedan solucionar y parte del personal que realiza las pruebas se pueda dedicar a otras actividades.
- Como otros planes, el plan de prueba no es un documento estático. Debe revisarse regularmente puesto que la prueba es una actividad dependiente del avance de la implementación. Si parte del sistema a probar está incompleto, el proceso de prueba del sistema no puede comenzar.
- Debes contar con un plan de pruebas global y tantos planes de pruebas específicos como sean necesarios para cubrir el alcance del plan global.
- La planificación de las pruebas comienza desde las fases iniciales del desarrollo tal y como muestra el siguiente diagrama:

© JMA 2019. All rights reserved

## Proceso total de pruebas



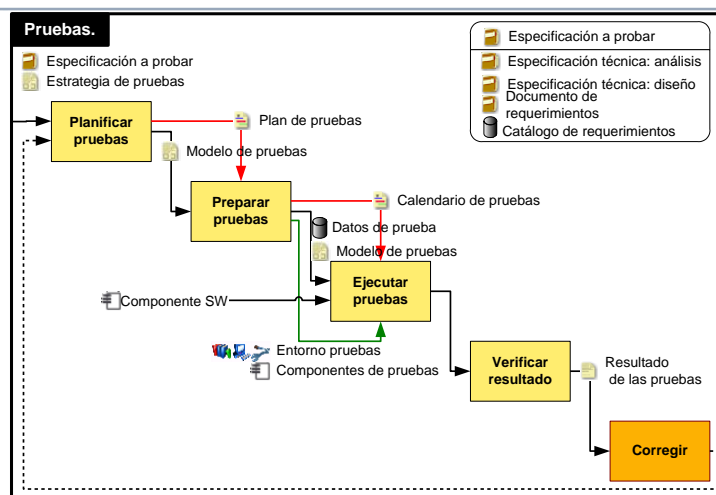
© JMA 2019. All rights reserved

# Metodología

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

© JMA 2019. All rights reserved

# Metodología



© JMA 2019. 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 2019. All rights reserved

## Las tres partes del test: AAA

- **Arrange (Preparar):**
  - Inicializa objetos y establece el valor de los datos que se pasa al método en pruebas de tal forma que los resultados sean predecibles.
- **Act (Actuar)**
  - Invoca al método a probar con los parámetros preparados.
- **Assert (Afirmar)**
  - Comprobar si la acción del método probado se comporta de la forma prevista. Puede tomar la forma de:
    - Aserción: Es una afirmación que se hace sobre el resultado y puede ser cierta o no.
    - Expectativa: Es la expresión del resultado esperado que puede cumplirse o no.

© JMA 2019. 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 2019. 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.

© JMA 2019. All rights reserved

## Arrange (Preparar)

- **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 2019. All rights reserved

## Objetos mock

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

© JMA 2019. All rights reserved



## 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 2019. All rights reserved

## Fundamentos de diseño

- Programar para la interfaz, no para la herencia.
- Favorecer la composición antes que la herencia.
- Delegación.
- Doble Herencia.
- 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.

© JMA 2019. All rights reserved

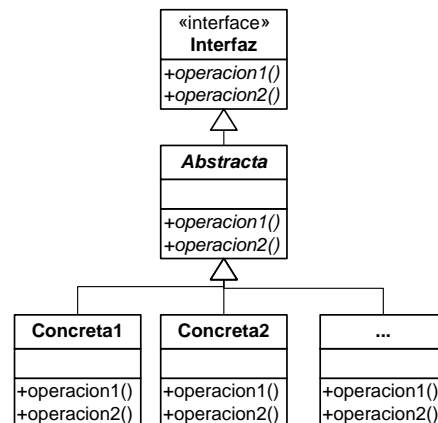
## Programar con interfaces

- Reutilizar la implementación de la clase base es la mitad de la historia.
- Ventajas:
  - Reducción de dependencias.
  - El cliente desconoce la implementación.
  - La vinculación se realiza en tiempo de ejecución.
  - Da consistencia (contrato).
- Desventaja:
  - Indireccionamiento.

© JMA 2019. 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 2019. 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.

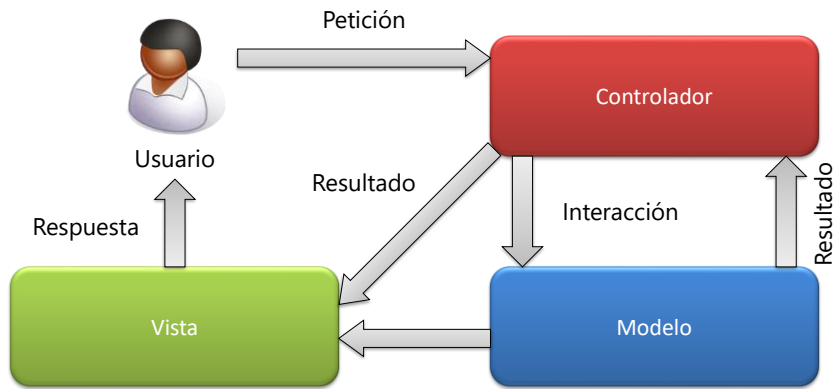
© JMA 2019. All rights reserved

## 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 2019. All rights reserved

## El patrón MVC



© JMA 2019. 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**

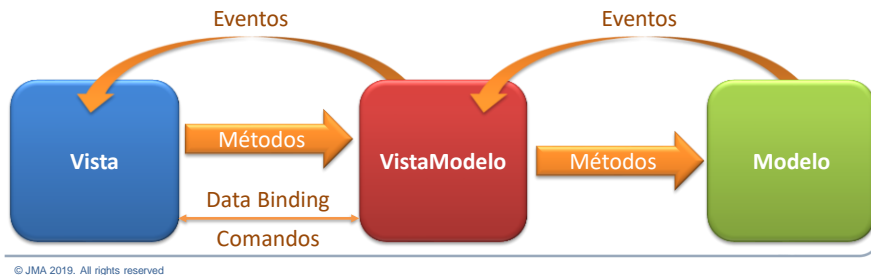


- **Intermediario** entre Modelo y Vista
- **Mapa acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

© JMA 2019. 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.



## ¿Cuáles son los beneficios del patrón MVVM?

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

## Desarrollo Guiado por Pruebas (TDD)

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

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved



---

# HERRAMIENTAS DE PRUEBA DE VISUAL STUDIO

---

© JMA 2019. 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, etc., 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 2019. 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 2019. 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).

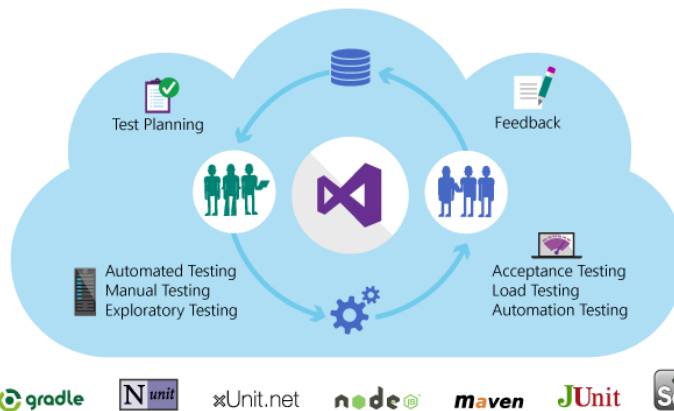
© JMA 2019. All rights reserved

## 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.

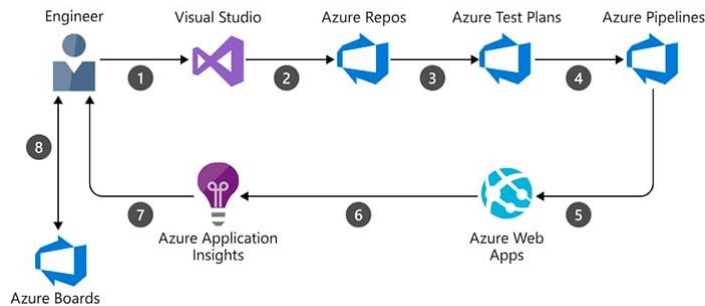
© JMA 2019. All rights reserved

## Azure DevOps



© JMA 2019. All rights reserved

# Azure DevOps



© JMA 2019. All rights reserved

# Azure DevOps

- Azure Repo: Repositorios privados de Git ilimitados, hospedados en la nube para su proyecto. Solicitudes de incorporación de cambios colaborativas, administración avanzada de archivos y mucho más.
- Azure Test Plans: Solución de pruebas planeadas y exploratorias todo en uno
- Azure Pipelines: CI/CD que funciona con cualquier lenguaje, plataforma y nube. Conéctese a GitHub o a cualquier otro repositorio de Git y lleve a cabo implementaciones continuas.
- Azure Board: Realice un seguimiento eficaz del trabajo con paneles kanban, trabajos pendientes, paneles de equipo e informes personalizados.
- Azure Artifacts: Los paquetes de Maven, npm y NuGet se alimentan de orígenes públicos y privados.
- Azure Web Apps es un servicio PaaS para hospedar aplicaciones web, API REST y back-ends para dispositivos móviles. No solo en .NET, hay varias opciones de plataforma de desarrollo adicionales compatibles.
- Application Insights es un servicio propio de Application Performance Management (APM) extensible para desarrolladores web en varias plataformas.

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## Marcos de pruebas

- 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 2019. All rights reserved

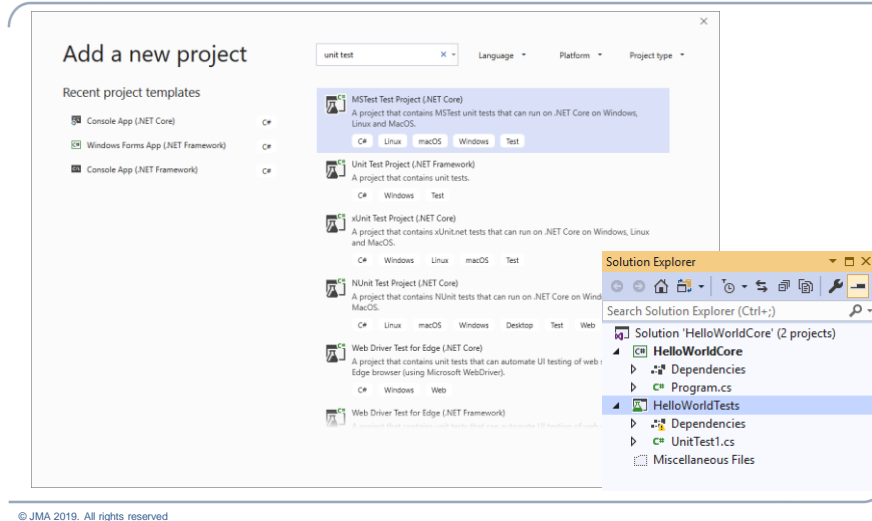
## Proyecto de prueba unitaria

- 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.

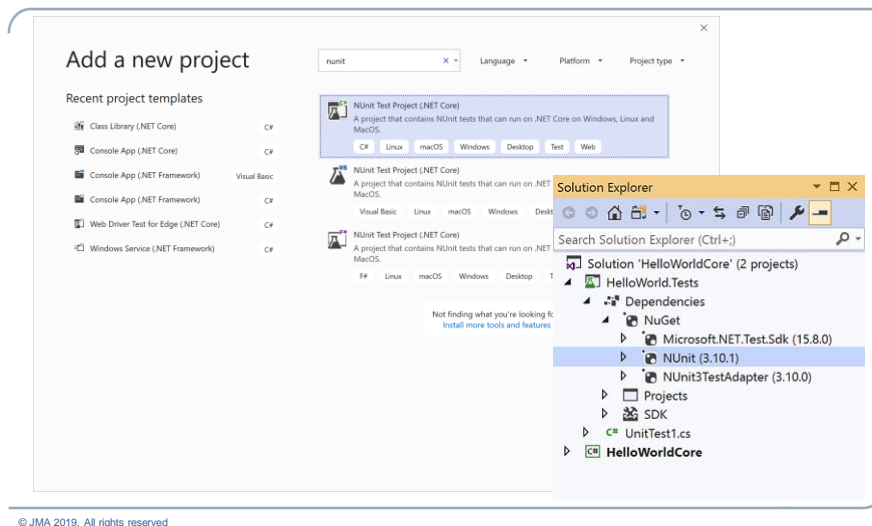
© JMA 2019. All rights reserved



# Proyecto de prueba unitaria

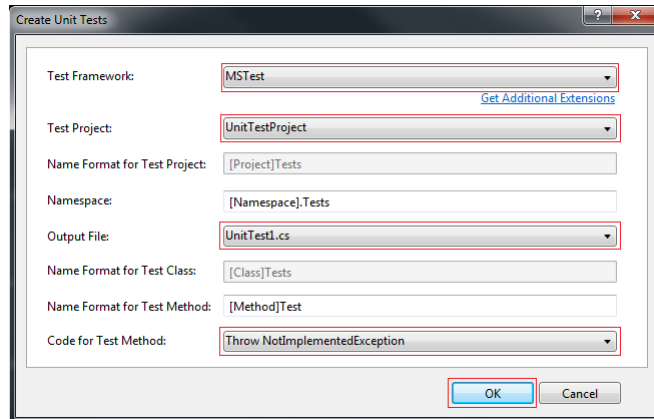


# Marcos de pruebas de terceros



## Generar proyecto de prueba unitaria

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



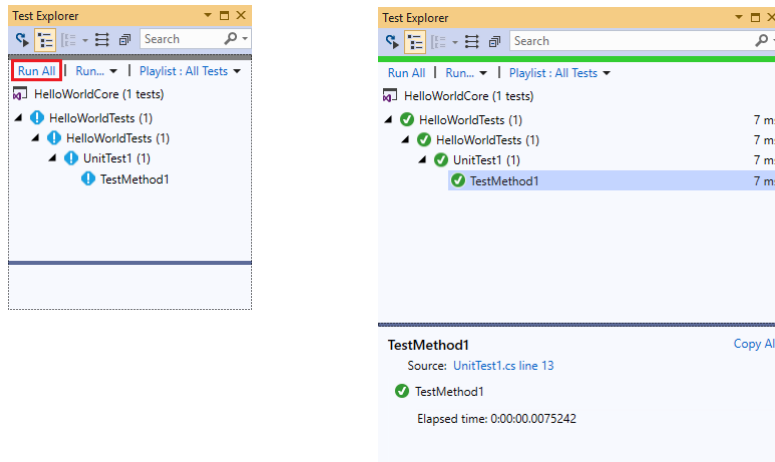
© JMA 2019. All rights reserved

## 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 2019. All rights reserved

## Ejecutar pruebas unitarias



© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

# Selenium

- <http://www.seleniumhq.org/>
- 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 de programación populares incluyendo Java, C#, 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. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias máquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

© JMA 2019. All rights reserved

## JMeter

- <http://jmeter.apache.org>
- JMeter es una herramienta de pruebas de carga para llevar a cabo 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 a cabo 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 2019. All rights reserved

## TestLink

- <http://testlink.org/>
- TestLink es una herramienta web de gestión de pruebas que ayuda a gestionar las pruebas funcionales de un proyecto permitiendo realizar las tareas relacionadas con el proceso de aseguramiento de calidad de software tales como: gestión de requerimientos, diseño de casos de prueba, planes de pruebas, ejecución de casos de prueba, seguimiento de informes de resultados del proceso de pruebas.
- Es una herramienta gratuita que permite crear y gestionar casos de pruebas y organizarlos en planes de prueba. Estos planes permiten a los miembros del equipo ejecutar casos de test y registrar los resultados dinámicamente, generar informes, mantener la trazabilidad con los requerimientos, así como priorizar y asignar tareas.
- Permite gestionar tantos proyectos de pruebas como sean necesarios, accesibles con diferentes roles de usuario con distintos permisos para los integrantes de un equipo de desarrollo.
- Los proyectos pueden definir diferentes plataformas de pruebas y mantener un inventario de las mismas.

© JMA 2019. All rights reserved

## Mantis

- <https://www.mantisbt.org/>
- Mantis Bug Tracker es un sistema de gestión de incidencias, control de cambios y control de errores, es una aplicación web OpenSource multiplataforma que permite gestionar las incidencias de la empresa, sistemas o proyectos.
- Es un sistema fácil de usar y adaptable a muchos escenarios, tanto para incidencias técnicas, peticiones de soporte o bugs de un sistema.
- Es una aplicación realizada con php y mysql que destaca por su facilidad y flexibilidad de instalar y configurar.
- Mantis es una aplicación que permite a distintos usuarios crear tickets de cualquier tipo.
- A la hora de notificar una incidencia, el usuario tiene muchas opciones y campos a rellenar con el fin de hacer más fácil el trabajo del encargado de resolver el ticket.
- Mantis permite notificar a los usuarios novedades por correo electrónico.
- Se puede especificar un número indeterminado de estados para cada tarea (nueva, se necesitan más datos, aceptada, confirmada, asignada, resuelta, cerrada) y configurar la transición de estados.
- Permite la carga de plugins específicos de la plataforma que añaden funcionalidades extra.

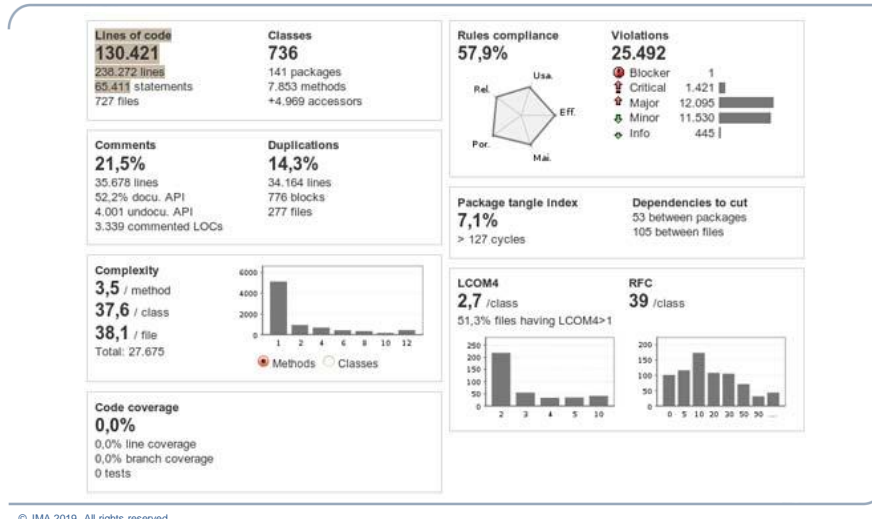
© JMA 2019. All rights reserved

## Sonar

- <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 2019. All rights reserved

# Sonar



# GenerateData

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



# Mockaroo

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

© JMA 2019. All rights reserved

## PRUEBAS: MStest

© JMA 2019. All rights reserved

## Introducción

- 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## Propiedades de TestContext

- **TestName:** Obtiene el nombre de la prueba.
- **FullyQualifiedTestClassName:** 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 cuando la prueba se utiliza 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 esta 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 2019. 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 2019. 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 las dos colecciones 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 rslt = privado.Invoke(arrange, new object[] { 2, 3 });
Assert.IsNotNull(rslt);
```

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

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

© JMA 2019. All rights reserved

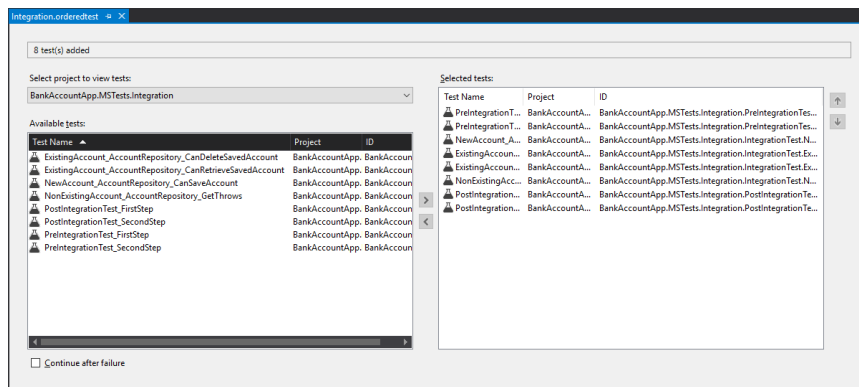
# Pruebas ordenadas (Obsoletas)

**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 2019. All rights reserved

## Pruebas ordenadas



© JMA 2019. 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 2019. All rights reserved

## DATA DRIVEN TESTING

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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=\\sqlexpress;Initial Catalog=tempdb;Integrated Security=True", "Data", DataAccessMethod.Sequential), TestMethod]

© JMA 2019. 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.TestTools.UnitTesting.Extensions" />
  </configSections>
  <connectionStrings>
    <add name="MyExcelConn" connectionString="Dsn=Excel Files;dbq=data.xlsx;defaultdir=.;
      driverid=790;maxbufferize=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 2019. All rights reserved

<https://nunit.org/>

## PRUEBAS: NUNIT

© JMA 2019. 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, la versión 3, 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 2019. 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.
  - 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 2019. 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 2019. 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 2019. 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:
  - [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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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
  - .Positive
  - .Negative
  - .InstanceOf
  - .IsNotInstanceOf
  - .AssignableFrom
  - .IsNotAssignableFrom
  - .Throws
- .ThrowsAsync
  - .DoesNotThrow
  - .DoesNotThrowAsync
  - .Catch
  - .CatchAsync
  - .Pass
  - .Fail
  - .Ignore
  - .Inconclusive
- StringAssert**
  - .Contains
  - .DoesNotContain
  - .StartsWith
  - .DoesNotStartsWith
  - .EndsWith
  - .DoesNotEndWith
  - .AreEqualIgnoringCase
  - .AreNotEqualIgnoringCase
  - .IsMatch
  - .DoesNotMatch
- CollectionAssert**
  - .AllItemsAreInstancesOfType
  - .AllItemsAreNotNull
  - .AllItemsAreUnique
- .AreEqual
  - .AreEquivalent
  - .AreNotEqual
  - .AreNotEquivalent
  - .Contains
  - .DoesNotContain
  - .IsSubsetOf
  - .IsNotSubsetOf
  - .IsEmpty
  - .IsNotEmpty
  - .IsOrdered
- FileAssert**
  - .AreEqual
  - .AreNotEqual
  - .Exists
  - .DoesNotExist
- DirectoryAssert**
  - .AreEqual
  - .AreNotEqual
  - .Exists
  - .DoesNotExist

© JMA 2019. 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`.
- 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 2019. 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 esta 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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:  

```
[TestMethod]
[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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 rsIt) {
```

© JMA 2019. 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 2019. 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 2019. 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 2019. All rights reserved

<https://xunit.net/>

**PRUEBAS: XUNIT.NET**

© JMA 2019. All rights reserved

---

Visual Studio Enterprise

## MICROSOFT FAKES

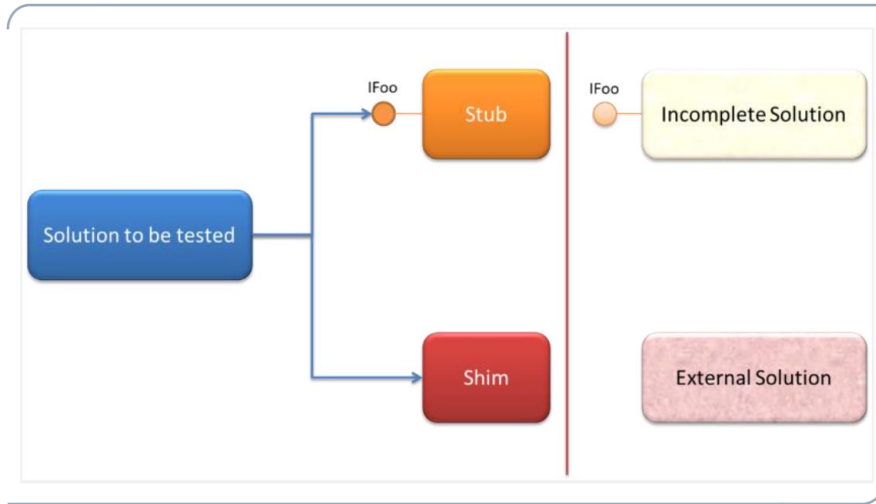
© JMA 2019. All rights reserved

## 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 2019. All rights reserved

## Escenarios



© JMA 2019. 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 o crearse mediante un generador.

© JMA 2019. 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 2019. 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 2019. 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 archivos .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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

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

**moq**

© JMA 2019. 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.

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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.InRange<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.In<int>(4,5,6))).Returns("set");
mock.Setup(obj => obj.Get(It.NotIn<int>(4,5,6))).Returns("unset");
```

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## Callbacks

- Moq permite, mediante el método Callbacks, hacer un seguimiento de las llamada 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 2019. 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 2019. All rights reserved

## MÉTRICAS PARA EL PROCESO DE PRUEBAS DE SOFTWARE

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 medir 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## BUENAS PRACTICAS

© JMA 2019. 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. Milisegundos.
- Aislada. Las pruebas unitarias son independientes, se pueden ejecutar de forma aislada y no tienen ninguna dependencia en ningún factor externo, como sistemas de archivos o bases de datos.
- Reiterativa. La ejecución de una prueba unitaria debe ser coherente con sus resultados, es decir, devolver siempre el mismo resultado si no cambia nada entre ejecuciones.
- Autocomprobada. La prueba debe ser capaz de detectar automáticamente si el resultado ha sido correcto o incorrecto sin necesidad de intervención humana.
- Oportuna. Una prueba unitaria no debe tardar un tiempo desproporcionado en escribirse en comparación con el código que se va a probar. Si observa que la prueba del código tarda mucho en comparación con su escritura, considere un diseño más fácil de probar.

© JMA 2019. 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 2019. 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.
- Las secciones solo deben una vez como máximo y en el orden establecido.
- Minimiza la posibilidad de mezclar aserciones con el código para "actuar".

© JMA 2019. 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 2019. 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 2019. All rights reserved

## Sustituir literales por constantes

- La asignación de literales a constantes permite dar nombre a los valores, aportando semántica.
- Evita la necesidad de que el lector de la prueba inspeccione el código de producción con el fin de averiguar lo que significa un valor, que hace el valor sea especial.  
`Assert.IsTrue(rslt.Length <= 10)`
- Muestra explícitamente lo que se intenta probar, en lugar de lo que se intenta lograr.  
`const string VARCHAR_LEN = 10;`  
`Assert.IsTrue(rslt.Length <= VARCHAR_LEN)`
- 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.

© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## Cubrir aspectos no evidentes

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

© JMA 2019. 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 2019. All rights reserved

---

Visual Studio Enterprise

## INTELLITEST

© JMA 2019. All rights reserved

## PEX

- Pex (P rogram Ex ploration) 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, a partir de Visual Studio 2015, es la evolución de Pex.

© JMA 2019. 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 IntelliTest, 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 2019. 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 límites 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 válidos, ...
- 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 2019. All rights reserved

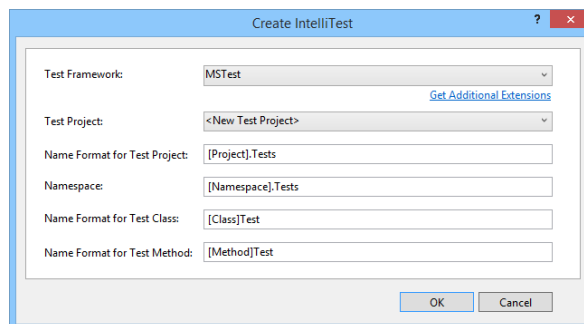
## Limitaciones

- 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 2019. All rights reserved

## Explorar el código

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



© JMA 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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 2019. 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.ClassifyBySideLengt...

© JMA 2019. 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 2019. All rights reserved

## Configuración en cascada

- El concepto de configuración en cascada de 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 2019. 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 2019. 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 2019. All rights reserved

## PRUEBAS DE ACCESO A DATOS

© JMA 2019. 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 2019. 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 pone en un estado conocido antes de cada test.
- Usar múltiples conjuntos de datos pequeños en lugar de uno grande. Cada prueba necesita un conjunto de tablas y de registros, no necesariamente toda la base de datos.
- Inicializar los datos comunes sólo una vez para todos los tests. Si hay datos que sólo son de lectura, no tenemos por qué tocarlos si nos aseguramos de que las pruebas no los modifican.

© JMA 2019. 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 2019. All rights reserved

## Ciclo de vida

- El ciclo de vida de las pruebas es el siguiente:
  1. Eliminar el estado previo de la BD resultante de pruebas anteriores (en lugar de restaurarla tras cada test).
  2. Cargar los datos necesarios para las pruebas de la BD (sólo los necesarios para cada test).
- 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 2019. 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 2019. 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 2019. 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 2019. All rights reserved

## Patrones DDD

- **Repositorio:** Martin Fowler dice que un 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:** Martin Fowler dice que una 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:** Martin Fowler utiliza el nombre lazy load para describir "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. Por ejemplo, considere el código siguiente.

© JMA 2019. All rights reserved

## Framework EF Mock

- <https://www.nuget.org/packages/Effort.EF6/>
- <https://www.nuget.org/packages/EfCore.TestSupport/>
- <https://www.nuget.org/packages/EntityFrameworkTesting/>
- <https://www.nuget.org/packages/EntityFramework.MoqHelper/>

© JMA 2019. All rights reserved

## TEST DRIVEN DEVELOPMENT

© JMA 2019. 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 2019. 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.

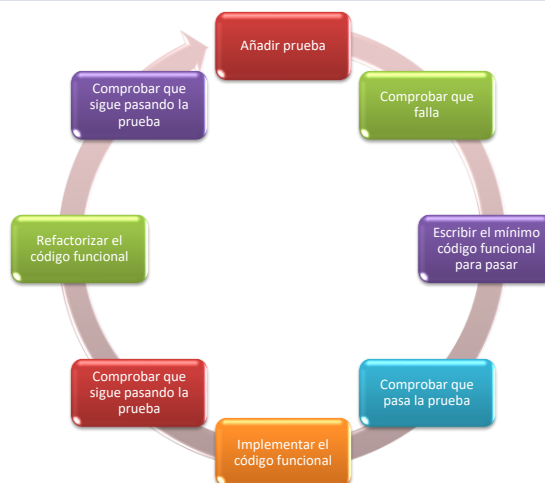
© JMA 2019. All rights reserved

## 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 2019. All rights reserved

## Ritmo TDD



© JMA 2019. All rights reserved

## Ciclo de vida

1. Crear los proyectos de código fuente y de pruebas.
2. Agregar "Prueba unitaria ..." al proyecto de prueba: NuevoTipoTests
3. Agregar nuevo método de prueba a la clase de prueba recién creada: NuevoMetodoTest
4. Codificar el nuevo método de prueba que compruebe el resultado obtenido para una entrada específica.  

```
[TestMethod()]
public void NuevoMetodoTest() {
    var arrange = new NuevoTipo();
    Assert.AreEqual("OK", arrange.NuevoMetodo());
}
```
5. Generar el tipo a partir del código de prueba colocando el cursor en NuevoTipo y, en el menú de bombilla, Generar nuevo tipo. En el cuadro de diálogo Generar tipo, establecer el proyecto de código fuente como destino.

© JMA 2019. All rights reserved

## Ciclo de vida

6. Generar el método a partir del código de prueba colocando el cursor en NuevoMetodo y, en el menú de bombilla, Generar método.
7. Ejecutar la prueba unitaria, la prueba se ejecuta pero no se supera: Se produce excepción NotImplementedException.
8. A partir del código de prueba colocando el cursor en NuevoMetodo y, clic derecho, Ir a definición. Sustituir el código generado por código constante que supere la aserción.  

```
public string NuevoMetodo() {
    return "OK";
}
```
9. Ejecutar la prueba unitaria, la prueba se ejecuta y se supera. En caso contrario depurar el código del método de prueba y el código constante para volver a ejecutar la prueba unitaria hasta que se supere.
10. Una vez comprobado el correcto funcionamiento, se sustituye el código constante por la implementación real.

© JMA 2019. All rights reserved

## Ciclo de vida

11. Ejecutar la prueba unitaria y hay que modificar la implementación hasta que se supere la prueba, sin modificar el código del método de prueba.
12. Se pueden añadir nuevos métodos de prueba que cubran el casos de prueba completo. Hay que evitar modificar las pruebas existentes que se completan correctamente.
13. Refactorizar el código implementado para mejorarlo sin añadir funcionalidad ni modificar el código del método de prueba.
14. Ejecutar la prueba unitaria que debe seguir siendo superada, en caso contrario habrá modificar código refactorizado hasta que se supere la prueba.

© JMA 2019. 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 2019. All rights reserved

---

# BEHAVIOUR DRIVER DEVELOPMENT

---

© JMA 2019. All rights reserved

## Pruebas de Aceptación: 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 2019. All rights reserved



## Gherkin

- Gherkin, es un lenguaje comprensible por humanos y por ordenadores, con el que vamos a describir las funcionalidades, definiendo el comportamiento del software, sin entrar en su implementación. Se trata de un lenguaje fácil de leer, fácil de entender y fácil de escribir. Es un lenguaje de los que Martin Fowler llama 'Business Readable DSL', es decir, 'Lenguaje Específico de Dominio legible por Negocio'.
- Utilizar Gherkin nos va a permitir crear una documentación viva a la vez que automatizamos los tests, haciéndolo además con un lenguaje que puede entender negocio.
- 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:

© JMA 2019. All rights reserved

## Gherkin

- 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 2019. All rights reserved

## Cucumber

- Cucumber es una de las herramientas que podemos utilizar para automatizar nuestras pruebas en BDD. Cucumber nos va permitir ejecutar descripciones funcionales en Gherkin como pruebas de software automatizadas.
- Cucumber fue creada en 2008 por Aslak Hellesoy y está escrito en Ruby, aunque tiene implementaciones para casi cualquier lenguaje de programación: JRuby (usando Cucumber-JVM), Java, Groovy, JavaScript, JavaScript (usando Cucumber-JVM y Rhino), Clojure, Gosu, Lua, .NET (usando SpecFlow), PHP (usando Behat), Jython, C++ o Tcl.
- Cucumer es probablemente la herramienta más conocida y más utilizada para automatizar pruebas en BDD, pero existen otros frameworks con los que poder hacer BDD para los lenguajes de programación más habituales.

© JMA 2019. All rights reserved

## SpecFlow

<https://specflow.org/>

- Es una herramienta de código abierto para proyectos .NET que se creo 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 2019. All rights reserved

<https://www.selenium.dev>

# SELENIUM

© JMA 2019. All rights reserved

## Introducción

- 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 de programación populares incluyendo Java, C#, 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. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias máquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

© JMA 2019. All rights reserved

# Historia

- Cuando Selenium 1 fue lanzado en el año 2004, surgió por la necesidad de reducir el tiempo dedicado a verificar manualmente el comportamiento consistente en el front-end de una aplicación web. Hizo uso de las herramientas disponibles en ese momento y confió en gran medida en la inyección de JavaScript en la página web bajo prueba para emular la interacción de un usuario.
- Si bien JavaScript es una buena herramienta para permitirte la introspección de las propiedades del DOM y para hacer ciertas observaciones del lado del cliente que de otro modo no se podría hacer, se queda corto en la capacidad de replicar naturalmente las interacciones de un usuario como usar el mouse y el teclado.
- Desde entonces, Selenium ha crecido y ha madurado bastante, convirtiéndose en una herramienta ampliamente utilizada. Selenium ha pasado de ser de un kit de herramientas de automatización de pruebas de fabricación casera a la librería de facto de automatización de navegadores del mundo.
- Así como Selenium RC hizo uso de las herramientas de oficio disponibles en ese momento, Selenium WebDriver impulsa esta tradición al llevar la parte de la interacción del navegador al territorio del proveedor del mismo y pedirles que se responsabilicen de las implementaciones de back-end orientadas al navegador. Recientemente este esfuerzo se ha convertido en un proceso de estandarización del W3C donde el objetivo es convertir el componente WebDriver en Selenium en la librería de control remoto de tu agente de usuario.

© JMA 2019. All rights reserved

## Selenium controla los navegadores web

- Selenium significa muchas cosas pero en su núcleo, es un conjunto de herramientas para la automatización de navegadores web que utiliza las mejores técnicas disponibles para controlar remotamente las instancias de los navegadores y emular la interacción del usuario con el navegador.
- Permite al código simular interacciones básicas realizadas por los usuarios finales; insertando texto en los campos, seleccionando valores de menús desplegables y casillas de verificación, y haciendo clics en los enlaces de los documentos. También provee muchos otros controles tales como el movimiento del mouse, la ejecución arbitraria de JavaScript y mucho más.
- A pesar de que es usado principalmente para pruebas de front-end de sitios webs, Selenium es en esencia una librería de agente de usuario para el navegador. Las interfaces son ubicuas a su aplicación, lo que fomenta la composición con otras librerías para adaptarse a su propósito.

© JMA 2019. All rights reserved

## Un API para gobernarlos a todos

- Uno de los principios fundamentales del proyecto es permitir una interfaz común para todas las tecnologías de los (principales) navegadores.
- Los navegadores web son aplicaciones increíblemente complejas y de mucha ingeniería, realizando operaciones completamente diferentes pero que usualmente se ven iguales al hacerlo. Aunque el texto se presente con las mismas fuentes, las imágenes se muestren en el mismo lugar y los enlaces te llevan al mismo destino.
- Lo que sucede por debajo es tan diferente como la noche y el día. Selenium abstraer estas diferencias, ocultando sus detalles y complejidades a la persona que escribe el código. Esto le permite escribir varias líneas de código para realizar un flujo de trabajo complicado, pero estas mismas líneas se ejecutarán en Firefox, Internet Explorer, Chrome y los demás navegadores compatibles.

© JMA 2019. All rights reserved

## Herramientas y soporte

- El diseño minimalista de Selenium le da la versatilidad para que se pueda incluir como un componente en otras aplicaciones. La infraestructura proporcionada debajo del catálogo de Selenium te da las herramientas para que puedas ensamblar tu grid de navegadores de modo que tus pruebas se puedan ejecutar en diferentes navegadores a través de diferentes sistemas operativos y plataformas.
- Imagina un granja de servidores en tu sala de servidores o en un centro de datos, todos ejecutando navegadores al mismo tiempo e interactuando con los enlaces en tu sitio web, formularios y tablas—probando tu aplicación 24 horas al día. A través de la sencilla interfaz de programación proporcionada para los lenguajes más comunes, estas pruebas se ejecutarán incansablemente en paralelo, reportando cuando ocurran errores.
- Es un objetivo ayudar a que esto sea una realidad para ti, proporcionando a los usuarios herramientas y documentación para controlar no solo los navegadores pero también para facilitar la escalabilidad e implementación de tales grids.

© JMA 2019. All rights reserved

# Automatización de pruebas

- Las pruebas funcionales de usuario final, como las pruebas de Selenium son caras de ejecutar, requieren abrir un navegador e interactuar con él. 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.
- Las pruebas manuales son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización. Selenium permite ejecutar y repetir las mismas pruebas en múltiples navegadores de diferentes sistemas operativos.
- Una vez tomada la decisión de hacer pruebas en el navegador, y que tengas tu ambiente de Selenium listo para empezar a escribir pruebas, generalmente realizaras alguna combinación de estos tres pasos:
  - Preparar los datos
  - Realizar un conjunto discreto de acciones
  - Evaluar los resultados
- Querrás mantener estos pasos tan cortos como sea posible; una o dos operaciones deberían ser suficientes la mayor parte del tiempo. La automatización del navegador tiene la reputación de ser "frágil", pero en realidad, esto se debe a que los usuarios suelen exigirle demasiado.
- Manteniendo las pruebas cortas y usando el navegador web solo cuando no tienes absolutamente ninguna alternativa, puedes tener muchas pruebas con fragilidad mínima.
- Una clara ventaja de las pruebas de Selenium es su capacidad inherente para probar todos los componentes de la aplicación, desde el backend hasta el frontend, desde la perspectiva del usuario. En otras palabras, aunque las pruebas funcionales pueden ser caras de ejecutar, también abarcan a la vez grandes porciones críticas para el negocio.

© JMA 2019. All rights reserved

## Tipos de pruebas

- **Pruebas funcionales:** Este tipo de prueba se realiza para determinar si una funcionalidad o sistema funciona correctamente y sin problemas. Se comprueba el sistema en diferentes niveles para garantizar que todos los escenarios están cubiertos y que el sistema hace lo que se supone que debe de hacer. Es una actividad de verificación que responde la pregunta: ¿Estamos construyendo el producto correctamente?
  - Para aplicaciones web, la automatización de esta prueba puede ser hecha directamente con Selenium simulando los retornos esperados. Esta simulación podría hacerse mediante grabación/reproducción o mediante los diferentes lenguajes soportados.
- **Pruebas de aceptación:** Este tipo de prueba se realiza para determinar si una funcionalidad o un sistema cumple con las expectativas y requerimientos del cliente. Este tipo de pruebas implican la cooperación o retroalimentación del cliente, siendo una actividad de validación que responde la pregunta: ¿Me están construyendo el producto correcto?
  - Las pruebas de aceptación son un subtipo de pruebas funcionales, por lo que la automatización se puede hacer directamente con Selenium simulando el comportamiento esperado del usuario mediante grabación/ reproducción.

© JMA 2019. All rights reserved

## Tipos de pruebas

- Pruebas de regresión: Este tipo de pruebas generalmente se realiza después de un cambio, corrección o adición de funcionalidad.
  - Para garantizar que el cambio no ha roto ninguna de las funcionalidades existentes, algunas pruebas ya ejecutadas se ejecutan nuevamente. El conjunto de pruebas ejecutadas nuevamente puede ser total o parcial, y puede incluir varios tipos diferentes, dependiendo del equipo de desarrollo y la aplicación.
- Las pruebas no funcionales tales como rendimiento, de carga, de estrés, ... aunque se pueden realizar con Selenium hay otras opciones, como Jmeter, que las realizan mas eficientemente y además suministran métricas que incluyen rendimiento, latencia, pérdida de datos, tiempos de carga de componentes individuales.
- De igual forma, las pruebas unitarias se pueden realizar con Selenium pero, como se deben ejecutar continuamente, suele ser demasiado costoso.

© JMA 2019. All rights reserved

## Selenium IDE

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas. Es una solución simple y llave en mano para crear rápidamente pruebas de extremo a extremo confiables.
- Selenium IDE no requiere una configuración adicional aparte de instalar una extensión en el navegador pero solo está disponible para Firefox y Chrome.
- Selenium IDE es muy flexible, registra múltiples localizadores para cada elemento con el que interactúa, si un localizador falla durante la reproducción, los demás se probarán hasta que uno tenga éxito.
- Mediante el uso del comando de ejecución se puede reutilizar un caso de prueba dentro de otro.
- Dispone una estructura de flujo de control extensa, con comandos condicionales, bucles, ... que permite modelizar escenarios complejos.

© JMA 2019. 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, C# 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 2019. All rights reserved

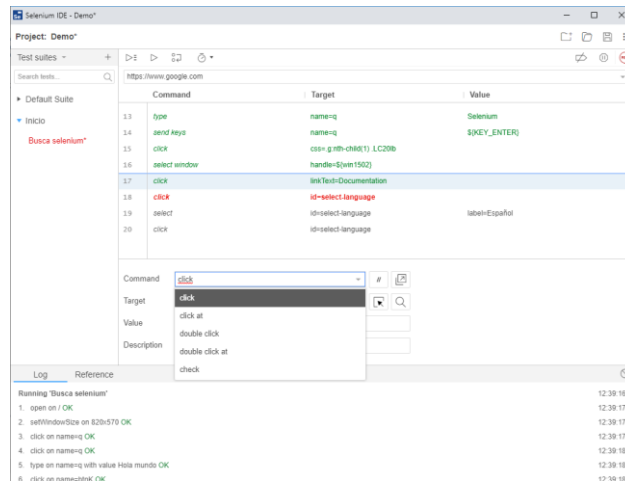
## Empezando

- Instale Selenium IDE desde la tienda web Chrome o Firefox .
- Una vez instalado, inícielo haciendo clic en su icono desde la barra de menú de su navegador.
- Aparecerá un cuadro de diálogo de bienvenida con las siguientes opciones:
  - Grabar una nueva prueba en un nuevo proyecto
  - Abrir un proyecto existente
  - Crea un nuevo proyecto
  - Cierra el IDE
- Si se da grabar, pedirá el nombre del proyecto y la URL base de la aplicación que se está probando (se utiliza en todas las pruebas del proyecto).
- Después de completar esta configuración, se abrirá la ventana IDE y una nueva ventana del navegador, cargará la URL base y comenzará a grabar. Interactúe con la página y cada una de las acciones se registrará en el IDE. Para detener la grabación, cambie a la ventana IDE y haga clic en el icono de grabación.

© JMA 2019. All rights reserved



# Empezando



© JMA 2019. All rights reserved

# Empezando

- La grabación se puede editar, modificar, reproducir, depurar, guardar y exportar.
- Cada grabación es una prueba. El proyecto puede contener múltiples pruebas que se organizan agrupándolas en suites. Por defecto se crea la "Default Suite" a la que se le asigna la prueba de la grabación inicial. Las pruebas y las suites se pueden renombrar, ejecutar o eliminar.
- El proyecto se puede descartar o guardar un único archivo JSON con una extensión .side para recuperarlo posteriormente o ejecutarlo en la línea de comandos.
- La reproducción de la prueba en el IDE permite establecer puntos de ruptura e inspeccionar valores.
- Se puede reanudar la grabación en cualquier punto.

© JMA 2019. All rights reserved

## Scripts

- 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.
- 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 2019. All rights reserved

## Localizadores

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

© JMA 2019. All rights reserved

## Acciones

- Sobre elementos de los formularios:
  - click, click at, double click, double click at, check, uncheck, edit content, send keys, type, select, add selection, remove selection, submit
- Del ratón:
  - mouse move at, mouse over, mouse out, mouse down, mouse down at, mouse up, mouse up at, drag and drop to object
- Sobre cuadros de dialogo alert, prompt y confirm:
  - choose ok on next confirmation, choose cancel on next confirmation, choose cancel on next prompt, webdriver choose ok on visible confirmation, webdriver choose cancel on visible confirmation, webdriver choose cancel on visible prompt
- Sobre la ejecución de la prueba:
  - open, execute script, run, echo, debugger, pause, close

© JMA 2019. 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 2019. All rights reserved

## Flujo de control

- Selenium IDE viene con comandos que permiten agregar lógica condicional y bucles a las pruebas, para ejecutar comandos (o un conjunto de comandos) solo cuando se cumplen ciertas condiciones o repetidamente en función de criterios predefinidos.
- Las condiciones en su aplicación se verifican mediante el uso de expresiones de JavaScript.
- Los comandos de flujo de control Control Flow funcionan son comandos de apertura y cierre para denotar un conjunto (o bloque) de comandos.
- Aquí están cada uno de los comandos de flujo de control disponibles acompañados de sus comandos complementarios de cierre.
  - `if ... else if ... else ... end` // else if y else son opcionales
  - `times ... end` // Bucle for
  - `forEach ... end`
  - `while ... end`
  - `do ... repeat if` // la condicion en repeat if
- Se pueden anidar los comandos de control de flujo según sea necesario.

© JMA 2019. 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 todas 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 2019. All rights reserved

## Ejecutar en línea de comandos

- Requiere tener instalado NodeJS (<https://nodejs.org>)
- Instalar CLI
  - `npm install -g selenium-side-runner`
- 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 2019. All rights reserved

## Exportación

- Se puede exportar una prueba o un conjunto de pruebas a código de WebDriver haciendo clic con el botón derecho en una prueba o un conjunto, seleccionando Export, eligiendo el idioma de destino y haciendo clic Export.
- Actualmente, se admite la exportación a los siguientes idiomas y marcos de prueba.
  - C# NUnit
  - C# xUnit
  - Java JUnit
  - JavaScript Mocha
  - Python pytest
  - Ruby RSpec
- Esta previsto admitir todos los lenguaje de programación soportados oficialmente para Selenium en al menos un marco de prueba para cada lenguaje.

© JMA 2019. 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 2019. All rights reserved

## Patrón Page Objects

- <https://martinfowler.com/bliki/PageObject.html>
- Cuando se escriben pruebas de una página web, hay que acceder a los elementos dentro de esa página web para hacer clic en los elementos, teclear entradas y determinar lo que se muestra.
- Sin embargo, si se escriben pruebas que manipulan los elementos HTML directamente, las pruebas serán frágiles ante los cambios en la interfaz de usuario.
- Un objeto de página envuelve una página HTML, o un fragmento, con una API específica de la aplicación, lo que permite manipular los elementos de la página sin excavar en el HTML.

© JMA 2019. All rights reserved

## Patrón Page Objects

- La regla básica para un objeto de página es que debe permitir que un cliente de software haga cualquier cosa y vea todo lo que un humano puede hacer.
- El objeto de página debe proporcionar una interfaz que sea fácil de programar y oculta en la ventana.
- Entonces, para acceder a un campo de texto, debe tener métodos de acceso que tomen y devuelvan una cadena, las casillas de verificación deben usar valores booleanos y los botones deben estar representados por nombres de métodos orientados a la acción.
- El objeto de la página debe encapsular los mecanismos necesarios para encontrar y manipular los datos en el propio control gui.
- A pesar del término objeto de "página", estos objetos no deberían construirse para cada página, sino para los elementos significativos en una página.
- Los problemas de concurrencia y asincronía son otro tema que un objeto de página puede encapsular.

© JMA 2019. All rights reserved

## Ventajas del patrón Page Objects

- De acuerdo con patrón Page Object, deberíamos mantener nuestras pruebas y localizadores de elementos por separado, esto mantendrá el código limpio y fácil de entender y mantener.
- El enfoque Page Object hace que el programador de marcos de automatización de pruebas sea más fácil, duradero y completo.
- Otra ventaja importante es que nuestro repositorio de objetos de página es independiente de las pruebas de automatización. Mantener un repositorio separado para los objetos de la página nos ayuda a usar este repositorio para diferentes propósitos con diferentes marcos como, podemos integrar este repositorio con otras herramientas como JUnit / NUnit / PHPUnit, así como con TestNG / Cucumber / etc.
- Los casos de prueba se vuelven cortos y optimizados, ya que podemos reutilizar los métodos de objetos de página.
- Los casos de prueba se centran solamente en el comportamiento.
- Cualquier cambio en la IU se puede implementar, actualizar y mantener fácilmente en los objetos y clases de página sin afectar a los casos de pruebas que no estén implicados.

© JMA 2019. All rights reserved

Encuesta

**<https://tecnofor.es/evaluacion-2173>**

© JMA 2019. All rights reserved