


Buenas Practicas en Desarrollo de Software



© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

Introducción

- Clean Code es el título de un libro escrito por Robert C. Martin (Uncle Bob) en 2008 donde nos habla de cómo escribir «código limpio», el código bien estructurado, fácil de comprender, robusto y, a su vez, fácil de mantener. Ese código donde no hay código duplicado. Código que se apoya en patrones de diseño.
- El mensaje principal del libro es: El código limpio (clean code), no es algo recomendado o deseable, es algo vital para las compañías y los programadores. La razón es que cada vez que alguien escribe código enmarañado y sin pruebas unitarias (código no limpio), otros tantos programadores pierden mucho tiempo intentando comprenderlo. Incluso el propio creador de un código no limpio si lo lee a los 6 meses será incapaz de comprenderlo y por tanto de evolucionarlo.
- El clean code no es una metodología, es una filosofía, una forma de pensar y entender como se debe escribir el código, que debe ser interiorizada y practicada de forma natural.

© JMA 2020. All rights reserved

Destinatarios

- El Clean Code está dirigido para todos aquellos desarrolladores de software que deseen mejorar el diseño y la calidad de su código. No importa el nivel que tengan.
- Para los desarrolladores más noveles, debería ser el punto de partida de su aprendizaje para que empiecen a forjar su propio estilo sin vicios adquiridos.
- Para los desarrolladores más experimentados, supone una revisión formal de determinados aspectos que les permitirá valorar pros y contras, que mejoren su calidad.
- Es especialmente interesante para desarrolladores que trabajan habitualmente en equipo ya que intenta sensibilizarnos acerca de la importancia de escribir código para que lo entiendan los demás.

© JMA 2020. All rights reserved

Contexto

- El Clean Code es parte de un contexto mas amplio que incluye:
 - Agile Software Development, Principles, Patterns, and Practices
 - Principios SOLID
 - Test Driven Development
 - Arquitectura Clean

© JMA 2020. All rights reserved

Martin Fowler: Deuda técnica

- Del mismo modo que una empresa incurre en una deuda para aprovechar una oportunidad de mercado, los desarrolladores pueden incurrir en deuda técnica para cumplir un plazo importante. El mayor costo de la deuda técnica es el hecho de que ralentiza su capacidad para ofrecer funciones futuras.
- Al igual que una deuda financiera, la deuda técnica incurre en pagos de intereses, que vienen en la forma del esfuerzo adicional que tenemos que hacer en el desarrollo futuro debido a la elección de diseño rápida y sucia. Podemos elegir continuar pagando los intereses, o podemos pagar el principal refactorizando el diseño rápido y sucio en el mejor diseño. Aunque cuesta pagar el principal, ganamos por pagos de intereses reducidos en el futuro. Si la deuda crece lo suficiente, eventualmente la compañía gastará más en el servicio de su deuda de lo que invierte en aumentar el valor de sus otros activos. La deuda técnica acumulada se convierte en un gran desincentivo para trabajar en un proyecto.

© JMA 2020. All rights reserved

Deuda técnica

Causas

- Código heredado
- Prisas
- Presión de fechas de entrega
- Desconocimiento
- Carencia de obligación/rutina

Consecuencias

- Código duplicado
- Código sucio
- Código espagueti
- Escasez/inexistencia de pruebas
- Escasez/inexistencia de documentación
- Proyectos favelas

© JMA 2020. All rights reserved

Deuda técnica

Intereses de la deuda

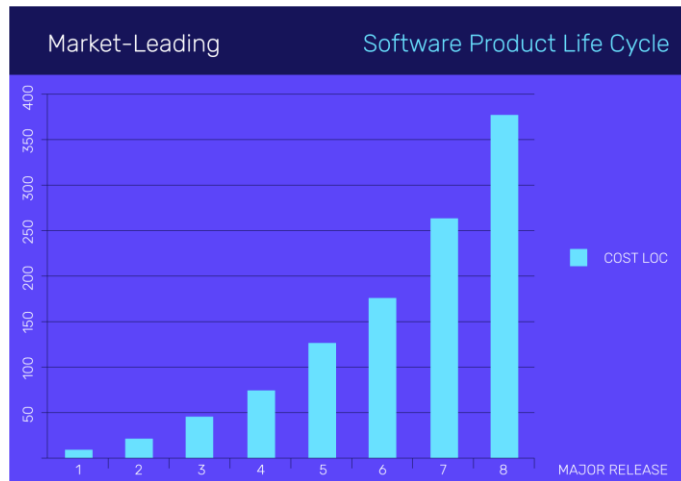
- Sobreesfuerzos del equipo para el mantenimiento y la evolución del proyecto.
- Ralentización
- Malestar, estrés y abandonos del equipo.
- Mala imagen.
- Penalizaciones.

Inversión

- Hacer código
 - más limpio
 - con test
 - con documentación
- El mayor coste de un proyecto de software es su mantenimiento a largo plazo
- Aumentar el coste inicial disminuirá, y mucho, el coste final

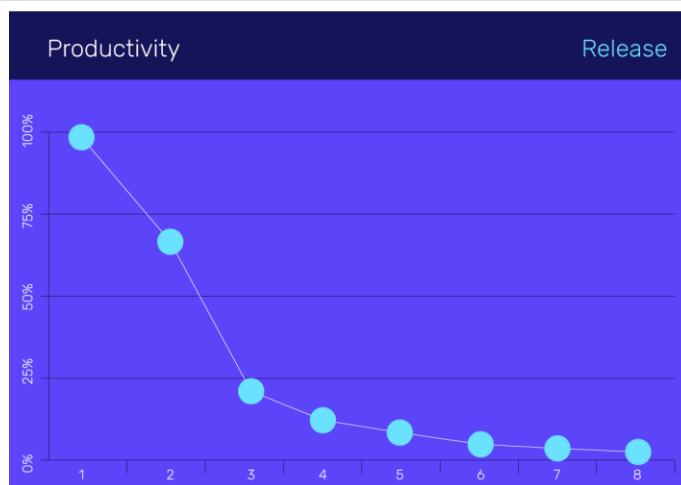
© JMA 2020. All rights reserved

Costo por línea de código a lo largo del tiempo.



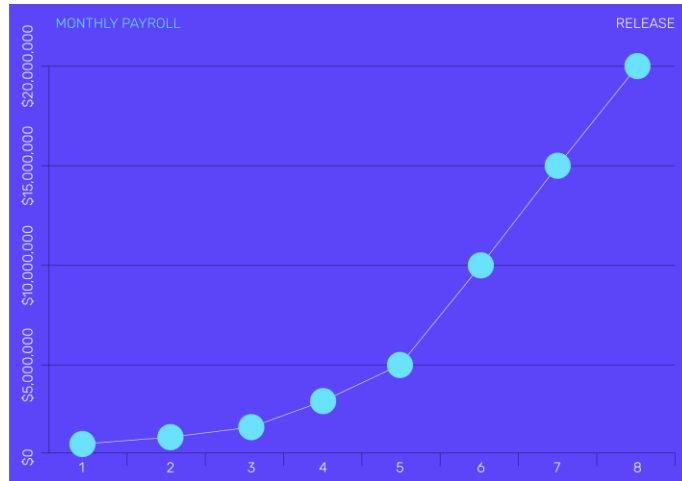
© JMA 2020. All rights reserved

Productividad del desarrollo por liberación.



© JMA 2020. All rights reserved

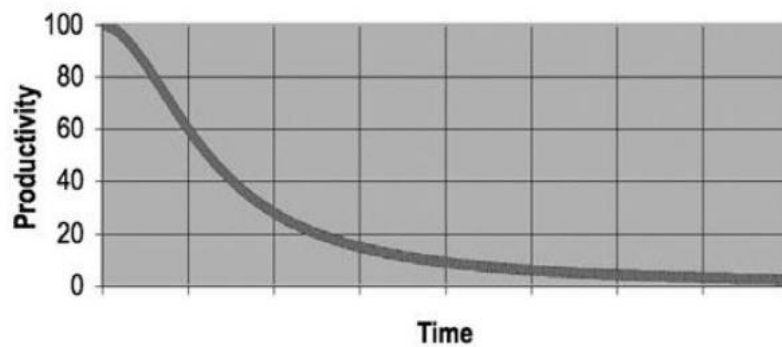
Costo de desarrollo mensual por liberación.



© JMA 2020. All rights reserved

Deuda técnica

Costo de poseer código “No Mantenable”



© JMA 2020. All rights reserved

Aseguramiento de la calidad

- Hay 5 principios fundamentales respecto a las metodologías de pruebas que deben quedar claros desde el primer momento:
 - Las pruebas exhaustivas no son viables
 - 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 2020. 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 2020. 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 2020. 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 2020. 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 2020. 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 ejecuten 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 2020. All rights reserved

Calidad de software

- La calidad es uno de los pilares fundamentales del desarrollo de sistemas de información.
- Las pruebas permiten valorar la calidad del desarrollo y corregir los defectos encontrados. Las pruebas, por si mismas, no aumentan la calidad del producto desarrollado, únicamente detectan errores en el producto.
- Para asegurar la calidad del producto final es necesario que todo el proceso de desarrollo se realice de la forma correcta y en el momento justo, utilizando las metodologías, técnicas y buenas prácticas adecuadas por personas cualificadas.

© JMA 2020. All rights reserved

Clean Code y Calidad de software

- Robert C. Martin propone seguir una serie de guías y buenas prácticas a la hora de escribir el código que se enmarcan en la creación de código de calidad.
- En concreto se centra en el más bajo de los niveles: la escritura del código.
- Trata aspectos básicos como el formato, estilo, nomenclaturas, convenciones, comentarios, ...
- Así mismo trata aspectos estructurales de como definir funciones y clases correctamente, qué uso hay que dar a los objetos y a las estructuras de datos, la relación entre excepciones y la lógica de negocio, cómo elegir y utilizar el código de terceros.
- Adicionalmente, trata la importancia de las pruebas unitarias y qué reglas deben cumplir, cómo construir sistemas basados en POJO's y sostenidos por tests, definición de reglas que nos ayudarán en nuestro diseño, observaciones al diseño de sistemas concurrentes.

© JMA 2020. All rights reserved

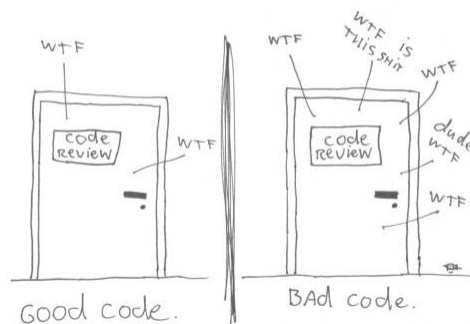
Code Smells

- ¿Cómo detectar un código mal oliente?
- Estos son los síntomas que podemos encontrar en el código fuente de un sistema que indican que muy probablemente existan problemas más profundos de calidad de código, de diseño o de ambos.
 - Rigidez es la tendencia que posee el software a ser difícil de cambiar, incluso en formas sencillas o cambios mínimos.
 - Fragilidad es la tendencia que posee un programa para romperse en muchos lugares cuando un simple cambio es realizado.
 - Inamovilidad es la dificultad de separar el sistema en componentes que pueden ser reutilizados en otros sistemas.
 - Viscosidad se presenta cuando hacer las cosas incorrectamente es más fácil que hacerlas del modo correcto.
 - Complejidad innecesaria es cuando hay muchos elementos que actualmente no son útiles.
 - Ambiente de desarrollo lento e ineficiente.
 - Tiempos muy largos de compilación
 - Subir el código toma horas
 - Hacer el despliegue toma varios minutos
 - Repetición innecesaria es cuando el código posee estructuras repetidas que pueden ser unificadas bajo una sola abstracción.
 - Opacidad es la tendencia que posee un módulo a ser difícil de leer y comprender.

© JMA 2020. All rights reserved

Métrica WTF

The ONLY valid MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



WTF: Acrónimo de What The Fuck (Vulgarismo)

(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

© JMA 2020. All rights reserved

WTF

```
static int STRTXG(bool b, int i0, int i1, int i2, int i3, int i4) {
    if (b == true)
        return 0;
    else {
        switch (i0) {
            case -1:
                return STXTGH(i2);
            case 920:
                return SFFMAB(i1, i3);
            case 1222:
                return SFFMAB(i1, i3);
            case 824:
                return SGXECB(i4);
            //TODO: Uncomment Later
            //case 3:
            //    return STXGMB(i1) + STXKSB(i1) + STXNRB(i1);
            default:
                return 42;
        }
    }
    return 0;
}
```

© JMA 2020. All rights reserved

WTF

```
static int STRTXG(bool b, int i0, int i1, int i2, int i3, int i4) {
    if (b == true)
        return 0;
    else {
        switch (i0) {
            case -1:
                return STXTGH(i2);
            case 920:
                return SFFMAB(i1, i3);
            case 1222:
                return SFFMAB(i1, i3);
            case 824:
                return SGXECB(i3);
            //TODO: Uncomment Later
            //case 3:
            //    return STXGMB(i1) + STXKSB(i1) + STXNRB(i1);
            default:
                return 42;
        }
    }
    return 0;
}
```

© JMA 2020. All rights reserved

WTF

- ¿Qué puerta representa tu código?
- ¿Qué puerta representa tu equipo o tu empresa?
- ¿Por qué estamos en esa habitación?
- ¿Es solo una revisión de código normal o hemos encontrado una serie de problemas horribles poco después del lanzamiento?
- ¿Estamos depurando en pánico, analizando el código que pensamos que funcionaba?
- ¿Los clientes huyen despavoridos y los gerentes nos respiran en el cogote?
- ¿Cómo asegurarnos de terminar detrás de la puerta correcta cuando las cosas se ponen difíciles?
- La respuesta es: la maestría.
- Hay dos factores para conseguir la maestría: conocimiento y trabajo. Debes obtener el conocimiento de los principios, patrones, prácticas y heurísticas que conoce un maestro; también debes machacar ese conocimiento, con tus dedos, ojos e instinto, trabajando duro y practicando.

© JMA 2020. All rights reserved

WTF



© JMA 2020. All rights reserved

Fundamentos

- “Escribir código que entienda la computadora es una técnica; escribir código que entienda un ser humano es un Arte” – Robert Martin
- El campo @author de un Javadoc nos dice quiénes somos. Somos autores. Y una cosa sobre los autores es que tienen lectores. De hecho, los autores son responsables de comunicarse bien con sus lectores. La próxima vez que escriba una línea de código, recuerde que es un autor, escribiendo para lectores que juzgarán su esfuerzo.
- La proporción de tiempo dedicado a leer frente a escribir es bastante superior a 10: 1. Estamos constantemente leyendo el código antiguo como parte del esfuerzo para escribir código nuevo. Debido a que esta relación es tan alta, queremos que la lectura del código sea fácil, incluso si dificulta la escritura. Por supuesto, no hay forma de escribir código sin leerlo, por lo que facilitar la lectura facilita la escritura.
- No hay escapatoria de esta lógica. No puede escribir código si no puede leer el código circundante. El código que intenta escribir hoy será difícil o fácil de escribir, dependiendo de lo difícil o fácil que sea leer el código circundante. Entonces, si quiere ir rápido, si quiere terminar rápidamente, si desea que su código sea fácil de escribir, hágalo fácil de leer.

© JMA 2020. All rights reserved

Fundamentos

- Cada sistema está construido a partir de un lenguaje específico de dominio diseñado por los programadores para describir dicho sistema. Las clases son los sujetos y los métodos los predicados.
- El arte de la programación es, y siempre ha sido, el arte del diseño del lenguaje.
- Los programadores experimentados piensan en los sistemas como historias para ser contadas en lugar de programas para ser escritos. Utilizan las prestaciones del lenguaje de programación elegido para crear un lenguaje mucho más rico y expresivo que pueda usarse para contar dicha historia.
- Parte de ese lenguaje específico de dominio es la jerarquía de funciones que describen todas las acciones que tienen lugar dentro de ese sistema. En un acto artístico de recurrencia, esas acciones se escriben para usar el lenguaje específico del dominio que definen para contar su propia pequeña parte de la historia.

© JMA 2020. All rights reserved

Fundamentos

- Con el tiempo, la automatización de las pruebas, y en especial las unitarias, se han convertido en una parte fundamental y formal del proceso de desarrollo de software.
- Antiguamente, los fragmentos de código se probaban de forma manual e informal. Hoy en día, se escribiría una prueba que se asegurara de que cada rincón y grieta de ese código funcionara como se esperaba. Se aislaría el código de las dependencias y las burlaría para tener un control absoluto sobre el.
- Una vez que pasaran un conjunto de pruebas, nos aseguraríamos de que esas pruebas fueran convenientes para cualquier otra persona que necesitara trabajar con el código. Nos aseguraríamos de que las pruebas y el código se registraran juntos en el mismo paquete fuente.
- Los movimientos Agile y TDD han animado a muchos programadores a escribir pruebas unitarias automatizadas.
- Muchas de las estrategias propuestas en clean code están destinadas a facilitar la creación de las pruebas, sin olvidar que su automatización requiere código que debe cumplir como el resto del código.

© JMA 2020. All rights reserved

La regla KISS

- El principio KISS (del inglés Keep It Simple, Stupid!) «¡Mantenlo sencillo, estúpido!» es un acrónimo usado como principio de diseño.
- El principio KISS establece que la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos; por ello, la simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad innecesaria debe ser evitada.
- Paradójicamente la filosofía KISS no es fácil de lograr porque estamos acostumbrados a creer que cualquier proceso está más trabajado cuanto más complicado parece. Sin embargo, la experiencia nos demuestra que la clave del éxito está en la sencillez y que todo funciona mejor si es simple. ¿Por qué?, porque todo lo que es simple es fácil de cambiar, de recordar, de adaptar y de mantener, y porque la complejidad está ligada al desorden y a las contradicciones lógicas, lo que nos genera incertidumbre y, casi siempre, desinterés y abandono.

© JMA 2020. All rights reserved

La regla del Boy Scout

- No es suficiente escribir bien el código. El código debe mantenerse limpio con el tiempo. Todos hemos visto que el código se pudre y degrada a medida que pasa el tiempo. Por lo tanto, debemos desempeñar un papel activo en la prevención de esta degradación.
- La regla del Boy Scout dice:
 - «deja el campamento más limpio de como te lo encontraste»
- Ampliándola a otros ámbitos sería algo así como:
 - «deja las cosas mejor de como te las encontraste»
- Sin olvidar:
 - «no hagas lo que no te gusta que te hagan»
- Muchas veces, revisando código, nos encontramos con que el nombre de una variable no es demasiado intuitivo o con un fragmento de código duplicado. Resolviendo este tipo de matices (en vez de mirar hacia otro lado y pasar de largo), estaremos aplicando la regla del Boy Scout.

© JMA 2020. All rights reserved

PRUEBAS UNITARIAS

© JMA 2020. All rights reserved

Introducción

- El código de prueba es tan importante como el código de producción. No es un ciudadano de segunda clase. Requiere reflexión, diseño y cuidado. Debe mantenerse tan limpio como el código de producción.
- Son las pruebas unitarias las que mantienen nuestro código de producción flexible, mantenible y reutilizable.
- Sin pruebas, cada cambio es un posible error. No importa cuán flexible sea la arquitectura o bien particionado el diseño, sin pruebas, será reacio al cambio debido al temor de introducir errores no detectados.
- Con el respaldo de las pruebas se podrá cambiar, refactorizar y experimentar para mejorar el código, la arquitectura y el diseño sin ningún miedo.

© JMA 2020. All rights reserved

Test Driven Development (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
 - No hay que crear nunca más de una prueba que falle
 - El código creado debe ser el mínimo para que la prueba pase
 - Eliminar el Código Duplicado empleando Refactorización

© JMA 2020. All rights reserved

Ritmo TDD

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

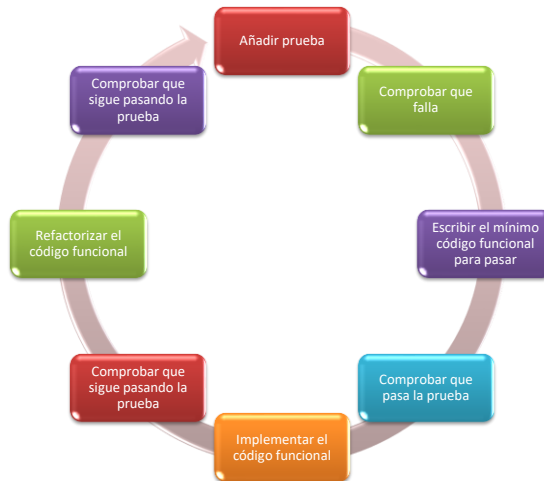
© JMA 2020. All rights reserved

Estrategia RED – GREEN

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

© JMA 2020. All rights reserved

Ritmo TDD



© JMA 2020. All rights reserved

Refactorizar el código en pruebas

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

© JMA 2020. All rights reserved

Beneficios de TDD

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

© JMA 2020. All rights reserved

Pruebas limpias

- Si no se mantienen limpias las pruebas, se perderán y con ellas lo que mantiene limpio el resto del código.
- Las pruebas requieren el mismo mantenimiento que el resto del código para ajustarlas a la evolución del sistema.
- La legibilidad en las pruebas es incluso mas importante que en el código de producción, deben decir mucho con el menor número de expresiones posibles.
- El código de la prueba debe seguir las mismas reglas de limpieza que el resto del código: nomenclatura, claridad, simplicidad y densidad de expresión.
- Adicionalmente, complementan la comprensión del código de producción con casos de ejemplo del uso de dicho código.

© JMA 2020. All rights reserved

Patrones

- Los casos de prueba se pueden estructurar siguiendo diferentes patrones:
 - ARRANGE-ACT-ASSERT: Preparar, Actuar, Afirmar
 - GIVEN-WHEN-THEN: Dado, Cuando, Entonces
 - BUILD-OPERATE-CHECK: Generar, Operar, Comprobar
- Aunque con diferencias conceptuales, todos dividen el proceso en tres fases:
 - Una fase inicial donde montar el escenario de pruebas que hace que el resultado sea predecible.
 - Una fase intermedia donde se realizan las acciones que son el objetivo de la prueba.
 - Una fase final donde se comparan los resultados con el escenario previsto.

© JMA 2020. All rights reserved

Preparación mínima

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

© JMA 2020. All rights reserved

Actuación mínima

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

© JMA 2020. All rights reserved

Evitar varias aserciones

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

© JMA 2020. All rights reserved

Lenguaje específico del dominio

- 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 más o menos de lo necesario para la prueba.
 - Menor posibilidad de compartir el estado entre las pruebas, lo que crea dependencias no deseadas entre ellas.
- Cada prueba normalmente tendrá requisitos diferentes para funcionar y ejecutarse. Los métodos de instalación y desmontaje son únicos, pero se pueden crear tantos métodos auxiliares como escenarios reutilizables se necesiten.

© JMA 2020. All rights reserved

Principios F.I.R.S.T.

- **Fast:** Los tests deben ser rápidos, del orden de milisegundos, hay cientos sino miles de tests en un proyecto que se deben ejecutar continuamente.
- **Independent:** Los tests no deben depender del entorno ni de ejecuciones de tests anteriores, se pueden ejecutar en cualquier orden.
- **Repeatable:** Los tests deben ser repetibles, para cualquier entorno, 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. El resultado debe ser booleano: pasa o falla.
- **Timely (Oportuno):** Los tests deben crearse en el momento oportuno, antes del código, y ejecutarse en el momento oportuno, después de cada cambio en el código.

© JMA 2020. All rights reserved

Aislar las pruebas

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

© JMA 2020. All rights reserved

Cubrir aspectos no evidentes

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

© JMA 2020. All rights reserved

Respetar los limites de las pruebas

- La pruebas unitarias ejercitan profundamente los componentes de formar aislada centrándose en la funcionalidad, los cálculos, las reglas de dominio y semánticas de los datos. Opcionalmente la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
- Las pruebas de integración se basan en componentes ya probados (unitaria o integración) o en dobles de pruebas y se centran en la estructura de llamadas, secuencias o colaboración, 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 tener un aspecto. Las pruebas funcionales del sistema son las pruebas de integración de todo el sistema centrándose en compleción de la funcionalidades y los procesos de negocio, su estructura, disponibilidad y accesibilidad.

© JMA 2020. All rights reserved

NOMBRES CON SENTIDO

© JMA 2020. All rights reserved

Introducción

- Los nombres están en todas partes en el software. Nombramos variables, funciones, argumentos, clases, métodos, propiedades, eventos, espacios de nombre y paquetes. Nombramos los archivos fuente y los directorios que los contienen, los ensamblados o paquetes, ...
- Debido a que lo hacemos continuamente, será mejor que lo hagamos bien. Lo que sigue son algunas reglas simples para crear buenos nombres.
- Todos los nombres deben ser intencionados, significativos y descriptivos.
 - Evitar:
 - `int d; // elapsed time in days`
 - A favor de:
 - `int elapsedTimeInDays;`
 - `int daysSinceCreation;`
- Cuando un nombre esta compuesto por mas de una palabra, dado que no se pueden insertar espacios, se recomienda el uso de mayúsculas y minúsculas para facilitar la legibilidad, evitando (salvo fuerza mayor) separadores como `_` más complicados de escribir y pueden ser operadores. Existen dos tipos de notaciones:
 - PascalCase (también conocida como UpperCamelCase), la inicial de cada una de las palabras en mayúscula y el resto en minúsculas. Ejemplo: `EjemploDePascalCase`.
 - CamelCase (o lowerCamelCase), igual que la anterior con la excepción de que la primera letra del nombre es minúscula. Ejemplo: `ejemploDeCamelCase`.

© JMA 2020. All rights reserved

¿Qué hace este código?

```
public List<int[]> getThem() {  
    List<int[]> list1 = new List<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

© JMA 2020. All rights reserved

¿Y este código?

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new List<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

© JMA 2020. All rights reserved

Evitar la desinformación

- Usar nombres confusos que inducen a falsos significados.
- No reutilizar nombres que ya tienen uso común para otra cosa: hp, aix, sco.
- No usar palabras con significado como parte del nombre (List, Collection, ...) si realmente no lo representan (no es una lista o una colección): UserList. En su lugar utilizar términos genéricos que no tengan un reflejo en una estructura: Group. O mejor, utilizar el singular para elementos individuales o el plural para múltiples elementos: User y Users.
- Evitar nombres muy similares que requieran una lectura cuidadosa: XYZControllerForHandlingOfStrings y XYZControllerForStorageOfStrings. Los entornos modernos autocompletan el código y pueden llevar a selecciones erróneas.
- Siguen estando raleziados por la Uivenrsdiad ignlsea de Cmdibrage, no ipmotra el odren en el que las ltears etsén ersciats, la úicna csoa ipormtnate es que la pmrيرة y la última ltera esétn ecsritas en la psióción cocrreta. El retso peuden etsar ttaolmnte doardsendo y aún pordás lerelo sin pobrleams, pquore no lemeos cada ltera en sí msima snio cdaa paalbra etenra.
- Evitar efectos visuales:
 int l = 10, o = 10;
 int resultado = l == o ? 1 : 0;

© JMA 2020. All rights reserved

Distinciones con sentido

- Si el compilador protesta porque el nombre ya esta siendo utilizado: a bien hay que reutilizar el elemento ya definido o el nombre esta mal escogido. Si los nombres tienen que ser distintos, también deben tener un significado diferente.
- Evitar palabras polisémicas y sinónimos.
- Evitar abreviaciones, prefijos y las palabras redundantes que no aportan significado:
 - Nombre, ElNombre, strNombre
 - Producto, ProductoInfo, ProductoData
- No utilizar nombres con series numéricas (p1, p2, p3), no desinforman porque ni siquiera informan:
 - puntoInicial, puntoFinal
 - puntoA, puntoB, puntoC.
- Hay que utilizar aquellos que se puedan pronunciar, recordar y sean ortográficamente correctos (klass): evitan errores de comunicación y transcripción.

© JMA 2020. All rights reserved

Que se puedan buscar

- Los nombres breves (una sola letra) solo se deberían utilizar en bucles cortos (como índices) y en métodos o funciones de muy pocas líneas (usando iniciales).
- Para escribir menos, los entornos modernos permiten refactorizar (cambiar nombre de todas las ocurrencias) un nombre corto a un nombre completo.
- La asignación de literales a constantes permite dar nombre a los valores, aportando semántica.
- Habitualmente se sigue el convenio de nombrar las constantes en mayúsculas utilizando el _ como separador.
 - `for(int i = 0; i < 5; i++) {`
 - `for(int i = 0; i < WORK_DAYS_PER_WEEK; i++) {`
- Aunque siempre que sea posible es recomendable usar valores dinámicos frente a constantes:
 - `for(int i = 0; i < days.count; i++) {`

© JMA 2020. All rights reserved

Codificación

- Como regla general ya tenemos suficientes codificaciones como para imponer nuevas con los nombres, por lo que se deben evitar. Requieren aprendizaje, traducción simultánea y dificultan la legibilidad y su pronunciación.
- La notación húngara, añadir el tipo de datos al nombre, tenía sentido en su momento (ausencia de tipos) pero actualmente dificultan la legibilidad y el cambio de tipos.
- Añadir prefijos o sufijos que indiquen el tipo de elemento (variable, parámetro, atributo) no tiene sentido en los entornos modernos que colorean el código. Se puede utilizar prefijos (btn, txt, ...) para agilizar la búsqueda en el autocompletar.
- Una excepción es su uso en la inyección de dependencias y la implementación de interfaces:
 - Interfaz: IShapeFactory, Clase: ShapeFactory
 - Interfaz: ShapeFactory, Clase: ShapeFactoryImp
- Es mejor usar el sufijo en la clase por que será usada muchas menos veces.

© JMA 2020. All rights reserved

Objetos

- Las clases e instancias son objetos por lo que sus nombres deben ser sustantivos.
 - Usar nombres técnicos cuando el término sea técnico (Factory, Observable, Facade, ...) y nombres de dominio para conceptos de dominio.
- Los atributos y propiedades son características por lo que sus nombres deben ser sustantivos, adjetivos o verbos sustantivados.
- Los métodos y funciones son acciones por lo que sus nombres deben ser verbos o predicados.
 - Utilizar los prefijos get y set para métodos de acceso e is para booleanos.
 - Utilizar siempre el mismo verbo y grupo para la misma acción:
 - read/write o load/save.
 - No utilizar el mismo verbo para acciones diferentes:
 - add ¿sum o append?
 - Usar métodos estáticos con el nombre del tipo de argumento en lugar de sobrecargar constructores:
 - Complex.FromRealNumber(23.0) mejor que new Complex(23.0)

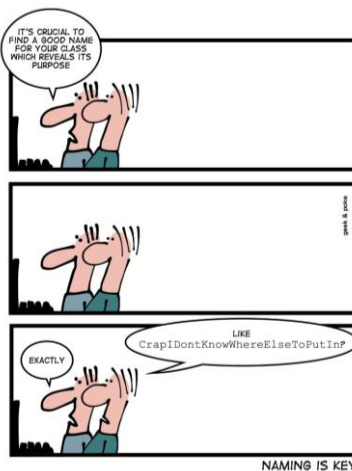
© JMA 2020. All rights reserved

Buenos nombres

- No usar juegos de palabras, jergas, chistes o sutilezas que puede que no comparta el lector. Hay que primar la claridad sin ambigüedades antes que el entretenimiento.
- Hay que elegir una palabra por concepto y mantenerla invariante siempre que se utilice el concepto.
 - get, fetch y retrieve son sinónimos, se debe seleccionar una.
- Sin juegos de palabras (polisemia), si se utiliza add para sumar o concatenar, no se debe utilizar para añadir a una colección, se podría utilizar append.
- Evitar asignaciones mentales, que el lector no tenga que traducir lo que lee a lo que conoce.
- Añadir contexto a las variables, por ejemplo, agrupándolas en clases o utilizando prefijos o sufijos cuando sea necesario.
- Los nombres cuanto más cortos mejor, siempre que sean explícitos y claros.

© JMA 2020. All rights reserved

El nombre es la clave



© JMA 2020. All rights reserved

FUNCIONES

© JMA 2020. All rights reserved

Responsabilidad única

- Principio:
LAS FUNCIONES SOLO DEBEN HACER UNA COSA. DEBEN HACERLO BIEN Y DEBE SER LO ÚNICO QUE HAGAN.
 - El problema de esta afirmación es como realizar operaciones complejas en una función.
 - La respuesta se encuentra en los niveles de abstracción: un problema complejo se descompone en partes menos complejas, que a su vez se descomponen en partes mas simples, y así sucesivamente hasta llegar a partes que solo hagan una cosa.
 - Cada descomposición genera un nivel de abstracción. Los niveles de abstracción pueden variar para cada una de las partes.
 - Si una función solo realiza los pasos situados a un nivel por debajo de la función, entonces hace una cosa. La mezcla de niveles de abstracción en una función siempre resulta confusa y complica las cosas.
 - La idea se basa en que es preferible tener muchas funciones sencillas frente a unas pocas complejas.
-

© JMA 2020. All rights reserved

Tamaño de las funciones

- La primera regla de las funciones es que deben ser pequeñas. La segunda regla de las funciones es que deberían ser todavía más pequeñas.
- Idealmente unas de 4 ó 5 líneas, no mas de 20 líneas (salvo excepciones): lo que se pueda leer con comodidad en una pantalla normal, sin tener que hacer scroll que provoca desplazamientos adelante atrás y viceversa.
- Sin trampas, líneas cortas, no mas de 80 caracteres aproximadamente (en los entornos modernos depende de las barras de herramientas).

© JMA 2020. All rights reserved

Tamaño reducido

- Los bloques de los bucles y condicionales deberían tener una sola línea: una invocación a la función que lo resuelve. No solo reduce el tamaño, añade valor documental si el nombre está bien elegido. De esta forma se reduce también el nivel de anidamientos a uno.
- Es complicado reducir un switch por su naturaleza múltiple, se consigue implementando cada caso en su correspondiente función. Como alternativa, si es posible, escondiéndolas en clases abstractas (patrón Factory+strategy) o utilizando el polimorfismo evita la necesidad de condicionales múltiples.
- Si la función requiere secciones (declaraciones, inicializaciones, operaciones ...) es que no cumple el principio de responsabilidad única o es demasiado larga.

© JMA 2020. All rights reserved

Condiciones

- Las expresiones condicionales utilizadas en las instrucciones if y en los bucles pueden tener una considerable complejidad.
- Envolver las expresiones condicionales complejas en funciones booleanas permite ganar legibilidad aportando semántica y evita que el lector tenga que evaluar la expresión para deducir su significado.
- Extraer la condición a una función simplifica la creación de pruebas unitarias que ejerciten profundamente la condición utilizando técnicas de Análisis de Valores Límite.

© JMA 2020. All rights reserved

Nomenclatura

- Hay que respetar las reglas de nomenclatura: cuanto mas concreta sea la función mas fácil y corto será su nombre, pero es preferible un nombre largo autodescriptivo que un nombre corto críptico.
- El principio de Least Surprise (Ward): las funciones, métodos o clases deben hacer lo que (razonablemente) se espera de ellas. En base a su nombre, el comportamiento debe ser obvio para cualquiera sin tener que sumergirse en su código: `lista.deleteItem(item)`.
- Sin efectos secundarios, no debe hacer ni mas ni menos de lo que promete en su nombre, sin hacer cosas adicionales ocultas (que no sean obvias) como realizar modificaciones en una consulta. Estos efectos tienen consecuencias inesperadas difíciles de controlar.

© JMA 2020. All rights reserved

Regla descendente

- El objetivo es leer el código como un texto, de arriba a bajo:

```
function void HacerAlgo() {  
    HacerEsto();  
    HacerAquello();  
    HacerLoOtro();  
}
```
- Se lee (si un párrafo es muy largo, algo huele mal):
 - Para Hacer Algo hay que Hacer Esto, Hacer Aquello y Hacer Lo Otro.
 - Para Hacer Esto hay que ...
 - Para Hacer Aquello hay que ...

© JMA 2020. All rights reserved

Parámetros y argumentos

- El número ideal es cero, después uno (mónadico) y, por último, dos (diádico). Se debe evitar las funciones de tres parámetros (triádico) y las de más de tres parámetros (poliádico) deben tener una justificación especial.
- El número de parámetros afectan a la complejidad conceptual y, especialmente, a las pruebas: dispara la combinatoria de los valores de los argumentos a probar.
- Las dos formas monádicas habituales son para: preguntar sobre un valor (`boolean fileExists("MyFile");`) o transformar un valor (`InputStream fileOpen("MyFile");` de cadena a stream). Un caso menos habitual es el evento, usa el argumento para modificar el estado del sistema, por lo general no devuelven nada. Si es necesario devolver un valor, se debe usar el valor retorno y no utilizar un parámetro de salida, que resulta confuso. Devolver dos valores, valor de retorno y parámetro de salida, denota exceso de complejidad, huele mal.

© JMA 2020. All rights reserved

Parámetros y argumentos

- Las funciones diádicas requieren el doble de atención por tener el doble de parámetros, se debe tender a que los argumentos sigan y tengan un orden natural (estén relacionados), cuando estos carecen de dicho orden se deben convertir en monódicas: pasar uno de los argumentos a atributo de la clase, crear una clase (un argumento en el constructor, un método con el otro), ...
- Cuando una función requiere dos o más argumentos se suelen encapsular en un objeto, esto no es una trampa, pues suele suceder que los valores antes pasados como argumentos individuales pertenezcan a un concepto que requiere nombre propio.
- En todos los casos, usar un parámetro selector apesta, indica que la función hace mas de una cosa.

© JMA 2020. All rights reserved

Parámetros y argumentos

- Un número variable de argumentos es en realidad un único parámetro (de tipo array, colección, ...)
 - Avg(1, 2, 4, 8, 16) → public double Avg(double... args)
- Los nombres de los parámetros documentan y deben estar acordes al contexto, dado que complementan al nombre de la función.
 - Monádica: deben formar un par: verbo (función) y sustantivo (argumento).
 - writeField(name)
 - Diádicas: codificar los nombres de los parámetros en el nombre de la función:
 - assertExpectedEqualsActual(expected, actual)

© JMA 2020. All rights reserved

Consultas vs Comandos

- Una función que devuelve información del objeto es una consulta.
- Una función que cambia el estado del objeto es un comando.
- Una función que cambia el estado del objeto y devuelve información del objeto es confusa, rompe el principio de responsabilidad única.
- Un comando podría devolver el resultado de la ejecución de la acción aunque no suele ser necesario porque se puede comprobar en el nuevo estado del objeto. No debe devolver códigos de error, sería una consulta.

© JMA 2020. All rights reserved

Excepciones

- Devolver códigos de error presentan múltiples problemas:
 - Incumple la separación entre consultas y comandos.
 - El error se debe procesar de forma inmediata, complicando el código mezclando la invocación con el tratamiento del error.
 - El tratamiento del error es opcional, se puede ignorar.
 - Los códigos de error, por ejemplo si son una enumeración, hay que mantenerlos y crean dependencias que requieren recompilaciones masivas.
- Los errores se deben notificar mediante excepciones:
 - El tratamiento de la excepción es obligatorio.
 - Simplifica el código: separa las invocaciones del tratamiento, incluso a diferentes niveles.
 - Las nuevas excepciones son derivaciones de la clase excepción apropiada y no requieren recompilación.

© JMA 2020. All rights reserved

Tratamiento de la excepción

- Los bloques try/catch son densos por naturaleza. Mezclan el flujo del proceso con el tratamiento de las excepciones, vulnerando el principio de responsabilidad única y, en caso de múltiples excepciones, alargando excesivamente las función.
- Por ello conviene extraer el cuerpo de los bloques en funciones individuales.
- La función del bloque try simplifica la creación de pruebas que deben generar excepciones.
- Extraer una función por cada bloque catch que solo debe tratar el error. Adicionalmente, esto permite reutilizar los tratamientos de error.
- La instrucción try debe ser la única de la función.

© JMA 2020. All rights reserved

DRY

- Las duplicidades son una fuente inagotable de problemas.
- El principio DRY (Don't Repeat Yourself) viene a recordarnos este hecho: No te repitas.
- El código duplicado hace que nuestro código sea más difícil de mantener y comprender además de generar posibles inconsistencias. Hay que evitar el código duplicado siempre. Para ello, dependiendo del caso concreto, la refactorización, la abstracción o el uso de patrones de diseño pueden ser nuestros mejores aliados.

© JMA 2020. All rights reserved

Dijkstra

- Una de las reglas de programación estructurada es la de Dijkstra: cada función, y cada bloque dentro de una función, debe tener una entrada y una salida. Seguir estas reglas significa que solo debe haber un return por función, no se deben usar break o continue en los bucles y, bajo ningún concepto, el goto.
- Si bien puede tener sentido en funciones o bucles muy largos, en funciones cortas el uso adecuado de varios return y del continue evita un anidamiento excesivo, el correcto uso del break puede simplificar considerablemente la condición del bucle (y el riesgo que conllevan las condiciones complejas).

© JMA 2020. All rights reserved

Refinamiento sucesivo

- La escritura de software es como cualquier otro tipo de escritura creativa. Nadie suele ser capaz de escribir la versión definitiva a la primera: es habitual que el primer borrador sea desorganizado y torpe, por lo que se debe retocar y reestructurar hasta se lea adecuadamente.
- En el primer esbozo las funciones salen largas y complicadas: tienen muchas sangrías, bucles anidados, largas listas de argumentos, los nombres son arbitrarios, hay código duplicado ...
- Mediante un proceso de refinamiento sucesivo, se va mejorando el código dividiendo en funciones, eliminando duplicaciones, cambiando nombres, reordenando ...
- Si se dispone del conjunto de pruebas unitarias, este proceso de mejora, también conocido como refactorización, se puede realizar con garantías.
- Con la experiencia, el primer esbozo ira siendo cada vez mas fino, pero no evita el refinamiento, lo aligera.

© JMA 2020. All rights reserved

COMENTARIOS

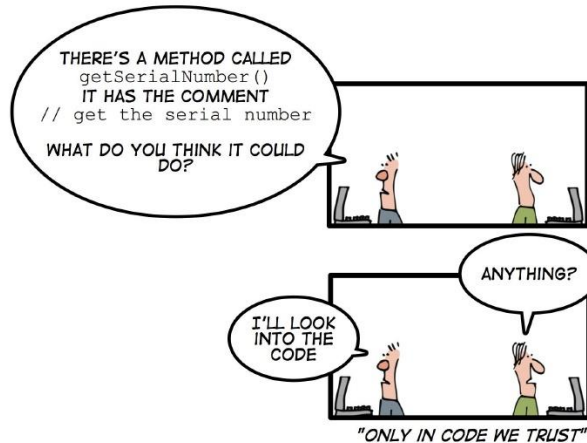
© JMA 2020. All rights reserved

Comentar solamente lo necesario

- Este principio afirma que los comentarios pueden hacerse; sin embargo, estos deben ser necesarios.
- Citando a Robert C. Martin:
 - "Un comentario es un síntoma de no haber conseguido escribir un código claro"*
 - No se debería tener la necesidad de escribir un comentario a excepción de unos pocos casos, si se sigue y aplica lo visto anteriormente: poner nombres con sentido a las entidades del software y a sus variables, a escribir funciones cortas que hagan una sola cosa y que la secuencia de llamadas en las funciones vayan contando la historia, contando qué hace el software.
 - Siempre que te veas en la necesidad de escribir un comentario, intenta reescribir el código o nombrar las cosas de otra forma, de tal forma que el comentario se vuelva irrelevante.
 - "Los comentarios mienten"*
 - Lo que ocurre es que, mientras los códigos son modificados, los comentarios no. Estos son olvidados, y por lo tanto, no retratan la funcionalidad real de los códigos.
 - Entonces, ya sabes; si vas a comentar el código, que sea solamente lo necesario y que sea revisado en conjunto con la versión del código que lo acompaña.
 - "El único comentario bueno es aquel que encuentras la manera de no escribirlo."*

© JMA 2020. All rights reserved

Solo en el código está la verdad



<http://geekandpoke.typepad.com>

© JMA 2020. All rights reserved

Tomarnos tiempo

- Nos debemos tomar tiempo para escribir un comentario.
- Se refiere a que cuando se escribe un comentario, al igual que pasa con el código, se debe hacer conscientemente de lo que queremos plasmar y hacerlo de una forma que aporte valor (comentario de calidad).
- Muchas veces no nos tomamos el tiempo necesario y nuestros comentarios terminan convirtiéndose en desinformación que hace menos legible el código; un claro ejemplo es dejar código comentado, este código es obsoleto y los programadores que lo vean muy rara vez lo borran creyendo que hay algo importante en ellos, de esta forma perduran en el tiempo.

© JMA 2020. All rights reserved

Aceptables

- **Comentarios legales:** cabe aclarar que en lo posible se debe hacer referencia a una licencia estándar o a un documento externo y no poner todos los términos y condiciones en el comentario.
- **Comentarios informativos:** solo cuando sea necesario, en los otros casos tratar de utilizar nombres con sentido.

```
// format matched hh:mm:ss dd, yyyy  
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\d*, \\d*");
```
- **Explicar la intención:** pueden ayudar a entender porque un programador tomó una determinada decisión al escribir una línea o bloque de código (decisión no evidente).

```
public int compareTo(Object o) {  
    // ...  
    return 1; // Forzar a que sean mayores  
}
```

© JMA 2020. All rights reserved

Aceptables

- **Clarificación:** estos comentarios suelen utilizarse cuando se implementa alguna API estándar que no puede ser modificada, aunque se puede envolver en una función pasarela con el nombre adecuado.

```
assertTrue(a.compareTo(a) == 0); // a == a  
assertTrue(a.compareTo(b) != 0); // a != b  
assertTrue(a.compareTo(b) == -1); // a < b  
assertTrue(b.compareTo(a) == 1); // b > a
```
- **Advertir las consecuencias:** en ocasiones determinado código debe escribirse estrictamente de una manera o hacer uso de algún componente, en estos casos es útil advertir la consecuencia si se modifica o cambia el código.

```
// Ojo: Modificado por el DMA de la UART, no optimizar.  
public volatile int UART_Flag = 0;
```

© JMA 2020. All rights reserved

Aceptables

- **Comentario TODO:** estos comentarios hacen referencia a una tarea que el programador piensa que se debe hacer pero que no realizó: una implementación por hacer, una refactorización o vigilar un cambio normativo pendiente de aprobación. Se debe eliminar una vez realizado.
- **Amplificación:** estos comentarios buscan amplificar la importancia de un determinado fragmento de código con el fin de que no pase como irrelevante.
`// IMPORTANTE: Recortar espacios para ordenar correctamente receiverLog.add(message.trim());`
- **Javadoc en API públicas:** sólo cuando el API es pública se debe describir, pero se debe hacer con calidad: “los javadoc pueden ser tan ambiguos, amplios y descorteses como cualquier otro tipo de documento”.

© JMA 2020. All rights reserved

Perniciosos

- **Mascullar:** Hablar entre dientes o escribir comentarios “porque toca” en los que no se entiende nada o crean mas dudas de las que aclaran.
- **Comentarios redundantes:** Comentarios que no hacen más que narrar en lenguaje natural lo que hace el código, sin aportar nada y consumiendo atención del lector, espacio y tiempo al desarrollador. Estos comentarios se conocen también como ruido blanco.
- **Comentarios confusos:** A veces hay comentarios tan mal redactados que cuesta mas entender el comentario que el código
- **Comentarios falsos:** Debido a una evolución posterior del código, nos encontramos con comentarios que ya no reflejan el comportamiento real del código, son falsos (independientemente de la intención). Leer solo el comentario puede llevar a falsas expectativas que acabaran en una sesión de depuración por que no se cumplen las expectativas.

© JMA 2020. All rights reserved

Perniciosos

- **Comentarios de control de versiones:** Estos comentarios, desde que existen los sistemas de control de versiones, solo aportan ruido y desinformación. Son un legado de tiempos oscuros en los que los programadores se veían obligados a registrar sus acciones en forma de comentarios en el código.
// 06/01/2005 : Corregido bug calculo de regalos [JMA]
// 25/12/2004 : Cambiado calculo de regalos [PEC]
// 02/02/2005 : Agregado calculo de regalos [JMA]
// ...
- **Código comentado:** Este es un tipo de comentario especialmente dañino. Si en algún momento ese código ha dejado de usarse, hay que borrarlo. No hay peligro, si luego se necesita, se recupera del sistema de control de versiones.
- **Autoría y menciones:** La manía de firmar el código cuando el programador no sabe que el control de versiones ya lo sabe.
// Added by Rick

© JMA 2020. All rights reserved

Perniciosos

- **Comentarios ruidosos:** No aportan nada, solo ruido, o simplemente sobran:
// Constructor predeterminado
public MyClass() {

} catch (Exception e) {
// Esto falla seguro!!!
- **Marcadores de posición:** A algunos programadores les gusta marcar visualmente un determinado punto en el código, los entornos modernos permiten colapsar y organizar el código sin chapuzas manuales.
// Métodos privados //////////////////////////////////////
- **Comentarios de cierre de llave:** Hay programadores que escriben un comentario en el cierre de llaves para indicar a que bloque pertenecen, esto se hace necesario cuando el código es demasiado largo:
} // while
} // if
} // main
- **Comentarios con HTML:** No es un formato que en los comentarios ayude al desarrollador, más bien hace el comentario más difícil de interpretar. Si una herramienta externa desea formatear el comentario, que lo haga sin tener que ensuciar el código fuente.

© JMA 2020. All rights reserved

Perniciosos

- **Información no local:** Añadir información de un sistema externo (como un puerto de conexión), no solo no aporta nada, si no que tampoco tenemos ningún tipo de control sobre esa información.
- **Demasiada información:** ¿Para que tanto texto? ¿y con que sentido? No pongas discusiones históricas ni datos irrelevantes en tus comentarios, puede parecer útil, pero solo te aporta algo a ti.

```
/*  
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)  
Part One: Format of Internet Message Bodies  
section 6.8. Base64 Content-Transfer-Encoding  
The encoding process represents 24-bit groups of input bits as output  
strings of 4 encoded characters. Proceeding from left to right, a  
24-bit input group is formed by concatenating 3 8-bit input groups.  
These 24 bits are then treated as 4 concatenated 6-bit groups, each  
of which is translated into a single digit in the base64 alphabet.
```

© JMA 2020. All rights reserved

Perniciosos

- **Conexiones no obvias:** La conexión entre un código y su comentario debe ser obvio, sin hacer alusiones a otros temas o detalles técnicos no relevantes. En el siguiente comentario: ¿Que es un filter byte? ¿Tiene relación con el +1 o con el *3? ¿O con ambos? ¿Un pixel es un byte? ¿Por qué 200?
/*
* start with an array that is big enough to hold all the pixels
* (plus filter bytes), and an extra 200 bytes for header info
*/
this.pngBytes = new byte[(((this.width + 1) * this.height * 3) + 200];
- **Cabeceras de funciones:** Las funciones cortas no necesitan descripción y un nombre bien elegido para un función pequeña que hace una sola cosa suele ser mejor que un comentario de cabecera.
- **Javadocs en código no público:** Los javadocs aportan valor en APIs públicas, fuera de ahí no tienen mucho sentido.

© JMA 2020. All rights reserved

FORMATO

© JMA 2020. All rights reserved

Introducción

- El formato hace referencia al continente mas que al contenido, por bien redactado que esté el contenido se necesitan reglas para escribir el código que faciliten la lectura. El formateo del código trata de la comunicación, y la comunicación debe ser el pilar de un desarrollador profesional, antes incluso de que el código funcione.
 - El estilo de codificación y la legibilidad establecen precedentes que continúan afectando a la capacidad de mantenimiento y la extensibilidad mucho después de que el código original haya cambiado totalmente. Tu estilo y disciplina sobreviven, aunque tu código no lo haga.
 - Los entornos de desarrollo modernos automatizan gran parte del formateo, por lo que es imperdonable encontrar código mal formateado.
-

© JMA 2020. All rights reserved

Formateo vertical

- El formato vertical se refiere al tamaño (en número de líneas) que debe tener un archivo fuente, así como su organización.
- Un archivo pequeño es más fácil de entender que uno muy extenso, pero muchos archivos pequeños son más difíciles de manejar que unos pocos mayores.
- Hay lenguajes, como el Java, que imponen una correspondencia entre el código y los ficheros, para el resto es importante llegar a un equilibrio entre el número de ficheros y la longitud de los mismos, siendo importantes los nombres de los mismos para facilitar la localización del código.

© JMA 2020. All rights reserved

Metáfora del periódico

- Piensa en un artículo de un periódico bien escrito (o un blog, en los tiempos que corren).
- En la parte superior esperas una cabecera que cuenta de que va el artículo y te permite decidir si quieres leerlo o no. Los primeros párrafos te dan una sinopsis de la historia, ocultando los detalles mientras te da los conceptos generales. Según continuas bajando se incrementan los detalles hasta que tienes todos los datos.
- Así es como debería ser un fichero fuente. El nombre debe ser simple pero explicativo, lo suficiente para decirnos si estamos en el módulo adecuado o no. Las primeras líneas muestran los conceptos y algoritmos de alto nivel. Y el detalle se incrementa según bajamos.

© JMA 2020. All rights reserved

Separación vertical entre conceptos

- Casi todo el código se lee de izquierda a derecha y de arriba a abajo. Cada línea representa una expresión y cada grupo de líneas representa un concepto completo. Esos conceptos deben separarse unos de otros con líneas en blanco. Si los conceptos se agrupan verticalmente, la densidad vertical implica una asociación cerrada.
- Ésta regla tan simple tiene un profundo efecto en la visualización del código. Cada línea en blanco es un señal visual que identifica y separa un nuevo concepto.

```
package fitnessse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = ""'+?''";
    private static final int MAX = 14;

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        // ...
    }

    public String render() throws Exception {
        // ...
    }
}
```

© JMA 2020. All rights reserved

Distancia y orden vertical

- Los conceptos relacionados no deberían estar demasiado lejos unos de otros en el mismo archivo. Y por supuesto, no deben estar en archivos diferentes. Cuanto más relacionados están, más cerca tienen que colocarse.
 - Declaración de variables: Deben declararse todas juntas en un mismo bloque al inicio de la función, excepto las variables de control de un bucle deberían declararse dentro de este. Si hay que declarar variables en un bloque anidado seguramente hay que descomponerlo en una nueva función.
 - Declaración de atributos: Deben declararse al principio de la clase.
 - Funciones dependientes: Si una función invoca a otra, deberían estar cerca y la invocada por debajo (aunque en algunos lenguajes esto no es posible). Esto hace que el flujo del programa sea natural y por tanto fácil de leer.
 - Afinidad: Además de por uso e invocación, hay que agrupar las funciones conceptualmente relacionadas.
- Si hay muchos atributos, y algunos se usan solo en unos métodos, o hay tantos métodos que la separación vertical es muy grande, seguramente se debería dividir la clase.
- Al igual que en un artículo, esperamos que los conceptos importantes estén primero y según bajamos, se encuentren los detalles de bajo nivel. Este orden vertical nos permite conocer la funcionalidad principal de un archivo echando un vistazo a las primeras líneas.

© JMA 2020. All rights reserved

Formateo horizontal

- El formateo horizontal hace referencia a las líneas individuales. Las líneas no deberían exceder los 80-100 caracteres aproximadamente (en los entornos modernos las barras de herramientas reducen el espacio de edición).
- Para la separación horizontal y densidad utilizaremos espacios en blanco horizontales para asociar cosas que están fuertemente relacionadas y disociar cosas que están más débilmente relacionadas. Rodeando los operadores de asignación con espacios en blanco los acentuamos. El paréntesis de apertura sin espacio previo asocia la condición a la instrucción o los argumentos a la función.

```
if(a < b && b < c)
    a = suma(a, b) * suma(a, c);
```

© JMA 2020. All rights reserved

Formateo horizontal

- Usar algún tipo de alineamiento horizontal para acentuar las estructuras hace que veamos la información columnada, lo que altera el orden de lectura, por lo que no es recomendable.

```
private    Socket          socket;
private    InputStream      input;
private    OutputStream     output;
```

- El sangrado (indentación) establece una jerarquía visual que facilita la identificación de bloques y sub bloques. No se debería romper aunque el bloque solo tenga una sola línea.

© JMA 2020. All rights reserved

Reglas de equipo

- Para gustos hay colores. Un equipo tiene que acordar que reglas de formateo se van a seguir. Si cada uno sigue las suyas va a ser fuente de problemas: añadiendo una carga cognitiva innecesaria al trabajar con el código de otro, o perdiendo tiempo en modificar el formateo.
- Los entornos de desarrollo modernos disponen de reglas predefinidas que aplican automática o manualmente, para evitar conflictos se pueden utilizar las establecidas por defecto. En caso de variarlas, todos los entornos deberían tener la misma configuración.

© JMA 2020. All rights reserved

PROCESAR ERRORES

© JMA 2020. All rights reserved

Introducción

- El manejo de errores es una practica fundamental de la codificación, pero no debe ser dominante: que entierre el código del proceso bajo el tratamiento de errores. El manejo de errores es importante pero no debe oscurecer la lógica del proceso.
- Como ya vimos anteriormente, es recomendable el uso de excepciones frente a los códigos de error: su tratamiento es opcional pero se deben procesar de forma inmediata, lo que confunde el flujo de ejecución y la aparición de nuevos códigos desencadenan modificaciones en cascada.
- En lenguajes como el Java las excepciones son declarativas, lo que puede conllevar que la aparición de nuevas causas de excepción también desencadenen una cascada de modificaciones en la pila de llamadas.

© JMA 2020. All rights reserved

try-catch-finally

- Las excepciones definen un ámbito en el programa, el bloque try de la instrucción try-catch delimita dicho ámbito (similar a las transacciones). Los bloques catch separan la lógica del tratamiento de errores. Es conveniente empezar con la instrucción try implementando los bloque catch, de tal forma que se defina que esperar independientemente de que se produzca una excepción.
- La excepción debe incluir un contexto para determinar la causa, origen y ubicación del error. En caso de propagar excepciones no tratadas es conveniente relanzar la excepción original o incluirla en la nueva excepción para no perder la trazabilidad.

```
    } catch (Exception e) {  
        // ...  
        throw e;  
    } catch (Exception e) {  
        throw new StorageException("retrieval error", e);
```

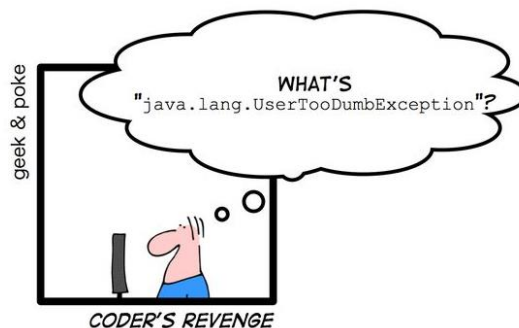
© JMA 2020. All rights reserved

Excepciones propias

- El tratamiento de excepciones requiere la creación de excepciones propias.
- Existen varias formas de clasificar los errores: por origen (componente, módulo, API, ...), por tipo (dispositivos, red, código, ...), ...
- La recomendación es definir las excepciones en función a como se capturan en los bloques catch para dar el tratamiento. La mayoría de los sistemas disponen de una jerarquía de clases para derivar las nuevas excepciones y utilizan el sufijo `Exception` en su nombre.
- Al usar APIs de terceros siempre es conveniente envolver sus excepciones en excepciones propias (patrón facade) y definir un controlador por encima para procesar las cancelaciones.

© JMA 2020. All rights reserved

Excepciones propias



© JMA 2020. All rights reserved

Null

- Null representa la ausencia de valor o valor vacío. Los nulos son una fuente inagotable de errores y excepciones (billion dollar mistake).
- Si una función devuelve un valor, debe devolverlo o, en caso de no ser posible, generar una excepción. Nunca debería devolver null, es lo mismo que no devolver valor pero obligando a comprobar el valor o incurrir en una NullPointerException sin contexto.
- Siguiendo la misma línea, un argumento nunca debería ser nulo: si el parámetro es opcional debería ser tratado como tal (sobrecarga, valores por defecto, ...). Para no abusar de la NullPointerException, en caso de recibir un null no esperado, se debería generar una excepción específica (como IllegalArgumentException) o utilizar aserciones.

© JMA 2020. All rights reserved

OBJETOS vs ESTRUCTURAS DE DATOS

© JMA 2020. All rights reserved

Asimetría entre clases y estructuras

- Las estructuras muestran sus datos y carecen de funcionalidad significativa. Los objetos ocultan sus datos tras atracciones y exponen métodos para operar con los mismos.
- Punto concreto:

```
public class Point {  
    public double x;  
    public double y;  
}
```
- Punto abstracto:

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

© JMA 2020. All rights reserved

Procedural

```
public class Square {  
    public Point topLeft;  
    public double side;  
}  
  
public class Circle {  
    public Point center;  
    public double radius;  
}  
  
public class Geometry {  
    public final double PI = 3.141592653589793;  
    public double area(Object shape) throws NoSuchShapeException {  
        if (shape instanceof Square) {  
            return Math.pow(((Square)shape).side);  
        }  
        if (shape instanceof Circle) {
```

© JMA 2020. All rights reserved

Orientado a objetos

```
public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side*side;
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

© JMA 2020. All rights reserved

Procedural vs Orientado a objetos

Procedural

- El código procedural (código que usa estructuras de datos) facilita la adición de nuevas funciones sin cambiar las estructuras de datos existentes.
- El código procedural dificulta agregar nuevas estructuras de datos porque todas las funciones deben cambiar.

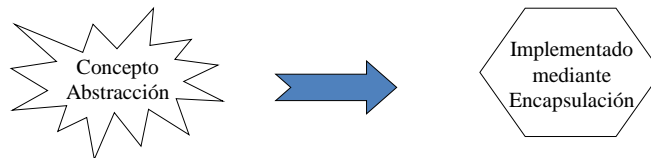
Orientado a objetos

- El código orientado a objetos facilita agregar nuevas clases sin cambiar las funciones existentes (polimorfismo).
- El código orientado a objetos dificulta agregar nuevas funcionalidades porque hay que cambiar todas las clases derivadas o que implementen el interfaz.

© JMA 2020. All rights reserved

Abstracción

- Consiste en captar las características esenciales de un objeto, así como su comportamiento.
- Una buena abstracción se centra en la vista externa del objeto, eliminando todos los detalles pocos importantes.
- Una clase bien diseñada expone un número mínimo de métodos cuidadosamente elegidos para proporcionar el comportamiento esencial de la clase.
- La Abstracción es un concepto teórico, y los lenguajes Orientados a Objeto utilizan la encapsulación para aplicar este concepto.



© JMA 2020. All rights reserved

Encapsulación

- Es la propiedad que permite asegurar que el contenido de la información de su objeto está oculta al mundo exterior.
- La encapsulación también es conocido como OCULTACION DE INFORMACION.
- El encapsulamiento proporciona la abstracción y se implementa mediante los modificadores de acceso a los miembros de las clases:
 - Privado: solo accesible desde la propia clase.
 - Protegido: accesible desde la propia clase y sus herederos.
 - Publico: sin restricción de acceso.

© JMA 2020. All rights reserved

Encapsulación

- Los atributos deben ser siempre privados:
 - Ocultan los detalles concretos de la implementación.
 - Evitan el acoplamiento, la clase controla el estado de su memoria sin la indeterminación de accesos externos.
 - Facilitan el versionado, se puede cambiar completamente la estructura de memoria sin efecto secundarios en terceros.
- La funcionalidad de alto nivel se expone como métodos públicos. Los métodos de menor nivel serán privados o protegidos.
- Dado que los atributos deberían ser privados, las clases base deberían proveer de una batería de métodos protegidos que permitan la personalización mediante la sobreescritura y regulen el acceso a la memoria heredada.
- En caso de disponer de propiedades, por su naturaleza, están destinadas a ser miembros públicos de las clases.
- La parte pública de la clase configura el interfaz implícito de la clase, se puede y se deben crear interfaces explícitos.
- Las parte pública (y protegida) debe permanecer inalterable con el paso de versiones o, por lo menos, durante un periodo razonable después de la notificación de su obsolescencia.

© JMA 2020. All rights reserved

Ley de Demeter

- La Ley de Demeter es un mecanismo de detección de acoplamiento, y nos viene a decir que nuestro objeto no debería conocer las entrañas de otros objetos con los que interactúa. Si queremos que haga algo, debemos pedírselo directamente en vez de navegar por su estructura.
- Los objetos con los que puede interactuar un método de la clase son:
 - el mismo
 - los contenidos en variables locales
 - los suministrado por los argumentos
 - los contenidos en atributos de la instancia
- Esta ley se refiere a situaciones como esta:

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```
- El problema, conocido como choque de trenes (un vagón por método), es la necesidad de conocer detalles de las clases (para realizar una operación) que deberían estar resueltos a este nivel de abstracción. El código será muy propenso a requerir modificaciones, pues la línea depende de varias clases que pueden cambiar en cualquier momento.

© JMA 2020. All rights reserved

Estructuras en orientación a objetos

- Los DTOs (Data Transfer Objects) son el paradigma de clases sin métodos y todos sus atributos públicos, son meros contenedores de información para comunicarse con sistemas remotos, bases de datos ...
- Encapsular los atributos (Java Beans) no aporta valor pero dificulta la legibilidad.
- Los Active Record son DTOs con métodos asistentes, como save y find, para simplificar la carga y descarga de datos, que no hacen que la estructura sea un objeto.

© JMA 2020. All rights reserved

CLASES

© JMA 2020. All rights reserved

S.O.L.I.D.

- SOLID es el acrónimo que acuñó Michael Feathers, basándose en los 5 principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su artículo “Design Principles and Design Patterns”.
- Los objetivos de estos 5 principios a la hora de escribir código son:
 - Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
 - Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
 - Permitir la escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.
- La aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de patrones de diseño, que permitirán minimizar el acoplamiento (grado de interdependencia que tienen dos unidades de software entre sí) y maximizar la cohesión (grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado).

© JMA 2020. All rights reserved

S.O.L.I.D.

S

- Single Responsibility Principle (SRP)
- Principio de responsabilidad única

O

- Open/Closed Principle (OCP)
- Principio de abierto-cerrado

L

- Liskov Substitution Principle (LSP)
- Principio de sustitución de Liskov

I

- Interface Segregation Principle (ISP)
- Principio de segregación de interfaces

D

- Dependency Inversion Principle (DIP)
- Principio de inversión de dependencias

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Responsabilidad Única

- “A class should have one, and only one, reason to change.”
- La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio “una clase debería tener una, y solo una, razón para cambiar”. Es esto, precisamente, “razón para cambiar”, lo que Robert C. Martin identifica como “responsabilidad”.
- El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.
- El propio Bob resume cómo hacerlo: “Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes”.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Abierto/Cerrado

- “You should be able to extend a classes behavior, without modifying it.”
- El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.
- El principio Open/Closed se suele resolver utilizando polimorfismo.
- Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías, frameworks, microservicios.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Sustitución de Liskov

“Base classes must be substitutable for their derived classes.”

- La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que “las clases base deben poder sustituirse por sus clases derivadas”.
- Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.
- Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Segregación de la Interfaz

“Make fine grained interfaces that are client specific.”

- En el cuarto principio de SOLID, el tío Bob sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para una finalidad concreta.
- En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces especializadas que definan unos pocos métodos que tener una interface generalista que fuerce a implementar muchos métodos a los que no se dará uso.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Inversión de Dependencias

“Depend on abstractions, not on concretions.”

- Llegamos al último principio: “Depende de abstracciones, no de clases concretas”.
- Así, Robert C. Martin recomienda:
 - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
 - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
- El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases. Las abstracciones se logran mediante los interfaces.

© JMA 2020. All rights reserved

Introducción

- Hay que prestar la misma atención al contenedor, la clase, que a las funciones y las instrucciones.
- Los miembros de las clases deberían estar ordenados, por ejemplo siguiendo la convención estándar de Java: primero las constantes estáticas públicas (si existen), a continuación los atributos de clase (privados) seguidos de los atributos de instancia (privados), terminando con los métodos (primero los de clase y después los de instancia).
- Los métodos, al ser funciones, seguirán las recomendaciones de las mismas: primero el método de mayor nivel de abstracción (normalmente público) seguido por los de menor nivel de abstracción (normalmente privados). A continuación el siguiente método de mayor nivel de abstracción y así sucesivamente.

© JMA 2020. All rights reserved

Tamaño de las clases

- Al igual que pasa con las funciones, la primera regla de las clases es que deben ser pequeñas. La segunda regla de las clases es que deberían ser todavía más pequeñas. En las funciones la medida eran las líneas, en las clases son las responsabilidades.
- El nombre de la clase, sustantivos, debe describir las responsabilidades que desempeña, debe ser conciso. Los nombres ambiguos o generales denotan un exceso de responsabilidades, que rompen el principio de responsabilidad única.
- Todas las responsabilidades de la clase deberían tener el mismo nivel de abstracción.

© JMA 2020. All rights reserved

Indicadores del incumplimiento del principio de responsabilidad única

- **Si en una misma clase están involucradas dos capas de la arquitectura:** En toda arquitectura, por simple que sea, debería haber una separación entre la presentación, la lógica de negocio y la persistencia.
- **El número de imports/using:** Si necesitamos importar demasiadas clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más. Los paquetes o espacios de nombres son indicadores de las actividades implicadas.
- **El número de métodos públicos:** Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos.

© JMA 2020. All rights reserved

Indicadores del incumplimiento del principio de responsabilidad única

- **Como usan los métodos a los atributos de la clase:** si en la clase podemos hacer subconjuntos de métodos con los atributos que manejan y apenas hay intersecciones, seguramente cada subconjunto es una clase componente de la principal y las intersecciones representan las interacciones de las clases componentes.
- **Problemas para crear las pruebas unitarias:** si tenemos problemas para identificar los casos de prueba u obtener la granularidad deseada, seguramente la clase es muy compleja y debería ser dividida en componentes mas elementales hasta el grado en que sean fáciles de probar.
- **Frecuencia de modificaciones:** Si una clase se ve afectada desmesuradamente por modificaciones ante los cambios del sistema es que incumple “una razón para cambiar”.

© JMA 2020. All rights reserved

Cohesión

- La cohesión se refiere al grado en que los elementos de un módulo permanecen juntos, mide la fuerza de la relación entre sus piezas de funcionalidad, cuanto mas estrecha mas alta será.
- Las clases deben tener un pequeño número de atributos y sus métodos deben manipular una o más de esos atributos. En general, cuantas más atributos manipule un método, más cohesivo será ese método para su clase. Una clase en la que cada atributo es utilizado por cada método tiene máxima cohesión.
- En general, no es aconsejable ni posible crear clases de máxima cohesión, pero si que sea lo mas alta posible. Cuando la cohesión es alta, significa que los métodos y los atributos de la clase son codependientes y se unen como un todo lógico.
- Las estrategia de mantener cortos los métodos y sus listas de parámetros puede suponer una proliferación de atributos que solo utilizan un subconjunto de métodos, bajando la cohesión. Cuando esto sucede, casi siempre significa que hay al menos una clase tratando de salir de la clase original. Hay que intentar separar las clases para maximizar la cohesión.

© JMA 2020. All rights reserved

Aislar los cambios

- Las funcionalidades de los sistemas dependen de las necesidades de las organizaciones, con el paso del tiempo las organizaciones evolucionan y cambian, por lo que cambian sus necesidades que implican cambios funcionales: es inevitable.
- Una clase que dependa de detalles concretos esta en peligro si dichos detalles cambian. Una clase que depende de otras clases corre el riesgo de que dichas clases tengan que cambiar. Se puede recurrir a los interfaces y las clases abstractas para aislar y minimizar el impacto de dichos cambios.
- El Principio de Inversión de Dependencias dice: Depende de abstracciones, no de clases concretas. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
- Los interfaces permiten definir abstracciones, que al no tener implementación no dependen de detalles. Mientras no cambien las abstracciones no cambiarán los interfaces.

© JMA 2020. All rights reserved

Aislar los cambios

- Las dependencias deben depender de abstracciones, es decir, de interfaces. Un módulo o clase debe depender de interfaces, no de implementaciones concretas. Según cambian los detalles, se cambian las implementaciones concretas o se crean nuevas, sin afectar a los dependientes.
- Este mecanismo nos obliga trabajar con abstracciones y a organizar nuestro código de una manera muy específica: hay que crear interfaces, las clases deben implementar interfaces y los tipos en las declaraciones (atributos, propiedades, parámetros y retornos, variables, ...) deben ser siempre interfaces, nunca clases.
- Es habitual complementar este mecanismo con un inyector de dependencias: el mecanismo que se encarga de instanciar los objetos y pasarlos allá donde se necesiten.
- Trabajar con abstracciones permite aislar las pruebas, que pueden comenzar antes incluso de que estén las implementaciones concretas.
- Esta reestructuración a la larga compensa por la flexibilidad que otorga a la arquitectura de nuestra aplicación.

© JMA 2020. All rights reserved

Organizar los cambios

- El Principio de Segregación de la Interfaz sugiere: Haz interfaces que sean específicas para un tipo de cliente.
- La problemática surge cuando las interfaces intentan definir más cosas de las debidas (fat interfaces). Obligara a las clases a implementar métodos que no utiliza: habitualmente lanzando una excepción o simplemente no haciendo nada. Esto rompe el Principio de Sustitución de Liskov: lanza excepciones en momentos inesperados o, en las implementaciones «por defecto», tiene efectos secundarios de difícil depuración.
- Es preferible contar con muchas interfaces especializadas que definan unos pocos métodos que tener unas pocas interfaces generalistas que fuercen a implementar muchos métodos a los que no se dará uso.
- Hay que tener en cuenta que se pueden crear fácilmente interfaces generalistas heredando (sintetizando) múltiples interfaces especializadas, pero no viceversa, una interface generalista requiere refactorización para segregarla en interfaces especializadas. Como alternativa, el patrón Adapter nos permite convertir unas interfaces en otras, por lo que se pueden usar adaptadores que conviertan la interfaz antigua en las nuevas.

© JMA 2020. All rights reserved

Observaciones

- Hacer que el software funcione y que sea limpio son dos actividades muy diferentes.
- La mayoría tenemos limitaciones, por lo que nos enfocamos en hacer que el código funcione más que en la organización y la limpieza. Lo cual es totalmente apropiado. Mantener la separación de preocupaciones es tan importante en nuestras actividades de programación como lo es en nuestros programas.
- El problema es pensar que hemos terminado una vez que el programa funciona y olvidamos la otra responsabilidad: la organización y limpieza. Pasamos al siguiente problema en lugar de retroceder y dividir las clases superpobladas en unidades desacopladas con responsabilidades únicas.
- Muchos desarrolladores temen que una gran cantidad de clases pequeñas y de un solo propósito dificulten la comprensión del panorama general. Les preocupa que deben navegar de una clase a otra para descubrir cómo se logra un trabajo más grande.

© JMA 2020. All rights reserved

Observaciones

- Sin embargo, un sistema con muchas clases pequeñas no tiene más partes móviles que un sistema con unas pocas clases grandes, hay tanto que aprender en uno como en el otro.
- Una buena estructuración en jerarquías de clase, espacios de nombres y paquetes, así como las utilidades de navegación de los entornos modernos, minimizan la problemática de muchas clases pequeñas, que aportan una flexibilidad, reusabilidad y mantenibilidad que no pueden suplir las clases grandes.
- Aun así, como el resto de principios y buenas practicas, puede parecer que no siempre compensa, en cuyo caso sólo hay que aplicarlas cuando sea necesario, donde corresponda y sin obsesionarnos con cumplirlas en cada punto del desarrollo. Pero si esperamos a que sea necesario, hay que tener en cuenta el que seguramente estemos incurriendo en una deuda técnica que habrá que abonar en el momento aplicar los principios y buenas practicas omitidos.

© JMA 2020. All rights reserved

LÍMITES

© JMA 2020. All rights reserved

Introducción

- El termino Boundaries es traducido habitualmente en el contexto de Clean Code por límites, aunque una traducción mas adecuada sería fronteras.
- El concepto de límites hace referencia a la interacción del código de desarrollo propio con el código de terceros: adquirido o de código abierto. El código de terceros nos permite obtener funcionalidad mas rápidamente a un coste menor.
- El código de terceros tiende a ser generalista, con un amplio conjunto de opciones y capacidades, que atienda múltiples escenarios y necesidades. No tenemos control sobre el.
- El código propio es mucho mas especifico al estar enfocado un conjunto de necesidades muy concretas y con lo mínimo necesario.
- Esta discrepancia conceptual produce tensiones y fricciones en la frontera que separa el código propio del código ajeno: El exceso de funcionalidad externa aporta mas de lo necesario y deseable.

© JMA 2020. All rights reserved

Frontera

- La solución pasa por crear una capa fronteriza que actúe de intermediaria entre los interfaces ajenos y los propios, solucione las discrepancias y proteja ante cambios en el versionado del código ajeno.
- La capa debe envolver el código de terceros, encapsulando la librería genérica usada para ajustar su interfaz únicamente a nuestras necesidades.
- No se trata de encapsular todo, sino de limitar las dependencias de nuestro sistema con el código ajeno a un único punto: la frontera.

© JMA 2020. All rights reserved

Aprender con pruebas unitarias

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

© JMA 2020. All rights reserved

Pruebas de aprendizaje

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

© JMA 2020. All rights reserved

Código desconocido

- Las técnicas y conceptos utilizados con el código ajeno se puede aplicar al código desconocido.
- El código desconocido hace referencia al códigos externo: se conoce la frontera interior, las necesidades del código propio, pero se desconoce las forma que tendrá la frontera exterior, que código propio, ajeno o API le dará soporte.
- La frontera interior se define mediante interfaces que formalizan las abstracciones con las que interactúa el código interior. Cuando se resuelva la indeterminación del código desconocido y pase a ser conocido, usando el patrón Adapter se convertirán las interfaces de la frontera interior a las nuevas.
- Esto permite avanzar con garantías en sistemas que aun no están completamente definidos.

© JMA 2020. All rights reserved

Límites limpios

- En los límites ocurren los cambios y deben ser especialmente limpios.
- Los buenos códigos acomodan los cambios con un mínimo de modificaciones.
- Al trabajar con código ajeno hay que tener especial cuidado en proteger nuestro código y minimizar el impacto del versionado ajeno que no controlamos.
- Hay que minimizar las dependencias con una separación evidente y pruebas que establezcan las expectativas.
- La definición y uso de Wrappers o Adapters ayudan en esta labor.

© JMA 2020. All rights reserved

SISTEMAS

© JMA 2020. All rights reserved

Introducción

- Los sistemas también deben ser limpios. Una arquitectura invasiva afecta a la lógica del dominio y a la agilidad. Cuando la lógica del dominio se ve afectada, la calidad se resiente porque se ocultan los errores y las historias se vuelven más difíciles de implementar. Si la agilidad se ve comprometida, la productividad sufre y los beneficios de TDD se pierden.
 - En todos los niveles de abstracción, la intención de los objetos debe ser clara. Esto solo sucederá si se crean POJO y se utilizan mecanismos similares a los aspectos para incorporar otras inquietudes de implementación de manera no invasiva.
 - Se esté diseñando sistemas o módulos individuales, siempre se debe usar lo más simple que funcione.
-

© JMA 2020. All rights reserved

Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- La técnica de instanciación tardía tiene ventajas: no se instancia a no ser que sea necesario, lo que acelera el tiempo de arranque y evita los null.

```
public Service getService() {  
    if (service == null) service = new MyServiceImpl(...);  
    return service;  
}
```
- Como inconvenientes: no se podrá compilar sin antes resolver la dependencias (aun si no se usa), no se puede sustituir por un doble de pruebas, crea problemas de modularidad y duplicidad diseminados por todo el sistema.

© JMA 2020. All rights reserved

Construcción

- Una forma de separar y centralizar la construcción del uso consiste en trasladar todos los aspectos de la construcción al main o módulos invocados por main y diseñar el resto del sistema suponiendo que todos los objetos se han instanciado y conectado correctamente. Inviabile salvo para sistemas muy pequeños.
- La instanciación de factorías (patrón Factoría Abstracta) en el main permite delegar la instanciación pero mantener separados los detalles de la construcción del uso de la instancia. Costoso en términos de arranque.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.

© JMA 2020. All rights reserved

Evolución

- Los sistemas de software son únicos en comparación con los sistemas físicos. Sus arquitecturas pueden crecer y variar gradualmente, si se mantienen adecuadamente la separación de conceptos.
- Conseguir sistemas perfectos y completos a la primera es un imposible. En cambio, deberíamos implementar solo las historias de hoy, refactorizar y expandir el sistema para implementar nuevas historias mañana. Esta es la esencia de la agilidad iterativa e incremental. El desarrollo basado en pruebas, con la refactorización y el código limpio que producen hacen que esto funcione a nivel de código.

© JMA 2020. All rights reserved

Aspectos transversales

- La programación orientada a aspectos (AOP - Aspect Oriented Programming) es un paradigma de programación que intenta formalizar y representar de forma concisa los elementos que son transversales a todo el sistema.
- Para su implementación en Java disponemos de:
 - Proxies de Java
 - Spring AOP, JBoss AOP, ...
 - AspectJ

© JMA 2020. All rights reserved

EMERGENCIA

© JMA 2020. All rights reserved

Limpieza a través de diseños emergentes

- Según Ken Beck un hay 4 reglas que son fundamentales para crear un software bien diseñado.
- Es aquel que:
 1. Ejecuta todas las pruebas
 2. No contiene duplicados
 3. Expresa la intención del programador
 4. Minimiza el numero de clases y métodos

© JMA 2020. All rights reserved

Ejecuta todas las pruebas

- Un sistema puede funcionar de excelente manera sobre el papel pero si no existe una manera de comprobar que funciona, el esfuerzo será cuestionable.
- Un sistema minuciosamente probado y que supera todas las pruebas en todo momento se denomina comprobable.
- Un sistema comprobable es un sistema confiables.
- La creación de pruebas conduce a obtener mejores diseños.

© JMA 2020. All rights reserved

Refactorizar

- Una vez creadas las pruebas, se debe mantener limpio el código y las clases. Para ello, se refactoriza el código progresivamente.
- La refactorización es la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin esto suponer alteraciones en su comportamiento externo.
- El resto de las reglas, 2 al 4, se aplican utilizando la refactorización.

© JMA 2020. All rights reserved

No contiene duplicados

- Los duplicados significan mas trabajo innecesario.
- Suponen un esfuerzo adicional como: líneas de código similar o duplicación en implementación.
- Producen inconsistencias: cuando los cambios no se han aplicado de la misma forma a todos los duplicados.
- El código limpio requiere reducir código, aunque sean unas pocas líneas.

© JMA 2020. All rights reserved

Expresa la intención del programador

- Es fácil generar código, que nosotros a la hora de programar entendamos, pero ¿los demás como podrán entenderlo?
- El principal coste de un proyecto de software es su mantenimiento a largo plazo, por ello cuanto mas claro sea el código, menos tiempo emplearan otros en intentar compréndelo para poder mantenerlo.
- El programador puede expresarse de forma entendible:
 - Eligiendo buenos nombres: los nombre de una clase o función deben ser concordantes con sus funcionalidades.
 - Manteniendo las funciones y clases cortas. Lo que usualmente facilita su entendimiento y modificación.
 - Usando una nomenclatura estándar.
 - Con pruebas bien escritas que actúan de documentación.

© JMA 2020. All rights reserved

Clases y métodos mínimos

- El esfuerzo por reducir el tamaño de clases y métodos puede llevar a crear demasiadas clases y métodos mínimos.
- Esta regla matiza el esfuerzo anterior: minimizar la cantidad de clases y métodos.
- Debe evitarse el dogmatismo sin sentido en pro de un enfoque mas pragmático.
- De las cuatro reglas es la que menor prioridad tiene, es mucho mas importante contar con pruebas, eliminar duplicados y expresarse correctamente.

© JMA 2020. All rights reserved

CONCURRENCIA

© JMA 2020. All rights reserved

Introducción

- El hecho de separar el qué del cuándo en programación concurrente hace que tengamos que tomar precauciones para no tener problemas. Las prácticas a adoptar son:
 - SRP, un único motivo para cambiar en cada clase → separar el código que controla la concurrencia del código de la aplicación
 - Encapsula datos y protégelos con `synchronized` cuando se compartan entre procesos
 - Intentar minimizar la compartición de datos entre procesos, los procesos deben ser independientes
 - Utiliza las clases del entorno de desarrollo específicas de concurrencia (ej. `Java.util.concurrent`)
- Distintos modelos: productor-consumidor, lector-escritor, la cena de los filósofos (condiciones de carrera), ...
- Planificar y probar concienzudamente el código de cierre de un proceso, para evitar bloqueos en nuevos procesos.
- Recomendaciones:
 - Primero que funcione sin procesos, esto permite identificar los fallos que no tienen que ver con concurrencia
 - No ignorar los fallos que no se pueden reproducir
 - Se tienen que poder probar, mejor diseñar con esto en mente

© JMA 2020. All rights reserved

ARQUITECTURA

© JMA 2020. All rights reserved

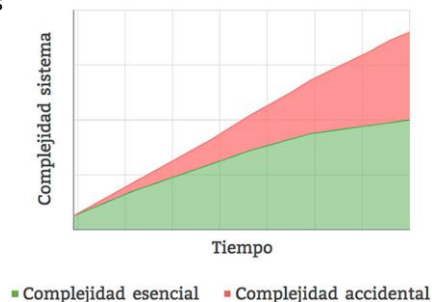
Arquitectura ágil

- Es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz y dispone de las pruebas unitarias adecuadas de la arquitectura del sistema.
- Es posible iniciar un proyecto de software con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura.
- Los estándares facilitan la reutilización de ideas y componentes, reclutar personas con experiencia relevante, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares a veces puede llevar demasiado tiempo para que la industria espere, y algunos estándares pierden contacto con las necesidades reales de los adoptantes a los que están destinados a servir. Hay que usar estándares cuando añadan un valor demostrable.

© JMA 2020. All rights reserved

Complejidad

- La complejidad esencial está causada por el problema a resolver y nada puede eliminarlo; si los usuarios quieren que un programa haga 30 cosas diferentes, entonces esas 30 cosas son esenciales y el programa debe hacer esas 30 cosas diferentes.
- La complejidad accidental se relaciona con problemas que los desarrolladores crean y pueden solucionar; por ejemplo, los detalles de escribir y optimizar el código o las demoras causadas por el procesamiento por lotes.
- La complejidad accidental ha disminuido sustancialmente y los programadores de hoy dedican la mayor parte de su tiempo a abordar la complejidad esencial.



© JMA 2020. All rights reserved

Arquitectura Limpia

- En los últimos años, hemos visto una amplia gama de ideas sobre la arquitectura de los sistemas. Éstos incluyen:
 - Arquitectura Hexagonal (también conocida como Puertos y Adaptadores) por Alistair Cockburn y adoptada por Steve Freeman y Nat Pryce en su libro Growing Object Oriented Software
 - Arquitectura de cebolla (Onion Architecture) de Jeffrey Palermo
 - Screaming Architecture de Robert Martin
 - DCI de James Coplien y Trygve Reenskaug.
 - BCE por Ivar Jacobson de su libro Ingeniería de software orientada a objetos: un enfoque basado en casos de uso.
- Aunque todas estas arquitecturas varían algo en sus detalles, son muy similares. Todos tienen el mismo objetivo, que es la separación de conceptos. Todos logran esta separación dividiendo el software en capas. Cada uno tiene al menos una capa para las reglas de negocio y otra para las interfaces.

© JMA 2020. All rights reserved

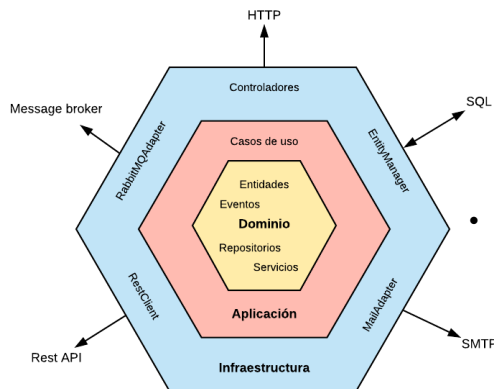
Arquitecturas multicapas

- Cada capa se apoya en la capa subsiguiente, capas horizontales, que depende de las capas debajo de ella, que a su vez dependerá de alguna infraestructura común y servicios públicos.
- El gran inconveniente de esta arquitectura en capas de arriba hacia abajo es el acoplamiento que crea.
- Hay sistemas estrictos (strict layered systems) y relajados (relaxed layered systems) que determinarán el nivel de acoplamiento de las dependencias entre capas.
- Las preocupaciones transversales provocan la aparición de capas verticales.
- Todo esto crea una complejidad accidental innecesaria.



© JMA 2020. All rights reserved

Arquitectura Hexagonal

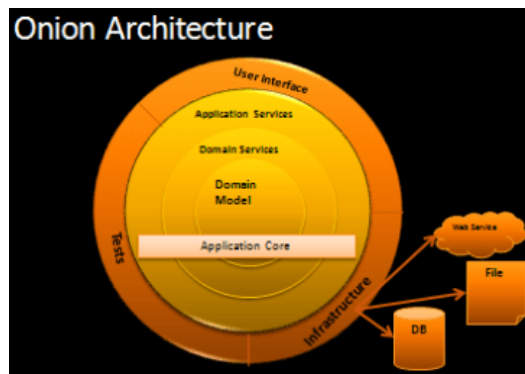


- Propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo. Mediante el principio de inversión de dependencias, en vez de hacer uso explícito, nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas.
- También llamada puertos y adaptadores:
 - Puerto: definición de una interfaz pública.
 - Adapter: especialización de un puerto para un contexto concreto.

© JMA 2020. All rights reserved

Onion Architecture

El termino Onion Architecture o Arquitectura cebolla fue acuñada por Jeffrey Palermo, es un patrón arquitectónico basado en capas concéntricas, de fuera a dentro (núcleo), y en mecanismos de inyección de dependencias, para disminuir el el acoplamiento entre las capas.



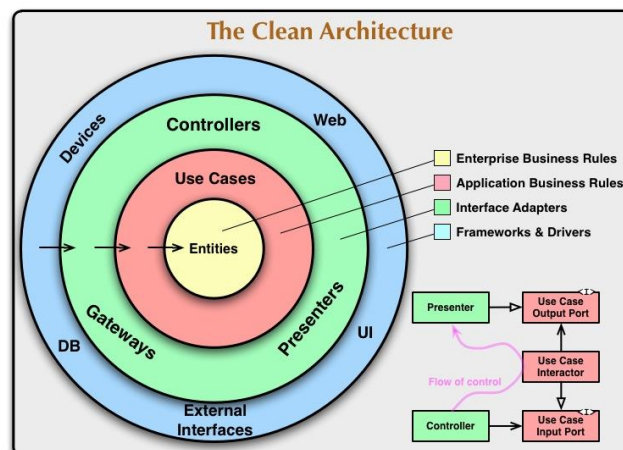
© JMA 2020. All rights reserved

Arquitectura Limpia

- Independiente de los marcos: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de funciones. Esto permite utilizar dichos marcos como herramientas, en lugar de tener que ajustar el sistema a las limitaciones impuestas.
- Comprobable: Las reglas de negocio se pueden probar sin la interfaz de usuario, la base de datos, el servidor web o cualquier otro elemento externo.
- Independiente de la IU: La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría reemplazarse por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas de negocio.
- Independiente de la base de datos: Puede cambiar Oracle o SQL Server por Mongo, BigTable, CouchDB u otra cosa. Las reglas de negocio no están vinculadas a la base de datos.
- Independiente de cualquier dependencia externa: De hecho, las reglas de negocio simplemente no saben nada sobre el mundo exterior.

© JMA 2020. All rights reserved

Arquitectura Limpia



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

© JMA 2020. All rights reserved

La regla de la dependencia

- El diagrama presenta una arquitectura basada en “capas de cebolla”, enfoque propuesto por Jeffrey Palermo.
- Los círculos concéntricos representan diferentes áreas del software. En general, cuanto más avanza, mayor nivel se vuelve el software. Los círculos exteriores son mecanismos. Los círculos internos son políticas.
- La regla primordial que hace que esta arquitectura funcione es la regla de dependencia. Esta regla dice que las dependencias del código fuente solo pueden apuntar hacia adentro. Nada en un círculo interno puede saber nada sobre algo en un círculo externo. En particular, el nombre de algo declarado en un círculo exterior no debe ser mencionado por el código en un círculo interior. Eso incluye funciones, clases, variables o cualquier otra entidad de software nombrada.
- Del mismo modo, los formatos de datos utilizados en un círculo exterior no deben ser utilizados por un círculo interior, especialmente si esos formatos son generados por un marco en un círculo exterior. No queremos que nada en un círculo externo impacte en los círculos internos.

© JMA 2020. All rights reserved

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaríamos que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

© JMA 2020. All rights reserved

Casos de uso

- Esta capa contiene las reglas de negocio específicas de la aplicación. Encapsula e implementa todos los casos de uso del sistema. Éstos orquestan el flujo de datos hacia-desde las entidades y hacen que las entidades usen su lógica de negocio para conseguir el objetivo del caso de uso.
- No esperamos que los cambios en esta capa afecten a las entidades. Tampoco esperamos que esta capa se vea afectada por cambios en las externalidades como la base de datos, la interfaz de usuario o cualquiera de los marcos comunes. Esta capa está aislada de tales preocupaciones.
- Sin embargo, si es esperable que los cambios en el funcionamiento de la aplicación o en las reglas del negocio vayan a afectar a los casos de uso y, por lo tanto, a esta capa. Es decir, si los detalles de un caso de uso cambian, algún código de esta capa se verá afectado y habrá que cambiarlo.

© JMA 2020. All rights reserved

Adaptadores de interfaz

- Esta capa contiene un conjunto de adaptadores que convierten los datos desde el formato más conveniente para el caso de uso y entidades al formato más conveniente o aceptado por elementos externos como la base de datos o la UI. Por ejemplo, esta capa es la que contendrá por completo la arquitectura MVC de una GUI.
- Los presentadores, vistas y controladores pertenecen a esta capa. Los modelos son sólo estructuras que se pasan desde los controladores a los casos de uso y de vuelta desde los casos de uso a los presentadores y/o vistas.
- Del mismo modo, los datos son convertidos en esta capa, desde la forma de las entidades y casos de uso a la forma conveniente para la herramienta de persistencia de datos.
- En este círculo se encuentran también los demás adaptadores encargados de convertir datos obtenidos de una fuente externa, como un servicio externo, al formato interno usado por los casos de uso y las entidades.

© JMA 2020. All rights reserved

Frameworks y drivers

- El círculo más externo se compone generalmente de frameworks y herramientas como la base de datos, el framework web, etc.
- Generalmente en esta capa sólo se escribe código que actúa de “pegamento” con los círculos interiores.
- En esta capa es donde van todos los detalles. La web es un detalle. La base de datos es un detalle. Se mantienen estas cosas afuera, donde no pueden hacer mucho daño.

© JMA 2020. All rights reserved

Cruzando fronteras

- En la parte inferior derecha del diagrama hay un ejemplo de cómo se cruzan las fronteras del círculo. Muestra cómo los controladores y presentadores se comunican con los casos de uso del círculo interno.
- El control del flujo empieza en el controlador, atraviesa el caso de uso y finaliza en el presentador. La dependencia a nivel de código fuente apunta hacia adentro, hacia los casos de uso.
- Se resuelve esta supuesta contradicción usando el principio de inversión de dependencia, es decir, desarrollando interfaces y relaciones de herencia de tal manera que las dependencias de código fuente se oponen al flujo de control en sólo algunos puntos a través de la frontera.
- Considerando que los casos de uso necesitan llamar al presentador, esta llamada no debe ser directa ya que se violaría el principio de la regla de dependencia. Ninguna implementación de un círculo exterior puede ser llamado por un círculo interior. En esta situación el caso de uso llama a una interfaz definida en el círculo interno y hace que el presentador implemente dicha interfaz en el círculo exterior.

© JMA 2020. All rights reserved

Datos que cruzan los límites

- Las estructuras de datos que se pasan a través de las fronteras deben ser simples.
- No se debería pasar entidades o filas de base de datos. Muchos frameworks de base de datos devuelve los datos en el formato de respuesta de una query. Por ejemplo, un RowStructure. Pasar los datos en ese formato violaría la regla de dependencia obligando a un círculo interno saber algo sobre un círculo exterior.
- Las estructuras de datos que tienen cualquier tipo de dependencia con capas exteriores viola la regla de dependencia.
- Los datos que cruzan las fronteras deben ser estructuras de datos simples. Se puede utilizar estructuras básicas u objetos Data Transfer. O los datos pueden ser argumentos en las llamadas a funciones. O bien, almacenar en un diccionario o hash. Cuando se pasan datos a través de una frontera, siempre debe ser en la forma más conveniente para el círculo interior.

© JMA 2020. All rights reserved

Beneficios

- Cumplir con estas simples reglas no es difícil y ahorrará muchos dolores de cabeza en el futuro.
- Al separar el software en capas y cumplir con la regla de dependencia, obliga a que nuestro dominio no esté acoplado a nada externo mediante el uso de interfaces propias del dominio que son implementadas por elementos externos (Alta reutilización: Principio de Responsabilidad Única (SRP) de SOLID).
- Esto creará un sistema que es intrínsecamente comprobable, con todos los beneficios que ello implica (Alta testabilidad: Aplicación del Principio de Inversión de Dependencias (DIP) de SOLID para la interacción del dominio con el resto de elementos)
- Permite estar preparado para cambiar los detalles de implementación, cuando alguna de las partes externas del sistema se vuelve obsoleta, como la base de datos o el marco web, se pueden reemplazar con un mínimo de esfuerzo (Alta tolerancia al cambio: Principio de Abierto/Cerrado (OCP) de SOLID derivado de la aplicación del DIP).
- Permite postergar decisiones importantes.

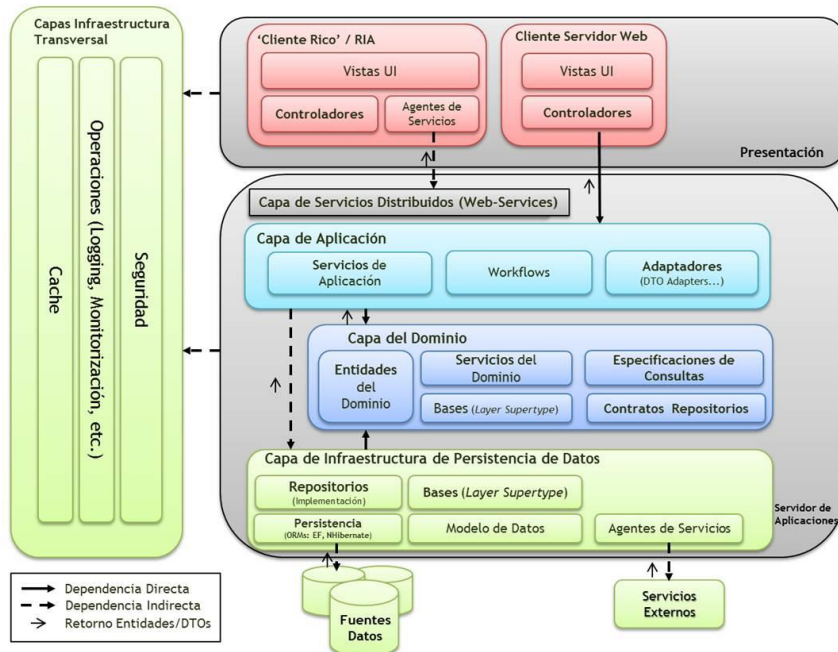
© JMA 2020. All rights reserved

Postergar decisiones importantes

- Una buena arquitectura nos permite postergar decisiones importantes, esto no significa que estemos obligados a hacerlo. Sin embargo, al poder postergarlas tenemos muchísima flexibilidad.
- Cuanto más tarde tomemos una decisión más conocimiento tendremos del negocio y del entorno técnico:
 - Soluciones más simples/sencillas
 - Soluciones más pequeñas y manejables
 - Mayor productividad, minimiza: hacer cosas que no aporte valor real al final, retrabajos y adaptaciones
- Nuestro objetivo es tener mucho juego de cintura, reaccionar bien y rápido a cualquier cosa y sin echarnos las manos a la cabeza. Cuanto más postponemos, añadimos menos cosas innecesarias, menos llenamos la mochila y es más difícil coger lastre, por lo que podemos viajar ligeros. Como cualquiera sabe: una mudanza se hace fácil si tenemos pocas cosas.

© JMA 2020. All rights reserved

Arquitectura N-Capas con Orientación al Dominio



Capas de DDD

- **Interface de usuario (User Interface)**
 - Responsable de presentar la información al usuario, interpretar sus acciones y enviarlas a la aplicación.
- **Aplicación (Application)**
 - Responsable de coordinar todos los elementos de la aplicación. No contiene lógica de negocio ni mantiene el estado de los objetos de negocio. Es responsable de mantener el estado de la aplicación y del flujo de esta.
- **Dominio (Domain)**
 - Contiene la información sobre el Dominio. Es el núcleo de la parte de la aplicación que contiene las reglas de negocio. Es responsable de mantener el estado de los objetos de negocio. La persistencia de estos objetos se delega en la capa de infraestructura.
- **Infraestructura (Infrastructure)**
 - Esta capa es la capa de soporte para el resto de capas. Provee la comunicación entre las otras capas, implementa la persistencia de los objetos de negocio y las librerías de soporte para las otras capas (Interface, Comunicación, Almacenamiento, etc..)
- Dado que son capas conceptuales, su implementación puede ser muy variada y en una misma aplicación, tendremos partes o componentes que formen parte de cada una de estas capas.

© JMA 2020. All rights reserved

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
 - Debe ser pública (no puede ser estar anidada ni final o tener miembros finales)
 - Deben tener un constructor público sin ningún tipo de argumentos.
 - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
 - Debe tener una clave primaria
 - Debería sobrescribir equals para la identidad

© JMA 2020. All rights reserved

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona un forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
 - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
 - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

© JMA 2020. All rights reserved

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red o de diferentes capas.
- Su finalidad es:
 - Desacoplar del nivel de servicio de la capa de base de datos.
 - Quitar las referencias circulares.
 - Ocultar determinadas propiedades que los clientes no deberían ver.
 - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
 - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más conveniente para los clientes.
 - Evitar el "exceso" y las vulnerabilidades por publicación.

© JMA 2020. All rights reserved

Repositorio

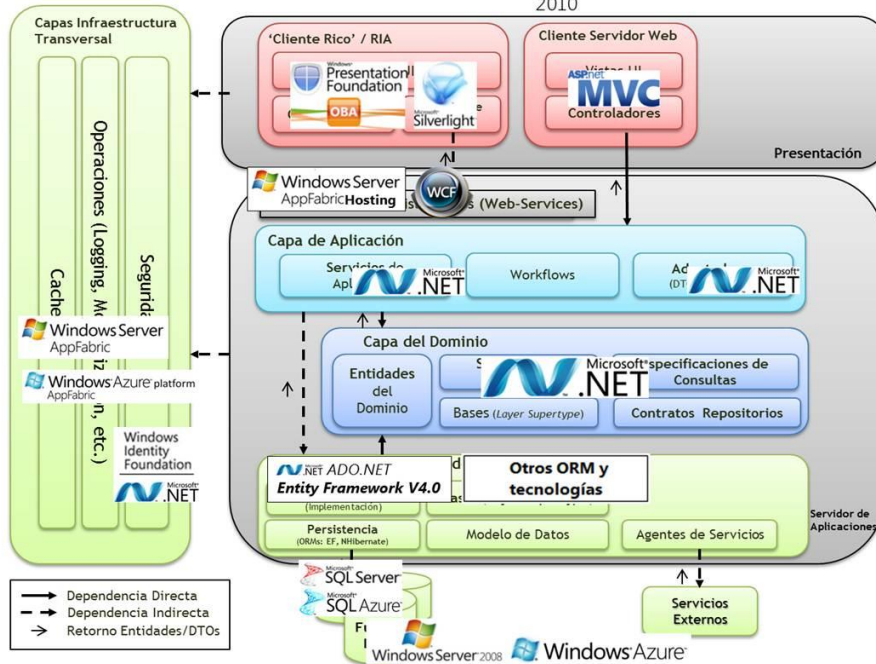
- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

© JMA 2020. All rights reserved

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen.
- Podemos dividir los servicios en tres tipos diferentes:
 - Domain services
 - Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
 - Application services
 - Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
 - Infrastructure services
 - Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

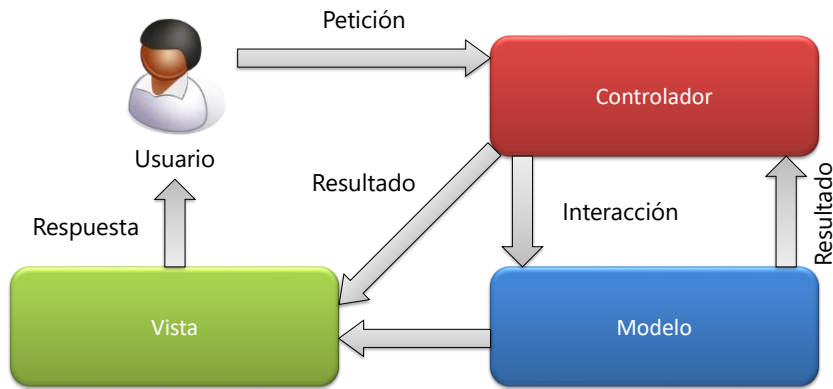
© JMA 2020. All rights reserved



El patrón MVC

- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, ...) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net ...)

El patrón MVC



© JMA 2020. All rights reserved

El patrón MVC



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



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

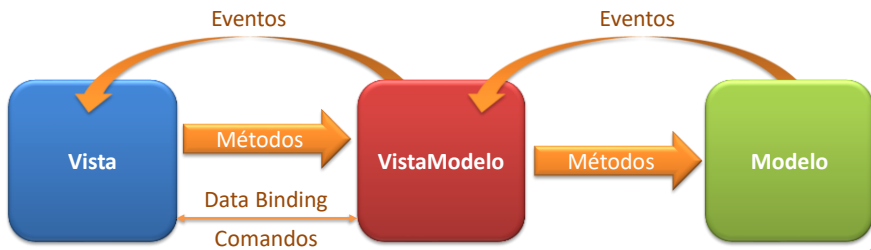


- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

© JMA 2020. All rights reserved

Model View ViewModel (MVVM)

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



© JMA 2020. All rights reserved

¿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.

© JMA 2020. All rights reserved

Inversión de Control

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

© JMA 2020. All rights reserved

Inyección de Dependencias

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

© JMA 2020. All rights reserved

Desarrollo Dirigido por Comportamiento (BDD)

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

© JMA 2020. All rights reserved

BDD

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

© JMA 2020. All rights reserved

HERRAMIENTAS DE PRUEBA DE VISUAL STUDIO

© JMA 2020. All rights reserved

Herramientas de prueba de Visual Studio

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

PRUEBAS: MSTest

© JMA 2020. 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 2020. All rights reserved

Casos de pruebas

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

© JMA 2020. All rights reserved

Clases y métodos de prueba

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

© JMA 2020. All rights reserved

Fixtures

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

© JMA 2020. All rights reserved

Ciclo de vida de instancia de prueba

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

© JMA 2020. All rights reserved

Ciclo de vida de instancia de prueba

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

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

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

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

© JMA 2020. All rights reserved

TestContext

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

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

© JMA 2020. All rights reserved

Propiedades de TestContext

- **TestName:** Obtiene el nombre de la prueba.
- **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 2020. All rights reserved

Métodos de prueba

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

© JMA 2020. All rights reserved

Métodos de prueba

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

© JMA 2020. All rights reserved

Aserciones

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

© JMA 2020. All rights reserved

Aserciones: Clase Assert

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

© JMA 2020. All rights reserved

Aserciones: Excepciones

- Todos los métodos Assert lanzan la excepción `AssertFailedException` cuando no se cumple la aserción, que marca la prueba como “Con error”.
- Se puede implementar la comprobación manualmente, en cuyo caso se utiliza el método `Assert.Fail()` para marcar la prueba como “Con error”.
- Cuando el código del método de pruebas todavía no 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 2020. All rights reserved

Aserciones: Clase `StringAssert`

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

© JMA 2020. All rights reserved

Aserciones: Clase CollectionAssert

- **AllItemsAreInstancesOfType**: Comprueba si todos los elementos de la colección especificada son instancias del tipo esperado y genera una `AssertFailedException` si el tipo esperado no está en la jerarquía de herencia de uno o más de los elementos.
- **AllItemsAreNotNull**: Comprueba si todos los elementos de la colección especificada no son nulos y genera una `AssertFailedException` si algún elemento es nulo.
- **AllItemsAreUnique**: Comprueba si todos los elementos de la colección especificada son únicos o no y arroja una `AssertFailedException` si dos elementos de la colección son iguales.
- **AreEqual**: Comprueba si las colecciones especificadas son iguales y genera una `AssertFailedException` si las dos colecciones no son iguales. La igualdad se define como tener los mismos elementos en el mismo orden y cantidad. Diferentes referencias al mismo valor se consideran iguales.
- **AreNotEqual**: Comprueba si las colecciones especificadas son desiguales y genera una `AssertFailedException` si 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 2020. All rights reserved

Documentar los resultados

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

© JMA 2020. All rights reserved

Control de ejecución

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

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

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

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

© JMA 2020. All rights reserved

Rasgos

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

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

© JMA 2020. All rights reserved

Listas de reproducción personalizadas

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

© JMA 2020. All rights reserved

Prueba unitaria con parámetros

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

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

© JMA 2020. All rights reserved

Pruebas unitarias para métodos genéricos

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

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

© JMA 2020. All rights reserved

Probar métodos no públicos

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

```
var arrange = new Tipo();  
MethodInfo privado = arrange.GetType().GetMethod(  
    "MyPrivateMethod", BindingFlags.NonPublic |  
    BindingFlags.Instance);  
var 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 2020. All rights reserved

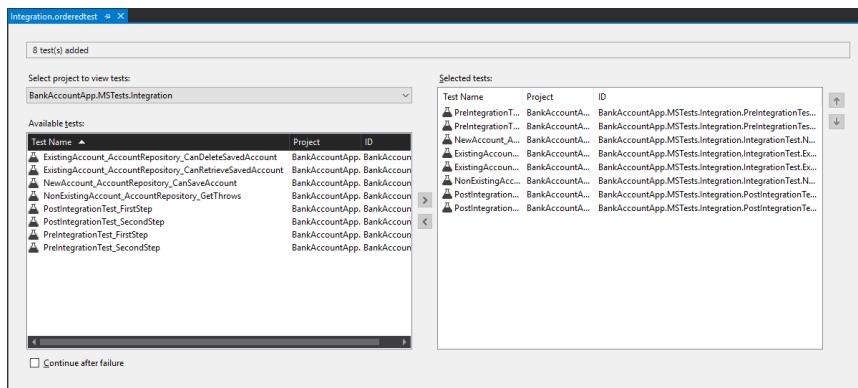
Pruebas ordenadas (Obsoletas)

MSTestv2 no es compatible con ordersTests.

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

© JMA 2020. All rights reserved

Pruebas ordenadas



© JMA 2020. All rights reserved

Pruebas genéricas

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

© JMA 2020. All rights reserved

DATA DRIVEN TESTING

© JMA 2020. All rights reserved

Data Driven Testing (DDT)

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

© JMA 2020. All rights reserved

Pruebas parametrizadas

- Los métodos de prueba pueden tener parámetros y hay disponibles varios atributos para indicar qué argumentos se suministran a las pruebas.
- Múltiples conjuntos de argumentos desencadenan la creación de múltiples pruebas. Todos los argumentos se crean en el punto de carga de las pruebas, por lo que resultados de los casos de prueba individuales se pueden consultar en los detalles del explorador de pruebas.
- Con el atributo [DataRow] se definen los datos insertados como argumentos en los parámetros de un método de prueba para un caso de prueba. El número de valores suministrados debe coincidir exactamente con el número de parámetros.

```
[TestMethod]
[DataRow(1, 6.2832)]
[DataRow(0.5, 3.1416)]
public void AreaDataTest(double radio, double expected) {
```
- Si no se supera uno de los caso de prueba se marca la prueba como no superada pero no detiene la ejecución del resto de los casos.

© JMA 2020. All rights reserved

Pruebas controladas por datos

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

© JMA 2020. All rights reserved

Método de prueba

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

```
[TestMethod]
[DeploymentItem(@"..\..\data.csv")]
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
"|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential)]
public void My_Test() {
```
- Para obtener acceso a los datos de la tabla se utiliza el indizador DataRow del TestContext.

```
public TestContext TestContext { get; set; }
```
- DataRow es un objeto System.Data.DataRow, por lo que se recuperan valores de columna mediante los nombres de columna o índice. Dado que los valores se devuelven como objetos, es necesario convertirlos al tipo adecuado:

```
int x = Convert.ToInt32(TestContext.DataRow[0]);
```

© JMA 2020. All rights reserved

Prueba unitaria con parámetros

- Para el fichero CSV

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

- Se realiza la prueba:

```
public TestContext TestContext { get; set; }  
  
public void AreaTestHelper(double radio, double expected) {  
    var arrange = new NuevoTipo();  
    Assert.AreEqual(expected, Math.Round(arrange.Area(radio), 4));  
}  
  
[TestMethod]  
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",  
    "|DataDirectory|\\AreaTest.csv", "AreaTest#csv", DataAccessMethod.Sequential)]  
public void AreaTest() {  
    AreaTestHelper(double.Parse(TestContext.DataRow[0].ToString()),  
        double.Parse(TestContext.DataRow["area"].ToString()));  
}
```

© JMA 2020. All rights reserved

Ejecutar la prueba y ver los resultados

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

© JMA 2020. All rights reserved

Tipos y atributos de origen de datos

- CSV
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\data.csv", "data#csv", DataAccessMethod.Sequential), DeploymentItem("data.csv"), TestMethod]
- Excel
DataSource("System.Data.Odbc", "Dsn=ExcelFiles;Driver={Microsoft Excel Driver (*.xls)};dbq=|DataDirectory|\\Data.xls;defaultdir=.;driverid=790;maxbufferSize=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 2020. All rights reserved

Configuración en app.config

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

[DataSource("MyExcelDataSource")]
```

© JMA 2020. All rights reserved

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

moq

© JMA 2020. All rights reserved

Introducción

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

© JMA 2020. All rights reserved

Dobles de prueba

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

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

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

© JMA 2020. All rights reserved

Dobles de prueba

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

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

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

© JMA 2020. All rights reserved

Suplantación

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

© JMA 2020. All rights reserved

Suplantación

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

© JMA 2020. All rights reserved

Suplantación

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

© JMA 2020. All rights reserved

Suplantación

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

© JMA 2020. All rights reserved

Suplantación

- Para devolver una secuencia de valores:

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

© JMA 2020. All rights reserved

Verificación

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

© JMA 2020. All rights reserved

Verificación

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

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

© JMA 2020. All rights reserved

Verificación

- Para verificar que se han utilizado todas las suplantaciones:

```
mock.VerifyAll();
```
- Para marcar como verificable una suplantación y realizar una verificación conjunta:

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

© JMA 2020. All rights reserved

Propiedades

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

© JMA 2020. All rights reserved

Eventos

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

© JMA 2020. All rights reserved

Callbacks

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

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

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

© JMA 2020. All rights reserved

LINQ to Mocks

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

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

- Múltiples suplantaciones en un solo doble:

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

- Múltiples suplantaciones y dobles:

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

© JMA 2020. All rights reserved

Conclusión

