Automatización de pruebas unitarias y de integración en el desarrollo de Software

Guión de jornadas

Escaleta de curso

Bienvenida

Introducción

El presente guión de jornadas corresponde al dictado del curso **Automatización de pruebas unitarias y de integración en el desarrollo de Software** desarrollado por el equipo del **Laboratorio de Software** (ex Laboratorio de Testing y Aseguramiento de la Calidad) del centro **INTI Córdoba**.

En él encontraremos una serie de temas e informaciones que apoyan y complementan el dictado del curso de carácter práctico, enfocado a generar en quienes lo atiendan una experiencia real y práctica del proceso de automatización de pruebas unitarias y pruebas de integración dentro del proceso de desarrollo de Software y del mismo proceso de Calidad, considerando también un consistente sustento teórico.

Así, reforzando la orientación pragmática del curso y el equipo capacitador, se tiene como objetivo que los asistentes finalicen el curso con una experiencia de caso real derivada directamente de la industria, de modo que puedan impactar en sus organizaciones y procesos los beneficios de estas temáticas con resultados tangibles y de provecho.

Consideraciones del material de jornadas

El formato de **guión** seleccionado para el material responde a una necesidad de dictado. Esto es, se establecen pautas, temáticas e informaciones de referencia pero el curso en desarrollo tomará los caminos que el grupo conformado para recibirlo y dictarlo consideren más oportunos y de criterio al momento.

Esto busca una participación activa y a modo de “coaching”, asegurando exponer realidades, acercar a los participantes y potenciar conocimientos a partir de esa apertura.

El (o los) caso(s) **práctico(s)** a tratarse será convenido por el equipo capacitador según requerimientos particulares de las jornadas y el grupo receptor.

Sobre el curso

El presente curso está basado en material de cursos y charlas sobre testing y testing de unidad en particular.

Presenta fundamentos y aplicación práctica de herramientas para sistematizar tareas vinculadas al testing, teniendo foco en un lenguaje y herramienta particular: **Java + Junit,** pero los contenidos abarcados son trasladables a otros contextos (lenguajes y/o herramientas para testing de unidad).

Día 1

Esta sección inicial apunta a establecer los conceptos básicos y darles un contenido para el desarrollo del curso **Automatización de pruebas unitarias y de integración en el desarrollo de Software**.

Como bienvenida se recorrerá un camino de sensibilización y contextualización que abran puerta a la importancia del tema, posicionen las ideas que guiarán el resto del desarrollo de la capacitación y expongan las principales misiones, tareas y componentes de la automatización de pruebas unitarias y de integración en el proceso de desarrollo de Software.

Introducción y calidad

Uno de los objetivos principales en el desarrollo de software es conseguir productos de alta calidad, ella involucra una serie de atributos que van definiendo espacios donde puede referirse una medida o un grado de calidad, tales como: confiabilidad, mantenibilidad, interoperabilidad, etc.

La confiabilidad es uno de los atributos de calidad más importantes, que implica un nivel de madurez y una respuesta a los requerimientos de la herramienta definiendo una fiabilidad o no de los mismos.

Una de las medidas de calidad mejor establecidas es la cantidad de defectos por líneas de código, es así dada su posibilidad de ser medida con un buen grado de exactitud y de determinar la confiabilidad que puede tenerse sobre el software.

Validación y verificación

Las tareas de verificación y validación ayudan a mostrar que el software cubre las expectativas para las cuales fue construido. Permiten analizar la calidad del software y tomar acciones para mejorarla.

Verificación: el software debería realizar lo que su especificación indica. Responde a la pregunta *¿Construimos el producto correctamente?*

Validación: el software debería hacer lo que el usuario requiere de él: *¿Construimos el producto correcto?*

Esto no significa garantizar que el software será completamente libre de defectos, sino comprobar que debe ser “lo suficientemente bueno” para el uso que se le pretende dar (y ese uso determina el grado de confianza que se requiere del mismo).

El proceso

V&V debería aplicarse en cada instancia del proceso de desarrollo, teniendo dos objetivos principales:

1. Descubrir defectos en el sistema
2. Medir si el sistema es “usable” en una situación de operación del mismo.

Una manera de realizar tareas de V&V es a través de análisis (de programas, modelos, especificaciones, documentos, etc.). En particular para referirnos a código fuente, existen las acciones de análisis estático y análisis dinámico.

Análisis

La descomposición en partes, o la metodología para llevar adelante un estudio estructural y profundo de un software se basa principalmente en el análisis que, refiriéndose a código fuente, puede ser de dos maneras según su momento y objeto.

Dinámico

El análisis **dinámico** examina las propiedades del software mediante su ejecución extrayendo información de las corridas, los modelos, ejecutables, prototipos y programas. Es sumamente “preciso” pero intrínsecamente incompleto.

Ejemplo paradigmático: testing funcional.

Estático

El análisis **estático** infiere propiedades del software mediante el examen de la documentación y el código fuente. Generalmente es impreciso; en sus versiones “automáticas” puede dar lugar a falsos positivos/negativos y con frecuencia es conservador.

Ejemplo paradigmático: inspección manual.

Otros: chequeo de tipos (en compilación)

Inspección manual

La inspección manual involucra personas que examinan el código fuente, para intentar descubrir defectos, y otros problemas de calidad “internos”: (ej., código no apropiadamente documentado, mal estructurado, que no respeta estándares, etc.)

Este tipo de inspección no requiere la ejecución del sistema (incluso se aplica a otras representaciones del sistema además del código: diseño, SRS, etc.) y es una técnica muy efectiva para descubrir errores, aunque difícilmente exhaustiva. Es también muy efectiva para “transmitir experiencia” entre desarrolladores.

Testing

Esencialmente, el **testing** consiste en comprobar el comportamiento de los programas en un conjunto de situaciones particulares (ej., para algunas entradas particulares).

Puede revelar la presencia de errores pero rara vez puede garantizar su ausencia.

Es la técnica de verificación funcional por excelencia pero se recomienda usarla en combinación con otras técnicas de V&V estáticas (e.g., revisión de código) para dar mejores resultados; y fundamentalmente debe realizarse de manera planificada.

¿Qué es testing?

Testing **NO** es:

- Depurar código
- Verificar que las funciones del software se implementen
- Demostrar que no hay defectos

Testing **SI** es:

un proceso destructivo que trata de encontrar defectos, poniendo al tester en una posición negativa para demostrar que algo es incorrecto. El testing detecta fallas pero no ahonda en los motivos de ellas, el objetivo es encontrarlas y una vez detectadas, si existe la necesidad de corregirlas es el desarrollador el que estará encargado de descubrir su origen y repararlas.

Algunas definiciones

- Es el proceso de ejecutar un programa o sistema con la intención de encontrar defectos. *(The art of software testing, Glenford Myers)*
- El proceso de analizar un ítem de software para detectar la diferencia entre las condiciones existentes y las condiciones requeridas y evaluar las características de los ítems del software. *(IEEE 829-1998 - Standard for Software Test Documentation)*
- El proceso de ejecutar un sistema o componente bajo condiciones específicas, observando o registrando sus resultados, y evaluando algún aspecto del sistema o componente. *(ANSI – 1990 – Std 610.12)*
- El testing es un proceso de ingeniería, un ciclo de vida concurrente que usa y mantiene el testware con el fin de medir la calidad del software bajo prueba. *(Systematic Software Testing, Rick Craig y Stefan Jaskiel)*

Misión del testing

La misión principal del proceso de testing es la de **generar información**.

La principal razón por la que los testers existen es para proveer información que pueda ser utilizada por otros y éstos la utilicen para generar cosas de valor. *(James Bach)*

El testing es una forma de medir la calidad.

Principios del testing

El testing es efectivo para demostrar la presencia de defectos en un software, pero no puede asegurar ni demostrar de manera alguna la ausencia de los mismos. Puede lograr la reducción de probabilidad de existencia de defectos no descubiertos pero el hecho de NO encontrar defectos en un software implica que esa no es una prueba de corrección.

El testing no es exhaustivo

El testing exhaustivo implica la prueba de todos los caminos posibles de ejecución de un software.

Supone ejecutar realmente todas las posibilidades de entradas y salidas de un sistema y sus diferentes combinaciones, lo que aseguraría una cobertura total y completa del funcionamiento y podría asegurar al 100% la ausencia de errores.

Esto es, claramente, imposible en la práctica debido a que la capacidad de procesamiento de cualquier computador es limitada, aún siendo que se trabaja a un nivel de rendimiento altísimo esta técnica llevaría, con seguridad, a una ejecución de tiempos infinitos.

Testing temprano

Las actividades de testing deberían iniciarse lo antes posible, refiriéndose a esta tarea como **testing temprano**.

Esto implica que comenzar a pensar en cómo vamos a probar un software, incluso cuando este todavía no ha sido desarrollado, tiene un sentido fuerte. Al empezar a definir pruebas y buscar formas de destruir la confiabilidad de lo que se está por construir nos permite tener una seguridad y una cobertura previa y predecible que no existiría si iniciamos el testing sobre el final del proceso.

Tiene como objetivo claro dar visibilidad de manera temprana al área de testing de cómo se va a probar lo desarrollado y disminuir los costos por correcciones de defectos.

Existen algunos requerimientos para realizar el testing temprano de manera eficiente:

- Definición de la estrategia y alcances de la prueba
- Organización del equipo de prueba

- Capacitación del equipo
- Establecimiento del esquema de reportes
- Provisión de recursos de hardware y de software

El testing debe validar al cliente

Además de la capacidad de encontrar y reparar defectos, el software probado debe satisfacer las necesidades y las expectativas del usuario para poder referirnos al concepto de **calidad**. El testing temprano ayuda también a esto ya que ataca la naturaleza del problema a resolver desde una perspectiva distinta a la del desarrollo, del diseño y del análisis

Contexto

La dependencia del contexto a la hora de plantear una prueba es fundamental, su proceso es diferente cuando varía este contexto.

Por ejemplo, a la hora de probar un sistema médico crítico no tendremos las mismas consideraciones que al probar un sistema de comercio electrónico. Los impactos de los errores son considerablemente distintos, pero no sólo eso, incluso la infraestructura, el proceso y la capacidad de quien realiza la prueba varía según el contexto.

Si probamos una aplicación de Televisión Digital nos encontramos con un conjunto de actores interventores (tanto humanos como tecnológicos) en el proceso mucho más amplio que a la hora de efectuar el test sobre una terminal magnética de boletos de transporte.

Esto nos exige una consideración especial del contexto y un análisis particular en cada caso a la hora de realizar una prueba de software.

Testing e Inspecciones

Las inspecciones y el testing son complementarias, ambas deberían usarse como parte del proceso de V&V para lograr una mejor medición y poder garantizar la calidad.

Las inspecciones suelen ser más efectivas en verificación (contrastación con la especificación), pero no en validación (contrastación con los requisitos reales del cliente). Éstas tienen limitaciones para poder chequear características no funcionales (performance, usabilidad, etc.).

El testing es parte explícita de procesos de desarrollo.

Testing en procesos de desarrollo

A lo largo de la historia del desarrollo tecnológico, y en particular del software, se han generado diversas maneras de llegar a la concreción de proyectos.

Mientras se diseminaba la novedad de la informática y exigía cada vez mayores esfuerzos de trabajo por su crecimiento e inserción en el mundo, aparecieron formalizaciones en las maneras de producir o crear software para enfrentar las demandas y problemas.

Así hacen aparición las metodologías o procesos de desarrollo de Software, buscando organizar y mejorar las prácticas y los resultados.

Si bien estos procesos han dado una significativa mejora y un marco dentro del cual surgieron conceptos y pudieron adoptarse otros, el desarrollo de software sigue teniendo sus matices, con procesos y herramientas diversos que continúan también un camino propio.

El **testing** en particular como parte de la medición de calidad cumple un rol específico en cada uno de estos procesos, pero siendo aplicado de diversas maneras, a diferentes momentos y con variadas consideraciones.

Procesos de desarrollo clásicos

Los modelos de proceso de desarrollo clásicos involucran generalmente las siguientes actividades:

- Fase de requisitos
- Fase de especificación
- Fase de planeamiento
- Fase de diseño
- Fase de implementación
- Integración y Testing
- Mantenimiento

El testing es una actividad evidente de V&V, posterior a implementación.

El problema aquí recae en los costos y complicaciones que generan estas metodologías. Si encontramos un error en la fase de codificación y es realmente un error de especificación de requisitos, eso implica que se han surcado todas las fases arrastrando un error y el único momento de encontrarlo y repararlos es sobre la etapa final de testing, generando una carga de re-trabajo más que considerable, aún que se esté trabajando con un modelo iterativo. Caso similar sería el de las modificaciones que pueden llegar a surgir en la especificación, luego de que esta etapa haya sido cerrada, dando como resultado avances sin validaciones y pérdidas de tiempo y dinero.

Procesos de desarrollo ágiles

Los “ágiles” son métodos de ingeniería del software basados en el desarrollo iterativo e incremental, donde los requerimientos y soluciones evolucionan mediante la colaboración de grupos auto

organizados y multidisciplinarios.

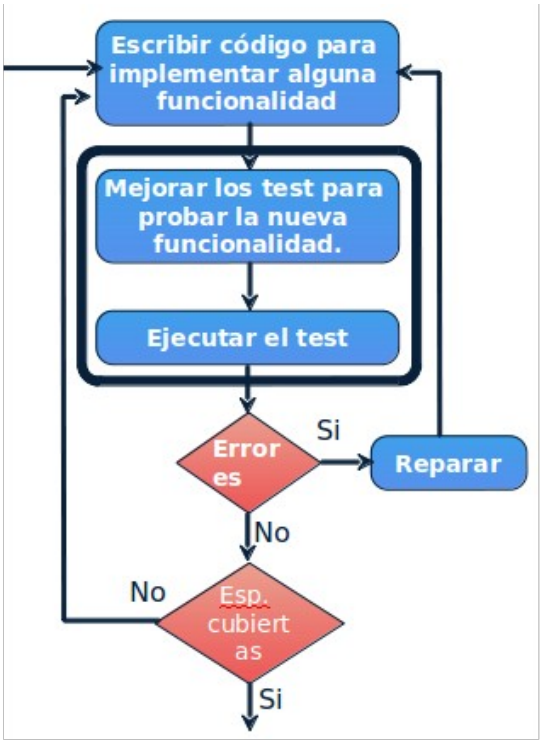
Existen muchos métodos de desarrollo ágil; la mayoría minimiza riesgos desarrollando software en lapsos cortos. El software desarrollado en una iteración, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Éstas no deben agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, sino que la meta es tener una demo (o producto entregable, sin errores) al final de cada iteración. Luego de eso el equipo vuelve a evaluar las prioridades del proyecto.

Los métodos ágiles enfatizan las comunicaciones cara a cara en vez de la documentación. La mayoría de los equipos ágiles están localizados en una simple oficina abierta. Se define al software funcional como la primer medida del progreso.

Combinado con la preferencia por las comunicaciones cara a cara, generalmente los métodos ágiles son criticados y tratados como "indisciplinados" por la falta de documentación técnica.

Dentro de estas metodologías el testing está presente en varias fases del proceso de desarrollo, principalmente como parte de la codificación. Se realiza el testing como parte de la implementación de cada funcionalidad.

Gráficamente:



Proceso de desarrollo guiádo por test (TDD)

De las siglas en inglés *Testing Driven Development*, esta metodología plantea centrar el desarrollo en las pruebas. Y no sólo

eso, si no que pretende crear y ejecutar primero las pruebas antes que el código que ellas probarán.

¿Tiene esto sentido?

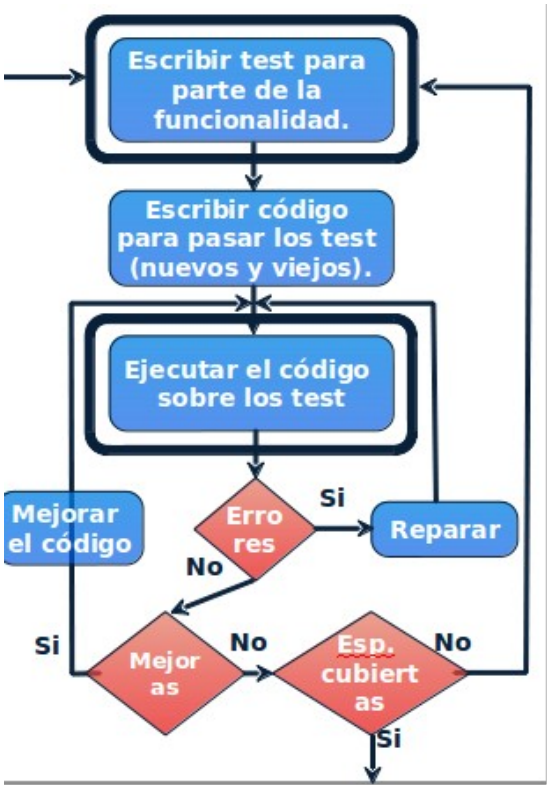
A un primer vistazo parece no tenerlo, pero una vez incorporada la metodología y practicada, cobra importante relevancia.

La intención es generar primero un requerimiento, del que se parte para generar una serie de casos de prueba, los suficientes como para asegurarnos una cobertura plena de lo que las funcionalidades deberían responder y abarcar. Luego de esto, el desarrollador toma un requerimiento, selecciona los casos de prueba correspondientes y comienza a escribir su prueba unitaria, la porción de código que se ejecutará automáticamente para probar el código funcional del sistema, desde lo más general hacia lo más particular. Paso siguiente es ejecutar esa prueba unitaria y, como es de esperarse, fallará al no tener una implementación del código. Esa falla, nos dictamina el *pequeño paso* que debemos dar en codificación, guiando al desarrollo a través de la prueba. Se construye la solución, también general, para que esa prueba sea exitosa y se vuelve a correr el test, verificando su éxito. Es este el momento de la refactorización, que será la que determine el nivel de cobertura que se tendrá sobre las pruebas y el nivel de reutilización que pueda tener lo que estamos construyendo.

Resumiendo los pasos:

1. Elegir un requisitos
2. Escribir una prueba
3. Verificar que la prueba falla
4. Escribir la implementación
5. Ejecutar las pruebas automatizadas
6. Eliminación de la duplicación o refactorización
7. Actualizar la lista de requerimientos

Gráficamente:



Terminología básica

Un **error** genera ningún, uno o más **defectos** que generan ninguna, una o más **fallas**.

- **Error:** Un error es una acción humana que produce un resultado incorrecto. Éste puede derivar en uno o más defectos y puede ser introducido en cualquier momento del ciclo de desarrollo.
- **Defecto (bug, fault)**: Es un desperfecto en un componente o sistema y al ejecutarse puede causar una falla en la función requerida. Puede ocurrir también que exista un defecto pero que no produzca una falla.
- **Falla:** Es una diferencia entre los resultados esperados y los reales, implica que se ha generado una desviación en la respuesta, servicio o resultado esperado de un componente o sistema. Esto ocurre cuando un programa no se comporta adecuadamente.

Existen una serie de normas y estándares para la nomenclatura del Testing y la Calidad de Software:

- **BS 7925-1:98 -** Glosario de términos de testing de software, basado principalmente en terminología de testing de componentes.
- **IEEE 1008:1993 –** Estándar para testing unitario de software.

- **IEEE 829:1998 –** Estándar para la documentación de pruebas de software.
- **IEEE 610.12:1990 –** Glosario de terminología de ingeniería de software.

Actualmente la **ISQTB** (International Software Testing Qualifications Board – Junta Internacional de Calificación de Prueba de Software) trabaja en la unificación de los términos usados en testing de software, habiendo liberado un Glosario de términos en su versión 2.2 liberada en Octubre de 2012.

Ejemplo de error, defecto y falla

A fines del ejemplo vamos a plantear primeramente el requerimiento y el caso de prueba que de inicio a la prueba:

- La función **doblar(int param)** devuelve el doble del valor pasado por parámetro cada vez que se ejecuta.
- El caso de prueba será **doblar(3)** y deberá retornar el valorsalto flujo de texto **6** para que sea exitosa.
- Planteamos el método:

```
int doblar(int param)
{
    int res;
    res = param * param;
    return(res);
}
```

- La llamada al método **doblar(3)** retorna el valor **9**
- El resultado representa una **FALLA**
- Esta falla se debe a un **DEFECTO** en la línea 3 del código
- El **ERROR** es tipográfico

Niveles de testing

Según el nivel de abstracción de las pruebas, encontramos que ellas pueden clasificarse de la siguiente forma:

Pruebas de componentes

Éstas pruebas son las encargadas de realizar el test sobre componentes de software de manera individual o independiente, buscando asegurar que el componente funciona como está especificado. Son conocidas también como “Pruebas unitarias”

Llamamos *componente* a un ítem de software para el cual existe una especificación.

Para este tipo de pruebas normalmente se utilizan frameworks

automatizados del tipo *Xunit*.

Pruebas de integración

Las pruebas de integración procuran exponer defectos entre las interfaces de los componentes y en su interacción. Implica un nivel "superior" de prueba, haciendo que, una vez probados los componentes individuales, se verifiquen las interacciones entre ellos e intenten detectar fallas que las pruebas de componentes no han podido encontrar.

Pruebas de sistema

Las pruebas de sistema verifican que el sistema integrado cumple con los requerimientos especificados. Intentarán detectar fallas de funcionalidad o desvío de requerimientos una vez las pruebas de componentes y de integración hayan alcanzado un nivel de madurez aceptable para este punto.

Pruebas de aceptación

Estas pruebas aseguran que el sistema cumpla con los requerimientos de "negocio" y, consecuentemente, que la lógica del mismo funcione correctamente.

Buenas prácticas para realizar estas pruebas son:

- Realizar una planificación
- Involucrar al usuario desde un comienzo de las pruebas
- Realizar versiones de Alpha y Beta testing

Utilizar contratos de aceptación

Testing unitario

El nivel de prueba de componente se relaciona directamente con el testing unitario, el cual es una manera de ejecutar pruebas de código a partir de generación de otra porción de código que ejecuta y guía esa prueba.

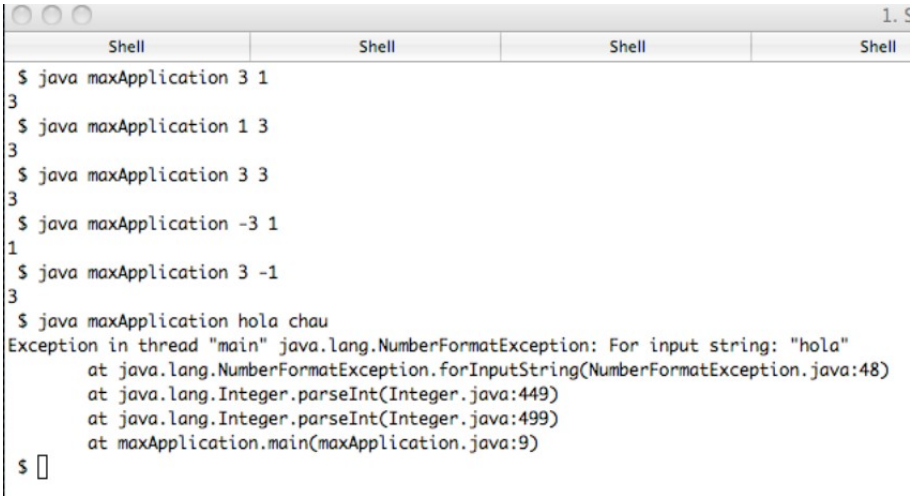
Testing ad-hoc

Todo programador en sus tareas cotidianas está habituado al testing, ya sea de manera consciente o inconsciente. En muchos casos la manera en la que llevamos adelante el testing es "ad hoc", es decir, no sistemática.

Supongamos que queremos testear la siguiente función de código:

```
public class maxApplication {

   public static void main(String[] args) {
          if (args.length != 2) {
               System.out.println("uso: maxApplication <int> <int>");
          }
          else {
               int a = Integer.parseInt(args[0]);
               int b = Integer.parseInt(args[1]);
               if (a>b) {
                    System.out.println(a);
               }
               else {
                    System.out.println(b);
               }
          }
   }
}
```

En este caso la aplicación puede ser testeada directamente desde la línea de comandos por su naturaleza:



Pero en el caso de que lo que tengamos no sea una aplicación en su totalidad, sino una funcionalidad interna o un método de cálculo y/o procesamiento, como por ejemplo:

```
public class SampleStaticRoutines {

   public static int max(int a, int b) {
          if (a>b) {
               return a;
          }
          else {
               return b;
          }
   }

}
```

El testing “ad hoc” se vuelve más difícil, tenemos que programar un arnés para la rutina (ej., un “main” que la invoque).

Dificultades

El testing “ad hoc” es bien simple para testear aplicaciones rápidamente, pero no almacena los tests para poder correrlos nuevamente en pruebas futuras, estos se pierden cada vez que se ejecutan.

Requiere la construcción manual de “arneses” para la prueba de funcionalidades “internas” (no directamente invocables desde la interfaz de la aplicación).

Requiere la inspección humana de la salida de los tests, decidiendo “manualmente” si el test pasó o no.

Sistematización del testing

Existen maneras de realizar una sistematización de ciertas modalidades de testing para abandonar algunas tareas manuales y avanzar en la simplificación e ingeniería de la etapa de pruebas.

Las librerías de apoyo al testing ayudan a sistematizar parte de las tareas manuales mencionadas:

- Permiten almacenar los tests como “scripts”,
- Definen un esquema estándar para scripts de tests,
- Permiten definir la salida esperada como parte del script,
- Construyen automáticamente arneses para la ejecución de los tests.

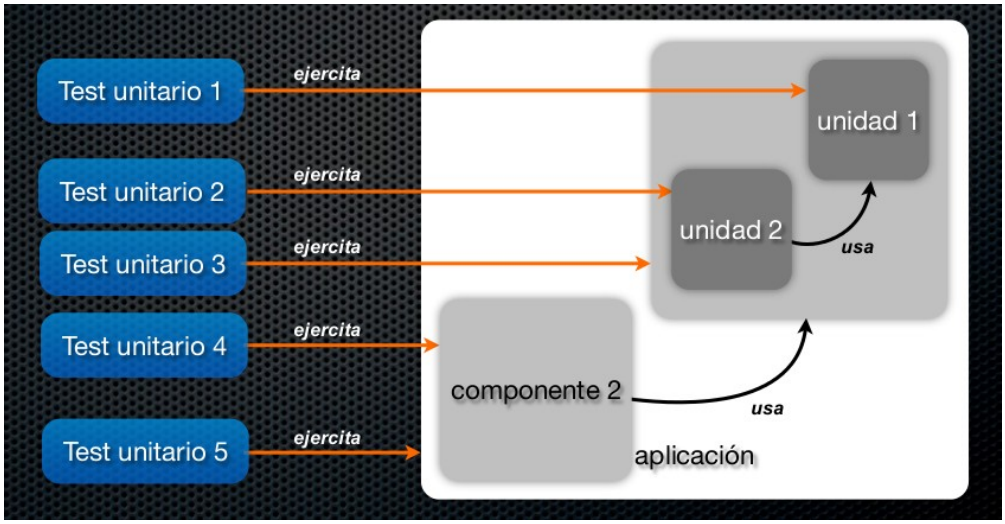
Testing unitario

Es una metodología para testear unidades individuales de código (**SUT: Software under test**), preferentemente de forma aislada de sus dependencias (**DOC: Depend-on component**).

Los tests unitarios usualmente son creados por los propios programadores durante el proceso de codificación.

El uso de esta metodología:

- Facilita los cambios,
- Simplifica la integración de componentes
- Sirve como documentación de código
- Contribuye al diseño del sistema
- Normalmente las unidades son las partes mas pequeñas de un sistema: funciones o procedimientos, clases, etc. pero la granularidad es variable.



Testing unitario con Junit

JUnit es una librería de apoyo al testing unitario para Java:

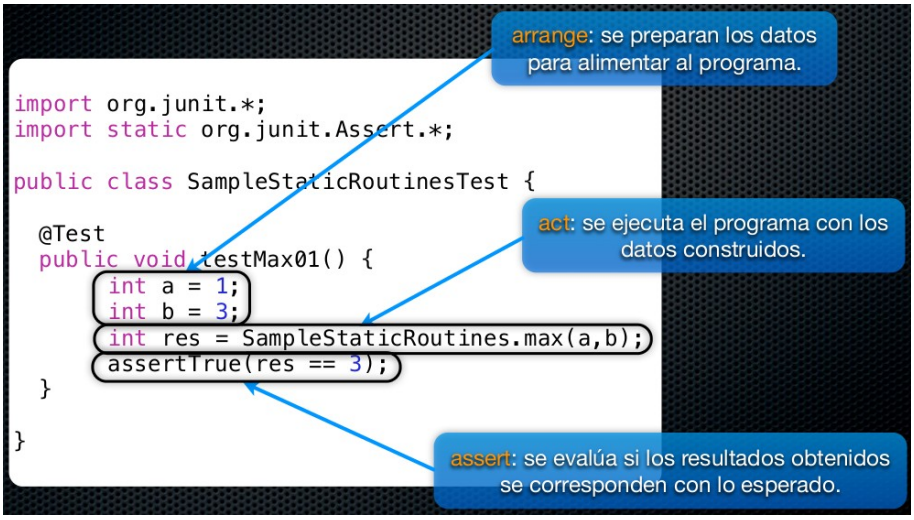
- Define la estructura básica de un test.
- Permite organizar tests en suites.
- Ofrece entornos para la ejecución de tests y suites.
- Reporta información detallada sobre las pruebas, en especial sobre las fallas.

JUnit es un conjunto de clases (*framework*) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase

cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

Estructura

Un test unitario desarrollado para JUnit responde a una estructura particular en la cual tendremos una inicialización de los datos que alimentarán al programa (arrange), otro donde se ejecutará el programa con estos datos de inicialización (act) y finalmente la contrastación de lo calculado contra lo esperado (assert)



Aserciones

Las aserciones o Asserts son las funciones propias ofrecidas por el framework de test unitario que permiten evaluar y contrastar los datos generados en el test contra los resultados esperados y así registrar la información de esa evaluación.

Algunas aserciones JUnit son:

- **assertTrue**(<expr>): verifica que <expr> evalúe a true.
- **assertFalse**(<expr>): verifica que <expr> evalúe a false.
- **assertEquals**(<expr1>, <expr2>): verifica que <expr1> y <expr2> evalúen al mismo valor.
- **assertArrayEquals**(<array1>, <array2>): verifica que <array1> y <array2> sean iguales, elemento a elemento.
- **assertNotNull**(<object>): verifica que <object> no sea null.
- **assertNull**(<object>): verifica que <object> sea null.
- **assertNotSame**(<object1>,<object2>): verifica que <object1> y <object2> no sean el mismo objeto.

Test "negativos"

Los tests negativos son aquellos en los que probamos que el SUT falla al ejecutar cierta porción de código bajo ciertas entradas.

En JUnit, los tests negativos los podemos capturar indicando que se espera que cierto test lance una excepción:

```
@Test(expected=IOException.class)
public void yourTestMethod() throws Exception {

   throw new IOException();
}
```

Web: http://junit.org/

Suites de test

Cuando el SUT está compuesto por varias clases, o simplemente varios métodos de una clase, resulta claro que debemos organizar los tests.

Los tests se organizan en test suites:

- Una test suite es simplemente un conjunto de tests.
- En JUnit, todo conjunto de tests definidos en una misma clase/archivo es una test suite.

Las suites de test son un conjunto de varios casos de prueba para un componente o sistema bajo prueba. Se usa para agrupar casos de prueba similares.

Suites de test y datos compartidos

En muchos casos, los tests de una suite comparten los datos que manipulan. En estos contextos, suele ser útil organizar los tests definiendo procesos comunes de fixture (o arrangement, o set up) para todos los tests.

Similarmente, se pueden definir procesos comunes de destrucción (o tear down), que se ejecuten luego de cada test.

Ejemplo

Consideremos un ejemplo de testing de una clase Minefield, que representa el estado de un campo minado en el juego "Buscaminas":

```
public class Minefield {

   private Mine[][] field;

   ...

   public int minedNeighbours(int x, int y) {
    ...
    }
}

public class Mine {

   private boolean isMined;
   private boolean isMarked;
   private boolean isOpened;

    ...
}
```

Para poder testear minedNeighbours, debemos crear el minefield, y ubicar minas en lugares específicos.

El proceso setUp en el siguiente ejemplo crea el escenario adecuado para la ejecución de tests de minedNeighbours. Se ejecuta antes de cada test.

```
public class MinefieldTest {

   private Minefield testingField;

   @Before
   public void setUp() throws Exception {
        if (testingField == null) {
            testingField = new Minefield();
        }
        for (int i=0; i<8; i++) {
            for (int j=0; j<8; j++) {
                testingField.removeMine(i, j);
                testingField.unmark(i, j);
                testingField.close(i, j);
            }
        }
        testingField.putMine(0, 0);
        testingField.putMine(3, 4);
        testingField.putMine(4, 3);
        testingField.putMine(2, 2);
        testingField.putMine(0, 7);
        testingField.putMine(7, 7);
        testingField.putMine(5, 1);
        testingField.putMine(4, 7);
   }

    ...
}
```

```
@Test
public void testNoOfMinedNeighbours01() {
   int number = testingField.minedNeighbours(1, 0);
   org.junit.Assert.assertTrue(number == 1);
}

@Test
public void testNoOfMinedNeighbours02() {
   int number = testingField.minedNeighbours(7, 7);
   org.junit.Assert.assertTrue(number == 0);
}
```

Sobre setUp y tearDown

Es muy importante que no haya dependencias entre tests diferentes para poder mantener conceptualmente las pruebas a nivel de unidad y concentrar esa validación sobre esa atomicidad

de la prueba.

Ya que no hay ninguna garantía sobre el orden en el que se ejecutan los tests , las rutinas setUp (@Before) y tearDown (@After) ayudan a garantizar la ausencia de dependencias entre diferentes tests.

El método anotado con @Before se ejecuta antes de cada test de la suite. El método anotado con @After se ejecuta después de cada test de la suite, aún si el test lanza excepciones.

La importancia del setUp global, es decir, anterior a cada uno de los tests, es que mantiene la independencia entre estos.

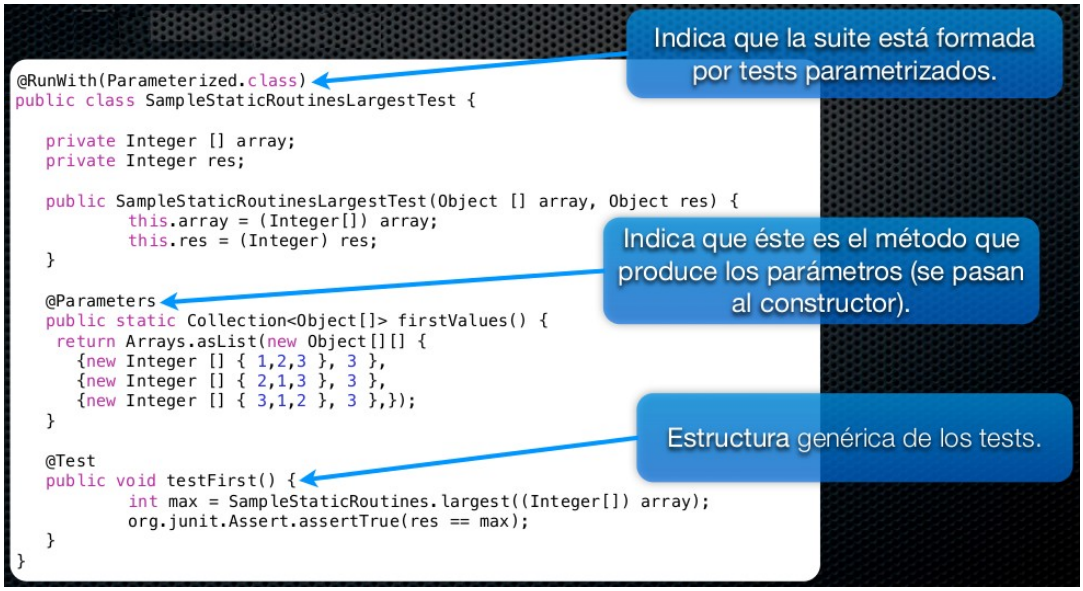
Test parametrizados por datos

Ya se han tratado casos en los cuales varios tests poseen exactamente la misma estructura, pero difieren en los datos que se utilizan para realizar el arrange del test.

Para estos casos, JUnit ofrece una forma conveniente de organizar los tests, separando su estructura de los datos que manipulan. La clave es:

- Definir datos para la operación de los tests como atributos de clase.
- Definir un generador de parámetros, que se usará para instanciar los datos para los tests.

Consideremos el método largest, que calcula el máximo de un arreglo. El siguiente test parametrizado lo prueba con varios datos diferentes:




```
@RunWith(Parameterized.class)
public class SampleStaticRoutinesLargestTest {

   private Integer [] array;
   private Integer res;

   public SampleStaticRoutinesLargestTest(Object [] array, Object res) {
           this.array = (Integer[]) array;
           this.res = (Integer) res;
   }

   @Parameters
   public static Collection<Object[]> firstValues() {
    return Arrays.asList(new Object[][] {
      {new Integer [] { 1,2,3 }, 3 },
      {new Integer [] { 2,1,3 }, 3 },
      {new Integer [] { 3,1,2 }, 3 },});
   }

   @Test
   public void testFirst() {
           int max = SampleStaticRoutines.largest((Integer[]) array);
           org.junit.Assert.assertTrue(res == max);
   }
}
```

Día 2

El día dos de curso estaremos centrados en la conceptualización y práctica de las técnicas de testing unitario más específicas, realizando ejercitaciones particulares de cada una de ellas para lograr absorber el contenido y la metodología de trabajo que exigen.

Dobles de pruebas

Normalmente el funcionamiento del SUT depende de otros componentes, por dos vías:

- Entrada indirecta: es un valor obtenido por invocaciones a un método de un DOC.
- Salida indirecta: es una potencial modificación al estado de un DOC.

Un doble de prueba reemplaza un DOC, aislando el SUT cuando:

- Es necesario controlar las entradas indirectas, para manejar el hilo de ejecución que se desea ejercitar.
- Es necesario monitorear las salidas indirectas, que son consecuencia del funcionamiento del SUT.

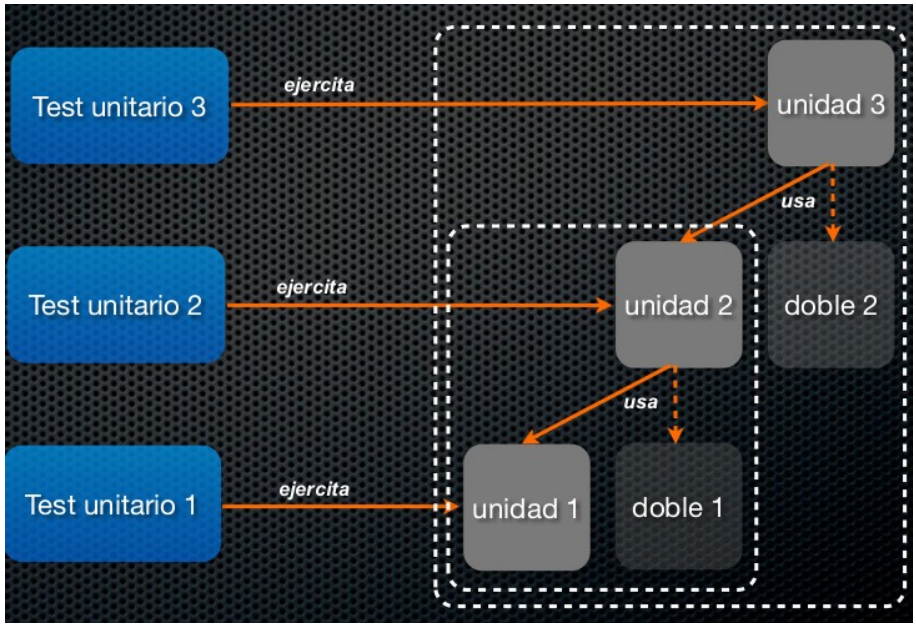
Así tenemos que los dobles de prueba son objetos controlados que reemplazan a un componente externo que genera una dependencia frente a otro en lo que a test respecta.

Estrategias de integración

Se identifican dos estrategias distintas de integración

Adentro hacia afuera

Tradicionalmente un proyecto Sofware se integra de esta manera.

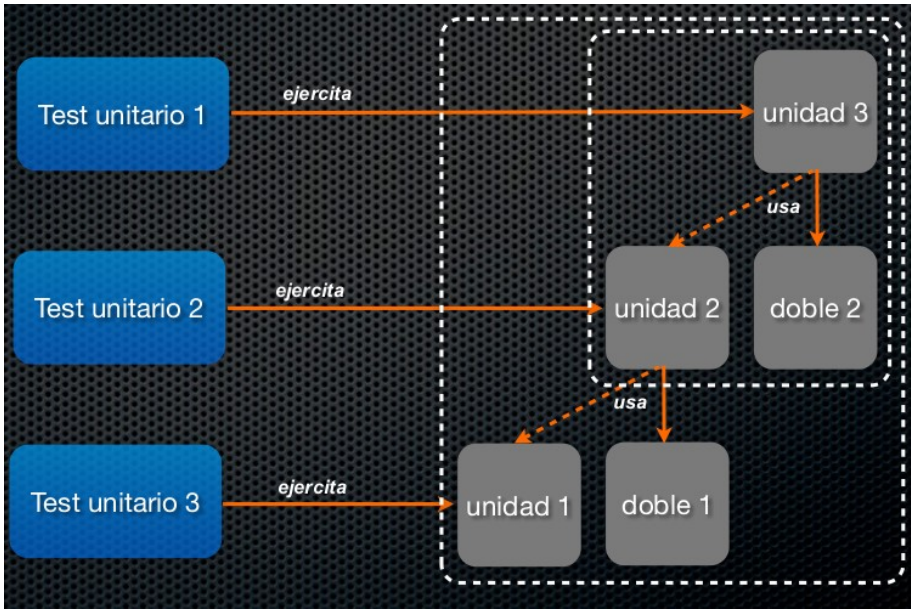


En la integración adentro hacia afuera existen las siguientes consideraciones:

- La codificación (y el testing) comienza en los componentes más internos y simples, y prosigue hacia los mas complejos.
- Requiere un diseño y arquitectura mas detallado, y requerimientos bien especificados.
- Responde a un proceso de desarrollo tradicional.
- Minimiza la necesidad de dobles, pero minimiza el feedback temprano.
- Cambios imprevistos en los requerimientos por la falta de feedback repercuten en el re-trabajo sobre el testing.

Afuera hacia adentro

El testing unitario se acomoda bien con la estrategia afuera hacia adentro:

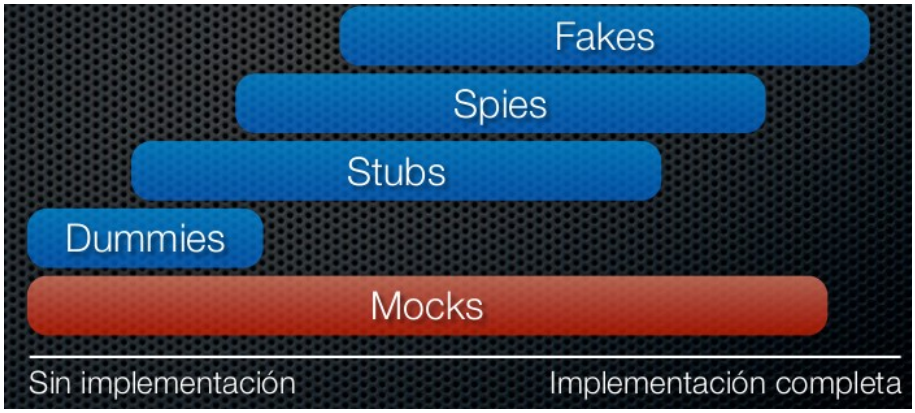


En la integración afuera hacia adentro existen las siguientes consideraciones:

- La codificación (y testing) comienza en los componentes más generales: prototipado rápido con feedback temprano.
- Los dobles se van reemplazando a medida que se avanza en la codificación de los componentes.
- La arquitectura y diseño se van (re)definiendo “en la marcha”.
- Responde a un proceso de desarrollo ágil.
- Maximiza la necesidad de dobles y el feedback, pero minimiza el re-trabajo por cambios en los requerimientos.

Tipos de dobles

Aunque los tipos parecen diferentes en teoría, en la práctica las diferencias se vuelven borrosas. Parece mas apropiado, pensar los dobles como miembros de un continuo:



Dummy

Es el más simple y primitivo. Son derivaciones de interfaces que no contienen ninguna implementación y se utilizan normalmente como valores para parámetros que nunca se utilizan.

Ejemplo

Queremos probar la clase *Order* y *OrderLine* que dependen de la interfaz *IshopDataAccess*:

```
public interface IShopDataAccess {
    double getProductPrice(int productId);
    void save(int orderId, Order o);
}
```

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;

    public OrderLine(Order owner) {
        if (owner == null)
            throw new ArgumentNullException("owner");
        this.owner = owner;
    }

    public double getTotal() {
        double unitPrice =
            owner.getDataAccess().getProductPrice(id);
        double total = unitPrice * quantity;
        return total;
    }
    ...
```

```
public class Order {

   private int id;
   private IShopDataAccess dataAccess;
   private OrderLineCollection orderLines;

   public OrderLineCollection getLines() {
       return orderLines;
   }

   public IShopDataAccess getDataAccess() {
       return dataAccess;
   }

   public void save() {
       this.dataAccess.save(this.id, this);
   }

   public Order(int id, IShopDataAccess dataAccess) {
       if (dataAccess == null)
         throw new ArgumentNullException("dataAccess");

       this.id = id;
       this.dataAccess = dataAccess;
       this.orderLines = new OrderLineCollection(this);
   }
   ...
```

Un Dummy sólo satisface dependencias formales y es suficiente siempre que la interfaz no sea ejercitada.

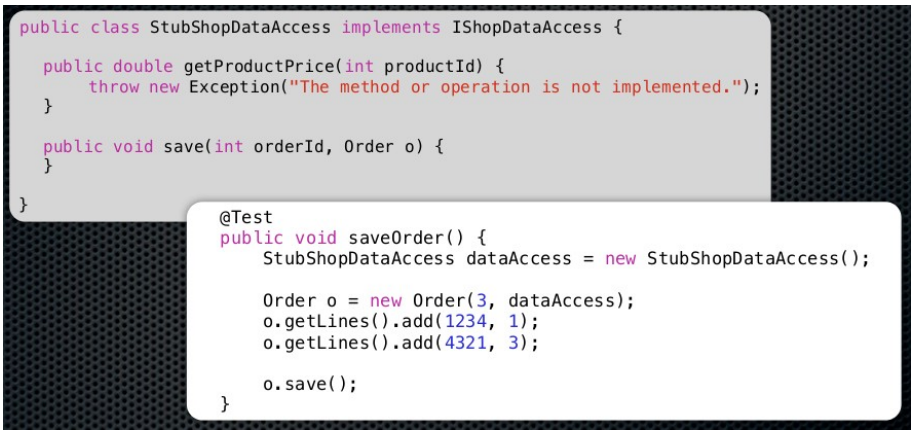
```
public class DummyShopDataAccess implements IShopDataAccess {

   public double getProductPrice(int productId) {
       throw new Exception("The method or operation is not implemented.");
   }

   public void save(int orderId, Order o) {
       throw new Exception("The method or operation is not implemented.");
   }

}
```

```
@Test
public void createOrder() {
    DummyShopDataAccess dataAccess = new DummyShopDataAccess();

    Order o = new Order(2, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    assertEquals(2, o.getLines().size());
}
```

Stub

Si el test invoca algún método, es necesario (a menos) **no** levantar una excepción, es decir, permitir su correcta ejecución hasta el final. Este caso extiende al Dummy ya que interpreta que la interfaz es ejercitada.

Lo vemos en el siguiente caso donde se llama a la ejecución del método **save()** de la clase **StubShopDataAccess**:



La diferencia en estos casos es más marcada cuando el método invocado no es meramente una ejecución externa, sino que éste devuelve un valor que nos servirá de **input indirecto** al método que estamos probando.

En estos casos la implementación más sencilla suele ser la de devolver **valores fijos** que permitan de esa manera controlar el input indirecto para verificar el comportamiento esperado dentro del test unitario.

Por ejemplo:

```
public class OrderLine {
    private int id;
    private int quantity;
    private Order owner;

    public OrderLine(Order owner) {
        if (owner == null)
            throw new ArgumentNullException("owner");
        this.owner = owner;
    }

    public double getTotal() {
        double unitPrice =
            owner.getDataAccess().getProductPrice(id);
        double total = unitPrice * quantity;
        return total;
    }
    ...
```

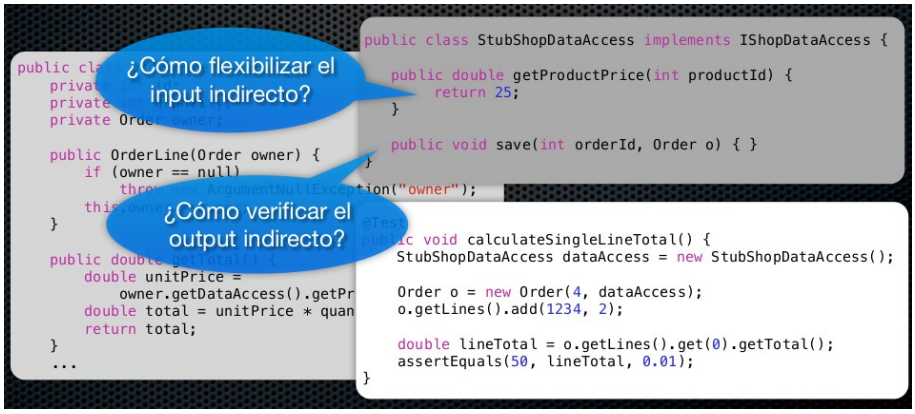
Devolvemos un valor fijo:

```
public class StubShopDataAccess implements IShopDataAccess {

   public double getProductPrice(int productId) {
        return 25;
   }

   public void save(int orderId, Order o) { }
}
```

Ejecutamos el test conociendo este valor y validando el resultado esperado, lo que nos permite conocer que la operatoria individual del Test realiza bien los cálculos, independientemente de si el método de devolución de precio es correcto (por estar respondiendo en modo Stub):

```
@Test
public void calculateSingleLineTotal() {
    StubShopDataAccess dataAccess = new StubShopDataAccess();

    Order o = new Order(4, dataAccess);
    o.getLines().add(1234, 2);

    double lineTotal = o.getLines().get(0).getTotal();
    assertEquals(50, lineTotal, 0.01);
}
```

De allí surge lo siguiente:



Spy

El doble Spy es utilizado para verificar el output indirecto de algún método que deba ser ejecutado desde la unidad a probar. Simplemente nos indicará si al realizar ciertos pasos se alcanzó la ejecución o no de dicho output indirecto, sabiendo que dichos pasos contemplaban esta ejecución.

Verificar el output indirecto requiere registrar las invocaciones y sus parámetros.

```
public class SpyShopDataAccess implements IShopDataAccess {
    private boolean saveWasInvoked;

   ...

    public boolean getSaveWasInvoked() {
        return this.saveWasInvoked;

}
```

Ejecutando el test unitario contrastamos si realmente el método fue ejecutado:

```
@Test
public void saveOrderWithDataAccessVerification() {
    SpyShopDataAccess dataAccess = new SpyShopDataAccess();

    Order o = new Order(5, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);
    o.save();

    assertTrue(dataAccess.getSaveWasInvoked());
}
```

Fake

El doble Fake es utilizado para flexibilizar el input indirecto de algún método que inyecte valores al código a probar. Implica crear una implementación “falsa” de lo que debería hacer el método del cual depende nuestro código bajo prueba, pero operando de una manera contraria al concepto de “hardcodeo” (que implica retornar valores fijos) sino realizando operaciones válidas, las cuales pueden también estar viciadas de errores.

Flexibilizar el input indirecto implica aproximarse a una implementación de producción:

```
public class FakeShopDataAccess implements IShopDataAccess {
    private ProductCollection products;

    public FakeShopDataAccess() {
        this.products = new ProductCollection();
    }

    public double getProductPrice(int productId) {
        if (this.products.contains(productId)) {
            return this.products.get(productId).getUnitPrice();
        }
        throw new ArgumentOutOfRangeException("productId");
    }

    List<Product> getProducts() {
        return this.products;
    }

    public void save() {
        ...
    }
    ...
```

```
@Test
public void calculateLineTotalsUsingFake() {
    FakeShopDataAccess dataAccess = new FakeShopDataAccess();
    dataAccess.getProducts().add(new Product(1234, 45));
    dataAccess.getProducts().add(new Product(2345, 15));

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 3);
    o.getLines().add(2345, 2);

    assertEquals(135, o.getLines().get(0).getTotal(), 0.01);
    assertEquals(30, o.getLines().get(1).getTotal(), 0.01);
}
```

Mock

Mock es una denominación general para dobles que controlan entrada y salida indirectas. La escritura de mocks manualmente es una tarea ardua y muy propensa a introducir errores, lo que en una etapa de testing y calidad no sería la mejor solución el hecho de sumar aún más posibilidades de falla.

Para contrarrestar esto, en general, los mocks se crean en tiempo de ejecución con la ayuda de una librería específica, verificada y validada, que permite:

- En una primera fase se especifica el comportamiento esperado.
- En una segunda fase, se crea el objeto cuyos métodos serán invocados.
- En una tercera fase se verifica el comportamiento ejercitado respecto al especificado.

De esta manera no es necesario escribir el código que implementa el mock sino que simplemente se utiliza una librería para esto.

EasyMock

Una de las librerías específicas para creación “on the fly” the mocks es la conocida como **EasyMock** escrita en Java.

Esta librería provee objetos Mock para interfaces (y objetos a través de extensión de clases) generándolos “al vuelo” usando el mecanismo proxy de Java. Debido a su estilo de grabado de resultados esperados la mayoría de los refactorings que se hagan al código no afectarán a estos objetos Mock.

Es software open source disponible bajo los términos de la licencia Apache 2.0.

Los ejemplos empleados en Spy y Fake pueden ser fácilmente implementados con EasyMock:

```
@Test
public void saveOrderAndVerifyExpectations() {
    IShopDataAccess dataAccess = createMock(IShopDataAccess.class);

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    // Record expectations
    dataAccess.save(6, o);
    replay(dataAccess);

    o.save();

    verify(dataAccess);
}
```

```
@Test
public void calculateLineTotalsUsingManualMock() {
    IShopDataAccess dataAccess = createMock(IShopDataAccess.class);

    expect(dataAccess.getProductPrice(1234)).andReturn(45.0);
    expect(dataAccess.getProductPrice(2345)).andReturn(15.0);
    replay(dataAccess);

    Order o = new Order(11, dataAccess);
    o.getLines().add(1234, 3);
    o.getLines().add(2345, 2);

    assertEquals(135, o.getLines().get(0).getTotal(), 0.01);
    assertEquals(30, o.getLines().get(1).getTotal(), 0.01);

}
```

Funciones principales

- **createMock**(<class>): crea un mock que implementa la interfaz <class>, sin registra del orden de invocación.
- **createNiceMock**(<class>): crea un mock que implementa la interfaz <class>, sin orden de invocación y que devuelve 0, null o false para las invocaciones no esperadas.
- **createStrictMock**(<class>): crea un mock que implementa la interfaz <class>, con registro del orden de invocación.
- **expect**(<inv>): registra la expectativa de llamada a <inv>.
- **expect**(<inv>).**andReturn**(<val>): además establece como resultado <val>.
- **expect**(<inv>).**andThrow**(<exc>): además establece la ocurrencia de la excepción <exc>.
- **expectLastCall**.**times**(<n>): establece la expectativa de <n> llamadas a la ultima invocación registrada.
- **anyInt**( ), **anyChar**( ), **anyObject**( )...: reemplazan los parámetros de las invocaciones que se registran.
- **replay**(<mock>): establece el comportamiento especificado sobre el <mock>.
- **verify**(<mock>): verifica el comportamiento especificado sobre <mock>.
- **reset**(<mock>): resetea el comportamiento especificado sobre <mock>, útil para fixtures compartidos.

Web: http://easymock.org/

Día 3

El día de cierre del curso ahondaremos en los conceptos de criterios de cobertura y de integración continua y en la práctica de una herramienta particular que sirve de apoyo a esta última tarea.

Se dará el cierre del curso haciendo una revisión de lo estudiado y aprendido, compartiendo las miradas y los próximos pasos a dar según los casos particulares de los capacitados.

Criterios de cobertura

Un criterio de testing es un mecanismo para decidir si una suite es adecuada o no. En general, se espera que un criterio de test cumpla con:

- Regularidad: Un criterio es regular si todos los conjuntos de casos test que satisfacen el criterio detectan en general los mismos errores.
- Validez: Un criterio es válido si para cualquier error en el programa hay un conjunto de casos de test que satisfagan tal criterio y detecten el error.

En general, es muy difícil conseguir criterios con buenas características de regularidad y validez.

Conseguir criterios con buenas características de regularidad y validez suele ser una tarea compleja.

Los criterios surgen a partir de entender que la forma más efectiva de hacer testing sería a través de una ejecución de pruebas exhaustivas, pero sabemos que esto es impracticable y es allí donde los criterios dan su aporte, son necesarios para realizar la selección de las pruebas.

El proceso guiado por un criterio identifica un conjunto de pruebas (clase) que es representativo de todas las pruebas posibles, haciendo que, por ejemplo, si dos pruebas encontrasen el mismo defecto, éstas deberían pertenecer a la misma clase y el producto bajo prueba se comportará de la misma manera para todas las de la misma clase.

Generalmente la definición de criterios la haremos una vez que los casos de prueba estén identificados y escritos, para luego ser agrupados en clases que respondan a los criterios de selección considerados.

Técnicas

Existen principalmente dos ramas para definir criterios de testing:

Caja negra (funcional)

Las técnicas de caja negra están basadas en la definición de requerimientos o de una descripción funcional del producto bajo prueba y se centran en *"qué hace"* el programa y no en *"cómo lo hace"*. Es por eso que son llamadas "de caja negra", porque no observan el comportamiento interno y generalmente las pruebas se conducen a nivel de interfaz de usuario.

Como no se observa el comportamiento interno del software, es necesario contar con una descripción de qué se espera del SUT (especificación).

Para diseñar los casos de test que conforman la suite, se usa el comportamiento esperado del sistema.

Encontramos las siguientes técnicas de caja negra para la derivación de casos de prueba:

Relevancia de las especificaciones

Para hacer testing de caja negra, se debe contar con una especificación del SUT.

Es más efectivo si se cuenta con especificaciones ricas:

- A nivel de rutinas: pre- y post-condiciones.
- A nivel de módulos: especificación de diseño, contratos de clases, etc.
- A nivel de sistema: una buena especificación de requisitos.

Caso de uso

La técnica pide crear al menos un caso de prueba para el escenario exitoso y uno para cada alternativo permitiendo definir una medida de cobertura de las pruebas.

El problema es que un mismo escenario puede ser accionado con más de una combinación de entradas haciendo que muchas de las combinaciones posibles queden sin probar. Hay que prestar especial atención también al cuidado de no sobreestimar la calidad del sistema.

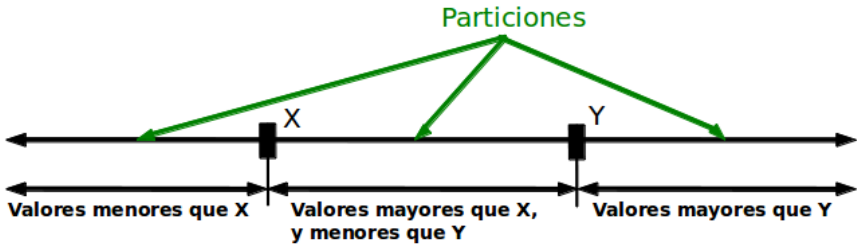
Particionando en clases de equivalencia

Esta técnica usa un modelo del componente que divide los valores de entrada y de salida en particiones de equivalencia. Esas particiones deben corresponder a casos similares, en los que el producto bajo prueba se comporta de la misma manera.

Entiende que si el producto funciona correctamente para una

prueba en una partición, se supone que lo hará para el resto de las pruebas de la misma partición.

Ejemplo de entrada numérica:



Deben considerarse las siguientes entradas:

- **Válidas:** que NO deberían ser rechazadas por el componente o sistema
- **Inválidas:** que deberían ser rechazadas por el componente o sistema

Las instrucciones para ejecutar la técnica son:

1. Identificar las particiones
    1. Si la entrada es sobre un rango de valores
        1. una partición válida para valores dentro del rango
        2. dos inválidas para cada lado fuera del rango
    2. Si la entrada es sobre un conjunto de valores
        1. una partición válida para valores dentro del conjunto
        2. una partición inválida para valores fuera del conjunto
    3. Asignar un número único a cada partición de equivalencia
2. Diseñar casos de prueba que cubran la mayor cantidad de particiones válidas
3. Diseñar un caso de prueba para cada clase inválida
    1. Si probamos múltiples clases inválidas con el mismo caso, el rechazo de una clase inválida puede enmascarar una falla en el rechazo de las otras entradas inválidas
    2. Existen dos aproximaciones:
        1. Mantener los mismos valores válidos para todas las pruebas que cubren clases inválidas
        2. Variar tanto valores válidos como inválidos

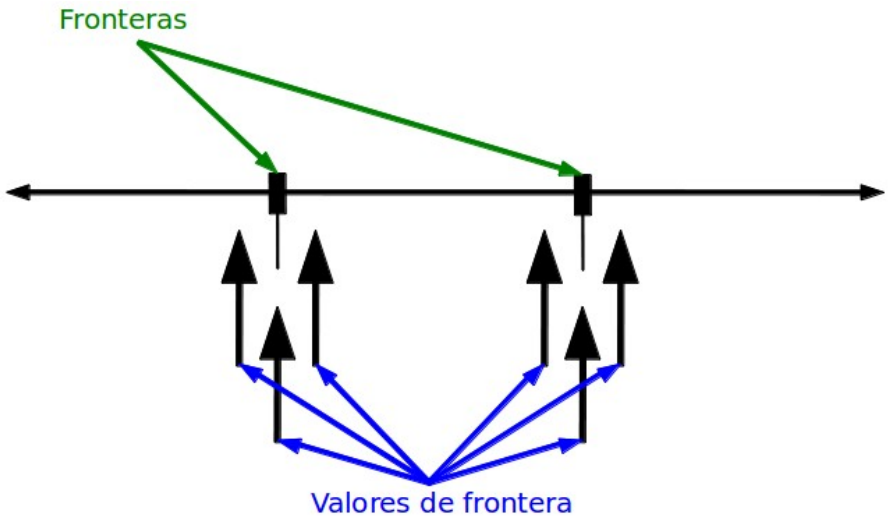
Valores frontera

Existe una frase enunciada por Boris Beizer, teórico de Software y de las ciencias de computación, que dice *“Los defectos se esconden en las esquinas y se congregan en los límites”*.

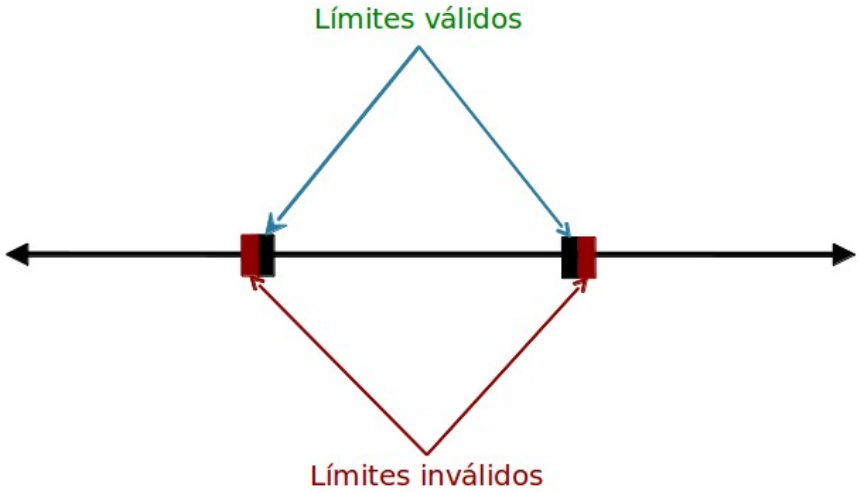
La técnica *valores de frontera* es una especialización de la de particiones de equivalencia orientada a entradas con valores ordenados.

La técnica parte de la identificación de las particiones de equivalencia y seleccionando los valores frontera de dos maneras:

1. Por cada entrada y salida con fronteras identificables se definen 3 casos de prueba por frontera:
    1. uno en la frontera
    2. uno de cada lado de la frontera con el menor incremento/decremento posible



2. Por cada entrada y salida con particiones identificables se definen 2 casos de prueba:
    1. uno para el máximo de cada partición válida o inválida
    2. uno para el mínimo de cada partición válida o inválida



La técnica permite agregar más casos de pruebas que se encuentren alejados de las fronteras, pero estos raramente encuentran más defectos.

Cuando el hecho de crear pruebas individuales para cada límite trae un problema de gasto de tiempo, se crean casos que prueban varias fronteras a la vez pero siempre buscando probar un límite inválido por vez.

Árbol de clasificación

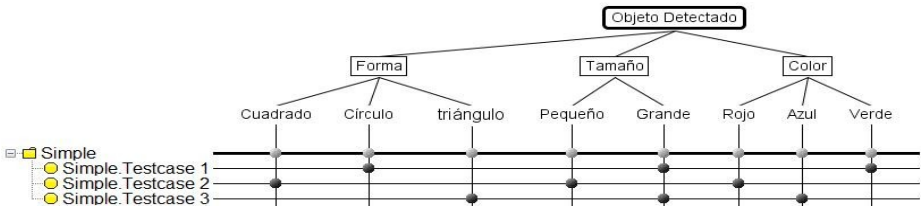
Esta técnica es también una especialización de la de particiones por equivalencia.

Los pasos de la técnica:

1. Para cada entrada se determinan los diferentes aspectos funcionales relevantes
    1. Se define una partición para cada aspecto
    2. Los aspectos se representan en forma de árbol
    3. Los casos de prueba se diseñan combinando las particiones de los diferentes aspectos

El árbol de clasificación organiza la información y permite seleccionar casos que cubran la mayor cantidad de combinaciones de los aspectos.

Por ejemplo:



Prueba de pares

Es una técnica para cubrir una cantidad significativa de combinaciones de valores (entradas, configuraciones).

Podemos pensar que un sistema a probar puede ejecutarse bajo una gran diversidad de configuraciones, con diferentes entornos, agregados, etc.

Al momento de calcular la cantidad posible de combinaciones podemos encontrarnos con que el número hace que sea complicado hacer un recorrido por todas ellas, por lo que se plantea el enfoque de probar todas las combinaciones de valores sólo entre dos variables.

Existen al menos dos métodos para seleccionar situaciones con todas las combinaciones de a pares:

- Arreglos ortogonales

- Arreglos de pares

Son métodos complejos pero existen algunas herramientas automáticas para computar las situaciones que facilitan la tarea.

Por ejemplo, supongamos que nuestro sistema a testear es web y podemos encontrar la siguiente variabilidad de entornos y configuraciones:

- 6 navegadores web
- 3 plug-ins necesarios para que el sistema funcione correctamente
- 7 diferentes Sistemas Operativos clientes que accederán a él
- 3 distintos servidores web en los que podría correr
- 3 distintos Sistemas Operativos que pueden alojar a los servidores
- Total: **1134 combinaciones posibles**

Sin embargo, esta técnica nos dice que es posible encontrar muchas de las posibles fallas con sólo 42 combinaciones, probando todas las combinaciones sólo entre 2 variables.

Manejo de datos

En los sistemas **ABML** (Alta, Baja, Modificación y Lectura) los datos tienen un ciclo de vida dentro de la aplicación que:

- Comienza cuando la entidad es creada (Alta)
- Durante su ciclo de vida, es modificada (Modificación) o leída (Lectura)
- Termina cuando es destruida (Baja)

Esta técnica se ejecuta de la siguiente manera:

1. Se crea una matriz ABML
    1. Por cada función del sistema determinamos:
        1. entidades usada en la función
        2. acciones (A, B, M o L) llevadas a cabo por esas entidades
    2. El resultado es registrado en la matriz
        1. una columna por cada entidad
        2. una fila por cada función
        3. las acciones de una función sobre una entidad específica en la celda correspondiente
2. Probar completitud (prueba estática):
    1. Examinar si por cada entidad se pueden realizar las 4 acciones
    2. Que se encuentren incompletas no implica un error,

pero si efectúa un llamado de atención

1. Entidades no creables no deberían ser destruibles y raramente modificables
2. Entidades no destruibles implican un aumento constante del volumen de datos
3. Entidades no modificables implican imposibilidades de corregir errores de tipo
4. Entidades no legibles suelen no tener sentido

3. Probar consistencia (prueba dinámica);
    1. Verificar que las funciones usen consistentemente una entidad
        1. Cada prueba comienza con una A, sigue con todas las posibles M y finaliza con una B
        2. Luego de cada acción (A, B o M), se realiza la L correspondiente, para verificar que la entidad fue correctamente procesada
        3. Para cada entidad relevante, todas las ocurrencias de las acciones (A, B, M o L) en todas las funciones deben ser cubiertas
    2. Variante más intensiva: realizar todas las L luego de un A, B o M.

Tabla de decisión

Las tablas de decisión como técnica son utilizadas para especificar las reglas de negocio de un sistema, las entradas y las posibles respuestas que el sistema tendrá frente a ellas.

Estas tablas consisten en:

- Una serie de condiciones de entrada
- Una serie de acciones a ejecutar y sus características
- Un conjunto de reglas que definen qué acciones se ejecutan para cada combinación de condiciones de entrada

La técnica se desarrolla de la siguiente manera:

1. Construir la tabla en función de los requisitos
2. Diseñar casos de prueba en función del tipo de condición:
    1. Condición binaria: un caso de prueba por cada valor
    2. Rango de valores: aplicar la técnica de partición de equivalencia o, mejor aún, de análisis de frontera

Una columna colapsada debe tratarse como dos columnas separadas.

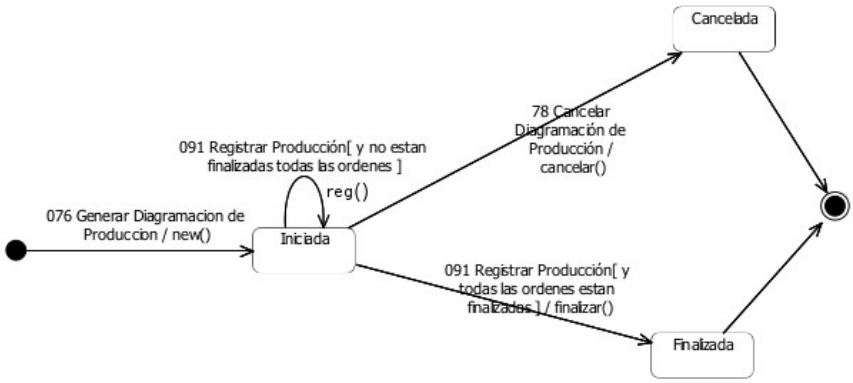
Diagrama de transición

Esta técnica es una manera de modelar programas reactivos, contando con los siguientes componentes:

- **Estado:** condición distinguible del sistema que persiste por un período de tiempo significativo
- **Evento:** acontecimiento, interno o externo, reconocible por el sistema
  - Pueden ser señales, invocaciones, paso del tiempo, etc.
  - En cada estado los eventos no explicitados son considerados inválidos para ese estado
- **Transición:** Relación entre estados que representa el paso de uno hacia el otro
  - El estado resultante puede ser el estado de partida
  - Son disparadas por eventos
  - Su ejecución puede estar restringida por condiciones
  - Su ejecución es ininterrumpible y su duración en el tiempo no es significativa
- **Acción:** comportamiento no interrumpible ejecutado a la entrada o salida de un estado
  - Su ejecución no dura una cantidad significativa de tiempo
  - Las acciones son ejecutadas sin importar cómo se ingresa o se egresa de un estado

Los casos de prueba derivados por esta técnica están guiados por los siguientes principios:

- Cubrir todas las transiciones
- Cubrir todos los estados
- Cubrir algunas secuencias de transiciones
- Considerar eventos inválidos

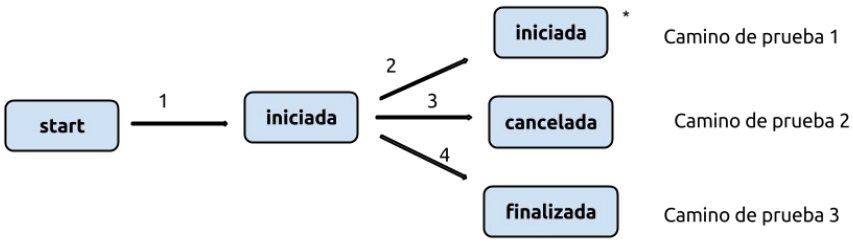


Pasos de la técnica:

1. Componer la tabla de estado-evento
    1. Listar eventos por filas
    2. Agregar una columna para el estado inicial
    3. En cada celda de combinación evento-estado legal colocar el estado resultante con un identificador único de transición
        1. Las combinaciones ilegales se marcan con un punto
    4. Consecutivamente ir agregando una columna por cada estado resultante, y proseguir de igual manera hasta que todos los estados estén listados

Evento	Estado			
	start	iniciada	cancelada	finalizada
`new()`	1 - iniciada	•	•	•
`reg()`	•	2 - iniciada	•	•
`cancelar()`	•	3 - cancelada	•	•
`finalizar()`	•	4 - finalizada	•	•

2. Componer el árbol de transición
    1. Los nodos del árbol son estados y las aristas son transiciones
    2. La raíz del árbol es el estado inicial
    3. Siguiendo la tabla agregamos una arista y un nodo (estado resultante) por cada transición. La arista es marcada con el identificador
    4. Por cada nodo del siguiente nivel, que no aparezca en los niveles precedentes, se procede de igual manera
    5. Las “hojas” del árbol que no sean un estado final o el estado inicial son puntos terminales provisionales y se marcan con *



3. Derivar casos de prueba legales
    1. Con la ayuda del árbol de transiciones y la tabla de estado-evento, se definen los casos de pruebas legales
    2. Cada camino en el árbol de prueba es un caso de prueba. Cada camino posee el evento, las acciones esperadas y el estado resultante.
4. Derivar casos de prueba ilegales

1. Las combinaciones ilegales de estado-evento se pueden obtener de la tabla
2. El resultado esperado es que el sistema no reaccione
3. Si el estado no es inicial, se puede combinar con un caso de prueba legal que conduzca a dicho estado.

Caja Blanca (estructural)

En los criterios o técnicas de caja blanca, a diferencia de los de caja negra, el análisis se realiza teniendo acceso al código fuente que da vida al producto a probar. Es, entonces, foco de estas técnicas la implementación.

Muchos de este tipo de criterios exploran la estructura del código, buscando a través de ese análisis generar suites que ejerciten dicho código de diferentes maneras y permitan una automatización y mensurabilidad.

Cobertura basada en flujo de control

De sentencia

Para satisfacer el criterio de cobertura de sentencias se debe asegurar que se ejecutan todas las sentencias del programa al menos una vez por algún test de la suite, siendo éste uno de los criterios de caja blanca más débiles. Puede dar problemas como errores en condiciones compuestas y ramificaciones que son ignoradas por la ejecución de las pruebas.

Entendemos a una sentencia ejecutable como a aquella que esté asociada al código máquina (asignaciones, evaluaciones de guardas, llamadas a funciones, inicialización de variables, etc.).

En una gran mayoría de casos este criterio puede ser satisfecho con suites pequeñas.

El siguiente programa determina si un año dado, es o no bisiesto:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
        b = true;
    return b;
}
```

Con la entrada **{ a = 2008 }** nos alcanza para conseguir cumplir con cobertura de sentencias.

De decisión

Una decisión es un punto en el código en el que se produce una

ramificación o bifurcación (Ej: condiciones de ciclos, if-then-else).

La técnica de cobertura de decisión se satisface si todos los resultados de las decisiones del programa son ejecutados al menos una vez por al menos un test de la suite. Ella es más fuerte que la cobertura de sentencias y su satisfacción incluye la satisfacción de esta última.

El siguiente programa determina si una cadena es capicúa:

```
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    while (index<(l-1)) {
        if (list[index] != list[(l-index)-1]) {
            return false;
        }
        index++;
    }
    return true;
}
```

¿Qué entradas deberíamos dar para conseguir cobertura de decisión?

{ list = “neuquen”, list = “pero”}

Cobertura de condiciones

Una decisión puede estar compuesta por una o más condiciones, donde una condición es una expresión booleana que es evaluada para determinar el resultado de una decisión.

Ej:

```
if (index < count(list) && !found …
```

La satisfacción de esta técnica se dará cuando cada condición (de cada decisión) es ejecutada por verdadero y por falso por al menos un test de la suite, lo que no implica que sean todas las combinaciones. Cobertura de condición no es más fuerte que cobertura de decisión, estos métodos son incomparables.

De caminos

El grafo de flujo de control de un programa es una representación, mediante grafos dirigidos, del flujo de control del programa:

- Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente
- Los arcos del grafo representan transferencias de control entre nodos

La técnica se satisface cuando todos los caminos del grafo de

flujo de control son recorridos por al menos un test de la suite.

Es un criterio muy fuerte pero que para ser alcanzado puede requerir la creación de suites muy grandes, por lo que para hacerlo practicable se le suelen imponer restricciones como:

- Cobertura de caminos simples: requiere cubrir caminos sin repetición de arcos
- Cobertura de caminos elementales: requiere cubrir caminos sin repetición de nodos

Si una suite satisface cobertura de caminos, ésta satisface cobertura de decisiones y, por lo tanto, de sentencias.

Consideremos el siguiente programa:

```
public static int mcd(int x, int y) {
    int a = x;
    int b = y;
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

¿Qué entradas deberíamos dar para conseguir cobertura de caminos simples?

{ {a=1, b=0}, {a=1, b=1}, {a=2, b=1}, {a=1, b=2} }

CodeCover

CodeCover es una herramienta para medir cobertura de una test suite, de acuerdo a algunos criterios de caja blanca:

- Diseñada para Java+JUnit.
- Se puede instalar como un plugin de Eclipse.
- Muestra visualmente el código cubierto + estadísticas de cobertura.

Web: http://codecover.org/index.html

Mutación

Otro criterio para evaluar suites de testing es el basado en mutación:

- Es un criterio de caja blanca, se basa en análisis del código a testear.

- Es un enfoque diferente a los vistos anteriormente.
- Según este criterio, una suite es adecuada si es efectiva para descubrir bugs inyectados:
  - los bugs se "inyectan" en el código, "mutando" operaciones del mismo
  - cada variante del código original es un mutante
- El objetivo: "matar" todos los mutantes usando tests:
  - un mutante está muerto si existe un test de la suite que no falla en el original, pero si en el mutante.

Consideremos el siguiente programa:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
        b = true;
    return b;
}
```

El siguiente es un mutante del anterior:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) || (a%100!= 0))
        b = true;
    return b;
}
```

Jumble

Jumble es una herramienta para evaluar suites de acuerdo a mutación:

- Diseñada para Java+JUnit.
- Se puede instalar como un plugin de Eclipse.
- Crea mutantes a partir del código original:
  - A nivel de bytecode.
  - Mutantes: cambios en operadores booleanos y aritméticos, etc.
  - Corre suites JUnit y mide score (% de mutantes muertos)
  - Reporta los mutantes que quedaron vivos.
    - útil para mejorar la suite!

Web: http://jumble.sourceforge.net/

Integración continua

La integración continua es un modelo informático propuesto inicialmente por Martin Fowler que consiste en hacer *integraciones automáticas* de un proyecto lo más a menudo posible para así poder detectar fallos cuanto antes. Entendemos por integración la compilación y ejecución de tests de todo un proyecto.

El proceso suele ser, cada cierto tiempo (horas), descargarse las fuentes desde el gestor de versiones (por ejemplo CVS, Git, Subversion, Mercurial o Microsoft Visual SourceSafe o Team Foundation Source Control) compilarlo, ejecutar tests y generar informes.

Para esto se utilizan distintas aplicaciones que se encargan de controlar las ejecuciones, apoyadas en otras herramientas que se encargan de realizar las compilaciones, ejecutar los tests y realizar los informes.

A menudo la integración continua está asociada con las metodologías de programación extrema y desarrollo ágil.

Tiene como ventajas:

- Los desarrolladores pueden detectar y solucionar problemas de integración de forma continua, evitando el caos de última hora cuando se acercan las fechas de entrega.
- Disponibilidad constante de una build para pruebas, demos o lanzamientos anticipados.
- Ejecución inmediata de las pruebas unitarias.
- Monitorización continua de las métricas de calidad del proyecto.

Jenkins

Jenkins es un software de Integración continua open source escrito en Java. Está basado en el proyecto Hudson y es, dependiendo de la visión, un fork del proyecto o simplemente un cambio de nombre.

Jenkins proporciona integración continua para el desarrollo de software. Es un sistema corriendo en un servidor que es un contenedor de servlets, como Apache Tomcat. Soporta herramientas de control de versiones y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como scripts de shell y programas batch de Windows. Liberado bajo licencia MIT, Jenkins es software libre.

Web: http://jenkins-ci.org/

Cierre del curso

Bibliografía

1. Perry, William E., “Effective Methods for Software Testing”, *Wiley*, 3ra edición, 2006
2. Black, Rex, “Managing the Testing Process”, *Wiley,* 2a edición, 2002
3. Kaner, Cem, Falk, Jack, et. al, “Testing computer Software”, *Wiley*, 2a edición, 1999
4. Wikipedia, la enciclopedia libre