Testing

por
Fernando Dodino
Franco Bulgarelli
Gastón Prieto
Pablo Tesone
Juan Pablo Picasso
Gabriel Szlechtman

Versión 3.2
Abril 2016

Indice

"Las pruebas sólo pueden demostrar
la presencia de errores,
no la ausencia de ellos"
E.Dijkstra

1. Introducción

1.1. ¿Qué significa probar?

Es la actividad que tiene por objetivo verificar que el sistema (o una parte de él) funciona de acuerdo a lo especificado. Puede abarcar tanto a los requisitos funcionales como a los no funcionales.

1.2. ¿Para qué probamos?

Porque errar es humano, al diseñar o construir un sistema podemos equivocarnos, obviar aspectos importantes, o malinterpretar los requerimientos. Una de las pocas cosas de las que podemos estar seguros es de que cualquier sistema que construyamos tendrá errores y defectos, entonces el objetivo de probar nuestro sistema será encontrarlos antes de que éstos lleguen al usuario final.

Es decir, mediante las pruebas buscaremos mejorar las cualidades de

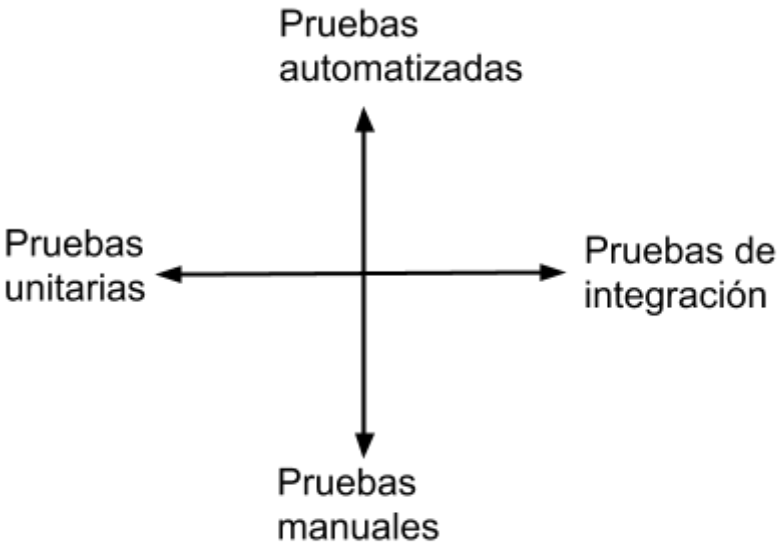
- *Robustez*[1]
- *Eficiencia*, aquí incluimos tanto en el tiempo de respuesta del sistema para el usuario como en el tiempo que tardamos en resolver cada requerimiento
- *Corrección*: si hace lo que el cliente necesita o al menos lo que pidió
- *Consistencia*: si el sistema se comporta siempre de la misma manera ante un mismo evento y las tareas similares se hacen siguiendo pasos similares.
- *Completitud*: si contempla todas las posibles situaciones a darse en la práctica[2].

2. Tipos de pruebas

2.1. Ejes principales

Formas de clasificar a las pruebas hay muchas, según distintos ejes. En este apunte presentaremos el siguiente modelo: granularidad vs. automatización.

[1] El lector interesado puede ver la definición formal en el apunte *Cualidades de diseño*
[2] Recordemos no obstante que llegar a cubrir el 100% de los casos posibles tiene un costo que debe justificar dicha inversión



- **Granularidad:** nos habla de cuanto abarcamos vs. cuanto “apretamos”. Tenemos por un lado pruebas unitarias que prueban componentes individuales con un alto nivel de detalle, y por el otro pruebas de integración, que prueban la interconexión entre componentes
- **Automatización:** nos habla de cuán repetibles son las pruebas sin asistencia humana. Tenemos desde pruebas manuales que deben ser llevadas integramente a cabo por humanos, hasta pruebas automatizadas que son ejecutadas de forma independiente por una máquina.

Es importante notar que ningunas de estas categorías son taxativas: podemos encontrarnos con pruebas más unitarias que otras, y con pruebas manuales que tienen partes automatizadas.

2.1.1. Pruebas de integración vs unitarias

¿Qué es mejor? ¿Realizar pruebas unitarias o de integración?

La pruebas de integración son tentadoras justamente porque al probar integración entre componentes, implícitamente están probando también los componentes mismos. Entonces, ¿significa esto que las pruebas de integración a la larga se solapan con las de unidad, y que por tanto, estas últimas son redundantes?

Si bien esa idea puede sonar correcta, lo cierto es que las pruebas unitarias y de integración se complementan. Cuando tenemos pruebas unitarias:

- resulta **simple razonar** sobre las mismas
- tenemos **control fino** sobre lo que se está probando
- pero para mantener la unitariedad a veces es necesario mockear (ver más adelante) partes extensas de la funcionalidad, con lo que se vuelven bastante abstractas y burocráticas. Ergo, la unitariedad pura lleva a construir pruebas difíciles de mantener.

Por otro lado, las pruebas de integración:

- Son **más realistas**, un verde me da más seguridad: no sólo tengo componentes adecuados, sino que además interconectan adecuadamente.

- Son más **difíciles de construir**, dado que involucran múltiples componentes.
- Una falla en la prueba puede deberse a un error en cualquiera de los componentes, o a la interconección de los mismos. Ergo, la integración pura también lleva a construir pruebas difíciles de mantener, porque **se dificulta rastrear la causa del error**.

2.1.2. Pruebas automatizadas vs. manuales

Cuando nuestro sistema empieza a crecer, comenzamos a tener no decenas sino cientos o miles de pruebas interesantes. Ejecutar cada una de estas no solamente es tedioso, sino también propenso a error, y en definitiva, costoso en términos de tiempo y cantidad de personas necesarias para ejecutarlas. **De ahí que automatizar las pruebas es la única forma sustentable de probar nuestro sistema**.

Automatizar las pruebas parece ser una buena idea, sin embargo, vale la pena hacer una salvedad: a veces parecería que el usuario tiene un don para encontrar errores que quienes construyen el software jamás detectaron. ¿Por qué ocurre esto? Ciertamente el usuario no tiene ningún don especial, pero sí presenta características que lo ayudan a encontrar errores. Podemos pensar dos tipos de usuarios: los que tienen conocimiento del dominio y los que no.

El primer tipo de usuario es el destinatario de nuestro software, por un lado, y por otro, no tiene conocimiento sobre su implementación. Así que es un buen tester natural porque:

- Hace naturalmente pruebas de caja negra (ver más adelante), por lo que no se verá condicionado a probar lo que él cree que fallará porque sabe de la complejidad del código, sino simplemente las funcionalidades que él necesita. Si las pruebas las hace el programador, inconscientemente irá a probar aquello que sabe que es complejo y puede fallar, y omitirá probar aquellas cosas que quizás no son tan complejas, pero pueden contener fallas. Por lo que el tipo de pruebas que hace está sesgada, y con un sesgo muy diferente al del usuario final.
- No probará basado en lo que diga un caso de uso, una historia de usuario o cualquier otra especificación formal, sino en base a lo que el sistema REALMENTE tiene que hacer por él.
- Por último, el usuario final hace pruebas más intensivas que el programador, porque, cuando el software que construimos es su herramienta principal de trabajo, tendrá que usarla 8 hs al día, es decir, 8 hs corridas de testing. 8 hs durante las cuales condiciones infrecuentes seguramente ocurrirán. Por eso el usuario es bueno para encontrar bugs relacionados con condiciones de carrera (**race conditions**, lo deberían ver en Sistemas Operativos), o con **memory leaks** (cuando la aplicación no libera memoria correctamente, otro problema que en Sistemas Operativos tendrán mucho cuando hagan su TP)

Por otro lado, si contamos con un usuario con desconocimiento de dominio, éste prueba muchas veces condiciones absurdas que el programador directamente descarta en su mente. Es decir, como programadores del sistema estamos siempre tentados a probar los casos “felices” (happy path), y no probamos los casos de error (fallas de validaciones, por ejemplo).

Por todo esto es que las pruebas automatizadas son fundamentales pero no reemplazan a las pruebas manuales realizadas por un usuario, tenga o no conocimiento del dominio.

2.2. Pruebas de regresión

Estamos en el contexto de un desarrollo iterativo. Entonces, ¿cuándo creamos nuestras pruebas? Esa es fácil: durante cada iteración.

Eso significa que al finalizar cada iteración tendremos no sólo las pruebas construidas durante la misma, que prueban la funcionalidad creada o modificada durante ésta, sino también, las pruebas generadas en iteraciones anteriores. Entonces, ¿cuáles pruebas ejecuto?

Obviamente es esencial ejecutar las pruebas generadas durante la presente iteración, para validar que nuestro diseño e implementación es correcto. Pero debemos entender que al modificar o extender nuestros componentes, también podríamos haber impactado en componentes preexistentes, porque siempre existe algún grado de acoplamiento.

Por eso es que ejecutaremos también las pruebas anteriores, ya sean manuales o automatizadas (como se imaginarán, correr las pruebas automatizadas anteriores es mucho más simple que ejecutar las pruebas manuales).

Cuando hacemos esto, se dice que estamos ejecutando **pruebas de regresión**. Es decir, lo que hace a una prueba de regresión no es su estructura, sino el momento en que se la ejecuta.

2.3. Pruebas no funcionales

Se combina el software con todos los elementos que componen el sistema (HW, BD, personas). Algunos tipos de prueba en esta instancia:

- **Prueba de seguridad (Análisis de vulnerabilidad)**: tratan de hackear contraseñas, escuchar información que viaja por puertos, acceder a información confidencial, simular una transacción en forma anónima, obtener información de la base de datos, etc. para verificar que el sistema esté preparado para este tipo de ataques maliciosos.
- **Prueba de recuperación:** se introducen adrede fallas en servidores para medir cómo se comporta el sistema
- **Prueba de stress**: se estresa el sistema con una carga superior a la máxima que va a tener para ver si se degrada la performance.
- **Prueba de volumen**: se genera una carga grande de información (automática) para medir la performance.

3. Responsables de las pruebas

¿Quién hace las pruebas unitarias automatizadas? ¿quién las corre, el programador? ¿eso no hace que todo le tome el doble del tiempo?

> *La respuesta es sí: el programador se encarga de escribir los tests. Y sí, escribir tests es una inversión importante de tiempo. Es por eso que muchas veces los ingenieros de software NO los escriben*
> *Decisión que es buena o mala según se la justifique:*

** si es porque testear no aporta o toma tiempo, es una pésima decisión: es cierto que escribir tests consume tiempo, pero es una inversión que se recupera cuando tenés que invertir menos tiempo debuggeando, corrigiendo bugs y fumándote las puteadas del usuario.*

** si es porque estás aplicando criterio y hay algún test* ***particular*** *que no vale la pena escribirlo porque su redacción consume más tiempo que la construcción del código y además el valor agregado que me da esa prueba es bajo (porque por ejemplo, es una funcionalidad que casi con seguridad cambiará en muy corto plazo y no será muy usada), ok, es válido. Pero hay que saber defender ese criterio.*

Como regla general, siempre partan de la base de que escribir el test VALE LA PENA. VALE LA PENA invertir tiempo en su confección. Hay que "tomar mucha sopa"[3] para poder decir que un test no merece ser escrito :P

Respecto de si hay que escribir un test por clase, la realidad es que no. Normalmente hay que escribir MÁS DE UN test por clase :).
Es decir, tendremos que crear muchas pruebas por cada clase de mi sistema. Incluso, muchos conjuntos de pruebas por cada clase; recordemos que lo que estamos probando son componentes, no clases. Nuestros componentes son los objetos, por lo que muchas veces, según como construyamos nuestra instancia (como compongamos al objeto), tengamos componentes tan diversos que valdrá la pena probarlos como si fueran clases diferentes.

¿Quién hace pruebas de integración?

El programador/desarrollador/ingeniero de software debería hacer testing de integración. Igual acá ojo: no nos confundamos. Por pruebas de integración nos referimos a tests de poca granularidad. Éste puede ser tanto automatizado como manual (al testing manual de integración es a veces a lo que se le dice testing funcional).

Entonces, volviendo a tu pregunta, este perfil debería hacer las pruebas de integración pero fundamentalmente de forma automatizada, es decir, escribir código que pruebe al sistema de forma integral, juntando componentes. De hecho, esta es una de las tareas fundamentales de este perfil, tan importante como hacer pruebas unitarias o escribir el código productivo (el código posta).

Además, este perfil podría hacer testing funcional (manual), porque al hacerlo obtendrá feedback valioso y lo acercará al mundo del usuario final. Pero reconociendo que éste no es su principal campo de trabajo.

Bah, más o menos: si tenés una estructura organizacional que separa al tester (o al analista funcional) del programador, claramente el testing funcional es responsabilidad de los primeros, pero si esa separación no existe, entonces quien desarrolle también deberá hacer el testing funcional :)

[3] expresión que puede interpretarse como "hay que aquilatar mucha experiencia en el desarrollo de software"

4. Qué se debe testear

Determinar qué cosas testear y qué cosas no es una tarea muy importante, dado que si se definen mal los casos de prueba un error no será detectado hasta que sea muy tarde. Que los casos de testeo funcionen no significa que la aplicación vaya a funcionar correctamente cuando esté terminada, si los tests están mal la aplicación probablemente también.

Qué necesitamos tener para definir las pruebas:

- pensar un conjunto de datos representativos de las situaciones que pueden darse, también llamado **casuística** o **fixture**. *Ejemplo:* si estamos probando la división de dos números, deberíamos pensar en casos puntuales: a) el divisor es 1, b) el divisor es 0, c) el dividendo es 0 y el divisor es distinto de cero, d) dividendo y divisor son iguales o e) dividendo y divisor son distintos. Es importante tratar de abarcar todos los casos que tengan sentido, y no invertir tiempo en generar juegos de datos similares.
- las pruebas pueden tener
    - entradas válidas, entonces el sistema debe funcionar correctamente.
    - entradas inválidas, que deben provocar que se muestre un mensaje de error.
- si el resultado del test es el esperado, podemos decir que el test salió ok.

Así como es necesario testear todos los casos excepcionales también es preciso no gastar tiempo probando cosas que no tengan sentido[4]. No es necesario testear los accesors ni los métodos con muy poca lógica... pero si algún accesor debe realizar una operación especial (como sacar el legajo de un alumno con una función de hash por ejemplo) entonces es importante que sea testeado.

Cuando el usuario reporta un error, una práctica común es construir primero el test que lo haga fallar. Entonces se corrige el error y se corre nuevamente el test. La ventaja es que de ahora en más tendremos ese test para verificar cualquier comportamiento que se modifique (prueba de regresión).

5. Pruebas y debugging

Entendemos por debugging al proceso necesario para encontrar y resolver los errores (*bugs*) en cualquier sistema de software o hardware[5].

En esta sección haremos un repaso de estas técnicas, con el objetivo de poder compararlas y relacionarlas con las técnicas de testing. Las alternativas más frecuentes son:

- por fuerza bruta o printf debugging
- por rastreo hacia atrás / paso a paso
- eliminando el error

[4] La recomendación de Junit al respecto es “testear todo el código que pueda romperse por sí solo”.

[5] Coloquialmente a veces se utiliza el término *debugging* refiriendo sólo a un subconjunto de estas técnicas, con frecuencia las más antiguas, manuales y con menor contenido “metodológico”. Sin embargo creemos que es más adecuada esta acepción, englobando a todas las técnicas de detección y resolución de problemas.

5.1. Fuerza bruta

La fuerza del printf: se ejecuta una unidad de código imprimiendo valores de variables y textos por pantalla

```
def static main(String[] args){
        println("Pase por aca 1")
        var x = 10
        println("Pase por aca 2")
        x++
        println("Pase por aca 3")
        x++
        println("Pase por aca 4")
        x += 7
        println("Pase por aca 5")
        var y = 20
        println("Pase por aca 6")
        println("El minimo es " + (x < y) ? x : y)
        println("Pase por aca 7")
        println("OK")
}
```

Esto trae varios problemas, el principal es que el código queda muy desprolijo. Una vez testeado el programa hay que borrar las líneas de testeo, lo que puede generar nuevos errores al borrar algo que no corresponde. Esto genera un problema mucho peor, ya que como la unidad está probada y se cree que funciona correctamente corregir el error va a consumir más tiempo del necesario. Otro riesgo es olvidarse de borrar las líneas que imprimen información del programa, lo que puede provocar que al usuario final le aparezcan mensajes que él no debería ver.

5.2. Paso a paso

Los entornos de desarrollo actuales permiten detener la ejecución de un método mediante breakpoints (en cierta línea o al alcanzar una determinada condición) y analizar el estado de los objetos que participan del contexto de dicha ejecución, mediante “watches” o “inspectores”. Esto reduce el tiempo de escribir printf, debug.print o System.out.println, pero aún así hay que descubrir “al voleo” la relación entre variables que produce que el resultado no sea el esperado.

5.3. Rastreo hacia atrás

Busco en un archivo de log dónde se produjo el error y miro las condiciones en las que llegó para que diera error.
TODO relacionar con logs

5.4. Eliminar el error

Se comenta la línea donde hay error para ver si se produce un efecto colateral o ése es un

error aislado que no trae consecuencias posteriores.

Arreglar un error trae las *pruebas de regresión* necesarias por el **efecto colateral**. Lo difícil: arreglar algo tratando de no romper lo que anda…

5.5. ¿Qué tiene de malo el debugging interactivo?

- lleva tiempo
- cada error nuevo induce a invertir otra vez ese mismo tiempo y lo más probable es que no nos sirva nada haber solucionado un error anterior
- se convierte en una actividad tediosa
- se pierde el foco en lo que estamos haciendo (pasado un cierto tiempo quedamos envueltos en el código, absorbidos por el monitor y los inspectores de variables que tenemos que tener en nuestra cabeza)
- produce una falsa confianza a la hora de codificar (codifique mal ahora, debuggee después)
- en metodologías ágiles como XP (eXtreme Programming), el debuggeo se reemplaza por el pair programming[6], donde a medida que se escribe código la persona que no escribe está validándolo y obliga a que quien escriba justifique cada decisión de programación (se ahorra mucho más tiempo que debuggeando solo).

De todas maneras, sigue siendo una herramienta más, que puede complementarse con las otras técnicas descritas anteriormente. Además puede resultar útil cuando es difícil comprender todo el juego de variables que intervienen en un caso de uso.

6. Diseño de pruebas automatizadas

6.1. Introducción

Al construir pruebas, al igual que al construir sistemas, es necesario detenerse un momento a diseñar, y nuevamente tendremos que tomar decisiones, en función de cualidades de diseño. Y todo esto porque las pruebas evolucionarán a la par del sistema y presentarán muchos de sus mismos desafíos.

En la próxima sección hablaremos sobre cualidades de diseño de las pruebas. Si bien muchas de estas ideas aplicarán para pruebas tanto manuales como automatizadas, haremos foco fundamentalmente en estas últimas.

¿Por qué? Sucede que son herramientas fundamentales en el desarrollo de software que nos permiten:

1. Validar nuestro diseño, y verificar que una implementación se comporte acorde a las especificaciones

[6] El pair programming es una técnica de desarrollo de software donde dos personas trabajan en la misma estación de trabajo al mismo tiempo: una cumple el rol de analista y el otro de programador; los roles no son fijos sino que se van invirtiendo. Los resultados de las experiencias es que no se reduce el tiempo de construcción de software pero sí aumenta considerablemente su calidad.

2. Guiar nuestro diseño, y encarar cambios en el mismo con mayor seguridad.

Su fortaleza radica en que son código ejecutable, y no, por ejemplo, una especificación en un bibliorato. Éstas se encuentran normalmente junto con el código productivo, por lo que los cambios en las especificaciones del mismo podrán ser detectados rápidamente: el test dejará de correr.

Sin embargo, al tratarse de código, un desarrollo poco cuidadoso de los casos de prueba podría introducir problemas similares a los que surgen con el código productivo:

- **Baja expresividad:** las pruebas no revelan su intención, y por lo tanto, no se entiende qué es aquello que se está probando.
- **Baja cohesión**: un test prueba más funcionalidades de lo que debería
- **Alto acoplamiento**:
    - Modificar un test impacta en los demás
    - Pequeños cambios en el código productivo impactan en demasiados tests
- **Bajo nivel de abstracción**: las pruebas no tienen el nivel de abstracción suficiente y repiten lógica.

De darse estos problemas, nuestros tests se volverán inmantenibles y lejos de validar o guiar nuestro diseño serán un lastre para cambios futuros.

Es así como muchas veces surgen en proyectos prácticas tan aviesas como la que resume el lema “test que se rompe, test que se elimina”, y paulatinamente nuestra preciada batería de pruebas en la que tanto tiempo y esfuerzo se invirtió, terminará por desaparecer. Para evitar llegar a eso, debemos procurar desarrollar pruebas con niveles de calidad similar al del código productivo.

La moraleja: al desarrollar tests, también hay que diseñar.

6.2. Cualidades de diseño de las pruebas automatizadas

Al igual que para el código productivo, las cualidades de diseño aplican a los tests:

- **Corrección:** deben probar lo que la especificación dicta
- **Repetibles:** una prueba debe de forma consistente generar el mismo resultado mientras el código bajo prueba no cambie
- **Completos:** deben probar tantos escenarios y funcionalidades como sea posible, complementando unitariedad con integración.
- **Mantenibles:** no deben repetir lógica entre sí y evitar probar más de una vez lo mismo. Además deben ser expresivos y breves para facilitar su lectura. Es decir, buscaremos que modificar el test sea tan fácil como modificar el componente o funcionalidad bajo prueba
- **Baratos:** deben ser rápidos de desarrollar y mantener. Se debe invertir tiempo en probar sólo funcionalidad que valga la pena probar
- **Rápidos:** cuanto más tiempo tome el test en ser ejecutado, con menos frecuencia los ejecutará el responsable de hacerlo.
- **Independientes:** una prueba no debe tener interacción con otra, ni depender del orden de ejecución de las pruebas para poder funcionar. Es decir, debemos minimizar todo

posible acoplamiento entre dos tests.

Conceptos relacionados:

- Buscamos que el test unitario no tenga *efecto colateral* para que sea repetible e independiente de los demás. Si la ejecución modifica recursos externos, como bases de datos, el test debe incluir código para restaurar el ambiente hasta el punto anterior de comenzar el test.
- *Cohesión:* unidades altamente cohesivas (con un objetivo bien definido) facilitan el testing. Si un método resuelve tres problemas, es difícil dividirlo en partes para testearlas en forma individual.
- *Acoplamiento:* cuando un componente tiene alta interacción con otros componentes, es muy complicado hacer el testeo “unitario” (porque termino probando todo)

6.3. Anatomía de una prueba

Primero, ¿cómo está conformado un test? En toda prueba, ya sea automatizada o manual, siempre están presentes los siguientes elementos:

- **Precondiciones**: el conjunto de supuestos de los que partimos. Esto incluye la construcción de los datos de prueba (fixture), y llevar el sistema al estado propio del escenario que se quiere probar.
- **Operaciones**: aquella funcionalidad que está bajo prueba.
- **Postcondiciones**: lo que se espera que produzca la operación bajo las precondiciones dadas, es decir, el estado al que se lleva al sistema, efectos producidos o valores devueltos.

¿Les suena esta estructura? Es análoga a la de una demostración matemática: dado un conjunto de supuestos (*hipótesis*), después de aplicar ordenadamente un conjunto de reglas lógicas, se pueden realizar ciertas afirmaciones (*tesis*).

Entonces, para que una prueba automatizada revele su intención, **es importante que estos tres componentes queden explícitos en el test**.

De todas formas, como veremos, hay un cuarto elemento, que si bien no forma parte de la definición mínima de un test, sí será esencial a la hora de codificar pruebas automatizadas: el **nombre de la prueba**.

6.4. Estilos de testing

Podríamos escribir pruebas de varias formas, a continuación analizaremos dos: teorías y el estilo “tradicional”.

6.4.1. Teorías

Supongamos que queremos probar una operación tan simple como el factorial de un número. Una de las tantas pruebas que podríamos pensar es que:

Si tengo un número positivo (precondición), y calculo su factorial (operación), obtengo un número positivo (postcondición)

Otra prueba, un poco mejor y que incluye a la anterior y es de hecho, más estricta, es:

Si tengo un numero positivo (precondición), y calculo su factorial (operación), obtengo un número mayor o igual al número original (postcondición)

¿Como se vería esto en pseudocodógico?

```
siendo n > 0
si hago f  = factorial(n)
debe cumplirse que f >= n
```

Esta forma de plantear una prueba se la conoce como teoría. Es una forma de probar un código típica de una familia de frameworks de testing que llamaremos XCheck (donde X es el nombre posta del framework). Ejemplos de estos frameworks son ScalaCheck (Scala), QuickCheck (Haskell), Junit + Theories (Java)

Esta forma es bastante general y poderosa, que nos permite probar lo que se conoce como *invariantes*. Sin embargo, no todos los frameworks se la bancan. Entonces muchas veces usaremos una forma más rústica, pero también complementaria.

6.4.2. Pruebas tradicionales

Esta forma consistirá en dejar a mis precondiciones prefijadas con valores conocidos y no con valores que cumplan restricciones. Es decir, tendré que poner ejemplos concretos, por lo que a partir de una teoría, tendré que extraer varios tests. Estos ejemplos concretos se los conoce como ***fixtures***.

Algo de la siguiente forma:

Si tengo el 4, y calculo su factorial, obtengo 24
Si tengo el 1 y calculo su factorial, obtengo 1
Si tengo el 0 y calculo su factorial, obtengo 1
etc.

Por ejemplo, un pseudocódigo de una prueba así es el siguiente:

```
siendo n = 4
si hago f  = factorial(n)
debe cumplirse que f = 24
```

O, como solemos encontrarlas en frameworks xUnit, como JUnit, SUnit, NUnit:

```
deberiaSerIgualA(24, factorial(4))
```

En cualquier caso, **las pruebas deben tener un nombre que represente lo que se esta probando.**

por ejemplo:

factorial de 4 es 24:
```
  deberiaSerIgualA(24, factorial(4))
```

factorial de un positivo es mayor o igual al número original:
```
  siendo n > 0
  si hago f  = factorial(n)
  debe cumplirse que f >= n
```

Por último, cuando pienso pruebas, también debería pensar casos de prueba para las validaciones. Por ejemplo:

el factorial de un negativo debería fallar
```
   deberiaFallar( {   factorial(-2)  } )
```

6.5. Cualidades de las pruebas, en detalle

Ahora que tenemos un mejor entendimiento de cómo está conformado un test, veamos consideraciones de diseño específicas para cada componente del mismo.

6.5.1. Cualidades del fixture

Ahora, al diseñar un fixture, ¿qué cosas deberíamos cuidar? Algunas cualidades de un fixture son:

- **Simple y reducido**: Los fixtures deberían tender a ser simples y de tamaño reducido. Si para probar el escenario se requiere un fixture muy grande o complejo de construir, es probable que o bien se trate realmente de un caso de prueba muy complejo o, más comúnmente, se esté probando más de lo necesario.
- **Específico**: No construir un fixture general para todas las pruebas, sino uno específico para cada una. De lo contrario, mantener el fixture se vuelve dificultoso, dado los tests quedan acoplados entre sí a través del fixture, y una modificación mínima en el mismo terminará impactando normalmente en varios tests.
- **Mínima redundancia**: cualidad muchas veces contrapuesta a la anterior; se trata de evitar repetir la declaración de casos de fixtures idénticos o muy similares.
  Una forma de conciliar ambas cualidades es desarrollar fixtures jerárquicos: conjuntos de datos de prueba generales y esenciales para todas las pruebas, pero que luego son refinadas en cada test.[7]
- **Favorecer el diseño de código puro**: probar código puro (sin efectos) suele ser más fácil que probar código que no presenta esta característica. Esto no es algo propio del test, sino del diseño sistema productivo.

6.5.2. Cualidades de las operaciones

De igual forma, podemos encontrar algunas cualidades que deberían presentar el conjunto de

[7] Por ejemplo, en herramientas como xUnit, esto se puede lograr mediante uso de herencia entre conjuntos de tests o métodos setup generales, especializados en cada método de prueba.

las operaciones de la prueba:

- **Simples y reducidas**: Las operaciones involucradas en la prueba deberían ser pocas, idealmente sólo una. Si se necesita realizar muchas operaciones, probablemente se deba a que:
  - Faltan abstracciones en el código productivo (deficiencia del diseño).
  - Se está probando un escenario que no es completamente real, sino una simplificación del mismo. Esta secuencia de pasos debería ser encapsulada en el contexto del test[8].
- **No romper el encapsulamiento**: si se presenta esta necesidad, probablemente se deba a un alto grado de acoplamiento entre los componentes involucrados en la prueba. Quizás sea el momento de buscar nuevas abstracciones y extraer componentes.

6.5.3. Cualidades de las postcondiciones

- **Simples y reducidas**: Minimizar la cantidad de aserciones por test. Idealmente, debería haber una sola aserción por test. Si no es posible, es probable que también haya alto acoplamiento entre los componentes del sistema y mala asignación de responsabilidades, lo que lleva a que dos consecuencias de la ejecución de la operación no puedan ser probadas por separado.
- **Grado de abstracción**: A veces las aserciones provistas por el motor no son suficientes porque no entienden de mi modelo. Crear aserciones de alto nivel.

6.5.4. En general

Finalmente, podemos encontrar algunas cualidades generales para la prueba:

- **Completitud**: Probar casos felices (aquellos que constituyen el curso más común de uso del componente), casos borde y casos de error.
- **Evitar solapamiento**: No probar casos que (al momento de escribir el test) pertenecen a la misma clase de equivalencia de prueba.
- **Ser críticos con la funcionalidad**: si somos críticos respecto de los test que diseñamos y construimos, encontraremos que allí es muchas veces donde la falta de abstracciones queda en evidencia.

7. Frameworks de testing

7.1. Motivación

Mencionamos varias veces la palabra framework de testing. ¿Qué es esto? Sucede que para probar código necesito código. Es decir, tengo que escribir código que prueba código. Y para hacerlo necesito ciertas primitivas que podría implementar, como las marcadas en negrita.

Pero en lugar de implementarlas, ya hay gente que lo hizo en forma de biblioteca. Y no solo hizo una biblioteca de testing que me da funcionalidades para probar código, sino que también

[8] Por ejemplo, en xUnit, extrayendo un método dentro de la clase que contiene el conjunto de pruebas.

me dio un *framework* que sabe cómo tomar pruebas escritas de alguna forma que me propone, procesarlas y ejecutarlas. Y darme luego los resultados, con colorcitos y mensajitos lindos :P Es decir, el framework de testing es un motor que sabe ejecutar pruebas, asociado con el concepto de declarativdad.

SUNit, JUnit, ScalaCheck, son todos frameworks de testing.

En Junit por ejemplo, la última prueba

```
el factorial de un negativo debería fallar
   deberiaFallar( {   factorial(-2)  } )
```

se escribe de la siguiente forma:

```
@Test(expect=IllegalArgumentException.class)
public void elFactorialDeUnNumeroNegativoDeberiaFallar() {
    factorial(-2);
}
```

Y para ejecutarlo, tenemos que usar o bien maven o bien eclipse, que utilizará JUnit para darnos los resultados (cómo probar con Eclipse está explicado en el apunte principal / parte de este texto está desarrollado también en el apunte complementario).

7.2. Estilos

Existen varios estilos, xUnit, xCheck, xSpec. En la sección siguiente haremos foco en xUnit.

8. Testing con JUnit

JUnit es un framework que nos permite utilizar Unit Testing en Java (y lenguajes derivados, como Groovy, XTend, Scala, etc.), basado en XUnit. Las clases básicas de JUnit se ven en el siguiente diagrama de clases.

8.1. Un ejemplo sencillo

Tenemos la clase Cliente (con fines didácticos, obviamente):

```
class Cliente {
      static val DEUDA_MAXIMA = 100

      String nombre
      String apellido
      Float deuda = 0f

      new(String nombre, String apellido) {
         this.nombre = nombre
         this.apellido = apellido
      }

      def esMoroso() {
```

```
        deuda > 0
    }

    def pagar(Float monto) {
        if (deuda < monto)
            throw new MontoAPagarExcedeDeudaException

        deuda = deuda - monto
    }

    def comprar(Float monto) {
        if (deuda + monto > DEUDA_MAXIMA)
            throw new MontoMaximoExcedidoException

        deuda = deuda + monto
    }

    override toString() {
        nombre + " " + apellido
    }
```

Para crear tests unitarios desde el Eclipse usaremos la carpeta o Source Folder src/test/java (que Maven se encarga de crear por nosotros). La idea es mantener los tests separados del código de “negocio” (Cliente, en nuestro caso). Entonces creamos una nueva clase Xtend, de la siguiente manera:

- Asignamos el Source Folder recientemente creado
- Le ponemos un nombre representativo de la unidad que estamos testeando (no necesariamente es una sola clase, puedo agrupar varias clases en una unidad funcional)

Al confirmar la creación, nos queda el esqueleto de la clase.

En Xtend usando JUnit4 vamos a ver que los tests no son más que clases que tienen métodos anotados con el annotation **@Test**. Para poder usar JUnit4 debemos incorporar la dependencia a JUnit dentro del archivo pom.xml de Maven de nuestro proyecto. Por lo general los parent-projects van a a hacer esto por ustedes, pero si están en un proyecto específico pueden hacer

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

8.2. Fixture

Para poder realizar los tests vamos a necesitar crear instancias de los objetos cuyas clases queremos testear, y probablemente de otras también, esto se llama fixture (o juego de datos). Para esto debemos declarar las variables de instancia privadas del Test en la clase y luego escribir un método demarcándolo con la annotation **@Before** para instanciar los objetos

necesarios en cada test.

```
class ClienteTest {
	var Cliente rodri

	@Before
	def void init() {
		rodri = new Cliente("Rodrigo", "Girlando")
	}
```

En el caso en el que esté consumiendo recursos (abrir un archivo, un socket, pedir una conexión a la base, etc.), se debe escribir un método para liberarlos. Este método se puede llamar como sea pero debe estar marcado con la annotation **@After**.

8.3. Definición de tests

Veamos cómo queda la clase ClienteTest una vez que desarrollamos cuatro casos de prueba:

```
	@Test
	def void testClienteInicializadoNoEsMoroso() {
	   Assert.assertFalse(rodri.esMoroso)
	}

	@Test
	def void testClienteQueCompraEsMoroso() {
	   rodri.comprar(10f)
	   Assert.assertTrue(rodri.esMoroso)
	}

	@Test (expected=typeof(MontoAPagarExcedeDeudaException))
	def void testPagarMasQueLaDeuda() {
	   rodri.pagar(10f)
	}

	@Test (expected=typeof(MontoMaximoExcedidoException))
	def void testComprarSuperandoMaximoPermitido() {
	   rodri.comprar(10000f)
	}

	@Test
	def void testComprarDosVeces() {
	   rodri.comprar(20f)
	   rodri.comprar(25f)
	   Assert.assertEquals(45f, rodri.deuda, 0.01)
	}
```

1. En el caso del primer test, lo que se quiere probar es que cuando creo un cliente se inicializa correctamente, sin deuda. Por eso asume que es falso que rodri sea moroso

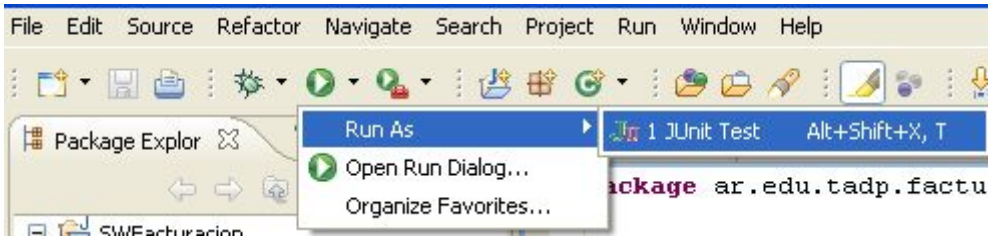
(assertFalse).

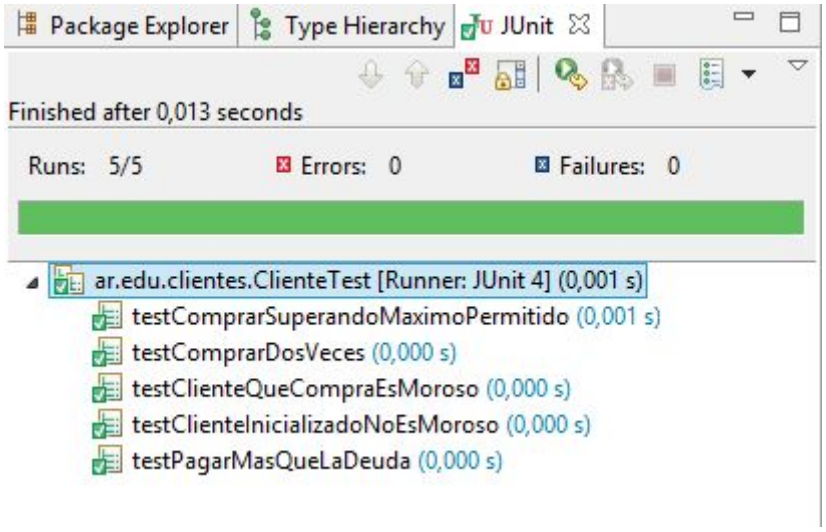
2. En el segundo test, lo que se quiere probar es que cuando un cliente nuevo compra, pasa a ser moroso. Por eso asume que es verdadera esta afirmación (assertTrue).
3. El tercer test está esperando que cuando un cliente nuevo quiera pagar (antes de comprar), la aplicación lance una excepción indicando que el monto a pagar excede la deuda. Por eso el parámetro expected de la annotation Test permite decir *declarativamente* eso, sin tener que hacer:

```
@Test
def void testPagarAlternativo() {
	try {
		rodri.pagar(10f)
	} catch (MontoAPagarExcedeDeudaException e) {
		return
	}
	fail()
}
```

4. El cuarto test es similar, se espera que la aplicación lance una excepción cuando un cliente quiera comprar más allá del monto máximo permitido
5. El último test compara el valor esperado de la deuda de un cliente que hace dos compras

Si queremos ejecutar los tests, hacemos:





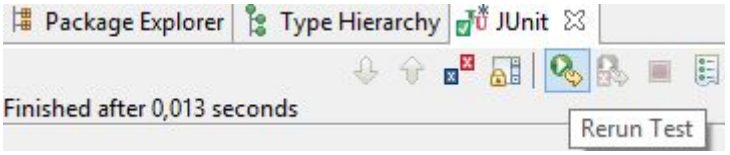
Se muestra una barra verde si los todos los tests pasaron OK. Además se puede visualizar el

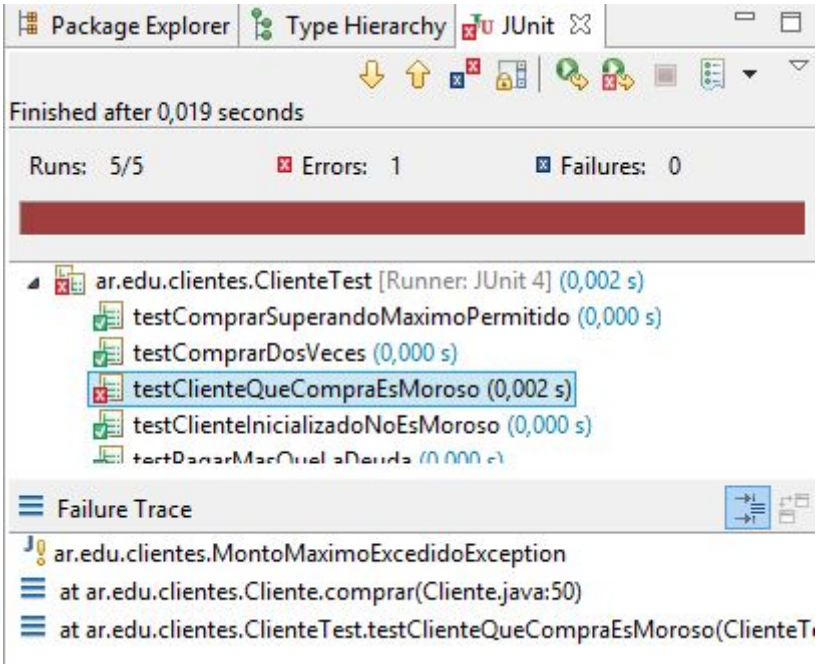
resultado de cada uno de los test ejecutados. Supongamos ahora que modificamos el método que chequea si un cliente que compró es moroso a:

```
@Test
def void testClienteQueCompraEsMoroso() {
	rodri.comprar(10000f)
	Assert.assertTrue(rodri.esMoroso)
}
```

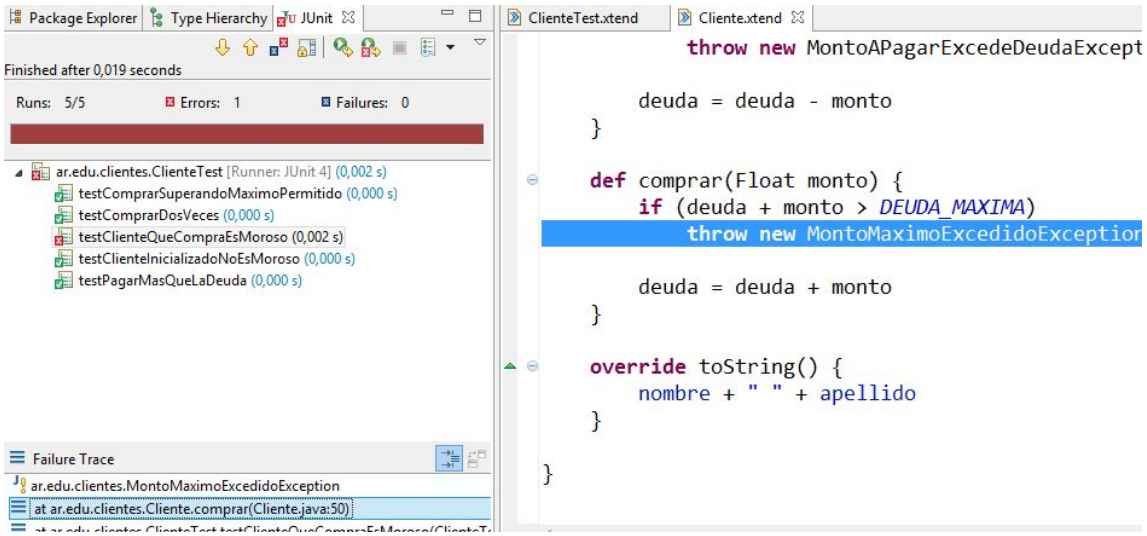
Por supuesto, esto no cumple la regla de negocio del máximo permitido (de hecho pasa a ser igual al testComprarEnExceso), pero los fines son didácticos…

En el Eclipse pedimos que vuelva a correr el test (haciendo click en el botón Rerun Test):





Ahora vemos que el testEsMoroso dio error. Si nos paramos sobre testEsMoroso vemos el Stack Trace y con un click sobre el Stack llegamos al código donde se generó el error:



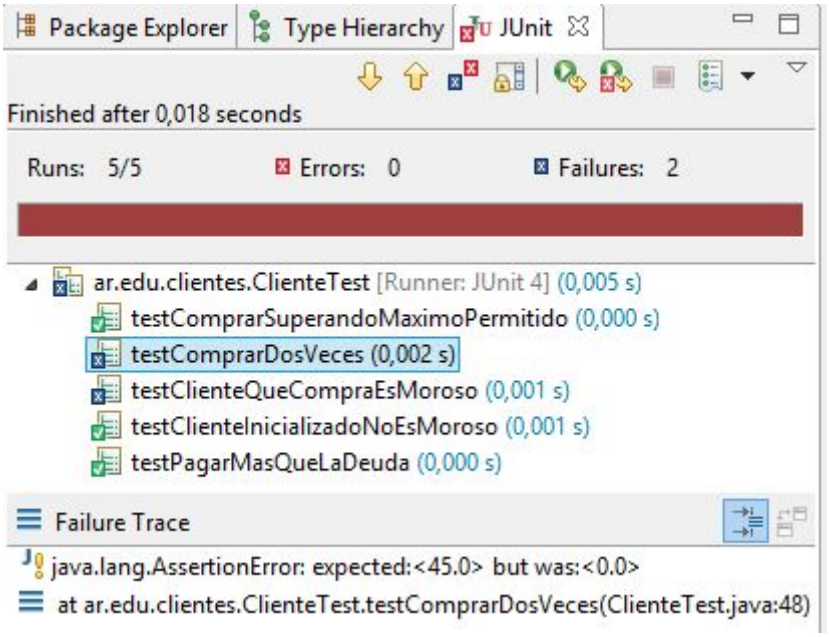
Dejamos el test como estaba originalmente:

```
@Test
 def void testClienteQueCompraEsMoroso() {
	rodri.comprar(10f)
	Assert.assertTrue(rodri.esMoroso)
 }
```

y ahora cambiamos el método que compra para que olvide agregar la deuda:

```
def comprar(Float monto) {
	if (deuda + monto > DEUDA_MAXIMA)
		throw new MontoMaximoExcedidoException

	//	deuda = deuda + monto
 }
```

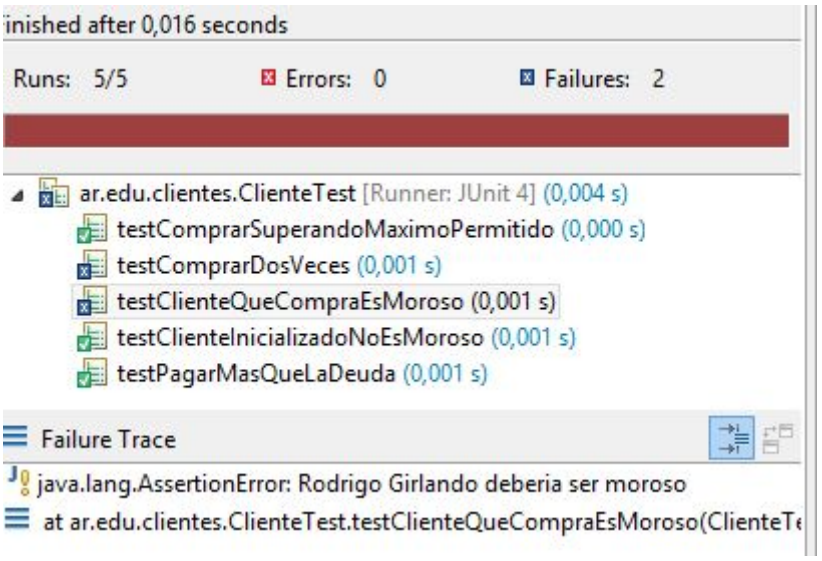


Ahora algunos tests muestran fallas (Failures). ¿Qué significa AssertionError? Que esperaba

que la deuda fuera de $ 45, pero el cliente no tenía deuda (lo mismo pasó con el test de morosidad).

Una buena práctica al asumir cosas, es dejar un mensaje descriptivo cuando el test falle, indicando qué es lo que se esperaba:

```
@Test
def void testClienteQueCompraEsMoroso() {
      rodri.comprar(10f)
      Assert.assertTrue(rodri + " deberia ser moroso", rodri.esMoroso)
}
```



¿Qué diferencia hay entre error y falla?

El test de un método puede estar “no ok” por dos motivos:

- Si dio error al invocarse (error), como en el primer caso
- Si no dio error, pero su comportamiento no fue el esperado (failure = falla, “Rodrigo Girlando debería ser moroso”)

8.4. Definición de Teorías

```
@Theory
void <A> cardinalidadDeUnionDeConjuntosDisjuntosEsLaSumaDeSusCardinalidades(
    Conjunto<A> conjunto1, Conjunto<A> conjunto2) {
    //hipótesis
    assumeTrue(conjunto1.esDisjunto(conjunto2));

    //tesis
    assertEquals(conjunto1.size() + conjunto2.size(),
     conjunto1.union(conjunto2).size());
}
```

En lugar de definir los valores necesarios, establezco las reglas para que un motor los construya, con mayor rigurosidad de la que yo programador puedo darle. Aunque este tipo de pruebas se llevan bien para código puro (de hecho, este estilo de prueba unitaria es propio de la biblioteca estándar de testing de Haskell, QuickCheck), también es posible aplicarlas a código con efectos. Por ejemplo, es una verdad universal que toda golondrina que se precie, cuando vuela a una ciudad, pierde energía:

```
@Theory
void lasGolondrinasPierdenEnergiaAlVolar(Ave ave, Ciudad destino) {
    assumeTrue(!ave.estaEn(destino));

    int energiaInicial = ave.getEnergia()
    ave.volarA(destino)

    assertThat(ave.getEnergia(), lessThan(energiaInicial));
}
```

Lo cual no prueba que el cálculo sea el correcto para específicamente las GolondrinaTijerita, pero si una propiedad fundamental que se debe cumplir para todo tipo de ave, independientemente de si está cansada, de si la ciudad está al norte o el sur de su ubicación actual, etc. El programador entonces solo debe preocuparse de construir un buen fixture:

```
@DataPoints
public static Ciudad[] ciudades() {
  return new Ciudad[]{ /*estambul, buenosAires, etc*/ } ;
}

@DataPoints
public static Ave[] aves() {
   return new Ave[]{ new Golondrina("pepita"), new Halcon("charly"),/*etc*/
};
}
```

Es por eso que este tipo de test es complementario a las pruebas tradicionales, dado que por ser más abstracto, realiza pruebas menos detalladas, pero más abarcativas.

9. Impostores

9.1. Mock objects

En programación orientada a objetos, los objetos impostores[9] son objetos que simulan el comportamiento de los objetos reales, de forma controlada. Generalmente son programados para verificar el comportamiento de otro objeto, de forma parecida a que los diseñadores de autos utilizan muñecos (*crash test dummies*) para verificar el comportamiento de un auto durante un accidente.

[9] También llamados fake objects/mock objects. Pueden consultar www.mockobjects.com

9.2. ¿Para qué usar mock objects?

Cada vez que tengamos un objeto que

- provea valores no controlables o aleatorios (por ejemplo, la fecha actual o la temperatura ambiente)
- tenga estados difíciles de crear o reproducir (por ejemplo, un error de red)
- tenga efecto colateral (por ejemplo, una base de datos, la cual debe ser inicializada antes del test)
- o tenga restricciones de performance (por ejemplo, al correr 200 veces una consulta a la base de datos)
- aún no exista o su comportamiento pueda cambiar
- deba incluir información y métodos exclusivamente para propósitos de testeo (no para su objetivo real)

podemos trabajar con objetos "imitadores" para conservar la simplicidad y el aislamiento de los tests unitarios.

Por ejemplo, un programa de reloj de alarma que hace que una campana suene a un momento determinado podría obtener la hora actual del mundo exterior. Para construir una prueba unitaria, el test debe esperar hasta que sea el horario de la alarma para saber si sonó correctamente. Si se utiliza un *mock object* en vez del objeto real, puede ser programado para proveer la hora en la que debe sonar la alarma (sea esa o no la hora actual) para poder testear el programa de alarma de forma aislada.

9.3. Stubs y mocks

Martin Fowler[10] propone varias categorías de este tipo de objetos, nos interesan dos en particular:

- **stubs**: simplifican la complejidad del objeto que imitan, ofreciendo respuestas predefinidas en forma determinística. Son ideales para tests que verifican *el estado* de los objetos que participan de la prueba.
- **mocks**: no solamente se modelan con lógica necesaria para que el test funcione, sino que definen algunas restricciones o *expectativas*: cuántas veces deben ser invocados determinados mensajes, qué parámetros esperan recibir, etc.

9.4. Implementación de un Stub

Los stubs tienen la misma interfaz que los objetos que imitan (polimorfismo), permitiendo a los clientes del objeto desconocer si utilizan el objeto real o el imitado.

Un ejemplo breve: desarrollamos la clase Cliente y queremos testearla, pero todavía no está implementada la clase Factura. Sabemos que un cliente tendrá una colección de facturas pero todavía no sabemos cómo se va a implementar. Entonces al cliente real le asociamos un objeto StubFactura que devuelva un saldo *x* con un valor fijo. Ahora sí el cliente puede calcular el saldo pendiente de sus facturas y determinar si es moroso o no:

>>Cliente

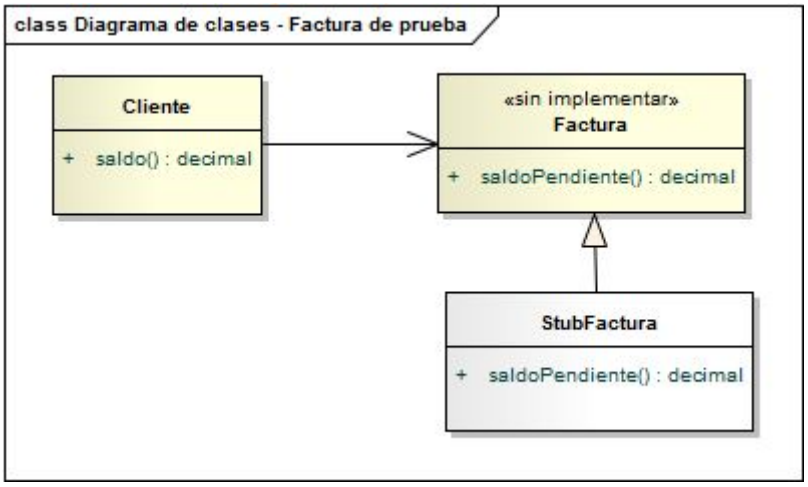
```
def getSaldo() {
    facturas.fold(0d, [ acum, factura | acum + factura.saldoPendiente])
}
```

[10] Ver el artículo http://martinfowler.com/articles/mocksArentStubs.html y creemos que mejor explicado está en https://adamcod.es/2014/05/15/test-doubles-mock-vs-stub.html

Tenemos una implementación “de resguardo” StubFactura, que implementa la misma interfaz que Factura (entiende el método saldoPendiente).

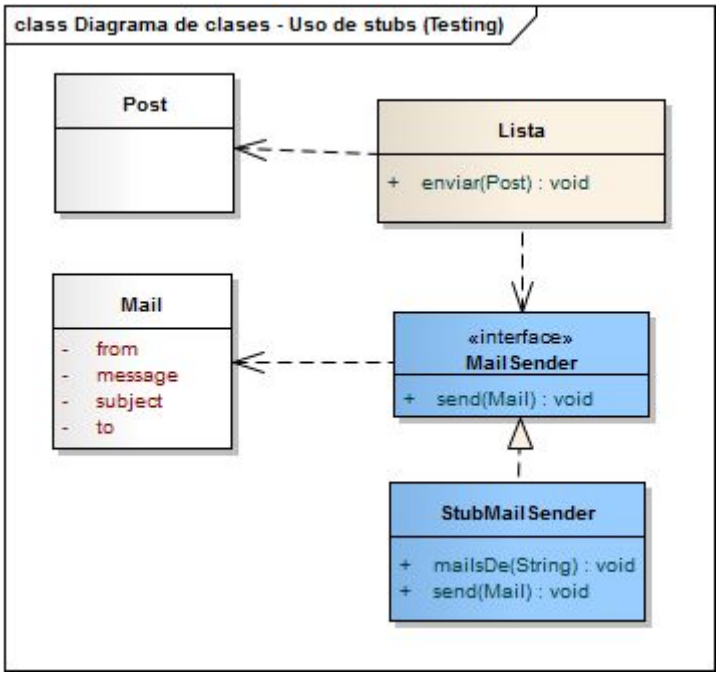
>>StubFactura

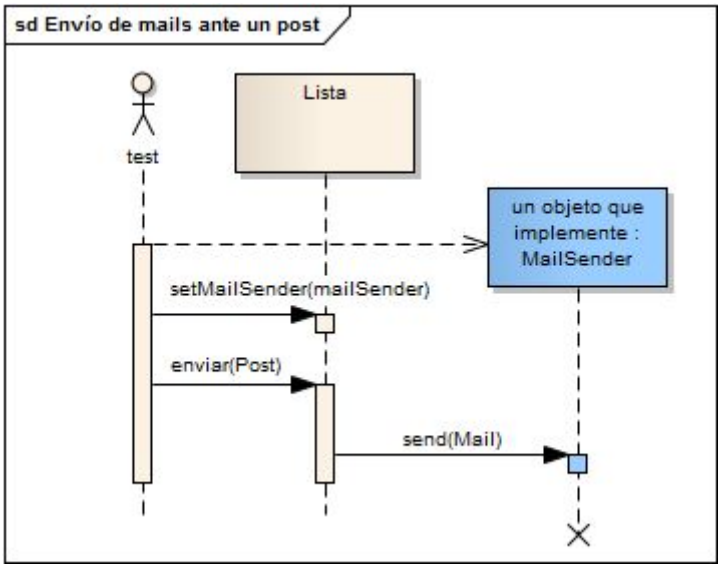
```
def StubFactura extends Factura {
      override double saldoPendiente() { 0 }
}
```



9.5. Otro stub: Lista de correo

Tenemos una lista de correo con tres usuarios: gato, pelu y dodi (dodi@uni.edu.ar). Cada vez que alguien envía un mensaje a la lista los demás lo reciben en sus respectivas casillas. Contamos con un objeto que modela la Lista y que tiene un método enviar(), que recibe un Post y lo transforma en un objeto Mail para enviar el mensaje send a un objeto que implementa la interfaz MailSender.





Si dodi envía un mensaje, ¿qué deberíamos probar? Podríamos revisar la casilla de mails enviados de dodi o los mails recibidos de gato y pelu, pero vamos a chequear *que el servidor haya despachado el mail*.

Cuando el sistema esté en funcionamiento la lista conocerá un objeto MailSender real, que enviará mails vía smtp. Pero en los tests unitarios sería bueno no enviar mails, entonces tendremos un objeto StubMailSender.

En el método init del test (o bien al inicializar los atributos de la clase Test) inyectamos la dependencia de la lista de correo con un StubMailSender (para ello, tenemos que asegurar que el mailSender sea una propiedad seteable desde afuera del objeto Lista de Correo).

>>Código del Test

```
StubMailSender stubMailSender = new StubMailSender

def void init() {
	...
	listaAlumnos.mailSender = stubMailSender[11]
```

9.6. Test de estado

En el test tenemos:

- una precondición: dodi no envió ningún mensaje a la lista
- la acción que consiste en que dodi envíe un mensaje
- y como postcondición queremos verificar que se haya enviado el mismo mensaje desde la casilla de mail de dodi.

```
@Test
def void alumnoPuedeEnviarMailAListaAbierta() {
    Assert.assertEquals(0, stubMailSender.mailsDe("dodi@uni.edu.ar").size)
    listaAlumnos.enviar(mensajeAlumno)
    Assert.assertEquals(1, stubMailSender.mailsDe("dodi@uni.edu.ar").size)
}
```

[11] Pregunta para el lector: ¿qué pasaría en lugar de la propiedad mailSender se trabajara con un Singleton?

El test trabaja sobre el **estado final del stub**, para lo cual vamos a modelar un mapa cuya clave es el mail emisor y cuyo valor es un conjunto de mails enviados, sin repetir el mensaje[12]:

```
class StubMailSender implements MessageSender {
	Map<String, Set<String>> mailsEnviados

	new() {
		mailsEnviados = new HashMap<String, Set<String>>
	}

	override send(Mail mail) {
		var mensajes = mailsDe(from)
		mensajes.add(message)
		mailsEnviados.put(from, mensajes)
	}

	def Set<String> mailsDe(String from) {
		var Set<String> mensajes = mailsEnviados.get(from)
		if (mensajes == null) {
			mensajes = new HashSet<String>
		}
		mensajes
	}

}
```

9.7. Implementación de un Mock para la lista de correo

Ahora vamos a trabajar con un framework de mocking llamado Mockito, que funciona para cualquier lenguaje que soporte la JDK[13]. Entonces vamos a complementar el test que verifica el estado del stub con un **test de comportamiento**, donde se definirá un mock sobre el MailSender y luego se verificará cuántas veces se envía el mensaje send a este objeto mockeado:

```
@Test
def void testEnvioPostAListaAlumnosLlegaATodosLosOtrosSuscriptos() {
	//creacion de mock
	val mockedMailSender = mock(typeof(MailSender))
	listaAlumnos.mailSender = mockedMailSender

	// un alumno envía un mensaje a la lista que tiene 3 integrantes
	// uno es el que envía el post
	listaAlumnos.enviar(mensajeAlumno)

	// test de comportamiento, verifico que se enviaron 2 mails
	// porque el que envía el post no recibe el mail
	verify(mockedMailSender, times(2)).send(any(typeof(Mail)))
}
```

[12] El uso del contrato Set permite que la lista envíe los mails en forma individual a cada usuario (n mails, uno por usuario) o bien que se envíe un único mail a todos los suscriptos.

[13] esto incluye a Java, XTend, Groovy y Scala

Mockito -al igual que otros frameworks- permite no sólo definir métodos stubs para los objetos que pertenezcan a una clase o que implementen una determinada interfaz (como en el caso del MailSender), sino también establecer **expectativas**: no alcanza con que se envíe un mensaje send al mailSender mockeado, debe ocurrir exactamente 2 veces, e inclusive podría decirle qué parámetros espero recibir.

10. Cobertura

Una vez que tenemos tests, una pregunta que surge es: ¿qué proporción de nuestro sistema está *cubierta por pruebas*? Dicho de otra forma, ¿cuanta funcionalidad de nuestro sistema está siendo validada automáticamente?

Ésta es la noción de cobertura[14]: el porcentaje del código ejecutado por nuestro tests automatizados. Por ejemplo, una cobertura de 60% indicaría que el 60% de las ramas de ejecución del código son evaluadas al correr el conjunto completo de tests.

La cobertura es una métrica a la que típicamente se le da mucha importancia y se la considera una medida de la confianza que se tiene en el código. Aquí hay algunas verdades y algunos mitos: por un lado es cierto que, usualmente, tener niveles de cobertura altos nos dan un cierto grado de confianza en la corrección del diseño y la implementación, que estamos en condiciones de encarar refactors importantes, y que la distancia de la aplicación al caos (robustez) es alta.

Pero, por otro lado, ¿Cuál es, cuantitativamente hablando, un nivel alto de cobertura? Dar un número es un sinsentido, ya que depende de las tecnologías y necesidades concretas de cada proyecto. No sólo la tecnología muchas veces hará imposible obtener 100% de cobertura, sino que a veces no vale la pena. Por otro lado, aún si se tuviera una cobertura del 100% esto no garantizaría el barrido de la totalidad de los escenarios de los casos de uso del sistema, dado que muchos flujos quedan ocultos por los frameworks, que típicamente son dejados fuera del análisis por las herramientas de cobertura.

Otro aspecto importante a considerar es que la cobertura habla de porcentaje de código ejecutado en el tests, pero no de la cantidad o calidad de las aserciones en los mismos. Es por eso que muchas veces existen tests que no agregan cobertura, dado que se solapa con la de otros, pero aún así vale la pena hacerlo, porque valida otros aspectos de la misma porción de código.

Moraleja: la cobertura es más valiosa como heurística que como métrica absoluta.

11. Integración Continua

Consiste en hacer *integraciones automáticas* “en un Entorno Aséptico” (compilar el proyecto y ejecutar los tests) de un proyecto periódicamente (semanales, diariamente, varias veces al día, dependiendo del caso) para poder detectar errores lo antes posible.

[14] http://c2.com/cgi/wiki?CodeCoverage

Cuando decimos hacer integraciones automáticas nos referimos a **compilar el proyecto y ejecutar los tests.**

Dependiendo de la criticidad del proyecto y la cantidad de requerimientos/cambios que tenga convendrá una periodicidad más granular. En general en proyectos activos se hace **diariamente**.

El tema de entorno aséptico es un entorno que está creado para tal fin y tenga las herramientas necesarias para ejecutar el proyecto, y que no existan otros factores que puedan afectar a esta ejecución.

El proceso suele incluir:

- Descargar los programas fuentes desde el Sistema Versionador que utilicemos (Git, SVN, Mercurial,Team Fundation Source Control, u otro).
- Compilar el código descargado generando los ejecutables del proyecto.
- Ejecutar tests diseñados de forma automática.
- Generar informes que servirán para tomar decisiones la mañana siguiente ;). (corregir lo que corresponda, o seguir trabajando tranquilos con nuestra RED DE CONTENCIÓN que son TODOS nuestros tests en VERDE!!!)

Entre las aplicaciones que son utilizadas para IC, dependiendo de la tecnología podemos nombrar las siguientes:

Java: Hudson, Jenkins, CruiseControl, Anthill, Travis

.Net: Team Foundation Build, CruiseControl.net, TeamCity

12. Test-Driven Development

Hace algunos años surgió un esquema de trabajo que consiste en diseñar casos de prueba unitarios *antes de escribir código*. Pensar qué se debería “pedir” que haga el sistema y no cómo resolverlo.

Requisitos:

- Se necesita poder automatizar la prueba unitaria (mediante JUnit, SUnit, etc)

Beneficios:

- Pienso primero en la necesidad (objetivo), desde el punto de vista de quien usa el código, entonces me pongo a pensar qué es lo que quiero, en lugar de tirar código suelto.
- Sólo se escribe el código que se necesita (menor tiempo de codificación a pesar de tener mayor cantidad de líneas de código porque sumo desarrollo + tests). Esto ayuda a la mantenibilidad y legibilidad.
- Facilita las modificaciones posteriores. Al tener el código con tests automatizados se puede verificar que una modificación no afecta la funcionalidad desarrollada anteriormente.

Limitaciones:

- No garantiza calidad de código en sí: debe combinarse con técnicas de revisión de código o *pair programming*. Para utilizar TDD se requiere vasta experiencia en el

desarrollo de software (*seniority*).

- Hay que validar tanto el juego de datos como los tests con alguien que conozca el negocio (se requiere no sólo conocimiento técnico sino también funcional)
- La metodología se complica cuando pasamos a validar la arquitectura del sistema (servicios externos, persistencia, interfaz de usuario). Si hay *efecto*, los tests dejan de ser repetibles: si quiero pagar n veces una factura, una cosa es generar una factura ad-hoc y pagarla y otra cosa es que tome una factura de la base de datos, la pague y modifique el estado de la factura en la base de datos. Para ello, se debe recurrir a la técnica de emulación de objetos, mejor conocida como *mock objects*.
- También es complicado cuando lo que deseo testear es complejo o posee muchas dependencias. **Ejemplo:** un cliente tiene facturas que tienen ítems que tienen productos. Si yo quiero evaluar si un cliente es moroso ni bien lo creo, necesito tener instanciados clientes, facturas, ítems y productos. Si dividí el trabajo entre 3 programadores y quiero evaluar si un cliente es moroso, necesito que los demás programadores terminen de construir las facturas y los productos (con sus correspondientes test unitarios). Si no puedo esperar 1 mes para codificar el test, ¿qué hago? Lo mismo ocurre cuando tengo que comunicarme con una aplicación externa al sistema que estoy desarrollando, o un servidor de tarjeta de crédito, o una base de datos. Este es otro caso en donde conviene utilizar **Mock Objects**.

13. Bibliografía y enlaces

- www.junit.org
- ***Frequently Asked Questions de JUnit:*** http://junit.sourceforge.net/doc/faq/faq.htm
- ***12 razones para escribir test unitarios:*** http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html

- para entornos Java-like: https://code.google.com/p/mockito/, http://www.easymock.org/, JMock, JMockit, PowerMock
- para Ruby y Javascript: Mocha
- para .Net tienen Rhino Mocks, MOQ, NMock/NMock2, entre otros