# Diseño Agil con TDD

Carlos Blé Jurado y colaboradores.
Prologo de José Manuel Beas

Primera Edición, Enero de 2010
www.iExpertos.com



# Índice general

*A la memoria de nuestro querido gatito Lito, que vivió con total atención y entrega cada instante de su vida*

# Prólogo

Érase una vez que se era, un lejano país donde vivían dos cerditos, Pablo y Adrián que, además, eran hermanos. Ambos eran los cerditos más listos de la granja y, por eso, el gallo Iván (el gerente de la misma) organizó una reunión en el establo, donde les encargó desarrollar un programa de ordenador para controlar el almacén de piensos. Les explicó qué quería saber en todo momento: cuántos sacos de grano había y quién metía y sacaba sacos de grano del almacén. Para ello sólo tenían un mes pero les advirtió que, en una semana, quería ya ver algo funcionando. Al final de esa primera semana, eliminaría a uno de los dos.

Adrián, que era el más joven e impulsivo, inmediatamente se puso manos a la obra. "¡No hay tiempo que perder!", decía. Y empezó rápidamente a escribir líneas y líneas de código. Algunas eran de un reciente programa que había ayudado a escribir para la guardería de la vaca Paca. Adrián pensó que no eran muy diferentes un almacén de grano y una guardería. En el primero se guardan sacos y en el segundo, pequeños animalitos. De acuerdo, tenía que retocar algunas cosillas para que aquello le sirviera pero bueno, esto del software va de reutilizar lo que ya funciona, ¿no?

Pablo, sin embargo, antes de escribir una sola línea de código comenzó acordando con Iván dos cosas: qué era exactamente lo que podría ver dentro de una semana y cómo sabría que, efectivamente, estaba terminada cada cosa. Iván quería conocer, tan rápido como fuera posible, cuántos sacos de grano había en cada parte del almacén porque sospechaba que, en algunas partes del mismo, se estaban acumulando sacos sin control y se estaban estropeando. Como los sacos entraban y salían constantemente, no podía saber cuántos había y dónde estaban en cada instante, así que acordaron ir contabilizándolos por zonas y apuntando a qué parte iba o de qué parte venía, cada vez que entrara o saliera un saco. Así, en poco tiempo podrían tener una idea clara del uso que se estaba dando a las distintas zonas del almacén.

Mientras Adrián adelantaba a Pablo escribiendo muchas líneas de código, Pablo escribía primero las pruebas automatizadas. A Adrián eso le parecía una pérdida de tiempo. ¡Sólo tenían una semana para convencer a Iván!

Al final de la primera semana, la demo de Adrián fue espectacular, tenía un control de usuarios muy completo, hizo la demostración desde un móvil y enseñó, además, las posibilidades de un generador de informes muy potente que había desarrollado para otra granja anteriormente. Durante la demostración hubo dos o tres problemillas y tuvo que arrancar de nuevo el programa pero, salvo eso, todo fue genial. La demostración de Pablo fue mucho más modesta, pero cumplió con las expectativas de Iván y el programa no falló en ningún momento. Claro, todo lo que enseñó lo había probado muchísimas veces antes gracias a que había automatizado las pruebas. Pablo hacía TDD, es decir, nunca escribía una línea de código sin antes tener una prueba que le indicara un error. Adrián no podía creer que Pablo hubiera gastado más de la mitad de su tiempo en aquellas pruebas que no hacían más que retrasarle a la hora de escribir las funcionalidades que había pedido Iván. El programa de Adrián tenía muchos botones y muchísimas opciones, probablemente muchas más de las que jamás serían necesarias para lo que había pedido Iván, pero tenía un aspecto "muy profesional".

Iván no supo qué hacer. La propuesta de Pablo era muy robusta y hacía justo lo que habían acordado. La propuesta de Adrián tenía cosillas que pulir, pero era muy prometedora. ¡Había hecho la demostración desde un móvil! Así que les propuso el siguiente trato: "Os pagaré un 50 % más de lo que inicialmente habíamos presupuestado, pero sólo a aquel de los dos que me haga el mejor proyecto. Al otro no le daré nada.". Era una oferta complicada porque, por un lado, el que ganaba se llevaba mucho más de lo previsto. Muy tentador. Pero, por el otro lado, corrían el riesgo de trabajar durante un mes completamente gratis. Mmmmm.

Adrián, tan impulsivo y arrogante como siempre, no dudó ni un instante. "¡Trato hecho!", dijo. Pablo explicó que aceptaría sólo si Iván se comprometía a colaborar como lo había hecho durante la primera semana. A Iván le pareció razonable y les convocó a ambos para que le enseñaran el resultado final en tres semanas.

Adrián se marchó pitando y llamó a su primo Sixto, que sabía mucho y le aseguraría la victoria, aunque tuviera que darle parte de las ganancias. Ambos se pusieron rápidamente manos a la obra. Mientras Adrián arreglaba los defectillos encontrados durante la demo, Sixto se encargó de diseñar una arquitectura que permitiera enviar mensajes desde el móvil hasta un webservice que permitía encolar cualquier operación para ser procesada en paralelo por varios servidores y así garantizar que el sistema estaría en disposición de dar servicio 24 horas al día los 7 días de la semana.

Mientras tanto, Pablo se reunió con Iván y Bernardo (el encargado del almacén) para ver cuáles deberían ser las siguientes funcionalidades a desarrollar. Les pidió que le explicaran, para cada petición, qué beneficio obtenía la granja con cada nueva funcionalidad. Y así, poco a poco, fueron elaborando una lista de funcionalidades priorizadas y resumidas en una serie de tarjetas. A continuación, Pablo fue, tarjeta a tarjeta, discutiendo con Iván y Bernardo cuánto tiempo podría tardar en terminarlas. De paso, aprovechó para anotar algunos criterios que luego servirían para considerar que esa funcionalidad estaría completamente terminada y eliminar alguna ambigüedad que fuera surgiendo. Cuando Pablo pensó que, por su experiencia, no podría hacer más trabajo que el que ya habían discutido, dio por concluida la reunión y se dispuso a trabajar. Antes que nada, resolvió un par de defectos que habían surgido durante la demostración y le pidió a Iván que lo validara. A continuación, se marchó a casa a descansar. Al día siguiente, cogió la primera de las tarjetas y, como ya había hecho durante la semana anterior, comenzó a automatizar los criterios de aceptación acordados con Iván y Bernardo. Y luego, fue escribiendo la parte del programa que hacía que se cumplieran esos criterios de aceptación. Pablo le había pedido ayuda a su amigo Hudson, un coyote vegetariano que había venido desde América a pasar el invierno. Hudson no sabía programar, pero era muy rápido haciendo cosas sencillas. Pablo le encargó que comprobara constantemente los criterios de aceptación que él había automatizado. Así, cada vez que Pablo hacía algún cambio en su programa, avisaba a Hudson y este hacía, una tras otra, todas las pruebas de aceptación que Pablo iba escribiendo. Y cada vez había más. ¡Este Hudson era realmente veloz e incansable!

A medida que iba pasando el tiempo, Adrián y Sixto tenían cada vez más problemas. Terminaron culpando a todo el mundo. A Iván, porque no les había explicado detalles importantísimos para el éxito del proyecto. A la vaca Paca, porque había incluido una serie de cambios en el programa de la guardería que hacía que no pudieran reutilizar casi nada. A los inventores de los SMS y los webservices, porque no tenían ni idea de cómo funciona una granja. Eran tantos los frentes que tenían abiertos que tuvieron que prescindir del envío de SMS y buscaron un generador de páginas web que les permitiera dibujar el flujo de navegación en un gráfico y, a partir de ahí, generar el esqueleto de la aplicación. ¡Eso seguro que les ahorraría mucho tiempo! Al poco, Sixto, harto de ver que Adrián no valoraba sus aportaciones y que ya no se iban a usar sus ideas para enviar y recibir los SMS, decidió que se marchaba, aún renunciando a su parte de los beneficios. Total, él ya no creía que fueran a ser capaces de ganar la competición.

Mientras tanto, Pablo le pidió un par de veces a Iván y a Bernardo que le validaran si lo que llevaba hecho hasta aquel momento era de su agrado y les

hizo un par de demostraciones durante aquellas 3 semanas, lo que sirvió para corregir algunos defectos y cambiar algunas prioridades. Iván y Bernardo estaban francamente contentos con el trabajo de Pablo. Sin embargo, entre ellos comentaron más de una vez: "¿Qué estará haciendo Adrián? ¿Cómo lo llevará?".

Cuando se acercaba la fecha final para entregar el programa, Adrián se quedó sin dormir un par de noches para así poder entregar su programa. Pero eran tantos los defectos que había ido acumulando que, cada vez que arreglaba una cosa, le fallaba otra. De hecho, cuando llegó la hora de la demostración, Adrián sólo pudo enseñar el programa instalado en su portátil (el único sitio donde, a duras penas, funcionaba) y fue todo un desastre: mensajes de error por todos sitios, comportamientos inesperados... y lo peor de todo: el programa no hacía lo que habían acordado con Iván.

Pablo, sin embargo, no tuvo ningún problema en enseñar lo que llevaba funcionando desde hacía mucho tiempo y que tantas veces había probado. Por si acaso, dos días antes de la entrega, Pablo había dejado de introducir nuevas características al programa porque quería centrarse en dar un buen manual de usuario, que Iván había olvidado mencionar en las primeras reuniones porque daba por sentado que se lo entregarían. Claro, Adrián no había tenido tiempo para nada de eso.

Moraleja:

Además de toda una serie de buenas prácticas y un proceso de desarrollo ágil, Pablo hizo algo que Adrián despreció: acordó con Iván (el cliente) y con Bernardo (el usuario) los criterios mediante los cuáles se comprobaría que cada una de las funcionalidades estaría bien acabada. A eso que solemos llamar "criterios de aceptación", Pablo le añadió la posibilidad de automatizar su ejecución e incorporarlos en un proceso de integración continua (que es lo que representa su amigo Hudson en este cuento). De esta manera, Pablo estaba siempre tranquilo de que no estaba estropeando nada viejo con cada nueva modificación. Al evitar volver a trabajar sobre asuntos ya acabados, Pablo era más eficiente. En el corto plazo, las diferencias entre ambos enfoques no parecen significativas, pero en el medio y largo plazo, es evidente que escribir las pruebas antes de desarrollar la solución es mucho más eficaz y eficiente.

En este libro que ahora tienes entre tus manos, y después de este inusual prólogo, te invito a leer cómo Carlos explica bien clarito cómo guiar el desarrollo de software mediante la técnica de escribir antes las pruebas (más conocido como TDD).

# Agradecimientos

Una vez oí a un maestro zen decir que la gratitud que expresa una persona denota su estado momentáneo de bienestar consigo misma. Estoy muy contento de ver que este proyecto que se empezó hace casi año y medio, ha concluido con un resultado tan gratificante.

Tengo que agradecer cosas a miles de personas pero para no extenderme demasiado, lo haré hacia los que han tenido una relación más directa con el libro.

En primer lugar tengo que agradecer a Dácil Casanova que haya sido la responsable de calidad número uno en el proyecto. Sin ella este libro no se hubiera escrito de esta forma y en este momento. Quizás no se hubiese escrito nunca. No sólo tengo que agradecer todo lo muchísimo que me aporta en lo personal sino que además me animó constantemente a no publicar el libro hasta que estuviera hecho lo mejor posible. Ha revisado mi redacción corrigiendo mil faltas de ortografía y signos de puntuación.

El toque de calidad que dan los demás coautores al libro, es también un detallazo. Además de escribir texto han sido buenos revisores. Estoy profundamente agradecido a Juan Gutiérrez Plaza por haber escrito el capítulo 11 y por haber hecho correcciones desde la primera revisión del libro hasta la última. Ha sido un placer discutir juntos tantos detalles técnicos. Gracias a Fran Reyes Perdomo tenemos un gran apéndice sobre Integración Contínua que de no haber sido por él no estaría en el libro, con lo importante que es esta práctica. Mil gracias Fran. Gregorio Mena es el responsable de que el capítulo 1 no sea un completo desastre. Ha sido para mí el más difícil de escribir de todo el libro y a ningún revisor le acababa de convencer. Gregorio ha refactorizado medio capítulo dándole un toque mucho más serio. Espero que sigamos trabajando juntos Gregorio :-) Para terminar con los coautores quiero agradecer a José Manuel Beas que haya escrito el prólogo del libro a pesar de lo muy ocupado que está liderando la comunidad ágil española y

haciendo de padre de familia. Un bonito detalle JM ;-D

A continuación quiero agradecer a la decena de personas que han leído las revisiones del libro previas a la publicación y que han aportado correcciones, palabras de ánimo e incluso texto. Agradecimientos especiales a Esteban Manchado Velázquez que tiene muchísimo que ver con la calidad de la redacción y que ha sido uno de los revisores más constantes. Yeray Darias además de revisar, escribió el capítulo que le pedí sobre DDD, aunque finalmente queda para un libro futuro. Mi buen amigo Eladio López de Luis ha dejado algún comentario en cada revisión que he ido enviando. Alberto Rodríguez Lozano ha sido una de las personas más influyentes en las correcciones del capítulo 9, aportando comentarios técnicos de calidad. No puedo olvidar al resto de revisores que han seguido el libro en distintas etapas aportando también comentarios muy brillantes: Néstor Bethencourt, Juan Jorge Pérez López, Pablo Rodríguez Sánchez, José Ramón Díaz, Jose M. Rodríguez de la Rosa, Víctor Roldán y Néstor Salceda. También agradezco a todas las personas que han leído alguna de estas revisiones previas y me han enviado emails personales de agradecimiento.

Hadi Hariri me influenció mucho para que partiese el libro en dos y dejase para éste, solo aquellos temas de relación directa con TDD.

Ahora, más allá de los que han tenido relación directa con el libro, quiero agradecer a Rodrigo Trujillo, ex director de la Oficina de Software Libre de la Universidad de La Laguna (ULL), que me diera la oportunidad de dirigir un proyecto y carta blanca para aplicar TDD desde el primer día, porque durante el año que trabajé ahí aprendí toneladas de cosas. Rodrigo es una de las personas que más se mueve en la ULL; no deja de intentar abrir puertas a los estudiantes que están terminado ni de preocuparse por la calidad de su universidad.

Gracias también a José Luís Roda, Pedro González Yanes y Jesús Alberto González por abrirme las puertas de la facultad de informática y también por permitirme que impartiese mi primer curso completo de TDD. De la facultad tengo también que agradecer a Marcos Colebrook que me diese el empujón que necesitaba para terminar la carrera, que la tenía casi abandonada. Sin Marcos y Goyo, el cambio de plan hubiera hecho que abandonase la idea de terminar la carrera.

A Esteban Abeledo y al resto del Colegio de Informaticos de Canarias les agradezco mucho hayan homologado nuestros cursos de TDD.

Los alumnos de mis cursos de TDD tienen mucho que ver con el resultado final del libro ya que he volcado en él muchas de sus dudas y comentarios típicos. Gracias a los dos grupos que he tenido en Tenerife y al de Sevilla, todos en 2009. Mención especial a las personas que ayudaron a que saliesen adelante: Álvaro Lobato y los ya citados Gregorio, Pedro y Roda.

Gracias a todos los que están promoviendo XP en España. A saber: Alfredo Casado, Xavier Gost, Leo Antolí, Agustín Yagüe, Eric Mignot, Jorge Jiménez, Iván Párraga, Jorge Uriarte, Jesús Pérez, David Esmerodes, Luismi Cavallé, David Calavera, Ricardo Roldán y tantos otros profesionales de la comunida Agile Spain, entre los cuales están los coautores y revisores de este libro. Y también a los que están promoviendo XP en Latinoamérica: Fabián Figueredo, Carlos Peix, Angel López, Carlos Lone y tantos otros. Y al pibe que vos viste nos rrre-ayudó a traer el Agile Open a España, Xavier Quesada ;-)

Alfredo Casado además ha escrito la sinopsis del libro.

Quiero saludar a todas las personas con las que he trabajado en mi paso por las muchas empresas en que he estado. De todos he aprendido algo. Gracias a los que me han apoyado y gracias también a los que me han querido tapar, porque de todos he aprendido cosas. Ha sido tremendamente enriquecedor cambiar de empresa y de proyectos. Thank you all guys at Buy4Now in Dublin during my year over there in 2007. También he aprendido mucho de todos aquellos desarrolladores con los que he trabajado en proyectos open source, en mi tiempo libre.

A quienes han confiado en mí para impartir cursos de materias diversas, porque han sido claves para que desarrolle mi capacidad docente; Agustín Benito, Academia ESETEC, Armando Fumero, Innova7 y Rodrigo Trujillo.

Agradecimientos también a Pedro Gracia Fajardo, una de las mentes más brillantes que he conocido. Un visionario. Pedro fue quién me habló por primera vez de XP en 2004. En aquel entonces nos creíamos que éramos ágiles aunque en realidad lo estábamos haciendo fatal. La experiencia sirvió para que yo continuase investigando sobre el tema.

Gracias a la comunidad TDD internacional que se presenta en un grupo de discusión de Yahoo. Aunque ellos no leerán estas líneas por ser en español, quiero dejar claro que sin su ayuda no hubiese sabido resolver tantos problemas técnicos. Gracias a Kent Beck y Lasse Koskela por sus libros, que han sido para mí fuente de inspiración.

Aprovecho para felicitar a Roberto Canales por su libro, Informática Profesional[13]. Es una pena que me haya puesto a leerlo cuando ya tenía escrito este libro, a pocos días de terminar el proyecto. Si lo hubiese leído en octubre, cuando Leo me lo regaló, me hubiese ahorrado bastantes párrafos de la parte teórica. Es un pedazo de libro que recomiendo a todo el mundo. Gracias Roberto por brindarnos un libro tan brillante. Un libro, un amigo.

Gracias a Angel Medinilla, Juan Palacio, Xavi Albaladejo y Rodrigo Corral, por entrenar a tantos equipos ágiles en nuestro país. Angel además me regala muy buenos consejos en mi nueva etapa en iExpertos.

Por último no quiero dejar de decirle a mi familia que sin ellos esto

no sería posible. A Dácil de nuevo por todo lo muchísimo que me aporta diariamente. A mis padres por traerme al mundo y enseñarme tantas cosas. A mis hermanas por querer tanto a su hermano mayor. A mis primos. A mi tía Tina por acogerme en su casa durante mis años de universidad y darme la oportunidad de estudiar una carrera, ya que mis padres no podían costearmelo. Ella me ayuda siempre que lo necesito. A mis abuelos por estar siempre ahí, como mis segundos padres.

# Autores del libro

**Carlos Blé Jurado**

Nací en Córdoba en 1981, hijo de cordobeses pero cuando tenía 4 años emigramos a Lanzarote y, salvo algunos años intercalados en los que viví en Córdoba, la mayor parte de mi vida la he pasado en Canarias. Primero en Lanzarote y después en Tenerife.
Mi primer apellido significa *trigo* en francés. Lo trajo un francés al pueblo cordobés de La Victoria en tiempos de Carlos III.
Cuando tenía 6 años mi padre trajo a casa un 8086 y aquello me fascinó tanto que supe que me quería dedicar a trabajar con ordenadores desde entonces. He tenido también Amiga 500, Amiga 1200, y luego unos cuantos PC, desde el AMD K6 hasta el Intel Core Duo de hoy.

Soy ingeniero técnico en informática de sistemas. Para mí el título no es ninguna garantía de profesionalidad, más bien hago un balance negativo de mi paso por la Universidad, pero quería ponerlo aquí para que los que padecen de titulitis vean que el que escribe es titulado.

La primera vez que gané dinero con un desarrollo de software fue en 2001. Poco después de que instalase Debian GNU/Linux en mi PC. En 2003 entré de lleno en el mercado laboral. Al principio trabajé administrando sistemas además de programar.

En 2005 se me ocurrió la genial idea de montar una empresa de desarrollo de software a medida con mis colegas. Sin tener suficiente experiencia como desarrollador y ninguna experiencia como empresario ni comercial, fue toda

una aventura sobrevivir durante los dos años que permanecí en la empresa. Fueron dos años equivalentes a cuatro o cinco trabajando en otro sitio. Ver el negocio del software desde todos los puntos de vista, me brindó la oportunidad de darme cuenta de muchas cosas y valorar cuestionas que antes no valoraba. Aprendí que no podía tratar al cliente como si fuera un tester. No podía limitarme a probar las aplicaciones 10 minutos a mano antes de entregarlas y pensar que estaban hechas. Aprendí que la venta era más decisiva que el propio software. También aprendí cosas feas como que Hacienda no somos todos, que los concursos públicos se amañan, que el notario da más miedo que el dentista, que el pez grande se come al pequeño y muchas otras. Al final me dí cuenta de que la odisea me sobrepasaba y no era capaz de llevar la empresa a una posición de estabilidad donde por fín dejase de amanecerme sentado frente a la pantalla. Cansado de los morosos y de que la administración pública nos tuviera sin cobrar meses y meses, mientras estaba con el agua al cuello, en 2007 me mandé a mudar a Irlanda para trabajar como desarrollador. Aprendí lo importante que era tener un equipo de QA[1] y me volqué con los tests automatizados. Llegué a vivir el *boom* de los sueldos en Dublin, cobrando 5000 euros mensuales y sin hacer ni una sola hora extra. En 2008 regresé a Tenerife.

En la actualidad he vuelto a emprender. Desarrollo software profesionalmente de manera vocacional. Mi espíritu emprendedor me lleva a poner en marcha nuevas ideas en la nube. Además me dedico a formar a desarrolladores impartiendo cursos sobre TDD, código limpio, metodología y herramientas de programación. En lugar de intentar trabajar *en mi* empresa, trabajo *para la* empresa[2], cuya web es www.iExpertos.com

Habitualmente escribo en *www.carlosble.com* y en el blog de iExpertos.

He escrito la mayor parte del libro con la excepción de los fragmentos que nos han regalado los demás autores.

[1] Quality Assurance

[2] El matiz viene del libro, El Mito del Emprendedor, de Michael E. Gerber

**Jose Manuel Beas**

En 2008 decidió aprovechar sus circunstancias personales para tomarse un respiro, mirar atrás, a los lados y, sobre todo, hacia adelante. Y así, aprovechó ese tiempo para mejorar su formación en temas como Scrum asistiendo al curso de Angel Medinilla. Pero también quiso poner su pequeño granito de arena en el desarrollo y la difusión de Concordion, una herramienta de código abierto para realizar pruebas de aceptación, y rellenar su "caja de herramientas" con cosas como Groovy y Grails. Pero sobre todo vió que merecía la pena poner parte de su energía en revitalizar una vieja iniciativa llamada Agile Spain y buscar a todos aquellos que, como él, estuvieran buscando maneras mejores de hacer software. Y vaya que si los encontró. Actualmente es el Presidente de la asociación Agile Spain, que representa a la comunidad agilista en España y organizadora de los Agile Open Spain y las Conferencias Agile Spain. También participa en la elaboración de los podcasts de Podgramando.es, una iniciativa de "agilismo.es powered by iExpertos.com". Puedes localizarlo fácilmente a través del portal agilismo.es, en la lista de correo de Agile Spain o en su blog personal *http://jmbeas.iexpertos.com*.

**Juan Gutiérrez Plaza**

Escribir, he escrito poco, pero aprender, muchísimo. He intentado hacer algo con sentido con mi oxidado Python en el capítulo 11 aunque más que escribir, ha sido discutir y rediscutir con Carlos sobre cual era la mejor forma de hacer esto y aquello (siempre ha ganado él). También he sacado del baúl de los recuerdos mi LaTeX y he revisado mucho. Cuando no estoy discutiendo con la gente de Agile Spain, trabajo como "Agile Coach" en F-Secure donde intento ayudar a equipos de Finlandia, Malasia, Rusia y Francia en su transición a las metodologías ágiles (tanto en gestión como en prácticas de software entre las que se incluye, por supuesto, TDD). ¿Pero cómo he llegado hasta aquí? Mi devoción los ordenadores me llevó a estudiar la carrera de ingeniería en informática en la UPM de Madrid. Trabajé en España por un tiempo antes de decidir mudarme a Finlandia en el 2004 para trabajar en Nokia. En las largas y oscuras "tardes" del invierno Finlandés estudié un master en "industrial management" antes de cambiar a F-Secure. Cuando el tiempo lo permite, escribo en *http://agilizar.es*

**Fran Reyes Perdomo**

Soy un apasionado desarrollador de software interesado en "prácticas ágiles". Llevo cerca de 4 años trabajando para la rígida Administración pública con un fantastico equipo. Conocí a Carlos Blé en un provechoso curso de TDD que impartió para un grupo de compañeros. Entre cervezas (una fase importante para asimilar lo aprendido), compartimos ideas y experiencias del mundo del software, y me habló además del proyecto en el que se encontraba embarcado, en el cual me brindó la oportunidad de participar con un pequeño apéndice sobre integración continua. Una práctica, que intentamos, forme parte del día a día en nuestros proyectos. *http://es.linkedin.com/in/franreyesperdomo*

**Gregorio Mena**

Mi corta vida profesional ha sido suficiente para dar sentido a la frase de Horacio "Ningún hombre ha llegado a la excelencia en arte o profesión alguna, sin haber pasado por el lento y doloroso proceso de estudio y preparación". Aunque en mi caso el camino no es doloroso, sino apasionante. Siguiendo esta filosofía, intento formarme y fomentar la formación, por lo que he organizado un curso de TDD impartido con gran acierto por Carlos Ble y voy a participar en futuras ediciones. Trabajo desde iExpertos para que entre todos hagamos posible el primer curso de Scrum en Canarias, pues también colaboro con la plataforma ScrumManager. Ha sido esta forma de ver nuestra profesión la que me llevó a colaborar con Carlos en este libro. Pensaba aportar algo, pero lo cierto es que lo que haya podido aportar no tiene comparación con lo que he tenido la suerte de recibir. Por ello debo dar a Carlos las gracias por ofrecerme esta oportunidad y por el esfuerzo que ha hecho para que este libro sea una realidad para todos los que vemos en la formación continua el camino al éxito.
Habitualmente escribo en *http://eclijava.blogspot.com*.

# Convenciones y Estructura

Este libro contiene numerosos bloques de código fuente en varios lenguajes de programación. Para hacerlos más legibles se incluyen dentro de rectángulos que los separan del resto del texto. Cuando se citan elementos de dichos bloques o sentencias del lenguaje, se usa `este_tipo_de_letra`.

A lo largo del texto aparecen referencias a sitios web en el pie de página. Todas ellas aparecen recopiladas en la página web del libro.

Las referencias bibliográficas tienen un formato como este:[3]. Las últimas páginas del libro contienen la bibliografía detallada correspondiente a todas estas referencias.

Otros libros de divulgación repiten determinadas ideas a lo largo de varios capítulos con el fin de reforzar los conceptos. En la era de la infoxicación[3] tal repetición sobra. He intentado minimizar las repeticiones de conceptos para que el libro se pueda revisar rápidamente, por lo que es recomendable una segunda lectura del mismo para quienes desean profundizar en la materia. Será como ver una película dos veces: en el segundo pase uno aprecia muchos detalles que en el primero no vio y su impresión sobre el contenido puede variar. En realidad, es que soy tan mal escritor que, si no lee el libro dos veces no lo va a entender. Hágase a la idea de que tiene 600 páginas en lugar de 300.

Sobre los paquetes de código fuente que acompañan a este libro (listo para descarga en la web), el escrito en C# es un proyecto de *Microsoft Visual Studio 2008 (Express Edition)* y el escrito en Python es una carpeta de ficheros.

[3] http://es.wiktionary.org/wiki/infoxicaci%C3%B3n

# La Libertad del Conocimiento

El conocimiento ha sido transmitido de individuo a individuo desde el comienzo mismo de la vida en el planeta. Gracias a la libre circulación del conocimiento hemos llegado a donde estamos hoy, no sólo en lo referente al avance científico y tecnológico, sino en todos los campos del saber.

Afortunadamente, el conocimiento no es propiedad de nadie sino que es como la energía; simplemente fluye, se transforma y nos transforma. De hecho no pertenece a las personas, no tiene dueño, es de todos los seres. Observando a los animales podemos ver cómo los padres enseñan a su prole las técnicas que necesitarán para desenvolverse en la vida.

El conocimiento contenido en este libro no pertenece al autor. No pertenece a nadie en concreto. La intención es hacerlo llegar a toda persona que desee aprovecharlo.

En Internet existe mucha documentación sobre la libertad del conocimiento, empezando por la entrada de la Wikipedia[4]. Este argumento es uno de los principales pilares del software libre[5], al que tanto tengo que agradecer. Las principales herramientas que utilizaremos en este libro están basadas en software libre: frameworks xUnit, sistemas de control de versiones y frameworks para desarrollo de software. Personalmente me he convertido en usuario de la tecnología Microsoft .Net gracias al framework Mono[6], desarrollado por Novell con licencia libre. De no haber sido por Mono probablemente no hubiera conocido C#.

El libro ha sido maquetado usando LaTeX, concretamente con teTex y MikTeX(software libre) y ha requerido de multitud de paquetes libres desarro-

[4] http://es.wikipedia.org/wiki/Conocimiento_libre
[5] http://es.wikipedia.org/wiki/Código_libre
[6] http://www.mono-project.com

llados por la comunidad. Para la edición del texto se ha usado Texmaker, Dokuwiki, Vim y Emacs. El versionado de los ficheros de texto se ha llevado a cabo con Subversion. Los diagramas de clases los ha generado SharpDevelop, con el cual también he editado código. Estoy muy agradecido a todos los que han escrito todas estas piezas. En la web del libro se encuentra el esqueleto con el que se ha maquetado para que, quien quiera, lo use para sus propias publicaciones.

Pese a mi simpatía por el software de fuente abierta, este libro va más allá de la dicotomía software libre/software propietario y se centra en técnicas aplicables a cualquier lenguaje de programación en cualquier entorno. Uno de los peores enemigos del software libre es el fanatismo radical de algunos de sus evangelizadores, que raya en la mala educación y empaña el buen hacer de los verdaderos profesionales. Es mi deseo aclarar que mi posición personal no es ni a favor ni en contra del software propietario, simplemente me mantengo al margen de esa contienda.

# La Web del Libro

Los enlaces a sitios web de Internet permanecen menos tiempo activos en la red que en las páginas de un libro impreso; la lista de referencias se mantendrá actualizada en un sitio web dedicado al libro:

http://www.dirigidoportests.com

Si el lector desea participar informando de enlaces rotos, podrá hacerlo dirigiéndose a la web del libro o bien mediante correo electrónico al autor:

carlos[Arroba]iExpertos[Punto]com

El código fuente que acompaña al libro se podrá descargar en la misma web.

Si bien es cierto que escribir el libro ha sido un placer, también es cierto que ha sido duro en ocasiones. Ha supuesto casi año y medio de trabajo y, dado que el libro es gratis y ni siquiera su venta en formato papel se traducirá en algo de beneficio, en la web es posible hacer donaciones. Si el libro le gusta y le ayuda, le invito a que haga una donación para que en el futuro puedan haber más libros libres como este.

# ¿Cuál es el Objetivo del Libro?

El objetivo fundamental del libro es traer a nuestro idioma, el español, conocimiento técnico que lleva años circulando por países de habla inglesa. No cabe duda de que los que inventaron la computación y el software nos siguen llevando ventaja en cuanto a conocimiento se refiere. En mi opinión, es una cuestión cultural en su mayor parte pero, sea como fuere, no podemos perder la oportunidad de subirnos al carro de las nuevas técnicas de desarrollo y difundir el conocimiento proporcionado por nuestros compañeros angloparlantes. Sólo así competiremos en igualdad de condiciones, porque a día de hoy cada vez más clientes apuestan por el outsourcing. Conocimiento es poder.

A día de hoy, por suerte o por desgracia, no nos queda más remedio que dominar el inglés, al menos el inglés técnico escrito, y es conveniente leer mucho en ese idioma. Se aprende muchísimo porque no sólo lo usan los nativos de habla inglesa sino que se ha convertido en el idioma universal en cuanto a tecnología. Sin embargo, reconozco que yo mismo hace unos años era muy reacio a leer textos en inglés. He tenido que hacer un gran esfuerzo y leer mucho con el diccionario en la mano hasta llegar al punto en que no me cuesta trabajo leer en inglés (además de vivir una temporada fuera de España). Conozco a pocos compañeros que hagan este esfuerzo. Es comprensible y normal que la mayoría se limite a leer lo que está documentado en español. Al fin y al cabo es de los idiomas más hablados del planeta. Por eso concluyo que hay que traer la información a nuestro idioma para llegar a más gente, aunque el buen profesional deberá tener presente las múltiples ventajas que le aportará el dominio del inglés escrito. Cuando no dominamos el inglés nos perdemos muchos matices que son significativos[7].

Apenas hay libros sobre agilismo en español. Los únicos libros novedosos que se editan en nuestra lengua relacionados con el mundo del software,

[7] http://jmbeas.iexpertos.com/hablar-ingles-es-facil-si-sabes-como/

tratan sobre tecnologías muy específicas que hoy valen y mañana quizás no. Está muy bien que haya libros sobre Java, sobre .Net o sobre Ruby, pero no tenemos que limitarnos a ello. El único libro sobre agilismo que hay a día de hoy es el de Scrum, de Juan Palacio[15]. Por otro lado, Angel Medinilla ha traducido el libro de Henrik Kniberg[8], *Scrum y XP desde las Trincheras* y Leo Antolí ha traducido *The Scrum Primer*. Estos regalos son de agradecer.

Ahora que existen editoriales en la red tales como *Lulu.com*, ya no hay excusa para no publicar contenidos técnicos. Personalmente me daba reparo afrontar este proyecto sin saber si alguna editorial querría publicarlo pero ya no es necesario que las editoriales consideren el producto comercialmente válido para lanzarlo a todos los públicos. Otro objetivo del libro es animar a los muchos talentos hispanoparlantes que gustan de compartir con los demás, a que nos deleiten con su conocimiento y sus dotes docentes. ¿Quién se anima con un libro sobre Programación Extrema?.

No me cabe duda de que las ideas planteadas en este libro pueden resultarles controvertidas y desafiantes a algunas personas. El lector no tiene por qué coincidir conmigo en todo lo que se expone; tan sólo le invito a que explore con una mente abierta las técnicas aquí recogidas, que las ponga en práctica y después decida si le resultan o no valiosas.

# Parte I

# Base Teórica

Capítulo 1

# El Agilismo

Para definir qué es el agilismo, probablemente basten un par de líneas. Ya veremos más adelante, en este mismo capítulo, que el concepto es realmente simple y queda plasmado en cuatro postulados muy sencillos. Pero creo que llegar a comprenderlo requiere un poco más de esfuerzo y, seguramente, la mejor manera sea haciendo un repaso a la historia del desarrollo de software, para al final ver como el agilismo no es más que una respuesta lógica a los problemas que la evolución social y tecnológica han ido planteando.

Ya desde el punto de partida es necesario hacer una reflexión. Al otear la historia nos damos cuenta de que el origen del desarrollo de software está a unas pocas décadas de nosotros. Para llegar al momento en el que el primer computador que almacenaba programas digitalmente corrió exitosamente su primer programa, sólo tenemos que remontarnos al verano de 1948[1]. Esto nos hace reflexionar sobre el hecho de que nos encontramos ante una disciplina que es apenas una recién nacida frente a otras centenarias con una base de conocimiento sólida y estable. Por nuestra propia naturaleza nos oponemos al cambio, pero debemos entender que casi no ha transcurrido tiempo como para que exijamos estabilidad.

Siguiendo la ley de Moore[2], los componentes hardware acaban duplicando su capacidad cada año. Con lo que, en muy poco tiempo, aparecen máquinas muy potentes capaces de procesar miles de millones de operaciones en segundos. A la vez, los computadores van reduciendo su tamaño considerablemente, se reducen los costes de producción del hardware y avanzan las comunicaciones entre los sistemas. Todo esto tiene una consecuencia evidente: los computadores ya no sólo se encuentran en ámbitos muy restringidos, como el militar o el científico.

[1] http://en.wikipedia.org/wiki/Tom_Kilburn
[2] http://en.wikipedia.org/wiki/Moore%27s_Law

Al extenderse el ámbito de aplicación del hardware (ordenadores personales, juegos, relojes, ...), se ofrecen soluciones a sistemas cada vez más complejos y se plantean nuevas necesidades a una velocidad vertiginosa que implican a los desarrolladores de Software. Sin información y conocimiento suficiente, unos pocos "aventureros" empiezan a desarrollar las primeras aplicaciones que dan respuesta a las nuevas necesidades pero es un reto muy complejo que no llega a ser resuelto con la inmediatez y la precisión necesarias. Los proyectos no llegan a buen puerto, o lo hacen muy tarde.

En la década de los cincuenta nos encontramos con otro hito importante. En el ámbito militar, surge la necesidad de profesionalizar la gestión de proyectos para poder abordar el desarrollo de complejos sistemas que requerían coordinar el trabajo conjunto de equipos y disciplinas diferentes en la construcción de sistemas únicos. Posteriormente, la industria del automóvil siguió estos pasos. Esta nueva disciplina se basa en la planificación, ejecución y seguimiento a través de procesos sistemáticos y repetibles.

Hasta este punto, hemos hablado sólo de desarrollo de software y no de ingeniería de software, ya que es en 1968 cuando se acuña este término en la NATO Software Engineering Conference[3].En esta conferencia también se acuña el término crisis del software para definir los problemas que estaban surgiendo en el desarrollo y que hemos comentado anteriormente.

Los esfuerzos realizados producen tres áreas de conocimiento que se revelaron como estratégicas para hacer frente a la crisis del software[4]:

- Ingeniería del software: este término fue acuñado para definir la necesidad de una disciplina científica que, como ocurre en otras áreas, permita aplicar un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

- Gestión Predictiva de proyectos: es una disciplina formal de gestión, basada en la planificación, ejecución y seguimiento a través de procesos sistemáticos y repetibles.

- Producción basada en procesos: se crean modelos de procesos basados en el principio de Pareto[5], empleado con buenos resultados en la producción industrial. Dicho principio nos indica que la calidad del resultado depende básicamente de la calidad de los procesos.

En este punto, con el breve recorrido hecho, podemos sacar conclusiones reveladoras que luego nos llevarán a la mejor comprensión del agilismo. Por un lado, la gestión predictiva de proyectos establece como criterios de éxito

[3] http://en.wikipedia.org/wiki/Software_engineering
[4] http://www.navegapolis.net/files/Flexibilidad_con_Scrum.pdf
[5] http://es.wikipedia.org/wiki/Principio_de_Pareto

obtener el producto definido en el tiempo previsto y con el coste estimado. Para ello, se asume que el proyecto se desarrolla en un entorno estable y predecible. Por otro, se empiezan a emular modelos industriales e ingenieriles que surgieron en otros ámbitos y con otros desencadenantes.

Debemos tener en cuenta que, al principio, el tiempo de vida de un producto acabado era muy largo; durante este tiempo, generaba beneficios a las empresas, para las que era más rentable este producto que las posibles novedades pero, a partir de los ochenta, esta situación empieza a cambiar. La vida de los productos es cada vez más corta y una vez en el mercado, son novedad apenas unos meses, quedando fuera de él enseguida. Esto obliga a cambiar la filosofía de las empresas, que se deben adaptar a este cambio constante y basar su sistema de producción en la capacidad de ofrecer novedades de forma permanente.

Lo cierto es que ni los productos de software se pueden definir por completo a priori, ni son totalmente predecibles, ni son inmutables. Además, los procesos aplicados a la producción industrial no tienen el mismo efecto que en desarrollo de software, ya que en un caso se aplican sobre máquinas y en otro, sobre personas. Estas particularidades tan características del software no tuvieron cabida en la elaboración del modelo más ampliamente seguido hasta el momento: El modelo en cascada. En el siguiente punto de este capítulo veremos una breve descripción de dicho modelo, para entender su funcionamiento y poder concluir por qué en determinados entornos era preciso un cambio. Como ya comentaba al principio, el objetivo es ver que el agilismo es la respuesta a una necesidad.

## 1.1. Modelo en cascada

Este es el más básico de todos los modelos[6] y ha servido como bloque de construcción para los demás paradigmas de ciclo de vida. Está basado en el ciclo convencional de una ingeniería y su visión es muy simple: el desarrollo de software se debe realizar siguiendo una secuencia de fases. Cada etapa tiene un conjunto de metas bien definidas y las actividades dentro de cada una contribuyen a la satisfacción de metas de esa fase o quizás a una subsecuencia de metas de la misma. El arquetipo del ciclo de vida abarca las siguientes actividades:

- Ingeniería y Análisis del Sistema: Debido a que el software es siempre parte de un sistema mayor, el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.

[6] Bennington[4], Pag. 26-30

- Análisis de los requisitos del software: el proceso de recopilación de los requisitos se centra e intensifica especialmente en el software. El ingeniero de software debe comprender el ámbito de la información del software así como la función, el rendimiento y las interfaces requeridas.

- Diseño: el diseño del software se enfoca en cuatro atributos distintos del programa; la estructura de los datos, la arquitectura del software, el detalle procedimental y la caracterización de la interfaz. El proceso de diseño traduce los requisitos en una representación del software con la calidad requerida antes de que comience la codificación.

- Codificación: el diseño debe traducirse en una forma legible para la maquina. Si el diseño se realiza de una manera detallada, la codificación puede realizarse mecánicamente.

- Prueba: una vez que se ha generado el código comienza la prueba del programa. La prueba se centra en la lógica interna del software y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

- Mantenimiento: el software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán debidos a que se haya encontrado errores, a que el software deba adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos) o a que el cliente requiera ampliaciones funcionales o del rendimiento.

En el modelo vemos una ventaja evidente y radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software. Pero el modelo se aplica en un contexto, así que debemos atender también a él y saber que:

- Los proyectos reales raramente siguen el flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma.

- Normalmente, al principio, es difícil para el cliente establecer todos los requisitos explícitamente. El ciclo de vida clásico lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos productos.

- El cliente debe tener paciencia. Hasta llegar a las etapas finales del proyecto no estará disponible una versión operativa del programa. Un error importante que no pueda ser detectado hasta que el programa esté funcionando, puede ser desastroso.

## 1.2. Hablemos de cifras

Quizás nos encontremos en un buen punto para dejar de lado los datos teóricos y centrarnos en cifras reales que nos indiquen la magnitud del problema que pretendemos describir. Para ello nos basaremos en los estudios realizados por un conjunto de profesionales de Massachussets que se unió en 1985 bajo el nombre de Standish Group[7]. El objetivo de estos profesionales era obtener información de los proyectos fallidos en tecnologías de la información (IT) y así poder encontrar y combatir las causas de los fracasos. El buen hacer de este grupo lo ha convertido en un referente, a nivel mundial, sobre los factores que inciden en el éxito o fracaso de los proyectos de IT. Factores que se centran, fundamentalmente, en los proyectos de software y se aplican tanto a los desarrollos como a la implementación de paquetes (SAP, Oracle, Microsoft, etc.)

A lo largo del tiempo, el Standish Group reveló 50.000 proyectos fallidos y en 1994 se obtuvieron los siguientes resultados:

- Porcentaje de proyectos que son cancelados: 31 %
- Porcentaje de proyectos problemáticos: 53 %
- Porcentaje de proyectos exitosos: 16 % (pero estos sólo cumplieron, en promedio, con el 61 % de la funcionalidad prometida)

Atendiendo a estos resultados poco esperanzadores, durante los últimos diez años, la industria invirtió varios miles de millones de dólares en el desarrollo y perfeccionamiento de metodologías y tecnologías (PMI, CMMI, ITIL, etc.). Sin embargo, en 2004 los resultados seguían sin ser alentadores:

- Porcentaje de proyectos exitosos: crece hasta el 29 %.
- Porcentaje de proyectos fracasados: 71 %.

Según el informe de Standish, las diez causas principales de los fracasos, por orden de importancia, son:

- Escasa participación de los usuarios
- Requerimientos y especificaciones incompletas
- Cambios frecuentes en los requerimientos y especificaciones
- Falta de soporte ejecutivo
- Incompetencia tecnológica

[7] http://www.standishgroup.com

- Falta de recursos
- Expectativas no realistas
- Objetivos poco claros
- Cronogramas irreales
- Nuevas tecnologías

Cabe destacar de estos resultados que siete de los factores nombrados, son factores humanos. Las cifras evidencian la existencia de un problema, al que, como veremos a continuación, el agilismo intenta dar respuesta.

En el libro de Roberto Canales[13] existe más información sobre los métodos en cascada, las metodologías ágiles y la gestión de proyectos.

## 1.3. El manifiesto ágil

Hasta ahora hemos visto que quizás para algunos proyectos se esté realizando un esfuerzo vano e incluso peligroso: intentar aplicar prácticas de estimación, planificación e ingeniería de requisitos. No es conveniente pensar que estas prácticas son malas en sí mismas o que los fracasos se deben a una mala aplicación de estas, sino que deberíamos recapacitar sobre si estamos aplicando las prácticas adecuadas.

En 2001, 17 representantes de nuevas metodologías y críticos de los modelos de mejora basados en procesos se reunieron, convocados por Kent Beck, para discutir sobre el desarrollo de software. Fue un grito de ¡basta ya! a las prácticas tradicionales. Estos profesionales, con una dilatada experiencia como aval, llevaban ya alrededor de una década utilizando técnicas que les fueron posicionando como líderes de la industria del desarrollo software. Conocían perfectamente las desventajas del clásico modelo en cascada donde primero se analiza, luego se diseña, después se implementa y, por último (en algunos casos), se escriben algunos tests automáticos y se martiriza a un grupo de personas para que ejecuten manualmente el software, una y otra vez hasta la saciedad. El manifiesto ágil[8] se compone de cuatro principios. Es pequeño pero bien cargado de significado:

[8]La traducción del manifiesto es de Agile Spain http://www.agile-spain.com/manifiesto_agil

Estamos descubriendo mejores maneras de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de esta experiencia hemos aprendido a valorar:

- **Individuos e interacciones** sobre *procesos y herramientas*
- **Software que funciona** sobre *documentación exhaustiva*
- **Colaboración con el cliente** sobre *negociación de contratos*
- **Responder ante el cambio** sobre *seguimiento de un plan*

Esto es, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.

Tras este manifiesto se encuentran 12 principios de vital importancia para entender su filosofía[9]:

- Nuestra máxima prioridad es satisfacer al cliente a través de entregas tempranas y continuas de software valioso.
- Los requisitos cambiantes son bienvenidos, incluso en las etapas finales del desarrollo. Los procesos ágiles aprovechan al cambio para ofrecer una ventaja competitiva al cliente.
- Entregamos software que funciona frecuentemente, entre un par de semanas y un par de meses. De hecho es común entregar cada tres o cuatro semanas.
- Las personas del negocio y los desarrolladores deben trabajar juntos diariamente a lo largo de todo el proyecto.
- Construimos proyectos entorno a individuos motivados. Dándoles el lugar y el apoyo que necesitan y confiando en ellos para hacer el trabajo.
- El método más eficiente y efectivo de comunicar la información hacia y entre un equipo de desarrollo es la conversación cara a cara.

[9]Traducción libre de los principios publicados en http://www.agilemanifesto.org/principles.html

- La principal medida de avance es el software que funciona.
- Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben poder mantener un ritmo constante.
- La atención continua a la excelencia técnica y el buen diseño mejora la agilidad.
- La simplicidad, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de la autoorganización de los equipos.
- A intervalos regulares, el equipo reflexiona sobre cómo ser más eficaces, a continuación mejoran y ajustan su comportamiento en consecuencia.

Este libro no pretende abarcar el vasto conjunto de técnicas y metodologías del agilismo pero, considerando la poca literatura en castellano que existe actualmente sobre este tema, merece la pena publicar el manifiesto.

## 1.4. ¿En qué consiste el agilismo?: Un enfoque práctico

El agilismo es una respuesta a los fracasos y las frustraciones del modelo en cascada. A día de hoy, las metodologías ágiles de desarrollo de software están en boca de todos y adquieren cada vez más presencia en el mundo hispano, si bien llevan siendo usadas más de una década en otros países. El abanico de metodologías ágiles es amplio, existiendo métodos para organizar equipos y técnicas para escribir y mantener el software. Personalmente, me inclino hacia la Programación Extrema (eXtreme Programming, XP) como forma de atacar la implementación del producto y hacia Scrum como forma de gestionar el proyecto, pero el estudio de ambas en su totalidad queda fuera del alcance de este libro.

Por ilustrarlo a modo de alternativa al modelo en cascada: podemos gestionar el proyecto con Scrum y codificar con técnicas de XP; concretamente TDD[10] y Programación por Parejas[11], sin olvidar la propiedad colectiva del código y la Integración Continua[12].

[10] Test Driven Development o Desarrollo Dirigido por Test

[11] Pair Programming o Programación por Parejas, es otra de las técnicas que componen XP y que no vamos a estudiar en detalle en este libro. Véanse en la bibliografía los textos relacionados con XP para mayor información

[12] Véase el apéndice sobre Integración Continua al final del libro

Agilismo no es perfeccionismo, es más, el agilista reconoce que el software es propenso a errores por la naturaleza de quienes lo fabrican y lo que hace es tomar medidas para minimizar sus efectos nocivos desde el principio. No busca desarrolladores perfectos sino que reconoce que los humanos nos equivocamos con frecuencia y propone técnicas que nos aportan confianza a pesar ello. La automatización de procesos es uno de sus pilares. La finalidad de los distintos métodos que componen el agilismo es reducir los problemas clásicos de los programas de ordenador, a la par que dar más valor a las personas que componen el equipo de desarrollo del proyecto, satisfaciendo al cliente, al desarrollador y al analista de negocio.

El viejo modelo en cascada se transforma en una noria que, a cada vuelta (iteración), se alimenta con nuevos requerimientos o aproximaciones más refinadas de los ya abordados en iteraciones anteriores, puliendo además los detalles técnicos (no resolviendo defectos sino puliendo). Al igual que en el modelo tradicional, existen fases de análisis, desarrollo y pruebas pero, en lugar de ser consecutivas, están solapadas. Esta combinación de etapas se ejecuta repetidas veces en lo que se denominan iteraciones. Las iteraciones suelen durar de dos a seis semanas y en cada una de ellas se habla con el cliente para analizar requerimientos, se escriben pruebas automatizadas, se escriben líneas de código nuevas y se mejora código existente. Al cliente se le enseñan los resultados después de cada iteración para comprobar su aceptación e incidir sobre los detalles que se estimen oportunos o incluso reajustar la planificación. No es casual que hayamos situado las pruebas automáticas antes de la escritura de nuevo código ya que, como veremos en este libro, dentro del agilismo se contempla una técnica en la que las pruebas son una herramienta de diseño del código (TDD) y, por tanto, se escriben antes que el mismo. Llegado el caso las pruebas se consideran ejemplos, requerimientos que pueden ser confirmados (o validados) por una máquina (validación automatizada).

Todo el equipo trabaja unido, formando una piña[13] y el cliente es parte de ella, ya no se le considera un oponente. La estrategia de juego ya no es el control sino la colaboración y la confianza. Del control se encargarán los procesos automáticos que nos avisarán de posibles problemas o puntos a mejorar. La jerarquía clásica (director técnico, analista de negocio, arquitecto, programador senior, junior ...) pierde sentido y los roles se disponen sobre un eje horizontal en lugar de vertical, donde cada cual cumple su cometido pero sin estar por encima ni por debajo de los demás. En lugar de trabajar por horas, trabajamos por objetivos y usamos el tiempo como un recurso más y no como un fin en sí mismo (lo cual no quiere decir que no existan fechas de

[13]de ahí el nombre de Scrum, que se traduce por Melé, palabra del argot del Rugby usada para designar la unión de los jugadores en bloque

entrega para cada iteración).

La esencia del agilismo es la habilidad para adaptarse a los cambios. Ejecutando las diversas técnicas que engloba, con la debida disciplina, se obtienen resultados satisfactorios sin lugar para el caos.

En cualquier método ágil, los equipos deben ser pequeños, típicamente menores de siete personas. Cuando los proyectos son muy grandes y hace falta más personal, se crean varios equipos. Nos encontramos ante el famoso "divide y vencerás".

El análisis no es exhaustivo ni se dilata indefinidamente antes de empezar la codificación, sino que se acota en el tiempo y se encuadra dentro de cada iteración y es el propio progreso de la implementación el que nos ayuda a terminar de descubrir los "pormenores". En el análisis buscamos cuáles son las historias de usuario y, las ambigüedades que puedan surgir, se deshacen con ejemplos concisos en forma de tests automáticos. Hablaremos sobre las historias de usuario en el capítulo de ATDD. Dichas historias contienen los requisitos de negocio y se ordenan por prioridad según las necesidades del cliente, a fin de desarrollar antes unas u otras.

Cada requisito debe implementarse en un máximo de una semana para que, al final de la iteración, el cliente pueda ver funcionalidad con valor de negocio.

El analista de negocio adoptará el rol de *dueño del producto* cuando el cliente no pueda participar tan frecuentemente como nos gustaría y cambiará los cientos de páginas de documentación en prosa por tests de aceptación[14] lo suficientemente claros como para que el cliente los apruebe y la máquina los valide. También se encargará de priorizar el orden de implementación de los requisitos acorde a lo que se hable en las reuniones con el cliente. Los desarrolladores estarán en contacto diario con los analistas para resolver cualquier duda del ámbito de negocio lo antes posible. La experiencia ha demostrado que una buena proporción podría ser 1:4, esto es, al menos un analista de negocio por cada cuatro desarrolladores[15].

Los cambios en los requisitos se suman a la planificación de las iteraciones siguientes y se priorizan junto con las demás tareas pendientes.

Los planes se hacen frecuentemente y se reajustan si hace falta. Siempre son planes de corta duración, menores de seis meses, aunque la empresa pueda tener una planificación a muy alto nivel que cubra más tiempo.

El código pertenece a todo el equipo (propiedad colectiva) y cualquier desarrollador está en condiciones de modificar código escrito por otro. Evi-

[14] En el capítulo sobre ATDD se describe este proceso

[15] Esta cifra puede ser relativa a las personas por grupo de trabajo, en los cuales los analistas estarán asignados con tiempo más reducido, es decir, estarán en más grupos. Por ejemplo con 16 desarrolladores y 2 analistas pueden hacerse 4 grupos de 4 desarrolladores y un analista pero cada analista en 2 grupos

tamos las situaciones del tipo... "esto sólo lo sabe tocar Manolo que lleva meses trabajando en ello".

Todo el equipo se reúne periódicamente, incluidos usuarios y analistas de negocio, a ser posible diariamente y si no, al menos una vez a la semana. Por norma general, se admite que sólo los desarrolladores se reúnan diariamente y que la reunión con el cliente/analista sea sólo una vez a la semana, ya se sabe que no vivimos en un mundo ideal. De hecho, nos contentaremos con que el cliente acuda a las reuniones de comienzo de iteración.

Las reuniones tienen hora de comienzo y de final y son breves. Cuando suena la alarma de fin de la reunión, es como si sonase la campana de incendio en la central de bomberos: cada uno de vuelta a su puesto de trabajo inmediatamente.

La superación de obstáculos imprevistos tiene prioridad sobre las convenciones o reglas generales de trabajo preestablecidas. Es decir, si hay que saltarse el protocolo de la empresa para resolver un problema que se nos ha atravesado, se hace. Por protocolo nos referimos a la forma en que habitualmente cooperan unas personas con otras, o tal vez la manera en que se lanzan nuevas versiones,...

Las grandes decisiones de arquitectura las toma todo el equipo, no son impuestas por el arquitecto. Sigue siendo recomendable utilizar patrones de diseño y otras buenas prácticas pero siempre dando máxima importancia y prioridad a los requisitos de negocio. Las arquitecturas ágiles son evolutivas, no se diseñan al completo antes de escribir el código de negocio. Este libro defiende particularmente las arquitecturas que emergen de los requisitos; TDD habla de que la arquitectura se forja a base de iterar y refactorizar, en lugar de diseñarla completamente de antemano.

La aplicación se ensambla y se despliega en entornos de preproducción a diario, de forma automatizada. Las baterías de tests se ejecutan varias veces al día. La cobertura[16] de los tests debe ser en torno al 60 % o mayor. En realidad, se trata de tener en cada iteración una cobertura aún mayor que en la anterior, no hay que ser demasiado precisos con el porcentaje.

Los desarrolladores envían sus cambios al repositorio de código fuente al menos una vez al día (commit).

Cada vez que se termina de desarrollar una nueva función, esta pasa al equipo de calidad para que la valide aunque el resto todavía no estén listas.

Partiendo de estas premisas, cada metodología o técnica ágil detalla con exactitud cómo se gestiona el proyecto y cómo es el proceso de desarrollo del código. A veces se pueden combinar varias metodologías aunque algunos autores recomiendan seguir al pie de la letra la metodología en cuestión sin

[16]La cobertura de código mediante tests se refiere al porcentaje de código que tiene tests asociados, considerando todos los cauces que puede tomar el flujo de ejecución

mezclar ni salirse del camino en ningún momento.

No podemos negar que las metodologías son a menudo disciplinas y que implantarlas no es sencillo, todo tiene su coste y tenemos que poner en la balanza las dificultades y los beneficios para determinar qué decisión tomamos frente a cada problema. En el libro de Canales[13] se habla precisamente de la implantación y puesta en marcha de metodologías en la empresa.

Esperemos que en el futuro cercano contemos con literatura en castellano sobre cada una de las metodologías ágiles más populares.

## 1.5. La situación actual

Son muchos los que se han dado cuenta de la necesidad de un cambio y, guiados por aquellos que ya lo han emprendido, han modificado el proceso de desarrollo para reaccionar ante esta crisis. La mayor parte de los que lo han hecho viven en el mundo anglosajón, en lugares como Norte América o Reino Unido o bien siguen su corriente, como las grandes potencias en expansión, India y China, que copian lo mejor del sistema anglosajón. Sin olvidar el país de la tecnología, Japón, que además de copiar marca tendencias. ¿Por qué estamos tardando tanto en apuntarnos a este movimiento?

En España, el objetivo de las universidades es la formación integral del alumno. No se pretende capacitarles para afrontar problemas concretos en circunstancias concretas, sino hacer que sean profesionales capacitados para afrontar con éxito su cometido sea cual sea la tendencia que les toque vivir. En definitiva, el objetivo principal es la creación, desarrollo, transmisión, difusión y crítica de la ciencia, la técnica, el arte y la cultura, promoviendo una visión integral del conocimiento. En el caso concreto de la informática, esto hace que no se imponga como requisito que los profesores sean profesionales que se dediquen o se hayan dedicado profesionalmente a construir software.

Esto no es bueno ni malo, cada cual cumple su función en las distintas etapas por las que pasamos, lo que es negativo es que aceptemos, sin la menor duda ni crítica, que lo que nos han enseñado es la única manera de sacar adelante los proyectos. Que ya no hay que aprender nada más. Es ciertamente dramático que, a pesar de esta realidad, miremos con tanto escepticismo las nuevas técnicas de gestión de proyectos software. Los grandes libros sobre software escritos en inglés en las últimas dos décadas, no están escritos por profesores de universidad sino por líderes de la industria con treinta años de batalla sobre sus espaldas. No pretendo abrir un debate sobre cuál es el objetivo de la universidad ni de la formación profesional, sino abrir un debate interior en cada uno de los que ya han salido de su largo periodo de formación y piensan que aquello que han aprendido es el único camino posible, todo lo que tienen que aplicar, cuando en realidad lo que han adquirido es tan sólo

una base y, en algunos casos, una autoconfianza peligrosamente arrogante.

La labor de nuestros profesores es fundamental y debemos estar agradecidos porque nos han enseñado las reglas de los lenguajes formales y nos han hablado de Alan Turing o del algoritmo de Edsger Dijkstra. No cabe duda de que, en esa etapa, hemos convertido el cerebro en un músculo bien entrenado. La tecnología cambia a velocidad de vértigo, no podemos esperar que en la universidad nos enseñen continuamente lo último que va saliendo porque en poco tiempo puede quedar obsoleto. Es más difícil que el modelo de la Máquina de Turing quede obsoleto. Recuerdo cuando me quejaba porque en la ingeniería técnica no había visto nada de ciertas herramientas de moda y ahora resulta que están extinguiéndose, que realmente no las necesito. Desgraciadamente, hay materias que llevan años en uso y a las que se les augura larga vida pero que todavía no han llegado a los temarios de los institutos ni de las universidades. Las cosas de palacio van despacio. Estoy convencido de que llegarán pero no podemos esperar a que nos lo cuenten ahí para aplicarlos, porque el cliente nos está pidiendo el producto ya. Necesita software de calidad ahora. Por tanto, el mensaje de fondo no es que todo lo aprendido durante nuestros años de estudiantes sea erróneo sino que el camino está empezando.

Todo lo dicho es aplicable a nuestros mentores en la empresa privada. Hoy día son muchas las empresas de tecnología que están compuestas por gente muy joven y con poca experiencia que no tiene más remedio que llevar la batuta como buenamente puede. La cuestión es plantearse que quizás la manera en que se resuelven los problemas no es la más apropiada. Por otro lado, es importante saber reconocer cuando estamos sacando el máximo partido a los métodos. Lo que pasa es que sin conocimiento, no podremos discernirlo. En cuanto a las empresas con personal experimentado, no están libres de caer en la autoreferencia y limitarse a reproducir lo que han hecho durante años. Esta rutina en sí misma no es negativa siempre que el cliente esté satisfecho y le estemos ofreciendo el mejor servicio y, al mismo tiempo, el personal se sienta realizado. Entonces la cuestión es... ¿Lo estamos haciendo?.

La parte artesanal de los programas de ordenador se podría deber a que desconocemos la totalidad de las variables de las que depende, porque si las conociésemos de antemano, no sería una ingeniería tan distinta del resto. Desde un punto de vista muy ingenieril, podemos considerar que la artesanía es simplemente una forma de producir demasiado compleja como para sintetizarla y reproducirla mecánicamente. Este arte no se desarrolla estudiando teoría sino practicando, al igual que a andar se aprende andando.

En el mundo tecnológico los meses parecen días y los años, meses. Las oportunidades aparecen y se desvanecen fugazmente y nos vemos obligados

a tomar decisiones con presteza. Las decisiones tecnológicas han convertido en multimillonarias a personas en cuestión de meses y han hundido imperios exactamente con la misma rapidez. Ahora nos está comenzando a llegar la onda expansiva de un movimiento que pone en entredicho técnicas que teníamos por buenas pero que con el paso de los años se están revelando insostenibles. Si bien hace poco gustábamos de diseñar complejas arquitecturas antes de escribir una sola línea de código que atacase directamente al problema del cliente, ahora, con la escasez de recursos económicos y la mayor exigencia de los usuarios, la palabra agilidad va adquiriendo valores de eficacia, elegancia, simplicidad y sostenibilidad. ¿Podemos beneficiarnos de esta nueva corriente?. Saber adaptarse al cambio es esencial para la evolución. ¿Nos adaptaremos a los cambios del entorno a tiempo?. Todos esos países de los que hablábamos son competidores en realidad y lo serán cada vez más dada la rápida expansión de Internet. ¿No estaríamos mejor si fuesen simplemente colegas?.

El software es una herramienta de presente y de futuro, creada para hacer más agradable la vida de los usuarios. Y, aunque tienda a olvidarse, también puede ser muy gratificante para los desarrolladores/analistas. Tendremos que valernos de confianza y dedicación junto con gusto por el trabajo para alcanzar esta meta pero... ¿Cómo podemos fomentar estas condiciones? Como ven, zarpamos con preguntas hacia el fascinante mundo del desarrollo ágil de software. Será una travesía que nos irá descubriendo las claves de cómo hacer mejor software al tiempo que nos sentimos más satisfechos con nuestro trabajo. Es posible escribir software de mayor calidad con menos complicaciones y aportar más a los negocios de las personas que lo utilizan.

Bienvenidos a bordo.

## 1.6. Ágil parece, plátano es

Se está usando mucho la palabra ágil[17] y, por desgracia, no siempre está bien empleada. Algunos aprovechan el término ágil para referirse a cowboy programming (programación a lo vaquero), es decir, hacer lo que les viene en gana, como quieren y cuando quieren. Incluso hay empresas que creen estar siguiendo métodos ágiles pero que en realidad no lo hacen (y no saben que no lo hacen). Existen mitos sobre el agilismo que dicen que no se documenta y que no se planifica o analiza. También se dice que no se necesitan arquitectos pero, no es cierto, lo que sucede es que las decisiones de arquitectura se toman en equipo.

El mal uso de la palabra ágil causa malas y falsas ideas sobre lo que

[17] agile en inglés, pronunciada como áyail

verdaderamente es. Llegado este punto, hay que mirar con lupa a quien dice que está siguiendo un desarrollo ágil, tal como pasa con quien dice que vende productos ecológicos. Hay quien cree que es ágil porque "habla mucho con el cliente". Quizás por eso aparecieron las certificaciones en determinadas metodologías ágiles aunque, como muchas otras certificaciones, son sólo papeles que no garantizan la profesionalidad de la persona certificada(confío en que las certificaciones de la agricultura ecológica sí sean auténticas). No nos debemos fiar de alguien que ha asistido dos días a un curso de Scrum y ya dice ser un maestro, a no ser que tenga años de experiencia que le avalen.

Adoptar una metodología supone aprendizaje y disciplina, como todo lo que está bien hecho y, quienes realmente quieren subirse a este carro, necesitarán la ayuda de personas expertas en la materia. En Internet existen multitud de grupos y foros donde se ofrece ayuda desinteresadamente y también existen profesionales que ofrecen formación y entrenamiento en estas áreas y se desplazan a cualquier parte del mundo para trabajar con grupos. En inglés hay bastante literatura al respecto y es más que recomendable leer varios libros, sobre todo aquellos cuyos autores firmaron el manifiesto ágil.

Sobre recursos en castellano, actualmente hay mucho movimiento y grandes profesionales en Agile Spain (comunidad de ámbito español) y en el Foro Agiles[18] (la comunidad latinoamericana, muy extendida en Argentina), que entre otras cosas, organiza el evento internacional anual Agiles, así como multitud de openspaces bajo la marca "Agile Open".

## 1.7. Los roles dentro del equipo

Saber distinguir las obligaciones y limitaciones de cada uno de los roles del equipo ayuda a que el trabajo lo realicen las personas mejor capacitadas para ello, lo que se traduce en mayor calidad. Roles distintos no necesariamente significa personas distintas, sobre todo en equipos muy reducidos. Una persona puede adoptar más de un rol, puede ir adoptando distintos roles con el paso del tiempo, o rotar de rol a lo largo del día. Hagamos un repaso a los papeles más comunes en un proyecto software.

- Dueño del producto
- Cliente
- Analista de negocio
- Desarrolladores

[18] http://tech.groups.yahoo.com/group/foro-agiles

- Arquitectos
- Administradores de Sistemas

Dueño del producto: Su misión es pedir lo que necesita (no el cómo, sino el qué) y aceptar o pedir correcciones sobre lo que se le entrega.

Cliente: Es el dueño del producto y el usuario final.

Analista de negocio: También es el dueño del producto porque trabaja codo a codo con el cliente y traduce los requisitos en tests de aceptación para que los desarrolladores los entiendan, es decir, les explica qué hay que hacer y resuelve sus dudas.

Desarrolladores: Toman la información del analista de negocio y deciden cómo lo van a resolver además de implementar la solución. Aparte de escribir código, los desarrolladores deben tener conocimientos avanzados sobre usabilidad y diseño de interfaces de usuario, aunque es conveniente contar con una persona experimentada para asistir en casos particulares. Lo mismo para la accesibilidad.

Administradores de sistemas: Se encargan de velar por los servidores y servicios que necesitan los desarrolladores.

En el mundo anglosajón se habla mucho del arquitecto del software. El arquitecto es la persona capaz de tomar decisiones de diseño pero además se le supone la capacidad de poder hablar directamente con el cliente y entender los requisitos de negocio. En lugar de un rol, es una persona que adopta varios roles. En el agilismo todos los desarrolladores son arquitectos en el sentido de que se les permite tomar decisiones de arquitectura conforme se va escribiendo o refactorizando código. Hay que resaltar que se hacen revisiones de código entre compañeros. Además, ante decisiones complejas se pide opinión a desarrolladores más experimentados. Recordemos que existe propiedad colectiva del código y fluidez de conocimiento dentro del equipo.

Parece que no son tan complicados los roles... sin embargo, los confundimos a menudo. En nuestra industria del software hemos llegado al extremo de el que el cliente nos dice a nosotros, los ingenieros, cómo tenemos que hacer las cosas. Nos dice que quiere una pantalla con tal botón y tales menús, que las tablas de la base de datos tienen tales columnas, que la base de datos tiene que ser Oracle... ¡y nosotros lo aceptamos!. Sabemos que la escasez de profesionalidad ha tenido mucho que ver con esto y, el hecho de no tener claro cuáles son los roles de cada uno, hace que no seamos capaces de ponernos en nuestro sitio. Quien dice el cliente, dice el dueño del producto o, llegado el caso, el analista. De hecho, a menudo nos encontramos con analistas de negocio que, cuando hacen el análisis, entregan al equipo de desarrollo interfaces de usuario (pantallas dibujadas con Photoshop o con cualquier otro diseñador) además de las tablas que creen que lleva la base de

datos y sus consultas. ¿No habíamos quedado en que el dueño del producto pide el qué y no dice el cómo?. Si la persona que tiene el rol de analista también tiene el rol de desarrollador, entonces es comprensible que diseñe una interfaz de usuario pero entonces no debería pintarla con un programa de diseño gráfico y endiñársela a otro, sino trabajar en ella. Las pantallas no se diseñan al comienzo sino al final, cuando los requisitos de negocio ya se cumplen. Los requisitos son frases cortas en lenguaje natural que ejecuta una máquina automáticamente, ya que tienen forma de test, con lo que se sabe cuándo se han implementado. Si las pantallas se diseñan primero, se contamina la lógica de negocio con la interpretación que el diseñador pueda hacer de los requisitos y corremos el riesgo de escribir un código sujeto a la UI en lugar de a los requisitos, lo cual lo hace difícil de modificar ante cambios futuros en el negocio. El dueño del producto tampoco debe diseñar las tablas de la base de datos a no ser que también adopte el rol de desarrollador pero, incluso así, las tablas son de los últimos[19] elementos que aparecen en el proceso de implementación del requisito, tal como ocurre con la interfaz gráfica. Es decir, vamos desde el test de aceptación a tests de desarrollo que acabarán en la capa de datos que pide persistencia. Pensamos en requisitos, implementamos objetos, luego bajamos a tablas en una base de datos relacional y finalmente le ponemos una carcasa a la aplicación que se llama interfaz gráfica de usuario. No al revés.

Si cada cual no se limita a ejercer su rol o roles, estaremos restringiendo a aquellos que saben hacer su trabajo, limitándoles de modo que no les dejamos hacer lo mejor que saben.

## 1.8. ¿Por qué nos cuesta comenzar a ser ágiles?

Si el agilismo tiene tantas ventajas, ¿Por qué no lo está practicando ya todo el mundo? La resistencia al cambio es uno de los motivos fundamentales. Todavía forma parte de nuestra cultura pensar que las cosas de toda la vida son las mejores. Ya se sabe... "Si es de toda la vida, es como debe ser". Si los ingenieros y los científicos pensásemos así, entonces tendríamos máquinas de escribir en lugar de computadoras (en el mejor de los casos). Existen fuertes razones históricas para ser tan reticentes al cambio pero los que trabajamos con tecnología podemos dar un paso al frente en favor del desarrollo. ¡Ojo! Podemos dejar nuestro lado conservador para otros aspectos no tecnológicos, que seguro nos aportará muchos beneficios. No se trata de ser progresista ni estamos hablando de política, limitémonos a cuestiones de ciencia y tecnología.

[19] Cuando la lógica de negocio es tan simple como guardar y recuperar un dato, es aceptable empezar por los datos.

¿Estamos preparados para darle una oportunidad a la innovación o nos quedamos con lo de "toda la vida", aunque sólo tenga una vida de medio siglo? (en el caso de programadores junior, sólo un par de años). Hemos aceptado una determinada forma de trabajar (en el mundo del software) y nos parece inmutable aún cuando esta industria todavía está en pañales. Hemos llegado al punto en que la informática ya no es una cuestión de matemáticos sino de especialistas en cada una de las muchas áreas de la informática. Ni siquiera se han jubilado aún los primeros expertos en software de la historia.

Cuando era niño no se me pasaba por la cabeza que todo el mundo llevase un teléfono móvil encima y se comunicase desde cualquier lugar (hace sólo 15 años). Hace poco no nos imaginábamos que compraríamos por Internet ni que las personas encontrarían pareja a través de la red. Los que han tenido confianza en el cambio y han sabido crecer orgánicamente trabajan en lo que les gusta y no tienen problemas para llegar a fin de mes. Las nuevas tecnologías son el tren de alta velocidad que une el presente con el futuro en un abrir y cerrar de ojos.

Ahora bien, aprender una nueva técnica supone esfuerzo. Es natural que nos dé pereza dar los primeros pasos hacia el cambio y por eso usamos mil excusas:

- Que es antinatural...
- Que está todo al revés...
- Que es un caos...
- Que no tenemos tiempo ahora para aprender eso...
- Que no tenemos los conocimientos previos para empezar...
- Que no sabemos por dónde empezar...
- Mañana empiezo...
- Es que,... es que...

Nos comportamos como lo haría un fumador al que le dicen que deje de fumar. La corriente popular en términos de software no es capaz de evolucionar lo suficientemente rápido como para que sus teorías sean las mejores. Hay que plantearse si seguirla es buena idea o conviene cambiar de corriente. Esto es,... ¿Prefiere la pastilla azul o la roja?[20]. No negaremos que hay que

[20]Célebre escena de la película Matrix en que Morfeo ofrece a Neo la posibilidad de despertar del sueño. Neo escoge la roja. En este caso despertar del sueño significa cambia a mejor, a diferencia de lo que sucede en esta película de ficción. La Pastilla Roja también es el título de un libro sobre Software Libre escrito por Juan Tomás García(http://www.lapastillaroja.net/resumen_ejecutivo.html)

hacer una inversión en tiempo y esfuerzo para aprender y poner en práctica una nueva forma de funcionar, la cuestión es que tal inversión se amortiza rápidamente. Si esperamos a mañana para que se den las condiciones perfectas y empezar a ser ágiles, quizás nunca llegue el día. ¿Acaso piensa que alguna vez tendrá tiempo y dinero de sobra para todo lo que quiera? El plan es más bien parecido al de echar monedas a la hucha para irse de vacaciones; pequeñas inversiones poco a poco. No se interprete que podemos jugar con el dinero del cliente, aprender no significa jugar con su dinero, ¿vale?.

Si aceptamos que el software siempre se puede mejorar, el siguiente paso es admitir que es positivo mantener un cierto aire inconformista en nuestra actitud profesional. La autocrítica nos lleva a escribir código de mayor calidad y a reconocer nuestros errores. El juicio sano sobre nuestro trabajo nos guía en la búsqueda de mejoras estrategias y nos ayuda a superar la pereza que nos produce la idea del cambio. Todos los días aprendemos algo nuevo. El día que deje de ser así habrá que reflexionar seriamente sobre si estamos ejerciendo bien el puesto de ingenieros de software.

Este capítulo ha sido compuesto a base de pinceladas procedentes diversos temas. No pretende ser completo, ya que se podría escribir un libro entero sobre metodologías, sino solo establecer un contexto de partida para el resto de capítulos.

Capítulo 2

# ¿Qué es el Desarrollo Dirigido por Tests? (TDD)

El Desarrollo Dirigido por Tests (Test Driven Development), al cual me referiré como **TDD**, es una técnica de diseño e implementación de software incluida dentro de la metodología XP. Coincido con Peter Provost[1] en que el nombre es un tanto desafortunado; algo como *Diseño Dirigido por Ejemplos* hubiese sido quizás mas apropiado. TDD es una técnica para diseñar software que se centra en tres pilares fundamentales:

- La implementación de las funciones justas que el cliente necesita y no más[2].
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

Cuando empezamos a leer sobre TDD creemos que se trata de una buena técnica para que nuestro código tenga una cobertura de tests muy alta, algo que siempre es deseable, pero es realmente una herramienta de diseño que convierte al programador en un "oficial de primera". O, si no les gustan las metáforas, convierte al programador en desarrollador[3]. TDD es la respuesta a las grandes preguntas de:

[1] http://www.youtube.com/watch?v=JMEO6T6gkAA
[2] Evitamos desarrollar funcionalidad que nunca será usada
[3] http://www.ericsink.com/No_Programmers.html

> ¿Cómo lo hago?, ¿Por dónde empiezo?, ¿Cómo sé qué es lo que hay que implementar y lo que no?, ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente?

No se trata de escribir pruebas a granel como locos, sino de diseñar adecuadamente según los requisitos.

Pasamos, de pensar en implementar tareas, a pensar en ejemplos certeros que eliminen la ambigüedad creada por la prosa en lenguaje natural (nuestro idioma). Hasta ahora estábamos acostumbrados a que las tareas, o los casos de uso, eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código. Con TDD intentamos traducir el caso de uso o tarea en $X$ ejemplos, hasta que el número de ejemplos sea suficiente como para describir la tarea sin lugar a malinterpretaciones de ningún tipo.

En otras metodologías de software, primero nos preocupamos de definir cómo va a ser nuestra arquitectura. Pensamos en las clases de infraestructura que van a homogeneizar la forma de trabajar en todos y cada uno de los casos, pensamos si vamos a usar un patrón Facade[4] y otro Singleton[5] y una comunicación mediante eventos, o DTOs, y una clase central que va a hacer esto y aquello... ¿Y si luego resulta que no necesitamos todo eso? ¿Cuánto vamos a tardar en darnos cuenta de ello? ¿Cuánto dinero vamos a malgastar? En TDD dejamos que la propia implementación de pequeños ejemplos, en constantes iteraciones, hagan emerger la arquitectura que necesitamos usar. Ni más ni menos. No es que nos despreocupemos por completo de las características técnicas de la aplicación a priori, es decir, lógicamente tendremos que saber si el desarrollo será para un teléfono móvil, para una web o para un pc de escritorio; más que nada porque tenemos que elegir unas herramientas de desarrollo conformes a las exigencias del guión. Sin embargo, nos limitamos a escoger el framework correspondiente y a usar su arquitectura como base. Por ejemplo, si escogíesemos Django[6] o ASP.NET MVC[7], ya tendríamos definida buena parte de la base antes de empezar a escribir una sola línea de código. No es que trabajemos sin arquitectura, lógicamente, si en los requisitos está la interoperabilidad en las comunicaciones, tendremos que usar servicios web o servicios REST, lo cual ya propicia un determinado soporte. Lo que eliminamos son las arquitecturas encima de esas arquitecturas, las que intentan que todo se haga siempre igual y tal como se le ocurrió al "genio" de la empresa. A ser posible, esas que nos obligan a modificar siete ficheros para cambiar una cadena de texto. TDD produce una arquitectura que emerge de la no-ambigüedad de los tests automatizados,

[4]Facade: http://es.wikipedia.org/wiki/Facade_(patrón_de_diseño)
[5]Singleton: http://es.wikipedia.org/wiki/Singleton
[6]http://www.djangoproject.com
[7]http://www.asp.net/mvc/

lo cual no exime de las revisiones de código entre compañeros ni de hacer preguntas a los desarrolladores más veteranos del equipo.

Las primeras páginas del libro de Kent Beck [3] (uno de los padres de la metodología XP) dan unos argumentos muy claros y directos sobre por qué merece la pena darle unos cuantos tragos a TDD, o mejor, por qué es beneficioso convertirla en nuestra herramienta de diseño principal. Estas son algunas de las razones que da Kent junto con otras destacadas figuras de la industria:

- La calidad del software aumenta (y veremos por qué).
- Conseguimos código altamente reutilizable.
- El trabajo en equipo se hace más fácil, une a las personas.
- Nos permite confiar en nuestros compañeros aunque tengan menos experiencia.
- Multiplica la comunicación entre los miembros del equipo.
- Las personas encargadas de la garantía de calidad adquieren un rol más inteligente e interesante.
- Escribir el ejemplo (test) antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobrediseñar.
- Cuando revisamos un proyecto desarrollado mediante TDD, nos damos cuenta de que los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza del puzzle.

Personalmente, añadiría lo siguiente:

- Incrementa la productividad.
- Nos hace descubrir y afrontar más casos de uso en tiempo de diseño.
- La jornada se hace mucho más amena.
- Uno se marcha a casa con la reconfortante sensación de que el trabajo está bien hecho.

Ahora bien, como cualquier técnica, no es una varita mágica y no dará el mismo resultado a un experto arquitecto de software que a un programador junior que está empezando. Sin embargo, es útil para ambos y para todo el rango de integrantes del equipo que hay entre uno y otro. Para el arquitecto es

su mano derecha, una guía que le hace clarificar el dominio de negocio a cada test y que le permite confiar en su equipo aunque tenga menos experiencia. Frecuentemente, nos encontramos con gente muy desconfiada que mira con lupa el código de su equipo antes de que nadie pueda hacer "commit" al sistema de control de versiones. Esto se convierte en un cuello de botella porque hay varias personas esperando por el jefe (el arquitecto) para que dé el visto bueno y a este se le acumula el trabajo. Ni que decir tiene que el ambiente de trabajo en estos casos no es nada bueno ni productivo. Cuando el jefe sabe que su equipo hace TDD correctamente puede confiar en ellos y en lo que diga el sistema de integración contínua[8] y las estadísticas del repositorio de código. Para el programador junior que no sabe por dónde va a coger al toro, o si es el toro quien le va a coger a él (o a ella), se convierte en el "Pepito Grillo" que le cuenta qué paso tiene que dar ahora. Y así, un paso tras otro, le guía en la implementación de la tarea que le ha sido asignada.

Cuando el equipo practica de esta manera, la comunicación fluye, la gente se vuelve más activa y la maquinaria funciona como un engranaje bien lubricado. Todos los que disfrutamos trabajando en el software llevamos dentro al personaje del buen arquitecto, entre muchos otros de nuestros personajes. La práctica que aquí se describe nos lo trae para que nos ayude y nos haga la vida más fácil.

## 2.1. El algoritmo TDD

La esencia de TDD es sencilla pero ponerla en práctica correctamente es cuestión de entrenamiento, como tantas otras cosas. El algoritmo TDD sólo tiene tres pasos:

- Escribir la especificación del requisito (el ejemplo, el test).
- Implementar el código según dicho ejemplo.
- Refactorizar para eliminar duplicidad y hacer mejoras.

Veámosla en detalle.

### 2.1.1. Escribir la especificación primero

Una vez que tenemos claro cuál es el requisito, lo expresamos en forma de código. Si estamos a nivel de aceptación o de historia, lo haremos con un framework tipo Fit, Fitnesse, Concordion o Cucumber. Esto es, ATDD. Si no, lo haremos con algún framework xUnit. ¿Cómo escribimos un test para

[8] http://es.wikipedia.org/wiki/Continuous_integration

un código que todavía no existe? Respondamos con otra pregunta ¿Acaso no es posible escribir una especificación antes de implementarla? Por citar un ejemplo conocido, las JSR (Java Specification Request) se escriben para que luego terceras partes las implementen... ¡ah, entonces es posible!. El framework Mono se ha implementado basándose en las especificaciones del ECMA-334 y ECMA-335 y funciona. Por eso, un test no es inicialmente un test sino un ejemplo o especificación. La palabra especificación podría tener la connotación de que es inamovible, algo preestablecido y fijo, pero no es así. Un test se puede modificar. Para poder escribirlo, tenemos que pensar primero en cómo queremos que sea la API del SUT[9], es decir, tenemos que trazar antes de implementar. Pero sólo una parte pequeña, un comportamiento del SUT bien definido y sólo uno. Tenemos que hacer el esfuerzo de imaginar cómo seria el código del SUT si ya estuviera implementado y cómo comprobaríamos que, efectivamente, hace lo que le pedimos que haga. La diferencia con los que dictan una JSR es que no diseñamos todas las especificaciones antes de implementar cada una, sino que vamos una a una siguiendo los tres pasos del algoritmo TDD. El hecho de tener que usar una funcionalidad antes de haberla escrito le da un giro de 180 grados al código resultante. No vamos a empezar por fastidiarnos a nosotros mismos sino que nos cuidaremos de diseñar lo que nos sea más cómodo, más claro, siempre que cumpla con el requisito objetivo. En los próximos capítulos veremos cómo mediante ejemplos.

### 2.1.2. Implementar el código que hace funcionar el ejemplo

Teniendo el ejemplo escrito, codificamos lo mínimo necesario para que se cumpla, para que el test pase. Típicamente, el mínimo código es el de menor número de caracteres porque mínimo quiere decir el que menos tiempo nos llevó escribirlo. No importa que el código parezca feo o chapucero, eso lo vamos a enmendar en el siguiente paso y en las siguientes iteraciones. En este paso, la máxima es no implementar nada más que lo estrictamente obligatorio para cumplir la especificación actual. Y no se trata de hacerlo sin pensar, sino concentrados para ser eficientes. Parece fácil pero, al principio, no lo es; veremos que siempre escribimos más código del que hace falta. Si estamos bien concentrados, nos vendrán a la mente dudas sobre el comportamiento del SUT ante distintas entradas, es decir, los distintos flujos condicionales que pueden entrar en juego; el resto de especificaciones de este bloque de funcionalidad. Estaremos tentados de escribir el código que los gestiona sobre la marcha y, en ese momento, sólo la atención nos ayudará a contener el

[9]Subject Under Test. Es el objeto que nos ocupa, el que estamos diseñando a través de ejemplos.

impulso y a anotar las preguntas que nos han surgido en un lugar al margen para convertirlas en especificaciones que retomaremos después, en iteraciones consecutivas.

### 2.1.3. Refactorizar

Refactorizar no significa reescribir el código; reescribir es más general que refactorizar. Segun Martín Fowler, refactorizar[10] es modificar el diseño sin alterar su comportamiento. A ser posible, sin alterar su API pública. En este tercer paso del algoritmo TDD, rastreamos el código (también el del test) en busca de líneas duplicadas y las eliminamos refactorizando. Además, revisamos que el código cumpla con ciertos principios de diseño (me inclino por S.O.L.I.D) y refactorizamos para que así sea. Siempre que llego al paso de refactorizar, y elimino la duplicidad, me planteo si el método en cuestión y su clase cumplen el Principio de una Única Responsabilidad[11] y demás principios.

El propio Fowler escribió uno de los libros más grandes de la literatura técnica moderna[7] en el que se describen las refactorizaciones más comunes. Cada una de ellas es como una receta de cocina. Dadas unas precondiciones, se aplican unos determinados cambios que mejoran el diseño del software mientras que su comportamiento sigue siendo el mismo. Mejora es una palabra ciertamente subjetiva, por lo que empleamos la métrica del código duplicado como parámetro de calidad. Si no existe código duplicado, entonces hemos conseguido uno de más calidad que el que presentaba duplicidad. Mas allá de la duplicidad, durante la refactorización podemos permitirnos darle una vuelta de tuerca al código para hacerlo más claro y fácil de mantener. Eso ya depende del conocimiento y la experiencia de cada uno. Los IDE como Eclipse, Netbeans o VisualStudio, son capaces de llevar a cabo las refactorizaciones más comunes. Basta con señalar un bloque de código y elegir la refactorización Extraer-Método, Extraer-Clase, Pull-up, Pull-down o cualquiera de las muchas disponibles. El IDE modifica el código por nosotros, asegurándonos que no se cometen errores en la transición. Al margen de estas refactorizaciones, existen otras más complejas que tienen que ver con la maestría del desarrollador y que a veces recuerdan al mago sacando un conejo de la chistera. Algunas de ellas tienen nombre y están catalogadas a modo de patrón y otras son anónimas pero igualmente eliminan la duplicidad. Cualquier cambio en los adentros del código, que mantenga su API pública, es una refactorización. La clave de una buena refactorización es hacerlo en pasitos muy pequeños. Se hace un cambio, se ejecutan todos los tests

[10] http://www.refactoring.com/

[11] Ver Capítulo7 en la página 111

y, si todo sigue funcionando, se hace otro pequeño cambio. Cuando refactorizamos, pensamos en global, contemplamos la perspectiva general, pero actuamos en local. Es el momento de detectar malos olores y eliminarlos. El verbo refactorizar no existe como tal en la Real Academia Española pero, tras discutirlo en la red, nos resulta la mejor traducción del término refactoring. La tarea de buscar y eliminar código duplicado después de haber completado los dos pasos anteriores, es la que más tiende a olvidarse. Es común entrar en la dinámica de escribir el test, luego el SUT, y así sucesivamente olvidando la refactorización. Si de las tres etapas que tiene el algoritmo TDD dejamos atrás una, lógicamente no estamos practicando TDD sino otra cosa.

Otra forma de enumerar las tres fases del ciclo es:

- Rojo
- Verde
- Refactorizar

Es una descripción metafórica ya que los frameworks de tests suelen colorear en rojo aquellas especificaciones que no se cumplen y en verde las que lo hacen. Así, cuando escribimos el test, el primer color es rojo porque todavía no existe código que implemente el requisito. Una vez implementado, se pasa a verde.

Cuando hemos dado los tres pasos de la especificación que nos ocupa, tomamos la siguiente y volvemos a repetirlos. Parece demasiado simple, la reacción de los asistentes a mis cursos es mayoritariamente incrédula y es que el efecto TDD sólo se percibe cuando se practica. Me gusta decir que tiene una similitud con un buen vino; la primera vez que se prueba el vino en la vida, no gusta a nadie, pero a fuerza de repetir se convierte en un placer para los sentidos. Connotaciones alcohólicas a un lado, espero que se capte el mensaje.

¿Y TDD sirve para proyectos grandes? Un proyecto grande no es sino la agrupación de pequeños subproyectos y es ahora cuando toca aplicar aquello de "divide y vencerás". El tamaño del proyecto no guarda relación con la aplicabilidad de TDD. La clave está en saber dividir, en saber priorizar. De ahí la ayuda de Scrum para gestionar adecuadamente el *backlog* del producto. Por eso tanta gente combina XP y Scrum. Todavía no he encontrado ningún proyecto en el que se desaconseje aplicar TDD.

En la segunda parte del libro se expondrá el algoritmo TDD mediante ejemplos prácticos, donde iremos de menos a más, iterando progresivamente. No se preocupe si no lo ve del todo claro ahora.

## 2.2. Consideraciones y recomendaciones

### 2.2.1. Ventajas del desarrollador experto frente al junior

Existe la leyenda de que TDD únicamente es válido para personal altamente cualificado y con muchísima experiencia. Dista de la realidad; TDD es bueno para todos los individuos y en todos los proyectos. Eso sí, hay algunos matices. La diferencia entre el desarrollador experimentado que se sienta a hacer TDD y el junior, es cómo enfocan los tests, es decir, qué tests escriben; más allá del código que escriben. El experto en diseño orientado a objetos buscará un test que fuerce al SUT a tener una estructura o una API que sabe que le dará buenos resultados en términos de legibilidad y reusabilidad. Un experto es capaz de anticipar futuros casos de uso y futuros problemas y será más cuidadoso diseñando la API test tras test, aplicando las buenas prácticas que conoce. El junior probablemente se siente a escribir lo que mejor le parece, sin saber que la solución que elige quizás le traiga quebraderos de cabeza más adelante. La ventaja es que, cuando se dé cuenta de que su diseño tiene puntos a mejorar y empiece a refactorizar, contará con un importantísimo respaldo detrás en forma de batería de tests. Por poco experimentado que sea, se cuidará de no diseñar una API que le resulte casi imposible de usar. Debe tenerse en cuenta que se supone que el principiante no está solo, sino que en un contexto XP, hay desarrolladores de más experiencia que supervisarán y habrá momentos en los que se programe en parejas. La figura de los líderes es importante en XP al igual que en otras metodologías, con la gran diferencia de que el líder ágil está para responder preguntas y ayudar a los demás y no para darles látigo. El líder debe intentar que las personas que trabajan con él estén contentas de trabajar ahí y quieran seguir haciéndolo.

### 2.2.2. TDD con una tecnología desconocida

La primera vez que usamos una determinada tecnología o incluso una nueva librería, es complicado que podamos escribir la especificación antes que el SUT, porque no sabemos las limitaciones y fortalezas que ofrece la nueva herramienta. En estos casos, XP habla de *spikes* (disculpen que no lo traduzca, no sabría como) .Un spike es un pequeño programa que se escribe para indagar en la herramienta, explorando su funcionalidad. Es hacerse alguna función o alguna aplicación pequeña que nos aporte el conocimiento que no tenemos. Si el spike es pequeño, y resulta que nos damos cuenta que su propio código es válido tal cual, entonces escribiremos el test justo a continuación, en lugar de dejarlo sin test. Sin un conocimiento básico de la API y las restricciones del sistema, no recomendaría lanzarse a escribir especificaciones. Hay que respetar el tiempo de aprendizaje con la herramienta y

avanzar una vez que tengamos confianza con ella. Intentar practicar TDD en un entorno desconocido es, a mi parecer, un antipatrón poco documentado. Tampoco es que descartemos forzosamente TDD, sino que primero tendremos que aprender a pilotar la máquina. Una vez sepamos si es de cambio manual o automático, dónde se encienden las luces y dónde se activa el limpia parabrisas, podremos echar a rodar. Es sólo cuestión de aplicar el sentido común, primero aprendemos a usar la herramienta y luego la usamos. Tenemos que evitar algo que pasa muy frecuentemente, minusvalorar el riesgo de no dominar las herramientas (y frameworks y lenguajes...)

### 2.2.3. TDD en medio de un proyecto

En la segunda parte del libro, la de los ejemplos prácticos, iniciamos el desarrollo de una aplicación desde cero. Igual que hacemos en los cursos que imparto. La pregunta de los asistentes aparece antes o después: ¿no se puede aplicar TDD en un proyecto que ya está parcialmente implementado? Claro que se puede, aunque con más consideraciones en juego. Para los nuevos requisitos de la aplicación, es decir, aquello que todavía falta por implementar, podremos aplicar eso de escribir el test primero y luego el código (¡y después refactorizar!). Es probable que el nuevo SUT colabore con partes legadas que no permiten la inyección de dependencias y que no cumplen una única responsabilidad[12]; código legado que nos dificulta su reutilización. El libro más recomendado por todos en los últimos tiempos sobre este asunto es, "Working Effectively with Legacy Code" de Michael C. Feathers[6]. Tratar con código legado no es moco de pavo. En general, por código legado entendemos que se trata de aquel que no tiene tests de ningún tipo. Mi recomendación, antes de ponerse a reescribir partes de código legado, es crear tests de sistema (y cuando el código lo permita, tests unitarios) que minimicen los posibles efectos colaterales de la reescritura. Si es una web, por ejemplo, agarrar Selenium o similar y grabar todos los posibles usos de la interfaz gráfica para poderlos reproducir después de las modificaciones y comprobar que todo el sistema se sigue comportando de la misma manera. Es un esfuerzo de usar y tirar porque estos tests son tremendamente frágiles, pero es mucho más seguro que lanzarse a reescribir alegremente. La siguiente recomendación es que la nueva API y la vieja convivan durante un tiempo, en lugar de reescribir eliminando la versión legada. Además de tener dos API podemos sobrecargar métodos para intentar que el código legado y su nueva versión convivan, si es que la API antigua nos sigue sirviendo. Viene siendo cuestión de aplicar el sentido común y recordar la ley de Murphy; "Si puede salir mal, saldrá mal". Otra alternativa para hacer TDD con código nuevo

[12]Ver Capítulo 7 en la página 111

que colabora con código legado es abusar de los objetos mock[13]. Digamos que los tests van a ser más frágiles de lo que deberían pero es mejor usar paracaídas que saltar sin nada. Y por supuesto, si el nuevo código es más independiente, podemos seguir haciendo TDD sin ningún problema. Se lo recomiendo encarecidamente.

[13]Los veremos en el Capítulo 6 en la página 95

Capítulo 3

# Desarrollo Dirigido por Tests de Aceptación (ATDD)

A pesar de la brevedad de este capítulo, puede considerarse probablemente *el más importante de todo el libro*. Si no somos capaces de entendernos con el cliente, ni la mejor técnica de desarrollo de todos los tiempos producirá un buen resultado.

La mayor diferencia entre las metodologías clásicas y la Programación Extrema es la forma en que se expresan los requisitos de negocio. En lugar de documentos de *word*, son ejemplos ejecutables. El Desarrollo Dirigido por Test de Aceptación (ATDD), técnica conocida también como Story Test-Driven Development (STDD), es igualmente TDD pero a un nivel diferente. Los tests de aceptación o de cliente son el criterio escrito de que el software cumple los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto.

Es el punto de partida del desarrollo en cada iteración, la conexión perfecta entre Scrum y XP; allá donde una se queda y sigue la otra.

ATDD/STDD es una forma de afrontar la implementación de una manera totalmente distinta a las metodologías tradicionales. El trabajo del analista de negocio se transforma para reemplazar páginas y páginas de requisitos escritos en lenguaje natural (nuestro idioma), por ejemplos ejecutables surgidos del consenso entre los distintos miembros del equipo, incluido por supuesto el cliente. No hablo de reemplazar toda la documentación, sino los requisitos, los cuales considero un subconjunto de la documentación.

El algoritmo es el mismo de tres pasos pero son de mayor zancada que en el TDD practicado exclusivamente por desarrolladores. En ATDD la lista de ejemplos (tests) de cada historia, se escribe en una reunión que incluye a

dueños de producto, desarrolladores y responsables de calidad. Todo el equipo debe entender qué es lo que hay que hacer y por qué, para concretar el modo en que se certifica que el sotfware lo hace. Como no hay única manera de decidir los criterios de aceptación, los distintos roles del equipo se apoyan entre sí para darles forma.

## 3.1. Las historias de usuario

Una historia de usuario posee similitudes con un caso de uso, salvando ciertas distancias. Por hacer una correspondencia entre historias de usuario y casos de uso, podríamos decir que el título de la historia se corresponde con el del caso de uso tradicional. Sin embargo, la historia no pretende definir el requisito. Escribir una definición formal incurre en el peligro de la imprecisión y la malinterpretación, mientras que contarlo con ejemplos ilustrativos, transmite la idea sin complicaciones. En ATDD cada historia de usuario contiene una lista de ejemplos que cuentan lo que el cliente quiere, con total claridad y ninguna ambigüedad. El enunciado de una historia es tan sólo una frase en lenguaje humano, de alrededor de cinco palabras, que resume qué es lo que hay que hacer. Ejemplos de historias podrían ser:

- *Formulario de inscripción*
- *Login en el sistema*
- *Reservar una habitación*
- *Añadir un libro al carrito de la compra*
- *Pago con tarjeta de crédito*
- *Anotar un día festivo en el canlendario*
- *Informe de los artículos más vendidos*
- *Darse de baja en el foro*
- *Buscar casas de alquiler en Tenerife*

Breves, concretas y algo estimables. Son el resultado de escuchar al cliente y ayudarle a resumir el requisito en una sola frase. Muy importante: Están escritas con el vocabulario del negocio del cliente, no con vocabulario técnico. Por sí misma una historia aislada es difícil de estimar incluso con este formato. Lo que las hace estimables y nos hace ser capaces de estimarlas cada vez mejor, es el proceso evolutivo que llamamos ágil. Esto es: a base de iterar, estimar en cada iteración y hacer restrospectiva al final de la misma, vamos refinando la habilidad de escribir historias y estimarlas.

Canales[13] da una guía estupenda para las estimaciones entre las páginas 305 y 319 de su libro. Es más que recomendable leerlo. Sin embargo, desde la página 320 hasta la 343, discrepo con su forma de afrontar el análisis. Antes de conocer ATDD, también trabajaba como nos dice en esas páginas pero la experiencia me ha enseñado que no es la mejor manera. Saltar de casos de uso a crear un diagrama de clases modelando entidades, es en mi opinión, peligroso cuanto menos. Los diagramas nos pueden ayudar a observar el problema desde una perspectiva global, de manera que nos aproximamos al dominio del cliente de una manera más intuitiva. Pueden ayudarnos a comprender el dominio hasta que llegamos a ser capaces de formular ejemplos concretos. En cambio, representar elementos que formarán parte del código fuente mediante diagramas, es una fuente de problemas. Traducir diagramas en código fuente, es decir el modelado, es en cierto modo opuesto a lo que se expone en este libro. Para mí, la única utilidad que tiene el UML[a] es la de representar mediante un diagrama de clases, código fuente existente. Es decir, utilizo herramientas que autogeneran diagramas de clases, a partir de código, para poder echar un vistazo a las entidades de manera global pero nunca hago un diagrama de clases antes de programar. Mis entidades emergen a base de construir el código conforme a ejemplos. En todos los "ejemplos" que aparecen en las citadas páginas, realmente lo que leemos son descripciones, no son ejemplos potencialmente ejecutables. Definir entidades/modelos y hablar de pantallas antes de que haya una lista de ejemplos ejecutables y código ejecutable que los requiera, es un camino problemático. Como artesano del software, no creo en los generadores de aplicaciones.

[a]Lenguaje de Modelado Universal

Cada historia provoca una serie de preguntas acerca de los múltiples contextos en que se puede dar. Son las que naturalmente hacen los desarrolladores a los analistas de negocio o al cliente.

- ¿Qué hace el sistema si el libro que se quiere añadir al carrito ya está dentro de él?
- ¿Qué sucede si se ha agotado el libro en el almacén?
- ¿Se le indica al usuario que el libro ha sido añadido al carrito de la compra?

Las respuestas a estas preguntas son afirmaciones, ejemplos, los cuales transformamos en tests de aceptación. Por tanto, cada historia de usuario tiene asociados uno o varios tests de aceptación (ejemplos):

- *Cuando el libro X se añade al carrito, el sistema devuelve un mensaje que dice: ''El libro X ha sido añadido al carrito''*
- *Al mostrar el contenido del carrito aparece el libro X*
- *El libro X ya no aparece entre los libros a añadir al carrito*

Cuantas menos palabras para decir lo mismo, mejor:

- *Añadir libro X en stock produce: ''El libro X ha sido añadido al carrito''*
- *Libro X está contenido en el carrito*
- *Libro X ya no está en catálogo de libros*

Las preguntas surgidas de una historia de usuario pueden incluso dar lugar a otras historias que pasan a engrosar el backlog o lista de requisitos: "Si el libro no está en stock, se enviará un email al usuario cuando llegue".

Los tests de aceptación son así; afirmaciones en lenguaje humano que tanto el cliente, como los desarrolladores, como la máquina, entienden. ¿La máquina? ¿cómo puede entender eso la máquina? Mágicamente no. El equipo de desarrollo tiene que hacer el esfuerzo de conectar esas frases con los puntos de entrada y salida del código. Para esto existen diversos frameworks libres y gratuitos que reducen el trabajo. Los más conocidos son FIT, Fitnesse, Concordion, Cucumber y Robot. Básicamente lo que proponen es escribir las frases con un formato determinado como por ejemplo HTML, usando etiquetas de una manera específica para delimitar qué partes de la frase son variables de entrada para el código y cuales son datos para validación del resultado de la ejecución. Como salida, Concordion por ejemplo, produce un HTML modificado que marca en rojo las afirmaciones que no se cumplieron, además de mostrar las estadísticas generales sobre cuántos tests pasaron y cuántos no. Veamos un ejemplo de la sintaxis de Concordion:

```
<html
    xmlns:concordion="http://www.concordion.org/2007/concordion">
 <body>
  <p>
    El saludo para el usuario
    <span
       concordion:set="\#firstName">Manolo</span>
    será:
     <span
      concordion:assertEquals="greetingFor(\#firstName)">
           ¡Hola  Manolo!</span>
  </p>
 </body>
</html>
```

Lógicamente, no le pedimos al cliente que se aprenda la sintaxis de Concordion y escriba el código HTML. Le pedimos que nos ayude a definir la frase o que nos la valide y luego, entre analistas de negocio, desarrolladores y testers (equipo de calidad), se escribirá el HTML. Lo interesante para el cliente es que el renderizado del HTML contiene el ejemplo que él entiende y es una bonita tarjeta que Concordion coloreará con ayuda de la hoja de estilos, subrayando en verde o en rojo según funcione el software. Concordion sabe dónde buscar la función `greetingsFor` y reconoce que el argumento con que la invocará es `Manolo`. Comparará el resultado de la ejecución con la frase `¡Hola Manolo!` y marcará el test como verde o rojo en función de ello. Un test de cliente o de aceptación con estos frameworks, a nivel de código, es un enlace entre el ejemplo y el código fuente que lo implementa. El propio framework se encarga de hacer la pregunta de si las afirmaciones son ciertas o no. Por tanto, su aspecto dista mucho de un test unitario o de integración con un framework xUnit.

Para cada test de aceptación de una historia de usuario, habrá un conjunto de tests unitarios y de integración de grano más fino que se encargará, primero, de ayudar a diseñar el software y, segundo, de afirmar que funciona como sus creadores querían que funcionase. Por eso ATDD o STDD es el comienzo del ciclo iterativo a nivel desarrollo, porque partiendo de un test de aceptación vamos profundizando en la implementación con sucesivos test unitarios hasta darle forma al código que finalmente cumple con el criterio de aceptación definido. No empezamos el diseño en torno a una supuesta interfaz gráfica de usuario ni con el diseño de unas tablas de base de datos, sino marcando unos criterios de aceptación que nos ayuden a ir desde el lado del negocio hasta el lado más técnico pero siempre concentrados en lo que el cliente demanda, ni más ni menos. Las ventajas son numerosas. En primer lugar, no trabajaremos en funciones que finalmente no se van a usar[1].

[1]El famoso YAGNI (You ain't gonna need it)

En segundo lugar, forjaremos un código que está listo para cambiar si fuera necesario porque su diseño no está limitado por un diseño de base de datos ni por una interfaz de usuario. Es más fácil hacer modificaciones cuando se ha diseñado así, de arriba a abajo, en vez de abajo a arriba. Si el arquitecto diseña mal la estructura de un edificio, será muy complejo hacerle cambios en el futuro pero si pudiera ir montando la cocina y el salón sin necesidad de estructuras previas, para ir enseñándolas al cliente, seguro que al final colocaría la mejor de las estructuras para darle soporte a lo demás. En la construcción de viviendas eso no se puede hacer pero en el software, sí. Y, además, es lo natural, aunque estemos acostumbrados a lo contrario. ¡Porque una aplicación informática no es una casa!. Dada esta metáfora, se podría interpretar que deberíamos partir de una interfaz gráfica de usuario para la implementación pero no es cierto. Ver el dibujo de una interfaz gráfica de usuario no es como ver una cocina. Primero, porque la interfaz gráfica puede o no ser intuitiva, utilizable y, a consecuencia de esto, en segundo lugar, no es el medio adecuado para expresar qué es lo que el cliente necesita sino que la interfaz de usuario es parte del *cómo* se usa.

## 3.2. Qué y no Cómo

Una de las claves de ATDD es justamente que nos permite centrarnos en el **qué** y no en el **cómo**. Aprovechamos los frameworks tipo Concordion para desarrollar nuestra habilidad de preguntar al cliente qué quiere y no cómo lo quiere. Evitamos a toda costa ejemplos que se meten en el cómo hacer, más allá del qué hacer:

- *Al rellenar el cuadro de texto de buscar y pulsar el botón contiguo, los resultados aparecen en la tabla de la derecha*
- *Al introducir la fecha y hacer click en el botón de añadir, se crea un nuevo registro vacío*
- *Los libros se almacenan en la tabla Libro con campos: id, titulo y autor*
- *Seleccionar la opción de borrar del combo, marcar con un tick las líneas a borrar y verificar que se eliminan de la tabla al pulsar el botón aplicar.*
- *Aplicación Flash con destornilladores y tornillos girando en 3D para vender artículos de mi ferretería por Internet*

Cuando partimos de especificaciones como estas corremos el riesgo de pasar por alto el verdadero propósito de la aplicación, la información con auténtico

valor para el negocio del cliente. Salvo casos muy justificados, el Dueño del Producto no debe decir cómo se implementa su solución, igual que no le decimos al fontanero cómo tiene que colocar una tubería. La mayoría de las veces, el usuario no sabe exactamente lo que quiere pero, cuando le sugerimos ejemplos sin ambigüedad ni definiciones, generalmente sabe decirnos si es o no es eso lo que busca. Uno de los motivos por los que el cliente se empeña en pedir la solución de una determinada manera es porque se ha encontrado con profesionales poco experimentados que no le han sabido sugerir las formas adecuadas o que no llegaron a aportarle valor para su negocio. Con ATDD nos convertimos un poco en psicólogos en lugar de pretender ser adivinos. A base de colaboración encontramos y clasificamos la información que más beneficio genera para el usuario.

Encuentro particularmente difícil practicar ATDD cuando los dueños de producto están mal acostumbrados al sistema clásico en el que el análisis de los requisitos acaba produciendo un diagrama de componentes o módulos y luego un diagrama de clases. En las primeras reuniones de análisis, se empeñan en que dibujemos ese diagrama de módulos en los que el sistema se va a dividir a pesar de que les explique que eso no aporta más valor a su negocio. Les digo que la abstracción de los requisitos en forma de módulos o grupos no sirve más que para contaminar el software con falsos requisitos de negocio y para limitarnos a la hora de implementar, aunque a veces les resulta difícil de ver en un principio. Los únicos módulos que hay que identificar son los que tienen valor de negocio, es decir, aquellos conjuntos lógicos que tengan relación con una estrategia de negocio. Por ejemplo, de cara a ofrecer determinados servicios: servicio de venta, de alquiler, de consultoría...

La forma en que comprenden el proceso iterativo, es sentándose frente a ellos en un lugar cómodo y adoptando el rol de psicólogo de las películas norteamericanas, abordando los ejemplos. Una vez llevo la voz cantante, empiezo a formular ejemplos para que me digan si son válidos o no. Al principio no son capaces de distinguir entre una descripción y un ejemplo preciso por lo que se apresuran a darme descripciones que consideran suficientes como para implementar el software pero que para mí, ajeno a su negocio, no lo son:

- *Buscando por Santa Cruz de Tenerife, aparece una lista de pisos en alquiler.*

Entonces reconduzco la conversación haciéndoles ver que su descripción se corresponde en realidad con varios ejemplos.

- *Buscando que el precio sea inferior a 600€, e introduciendo el texto "Santa Cruz de Tenerife", el sistema muestra una lista de pisos que no superan los 600€mensuales de alquiler y que se encuentran en la ciudad de Santa Cruz de Tenerife*
- *Buscando que el precio esté entre 500€y 700€y que tenga 2 habitaciones e introduciendo el texto "Santa Cruz de Tenerife", el sistema muestra una lista de pisos que cumplen las tres condiciones*
- *Buscando que tenga 3 habitaciones y 2 cuartos de baño, e introduciendo el texto "Santa Cruz de Tenerife", el sistema muestra una lista de pisos que cumplen las tres condiciones*
- *Buscando con el texto "Tenerife", el sistema muestra la lista de pisos de toda la provincia de Santa Cruz de Tenerife*
- *En la lista, cada piso se muestra mediante una fotografía y el número de habitaciones que tiene*

Para responder si los ejemplos son verdaderos o falsos, ellos mismos descubren dudas sobre lo que necesitan para su negocio. Dejan de ir teniendo pensamientos mágicos para ser conscientes de la precisión con que tenemos que definir el funcionamiento del sistema. A partir de ese momento, entienden que la distancia entre los expertos en desarrollo y los expertos en negocio va menguando y dejan de preocuparse por diagramas abstractos. Entonces dicen... "¿Tenemos que pensar todas estas cosas?" Y tengo que contarles que, aunque los ordenadores hayan avanzado mucho, no dejan de ser máquinas muy tontas. Les cuento que si esas decisiones sobre el negocio no me las validan ellos, tendré que decidir yo, que no soy experto en su negocio. Así comienzan a involucrarse más en el desarrollo y todos comenzamos a hablar el mismo idioma.

Al final, todo esto no consiste en otra cosa que en escribir ejemplos e implementarlos.

## 3.3. ¿Está hecho o no?

Otra ventaja de dirigir el desarrollo por las historias y, a su vez, por los ejemplos, es que vamos a poder comprobar muy rápido si el programa está cumpliendo los objetivos o no. Conocemos en qué punto estamos y cómo vamos progresando. El Dueño de Producto puede revisar los tests de aceptación y ver cuántos se están cumpliendo, así que nuestro trabajo gana una

confianza tremenda. Es una gran manera de fomentar una estrecha colaboración entre todos los roles el equipo. Piénselo bien: ¡la propia máquina es capaz de decirnos si el programa cumple las especificaciones el cliente o no!

De cara a hacer modificaciones en nuevas versiones del programa y lanzarlo a producción, el tiempo que tardamos en efectuar las pruebas de regresión disminuye de manera drástica, lo cual se traduce en un ahorro considerable.

> Los tests de regresión deben su nombre al momento en que se ejecutan, no a su formato ni a otras características. Antes de lanzar una nueva versión de un producto en producción, ejecutamos todas las pruebas posibles, tanto manuales como automáticas para corroborar que tanto las nuevas funciones como las existentes funcionan. Regresión viene de regresar, puesto que regresamos a funcionalidad desarrollada en la versión anterior para validar que no se ha roto nada. Cuando no se dispone de una completa batería de tests, la regresión completa de una aplicación puede llevar varios días en los que el equipo de calidad ejecuta cada parte de la aplicación con todas y cada una de sus posibles variantes. Hablando en plata; una tortura y un gasto económico importante.

## 3.4. El contexto es esencial

Fuera del contexto ágil, ATDD tiene pocas probabilidades de éxito ya que si los analistas no trabajan estrechamente con los desarrolladores y testers, no se podrá originar un flujo de comunicación suficientemente rico como para que las preguntas y respuestas aporten valor al negocio. Si en lugar de ejemplos, se siguen escribiendo descripciones, estaremos aumentando la cantidad de trabajo considerablemente con lo cual el aumento de coste puede no retornar la inversión. Si los dueños de producto (cliente y analistas) no tienen tiempo para definir los tests de aceptación, no tiene sentido encargárselos a los desarrolladores, sería malgastar el dinero. Tener tiempo es un asunto muy relativo y muy delicado. No entraremos en ese tema tan escabroso al que yo llamaría más bien estar dispuestos a invertir el tiempo, más que tener o no tener tiempo. ¿Alguien tiene tiempo?. La herramienta con la que se escriben los tests de aceptación tiene que minimizar la cantidad de código que requiere esa conexión entre las frases y el código del sistema, si no, el mantenimiento se encarecerá demasiado. ATDD/STDD es un engranaje que cuesta poner en marcha pero que da sus frutos, como se puede leer en este artículo de la revista *Better Software* de 2004[2].

[2]http://industriallogic.com/papers/storytest.pdf

Desgraciadamente no podemos extendernos más con respecto a ATDD/STDD, si bien se podría escribir un libro sobre ello. Mike Cohn escribió uno muy popular titulado User Stories Applied[5] que le recomiendo encarecidamente leer. Mención especial también al capítulo sobre ATDD de Lasse Koskela en Test Driven[9] y los sucesivos, que incluyen ejemplos sobre el framework FIT. Gojko Adzic[1] tiene un libro basado en FitNesse y por supuesto cabe destacar su famoso libro sobre Acceptance Testing[2]. Elisabeth Hendrickson, en colaboración con otros expertos de la talla de Brian Marick, publicó un *paper* que puede leerse online[3] e incluye ejemplos en el framework Robot.

En la parte práctica de este libro tendremos ocasión de ver algunos ejemplos más aunque, por motivos de espacio, no es exhaustiva.

[3] http://testobsessed.com/wordpress/wp-content/uploads/2008/12/atddexample.pdf

Capítulo 4

# Tipos de test y su importancia

La nomenclatura sobre tests puede ser francamente caótica. Ha sido fuente de discusión en los últimos años y sigue sin existir universalmente de manera categórica. Cada equipo tiende a adoptar sus propias convenciones, ya que existen distintos aspectos a considerar para denominar tests. Por aspecto quiero decir que, según cómo se mire, el test se puede clasificar de una manera o de otra. Así se habla, por ejemplo, del aspecto visibilidad (si se sabe lo que hay dentro del SUT o no), del aspecto potestad (a quién pertenece el test), etc. Dale H. Emery hace una extensa recopilación de los posibles aspectos o puntos de vista y los tipos que alberga cada uno de ellos[1]. No es la única que existe, ni sus definiciones son extensas, pero nos da una idea de la complejidad del problema. Michael Feathers también enumera varias clasificaciones de tests en un post reciente[2]. Por su parte, la Wikipedia aporta otro punto de vista complementario a los anteriores[3].

Definitivamente, cada comunidad usa unos términos diferentes. Una comunidad podría ser la de los que practicamos TDD, y otra podría ser la de los que escriben tests para aplicaciones ya implementadas que todavía no los tienen, por ejemplo. Un mismo test puede ser de varios tipos, incluso mirándolo desde la perspectiva de un solo equipo de desarrollo ya que su clasificación dependerá del aspecto a considerar. No existen reglas universales para escribir todos y cada uno de los tipos de tests ni sus posibles combinaciones. Es una cuestión que llega a ser artesanal. Sin embargo, más allá de los términos, es conveniente que tengamos una idea de cómo es cada tipo de test, según las convenciones que hayamos elegido, para ser coherentes a la hora de escribirlos. Cada vez que vamos a programar un test, tenemos que

[1] http://cwd.dhemery.com/2004/04/dimensions/

[2] http://blog.objectmentor.com/articles/2009/04/13/x-tests-are-not-x-tests

[3] http://en.wikipedia.org/wiki/Software_testing

estar seguros de por qué lo escribimos y de qué estamos probando. Es extremadamente importante tener claro qué queremos afirmar con cada test y por qué lo hacemos de esa manera. Podría ser que el hecho de no saber determinar qué tipo de test vamos a programar, indique que no estamos seguros de por qué lo escribimos. Dicho de otro modo, si no conseguimos darle nombre a lo que hacemos, quizás no sepamos por qué lo hacemos. Probablemente no tengamos claro el diseño que queremos, o puede que el test no esté probando lo que debe, o que no estemos delimitando responsabilidades adecuadamente, o quizás estemos escribiendo más tests de los que son necesarios. Es una heurística a tener en cuenta.

En el ámbito de TDD no hablamos de tests desde el aspecto visibilidad (típicamente tests de caja blanca y caja negra). Usamos otros términos, pero sabemos que un test de caja negra podría coincidir con un test unitario basado en el estado. Y un test de caja blanca podría coincidir con un test unitario basado en la interacción. No es una regla exacta porque, dependiendo de cómo se haya escrito el test, puede que no sea unitario sino de integración. Lo importante es ser capaces de entender la naturaleza de los tests.

En la siguiente sección veremos los términos más comunes dentro de la comunidad TDD y sus significados.

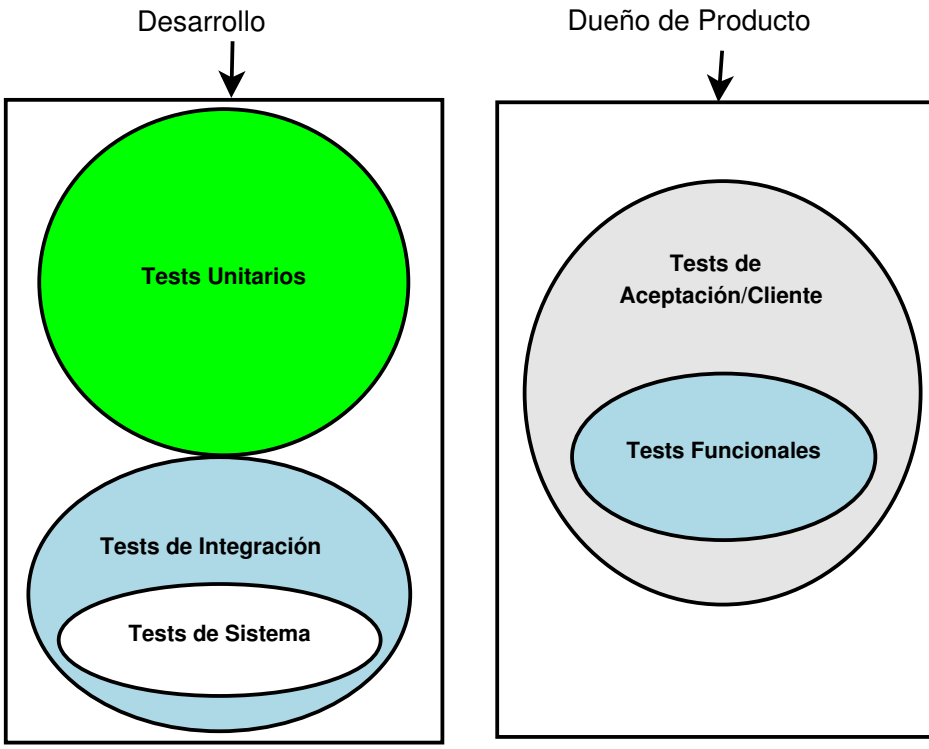
## 4.1. Terminología en la comunidad TDD

Desde el aspecto potestad, es decir, mirando los tests según a quién le pertenecen, distinguimos entre tests escritos por desarrolladores y tests escritos por el Dueño del Producto. Recordemos que el Dueño del Producto es el analista de negocio o bien el propio cliente. Lo ideal es que el analista de negocio ayude al cliente a escribir los tests para asegurarse de que las afirmaciones están totalmente libres de ambigüedad. Los tests que pertenecen al Dueño del Producto se llaman tests de cliente o de aceptación. Charlie Poole[4] prefiere llamarles tests de cliente ya que por aceptación se podría entender que se escriben al final cuando, realmente, no tiene que ser así. De hecho, en TDD partimos de tests de aceptación (ATDD) para conectar requerimientos con implementación, o sea, que los escribimos antes que nada. Cuando se escribe el código que permite ejecutar este test, y se ejecuta positivamente, se entiende que el cliente acepta el resultado. Por esto se habla de aceptación. Y también por esto es un término provocador, al haber clientes que niegan que un test de aceptación positivo signifique que aceptan esa parte del producto. Nosotros hablaremos de aceptación porque se usa más

[4]Uno de los padres de NUnit, con más de 30 años de experiencia: http://www.charliepoole.org/cp.php

en la literatura que test de cliente, aunque convendrá recordar lo peligrosa que puede llegar a ser esta denominación.

En el siguiente diagrama se muestra la clasificación de los tests típica de un entorno ATDD/TDD. A la izquierda, se agrupan los tests que pertenecen a desarrolladores y, a la derecha, los que pertenecen al Dueño del Producto. A su vez, algunos tipos de tests contienen a otros.




### 4.1.1. Tests de Aceptación

¿Cómo es un test de aceptación? Es un test que permite comprobar que el software cumple con un requisito de negocio. Como se vio en el capítulo de ATDD, un test de aceptación es un ejemplo escrito con el lenguaje del cliente pero que puede ser ejecutado por la máquina. Recordemos algunos ejemplos:

- *El producto X con precio 50€tiene un precio final de 55€después de aplicar el impuesto Z*
- *Si el paciente nació el 1 de junio de 1981, su edad es de 28 años en agosto de 2009*

¿Los tests de aceptación no usan la interfaz de usuario del programa? Podría ser que sí, pero en la mayoría de los casos la respuesta debe ser no. Los tests de carga y de rendimiento son de aceptación cuando el cliente los considera requisitos de negocio. Si el cliente no los requiere, serán tests de desarrollo.

### 4.1.2. Tests Funcionales

Todos los tests son en realidad funcionales, puesto que todos ejercitan alguna función del SUT[5], aunque en el nivel más elemental sea un método de una clase. No obstante, cuando se habla del aspecto funcional, se distingue entre test funcional y test no funcional. Un test funcional es un subconjunto de los tests de aceptación. Es decir, comprueban alguna funcionalidad con valor de negocio. Hasta ahora, todos los tests de aceptación que hemos visto son tests funcionales. Los tests de aceptación tienen un ámbito mayor porque hay requerimientos de negocio que hablan de tiempos de respuesta, capacidad de carga de la aplicación, etc; cuestiones que van más allá de la funcionalidad. Un test funcional es un test de aceptación pero, uno de aceptación, no tiene por qué ser funcional.

### 4.1.3. Tests de Sistema

Es el mayor de los tests de integración, ya que integra varias partes del sistema. Se trata de un test que puede ir, incluso, de extremo a extremo de la aplicación o del sistema. Se habla de sistema porque es un término más general que aplicación, pero no se refiere a administración de sistemas, no es que estemos probando el servidor web o el servidor SMTP aunque, tales servicios, podrían ser una parte de nuestro sistema. Así pues, un test del sistema se ejercita tal cual lo haría el usuario humano, usando los mismos puntos de entrada (aquí sí es la interfaz gráfica) y llegando a modificar la base de datos o lo que haya en el otro extremo. ¿Cómo se puede automatizar el uso de la interfaz de usuario y validar que funciona? Hay software

[5]Subject Under Test; el código que estamos probando

que permite hacerlo. Por ejemplo, si la interfaz de usuario es web, el plugin Selenium[6] para el navegador Mozilla Firefox[7] nos permite registrar nuestra actividad en una página web como si estuviéramos grabando un vídeo para luego reproducir la secuencia automáticamente y detectar cambios en la respuesta del sitio web. Pongamos que grabo la forma en que relleno un formulario con una dirección de correo electrónico incorrecta para que el sitio web me envíe un mensaje de error. Cada vez que quiera volver a comprobar que el sitio web responde igual ante esa entrada, sólo tengo que ejecutar el test generado por Selenium. Hay herramientas que permiten hacer lo mismo mediante programación: nos dan una API para seleccionar controles gráficos, y accionarlos desde código fuente, comprobando el estado de la ejecución con sentencias condicionales o asertivas. El propio Selenium lo permite. Una de las herramientas más populares es Watir[8] para Ruby y sus versiones para otros lenguajes de programación (Watin para .Net). Para aplicaciones escritas con el framework Django (Python), se utiliza el cliente web[9]. Para aplicaciones de escritorio, hay frameworks específicos como UIAutomation[10] o NUnitForms[11] que también permiten manipular la interfaz gráfica desde código.

Existen muchas formas de probar un sistema. Supongamos que hemos implementado un servidor web ligero y queremos validar que, cada vez que alguien accede a una página, se registra su dirección ip en un fichero de registro (log). Podríamos hacer un script con algún comando que se conecte a una URL del servidor, al estilo de Wget[12] desde la misma máquina y después buscar la ip 127.0.0.1 en el fichero de log con Grep[13]. Sirva el ejemplo para recalcar que no hay una sola herramienta ni forma de escribir tests de sistema, más bien depende de cada sistema.

Los tests de sistema son muy frágiles en el sentido de que cualquier cambio en cualquiera de las partes que componen el sistema, puede romperlos. No es recomendable escribir un gran número de ellos por su fragilidad. Si la cobertura de otros tipos de tests de granularidad más fina, como por ejemplo los unitarios, es amplia, la probabilidad de que los errores sólo se detecten con tests de sistema es muy baja. O sea, que si hemos ido haciendo TDD, no es productivo escribir tests de sistema para todas las posibles formas de uso del sistema, ya que esta redundancia se traduce en un aumento del costo

[6] http://seleniumhq.org/projects/ide/
[7] http://www.mozilla-europe.org/es/firefox/
[8] http://wtr.rubyforge.org/
[9] http://www.djangoproject.com/documentation/models/test_client/
[10] http://msdn.microsoft.com/en-us/library/ms747327.aspx
[11] http://nunitforms.sourceforge.net/
[12] http://es.wikipedia.org/wiki/GNU_Wget
[13] http://es.wikipedia.org/wiki/Grep

de mantenimiento de los tests. Por el contrario, si no tenemos escrito absolutamente ningún tipo de test, blindar la aplicación con tests de sistema será el paso más recomendable antes de hacer modificaciones en el código fuente. Luego, cuando ya hubiesen tests unitarios para los nuevos cambios introducidos, se podrían ir desechando tests de sistema.

¿Por qué se les llama tests de aceptación y tests funcionales a los tests de sistema? En mi opinión, es un error. Un test funcional es una frase escrita en lenguaje natural que utiliza el sistema para ejecutarse. En el caso de probar que una dirección de email es incorrecta, el test utilizará la parte del sistema que valida emails y devuelve mensajes de respuesta. Sin embargo, el requisito de negocio no debe entrar en cómo es el diseño de la interfaz de usuario. Por tanto, el test funcional no entraría a ejecutar el sistema desde el extremo de entrada que usa el usuario (la GUI), sino desde el que necesita para validar el requisito funcional. Si la mayoría de los criterios de aceptación se validan mediante tests funcionales, tan sólo nos harán falta unos pocos tests de sistema para comprobar que la GUI está bien conectada a la lógica de negocio. Esto hará que nuestros tests sean menos frágiles y estaremos alcanzando el mismo nivel de cobertura de posibles errores.

En la documentación de algunos frameworks, llaman test unitarios a tests que en verdad son de integración y, llaman tests funcionales, a tests que son de sistema. Llamar test funcional a un test de sistema no es un problema siempre que adoptemos esa convención en todo el equipo y todo el mundo sepa para qué es cada test.

En casos puntuales, un requisito de negocio podría involucrar la GUI, tal como pasa con el cliente de Gmail del iPhone por ejemplo. Está claro que el negocio en ese proyecto está directamente relacionado con la propia GUI. En ese caso, el test de sistema sería también un test funcional.

### 4.1.4. Tests Unitarios

Son los tests más importantes para el practicante TDD, los ineludibles. Cada test unitario o test unidad (unit test en inglés) es un paso que andamos en el camino de la implementación del software. Todo test unitario debe ser:

- Atómico
- Independiente
- Inocuo
- Rápido

Si no cumple estas premisas entonces no es un test unitario, aunque se ejecute con una herramienta tipo xUnit.

Atómico significa que el test prueba la mínima cantidad de funcionalidad posible. Esto es, probará un solo comportamiento de un método de una clase. El mismo método puede presentar distintas respuestas ante distintas entradas o distinto contexto. El test unitario se ocupará exclusivamente de uno de esos comportamientos, es decir, de un único camino de ejecución. A veces, la llamada al método provoca que internamente se invoque a otros métodos; cuando esto ocurre, decimos que el test tiene menor granularidad, o que es menos fino. Lo ideal es que los tests unitarios ataquen a métodos lo más planos posibles, es decir, que prueben lo que es indivisible. La razón es que un test atómico nos evita tener que usar el depurador para encontrar un defecto en el SUT, puesto que su causa será muy evidente. Como veremos en la parte práctica, hay veces que vale la pena ser menos estrictos con la atomicidad del test, para evitar abusar de los dobles de prueba.

Independiente significa que un test no puede depender de otros para producir un resultado satisfactorio. No puede ser parte de una secuencia de tests que se deba ejecutar en un determinado orden. Debe funcionar siempre igual independientemente de que se ejecuten otros tests o no.

Inocuo significa que no altera el estado del sistema. Al ejecutarlo una vez, produce exactamente el mismo resultado que al ejecutarlo veinte veces. No altera la base de datos, ni envía emails ni crea ficheros, ni los borra. Es como si no se hubiera ejecutado.

Rápido tiene que ser porque ejecutamos un gran número de tests cada pocos minutos y se ha demostrado que tener que esperar unos cuantos segundos cada rato, resulta muy improductivo. Un sólo test tendría que ejecutarse en una pequeña fracción de segundo. La rapidez es tan importante que Kent Beck ha desarrollado recientemente una herramienta que ejecuta los tests desde el IDE Eclipse mientras escribimos código, para evitar dejar de trabajar en código mientras esperamos por el resultado de la ejecución. Se llama JUnit Max[14]. Olof Bjarnason ha escrito otra similar y libre para Python[15]

Para conseguir cumplir estos requisitos, un test unitario aisla la parte del SUT que necesita ejercitar de tal manera que el resto está inactivo durante la ejecución. Hay principalmente dos formas de validar el resultado de la ejecución del test: validación del estado y validación de la interacción, o del comportamiento. En los siguientes capítulos los veremos en detalle con ejemplos de código.

[14] http://www.junitmax.com/junitmax/subscribe.html
[15] https://code.launchpad.net/ objarni/+junk/pytddmon

Los desarrolladores utilizamos los tests unitarios para asegurarnos de que el código funciona como esperamos que funcione, al igual que el cliente usa los tests de cliente para asegurarse que los requisitos de negocio se alcancen como se espera que lo hagan.

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

Como los acrónimos no dejan de estar de moda, cabe destacar que las características de los tests unitarios también se agrupan bajo las siglas F.I.R.S.T que vienen de: Fast, Independent, Repeatable, Small y Transparent. Repetible encaja con inocuo, pequeño caza con atómico y transparente quiere decir que el test debe comunicar perfectamente la intención del autor.

### 4.1.5. Tests de Integración

Por último, los tests de integración son la pieza del puzzle que nos faltaba para cubrir el hueco entre los tests unitarios y los de sistema. Los tests de integración se pueden ver como tests de sistema pequeños. Típicamente, también se escriben usando herramientas xUnit y tienen un aspecto parecido a los tests unitarios, sólo que estos pueden romper las reglas. Como su nombre indica, integración significa que ayuda a unir distintas partes del sistema. Un test de integración puede escribir y leer de base de datos para comprobar que, efectivamente, la lógica de negocio entiende datos reales. Es el complemento a los tests unitarios, donde habíamos "falseado" el acceso a datos para limitarnos a trabajar con la lógica de manera aislada. Un test de integración podría ser aquel que ejecuta la capa de negocio y después consulta la base de datos para afirmar que todo el proceso, desde negocio hacia abajo, fue bien. Son, por tanto, de granularidad más gruesa y más frágiles que los tests unitarios, con lo que el número de tests de integración tiende a ser menor que el número de tests unitarios. Una vez que se ha probado que dos módulos, objetos o capas se integran bien, no es necesario repetir el test para otra variante de la lógica de negocio; para eso habrán varios tests unitarios. Aunque los tests de integración pueden saltarse las reglas, por motivos de productividad es conveniente que traten de ser inocuos y rápidos. Si tiene que acceder a base de datos, es conveniente que luego la deje como estaba. Por eso, algunos frameworks para Ruby y Python entre otros, tienen la capacidad de crear una base de datos temporal antes de ejecutar la batería de tests, que se destruye al terminar las pruebas. Como se trata de una herramienta incorporada, también hay quien ejecuta los tests unitarios creando y destruyendo bases de datos temporales pero es una práctica que debe evitarse porque los segundos extra que se necesitan para eso nos hacen

perder concentración. Los tests unitarios deben pertenecer a suites[16] diferentes a los de integración para poderlos ejecutar por separado. En los próximos capítulos tendremos ocasión de ver tests de integración en detalle.

Concluimos el capítulo sin revisar otros tipos de tests, porque este no es un libro sobre cómo hacer pruebas de software exclusivamente sino sobre cómo construir software basándonos en ejemplos que ilustran los requerimientos del negocio sin ambigüedad. Los tests unitarios, de integración y de aceptación son los más importantes dentro del desarrollo dirigido por tests.

[16]Una suite es una agrupación de tests

Capítulo 5

# Tests unitarios y frameworks xUnit

En capítulos previos hemos citado xUnit repetidamente pero xUnit como tal no es ninguna herramienta en sí misma. La letra *x* es tan sólo un prefijo a modo de comodín para referirnos de manera genérica a los numerosos frameworks basados en el original SUnit. SUnit fue creado por Kent Beck para la plataforma SmallTalk y se ha portado a una gran variedad de lenguajes y plataformas como Java (JUnit), .Net (NUnit), Python (PyUnit), Ruby (Rubyunit), Perl (PerlUnit), C++ (CppUnit), etc. Si aprendemos a trabajar con NUnit y PyUnit como veremos en este libro, sabremos hacerlo con cualquier otro framework tipo xUnit porque la filosofía es siempre la misma. Además en Java, desde la versión 4 de JUnit, se soportan las anotaciones por lo que NUnit y JUnit se parecen todavía más.

Una clase que contiene tests se llama *test case* (conjunto de tests) y para definirla en código heredamos de la clase base *TestCase* del framework correspondiente (con JUnit 3 y Pyunit) o bien la marcamos con un atributo especial (JUnit 4 y NUnit). Los métodos de dicha clase pueden ser tests o no. Si lo son, serán tests unitarios o de integración, aunque podrían ser incluso de sistema. El framework no distingue el tipo de test que es, los ejecuta a todos por igual. Quienes debemos hacer la distinción somos nosotros mismos, una vez que tenemos claro qué queremos probar y por qué lo hacemos con un framework xUnit. En este capítulo todos los tests son unitarios.

En próximos capítulos veremos cómo escribir también tests de integración. Para etiquetar los métodos como tests, en Java y .Net usamos anotaciones y atributos respectivamente. En Python se hace precediendo al nombre del método con el prefijo `test` (ej: `def test_letter_A_isnot_a_number`), igual que pasaba con la versión 3 de JUnit. Los métodos que no están marca-

dos como tests, se utilizan para unificar código requerido por ellos. Es normal que haya código común para preparar los tests o para limpiar los restos su ejecución, por lo que xUnit provee una manera sencilla de organizar y compartir este código: los métodos especiales `SetUp` y `TearDown`. `SetUp` suele destinarse a crear variables, a definir el escenario adecuado para después llamar al SUT y `TearDown` a eliminar posibles objetos que no elimine el recolector de basura.

## 5.1. Las tres partes del test: AAA

Un test tiene tres partes, que se identifican con las siglas AAA en inglés: Arrange (Preparar), Act (Actuar), Assert (Afirmar).

Una parte de la preparación puede estar contenida en el método *SetUp*, si es común a todos los tests de la clase. Si la estapa de preparación es común a varios tests de la clase pero no a todos, entonces podemos definir otro método o función en la misma clase, que aúne tal código. No le pondremos la etiqueta de test sino que lo invocaremos desde cada punto en que lo necesitemos.

El acto consiste en hacer la llamada al código que queremos probar (SUT) y la afirmación o afirmaciones se hacen sobre el resultado de la ejecución, bien mediante validación del estado o bien mediante validación de la interacción. Se afirma que nuestras espectativas sobre el resultado se cumplen. Si no se cumplen el framework marcará en rojo cada falsa expectativa.

Veamos varios ejemplos en lenguaje C# con el framework NUnit. Tendremos que crear un nuevo proyecto de tipo librería con el IDE (VisualStudio, SharpDevelop, MonoDevelop...) e incorporar la DLL[1] *nunit.framework* a las referencias para disponer de su funcionalidad. Lo ideal es tener el SUT en una DLL y sus tests en otra. Así en el proyecto que contiene los tests, también se incluye la referencia a la DLL o ejecutable (.exe) del SUT, además de a NUnit. Estos ejemplos realizan la validación a través del estado. El estado del que hablamos es el de algún tipo de variable, no hablamos del estado del sistema. Recordemos que un test unitario no puede modificar el estado del sistema y que todos los tests de este capítulo son unitarios.

```
1 using System;
2 using NUnit.Framework;
3
4 namespace EjemplosNUnit
5 {
6     [TestFixture]
7     public class NameNormalizerTests
8     {
```

[1] Librería de enlace dinámico. Equivalente en .Net a los .jar de Java

```
9         [Test]
10        public void FirstLetterUpperCase()
11        {
12            // Arrange
13            string name = "pablo␣rodriguez";
14            NameNormalizer normalizer =
15                new NameNormalizer();
16            // Act
17            string result =
18                normalizer.FirstLetterUpperCase(
19                   name);
20            // Assert
21            Assert.AreEqual("Pablo␣Rodriguez", result);
22        }
23    }
24 }
```

Hemos indicado a NUnit que la clase es un conjunto de tests (test case), utilizando para ello el atributo `TestFixture`. El que la palabra *fixture* aparezca aquí, puede ser desconcertante, sería más claro si el atributo se llamase `TestCase`.

> El término *Fixture* se utiliza en realidad para hablar de los datos de contexto de los tests. Los datos de contexto o fixtures son aquellos que se necesitan para construir el escenario que require el test. En el ejemplo de arriba la variable `name` es una variable de contexto o fixture. Los datos de contexto no son exclusivamente variables sino que también pueden ser datos obtenidos de algún sistema de almacenamiento persistente. Es común que los tests de integración dependan de datos que tienen que existir en la base de datos. Estos datos que son un requisito previo, son igualmente datos de contexto o fixtures. De hecho se habla más de fixtures cuando son datos que cuando son variables. Algunos frameworks de tests como el de Django (que se basa en PyUnit) permiten definir conjuntos de datos de contexto mediante JSON[a] que son automáticamente insertados en la base de datos antes de cada test y borrados al terminar. Así, aunque los desarrolladores de NUnit decidieran llamar *TestFixture* al atributo que etiqueta un conjunto de tests, no debemos confundirnos con los datos de contexto. Charlie Poole comenta que es una buena idea agrupar tests dentro de un mismo conjunto cuando sus datos de contexto son comunes, por eso optaron por llamarle *TestFixture* en lugar de *TestCase*.
>
> [a] Un diccionario con sintaxis javascript

El nombre que le hemos puesto a la clase describe el conjunto de los tests

que va a contener. Debemos utilizar conjuntos de tests distintos para probar grupos de funcionalidad distinta o lo que es lo mismo: no se deben incluir todos los tests de toda la aplicación en un solo conjunto de tests (una sola clase).

En nuestro ejemplo la clase contiene un único test que está marcado con el atributo `Test`. Los tests siempre son de tipo `void` y sin parámetros de entrada. El nombre del test es largo porque es autoexplicativo. Es la mejor forma de documentarlo. Poner un comentario de cabecera al test, es un antipatrón porque vamos a terminar con un gran número de tests y el esfuerzo de mantener todos sus comentarios es muy elevado. De hecho es un error que el comentario no coincida con lo que hace el código y eso pasa cuando modificamos el código después de haber escrito el comentario. No importa que el nombre del método tenga cincuenta letras, no le hace daño a nadie. Si no sabemos cómo resumir lo que hace el test en menos de setenta letras, entonces lo más probable es que tampoco sepamos qué test vamos a escribir, qué misión cumple. Es una forma de detectar un mal diseño, bien del test o bien del SUT. A veces cuando un código requiere documentación es porque no está lo suficientemente claro.

En el cuerpo del test aparecen sus tres partes delimitadas con comentarios. En la práctica nunca delimitamos con comentarios, aquí está escrito meramente con fines docentes. La finalidad del test del ejemplo es comprobar que el método `FirstLetterUpperCase` de la clase `NameNormalizer` es capaz de poner en mayúscula la primera letra de cada palabra en una frase. Es un test de validación de estado porque hacemos la afirmación de que funciona basándonos en el estado de una variable. *Assert* en inglés viene a significar afirmar. La última línea dice: Afirma que la variable `result` es igual a "Pablo Rodriguez". Cuando NUnit ejecute el método, dará positivo si la afirmación es cierta o negativo si no lo es. Al positivo le llamamos luz verde porque es el color que emplea la interfaz gráfica de NUnit o símplemente decimos que el test pasa. Al resultado negativo le llamamos luz roja o bien decimos que el test no pasa.

Imaginemos que el código del SUT ya está implementado y el test da luz verde. Pasemos al siguiente ejemplo recalcando que todavía no estamos practicando TDD, sino símplemente explicando el funcionamiento de un framework xUnit.

```
1 [Test]
2 public void SurnameFirst()
3 {
4     string name = "gonzalo␣aller";
5     NameNormalizer normalizer =
6          new NameNormalizer();
7     string result =
8          normalizer.SurnameFirst(name);
```

```
9       Assert.AreEqual("aller, gonzalo", result);
10  }
```

Es otro test de validación de estado que creamos dentro del mismo conjunto de tests porque el SUT es un método de la misma clase que antes. Lo que el test comprueba es que el método `SurnameFirst` es capaz de recibir un nombre completo y devolver el apellido por delante, separado por una coma. Si nos fijamos bien vemos que la declaración de la variable `normalizer` es idéntica en ambos tests. A fin de eliminar código duplicado la movemos hacia el `SetUp`.

El conjunto queda de la siguiente manera:

```
1   namespace EjemplosNUnit
2   {
3       [TestFixture]
4       public class NameNormalizerTests
5       {
6           NameNormalizer _normalizer;
7
8           [SetUp]
9           public void SetUp()
10          {
11              _normalizer =
12                  new NameNormalizer();
13          }
14
15          [Test]
16          public void FirstLetterUpperCase()
17          {
18              string name = "pablo rodriguez";
19              string result =
20                  _normalizer.FirstLetterUpperCase(
21                      name);
22              Assert.AreEqual("Pablo Rodriguez", result);
23          }
24
25          [Test]
26          public void SurnameFirst()
27          {
28              string name = "gonzalo aller";
29              string result =
30                  _normalizer.SurnameFirst(
31                      name);
32              Assert.AreEqual("aller, gonzalo", result);
33          }
34      }
35  }
```

Antes de cada uno de los dos tests el framework invocará al método `SetUp` recordándonos que cada prueba es independiente de las demás.

Hemos definido `_normalizer` como un miembro privado del conjunto de tests. El guión bajo (underscore) que da comienzo al nombre de la variable, es una regla de estilo que nos ayuda a identificarla rápidamente como variable

de la clase en cualquier parte del código[2]. El método `SetUp` crea una nueva instancia de dicha variable asegurándonos que entre la ejecución de un test y la de otro, se destruye y se vuelve a crear, evitándo efectos colaterales. Por tanto lo que un test haga con la variable `_normalizer` no afecta a ningún otro. Podríamos haber extraido también la variable `name` de los tests pero como no se usa nada más que para alimentar al SUT y no interviene en la fase de afirmación, lo mejor es liquidarla:

```
1  namespace EjemplosNUnit
2  {
3      [TestFixture]
4      public class NameNormalizerTests
5      {
6          NameNormalizer _normalizer;
7
8          [SetUp]
9          public void SetUp()
10         {
11             _normalizer =
12                 new NameNormalizer();
13         }
14
15         [Test]
16         public void FirstLetterUpperCase()
17         {
18             string result =
19                 _normalizer.FirstLetterUpperCase(
20                     "pablo␣rodriguez");
21             Assert.AreEqual("Pablo␣Rodriguez", result);
22         }
23
24         [Test]
25         public void SurnameFirst()
26         {
27             string result =
28                 _normalizer.SurnameFirst(
29                     "gonzalo␣aller");
30             Assert.AreEqual("aller,␣gonzalo", result);
31         }
32     }
33 }
```

Nos está quedando un conjunto de tests tan bonito como los muebles que hacen en el programa de Bricomanía de la televisión. No hemos definido método `tearDown` porque no hay nada que limpiar explícitamente. El recolector de basura es capaz de liberar la memoria que hemos reservado; no hemos dejado ninguna referencia muerta por el camino.

La validación de estado generalmente no tiene mayor complicación, salvo

[2]Uncle Bob en Clean Code[11] y Xavier Gost en el Agile Open 2009 me han convencido definitivamente para que deje de utilizar esta regla en mi código pero encuentro que en el papel ayudará. No aconsejo utilizarla si disponemos de un IDE (pero el libro no lo es)

que la ejecución del SUT implique cambios en el sistema y tengamos que evitarlos para respetar las cláusulas que definen un test unitario. Veremos ejemplos en los próximos capítulos.

Continuemos con la validación de excepciones. Se considera validación de comportamiento, pues no se valida estado ni tampoco interacción entre colaboradores. Supongamos que a cualquiera de las funciones anteriores, pasamos como parámetro una cadena vacía. Para tal entrada queremos que el SUT lanze una excepción de tipo `EmptyNameException`, definida por nuestra propia aplicación. ¿Cómo escribimos esta prueba con NUnit?

```
1  [Test]
2  public void ThrowOnEmptyName()
3  {
4      try
5      {
6          _normalizer.SurnameFirst("");
7          Assert.Fail(
8              "Exception␣should␣be␣thrown");
9      }
10     catch (EmptyNameException){}
11 }
```

Cuando un test se ejecuta sin que una excepción lo aborte, éste pasa, aunque no haya ninguna afirmación. Es decir, cuando no hay afirmaciones y ninguna excepción interrumpe el test, se considera que funciona. En el ejemplo, esperamos que al invocar a `SurnameFirst`, el SUT lanze una excepción de tipo concreto. Por eso colocamos un bloque `catch`, para que el test no se interrumpa. Dentro de ese bloque no hay nada, así que la ejecución del test termina. Entonces se considera que el SUT se comporta como deseamos. Si por el contrario la ejecución del SUT termina y no se ha lanzado la excepción esperada, la llamada a `Assert.Fail` abortaría el test explícitamente señalando luz roja. Se puede escribir el mismo test ayudándonos de atributos especiales que tanto NUnit como JUnit (en este caso son anotaciones) incluyen.

```
1  [Test]
2  [ExpectedException("EmptyNameException",
3  ExpectedMessage="The␣name␣can␣not␣be␣empty" )]
4  public void ThrowOnEmptyName()
5  {
6      _normalizer.SurnameFirst("");
7  }
```

El funcionamiento y significado de los dos últimos tests es exactamente el mismo. Cuando se quiere probar que se lanza una excepción, se debe expresar exactamente cuál es el tipo de la excepción esperada y no capturar la excepción genérica (System.Exception en .Net). Si usamos la excepción genérica, estaremos escondiendo posibles fallos del SUT, excepciones inesperadas. Además, en el caso de PyUnit la llamada a `fail` no sólo detiene el

test marcándolo en rojo sino que además lanza una excepción, con lo cual el bloque `catch` la captura y obtenemos un resultado totalmente confuso. Creeríamos que el SUT está lanzando la excepción cuando en realidad no lo hace pero no lo advertiríamos.

Llega el turno de la validación de interacción. La validación de interacción es el recurso que usamos cuando no es posible hacer validación de estado. Es un tipo de validación de comportamiento. Es recomendable recurrir a esta técnica lo menos posible, porque los tests que validan interacción necesitan conocer cómo funciona por dentro el SUT y por tanto son más frágiles. La mayoría de las veces, se puede validar estado aunque no sea evidente a simple vista. Quizás tengamos que consultarlo a través de alguna propiedad del SUT en vez de limitarnos a un valor devuelto por una función. Si el método a prueba es de tipo `void`, no se puede afirmar sobre el resultado pero es posible que algún otro miembro de la clase refleje un cambio de estado.

El caso de validación de interacción más común es el de una colaboración que implica alteraciones en el sistema. Elementos que modifican el estado del sistema son por ejemplo las clases que acceden a la base de datos o que envían mensajes a través de un servicio web (u otra comunicación que salga fuera del dominio de nuestra aplicación) o que crean datos en un sistema de ficheros. Cuando el SUT debe colaborar con una clase que guarda en base de datos, tenemos que validar que la interacción entre ambas partes se produce y al mismo tiempo evitar que realmente se acceda a la base de datos. No queremos probar toda la cadena desde el SUT hacia abajo hasta el sistema de almacenamiento. El test unitario pretende probar exclusivamente el SUT. Tratamos de aislarlo todo lo posible. Luego ya habrá un test de integración que se encargue de verificar el acceso a base de datos. Para llevar a cabo este tipo de validaciones es fundamental la inyección de dependencias[3]. Si los miembros de la colaboración no se han definido con la posibilidad de inyectar uno en el otro, difícilmente podremos conseguir respetar las reglas de los tests unitarios. Con lenguajes interpretados como Python, es posible pero el código del test termina siendo complejo de entender y mantener, es sólo una opción temporal.

Un ejemplo vale más que las palabras, así que imaginemos un sistema que gestiona el expediente académico de los alumnos de un centro de formación. Hay una función que dado el identificador de un alumno (su número de expediente) y la nota de un examen, actualiza su perfil en base de datos. Supongamos que existen los objetos relacionales `Student` y `Score` y una clase `DataManager` capaz de recuperarlos y guardarlos en base de datos. El SUT se llama `ScoreManager` y su colaborador será `DataManager`, que

[3]Ver Capítulo 7 en la página 111

implementa la interfaz `IDataManager`. Las fases de preparación y acción las sabemos escribir ya:

```
[Test]
public void AddStudentScore()
{
    ScoreManager smanager = new ScoreManager();
    smanager.AddScore("23145", 8.5);
}
```

Pero el método `AddScore` no devuelve nada y además estará actualizando la base de datos. ¿Cómo validamos? Lo primero es hacer que el colaborador de `ScoreManager` se pueda inyectar:

```
[Test]
public void AddStudentScore()
{
    IDataManager dataManager= new DataManager();
    ScoreManager smanager = new ScoreManager(dataManager);

    smanager.AddScore("23145", 8.5);
}
```

La validación de la interacción se hace con frameworks de objetos *mock*, como muestra el siguiente capítulo pero para comprender parte de lo que hacen internamente los mocks y resolver este test sin ninguna otra herramienta externa, vamos a implementar la solución manualmente. Si el SUT va a invocar a su colaborador para leer de la base de datos, operar y guardar, podemos inyectar una instancia que se haga pasar por buena pero que en verdad no acceda a tal base de datos.

```
public class MockDataManager : IDataManager
{
    public IRelationalObject GetByKey(string key)
    {
        return new Student();
    }

    public void Save(IRelationalObject robject)
    {}

    public void Create(IRelationalObject robject)
    {}
}
```

La interfaz `IDataManager` tiene tres métodos; uno para obtener un objeto relacional dada su clave primaria, otro para guardarlo cuando se ha modificado y el último para crearlo en base de datos por primera vez. Actualizamos el test:

```
[Test]
public void AddStudentScore()
{
```

```
4       MockDataManager dataManager= new MockDataManager();
5       ScoreManager smanager = new ScoreManager(dataManager);
6
7       smanager.AddScore("23145", 8.5);
8   }
```

Vale, ya no accederá al sistema de almacenamiento porque la clase no implementa nada ¿pero cómo validamos que el SUT intenta hacerlo? Al fin y al cabo la misión de nuestro SUT no es meramente operar sino también coordinar el registro de datos. Tendremos que añadir algunas variables de estado internas para controlarlo:

```
1   public class MockDataManager : IDataManager
2   {
3       private bool _getKeyCalled = false;
4       private bool _saveCalled = false;
5
6       public IRelationalObject GetByKey(string key)
7       {
8           _getKeyCalled = true;
9           return new Student();
10      }
11
12      public void Save(IRelationalObject robject)
13      {
14          _saveCalled = true;
15      }
16
17      public void VerifyCalls()
18      {
19          if (!_saveCalled)
20              throw Exception("Save␣method␣was␣not␣called");
21          if (!_getKeyCalled)
22              throw Exception("GetByKey␣method␣was␣not␣called");
23      }
24
25      public void Create(IRelationalObject robject)
26      {}
27  }
```

Ya podemos hacer afirmaciones (en este caso verificaciones) sobre el resultado de la ejecución:

```
1   [Test]
2   public void AddStudentScore()
3   {
4       MockDataManager dataManager= new MockDataManager();
5       ScoreManager smanager = new ScoreManager(dataManager);
6
7       smanager.AddScore("23145", 8.5);
8
9       dataManager.VerifyCalls();
10  }
```

No hemos tocado la base de datos y estamos validando que el SUT hace esas dos llamadas a su colaborador. Sin embargo, la solución propuesta es costosa

de implementar y no contiene toda la información que necesitamos (¿cómo sabemos que el dato que se salvó era el correcto?). En el siguiente capítulo veremos una solución alternativa, los mocks generador por frameworks, que nos permitirá definir afirmaciones certeras basadas en expectativas con todo lujo de detalles.

Como lectura adicional recomiendo el libro de J.B Rainsberg[16]. Además el blog y los videos de las conferencias de este autor son una joya.

Capítulo 6

# Mocks y otros dobles de prueba

Antes de decidirnos a usar objetos mock (en adelante mocks) hay que contar hasta diez y pensarlo dos veces. Lo primero, es saber en todo momento qué es lo que vamos a probar y por qué. En las listas de correo a menudo la gente pregunta cómo deben usar mocks para un problema determinado y buena parte de las respuestas concluyen que no necesitan mocks, sino partir su test en varios y/o reescribir una parte del SUT. Los mocks presentan dos inconvenientes fundamentales:

- El código del test puede llegar a ser difícil de leer.
- El test corre el riesgo de volverse frágil si conoce demasiado bien el interior del SUT.

Frágil significa que un cambio en el SUT, por pequeño que sea, romperá el test forzándonos a reescribirlo. La gran ventaja de los mocks es que reducen drásticamente el número de líneas de código de los tests de validación de interacción y evitan que el SUT contenga *hacks* (apaños) para validar. En los tests de validación de estado, también se usan mocks o stubs cuando hay que acceder a datos procedentes de un colaborador. Por lo tanto, los mocks (y otros dobles) son imprescindibles para un desarrollo dirigido por tests completo pero, igualmente importante, es saber cuándo van a poner en jaque al test, o sea, cuándo debemos evitarlos.

Un mock es un tipo concreto de doble de test. La expresión "doble" se usa en el mismo sentido de los actores "dobles" en las películas de acción, ya que se hace pasar por un colaborador del SUT cuando en realidad no es la entidad que dice ser. Gerard Meszaros describe los distintos tipos de dobles de test en su libro[12] donde, además, sienta las bases de la nomenclatura. Martin Fowler publicó un artículo que se ha hecho muy popular basado en

esta nomenclatura; "Los mocks no son stubs", donde habla de los distintos dobles[1]. De ahí extraemos el siguiente listado de tipos de doble:

- Dummy: se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- Fake: tiene una implementación que realmente funciona pero, por lo general, toma algún atajo o cortocircuito que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- Stub: proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas; tal como una pasarela de email que recuerda cuántos mensajes envió.
- Mock: objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles. Las veremos a continuación.

El stub[2] es como un mock con menor potencia, un subconjunto de su funcionalidad. Mientras que en el mock podemos definir expectativas con todo lujo de detalles, el stub tan sólo devuelve respuestas preprogramadas a posibles llamadas. Un mock valida comportamiento en la colaboración, mientras que el stub simplemente simula respuestas a consultas. El stub hace el test menos frágil pero, al mismo tiempo, nos aporta menos información sobre la colaboración entre objetos. Para poder discernir entre usar un mock o un stub, volvemos a recalcar que primero hay que saber qué estamos probando y por qué. Los mocks tienen ventajas e inconvenientes sobre los stubs. Lo mejor será mostrarlas con ejemplos.

Los frameworks que generan mocks y stubs son muy ingeniosos. Son capaces de crear una clase en tiempo de ejecución, que hereda de una clase `X` o que implementa una interfaz `Y`. Tanto `X` como `Y` se pueden pasar como parámetro para que el framework genere una instancia de un mock o un stub que sea de ese tipo pero cuya implementación simplemente se limita a reaccionar tal como le indiquemos que debe hacerlo. En este libro usaremos el framework Rhino.Mocks versión 3.6 para .Net, EasyMock 2.5.2 para Java,

[1] Mi buen amigo Eladio López y yo lo tradujimos, aunque a día de hoy la traducción necesita ser mejorada: http://www.carlosble.com/traducciones/mocksArentStubs.html

[2] La traducción de stub sería *sobra* o *colilla*, así que todo el mundo ha optado por dejarlo en stub

Mockito 1.8.2 para Java y Pymox 0.5.1 para Python. Todos son software libre y se pueden descargar gratuitamente de la red.

Este capítulo posiblemente sea de los más difíciles de entender de todo el libro. Para una correcta comprensión del mismo es recomendable leerlo con el ordenador delante para ir ejecutando los ejemplos.

## 6.1. Cuándo usar un objeto real, un stub o un mock

Vamos a por el primer ejemplo para sacar a relucir los pros y los contras de las distintas alternativas que tenemos para diseñar código que trata de colaboraciones entre objetos. Supongamos que hemos escrito un control gráfico que muestra un calendario en pantalla para permitirnos seleccionar una fecha. Ahora nos piden que dibujemos los días festivos de determinados municipios en un color distinto, para lo cual tenemos que consultar un servicio remoto que lee, de una base de datos, los días festivos. El servicio necesita conocer qué año, qué mes y qué municipio nos ocupa, para devolver un vector con los días festivos del mes. La interfaz del servicio remoto es la siguiente:

```
1 public interface ICalendarService
2 {
3     int[] GetHolidays(int year,
4                   int month,
5                   string townCode);
6 }
```

El método es de lectura, de consulta. Lo que queremos diseñar es el trozo de código, de nuestra aplicación cliente, que obtiene los días festivos del servidor remoto. El SUT es el calendario cliente y su colaborador el servicio remoto. Estamos hablando de una aplicación cliente/servidor. Para diseñar la colaboración, no vamos a utilizar el servicio real porque el test no sería unitario, no se ejecutaría de manera veloz ni con independencia del entorno. Está claro que necesitamos un doble. ¿Qué tipo de doble de prueba utilizaremos? Generalmente, para los métodos de consulta se usan stubs pero el factor determinante para decantarse por un mock o un stub es el nivel de especificidad que se requiere en la colaboración. Vamos a estudiar las dos posibilidades resaltando las diferencias de cada una. Utilizaremos Rhino.Mocks para los ejemplos. Lo primero es añadir al proyecto las referencias a las DLL[3] *Rhino.Mocks*, *Castle.Core* y *Castle.DynamicProxy2*. Nuestro calendario cliente tiene tres propiedades que son `CurrentYear` `CurrentMonth`, `CurrentTown` que sirven para configurarlo.

[3]incluidas en el código fuente que acompaña al libro

El test, usando un mock, sería el siguiente:

```
using System;
using System.Collections.Generic;
using NUnit.Framework;
using Rhino.Mocks;

[TestFixture]
public class CalendarTests
{
 [Test]
 public void ClientAsksCalendarService()
 {
    int year = 2009;
    int month = 2;
    string townCode = "b002";
    ICalendarService serviceMock =
        MockRepository.GenerateStrictMock<ICalendarService>();
    serviceMock.Expect(
      x => x.GetHolidays(
       year, month, townCode)).Return(new int[] { 1, 5 });

    Calendar calendar = new Calendar(serviceMock);
    calendar.CurrentTown = townCode;
    calendar.CurrentYear = year;
    calendar.CurrentMonth = month;
    calendar.DrawMonth(); // the SUT

    serviceMock.VerifyAllExpectations();
 }
}
```

El código asusta un poco al principio pero, si lo estudiamos, veremos que no es tan complejo. En él, se distinguen tres bloques separados por líneas en blanco (AAA[4]). El primer bloque consiste en la generación del mock y la definición de expectativas. Con la llamada a `GenerateStrictMock`, el framework genera una instancia de una clase que implementa la interfaz `ICalendarService`. Nos ahorramos crear una clase impostora a mano como hicimos en el capítulo anterior. La siguiente línea define la primera expectativa (`Expect`) y dice que, sobre el propio objeto mock, en algún momento, se invocará al método `GetHolidays` con sus tres parámetros. Y además, dice que, cuando esa invocación se haga, el mock devolverá un array de dos elementos, 1 y 5. O sea, estamos diciéndole al mock que le van a invocar de esa manera y que, cuando ocurra, queremos que se comporte tal cual.

El siguiente bloque ya es el de acto (Act), donde se invoca al SUT. La última línea es la que verifica que todo fue según lo esperado (Assert), la que le dice al mock que compruebe que la expectativa se cumplió. Si no se cumplió, entonces el test no pasa porque el framework lanza una excepción. Que no se cumplió significa que la llamada nunca se hizo o que se hizo con

[4]Las tres partes de un test: Arrange, Act, Assert

otros parámetros distintos a los que explícitamente se pusieron en la expectativa o bien que se hizo más de una vez. Además, si en el acto se hacen llamadas al mock que no estaban contempladas en las expectativas (puesto que solo hay una expectativa, cualquier otra llamada al servicio sería no-contemplada), el framework hace fallar el test. La ausencia de expectativa supone fallo, si se produce alguna interacción entre SUT y mock, al margen de la descrita explícitamente. Esta es una restricción o una validación importante, según cómo se mire. Si el colaborador fuese un stub, la verificación (y por tanto sus restricciones), no se aplicaría, como veremos a continuación. Desde luego, el framework está haciendo una gran cantidad de trabajo por nosotros, ahorrándonos una buena suma de líneas de código y evitándonos código específico de validación dentro del SUT.

Si por motivos de rendimiento, por ejemplo, queremos obligar a que el SUT se comunique una única vez con el colaborador, siendo además de la forma que dicta el test, entonces un mock está bien como colaborador. Cualquier cosa que se salga de lo que pone el test, se traducirá en luz roja. Digo rendimiento porque quizás queremos cuidarnos del caso en que el calendario hiciese varias llamadas al servicio por despiste del programador o por cualquier otro motivo.

El código del SUT que hace pasar el test sería algo como:

```
1  public void DrawMonth()
2  {
3      // ... some business code here ...
4      int[] holidays =
5           _calendarService.GetHolidays(_currentYear,
6                _currentMonth, _currentTown);
7      // ... rest of business logic here ...
8  }
```

¿Cómo lo haríamos con un stub? ¿qué implicaciones tiene?. El stub no dispone de verificación de expectativas[5] sino que hay que usar el `Assert` de NUnit para validar el estado. En el presente ejemplo, podemos validar el estado, definiendo en el calendario cliente alguna propiedad `Holidays` de tipo array de enteros que almacenase la respuesta del servidor para poder afirmar sobre él. Al recurrir al stub, nos aseguramos que el SUT es capaz de funcionar puesto que, cuando invoque a su colaborador, obtendrá respuesta. El stub, al igual que el mock, simulará al servicio devolviendo unos valores:

```
1  [Test]
2  public void DrawHolidaysWithStub()
```

[5]Es decir, la llamada a `VerifyAllExpectations`. Aunque en realidad dicha función sí forma parte de la API para stubs en Rhino.Mocks, no verifica nada, siempre da un resultado positivo. Existe la posibilidad de llamar a `AssertWasCalled` pero el propio Ayende, autor de Rhino.Mocks no está seguro de que sea correcto según la definición de stub, con lo que podría optar por eliminarla en futuras versiones.

```
3  {
4      int year = 2009;
5      int month = 2;
6      string townCode = "b002";
7      ICalendarService serviceStub =
8          MockRepository.GenerateStub<ICalendarService>();
9      serviceStub.Stub(
10         x => x.GetHolidays(year, month, townCode)).Return(
11                         new int[] { 1, 5 });
12
13     Calendar calendar = new Calendar(serviceStub);
14     calendar.CurrentTown = townCode;
15     calendar.CurrentYear = year;
16     calendar.CurrentMonth = month;
17     calendar.DrawMonth();
18
19     Assert.AreEqual(1, calendar.Holidays[0]);
20     Assert.AreEqual(5, calendar.Holidays[1]);
21 }
```

La diferencia es que este test mantendría la luz verde incluso aunque no se llamase a `GetHolidays`, siempre que la propiedad `Holidays` de `calendar` tuviese los valores indicados en las afirmaciones del final. También pasaría aunque la llamada se hiciese cien veces y aunque se hicieran llamadas a otros métodos del servicio. Al ser menos restrictivo, el test es menos frágil que su versión con un mock. Sin embargo, nos queda sensación de que no sabemos si la llamada al colaborador se está haciendo o no. Para salir de dudas, hay que plantearse cuál es el verdadero objetivo del test. Si se trata de describir la comunicación entre calendario y servicio con total precisión, usaría un mock. Si me basta con que el calendario obtenga los días festivos y trabaje con ellos, usaría un stub. Cuando no estamos interesados en controlar con total exactitud la forma y el número de llamadas que se van a hacer al colaborador, también podemos utilizar un stub. Es decir, para todos aquellos casos en los que le pedimos al framework... "si se produce esta llamada, entonces devuelve X", independientemente de que la llamada se produzca una o mil veces. Digamos que son atajos para simular el entorno y que se den las condiciones oportunas. Al fin y al cabo, siempre podemos cubrir el código con un test de integración posterior que nos asegure que todas las partes trabajan bien juntas.

A continuación, vamos a por un ejemplo que nos hará dudar sobre el tipo de doble a usar o, incluso, si conviene un doble o no. Se trata de un software de facturación. Tenemos los objetos, `Invoice` (factura), `Line` y `TaxManager` (gestor de impuestos). El objeto factura necesita colaborar con el gestor de impuestos para calcular el importe a sumar al total, ya que el porcentaje de los mismos puede variar dependiendo de los artículos y dependiendo de la región. Una factura se compone de una o más líneas y cada línea contiene el artículo y la cantidad. Nos interesa inyectar el gestor de impuestos en el constructor

de la factura para que podamos tener distintos gestores correspondientes a distintos impuestos. Así, si estoy en Madrid inyectaré el `IvaManager` y si estoy en Canarias[6] el `IgicManager`.

¿Cómo vamos a probar esta colaboración? ¿utilizaremos un objeto real? ¿un stub? ¿un mock tal vez?. Partimos de la base de que el gestor de impuestos ya ha sido implementado. Puesto que no altera el sistema y produce una respuesta rápida, yo utilizaría el objeto real:

```
1  [TestFixture]
2  public class InvoiceTests
3  {
4      [Test]
5      public void CalculateTaxes()
6      {
7          Stock stock = new Stock();
8          Product product = stock.GetProductWithCode("x1abc3t3c");
9          Line line = new Line();
10         int quantity = 10;
11         line.AddProducts(product, quantity);
12         Invoice invoice = new Invoice(new TaxManager());
13         invoice.AddLine(line);
14
15         float total = invoice.GetTotal();
16
17         Assert.Greater(quantity * product.Price, total);
18     }
19 }
```

Las tres partes del test están separadas por líneas en blanco. En la afirmación, nos limitamos a decir que el total debe ser mayor que la simple multiplicación del precio del producto por la cantidad de productos. Usar el colaborador real (`TaxManager`) tiene la ventaja de que el código del test es sencillo y de que, los posibles defectos que tenga, probablemente sean detectados en este test. Sin embargo, el objetivo del test no es probar `TaxManager` (el colaborador) sino probar `Invoice` (el SUT). Visto así, resulta que si usamos un doble para `TaxManager`, entonces el SUT queda perfectamente aislado y este test no se rompe aunque se introduzcan defectos en el colaborador. La elección no es fácil. Personalmente, prefiero usar el colaborador real en estos casos en que no altera el sistema y se ejecuta rápido. A pesar de que el test se puede romper por causas ajenas al SUT, ir haciendo TDD me garantiza que, en el instante siguiente a la generación del defecto, la alerta roja se va a activar, con lo que detectar y corregir el error será cuestión de segundos. Los mocks no se inventaron para aislar dependencias sino para diseñar colaboraciones, aunque el aislamiento es un aspecto secundario que suele resultar beneficioso. Se puede argumentar que, al no haber usado un mock, no tenemos garantías de que el cálculo del impuesto fuese realizado por el gestor

[6] En Canarias no se aplica el IVA sino un impuesto llamado IGIC

de impuestos. Podría haberse calculado dentro del mismo objeto factura sin hacer uso de su colaborador. Pero eso supondría que, forzosamente, hemos producido código duplicado; replicado del gestor de impuestos. El tercer paso del algoritmo TDD consiste en eliminar la duplicidad, por lo que, si lo estamos aplicando, no es obligatorio que un mock nos garantice que se hizo la llamada al colaborador. La validación de estado que hemos hecho, junto con la ausencia de duplicidad, son suficientes para afirmar que el código va por buen camino. Así pues, la técnica no es la misma si estamos haciendo TDD que si estamos escribiendo pruebas de software a secas. Las consideraciones difieren. Si el gestor de impuestos accediese a la base de datos, escribiese en un fichero en disco o enviase un email, entonces seguro que hubiese utilizado un doble. Si no estuviese implementado todavía y me viese obligado a diseñar primero esta funcionalidad de la factura, entonces seguro usaría un stub para simular que la funcionalidad del gestor de impuestos está hecha y produce el resultado que quiero.

En palabras de Steve Freeman[14]: *"utilizamos mocks cuando el servicio cambia el mundo exterior; stubs cuando no lo hace - stubs para consultas y mocks para acciones". "Usa un stub para métodos de consulta y un mock para suplantar acciones"*. Es una regla que nos orientará en muchos casos. Aplicada al ejemplo anterior, funciona; usaríamos un stub si el `TaxManager` no estuviese implementado, ya que le hacemos una consulta.

Hay veces en las que un mismo test requiere de mocks y stubs a la vez, ya que es común que un SUT tenga varios colaboradores, siendo algunos stubs y otros mocks, dependiendo de la intención del test.

Antes de pasar a estudiar las peculiaridades de EasyMock veamos un ejemplo más complejo. Se trata del ejemplo de los estudiantes del capítulo anterior. Escribimos el mismo test con la ayuda de Rhino.Mocks:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using NUnit.Framework;
5  using Rhino.Mocks;
6
7  [Test]
8  public void AddStudentScore()
9  {
10     string studentId = "23145";
11     float score = 8.5f;
12     Student dummyStudent = new Student();
13
14     IDataManager dataManagerMock =
15         MockRepository.GenerateStrictMock<IDataManager>();
16     dataManagerMock.Expect(
17         x => x.GetByKey(studentId)).Return(dummyStudent);
18     dataManagerMock.Expect(
19         x => x.Save(dummyStudent));
20
```

```
21      ScoreManager smanager = new ScoreManager(dataManagerMock);
22      smanager.AddScore(studentId, score);
23
24      dataManagerMock.VerifyAllExpectations();
25  }
```

En este caso, el colaborador es un mock con dos expectativas. El orden de las expectativas también es decisivo en la fase de verificación: si la que aparece segunda se diese antes que la primera, el framework marcaría el test como fallido. A pesar del esfuerzo que hemos hecho por escribir este test, tiene un problema importante y es su fragilidad, ya que conoce cómo funciona el SUT más de lo que debería. No sólo sabe que hace dos llamadas a su colaborador sino que, además, conoce el orden y ni si quiera estamos comprobando que los puntos han subido al marcador del alumno. ¿Será que el SUT tiene más de una responsabilidad? Un código que se hace muy difícil de probar expresa que necesita ser reescrito. Reflexionemos sobre las responsabilidades. El `ScoreManager` está encargado de coordinar la acción de actualizar el marcador y guardar los datos. Podemos identificar entonces la responsabilidad de actualizar el marcador y separarla. La delegamos en una clase que se encarga exclusivamente de ello. Vamos a diseñarla utilizando un test unitario:

```
1   [Test]
2   public void ScoreUpdaterWorks()
3   {
4       ScoreUpdater updater = new ScoreUpdater();
5       Student student = updater.UpdateScore(
6                           new Student(), 5f);
7       Assert.AreEqual(student.Score, 5f);
8   }
```

El código del SUT:

```
1   public class ScoreUpdater : IScoreUpdater
2   {
3       public Student UpdateScore(Student student,
4               float score)
5       {
6           student.Score = student.Score + score;
7           return student;
8       }
9   }
```

Ahora, que ya podemos probarlo todo, reescribamos el test que nos preocupaba:

```
1   [Test]
2   public void AddStudentScore()
3   {
4       string studentId = "23145";
5       float score = 8.5f;
6       Student dummyStudent = new Student();
7
```

```
8       IDataManager dataManagerMock =
9           MockRepository.GenerateStrictMock<IDataManager>();
10      dataManagerMock.Expect(
11          x => x.GetByKey(studentId)).Return(dummyStudent);
12      dataManagerMock.Expect(
13          x => x.Save(dummyStudent));
14      IScoreUpdater scoreUpdaterMock =
15          MockRepository.GenerateStrictMock<IScoreUpdater>();
16      scoreUpdaterMock.Expect(
17          y => y.UpdateScore(dummyStudent, score)).Return(dummyStudent);
18
19      ScoreManager smanager =
20          new ScoreManager(dataManagerMock, scoreUpdaterMock);
21      smanager.AddScore(studentId, score);
22
23      dataManagerMock.VerifyAllExpectations();
24      scoreUpdaterMock.VerifyAllExpectations();
25  }
```

Hubo que modificar el constructor de `ScoreManager` para que aceptase otro colaborador. Ahora estamos seguros que se está probando todo. ¡Pero el test es idéntico a la implementación del SUT!

```
1   public void AddScore(string studentId, float score)
2   {
3       IRelationalObject student =
4           _dataManager.GetByKey(studentId);
5       Student studentUpdated =
6           _updater.UpdateScore((Student)student, score);
7       _dataManager.Save(studentUpdated);
8   }
```

Desde luego este test parece más un SUT en sí mismo que un ejemplo de cómo debe funcionar el SUT. Imita demasiado lo que hace el SUT. ¿Cómo lo enmendamos? Lo primero que hay que preguntarse es si realmente estamos obligados a obtener el objeto `Student` a partir de su clave primaria o si tiene más sentido recibirlo de alguna otra función. Si toda la aplicación tiene un buen diseño orientado a objetos, quizás el método `AddScore` puede recibir ya un objeto `Student` en lugar de su clave primaria. En ese caso, nos quitaríamos una expectativa del test. Vamos a suponer que podemos modificar el SUT para aceptar este cambio:

```
1   [Test]
2   public void AddStudentScore()
3   {
4       float score = 8.5f;
5       Student dummyStudent = new Student();
6       IScoreUpdater scoreUpdaterMock =
7            MockRepository.GenerateStrictMock<IScoreUpdater>();
8       scoreUpdaterMock.Expect(
9            y => y.UpdateScore(dummyStudent,
10                    score)).Return(dummyStudent);
11      IDataManager dataManagerMock =
12           MockRepository.GenerateStrictMock<IDataManager>();
```

```
13      dataManagerMock.Expect(
14           x => x.Save(dummyStudent));
15
16      ScoreManager smanager =
17      new ScoreManager(dataManagerMock, scoreUpdaterMock);
18      smanager.AddScore(dummyStudent, score);
19
20      dataManagerMock.VerifyAllExpectations();
21      scoreUpdaterMock.VerifyAllExpectations();
22  }
```

```
1   public void AddScore(Student student, float score)
2   {
3       Student studentUpdated =
4             _updater.UpdateScore(student, score);
5       _dataManager.Save(studentUpdated);
6   }
```

Ahora el test nos ha quedado con dos expectativas pero pertenecientes a distintos colaboradores. El orden de las llamadas no importa cuando son sobre colaboradores distintos, es decir, que aunque en el test hayamos definido la expectativa `UpdateScore` antes que `Save`, no se rompería si el SUT los invocase en orden inverso. Entonces el test no queda tan frágil.

En caso de que no podamos cambiar la API para recibir el objeto `Student`, sólo nos queda partir el test en varios para eliminar las restricciones impuestas por el framework con respecto al orden en las llamadas a los mocks. La idea es probar un solo aspecto de la colaboración en cada test mediante mocks, e ignorar lo demás con stubs. Veamos un ejemplo simplificado. Pensemos en la orquestación de unos servicios web. El SUT se encarga de orquestar (coordinar) la forma en que se realizan llamadas a distintos servicios. De forma abreviada el SUT sería:

```
1   public class Orchestrator
2   {
3       private IServices _services;
4
5       public Orchestrator(IServices services)
6       {
7           _services = services;
8       }
9
10      public void Orchestrate()
11      {
12          _services.MethodA();
13          _services.MethodB();
14      }
15  }
```

La orquestación consiste en invocar al servicio A y, a continuación, al servicio B. Si escribimos un test con tales expectativas, queda un test idéntico al SUT como nos estaba pasando. Vamos a partirlo en dos para probar cada colaboración por separado:

```
[TestFixture]
public class ServicesTests
{
    [Test]
    public void OrchestratorCallsA()
    {
        IServices servicesMock =
            MockRepository.GenerateStrictMock<IServices>();
        servicesMock.Expect(
            a => a.MethodA());
        servicesMock.Stub(
            b => b.MethodB());

        Orchestrator orchestrator =
            new Orchestrator(servicesMock);
        orchestrator.Orchestrate();

        servicesMock.VerifyAllExpectations();
    }

    [Test]
    public void OrchestratorCallsB()
    {
        IServices servicesMock =
            MockRepository.GenerateStrictMock<IServices>();
        servicesMock.Expect(
            b => b.MethodB());
        servicesMock.Stub(
            a => a.MethodA());

        Orchestrator orchestrator =
            new Orchestrator(servicesMock);
        orchestrator.Orchestrate();

        servicesMock.VerifyAllExpectations();
    }
}
```

El primer test tan sólo se encarga de probar que el servicio A se llama, mientras que se le dice al framework que da igual lo que se haga con el servicio B. El segundo test trabaja a la inversa. De esta forma, estamos diseñando qué elementos toman parte en la orquestación y no tanto el orden mismo. Así el test no es tan frágil y la posibilidad de romper los tests por cambios en el SUT disminuye. No obstante, si resulta que el orden de las llamadas es algo tan crítico que se decide escribir todas las expectativas en un solo test, se puede hacer siempre que tengamos conciencia de lo que ello significa: lanzar una alerta cada vez que alguien modifique algo del SUT. Es programar una alerta, no un test. Si de verdad es lo que necesitamos, entonces está bien. Rhino.Mocks permite crear híbridos[7] entre mocks y stubs mediante

[7]Leyendo a Fowler y Meszaros entiendo que en realidad no son híbridos sino stubs, ya que se les atribuye la posibilidad de recordar expectativas. Sin embargo la mayoría de los frameworks consideran que un stub símplemente devuelve valores, que no recuerda

la llamada `GenerateMock` en lugar de `GenerateStrictMock`. Así, los tests anteriores se podrían reescribir con un resultado similar y menos líneas de código:

```
1  [TestFixture]
2  public class ServicesHybridMocksTests
3  {
4      [Test]
5      public void OrchestratorCallsA()
6      {
7          IServices servicesMock =
8              MockRepository.GenerateMock<IServices>();
9          servicesMock.Expect(
10             a => a.MethodA());
11
12         Orchestrator orchestrator =
13             new Orchestrator(servicesMock);
14         orchestrator.Orchestrate();
15
16         servicesMock.VerifyAllExpectations();
17     }
18
19     [Test]
20     public void OrchestratorCallsB()
21     {
22         IServices servicesMock =
23             MockRepository.GenerateMock<IServices>();
24         servicesMock.Expect(
25             b => b.MethodB());
26
27         Orchestrator orchestrator =
28            new Orchestrator(servicesMock);
29         orchestrator.Orchestrate();
30
31         servicesMock.VerifyAllExpectations();
32     }
33 }
```

La diferencia es que el framework sólo falla si la llamada no se produce pero admite que se hagan otras llamadas sobre el SUT, incluso que se repita la llamada que definimos en la expectativa. Nos ahorra declarar la llamada stub en el test pero, por contra, se calla la posible repetición de la expectativa, lo cual seguramente no nos conviene.

Visto el ejemplo de los servicios, queda como ejercicio propuesto escribir tests para el ejemplo de los estudiantes que no hemos terminado de cerrar.

Quizás haya observado que en todo el capítulo no hemos creado ningún mock basado en una implementación concreta de una base de datos ni de ninguna librería del sistema (.Net en este caso). No es recomendable crear mocks basados en dependencias externas sino sólo en nuestras propias interfaces. De ahí el uso de interfaces como `IDataManager`. Aunque es posible hacer

nada.

mocks de clases, siempre usamos interfaces, puesto que la inyección de dependencias[8] es la mejor forma en que podemos gestionar tales dependencias entre SUT y colaboradores. Una vez escritos los tests unitarios, añadiremos tests de integración que se encargan de probar que aquellas de nuestras clases que hablan con el sistema externo, lo hacen bien. Escribiríamos tests de integración para la clase `DataManager`. En el caso de Python, si la API de la clase externa nos resulta suficientemente genérica, podemos hacer mock de la misma directamente, dado que en este lenguaje no necesitamos definir interfaces al ser débilmente tipado. Por eso, en Python, sólo crearía una clase tipo `DataManager` si viese que las distintas entidades externas con las que hablará son muy heterogéneas. Es un claro ejemplo en el que Python ahorra unas cuantas líneas de código.

## 6.2. La metáfora Record/Replay

Algunos frameworks de mocks como EasyMock (y en versiones anteriores, también Rhino.Mocks) usan la metáfora conocida como Record/Replay. Necesitan que les indiquemos explícitamente cuándo hemos terminado de definir expectativas para comenzar el acto. Afortunadamente, es fácil hacerlo, es una línea de código pero a la gente le choca esto de record y replay. Si hemos comprendido todos los tests de este capítulo, la metáfora no será ningún problema. EasyMock es para Java. Veamos la versión Java del ejemplo del calendario:

6.1: EasyMock

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3  import java.util.Calendar;
4  import static org.easymock.EasyMock.*;
5
6  public class CalendarTests {
7
8       @Test
9       public void drawHolidays()
10      {
11          int year = 2009;
12          int month = 2;
13          String townCode = "b002";
14          ICalendarService serviceMock =
15              createMock(ICalendarService.class);
16
17          expect(serviceMock.getHolidays(
18                  year, month, townCode)
19                  ).andReturn(new int[] { 1, 5 });
20
21          replay(serviceMock);
```

[8] Ver Capítulo 7 en la página 111

```
22
23          Calendar calendar = new Calendar(serviceMock);
24          calendar.set_currentTown(townCode);
25          calendar.set_currentYear(year);
26          calendar.set_currentMonth(month);
27          calendar.drawMonth();
28
29          verify(serviceMock);
30      }
31  }
```

Prácticamente todo igual. Lo único es que hemos tenido que decir explícitamente `replay(serviceMock)` para cambiar de estado al mock. Si se nos olvida activar el `replay`, el resultado del test es bastante raro puesto que, lo que debería ser el acto se sigue considerando la preparación y es desconcertante. Suele pasar al principio con este tipo de frameworks. La documentación de EasyMock es concisa, siendo recomendable dedicar unos minutos a revisarla para descubrir toda su funcionalidad. Al igual que la de Rhino.Mocks, contiene ejemplos que reforzarán los conceptos expresados en estas líneas. La metáfora que acabamos de ver no aporta ninguna ventaja al desarrollo y, de hecho, en Rhino.Mocks ya se hace implícita. El motivo por el que algunos frameworks la mantienen es para hacer más sencilla la implementación de los mocks internamente, es decir, que es una limitación en lugar de una ventaja.

Mi framework de mocks favorito en Java, a día de hoy, es Mockito[9] ya que es más sencillo que EasyMock. Produce un código más claro y la metáfora de record/replay ya se ha superado. Para mockito no hay que utilizar distintos métodos a la hora de crear un stub y un mock. En principio todos los dobles se crean con el método `mock`, aunque luego sea un stub. La diferencia queda implítica en la forma en que utilicemos el doble. Así por ejemplo, para definir un mock con una expectativa, podemos hacer lo siguiente:

6.2: Mockito

```
1  @Test
2  public void persistorSaves() throws Exception {
3          EmailAddress emailMock =
4                  mock(EmailAddress.class);
5          Persistor<EmailAddress> persistor =
6                  new Persistor<EmailAddress>();
7
8          persistor.Save(emailMock);
9
10         verify(emailMock).save();
11 }
```

El test dice que nuestro SUT (`Persistor`) tiene que invocar forzosamente en algún momento al método `save` del colaborador, que es de tipo `EmailAddress`.

[9] http://mockito.org

Los dobles de prueba en mockito son, por defecto, una especie de mock híbrido o relajado[10] que no tiene en cuenta el orden de las expectativas u otras posibles llamadas, salvo que se especifique con código, lo cual se agradece la mayoría de las veces. Es similar al `GenerateMock` de Rhino.Mocks.

Por último, nos queda ver cómo se trabaja en Python pero lo haremos en la parte práctica del libro, porque tanto con Python Mocker[11] como con PyMox[12], se trabaja practicamente igual a como acabamos de estudiar. Mocker funciona francamente bien pero su desarrollo lleva tiempo parado, mientras que el de Pymox sigue activo y su sintáxis se parece aún más a EasyMock o Rhino.Mocks.

Algunas de las preguntas que quedan abiertas en el presente capítulo, se resolverán en los siguientes.

[10] Como dijimos antes, hay quien les llama stubs pero resulta confuso. La nomenclatura de facto, dice que los stubs se limitan a devolver valores cuando se les pregunta

[11] http://labix.org/mocker

[12] http://code.google.com/p/pymox/

Capítulo 7

# Diseño Orientado a Objetos

TDD tiene una estrecha relación con el buen diseño orientado a objetos y por tanto, como no, con los principios S.O.L.I.D que veremos a continuación. En el último paso del algoritmo TDD, el de refactorizar, entra en juego nuestra pericia diseñando clases y métodos. Durante los capítulos prácticos haremos uso de todos estos principios y los nombraremos donde corresponda.

## 7.1. Diseño Orientado a Objetos (OOD)

Todos los lenguajes y plataformas actuales se basan en el paradigma de la programación orientada a objetos (OOP por sus siglas en inglés). Aunque a diario trabajamos con objetos, no todo el mundo comprende realmente lo que es el polimorfismo o para qué sirve una clase abstracta, por poner un ejemplo. La potencia de la orientación a objetos lleva implícita mucha complejidad y una larga curva de aprendizaje. Lo que en unos casos es una buena manera de resolver un problema, en otros es la forma de hacer el código más frágil. Es decir, no siempre conviene crear una jerarquía de clases, dependiendo del caso puede ser más conveniente crear una asociación entre objetos que colaboran. Desafortunadamente no hay reglas universales que sirvan para toda la gama de problemas que nos podamos encontrar pero hay ciertos principios y patrones que nos pueden dar pistas sobre cual es el diseño más conveniente en un momento dado.

Con fines docentes se suele explicar la OOP mediante ejemplos relacionados con el mundo que conocemos: véase el típico ejemplo de la clase Animal, de la que hereda la clase Mamífero, de la que a su vez hereda la clase Cuadrúpedo, de la que a su vez heredan las clases Perro y Gato... El símil no está mal en sí mismo, lo que sucede es que las clases en el software

no siempre casan directamente con los objetos del mundo real, porque el software difiere mucho de éste. Las clasificaciones naturales de los objetos, no tienen por qué ser clasificaciones adecuadas en el software. La conceptualización y el modelado son una espada de doble filo, pues como vamos a mostrar, la realidad es demasiado compleja de modelar mediante OOP y el resultado puede ser un código muy acoplado, muy difícil de reutilizar.

Veamos el ejemplo de la clase Rectangulo. El Rectangulo tiene dos atributos, Ancho y Alto y un método que es capaz de calcular el área. Ahora necesitamos una clase Cuadrado. En geometría el cuadrado es un rectángulo, por tanto, si copiamos esta clasificación diríamos que Cuadrado hereda de Rectangulo. Vale, entonces definimos Cuadrado extendiendo de Rectangulo. Ahora damos la clase Cuadrado a alguien que tiene que trabajar con ella y se encuentra con los atributos heredados, Ancho y Alto. Lo más probable es que se pregunte... ¿Qué significan el ancho y el alto en un cuadrado? Un cuadrado tiene todos sus lados iguales, no tiene ancho y alto. Este diseño no tienen sentido. Para este caso particular, si lo que queremos es ahorrarnos reescribir el método que calcula el área, podemos crear ese método en una tercera clase que colabora con Rectangulo y Cuadrado para calcular el área. Así Rectángulo sabe que cuando necesite calcular el área invocará al método de esta clase colaboradora pasando Ancho y Alto como parámetros y Cuadrado pasará dos veces la longitud de uno de sus lados.

Una de las mejores formas que hay, de ver si la API que estamos diseñando es intuitiva o no, es usarla. TDD propone usarla antes de implementarla, lo que le da in giro completo a la forma en que creamos nuestras clases. Puesto que todo lo hacemos con objetos, el beneficio de diseñar adecuadamente cambia radicalmente la calidad del software.

## 7.2. Principios S.O.L.I.D

Son cinco principios fundamentales, uno por cada letra, que hablan del diseño orientado a objetos en términos de la gestión de dependencias. Las dependencias entre unas clases y otras son las que hacen al código más frágil o más robusto y reutilizable. El problema con el modelado tradicional es que no se ocupa en profundidad de la gestión de dependencias entre clases sino de la conceptualización. Quién decidió resaltar estos principios y darles nombre a algunos de ellos fue Robert C. Martin, allá por el año 1995.

### 7.2.1. Single Responsibility Principle (SRP)

El principio que da origen a la S de S.O.L.I.D es el de una única responsabilidad y dice que cada clase debe ocuparse de un solo menester. Visto de otro modo, R. Martin dice que cada clase debería tener un único motivo para ser modificada. Si estamos delante de una clase que se podría ver obligada a cambiar ante una modificación en la base de datos y a la vez, ante un cambio en el proceso de negocio, podemos afirmar que dicha clase tiene más de una responsabilidad o más de un motivo para cambiar, por poner un ejemplo. Se aplica tanto a la clase como a cada uno de sus métodos, con lo que cada método también debería tener un solo motivo para cambiar. El efecto que produce este principio son clases con nombres muy descriptivos y por tanto largos, que tienen menos de cinco métodos, cada uno también con nombres que sirven perfectamente de documentación, es decir, de varias palabras: `CalcularAreaRectangulo` y que no contienen más de 15 líneas de código. En la práctica la mayoría de mis clases tienen uno o dos métodos nada más. Este principio es quizás el más importante de todos, el más sencillo y a la vez el más complicado de llevar a cabo. Existen ejemplos de código y una explicación más detallada del mismo en la propia web del autor[1]. Martin también habla en profundidad sobre SRP en su libro[10]. Hay una antigua técnica llamada Responsibility Driven Design (RDD), que viene a decir lo mismo que este principio, aunque es anterior a la aparición de SRP como tal. TDD es una excelente manera de hacer RDD o de seguir el SRP, como se quiera ver. Allá por el año 1989, Kent Beck y Ward Cunningham usaban tarjetas CRC[2] (Class, Responsibility, Collaboration) como ayuda para detectar responsabilidades y colaboraciones entre clases. Cada tarjeta es para una entidad, no necesariamente una clase. Desde que disponemos de herramientas que nos permiten el desarrollo dirigido por tests, las tarjetas CRC han pasado a un segundo plano pero puede ser buena idea usarlas parcialmente para casos donde no terminamos de ver claras las responsabilidades.

### 7.2.2. Open-Closed Principle (OCP)

Una entidad software (una clase, módulo o función) debe estar abierta a extensiones pero cerrada a modificaciones. Puesto que el software requiere cambios y que unas entidades dependen de otras, las modificaciones en el código de una de ellas puede generar indeseables efectos colaterales en cascada. Para evitarlo, el principio dice que el comportamiento de una entidad debe poder ser alterado sin tener que modificar su propio código fuente.

[1] http://www.objectmentor.com/resources/articles/srp.pdf
[2] http://c2.com/doc/oopsla89/paper.html

¿Cómo se hace esto?, Hay varias técnicas dependiendo del diseño, una podría ser mediante herencia y redefinición de los métodos de la clase padre, donde dicha clase padre podría incluso ser abstracta. La otra podría ser inyectando dependencias que cumplen el mismo contrato (que tienen la misma interfaz) pero que implementan diferente funcionamiento. En próximos párrafos estudiaremos la inyección de dependencias. Como la totalidad del código no se puede ni se debe cerrar a cambios, el diseñador debe decidir contra cuáles protegerse mediante este principio. Su aplicación requiere bastante experiencia, no sólo por la dificultad de crear entidades de comportamiento extensible sino por el peligro que conlleva cerrar determinadas entidades o parte de ellas. Cerrar en exceso obliga a escribir demasiadas líneas de código a la hora de reutilizar la entidad en cuestión. El nombre de Open-Closed se lo debemos a Bertrand Meyer y data del año 1988. En español podemos denominarlo el principio Abierto-Cerrado. Para ejemplos de código léase el artículo original de R. Martin[3]

### 7.2.3. Liskov Substitution Principle (LSP)

Introducido por Barbara Liskov en 1987, lo que viene diciendo es que si una función recibe un objeto como parámetro, de tipo X y en su lugar le pasamos otro de tipo Y, que hereda de X, dicha función debe proceder correctamente. Por el propio polimorfismo, los compiladores e intérpretes admiten este paso de parámetros, la cuestión es si la función de verdad está diseñada para hacer lo que debe, aunque quien recibe como parámetro no es exactamente X, sino Y. El principio de sustitución de Liskov está estrechamente relacionado con el anterior en cuanto a la extensibilidad de las clases cuando ésta se realiza mediante herencia o subtipos. Si una función no cumple el LSP entonces rompe el OCP puesto que para ser capaz de funcionar con subtipos (clases hijas) necesita saber demasiado de la clase padre y por tanto, modificarla. El diseño por contrato (Design by Contract) es otra forma de llamar al LSP. Léase el artículo de R. Martin sobre este principio[4].

### 7.2.4. Interface Segregation Principle (ISP)

Cuando empleamos el SRP también empleamos el ISP como efecto colateral. El ISP defiende que no obliguemos a los clientes a depender de clases o interfaces que no necesitan usar. Tal imposición ocurre cuando una clase o interfaz tiene más métodos de los que un cliente (otra clase o entidad)

[3] http://www.objectmentor.com/resources/articles/ocp.pdf
[4] http://www.objectmentor.com/resources/articles/lsp.pdf

necesita para sí mismo. Seguramente sirve a varios objetos cliente con responsabilidades diferentes, con lo que debería estar dividida en varias entidades. En los lenguajes como Java y C# hablamos de interfaces pero en lenguajes interpretados como Python, que no requieren interfaces, hablamos de clases. No sólo es por motivos de robustez del software, sino también por motivos de despliegue. Cuando un cliente depende de una interfaz con funcionalidad que no utiliza, se convierte en dependiente de otro cliente y la posibilidad de catástrofe frente a cambios en la interfaz o clase base se multiplica. Léase el artículo de R. Martin[5]

### 7.2.5. Dependency Inversión Principle (DIP)

La inversión de dependencias da origen a la conocida inyección de dependencias, una de las mejores técnicas para lidiar con las colaboraciones entre clases, produciendo un código reutilizable, sobrio y preparado para cambiar sin producir efectos "bola de nieve". DIP explica que un módulo concreto A, no debe depender directamente de otro módulo concreto B, sino de una abstracción de B. Tal abstracción es una interfaz o una clase (que podría ser abstracta) que sirve de base para un conjunto de clases hijas. En el caso de un lenguaje interpretado no necesitamos definir interfaces, ni siquiera jerarquías pero el concepto se aplica igualmente. Veámoslo con un ejemplo sencillo: La clase `Logica` necesita de un colaborador para guardar el dato `Dato` en algún lugar persistente. Disponemos de una clase `MyBD` que es capaz de almacenar `Dato` en una base de datos MySQL y de una clase `FS` que es capaz de almacenar `Dato` en un fichero binario sobre un sistema de ficheros NTFS. Si en el código de `Logica` escribimos literalmente el nombre de la clase `MyBD` como colaborador para persistir datos, ¿Cómo haremos cuando necesitamos cambiar la base de datos por ficheros binarios en disco?. No quedará otro remedio que modificar el código de `Logica`.

Si las clases `MyDB` y `FS` implementasen una misma interfaz `IPersistor` para salvar `Dato`, podriamos limitarnos a usar `IPersistor` (que es una abstracción) en el código de `Logica`. Cuando los requerimientos exigiesen un cambio de base de datos por ficheros en disco o viceversa, sólo tendríamos que preocuparnos de que el atributo `_myPersistor` de la clase `Logica`, que es de tipo `IPersistor` contuviese una instancia de `MyDB` o bien de `FS`. ¿Cómo resolvemos esta última parte?. Con la inyección de dependencias, que vamos a ver dentro del siguiente apartado, Inversión del Control. En los próximos capítulos haremos mucho uso de la inyección de dependencias con gran cantidad de listados de código. No se preocupe si el ejemplo no le que-

[5] http://www.objectmentor.com/resources/articles/isp.pdf

da demasiado claro. El artículo de R. Martin sobre DIP[6] es uno de los más amenos y divertidos sobre los principios S.O.L.I.D.

## 7.3. Inversión del Control (IoC)

Inversión del Control es sinónimo de Inyección de Dependencias (DI). El nombre fue popularizado por el célebre Martin Fowler pero el concepto es de finales de los años ochenta. Dado el principio de la inversión de dependencias, nos queda la duda de cómo hacer para que la clase que requiere colaboradores de tipo abstracto, funcione con instancias concretas. Dicho de otro modo, ¿Quién crea las instancias de los colaboradores? Retomemos el ejemplo de las clases de antes. Tradicionalmente la clase `Logica` tendría una sentencia de tipo "`_myPersistor = new MyDB()`" dentro de su constructor o de algún otro método interno para crear una instancia concreta, ya que no podemos crear instancias de interfaces ni de clases abstractas. En tiempo de compilación nos vale con tener el contrato pero en tiempo de ejecución tiene que haber alguien que se ponga en el pellejo del contrato. Si lo hacemos así volvemos al problema de tener que modificar la clase `Logica` para salvar en ficheros binarios. La solución es invertir la forma en que se generan las instancias. Habrá una entidad externa que toma la decisión de salvar en base de datos o en ficheros y en función de eso crea la instancia adecuada y se la pasa a `Logica` para que la asigne a su miembro `_myPersistor`. Hay dos formas, como parámetro en el constructor de `Logica` o bien mediante un *setter* o método que únicamente sirve para recibir el parámetro y asignarlo al atributo interno. La entidad externa puede ser otra clase o bien puede ser un contenedor de inyección de dependencias.

¿Qué son los *IoC Containers*? Son la herramienta externa que gestiona las dependencias y las inyecta donde hacen falta. Los contenedores necesitan de un fichero de configuración o bien de un fragmento de código donde se les indica qué entidades tienen dependencias, cuáles son y qué entidades son independientes. Afortunadamente hay una gran variedad de contenedores libres para todos los lenguajes modernos. Algunos de los más populares son Pinsor para Python, Spring Container para Java y .Net, Pico y Nano para Java, Needle y Copland para Ruby y Castle.Windsor para .Net. Habiendo preparado las clases de nuestro ejemplo para la inversión del control, podemos especificar al contenedor que inyecte `MyDB` o `FS` mediante un fichero de configuración que lee la aplicación al arrancar y conseguir diferente comportamiento sin tocar una sola línea de código. Demoledor. Si la aplicación es pequeña no necesitamos ningún contenedor de terceros sino que en nues-

[6] http://www.objectmentor.com/resources/articles/dip.pdf

tro propio código podemos inyectar las dependencias como queramos. Los contenedores son una herramienta pero no son imprescindibles. Su curva de aprendizaje puede ser complicada. En nuestro pequeño ejemplo hemos seguido la mayor parte de los principios S.O.L.I.D, aunque no hemos entrado a ver qué hacen las clases en detalle pero por lo menos queda una idea ilustrativa del asunto que nos ocupa. No se asuste, resulta más sencillo de lo que parece y sino, TDD no lo va a ir cascando todo, ya verá.

# Parte II

# Ejercicios Prácticos

Capítulo 8

# Inicio del proyecto - Test Unitarios

Vamos a escribir nuestra primera aplicación de ejemplo porque practicando es como realmente se aprende. En lugar de implementarla por completo y pulirla para luego escribir este capítulo, vamos a diseñarla juntos desde el principio para ver en realidad cómo se razona y se itera en el desarrollo dirigido por tests. La aplicación ocupará este capítulo y los siguientes para que tengamos la oportunidad de afrontar toda la problemática de cualquier aplicación real. Sin embargo, no es una aplicación empresarial como pudiera ser un software de facturación, sino que se basa en un dominio de negocio que todos conocemos. Se podría argumentar que el software que vamos a desarrollar no es muy común o muy real pero incluye todas las piezas que lleva una aplicación "más real". Imagínese que para entender el código fuente que nos ocupa tuviese uno que estudiar primero contabilidad o derecho. Tenga confianza en que nuestra aplicación de ejemplo es perfectamente válida y representa a pequeña escala el modo en que se desarrolla una aplicación mucho más grande. Nos adentramos ya en las vicisitudes de este pequeño gran desarrollo de software.

Sentémonos con el cliente para escucharle hablar sobre su problema y hacer un primer análisis. Lo que nos cuenta es lo siguiente:

*Quiero lanzar al mercado un software educativo para enseñar matemáticas a niños. Necesito que puedan jugar o practicar a través de una página web pero también a través del teléfono móvil y quizás más adelante también en la consola Xbox. El juego servirá para que los niños practiquen diferentes temas dentro de las matemáticas y el sistema debe recordar a cada niño, que tendrá un nombre de usuario y una clave de acceso. El sistema registrará todos los ejercicios que han sido completados y la puntuación obtenida para permitirles subir de nivel si progresan. Existirá un usuario tutor que se registra a la vez que el niño y que tiene la posibilidad de acceder al sistema y ver estadísticas de juego del niño. El tema más importante ahora mismo es la aritmética básica con números enteros. Es el primero que necesito tener listo para ofrecer a los profesores de enseñanza primaria un refuerzo para sus alumnos en el próximo comienzo de curso. El módulo de aritmética base incluye las cuatro operaciones básicas (sumar, restar, multiplicar y dividir) con números enteros. Los alumnos no solo tendrán que resolver los cálculos más elementales sino también resolver expresiones con paréntesis y/o con varias operaciones encadenadas. Así aprenderán la precedencia de los operadores y el trabajo con paréntesis: las propiedades distributiva, asociativa y conmutativa. Los ejercicios estarán creados por el usuario profesor que introducirá las expresiones matemáticas en el sistema para que su resultado sea calculado automáticamente y almacenado. El profesor decide en qué nivel va cada expresión matemática. En otros ejercicios se le pedirá al niño que se invente las expresiones matemáticas y les ponga un resultado. El programa dispondrá de una calculadora que sólo será accesible para los profesores y los jugadores de niveles avanzados. La calculadora evaluará y resolverá las mismas expresiones del sistema de juego. Cuando el jugador consigue un cierto número de puntos puede pasar de nivel, en cuyo caso un email es enviado al tutor para que sea informado de los logros del tutelado. El número mínimo de puntos para pasar de nivel debe ser configurable.*

Una vez escuchado el discurso del cliente y habiendo decidido que lo primero a implementar en los primeros sprints[1] será el motor de juego de la aritmética básica, nos disponemos a formular los criterios de aceptación para que el cliente los valide. Tal como dijimos en el capítulo sobre ATDD, los

[1] En Scrum un sprint es una iteración

tests de aceptación (o de cliente) son frases cortas y precisas escritas con el lenguaje del dominio de negocio. Son tan sencillas que al verlas el cliente sólo tiene que decir si son afirmaciones correctas o no. Son ejemplos. Los ejemplos evitan la ambigüedad que se puede inmiscuir en la descripción de una tarea. Sabemos que es muy peligroso dar por sentado cuestiones referentes a la lógica de negocio, que debemos ceñirnos exclusivamente a la funcionalidad que se requiere. El motor de juego para aritmética implica muchos ejemplos a poco que nos pongamos a pensar. ¿Qué pasa si el profesor no ha añadido suficientes expresiones a un nivel como para alcanzar la puntuación que permite pasar el nivel siguiente? ¿se pueden eliminar expresiones de un nivel? ¿se pueden añadir o modificar? ¿cómo afecta eso a las puntuaciones? ¿y si hay jugadores en medio del nivel? ¿se puede trasladar una expresión de un nivel a otro?. Cada pregunta se resolvería con varios ejemplos. Tenemos por delante un buen trabajo de análisis. Sin embargo, por fines docentes, empezaremos abordando cuestiones más simples como el propio funcionamiento de la calculadora. Los tests de aceptación iniciales son:

- "2 + 2", devuelve 4
- "5 + 4 * 2 / 2", devuelve 9
- "3 / 2", produce el mensaje ERROR
- "* * 4 - 2": produce el mensaje ERROR
- "* 4 5 - 2": produce el mensaje ERROR
- "* 4 5 - 2 -": produce el mensaje ERROR
- "*45-2-": produce el mensaje ERROR

Estos ejemplos de funcionamiento parecen una chorrada pero si no estuvieran ahí, podríamos haber pensado que se requerían paréntesis para definir la precedencia de los operadores o que la notación de los comandos era polaca inversa[2]. A través de los ejemplos queda claro que los diferentes elementos del comando se operan según la precedencia, donde la multiplicación y la división se operan antes que la suma y la resta. Y también sabemos que un resultado con decimales, no se permite porque nuestra aritmética básica trabaja únicamente con enteros. Otra cosa que queda clara es que los operadores y operandos de la expresión se separan por un espacio o sea, que las expresiones son cadenas de texto.

Lógicamente hay más ejemplos pero vamos a empezar la implementación ya para no dilatar más el análisis. Así de paso vemos el caso en que descubrimos nuevos requisitos de negocio cuando ya estamos implementando código, algo que sucede la gran mayoría de las veces.

[2]http://es.wikipedia.org/wiki/Notaci%C3%B3n_polaca_inversa

Manos a la obra. Abrimos un nuevo proyecto con nuestro IDE favorito por un lado y un editor de textos sencillo por otro. Le recomiendo encarecidamente que lo haga tal cual, literalmente, que vaya escribiendo todos los fragmentos de código fuente que se exponen mientras lee; así irá practicando desde ya. Antes de copiar directamente del libro intente pensar por usted mismo cuál es el siguiente paso a dar en cada momento. Este será su primer proyecto TDD si lo tiene a bien. Sucede que las decisiones de diseño las estoy tomando yo y pueden diferir de las que usted tome, lo cual no significa que las suyas sean inapropiadas, sino simplemente diferentes. No se desanime por ello. La limitación de un libro impreso es su ausencia de interactividad y no podemos hacer nada para solventar este inconveniente. Si cree que una determinada refactorización o decisión de diseño es inapropiada no dude en compartirlo mediante la web del libro.

A la pantalla del editor de texto le vamos a llamar *libreta* y es donde apuntaremos los ejemplos que nos quedan por hacer y todo lo que se nos ocurre mientras estamos escribiendo tests o lógica de negocio: normalmente cuando estamos concentrados en el test que nos ocupa, es común que vengan a la mente casos de uso que no estaban contemplados o bien dudas sobre la lógica de negocio. Esto no debe distraernos de lo que estamos haciendo ni tampoco influenciarnos, por eso lo anotamos en la libreta con una frase breve que nos permita volver a retomarlo cuando acabemos. Tenemos que centrarnos en una sola cosa cada vez para llevarla a cabo como es debido. Abrimos también la herramienta con que se ejecuta la batería de tests (*NUnit* gráfico o una consola donde invocar a `nunit-console` o el script Python que inicia *unittest*).

Típicamente creamos dos módulos, uno para la aplicación y otro para su batería de tests. Si estamos con C# serán dos DLLs y si estamos con Python serán dos paquetes distintos. Para no mezclar los dos lenguajes continuamente vamos a escribir primero en C# y luego veremos su contrapartida en Python. Los seguidores de uno de los lenguajes no deben saltarse los capítulos en los que se usa el otro porque en ambos capítulos existen explicaciones válidas para los dos lenguajes y referencias de uno a otro.

Ahora toca escoger uno de los tests de aceptación y pensar en una lista de elementos que nos hacen falta para llevarlo a cabo. Hacer un pequeño análisis del ejemplo y tratar de definir al menos tres o cuatro ejemplos más sencillos que podamos convertir en tests de desarrollo (unitarios y de integración o sólo unitarios) para empezar a hacer TDD. Estamos combinando ATDD con TDD. Los dos elementos que se me ocurren ahora mismo son una clase que sea capaz de operar enteros y un analizador que sea capaz de identificar los distintos elementos de una expresión contenida en una cadena de texto. Si no tuviera que escribir este capítulo con fines docentes seguramente empezaría

por trabajar en el analizador pero los ejemplos de la clase que hace cálculos van a ser más fáciles de asimilar inicialmente.

Generalmente los tests de aceptación se guardan por separado, al margen de la libreta que contiene las cuestiones relativas a desarrollo. Si empleamos un framework tipo Fit o Concordion los tests de aceptación tendrían su propio lugar pero por simplificar y de nuevo con fines docentes, mantendremos ambas cosas en la libreta. Vamos a agregarle los tests unitarios que nos gustaría hacer para resolver el primero de los tests de aceptación.

```
# Aceptación - "2 + 2", devuelve 4
▪ Sumar 2 al número 2, devuelve 4
▪ Restar 3 al número 5, devuelve 2
▪ La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación -"* 4 5 - 2": produce ERROR
# Aceptación -"* 4 5 - 2 : produce ERROR
# Aceptación -"*45-2-": produce ERROR
```

Vamos a por el primer test de desarrollo:

```
1
2  using System;
3  using System.Collections.Generic;
4  using System.Text;
5  using NUnit.Framework;
6  using SuperCalculator;
7
8  namespace UnitTests
9  {
10     [TestFixture]
11     public class CalculatorTests
12     {
13         [Test]
14         public void Add()
15         {
16             Calculator calculator = new Calculator();
17             int result = calculator.Add(2, 2);
18             Assert.AreEqual(4, result);
19         }
20     }
21 }
```

El código no compila porque todavía no hemos creado la clase `Calculator`. Sin embargo, ya hemos diseñado algo, casi sin darnos cuenta: hemos pensado en el nombre de la clase, en cómo es su constructor y en cómo es su primer método (cómo se llama, los parámetros que recibe y el resultado que

devuelve). Estamos diseñando la API tal cual la necesitamos, sin limitaciones ni funcionalidad extra. A continuación escribimos el mínimo código posible para que el test pase. Intente imaginar cuál es literalmente el mínimo código posible para que el test pase. Seguro que es más largo que el que de verdad es mínimo:

```
namespace SuperCalculator
{
    public class Calculator
    {
        public int Add(int arg1, int arg2)
        {
            return 4;
        }
    }
}
```

¡Qué código más simplón!, ¡No sirve para nada!. Pues sí, sí que sirve. Al seguir el algoritmo TDD y escribir literalmente el mínimo código posible para que el test pase, descubrimos una cualidad de la función: para valores de entrada diferentes presenta resultados diferentes. Vale vale, todos sabíamos eso, en este caso es muy evidente pero ¿y si no tuviéramos un conocimiento tan preciso del dominio?. Cuando no tenemos experiencia con TDD es muy importante que sigamos el algoritmo al pie de la letra porque de lo contrario no llegaremos a exprimir al máximo la técnica como herramienta de diseño. Ya que el test pasa, es decir, luz verde, necesitamos otro test que nos obligue a terminar de implementar la funcionalidad deseada, puesto que hasta ahora nuestra función de suma sólo funciona en el caso 2 + 2. Es como si estuviésemos moldeando una figura con un cincel. Cada test es un golpe que moldea el SUT. A este proceso de definir el SUT a golpe de test se le llama triangulación.

```
[Test]
public void AddWithDifferentArguments()
{
    Calculator calculator = new Calculator();
    int result = calculator.Add(2, 5);
    Assert.AreEqual(7, result);
}
```

Ejecutamos, observamos que estamos en rojo y acto seguido modificamos el SUT:

```
public int Add(int arg1, int arg2)
{
   return arg1 + arg2;
}
```

¿Por qué no hemos devuelto 7 directamente que es el código mínimo? Porque entonces el test anterior deja de funcionar y se trata de escribir el

código mínimo para que todos los tests tengan resultado positivo. Una vez hemos conseguido luz verde, hay que pensar si existe algún bloque de código susceptible de refactorización. La manera más directa de identificarlo es buscando código duplicado. En el SUT no hay nada duplicado pero en los tests sí: La llamada al constructor de `Calculator`, (véase línea 4 del último test). Advertimos que la instancia de la calculadora es un fixture y por lo tanto puede ser extraída como variable de instancia de la clase `CalculatorTests`. Eliminamos la duplicidad:

```
1  public class CalculatorTests
2  {
3    private Calculator _calculator;
4
5    [SetUp]
6    public void SetUp()
7    {
8        _calculator = new Calculator();
9    }
10
11   [Test]
12   public void Add()
13   {
14       int result = _calculator.Add(2, 2);
15       Assert.AreEqual(result, 4);
16   }
17
18   [Test]
19   public void AddWithDifferentArguments()
20   {
21       int result = _calculator.Add(2, 5);
22       Assert.AreEqual(result, 7);
23   }
```

Usar el `SetUp` no siempre es la opción correcta. Si cada uno de los tests necesitase de instancias de la calculadora distintas para funcionar (por ejemplo haciendo llamadas a diferentes versiones de un constructor sobrecargado), sería conveniente crearlas en cada uno de ellos en lugar de en la inicialización. Algunos como James Newkirk son tajantes en lo que respecta al uso del `SetUp` y dice que si por él fuera eliminaría esta funcionalidad de NUnit[a]. El color de este libro no es blanco ni negro sino una escala de grises: haga usted lo que su experiencia le diga que es más conveniente.

[a] http://jamesnewkirk.typepad.com/posts/2007/09/why-you-should-.html

Vamos a por la resta. Uy!, ya nos íbamos directos a escribir el código de la función que resta! Tenemos que estar bien atentos para dirigirnos primero

al test unitario. Al comienzo es muy común encontrarse inmerso en la implementación de código sin haber escrito un test que falla antes y hay que tener ésto en cuenta para no desanimarse. La frustración no nos ayudará. Si se da cuenta de que ha olvidado escribir el test o de que está escribiendo más código del necesario para la correcta ejecución del test, deténgase y vuelva a empezar.

```
[Test]
public void Substract()
{
    int result = calculator.Substract(5, 3);
    Assert.AreEqual(2, result);
}
```

Aquí acabamos de hacer otra decisión de diseño sin advertirlo. Estamos definiendo una API donde el orden de los parámetros de los métodos es importante. En el test que acabamos de escribir asumimos que a 5 se le resta 3 y no al revés. Esto es probablemente tan obvio que no nos lo hemos planteado tal que así pero si por un momento le hemos dado la vuelta a los parámetros mentalmente, ha tenido que llegarnos la pregunta de si se aceptan números negativos como resultado de la operación. Entonces la apuntamos en la libreta para no interrumpir lo que estamos haciendo y que no se nos olvide:

- *¿Puede ser negativo el resultado de una resta en nuestra calculadora?*
- *Confirmar que efectivamente el orden de los parámetros produce resultados diferentes*

# Aceptación - *"2 + 2", devuelve 4*

- *Sumar 2 al número 2, devuelve 4*
- *Restar 3 al número 5, devuelve 2*
- *La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'*

# Aceptación - *"5 + 4 * 2 / 2", devuelve 9*

# Aceptación - *"3 / 2", produce ERROR*

# Aceptación - *"* * 4 - 2": produce ERROR*

# Aceptación - *"* 4 5 - 2": produce ERROR*

# Aceptación - *"* 4 5 - 2 : produce ERROR*

# Aceptación - *"*45-2-": produce ERROR*

Como estamos en rojo vamos a ponernos en verde lo antes posible:

```
public int Substract(int ag1, int arg2)
{
   return 2;
}
```

De acuerdo, funciona. Ahora necesitamos otro test que nos permita pro-

bar los otros casos de uso de la función y miramos a la libreta. Como existe una duda, nos reunimos con el equipo y lo comentamos, preguntamos al cliente y .... decide que es aceptable devolver números negativos porque al fin y al cabo es como si hubiese un signo "menos" delante del número. Para aclararlo con un ejemplo se añade un test de aceptación a la libreta:

```
# Aceptación - "2 + 2", devuelve 4
▪ Sumar 2 al número 2, devuelve 4
▪ Restar 3 al número 5, devuelve 2
▪ La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
```

Y triangulamos. Escribimos el mínimo código necesario para pedirle a la resta que sea capaz de manejar resultados negativos:

```
1 [Test]
2 public void SubstractReturningNegative()
3 {
4     int result = calculator.Substract(3, 5);
5     Assert.AreEqual(-2, result);
6 }
```

Busquemos el color verde:

```
1 public int Substract(int arg1, int arg2)
2 {
3     return arg1 - arg2;
4 }
```

Y justo al escribir esto se nos viene otra serie de preguntas.

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
▪ ¿Cual es el número más pequeño que se permite como parámetro?
▪ ¿Y el más grande?
▪ ¿Qué pasa cuando el resultado es menor que el número más pequeño permitido?
▪ ¿Qué pasa cuando el resultado es mayor que el número más grande permitido?
```

Como puede apreciar hemos eliminado las cuestiones resueltas de la libreta.

Las nuevas cuestiones atañen a todas las operaciones de la calculadora. Ciertamente no sabemos si la calculadora correrá en un ordenador con altas prestaciones o en un teléfono móvil. Quizás no tenga sentido permitir más dígitos de los que un determinado dispositivo puede mostrar en pantalla aunque el framework subyacente lo permita. Transformarmos las preguntas en nuevos tests de aceptación:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
# Aceptación - Configurar el número más grande
  que puede usarse como argumento y como resultado
# Aceptación - Configurar el número más pequeño
  que puede usarse como argumento y como resultado
# Aceptación - Si el límite superior es 100 y
  alguno de los parámetros o el resultado es mayor que 100, ERROR
# Aceptación - Si el límite inferior es -100 y alguno
  de los parámetros o el resultado es menor que -100, ERROR
```

Simplifiquemos las frases para tener unos tests de aceptación más claros:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
# Aceptación - Límite Superior =100
# Aceptación - Límite Superior =500
# Aceptación - Límite Inferior = -1000
# Aceptación - Límite Inferior = -10
# Aceptación - Limite Superior=100 y parámetro mayor que 100, produce ERROR
# Aceptación - Limite Superior=100 y resultado mayor que 100, produce ERROR
# Aceptación - Limite Inferior=10 y parámetro menor que 10, produce ERROR
# Aceptación - Limite Inferior=10 y resultado menor que 10, produce ERROR
```

Ahora tenemos dos caminos. Podemos seguir adelante afrontando la segunda línea de la libreta, el analizador o podemos resolver la cuestión de los

límites. En TDD resolvemos el problema como si de un árbol se tratase. La raíz es el test de aceptación y los nodos hijos son tests de desarrollo, que unas veces serán unitarios y otras veces quizás de integración. Un árbol puede recorrerse de dos maneras; en profundidad y en amplitud. La decisión la tomamos en función de nuestra experiencia, buscando lo que creemos que nos va a dar más beneficio, bien en tiempo o bien en prestaciones. No hay ninguna regla que diga que primero se hace a lo ancho y luego a lo largo.

En este caso decidimos poder configurar la calculadora con el número más grande permitido y el más pequeño. Vamos a explorar a lo ancho. Es una funcionalidad que invocaríamos de esta manera:

```
1 [Test]
2 public void SubstractSettingLimitValues()
3 {
4      Calculator calculator = new Calculator(-100, 100);
5      int result = calculator.Substract(5, 10);
6      Assert.AreEqual(-5, result);
7 }
```

Estamos en rojo y nisiquiera es posible compilar porque el constructor de la clase `Calculator` no estaba preparado para recibir parámetros. Hay que tomar una decisión inmediata que no necesitamos apuntar en la libreta, ya que sin decidir no podemos ni compilar.

> ¿El constructor recibe parámetros? ¿Creamos dos versiones del constructor, con parámetros y sin ellos?

Por un lado si cambiamos el constructor para que acepte argumentos, necesitamos modificar el `SetUp` porque usa la versión anterior, lo cual nos recuerda que los tests requieren mantenimiento. Sin embargo, el costo de este cambio es mímino. Por otro lado podemos sobrecargar el constructor para tener ambas variantes pero... ¿qué pasa entonces cuando no indicamos los valores límite y se usan argumentos que superan el límite impuesto por el framework subyacente (en este caso .Net)?. No conviene decidir sin saber qué ocurrirá, mejor hacemos la prueba de sumar números muy grandes cuyo resultado excede dicho límite y observamos qué hace el *runtime* de .Net (`Int32.MaxValue + 1` por ejemplo). El resultado es el número más pequeño posible, un número negativo. Es como si un contador diese la vuelta. Es un comportamiento muy raro para una calculadora. Nos interesa más que sea obligatorio definir los límites. Bien pues ya podemos modificar el constructor para que admita los límites y los tests existentes que tras el cambio no compilen, con vistas a conseguir luz verde.

```
1 public class Calculator
2 {
3    public Calculator(int minValue, int maxValue) { }
4    ...
```

```
}
```

El último test que habíamos escrito (*SubstractSettingLimitValues*) no tiene nada diferente a los demás porque ya todos definen los límites; vamos a modificarlo escogiendo uno de los casos de uso de la lista. Tomamos el caso en que se excede el límite inferior y decidimos que en tal situación queremos lanzar una excepción de tipo `OverflowException`.

```
[Test]
public void SubstractExcedingLowerLimit()
{
    Calculator calculator = new Calculator(-100, 100);
    try
    {
        int result = calculator.Substract(10, 150);
        Assert.Fail("Exception␣is␣not␣being␣thrown␣when␣" +
          "exceeding␣lower␣limit");
    }
    catch (OverflowException)
    {
        // Ok, the SUT works as expected
    }
}
```

Efectivamente el test falla. Si el método *Substract* hubiese lanzado la excepción, ésta hubiese sido capturada por el bloque `catch` que silenciosamente hubiese concluido la ejecución del test. Un test que concluye calladito como este es un test con luz verde. No es obligatorio que exista una sentencia *Assert*, aunque es conveniente usarlas para aumentar la legibilidad del código. Para conseguir luz verde ya no vale lanzar la excepción sin hacer ninguna comprobación porque los otros tests fallarían. Necesitamos poder consultar los límites definidos y actuar en consecuencia. Esto nos recuerda que hay líneas en nuestra lista que tenemos que resolver antes que la que nos ocupa. Por tanto dejamos aparcado éste (como el test falla no se nos olvidará retomarlo, de lo contrario deberíamos apuntarlo en la libreta) y nos encargamos de los casos de definición y consulta de límites:

```
[Test]
public void SetAndGetUpperLimit()
{
    Calculator calculator = new Calculator(-100, 100);
    Assert.AreEqual(100, calculator.UpperLimit);
}
```

No compila, hay que definir la propiedad `UpperLimit` en `Calculator`. Puesto que la propiedad `LowerLimit` es exáctamente del mismo tipo que `UpperLimit`, aquí podemos atrevernos a escribir el código que asigna y recupera ambas.

```
public class Calculator
{
```

```
3         private int _upperLimit;
4         private int _lowerLimit;
5
6         public int LowerLimit
7         {
8             get { return _lowerLimit; }
9             set { _lowerLimit = value; }
10        }
11
12        public int UpperLimit
13        {
14            get { return _upperLimit; }
15            set { _upperLimit = value; }
16        }
17
18        public Calculator(int minValue, int maxValue)
19        {
20            _upperLimit = maxValue;
21            _lowerLimit = minValue;
22        }
```

Así, tiene sentido añadir otro `Assert` al test en que estamos trabajando y cambiarle el nombre ... ¿No habíamos dicho que era conveniente que un test tuviera un único `Assert` y probase una sóla cosa? Es que semánticamente o funcionalmente ambas propiedades de la clase son para lo mismo, desde el punto de vista del test: asignar valores y recuperar valores de variables de instancia. O sea que no estamos infringiendo ninguna norma. Reconocer qué es lo que el test está probando es importantísimo para separar adecuadamente la funcionalidad en sus respectivos métodos o clases. Cuando se escribe un test sin tener claro lo que se pretende, se obtiene un resultado doblemente negativo: código de negocio problemático y un test difícil de mantener.

```
1 [Test]
2 public void SetAndGetLimits()
3 {
4     Calculator calculator = new Calculator(-100, 100);
5     Assert.AreEqual(100, calculator.UpperLimit);
6     Assert.AreEqual(-100, calculator.LowerLimit);
7 }
```

El valor de los tests es que nos obligan a pensar y a descubrir el sentido de lo que estamos haciendo. Escribir tests no debe convertirse en una cuestión de copiar y pegar, sino en una toma de decisiones. Es por eso que en algunos casos es permisible incluir varios *Assert* dentro de un mismo test y en otros no; depende de si estamos probando la misma casuística aplicada a varios elementos o no.

Ejecutamos los tests y pasan todos menos `SubstractExcedingLowerLimit` por lo que nos ponemos manos a la obra y escribimos el mínimo código posible que le haga funcionar y no rompa los demás.

```
1 public int Substract(int arg1, int arg2)
```

```
2 {
3     int result = arg1 - arg2;
4     if (result < _lowerLimit)
5     {
6        throw new OverflowException("Lower limit exceeded");
7     }
8     return result;
9 }
```

Nos queda probar el caso en el que el resultado excede el límite superior y los casos en que los argumentos también exceden los límites. Vamos paso a paso:

```
1  [Test]
2  public void AddExcedingUpperLimit()
3  {
4      Calculator calculator = new Calculator(-100, 100);
5      try
6      {
7          int result = calculator.Add(10, 150);
8          Assert.Fail("This should fail: we're exceding upper limit");
9      }
10     catch (OverflowException)
11     {
12         // Ok, the SUT works as expected
13     }
14 }
```

He tomado el método `Add` en lugar de restar para no olvidar que estas comprobaciones se aplican a todas las operaciones de la calculadora. Implementación mínima:

```
1 public int Add(int arg1, int arg2)
2 {
3     int result = arg1 + arg2;
4     if (result > _upperLimit)
5     {
6         throw new OverflowException("Upper limit exceeded");
7     }
8     return result;
9 }
```

Funciona pero se ve claramente que este método de suma no hace la comprobación del límite inferior. ¿Es posible que el resultado de una suma sea un número menor que el límite inferior? Si uno de sus argumentos es un número más pequeño que el propio límite inferior, entonces es posible. Entonces es el momento de atacar los casos en que los parámetros que se pasan superan ya de por sí los límites establecidos.

```
1  [Test]
2  public void ArgumentsExceedLimits()
3  {
4      Calculator calculator = new Calculator(-100, 100);
5      try
6      {
```

```
7           calculator.Add(
8              calculator.UpperLimit + 1, calculator.LowerLimit - 1);
9           Assert.Fail("This␣should␣fail:␣arguments␣exceed␣limits");
10      }
11      catch (OverflowException)
12      {
13          // Ok, this works
14      }
15  }
```

Este test se asegura de no caer en el caso anterior (el de que el resultado de la suma es inferior al límite) y aprovecha para probar ambos límites. Dos comprobaciones en el mismo test, lo cual es válido porque son realmente la misma característica. A por el verde:

```
1   public int Add(int arg1, int arg2)
2   {
3       if (arg1 > _upperLimit)
4           throw new OverflowException(
5                 "First␣argument␣exceeds␣upper␣limit");
6       if (arg2 < _lowerLimit)
7           throw new OverflowException(
8                 "Second␣argument␣exceeds␣lower␣limit");
9       int result = arg1 + arg2;
10      if (result > _upperLimit)
11      {
12          throw new OverflowException("Upper␣limit␣exceeded");
13      }
14      return result;
15  }
```

¿Y qué tal a la inversa?

```
1   [Test]
2   public void ArgumentsExceedLimitsInverse()
3   {
4       Calculator calculator = new Calculator(-100, 100);
5       try
6       {
7           calculator.Add(
8               calculator.LowerLimit -1, calculator.UpperLimit + 1);
9           Assert.Fail("This␣should␣fail:␣arguments␣exceed␣limits");
10      }
11      catch (OverflowException)
12      {
13          // Ok, this works
14      }
15  }
```

Pintémoslo de verde!:

```
1   public int Add(int arg1, int arg2)
2   {
3       if (arg1 > _upperLimit)
4           throw new OverflowException(
5                 "First␣argument␣exceeds␣upper␣limit");
6       if (arg2 < _lowerLimit)
```

```
7           throw new OverflowException(
8                 "First argument exceeds lower limit");
9       if (arg1 < _lowerLimit)
10          throw new OverflowException(
11                "Second argument exceeds lower limit");
12      if (arg2 > _upperLimit)
13          throw new OverflowException(
14                "Second argument exceeds upper limit");
15
16      int result = arg1 + arg2;
17      if (result > _upperLimit)
18      {
19          throw new OverflowException("Upper limit exceeded");
20      }
21      return result;
22  }
```

La resta debería comportarse igual:

```
1   [Test]
2   public void ArgumentsExceedLimitsOnSubstract()
3   {
4       Calculator calculator = new Calculator(-100, 100);
5       try
6       {
7           calculator.Substract(
8                calculator.UpperLimit + 1, calculator.LowerLimit - 1);
9           Assert.Fail("This should fail: arguments exceed limits");
10      }
11      catch (OverflowException)
12      {
13          // Ok, this works
14      }
15  }
```

El test no pasa. Lo más rápido sería copiar las líneas de validación de la suma y pegarlas en la resta. Efectivamente podemos hacerlo, luego ver que los tests pasan y después observar que existe duplicidad y exige refactorizar. Esto es lo aconsejable para lo programadores menos experiementados. Sin embargo, algo tan evidente puede ser abreviado en un solo paso por el desarrollador experto. Estamos ante un caso perfecto para refactorizar extrayendo un método:

```
1   public bool ValidateArgs(int arg1, int arg2)
2   {
3       if (arg1 > _upperLimit)
4           throw new OverflowException(
5                 "First argument exceeds upper limit");
6       if (arg2 < _lowerLimit)
7           throw new OverflowException(
8                 "First argument exceeds lower limit");
9       if (arg1 < _lowerLimit)
10          throw new OverflowException(
11                "Second argument exceeds lower limit");
12      if (arg2 > _upperLimit)
13          throw new OverflowException(
```

```
14                 "Second argument exceeds upper limit");
15          return true;
16      }
17
18      public int Add(int arg1, int arg2)
19      {
20          ValidateArgs(arg1, arg2);
21
22          int result = arg1 + arg2;
23          if (result > _upperLimit)
24          {
25              throw new OverflowException("Upper limit exceeded");
26          }
27          return result;
28      }
29
30      public int Substract(int arg1, int arg2)
31      {
32          ValidateArgs(arg1, arg2);
33
34          int result = arg1 - arg2;
35          if (result < _lowerLimit)
36          {
37              throw new OverflowException("Lower limit exceeded");
38          }
39          return result;
40      }
```

Los tests pasan. ¿Queda más código duplicado?. Sí, todavía queda algo en el SUT y es la línea que llama al método de validación pero de eso nos encargarmos después. Tener una sola línea duplicada no es muy malo... ¿lo es? (la duda es buena querido lector; y va a ser que sí que es malo). ¿Están todos los casos de uso probados?. La libreta dice:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
# Aceptación - Límite Superior =100
# Aceptación - Límite Superior =500
# Aceptación - Límite Inferior = -1000
# Aceptación - Límite Inferior = -10
# Aceptación - Limite Superior=100 y parámetro mayor que 100, produce ERROR
# Aceptación - Limite Superior=100 y resultado mayor que 100, produce ERROR
# Aceptación - Limite Inferior=10 y parámetro menor que 10, produce ERROR
# Aceptación - Limite Inferior=10 y resultado menor que 10, produce ERROR
```

Las últimas líneas albergan múltiples ejemplos y retenerlos todos mentalmente es peligroso, es fácil que dejemos algunos atrás por lo que expandimos la lista:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
# Aceptación - Límite Superior =100
# Aceptación - Límite Superior =500
# Aceptación - Límite Inferior = -1000
# Aceptación - Límite Inferior = -10
▪ A: El primer argumento sobrepasa el límite superior
▪ B: El primer argumento sobrepasa el límite inferior
▪ C: El segundo argumento sobrepasa el límite superior
▪ D: El segundo argumento sobrepasa el límite inferior
▪ E: El resultado de una operación sobrepasa el límite superior
▪ F: El resultado de una operación sobrepasa el límite inferior
▪ Todos los casos de uso anteriores se aplican
   a todas las operaciones aritmeticas
```

No hemos probado por completo que la resta valida sus dos argumentos, sólo hemos probado los casos A y D restando. Necesitaríamos otro test más. Si escribimos dos tests para la validación en cada operación aritmética, vamos a terminar con una cantidad de tests muy grande e inútil (porque en verdad están todos probando la misma cosa) a base de copiar y pegar. Esto empieza a oler mal. Cuando se avecina la jugada de copiar y pegar tests a diestro y siniestro, la cosa huele mal. ¿Qué necesitamos probar? Necesitamos asegurarnos de que el validador valida y de que todas las operaciones aritméticas preguntan al validador. En verdad es ésto lo que queremos. Nos hemos dado cuenta al identificar un mal olor. De acuerdo, modificamos los dos tests que hacen al validador comprobar los argumentos:

```
1 [Test]
2 public void ArgumentsExceedLimits()
3 {
4     Calculator calculator = new Calculator(-100, 100);
5     try
6     {
7         calculator.ValidateArgs(
8             calculator.UpperLimit + 1, calculator.LowerLimit - 1);
9         Assert.Fail("This␣should␣fail:␣arguments␣exceed␣limits");
```

```
10      }
11      catch (OverflowException)
12      {
13          // Ok, this works
14      }
15  }
16
17  [Test]
18  public void ArgumentsExceedLimitsInverse()
19  {
20      Calculator calculator = new Calculator(-100, 100);
21      try
22      {
23          calculator.ValidateArgs(
24               calculator.LowerLimit -1, calculator.UpperLimit + 1);
25          Assert.Fail("This␣should␣fail:␣arguments␣exceed␣limits");
26      }
27      catch (OverflowException)
28      {
29          // Ok, this works
30      }
31  }
```

¿Cómo comprobamos ahora que las operaciones aritméticas validan primero sin repetir código? Porque tal como está ahora el test sería el mismo código, solo que cambiando `ValidateArgs` por `Add` o `Substract`. Lo que queremos validar no es el resultado de las funciones matemáticas, que ya está probado con otros tests, sino su comportamiento. Y cuando aparece la necesidad de validar comportamiento hay que detenerse un momento y analizar si las clases cumplen el *Principio de una sola responsabilidad*[3]. La clase `Calculator` se concibió para realizar operaciones aritméticas y ahora también está haciendo validaciones. Tiene más de una responsabilidad. De hecho el modificador `public` con que se definió el método `ValidateArgs` quedaba bastante raro, cualquiera hubiera dicho que se debería haber definido como privado. A menudo los métodos privados son indicadores de colaboración entre clases, es decir, puede que en lugar de definir el método como privado sea más conveniente extraer una clase y hacer que ambas cooperen.

Vamos a escribir el primer test que valida la cooperación entre la calculadora y el validador incluso aunque todavía no hemos separado el código... ¡El test siempre primero! y para ello nos servimos del framework *Rhino.Mocks*.

```
1  [Test]
2  public void SubstractIsUsingValidator()
3  {
4      int arg1 = 10;
5      int arg2 = -20;
6      int upperLimit = 100;
7      int lowerLimit = 100;
8      var validatorMock =
9          MockRepository.GenerateStrictMock<LimitsValidator>();
```

[3]Ver Capítulo 7 en la página 111

```
10      validatorMock.Expect(x => x.ValidateArgs(arg1, arg2));
11
12      Calculator calculator = new Calculator(validatorMock);
13      calculator.Add(arg1, arg2);
14
15      validatorMock.VerifyAllExpectations();
16  }
```

El código dice que hay un objeto que implementa la interfaz `LimitsValidator` y que se espera que se llame a su método `ValidateArgs`. Crea una instancia nueva de la calculadora y le inyecta el validador como parámetro en el constructor, aunque no es el validador verdadero sino un impostor (un mock). A continuación se ejecuta la llamada al método de suma y finalmente se le pregunta al mock si las expectativas se cumplieron, es decir, si se produjo la llamada tal cual se especificó. Hemos decidido modificar el constructor de la calculadora para tomar una instancia de un validador en lugar de los valores límite. Al fin y al cabo los límites sólo le sirven al validador. Parece que es lo que queríamos hacer pero... entonces, ¿para comprobar que todas las operaciones aritméticas hablan con el validador tenemos que copiar y pegar este test y modificarle una línea? ¡Sigue oliendo mal!

Los métodos de suma y resta no solo están realizando sus operaciones aritméticas respectivas, sino que incluyen una parte extra de lógica de negocio que es la que dice... antes y después de operar hay que validar. ¿No sería mejor si hubiese una clase que coordinase esto?. Desde luego el mal olor del copiar/pegar indica que hay que cambiar algo. Es cierto, si la responsabilidad de la calculadora (la clase `Calculator`, no la aplicación) es resolver operaciones pequeñas, que sea otra quien se encargue de operar comandos más complejos. Lo que queremos hacer tiene toda la pinta del patrón Decorador[4].

> En Python los decoradores existen como parte del lenguaje; son funciones que interceptan la llamada al método decorado. En este sentido tienen más potencia que los atributos de C# (lo que va entre corchetes), que no interceptan sino sólo marcan. Por lo tanto un decorador a lo Python parece apropiado aquí. Sin embargo sé por experiencia que tal herramienta del lenguaje debe limitarse a funciones muy pequeñas que añaden atributos a la función decorada. A veces con la propia carga de módulos Python en memoria se ejecuta el código de los decoradores con resultados impredecibles. Además si el código del decorador va más allá de etiquetar al decorado, estamos dejando de hacer programación orientada a objetos para regresar a la vieja programación procedimental.

[4] http://es.wikipedia.org/wiki/Decorator_(patrón_de_diseño)

En C# tenemos varias alternativas. La más común sería que la clase coordinadora implementase la misma interfaz de la calculadora y que tuviese una instancia de la calculadora internamente de manera que "envolviese" la llamada y le añadiese código[5]. Lo malo de esta solución es que nos lleva de nuevo a mucho código duplicado. Lo más elegante sería el patrón Proxy[6] para interceptar la llamada. Una opción es *Castle.DynamicProxy2*, que es la base de *Rhino.Mocks* pero la curva de aprendizaje que conlleva usarlo, aunque es suave nos desvía de la materia que estamos tratando, por lo que vamos a implementar nuestra propia forma de proxy. Vamos a modificar el test anterior para explicar con un ejemplo qué es lo que queremos:

```
1  [Test]
2  public void CoordinateValidation()
3  {
4      int arg1 = 10;
5      int arg2 = -20;
6      int result = 1000;
7      int upperLimit = 100;
8      int lowerLimit = -100;
9
10     var validatorMock =
11            MockRepository.GenerateStrictMock<LimitsValidator>();
12     validatorMock.Expect(x => x.SetLimits(
13                           lowerLimit, upperLimit)).Repeat.Once();
14     validatorMock.Expect(x => x.ValidateArgs(
15                           arg1, arg2)).Repeat.Once();
16
17     var calculatorMock =
18            MockRepository.GenerateStrictMock<BasicCalculator>();
19     calculatorMock.Expect(x => x.Add(arg1, arg2)).Return(result);
20
21     validatorMock.Expect(x => x.ValidateResult(
22                           result)).Repeat.Once();
23
24     CalcProxy calcProxy =
25          new CalcProxy(validatorMock,
26                        calculatorMock, lowerLimit, upperLimit);
27     calcProxy.BinaryOperation(calculatorMock.Add, arg1, arg2);
28
29     validatorMock.VerifyAllExpectations();
30     calculatorMock.VerifyAllExpectations();
31 }
```

Lo que dice este ejemplo o test es lo siguiente: Existe un validador al cual se invocará mediante los métodos `SetLimits` y `ValidateArgs` consecutivamente (y una sola vez cada uno). Existe una calculadora[7] que ejecutará su operación de suma y acto seguido el validador chequeará el resultado. Has-

[5] http://www.dofactory.com/Patterns/PatternDecorator.aspx#_self1

[6] http://es.wikipedia.org/wiki/Proxy_(patrón_de_diseño)

[7] Nótese que estamos usando interfaces como punto de partida para la generación de los *mocks*; el principal motivo es que así nos aseguramos que no se ejecuta la llamada en ninguna clase particular sino sólo en el *mock*

ta ahí hemos definido las expectativas. Ahora decimos que hay un proxy (`CalcProxy`) que recibe como parámetros de su constructor al validador, la calculadora y los límites máximos permitidos para las operaciones aritméticas. Queremos que exista un método `BinaryOperation` donde se indique el método de la calculadora a invocar y sus parámetros. Finalmente verificamos que la ejecución del proxy ha satisfecho las expectativas definidas. ¿Complicado ,no?

Como vimos en el capítulo anterior, el test es realmente frágil. Cuenta con todo lujo de detalles lo que hace el SUT. Es como si quisiéramos implementarlo. Personalmente descarto esta opción. Pensar en este test y escribirlo me ha ayudado a pensar en el diseño pero he ido demasiado lejos. Si puedo evitar los mocks en este punto mejor y como ninguna de las operaciones requeridas infringen las reglas de los tests unitarios, voy a seguir utilizando validación de estado. Es momento de replantearse la situación.

¿De qué manera podemos probar que el supuesto proxy colabora con validador y calculadora sin usar mocks? Respuesta: Podemos ejercitar toda la funcionalidad de que disponemos a través del proxy y fijarnos en que no haya duplicidad. Si no hay duplicidad y todos los "casos de uso" se gestionan mediante el proxy, entonces tiene que ser que está trabajando bien. Plantearlo así nos supone el esfuerzo de mover tests de sitio. Por ejemplo los de suma y resta los quitaríamos de la calculadora y los pondríamos en el proxy, ya que no los vamos a tener por duplicado. Empecemos por implementar primero el test de suma en el proxy:

```
1  [TestFixture]
2  public class CalcProxyTests
3  {
4      private Calculator _calculator;
5      private CalcProxy _calcProxy;
6
7      [Test]
8      public void Add()
9      {
10         _calculator = new Calculator();
11         _calcProxy = new CalcProxy(_calculator);
12         int result =
13            _calcProxy.BinaryOperation(_calculator.Add, 2, 2);
14         Assert.AreEqual(4, result);
15     }
16 }
```

Por cierto, hemos eliminado el test de suma del conjunto `CalculatorTests` (para no duplicar). De la clase `Calculator` he movido las propiedades de límite inferior y límite superior a una clase `Validator` junto con el método `ValidateArgs` por si en breve los reutilizase. El SUT mímimo es:

```
1  public class CalcProxy
2  {
```

```
3       private BasicCalculator _calculator;
4
5       public CalcProxy(BasicCalculator calculator)
6       {
7           _calculator = calculator;
8       }
9
10      public int BinaryOperation(
11              SingleBinaryOperation operation,
12              int arg1, int arg2)
13      {
14               return _calculator.Add(arg1, arg2);
15      }
16  }
```

He decidido que el primer parámetro del SUT es un delegado:

```
1  public delegate int SingleBinaryOperation(int a, int b);
```

> En lugar de pasar una función como primer parámetro de `BinaryOperation` podríamos haber usado una cadena de texto ("Add") pero la experiencia nos dice que las cadenas son frágiles y hacen el código propenso a errores difíciles de corregir y detectar. Si la persona que se está enfrentando a estas decisiones tuviese poca experiencia y hubiese decidido utilizar cadenas, igualmente tendría muchas ventajas al usar TDD. Seguramente su código incluiría un gran bloque `switch-case` para actuar en función de las cadenas de texto y en algún momento pudiera ser que tuviese que reescribir funciones pero al tener toda una batería de pruebas detrás, tales cambios serían menos peligrosos, le darían mucha más confianza. Así, aunque TDD no nos da siempre la respuesta a cuál es la mejor decisión de diseño, nos echa una mano cuando tenemos que retroceder y enmendar una decisión problemática.

En el capítulo 11 repetiremos la implementación con TDD pero sobre Python, así que no se preocupe si algo no le queda del todo claro. Serán los mismos casos que en este capítulo pero marcados por las particularidades de Python. Además en el capítulo 9 continuamos trabajando con TDD, avanzando en la implementación de la solución.

Vamos a triangular el proxy trasladando el test de la resta hasta él:

```
1   [TestFixture]
2   public class CalcProxyTests
3   {
4       private Calculator _calculator;
5       private CalcProxy _calcProxy;
6
7       [SetUp]
```

```
8       public void SetUp()
9       {
10          _calculator = new Calculator();
11          _calcProxy = new CalcProxy(_calculator);
12      }
13
14      [Test]
15      public void Add()
16      {
17          int result =
18              _calcProxy.BinaryOperation(_calculator.Add, 2, 2);
19          Assert.AreEqual(4, result);
20      }
21
22      [Test]
23      public void Substract()
24      {
25          int result =
26               _calcProxy.BinaryOperation(
27                    _calculator.Substract, 5, 3);
28          Assert.AreEqual(2, result);
29      }
30  }
```

Ya está más difícil buscar el código mínimo para que los dos tests pasen. No vamos a escribir un bloque condicional para conseguir luz verde porque eso no triangula a ninguna parte. Es hora de implementar algo más serio.

```
1   public int BinaryOperation(SingleBinaryOperation operation,
2                              int arg1, int arg2)
3   {
4       int result = 0;
5       MethodInfo[] calcultatorMethods =
6             _calculator.GetType().GetMethods(BindingFlags.Public |
7                                              BindingFlags.Instance);
8       foreach (MethodInfo method in calcultatorMethods)
9       {
10          if (method == operation.Method)
11          {
12              result = (int)method.Invoke(
13                    _calculator, new Object[] { arg1, arg2 });
14          }
15      }
16      return result;
17  }
```

Hemos usado un poco de magia *Reflection*[8] para buscar dinámicamente el método de la clase calculadora que toca invocar. Los dos tests pasan y ya han sido eliminados del conjunto en que se encontraban inicialmente. Estamos empezando a notar que reescribir no cuesta mucho cuando hacemos TDD. Una pregunta frecuente de quienes comienzan a aprender TDD es si los tests se pueden modificar. Aquí estamos viendo claramente que sí. Se pueden modificar tantas veces como haga falta porque un test es código vivo, tan

[8] http://msdn.microsoft.com/es-es/library/system.reflection(VS.95).aspx

importante como el SUT. Lo único inamovible es el test de aceptación porque ha sido definido por el cliente. Al menos es inamovible hasta la siguiente reunión de fin de sprint con el cliente (sprint si usamos Scrum). Terminemos de mover los tests de calculadora al proxy:

```
[TestFixture]
public class CalcProxyTests
{
    private Calculator _calculator;
    private CalcProxy _calcProxy;

    [SetUp]
    public void SetUp()
    {
        _calculator = new Calculator();
        _calcProxy = new CalcProxy(_calculator);
    }

    [Test]
    public void Add()
    {
        int result =
           _calcProxy.BinaryOperation(_calculator.Add, 2, 2);
        Assert.AreEqual(4, result);
    }

    [Test]
    public void Substract()
    {
        int result =
              _calcProxy.BinaryOperation(
                                    _calculator.Substract, 5, 3);
        Assert.AreEqual(2, result);
    }

    [Test]
    public void AddWithDifferentArguments()
    {
       int result =
             _calcProxy.BinaryOperation(_calculator.Add, 2, 5);
       Assert.AreEqual(7, result);
    }

    [Test]
    public void SubstractReturningNegative()
    {
        int result =
             _calcProxy.BinaryOperation(
                                      _calculator.Substract, 3, 5);
        Assert.AreEqual(-2, result);
    }
}
```

Perfecto, todos pasan estupendamente con un esfuerzo mínimo. Repasemos la libreta:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
# Aceptación - Límite Superior =100
# Aceptación - Límite Superior =500
# Aceptación - Límite Inferior = -1000
# Aceptación - Límite Inferior = -10
▪ A: El primer argumento sobrepasa el límite superior
▪ B: El primer argumento sobrepasa el límite inferior
▪ C: El segundo argumento sobrepasa el límite superior
▪ D: El segundo argumento sobrepasa el límite inferior
▪ E: El resultado de una operación sobrepasa el límite superior
▪ F: El resultado de una operación sobrepasa el límite inferior
▪ Todos los casos de uso anteriores se aplican
   a todas las operaciones aritméticas
```

Habíamos llegado a este punto para coordinar la validación de argumentos y resultados. No vamos a implementar el validador con su propio conjunto de tests para luego moverlos al proxy sino que ya con las ideas claras y el diseño más definido podemos ejercitar el SUT desde el proxy:

```
1  [Test]
2  public void ArgumentsExceedLimits()
3  {
4      CalcProxy calcProxyWithLimits =
5           new CalcProxy(new Validator(-10, 10), _calculator);
6
7      try
8      {
9          _calcProxy.BinaryOperation(_calculator.Add, 30, 50);
10         Assert.Fail(
11            "This␣should␣fail␣as␣arguments␣exceed␣both␣limits");
12     }
13     catch (OverflowException)
14     {
15             // Ok, this works
16     }
17 }
```

He decidido que el proxy tiene un constructor que recibe al validador y a la

calculadora. Al validador se le indican los valores límite vía constructor. El SUT:

```
public int BinaryOperation(SingleBinaryOperation operation,
                           int arg1, int arg2)
{
    _validator.ValidateArgs(arg1, arg2);

    int result = 0;
    MethodInfo[] calcultatorMethods =
          _calculator.GetType().GetMethods(BindingFlags.Public |
                                           BindingFlags.Instance);
    foreach (MethodInfo method in calcultatorMethods)
    {
        if (method == operation.Method)
        {
            result = (int)method.Invoke(
                      _calculator, new Object[] { arg1, arg2 });
        }
    }
    return result;
}
```

El método simplemente añade una línea al código anterior. Para que el test pase rescatamos el método de validación que teníamos guardado en el validador.

```
public class Validator : LimitsValidator
{
    private int _upperLimit;
    private int _lowerLimit;

    public Validator(int lowerLimit, int upperLimit)
    {
        SetLimits(lowerLimit, upperLimit);
    }

    public int LowerLimit
    {
        get { return _lowerLimit; }
        set { _lowerLimit = value; }
    }

    public int UpperLimit
    {
        get { return _upperLimit; }
        set { _upperLimit = value; }
    }

    public void ValidateArgs(int arg1, int arg2)
    {
        if (arg1 > _upperLimit)
            throw new OverflowException("ERROR");
        if (arg2 > _upperLimit)
            throw new OverflowException("ERROR");
    }

    public void SetLimits(int lower, int upper)
```

```
32      {
33          _lowerLimit = lower;
34          _upperLimit = upper;
35      }
36  }
```

Nos queda probar el límite inferior.

```
1   [Test]
2   public void ArgumentsExceedLimitsInverse()
3   {
4        CalcProxy calcProxyWithLimits =
5             new CalcProxy(new Validator(-10, 10), _calculator);
6
7        try
8        {
9            calcProxyWithLimits.BinaryOperation(
10                                  _calculator.Add, -30, -50);
11           Assert.Fail("
12  ␣␣␣␣␣␣␣␣␣␣␣␣This␣should␣fail␣as␣arguments␣exceed␣both␣limits");
13       }
14       catch (OverflowException)
15       {
16           // Ok, this works
17       }
18  }
```

El SUT junto con su posterior refactoring:

```
1   public class Validator : LimitsValidator
2   {
3       private int _upperLimit;
4       private int _lowerLimit;
5
6       public Validator(int lowerLimit, int upperLimit)
7       {
8           SetLimits(lowerLimit, upperLimit);
9       }
10
11      public int LowerLimit
12      {
13          get { return _lowerLimit; }
14          set { _lowerLimit = value; }
15      }
16
17      public int UpperLimit
18      {
19          get { return _upperLimit; }
20          set { _upperLimit = value; }
21      }
22
23      public void ValidateArgs(int arg1, int arg2)
24      {
25          breakIfOverflow(arg1, "First␣argument␣exceeds␣limits");
26          breakIfOverflow(arg2, "Second␣argument␣exceeds␣limits");
27      }
28
29      private void breakIfOverflow(int arg, string msg)
30      {
```

```
31          if (ValueExceedLimits(arg))
32              throw new OverflowException(msg);
33      }
34
35      public bool ValueExceedLimits(int arg)
36      {
37          if (arg > _upperLimit)
38              return true;
39          if (arg < _lowerLimit)
40              return true;
41          return false;
42      }
43
44      public void SetLimits(int lower, int upper)
45      {
46          _lowerLimit = lower;
47          _upperLimit = upper;
48      }
49  }
```

Ya podemos quitar de la libreta unas cuantas líneas:

```
# Aceptación - "2 + 2", devuelve 4
▪ La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
▪ E: El resultado de una operación sobrepasa el límite superior
▪ F: El resultado de una operación sobrepasa el límite inferior
▪ Todos los casos de uso anteriores se aplican
   a todas las operaciones aritméticas
```

Solo nos queda validar el resultado. Los dos ejemplos y su implementación son inmediatos. Pero siempre de uno en uno:

```
1   [Test]
2   public void ValidateResultExceedingUpperLimit()
3   {
4       try
5       {
6           _calcProxyWithLimits.BinaryOperation(
7                               _calculator.Add, 10, 10);
8           Assert.Fail(
9           "This␣should␣fail␣as␣result␣exceed␣upper␣limit");
10      }
11      catch (OverflowException)
```

```
12      {
13          // Ok, this works
14      }
15  }
```

8.1: CalcProxy

```
1   public int BinaryOperation(SingleBinaryOperation operation,
2                              int arg1, int arg2)
3   {
4       _validator.ValidateArgs(arg1, arg2);
5
6       int result = 0;
7       MethodInfo[] calcultatorMethods =
8          _calculator.GetType().GetMethods(BindingFlags.Public |
9                                           BindingFlags.Instance);
10      foreach (MethodInfo method in calcultatorMethods)
11      {
12          if (method == operation.Method)
13          {
14              result = (int)method.Invoke(
15                      _calculator, new Object[] { arg1, arg2 });
16          }
17      }
18      _validator.ValidateResult(result);
19      return result;
20  }
```

Le hemos añadido una línea al método para validar el resultado. El resto del SUT en el validador:

8.2: Validator

```
1   public void ValidateResult(int result)
2   {
3       breakIfOverflow(result, "Result␣exceeds␣limits");
4   }
```

```
1   [Test]
2   public void ValidateResultExceedingLowerLimit()
3   {
4       try
5       {
6           _calcProxyWithLimits.BinaryOperation(
7                                       _calculator.Add, -20, -1);
8           Assert.Fail(
9               "This␣should␣fail␣as␣result␣exceed␣lower␣limit");
10      }
11      catch (OverflowException)
12      {
13          // Ok, this works
14      }
15  }
```

Para este último test ni siquiera ha hecho falta tocar el SUT. La libreta queda así:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2"devuelve 0
```

Para recapitular un poco veamos de nuevo todos los tests que hemos escrito hasta el momento, que han quedado bajo el mismo conjunto de tests, `CalcProxyTests`. Al final no hemos utilizado ningún doble de test y todos son tests unitarios puesto que cumplen todas sus reglas.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using NUnit.Framework; // only nunit.framework dll is required
5  using SuperCalculator;
6  using Rhino.Mocks;
7
8  namespace UnitTests
9  {
10     [TestFixture]
11     public class CalcProxyTests
12     {
13         private Calculator _calculator;
14         private CalcProxy _calcProxy;
15         private CalcProxy _calcProxyWithLimits;
16
17         [SetUp]
18         public void SetUp()
19         {
20             _calculator = new Calculator();
21             _calcProxy =
22                 new CalcProxy(
23                         new Validator(-100, 100), _calculator);
24             _calcProxyWithLimits =
25                 new CalcProxy(new Validator(-10, 10), _calculator);
26         }
27
28         [Test]
29         public void Add()
30         {
31             int result =
32                 _calcProxy.BinaryOperation(_calculator.Add, 2, 2);
33             Assert.AreEqual(4, result);
34         }
35
36         [Test]
```

```
37          public void Substract()
38          {
39              int result =
40                     _calcProxy.BinaryOperation(
41                             _calculator.Substract, 5, 3);
42              Assert.AreEqual(2, result);
43          }
44
45          [Test]
46          public void AddWithDifferentArguments()
47          {
48              int result =
49                   _calcProxy.BinaryOperation(_calculator.Add, 2, 5);
50              Assert.AreEqual(7, result);
51          }
52
53          [Test]
54          public void SubstractReturningNegative()
55          {
56              int result =
57                      _calcProxy.BinaryOperation(
58                              _calculator.Substract, 3, 5);
59              Assert.AreEqual(-2, result);
60          }
61
62          [Test]
63          public void ArgumentsExceedLimits()
64          {
65              try
66              {
67                  _calcProxyWithLimits.BinaryOperation(
68                                          _calculator.Add, 30, 50);
69                  Assert.Fail(
70                  "This␣should␣fail␣as␣arguments␣exceed␣both␣limits");
71              }
72              catch (OverflowException)
73              {
74                  // Ok, this works
75              }
76          }
77
78          [Test]
79          public void ArgumentsExceedLimitsInverse()
80          {
81              try
82              {
83                  _calcProxyWithLimits.BinaryOperation(
84                                  _calculator.Add, -30, -50);
85                  Assert.Fail(
86                  "This␣should␣fail␣as␣arguments␣exceed␣both␣limits");
87              }
88              catch (OverflowException)
89              {
90                  // Ok, this works
91              }
92          }
93
94          [Test]
95          public void ValidateResultExceedingUpperLimit()
```

```
96          {
97              try
98              {
99                  _calcProxyWithLimits.BinaryOperation(
100                                 _calculator.Add, 10, 10);
101                 Assert.Fail(
102                 "This␣should␣fail␣as␣result␣exceed␣upper␣limit");
103             }
104             catch (OverflowException)
105             {
106                 // Ok, this works
107             }
108         }
109
110         [Test]
111         public void ValidateResultExceedingLowerLimit()
112         {
113             try
114             {
115                 _calcProxyWithLimits.BinaryOperation(
116                                 _calculator.Add, -20, -1);
117                 Assert.Fail(
118                 "This␣should␣fail␣as␣result␣exceed␣upper␣limit");
119             }
120             catch (OverflowException)
121             {
122                 // Ok, this works
123             }
124         }
125     }
126 }
```

Es un buen momento para hacer un "commit" en el sistema de control de versiones y cerrar el capítulo. Puede encontrar un archivo comprimido con el estado actual del proyecto en la web, para que lo pueda revisar si lo desea. En próximos capítulos podríamos hacer modificaciones sobre las clases actuales, por eso el archivo contiene expresamente la versión que hemos desarrollado hasta aquí.

En el próximo capítulo continuaremos el desarrollo de la Supercalculadora con C#, para seguir profundizando en la técnica del diseño dirigido por ejemplos o TDD. En el capítulo 11 implementaremos lo mismo desde el inicio con Python.

Capítulo 9

# Continuación del proyecto - Test Unitarios

En el último capítulo llegamos a conseguir que nuestra calculadora sumase y restase teniendo en cuenta los valores límite de los parámetros y del resultado. Continuamos el desarrollo por donde lo dejamos atendiendo a lo que pone la libreta:

```
# Aceptación - "2 + 2", devuelve 4
▪  La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

Es el momento de evaluar la cadena de texto que se utiliza para introducir expresiones y conectarla con la funcionalidad que ya tenemos. Empezamos a diseñar partiendo de un ejemplo, como siempre:

```
1  [TestFixture]
2  public class ParserTests
3  {
4      [Test]
5      public void GetTokens()
6      {
7          MathParser parser = new MathParser();
8          List<MathToken> tokens = parser.GetTokens("2␣+␣2");
```

```
9
10          Assert.AreEqual(3, tokens.Count);
11          Assert.AreEqual("2", tokens[0].Token);
12          Assert.AreEqual("+", tokens[1].Token);
13          Assert.AreEqual("2", tokens[2].Token);
14      }
15  }
```

Acabo de tomar varias decisiones de diseño: `MathParser` es una clase con un método `GetTokens` que recibe la expresión como una cadena y devuelve una lista de objetos tipo `MathToken`. Tales objetos todavía no existen pero prefiero pensar en la expresión como en una lista de objetos en lugar de una lista de cadenas. La experiencia me dice que devolver cadenas no me hará progresar mucho. La implementación mínima para alcanzar verde:

```
1   public class MathParser
2   {
3       public List<MathToken> GetTokens(string expression)
4       {
5           List<MathToken> tokens = new List<MathToken>();
6
7           tokens.Add(new MathToken("2"));
8           tokens.Add(new MathToken("+"));
9           tokens.Add(new MathToken("2"));
10
11          return tokens;
12      }
13  }
```

La simplicidad de este SUT nos sirve para traer varias preguntas a la mente. Afortunadamente las respuestas ya se encuentran en la libreta: sabemos qué expresiones son válidas y cuales no. Además sabemos que en caso de encontrar una cadena incorrecta lanzaremos una excepción. Podríamos triangular hacia el reconocimiento de las expresiones con sentencias y bloques de código varios pero las expresiones regulares son la mejor opción llegado este punto. En lugar de construir de una vez la expresión regular que valida todo tipo de expresiones matemáticas, vamos a triangular paso a paso. Una expresión regular compleja nos puede llevar días de trabajo depurando. Si construimos la expresión basándonos en pequeños ejemplos que vayan casando con cada subexpresión regular, más tarde, su modificación y sofisticación, nos resultara mas sencilla. TDD es ideal para diseñar expresiones regulares si mantenemos la máxima de escribir un test exclusivo para cada posible cadena válida. Vamos con el ejemplo que construirá la primera versión de la expresión regular:

```
1   [Test]
2   public void ValidateMostSimpleExpression()
3   {
4       string expression = "2␣+␣2";
5       bool result = _parser.IsExpressionValid(expression);
```

```
6
7       Assert.IsTrue(result);
8   }
```

En lugar de un método `void` me ha parecido mejor idea que devuelva verdadero o falso para facilitar la implementación de los tests.

En vez de retornar verdadero directamente podemos permitirnos construir la expresión regular más sencilla que resuelve el ejemplo:

9.1: MathParser

```
1   public bool IsExpressionValid(string expression)
2   {
3       Regex regex = new Regex(@"\d␣\+␣\d");
4       return regex.IsMatch(expression);
5   }
```

¿Qué tal se comporta con números de más de una cifra?

```
1   [Test]
2   public void ValidateMoreThanOneDigitExpression()
3   {
4       string expression = "25␣+␣287";
5       bool result = _parser.IsExpressionValid(expression);
6
7       Assert.IsTrue(result);
8   }
```

¡Funciona! No hemos tenido que modificar el SUT. Ahora vamos a probar con los cuatro operadores aritméticos. En lugar de hacer cuatro tests nos damos cuenta de que la expresión que queremos probar es la misma, aunque variando el operador. Eso nos da permiso para agrupar los cuatro usos en un solo ejemplo:

```
1   [Test]
2   public void ValidateSimpleExpressionWithAllOperators()
3   {
4       string operators = "+-*/";
5       string expression = String.Empty;
6       foreach (char operatorChar in operators)
7       {
8           expression = "2␣" + operatorChar + "␣2";
9           Assert.IsTrue(
10                      _parser.IsExpressionValid(expression),
11                      "Failure␣with␣operator:␣" + operatorChar);
12      }
13  }
```

El peligro de este ejemplo es que estemos construyendo mal la cadena, en cuyo caso diseñaríamos mal el SUT. Después de escribirlo la he mostrado por consola para asegurarme que era la que quería. En mi opinión merece la pena asumir el riesgo para agrupar tests de una forma ordenada. Fijése que en el `Assert` he añadido una explicación para que sea más sencilla la depuración de bugs.

Incrementamos la expresión regular para hacer el test pasar.

9.2: MathParser

```
public bool IsExpressionValid(string expression)
{
    Regex regex = new Regex(@"\d␣[+|\-|/|*]␣\d");

    return regex.IsMatch(expression);
}
```

El test pasa. Podemos eliminar el primero que habíamos escrito (`Validate-MostSimpleExpression`) ya que está contenido en el último. Es importante recordar que el código de los tests es tan importante como el del SUT y que por tanto debemos cuidarlo y mantenerlo.

Me asalta una duda... ¿podrá haber varios espacios entre los distintos elementos de la expresión? Preguntamos, nos confirman que sí es posible y anotamos en la libreta.

```
# Aceptación - "2 + 2", devuelve 4
▪ La cadena "2 + 2"tiene dos números y un operador que son '2', '2' y '+'
▪ Se permiten varios espacios entre símbolos
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

De acuerdo, probémoslo:

```
[Test]
public void ValidateWithSpaces()
{
    string expression = "2␣␣␣␣+␣␣␣287";
    bool result = _parser.IsExpressionValid(expression);
    Assert.IsTrue(result);
}
```

Mejoramos la expresión regular:

9.3: MathParser

```
public bool IsExpressionValid(string expression)
{
    Regex regex = new Regex(@"\d\s+[+|\-|/|*]\s+\d");

    return regex.IsMatch(expression);
}
```

¿Estará cubierto el caso en que no se dejan espacios?

```
[Test]
public void ValidateFailsNoSpaces()
{
     string expression = "2+7";
     bool result = _parser.IsExpressionValid(expression);
     Assert.IsFalse(result);
}
```

Pues sí, funciona sin que tengamos que tocar el SUT. Escogemos nuevas expresiones de la libreta:

```
[Test]
public void ValidateComplexExpression()
{
    string expression = "2␣+␣7␣-␣2␣*␣4";
    bool result = _parser.IsExpressionValid(expression);

    Assert.IsTrue(result);
}
```

Vaya, esta pasa incluso sin haber modificado la expresión regular. Resulta que, como una subcadena de la expresión casa, nos la está dando por buena. Busquemos un test que nos obligue a modificar la expresión regular:

```
[Test]
public void ValidateComplexWrongExpression()
{
    string expression = "2␣+␣7␣a␣2␣b␣4";
    bool result = _parser.IsExpressionValid(expression);

    Assert.IsFalse(result);
}
```

9.4: MathParser

```
public bool IsExpressionValid(string expression)
{
   Regex regex = new Regex(
         @"^\d((\s+)[+|\-|/|*](\s+)\d)+$");

   return regex.IsMatch(expression, 0);
}
```

Algunos tests que antes funcionaban están fallando. Vamos a retocar más la expresión regular:

9.5: MathParser

```
public bool IsExpressionValid(string expression)
{
    Regex regex = new Regex(
          @"^\d+((\s+)[+|\-|/|*](\s+)\d+)+$");

    return regex.IsMatch(expression, 0);
}
```

El hecho de que algunos otros tests se hubieran roto me ha creado cierta desconfianza. Vamos a probar unas cuantas expresiones más para verificar que nuestra validación es buena.

```
[Test]
public void ValidateSimpleWrongExpression()
{
    string expression = "2a7";
    bool result = _parser.IsExpressionValid(expression);

    Assert.IsFalse(result);
}
```

El test pasa. A por otro caso:

```
[Test]
public void ValidateWrongExpressionWithValidSubexpression()
{
    string expression = "2␣+␣7␣-␣2␣a␣3␣b";
    bool result = _parser.IsExpressionValid(expression);

    Assert.IsFalse(result);
}
```

También funciona. ¿Qué tal con dos operadores consecutivos?

```
[Test]
public void ValidateWithSeveralOperatorsTogether()
{
     string expression = "+␣+␣7";
     bool result = _parser.IsExpressionValid(expression);

     Assert.IsFalse(result);
}
```

Correcto, luz verde. La expresión que nos queda por probar de las que tiene la libreta es aquella que contiene números negativos:

```
[Test]
public void ValidateWithNegativeNumers()
{
     Assert.IsTrue(_parser.IsExpressionValid("-7␣+␣1"));
}
```

He aprovechado para simplificar el test sin que pierda legibilidad. Por cierto, está en rojo; hay que retocar la expresión regular.

9.6: MathParser

```
public bool IsExpressionValid(string expression)
{
   Regex regex = new Regex(
        @"^-{0,1}\d+((\s+)[+|\-|/|*](\s+)-{0,1}\d+)+$");
     return regex.IsMatch(expression, 0);
}
```

Funciona. Probemos alguna variante:

```
[Test]
public void ValidateWithNegativeNumersAtTheEnd()
{
    Assert.IsTrue(_parser.IsExpressionValid("7 - -1"));
}
```

Sigue funcionando. Vamos a por la última prueba del validador de expresiones.

```
[Test]
public void ValidateSuperComplexExpression()
{
    Assert.IsTrue(_parser.IsExpressionValid(
                    "-7 - -1 * 2 / 3 + -5"));
}
```

Me da la sensación de que nuestro validador de expresiones ya es suficientemente robusto. Contiene toda la funcionalidad que necesitamos por ahora. ¿Dónde estábamos?

```
# Aceptación - "2 + 2", devuelve 4
▪ La cadena "2 + 2"tiene dos números y un
  operador que son '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "* * 4 - 2": produce ERROR
# Aceptación - "* 4 5 - 2": produce ERROR
# Aceptación - "* 4 5 - 2 : produce ERROR
# Aceptación - "*45-2-": produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

¡Ah si!, le estábamos pidiendo al analizador que nos devolviera una lista con los elementos de la expresión. Habíamos hecho pasar un test con una implementación mínima pero no llegamos a triangular:

```
[Test]
public void GetTokensLongExpression()
{
    List<MathToken> tokens = _parser.GetTokens("2 - 1 + 3");

    Assert.AreEqual(5, tokens.Count);
    Assert.AreEqual("+", tokens[3].Token);
    Assert.AreEqual("3", tokens[4].Token);
}
```

Nótese que no repetimos las afirmaciones referentes a los tokens 0, 1 y 2 que ya se hicieron en el test anterior para una expresión que es casi igual a la actual.

### 9.7: MathParser

```
public List<MathToken> GetTokens(string expression)
{
    List<MathToken> tokens = new List<MathToken>();
    String[] items = expression.Split('␣');
    foreach (String item in items)
    {
        tokens.Add(new MathToken(item));
    }
    return tokens;
}
```

Tengo la sensación de que la clase `Parser` empieza a tener demasiadas responsabilidades. Refactoricemos:

```
public class MathLexer
{
    public List<MathToken> GetTokens(string expression)
    {
        List<MathToken> tokens = new List<MathToken>();
        String[] items = expression.Split('␣');
        foreach (String item in items)
        {
            tokens.Add(new MathToken(item));
        }
        return tokens;
    }
}

public class ExpressionValidator
{
    public bool IsExpressionValid(string expression)
    {
        Regex regex =
          new Regex(@"^-{0,1}\d+((\s+)[+|\-|/|*](\s+)-{0,1}\d+)+$");

        return regex.IsMatch(expression, 0);
    }
}

public class MathParser
{

}
```

Hemos tenido que renombrar algunas variables en los tests para que pasen después de esta refactorización pero ha sido rápido. Los he dejado dentro del conjunto de tests `ParserTests` aunque ahora se ha quedado vacía la clase `Parser`.

La libreta dice que ante una expresión inválida el analizador producirá una excepción. Escribamos un ejemplo que lo provoque:

```
[Test]
public void GetTokensWrongExpression()
{
    try
    {
```

```
6           List<MathToken> tokens = _lexer.GetTokens("2 - 1++ 3");
7           Assert.Fail("Exception did not arise!");
8       }
9       catch (InvalidOperationException)
10      { }
11  }
```

Nos hemos decantado por `InvalidOperationException`. Ahora podríamos escribir un "hack" veloz y triangular pero es un poco absurdo teniendo ya un validador de expresiones. Inyectemos el validador:

```
1   public class MathLexer
2   {
3       ExpressionValidator _validator;
4
5       public MathLexer(ExpressionValidator validator)
6       {
7           _validator = validator;
8       }
9
10      public List<MathToken> GetTokens(string expression)
11      {
12          if (!_validator.IsExpressionValid(expression))
13              throw new InvalidOperationException(expression);
14
15          List<MathToken> tokens = new List<MathToken>();
16          String[] items = expression.Split(' ');
17          foreach (String item in items)
18          {
19              tokens.Add(new MathToken(item));
20          }
21          return tokens;
22      }
23  }
```

¿Se creará bien la lista de tokens cuando haya varios espacios seguidos? Mejor lo apuntalamos con un test:

```
1   [Test]
2   public void GetTokensWithSpaces()
3   {
4       List<MathToken> tokens = _lexer.GetTokens("5 -   88");
5       Assert.AreEqual("5", tokens[0].Token);
6       Assert.AreEqual("-", tokens[1].Token);
7       Assert.AreEqual("88", tokens[2].Token);
8   }
```

Pues resulta que no funciona. Luz roja. Deberíamos poder partir por cualquier carácter en blanco:

9.8: MathLexer

```
1   public List<MathToken> GetTokens(string expression)
2   {
3       if (!_validator.IsExpressionValid(expression))
4           throw new InvalidOperationException(expression);
5
```

```
6       List<MathToken> tokens = new List<MathToken>();
7       String[] items = expression.Split((new char[] {' ', '\t'}),
8                         StringSplitOptions.RemoveEmptyEntries);
9       foreach (String item in items)
10      {
11          tokens.Add(new MathToken(item));
12      }
13      return tokens;
14  }
```

OK luz verde. Refactorizo un poco:

9.9: MathLexer

```
1   public List<MathToken> GetTokens(string expression)
2   {
3       if (!_validator.isExpressionValid(expression))
4           throw new InvalidOperationException(expression);
5
6       string[] items = splitExpression(expression);
7       return createTokensFromStrings(items);
8   }
9
10  private string[] splitExpression(string expression)
11  {
12      return expression.Split((new char[] { ' ', '\t' }),
13          StringSplitOptions.RemoveEmptyEntries);
14  }
15
16  private List<MathToken> createTokensFromStrings(string[] items)
17  {
18      List<MathToken> tokens = new List<MathToken>();
19      foreach (String item in items)
20      {
21          tokens.Add(new MathToken(item));
22      }
23      return tokens;
24  }
```

Limpiemos la lista para ver qué toca ahora:

```
# Aceptación - "2 + 2", devuelve 4
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

El primer test de aceptación de la lista nos exige comenzar a unir las distintas piezas que hemos ido creando. Por una lado sabemos que somos capaces de sumar y por otro ya conseguimos la lista de tokens de la expresión. Queda conectar ambas cosas. En este caso concreto el test de aceptación lo podemos expresar con NUnit. Aunque voy a intentar que cumpla con algunas normas de los tests unitarios (inocuo y rápido) no es atómico así que no me atrevería a llamarle unitario sino simplemente funcional.

```
1  [Test]
2  public void ProcessSimpleExpression()
3  {
4      MathParser parser = new MathParser();
5      Assert.AreEqual(4, parser.ProcessExpression("2␣+␣2"));
6  }
```

Antes de implementar el SUT mínimo, me parece buena idea escribir un par de tests unitarios que fuercen la colaboración entre los objetos que tenemos. Así no se me olvida utilizarlos cuando me adentre en detalles de implementación:

```
1  [Test]
2  public void ParserWorksWithCalcProxy()
3  {
4       CalculatorProxy calcProxyMock =
5             MockRepository.GenerateMock<CalculatorProxy>();
6      calcProxyMock.Expect(x =>
7             x.Calculator).Return(_calculator);
8      calcProxyMock.Expect(
9            x => x.BinaryOperation(_calculator.Add, 2,
10                                        2)).Return(4);
11
12     MathParser parser =
13           new MathParser(calcProxyMock);
14     parser.ProcessExpression("2␣+␣2");
15
16     calcProxyMock.VerifyAllExpectations();
17 }
```

Para escribir el test tuve que extraer la interfaz `CalculatorProxy` a partir de la clase `CalcProxy`. La intención es forzar la colaboración. No me gusta tener que ser tan explícito al definir la llamada a la propiedad `Calculator` del proxy en la línea 6. Siento que me gustaría que `Calculator` estuviese mejor encapsulado dentro del proxy. Es algo que tengo en mente arreglar tan pronto como un requisito me lo pida. Y seguro que aparece pronto. Conseguimos el verde rápido:

```
1  public class MathParser
2  {
3       CalculatorProxy _calcProxy;
4
5       public MathParser(CalculatorProxy calcProxy)
6       {
7             _calcProxy = calcProxy;
8       }
9
10      public int ProcessExpression(string expression)
11      {
12           return _calcProxy.BinayOperation(
13              _calcProxy.Calculator.Add, 2, 2);
14      }
15 }
```

Forcemos también la colaboración con `MathLexer`:

```
[Test]
public void ParserWorksWithLexer()
{
    List<MathToken> tokens = new List<MathToken>();
    tokens.Add(new MathToken("2"));
    tokens.Add(new MathToken("+"));
    tokens.Add(new MathToken("2"));
    Lexer lexerMock =
        MockRepository.GenerateStrictMock<Lexer>();
    lexerMock.Expect(
         x => x.GetTokens("2␣+␣2")).Return(tokens);

    MathParser parser = new MathParser(lexerMock,
          new CalcProxy(new Validator(-100, 100),
                                new Calculator()));
    parser.ProcessExpression("2␣+␣2");

    lexerMock.VerifyAllExpectations();
}
```

Extraje la interfaz `Lexer` para generar el mock. El SUT va tomando forma:

```
public class MathParser
{
    Lexer _lexer;
    CalculatorProxy _calcProxy;

    public MathParser(Lexer lexer, CalculatorProxy calcProxy)
    {
        _lexer = lexer;
        _calcProxy = calcProxy;
    }

    public int ProcessExpression(string expression)
    {
        List<MathToken> tokens = _lexer.GetTokens(expression);
        return _calcProxy.BinaryOperation(
                        _calcProxy.Calculator.Add,
                        tokens[0].IntValue,
                        tokens[2].IntValue);
     }
}
```

Modifiqué el test anterior ya que el constructor ha cambiado. Estos dos tests nos posicionan en un código mínimo sin llegar a ser el clásico `return 4`. Buen punto de partida para triangular hacia algo más útil.

```
[Test]
public void ProcessExpression2Operators()
{
     Assert.AreEqual(6 ,
             _parser.ProcessExpression("3␣+␣1␣+␣2"));
}
```

Voy a implementar un código mínimo que resuelva la operación procesando la entrada de izquierda a derecha:

9.10: MathParser

```
public int ProcessExpression(string expression)
{
     List<MathToken> tokens = _lexer.GetTokens(expression);
     MathToken total = tokens[0];
     for (int i = 0; i < tokens.Count; i++)
     {
          if (tokens[i].isOperator())
          {
               MathToken totalForNow = total;
               MathToken nextNumber = tokens[i + 1];
               int partialResult =
                    _calcProxy.BinaryOperation(
                         _calcProxy.Calculator.Add,
                         totalForNow.IntValue, nextNumber.IntValue);
               total = new MathToken(partialResult.ToString());
               i++;
          }
     }
     return total.IntValue;
}
```

Lo más sencillo que se me ha ocurrido es coger los operadores y dar por sentado que los operandos (números) están a su izquierda y derecha. Al escribir el código me he dado cuenta que necesitaba la función `isOperator` en la clase `MathToken` y he hecho uso de ella sin que esté implementada. Así pues dejo a un lado el SUT y voy a por un test que me ayude a implementar dicha función, ó sea, cambio de SUT un momento.

```
[TestFixture]
public class MathTokenTests
{
    [Test]
    public void isOperator()
    {
        MathToken numberToken = new MathToken("22");
        Assert.IsFalse(numberToken.isOperator());
    }
}
```

9.11: MathToken

```
public bool isOperator()
{
    string operators = "+-*/";
    foreach (char op in operators)
    {
        if (_token == op.ToString())
            return true;
    }
    return false;
}
```

Ahora el caso positivo:

```
[Test]
public void isOperatorTrue()
{
      MathToken numberToken = new MathToken("*");
      Assert.IsTrue(numberToken.isOperator());
}
```

Funciona. Podemos regresar al SUT anterior y ejecutar el test para comprobar que ... ¡también funciona!. La cosa marcha. Revisamos la lista a ver cómo vamos:

```
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

¡No hemos tenido en cuenta la precedencia de operadores! El test anterior no la exigía y me centré tanto en el código mínimo que cumplía con la especificación, que olvidé el criterio de aceptación de la precedencia de operadores. No pasa nada, seguro que es más fácil partir de aquí hacia la solución del problema que haber partido de cero. Cuando el problema se hace complejo como en el caso que nos ocupa, es especialmente importante no saltarse la regla de diseñar en pequeños pasos. Quizás si hubiésemos contemplado el caso complejo desde el principio hubiésemos olvidado casos que a la larga se hubiesen traducido en bugs.

Antes de seguir voy a refactorizar un poco los tests, moviendo los que son de validación de expresiones a un conjunto fuera de `ParserTests` ya que se está convirtiendo en una clase con demasiados tests[1].

Una vez movidos los tests tenemos que diseñar una estrategia para la precedencia de los operadores. De paso podemos añadir a la lista los casos en que se utilizan paréntesis en las expresiones, para irles teniendo en mente a la hora de tomar decisiones de diseño:

```
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
# Aceptación - "2 + -2", devuelve 0
```

Una vez más. Un ejemplo sencillo primero para afrontar el SUT:

```
[Test]
public void ProcessExpressionWithPrecedence()
{
      Assert.AreEqual(9, _parser.ProcessExpression("3␣+␣3␣*␣2"));
}
```

[1] Ver los cambios en el código fuente que acompaña al libro

Si el operador tuviera asociado un valor para su precedencia, podría buscar aquellos de mayor precedencia, operar y a continuación hacer lo mismo con los de menor. De paso para encapsular mejor la calculadora podría mover las llamadas al proxy a una clase operador. Como el test que nos ocupa me parece demasiado grande para implementar el SUT de una sola vez, voy a escribir otro test que me sirva para obtener la precedencia más alta de la expresión. Un poco más adelante retomaré el test que acabo de escribir.

```
[Test]
public void GetMaxPrecedence()
{
    List<MathToken> tokens = _lexer.GetTokens("3␣+␣3␣*␣2");
    MathOperator op = _parser.GetMaxPrecedence(tokens);
    Assert.AreEqual(op.Token, "*");
}
```

El SUT:

9.12: MathParser

```
public MathOperator GetMaxPrecedence(List<MathToken> tokens)
{
     int precedence = 0;
     MathOperator maxPrecedenceOperator = null;

     foreach (MathToken token in tokens)
     {
          if (token.isOperator())
          {
               MathOperator op = OperatorFactory.Create(token);
               if (op.Precedence >= precedence)
               {
                    precedence = op.Precedence;
                    maxPrecedenceOperator = op;
               }
          }
     }
     return maxPrecedenceOperator;
}
```

No compila porque las clases `MathOperator` y `OperatorFactory` no existen y el método `Create` tampoco. Voy a intentar que compile lo más rápidamente posible:

```
public class MathOperator
{
    int _precedence = 0;
    string _token = String.Empty;

    public string Token
    {
        get { return _token; }
        set { _token = value; }
    }

```

```
12      public int Precedence
13      {
14          get { return _precedence;  }
15      }
16  }
17
18  public class OperatorFactory
19  {
20      public static MathOperator Create(MathToken token)
21      {
22          MathOperator op = new MathOperator();
23          op.Token = token.Token;
24          return op;
25      }
26  }
```

Bien. El test para el método de obtener la máxima precedencia funciona parcialmente pero no hemos triangulado. Para ello tenemos que probar que la factoría de operadores les pone precedencia:

```
1   [TestFixture]
2   public class OperatorFactoryTests
3   {
4       [Test]
5       public void CreateMultiplyOperator()
6       {
7           MathOperator op = OperatorFactory.Create(new MathToken("*"));
8           Assert.AreEqual(op.Precedence, 2);
9       }
10  }
```

SUT mínimo:

9.13: OperatorFactory

```
1   public static MathOperator Create(MathToken token)
2   {
3       MathOperator op;
4       if (token.Token == "*")
5           op = new MathOperator(2);
6       else
7           op = new MathOperator(0);
8       op.Token = token.Token;
9       return op;
10  }
```

He tenido que añadir un constructor para `MathOperator` que reciba el valor de precedencia. El test pasa. Si escribo otro test para la división creo que por ahora tendré las precedencias resueltas:

```
1   [Test]
2   public void CreateDivisionOperator()
3   {
4       MathOperator op = OperatorFactory.Create(new MathToken("/"));
5       Assert.AreEqual(op.Precedence, 2);
6   }
```

9.14: OperatorFactory

```
public static MathOperator Create(MathToken token)
{
    MathOperator op;
    if ((token.Token == "*") || (token.Token == "/"))
        op = new MathOperator(2);
    else
        op = new MathOperator(0);
    op.Token = token.Token;
    return op;
}
```

Perfecto los tests pasan. Tanto `MathToken` como `MathOperator` comparten la propiedad `Token`. Empiezo a pensar que deberían compartir una interfaz. Podría refactorizar pero habrá que refactorizar más cosas pronto. Primero voy a terminar de implementar el SUT para el test que habíamos dejado en el aire:

```
[Test]
public void ProcessExpressionWithPrecedence()
{
    Assert.AreEqual(9, _parser.ProcessExpression("3␣+␣3␣*␣2"));
}
```

El SUT:

9.15: MathParser

```
public int ProcessExpression(string expression)
{
    List < MathToken > tokens = _lexer . GetTokens ( expression );
    while (tokens.Count > 1)
    {
        MathOperator op = GetMaxPrecedence(tokens);
        int firstNumber = tokens[op.Index -1].IntValue;
        int secondNumber = tokens[op.Index +1].IntValue;
        int result = op.Resolve(firstNumber, secondNumber);
        tokens[op.Index - 1] = new MathToken(result.ToString());
        tokens.RemoveAt(op.Index);
        tokens.RemoveAt(op.Index);
    }
    return tokens[0].IntValue;
}
```

He simplificado el algoritmo. Me limito a buscar el operador de mayor prioridad, operar los números a su izquierda y derecha y sustituir los tres elementos por el resultado. He necesitado una propiedad `Index` en el operator que no existía así que para poder compilar la añado:

9.16: MathParser

```
public MathOperator GetMaxPrecedence(List<MathToken> tokens)
{
    int precedence = 0;
    MathOperator maxPrecedenceOperator = null;
```

```
5
6       int index = -1;
7       foreach (MathToken token in tokens)
8       {
9           index++;
10          if (token.isOperator())
11          {
12              MathOperator op = OperatorFactory.Create(token);
13              if (op.Precedence >= precedence)
14              {
15                  precedence = op.Precedence;
16                  maxPrecedenceOperator = op;
17                  maxPrecedenceOperator.Index = index;
18              }
19          }
20      }
21      return maxPrecedenceOperator;
22  }
```

El método `Resolve` del operator tampoco existe. En mi cabeza es el método que encapsula el uso del proxy y a su vez la calculadora. Voy a implementar un stub rápido para compilar.

9.17: MathOperator

```
1   public int Resolve(int a, int b)
2   {
3       if (Token == "*")
4          return a * b;
5       if (Token == "+")
6          return a + b;
7       return 0;
8   }
```

Muy bien. Ahora pasan todos los tests menos aquel en el que forzábamos al analizador a utilizar el proxy, porque en el método `Resolve` hemos hecho un apaño rápido y feo. El primer cambio que voy a hacer es inyectar el proxy como parámetro en el método para utilizarlo:

9.18: MathOperator

```
1   public int Resolve(int a, int b, CalculatorProxy calcProxy)
2   {
3       if (Token == "*")
4           return a * b;
5       if (Token == "+")
6           return calcProxy.BinaryOperation(
7                   calcProxy.Calculator.Add, a, b);
8       return 0;
9   }
```

Perfecto. Todos los tests pasan. No he utilizado el proxy para la multiplicación porque no la tenemos implementada. Voy a añadir un par de tests sencillos de multiplicación y división para completar la funcionalidad de la clase `Calculator`. Los tests estaban dentro del conjunto de tests del proxy:

```
[Test]
public void Multiply()
{
    Assert.AreEqual(
        _calcProxy.BinaryOperation(_calculator.Multiply,
                                                2, 5), 10);
}
```

9.19: Calculator

```
public int Multiply(int arg1, int arg2)
{
    return arg1 * arg2;
}
```

```
[Test]
public void Division()
{
    Assert.AreEqual(
        _calcProxy.BinaryOperation(_calculator.Divide,
                                                10, 2), 5);
}
```

9.20: Calculator

```
public int Divide(int arg1, int arg2)
{
    return arg1 / arg2;
}
```

Bien, voy a completar el método que resuelve en el operador:

9.21: MathOperator

```
public int Resolve(int a, int b, CalculatorProxy calcProxy)
{
     if (Token == "*")
         return calcProxy.BinaryOperation(
                calcProxy.Calculator.Multiply, a, b);
     if (Token == "+")
         return calcProxy.BinaryOperation(
                calcProxy.Calculator.Add, a, b);
     return 0;
}
```

Todos los tests pasan. Aprovecho para refactorizar:

9.22: MathParser

```
public int ProcessExpression(string expression)
{
    List <MathToken> tokens = _lexer.GetTokens(expression);
    while (tokens.Count > 1)
    {
        MathOperator op = GetMaxPrecedence(tokens);
```

```
7           int firstNumber = tokens[op.Index -1].IntValue;
8           int secondNumber = tokens[op.Index +1].IntValue;
9           int result = op.Resolve(firstNumber,
10                                  secondNumber, _calcProxy);
11          replaceTokensWithResult(tokens, op.Index, result);
12      }
13      return tokens[0].IntValue;
14  }
15
16  private void replaceTokensWithResult(List<MathToken> tokens,
17      int indexOfOperator, int result)
18  {
19      tokens[indexOfOperator - 1] =
20                    new MathToken(result.ToString());
21      tokens.RemoveAt(indexOfOperator);
22      tokens.RemoveAt(indexOfOperator);
23  }
```

¿Cómo está la libreta?

```
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
# Aceptación - "2 + -2", devuelve 0
```

Estamos preparados para escribir un test de aceptación para la primera línea:

```
1  [Test]
2  public void ProcessAcceptanceExpression()
3  {
4      Assert.AreEqual(9, _parser.ProcessExpression("5␣+␣4␣*␣2␣/␣2"));
5  }
```

Se esperaba 9 pero se devolvió 5. ¡Ah claro! Es que el método de resolver no implementa la división ni la resta. Qué despiste. Voy a añadir la división para que el test pase y luego escribo otro test con una resta para estar obligado a implementarla.

9.23: MathOperator

```
1  public int Resolve(int a, int b, CalculatorProxy calcProxy)
2  {
3      if (Token == "*")
4          return calcProxy.BinaryOperation(
5                  calcProxy.Calculator.Multiply, a, b);
6      if (Token == "+")
7          return calcProxy.BinaryOperation(
8                  calcProxy.Calculator.Add, a, b);
9      if (Token == "/")
10         return calcProxy.BinaryOperation(
11                 calcProxy.Calculator.Divide, a, b);
12     return 0;
13 }
```

```
[Test]
public void ProcessAcceptanceExpressionWithAllOperators()
{
     Assert.AreEqual(8,
         _parser.ProcessExpression("5␣+␣4␣-␣1␣*␣2␣/␣2"));
}
```

9.24: MathOperator

```
public int Resolve(int a, int b, CalculatorProxy calcProxy)
{
    if (Token == "*")
       return calcProxy.BinaryOperation(
                 calcProxy.Calculator.Multiply, a, b);
    else if (Token == "+")
       return calcProxy.BinaryOperation(
                 calcProxy.Calculator.Add, a, b);
    else if (Token == "/")
       return calcProxy.BinaryOperation(
                 calcProxy.Calculator.Divide, a, b);
    else if (Token == "-")
       return calcProxy.BinaryOperation(
                calcProxy.Calculator.Substract, a, b);
    return 0;
}
```

¡Luz verde! ¿Algo por refactorizar? Ciertamente hay dos condicionales repetidas. En la factoría de operadores se pregunta por el token para asignar precedencia al operador y crearlo. En la resolución también se pregunta por el token para invocar al proxy. Si utilizamos polimorfismo podemos eliminar la condicional del método `Resolve` haciendo que cada operador específico implemente su propia resolución. Esta refactorización de hecho se llama así: reemplazar condicional con polimorfismo:

```
public abstract class MathOperator
{
    protected int _precedence = 0;
    protected string _token = String.Empty;
    int _index = -1;

    public MathOperator(int precedence)
    {
        _precedence = precedence;
    }

    public int Index
    {
        get { return _index; }
        set { _index = value; }
    }

    public string Token
    {
        get { return _token; }
    }
```

```
22
23      public int Precedence
24      {
25          get { return _precedence;  }
26      }
27
28      public abstract int Resolve(int a, int b,
29                      CalculatorProxy calcProxy);
30      }
31  }
32
33  public class MultiplyOperator : MathOperator
34  {
35      public MultiplyOperator()
36          : base(2)
37      {
38          _token = "*";
39      }
40
41      public override int Resolve(int a, int b,
42                                  CalculatorProxy calcProxy)
43      {
44          return calcProxy.BinaryOperation(
45                 calcProxy.Calculator.Multiply, a, b);
46      }
47  }
48
49  public class DivideOperator : MathOperator
50  {
51      public DivideOperator()
52          : base(2)
53      {
54          _token = "/";
55      }
56
57      public override int Resolve(int a, int b,
58                       CalculatorProxy calcProxy)
59      {
60          return calcProxy.BinaryOperation(
61                 calcProxy.Calculator.Divide, a, b);
62      }
63  }
64
65  public class AddOperator : MathOperator
66  {
67      public AddOperator()
68          : base(1)
69      {
70          _token = "+";
71      }
72
73      public override int Resolve(int a, int b,
74                                  CalculatorProxy calcProxy)
75      {
76          return calcProxy.BinaryOperation(
77                 calcProxy.Calculator.Add, a, b);
78      }
79  }
80
```

```
81  public class SubstractOperator : MathOperator
82  {
83      public SubstractOperator()
84          : base(1)
85      {
86          _token = "-";
87      }
88
89      public override int Resolve(int a, int b,
90                                  CalculatorProxy calcProxy)
91      {
92          return calcProxy.BinaryOperation(
93                 calcProxy.Calculator.Substract, a, b);
94      }
95  }
96
97  public class OperatorFactory
98  {
99      public static MathOperator Create(MathToken token)
100     {
101         if (token.Token == "*")
102             return new MultiplyOperator();
103         else if (token.Token == "/")
104             return new DivideOperator();
105         else if (token.Token == "+")
106             return new AddOperator();
107         else if (token.Token == "-")
108             return new SubstractOperator();
109
110         throw new InvalidOperationException(
111            "The␣given␣token␣is␣not␣a␣valid␣operator");
112     }
113 }
```

El código queda más claro y la condicional en un único sitio. Parecen muchas líneas pero ha sido una refactorización de tres minutos.

Aunque no hay ningún test que pruebe que el método `Create` lanza una excepción en caso que el token recibido no sea válido, el compilador me obliga a hacerlo, ya que de lo contrario tendría que devolver un objeto sin sentido. No veo la necesidad de escribir un test para ese caso porque ya tenemos un validador de expresiones y métodos que comprueban si un token es número u operador y ambas cosas están debidamente probadas.

Recordemos que la finalidad de TDD no es alcanzar una cobertura de tests del 100 % sino diseñar acorde a los requisitos mediante los ejemplos.

Repasemos la libreta:

```
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

Pensemos en las operaciones con paréntesis. En cómo resolver el proble-

ma. Aumentar la complejidad de la expresión regular que valida las expresiones matemáticas no me parece sostenible. Me da la sensación de que ir por ese camino hará el código más difícil de mantener, demasiado engorroso. Por otro lado no sabría utilizar expresiones regulares para comprobar que un paréntesis abierto casa con uno cerrado y cosas así. Si nos fijamos bien, el contenido de un paréntesis ha de ser una expresión de las que ya sabemos validar y resolver. Una expresión a su vez puede verse como una lista de tokens que en última instancia contiene un solo elemento que es un número. Vamos a partir el test de aceptación en unos cuantos tests de granularidad más fina para ir abordando poco a poco la implementación.

```
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
  - "(2 + 2)", se traduce en la expresión "2 + 2"
  - "((2) + 2)", se traduce en la expresión "2 + 2"
  - "(2 + 2", produce una excepción
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

El primero de los tests:

```
[Test]
public void GetExpressionsWith1Parenthesis()
{
    List<string> expressions =
        _lexer.GetExpressions("(2␣+␣2)");
    Assert.AreEqual(1, expressions.Count);
    Assert.AreEqual("2␣+␣2", expressions[0]);
}
```

De momento el test está dentro de `ParserTests` pero ya estoy pensando moverlo a un nuevo conjunto `LexerTests`. No voy a devolver un resultado fijo directamente sino a dar los primeros pasos en el algoritmo que encuentra expresiones dentro de los paréntesis.

9.25: MathLexer

```
public List<string> GetExpressions(string expression)
{
    List<string> expressions =
        new List<string>();
    Stack<char> parenthesis = new Stack<char>();
    foreach (char ch in expression)
    {
        if (ch == '(')
        {
            parenthesis.Push(ch);
            expressions.Add(String.Empty);
        }
        else if (ch == ')')
        {
```

```
15              parenthesis.Pop();
16          }
17          else
18          {
19              expressions[expressions.Count -1] +=
20                  ch.ToString();
21          }
22      }
23      return expressions;
24  }
```

Cada vez que se encuentra un paréntesis abierto se crea una nueva expresión. El algoritmo es simple. He utilizado una pila para llevar control de paréntesis abiertos y cerrados en vista de los próximos tests que hay en la libreta, aunque no estoy haciendo uso de ella al final del algoritmo. Eso lo dejaré para un test que lo requiera.

```
1   [Test]
2   public void GetExpressionsWithNestedParenthesis()
3   {
4       List<string> expressions =
5           _lexer.GetExpressions("((2)␣+␣2)");
6       Assert.AreEqual(1, expressions.Count);
7       Assert.AreEqual("2␣+␣2", expressions[0]);
8   }
```

El test falla porque la función está devolviendo dos expresiones, la primera de ellas vacía. Hay que limpiar expresiones vacías:

9.26: MathLexer

```
1   public List<string> GetExpressions(string expression)
2   {
3       List<string> expressions =
4           new List<string>();
5       Stack<char> parenthesis = new Stack<char>();
6       foreach (char ch in expression)
7       {
8           if (ch == '(')
9           {
10              parenthesis.Push(ch);
11              expressions.Add(String.Empty);
12          }
13          else if (ch == ')')
14          {
15              parenthesis.Pop();
16          }
17          else
18          {
19              expressions[expressions.Count -1] +=
20                  ch.ToString();
21          }
22      }
23      cleanEmptyExpressions(expressions);
24      return expressions;
25  }
```

```
26
27 private void cleanEmptyExpressions(List<string> expressions)
28 {
29     bool endOfList = false;
30     while (!endOfList)
31     {
32         endOfList = true;
33         for (int i = 0; i < expressions.Count; i++)
34             if (expressions[i].Length == 0)
35             {
36                 expressions.RemoveAt(i);
37                 endOfList = false;
38                 break;
39             }
40     }
41 }
```

Ya tenemos luz verde. Se me acaba de venir a la mente una pregunta. ¿Y si al leer las expresiones se forma una que no empieza por un número?. Ejemplo:

```
1  [Test]
2  public void GetNestedExpressions()
3  {
4      List<string> expressions =
5          _lexer.GetExpressions("((2␣+␣1)␣+␣2)");
6      Assert.AreEqual(3, expressions.Count);
7      foreach (string exp in expressions)
8          if ((exp != "2␣+␣1") &&
9              (exp != "+") &&
10             (exp != "2"))
11            Assert.Fail(
12                "Wrong␣expression␣split");
13 }
```

El test expresa mi decisión de evitar devolver expresiones del tipo "+ 1" prefiriendo los tokens sueltos, a las expresiones que no tienen sentido matemático por sí mismas. He tenido cuidado de no especificar en las afirmaciones las posiciones de las expresiones dentro del vector de expresiones para no escribir un test frágil. Lo que me interesa es el contenido de las cadenas y no la posición.

9.27: MathLexer

```
1  public List<string> GetExpressions(string expression)
2  {
3      List<string> expressions
4          = new List<string>();
5      Stack<int> parenthesis = new Stack<int>();
6      int index = 0;
7      foreach (char ch in expression)
8      {
9          if (ch == '(')
10         {
11             parenthesis.Push(index);
12             index++;
13             expressions.Add(String.Empty);
```

```
14          }
15          else if (ch == ')')
16          {
17              index = parenthesis.Pop();
18          }
19          else
20          {
21              expressions[index -1] +=
22                 ch.ToString();
23          }
24      }
25      cleanEmptyExpressions(expressions);
26      splitExpressionsStartingWithOperator(expressions);
27      return expressions;
28  }
29
30  private void splitExpressionsStartingWithOperator(
31       List<string> expressions)
32  {
33      Regex regex =
34          new Regex(@"^(\s*)[+|\-|/|*](\s+)");
35      bool endOfList = false;
36      while (!endOfList)
37      {
38          endOfList = true;
39          for (int i = 0; i < expressions.Count; i++)
40              if (regex.IsMatch(expressions[i]))
41              {
42                  string exp = expressions[i];
43                  exp = exp.Trim();
44                  string[] nexExps =
45                      exp.Split(new char[] { ' ', '\t' },
46                        2, StringSplitOptions.RemoveEmptyEntries);
47                  expressions[i] = nexExps[0];
48                  expressions.Insert(i + 1, nexExps[1]);
49                  endOfList = false;
50              }
51      }
52  }
```

La nueva función busca expresiones que empiecen por un operador y entonces las parte en dos; por un lado el operador y por otro el resto de la expresión. Por ahora no hace nada más.

El código está empezando a ser una maraña. Al escribir esta función me doy cuenta de que probablemente quiera escribir unos cuantos tests unitarios para cubrir otros usos de la misma pero el método es privado y eso limita la granularidad de los tests ya que no tengo acceso directo. Tener acceso desde el test a la función que se quiere probar sin pasar por otras funciones o métodos de entrada acelera la detección y corrección de defectos.

Recordemos que a veces los métodos privados sugieren ser movidos a clases colaboradoras. Vamos a hacer un poco de limpieza:

```
1   public class ExpressionFixer
2   {
```

```
3     public void CleanEmptyExpressions(List<string> expressions)
4     {
5         bool endOfList = false;
6           while (!endOfList)
7         {
8             endOfList = true;
9             for (int i = 0; i < expressions.Count; i++)
10                if (expressions[i].Length == 0)
11                {
12                    expressions.RemoveAt(i);
13                    endOfList = false;
14                    break;
15                }
16        }
17    }
18
19    public void SplitExpressionsStartingWithOperator(
20       List<string> expressions)
21    {
22        Regex regex =
23            new Regex(@"^(\s*)[+|\-|/|*](\s+)");
24        bool endOfList = false;
25        while (!endOfList)
26        {
27            endOfList = true;
28            for (int i = 0; i < expressions.Count; i++)
29                if (regex.IsMatch(expressions[i]))
30                {
31                    string exp = expressions[i];
32                    exp = exp.Trim();
33                    string[] nexExps =
34                       exp.Split(new char[] { ' ', '\t' },
35                         2, StringSplitOptions.RemoveEmptyEntries);
36                    expressions[i] = nexExps[0];
37                    expressions.Insert(i + 1, nexExps[1]);
38                    endOfList = false;
39                }
40        }
41    }
42 }
43
44 public class MathLexer : Lexer
45 {
46     ExpressionValidator _validator;
47     ExpressionFixer _fixer;
48
49     public MathLexer(ExpressionValidator validator,
50         ExpressionFixer fixer)
51     {
52         _validator = validator;
53         _fixer = fixer;
54     }
55
56    public List<string> GetExpressions(string expression)
57    {
58        List<string> expressions
59            = new List<string>();
60        Stack<int> parenthesis = new Stack<int>();
61        int index = 0;
```

```
62        foreach (char ch in expression)
63        {
64            if (ch == '(')
65            {
66                parenthesis.Push(index);
67                index++;
68                expressions.Add(String.Empty);
69            }
70            else if (ch == ')')
71            {
72                index = parenthesis.Pop();
73            }
74            else
75            {
76                expressions[index -1] +=
77                   ch.ToString();
78            }
79        }
80        _fixer.CleanEmptyExpressions(expressions);
81        _fixer.SplitExpressionsStartingWithOperator(expressions);
82        return expressions;
83    }
84    ...
```

He creado la nueva clase `ExpressionFixer` (reparador de expresiones) que se inyecta a `MathLexer`. Lógicamente he tenido que modificar las llamadas al constructor de lexer en los tests. Ahora me sigue pareciendo que hay duplicidad en los bucles de los dos métodos del reparador de expresiones. Vamos a afinar un poco más:

```
1   public class ExpressionFixer
2   {
3       public void FixExpressions(List<string> expressions)
4       {
5           bool listHasChanged = true;
6           while (listHasChanged)
7           {
8               listHasChanged = false;
9               for (int i = 0; i < expressions.Count; i++)
10                  if (DoesExpressionStartsWithOperator(expressions, i)
11                      || IsEmptyExpression(expressions, i))
12                  {
13                      listHasChanged = true;
14                      break;
15                  }
16          }
17      }
18
19      public bool IsEmptyExpression(List<string> expressions,
20                                    int index)
21      {
22          if (expressions[index].Length == 0)
23          {
24              expressions.RemoveAt(index);
25              return true;
26          }
27          return false;
```

```
28      }
29
30      public void DoesExpressionStartsWithOperator(
31         List<string> expressions, int index)
32      {
33          Regex regex =
34              new Regex(@"^(\s*)[+|\-|/|*](\s+)");
35          if (regex.IsMatch(expressions[index]))
36          {
37              string exp = expressions[index];
38              exp = exp.Trim();
39              string[] nexExps =
40                 exp.Split(new char[] { ' ', '\t' },
41                    2, StringSplitOptions.RemoveEmptyEntries);
42              expressions[i] = nexExps[0];
43              expressions.Insert(i + 1, nexExps[1]);
44              return true;
45          }
46          return false;
47      }
48  }
```

En `MathLexer` cambié las dos llamadas de las líneas 79 y 80 del penúltimo listado por una sola a `FixExpressions`. Todavía me parece que el código del último método tiene varias responsabilidades pero por ahora voy a parar de refactorizar y a anotarlo en la libreta para retomarlo en breve.

Ejecuto toda la batería de tests después de tanto cambio y veo que el test está funcionando pero que se ha roto el que habíamos escrito anteriormente. Voy a reescribirlo. Como no es un test de aceptación puedo cambiarlo sin problema.

```
1   [Test]
2   public void GetExpressionsWithNestedParenthesis()
3   {
4       List<string> expressions =
5           _lexer.GetExpressions("((2) + 2)");
6       foreach (string exp in expressions)
7           if ((exp != "2") &&
8               (exp != "+"))
9               Assert.Fail(
10                  "Wrong expression split");
11  }
```

Ahora ya funciona. Hay código duplicado en los tests. Lo arreglamos:

9.28: LexerTests

```
1   [Test]
2   public void GetExpressionsWithNestedParenthesis()
3   {
4       List<string> expressions =
5           _lexer.GetExpressions("((2) + 2)");
6       failIfOtherSubExpressionThan(expressions, "2", "+");
7   }
8
9   [Test]
```

```
10 public void GetNestedExpressions()
11 {
12     List<string> expressions =
13         _lexer.GetExpressions("((2␣+␣1)␣+␣2)");
14     Assert.AreEqual(3, expressions.Count);
15     failIfOtherSubExpressionThan(
16         expressions, "2␣+␣1", "+", "2");
17 }
18
19 private void failIfOtherSubExpressionThan(List<string> expressions,
20     params string[] expectedSubExpressions)
21 {
22     bool isSubExpression = false;
23     foreach(string subExpression in expectedSubExpressions)
24     {
25         isSubExpression = false;
26         foreach (string exp in expressions)
27             if (exp == subExpression)
28             {
29                 isSubExpression = true;
30                 break;
31             }
32         if (!isSubExpression)
33           Assert.Fail(
34                 "Wrong␣expression␣split:" + subExpression);
35     }
36 }
```

¿Cómo se comporta nuestra clase cuando el paréntesis aparece en la parte final de la expresión? Me ha surgido la duda mientras escribía el último SUT.

```
1 [Test]
2 public void GetExpressionWithParenthesisAtTheEnd()
3 {
4     List<string> expressions =
5         _lexer.GetExpressions("2␣+␣(3␣*␣1)");
6     failIfOtherSubExpressionThan(
7         expressions, "3␣*␣1", "+", "2");
8 }
```

De momento falla con una excepción. Corrijo el SUT:

9.29: MathLexer

```
1  public List<string> GetExpressions(string expression)
2  {
3      List<string> expressions
4          = new List<string>();
5      Stack<int> parenthesis = new Stack<int>();
6      int index = 1;
7      expressions.Add(String.Empty);
8      foreach (char ch in expression)
9      {
10         if (ch == '(')
11         {
12             parenthesis.Push(index);
13             index++;
14             expressions.Add(String.Empty);
```

```
15          }
16          else if (ch == ')')
17          {
18              index = parenthesis.Pop();
19          }
20          else
21          {
22              expressions[index -1] +=
23                 ch.ToString();
24          }
25      }
26      _fixer.FixExpressions(expressions);
27      return expressions;
28  }
```

El SUT no contemplaba que la expresión comenzase sin paréntesis (la línea 7 lo corrige). Ahora la ejecución no se interrumpe pero seguimos en rojo porque se están devolviendo las cadenas `"2 +"` y `"3 * 1"`. No estoy partiendo la expresión en caso que el operador quede al final. Vamos a escribir un test específico para el reparador de expresiones a fin de corregir el problema:

```
1   [TestFixture]
2   public class ExpressionFixerTests
3   {
4       [Test]
5       public void SplitExpressionWhenOperatorAtTheEnd()
6       {
7           ExpressionFixer fixer = new ExpressionFixer();
8           List<string> expressions = new List<string>();
9           expressions.Add("2␣+");
10          fixer.FixExpressions(expressions);
11          Assert.Contains("2", expressions);
12          Assert.Contains("+", expressions);
13      }
14  }
```

Efectivamente está fallando ahí. Voy a corregirlo y de paso a modificar un poco los nombres de los dos métodos para denotar que, a pesar de devolver verdadero o falso, modifican la lista de expresiones:

```
1   public class ExpressionFixer
2   {
3       public void FixExpressions(List<string> expressions)
4       {
5           bool listHasChanged = true;
6           while (listHasChanged)
7           {
8               listHasChanged = false;
9               for (int i = 0; i < expressions.Count; i++)
10                  if (IsNumberAndOperatorThenSplit(expressions, i) ||
11                      IsEmptyExpressionThenRemove(expressions, i))
12                  {
13                      listHasChanged = true;
14                      break;
15                  }
16          }
```

```
17      }
18
19      public bool IsNumberAndOperatorThenSplit(
20          List<string> expressions, int index)
21      {
22          Regex startsWithOperator =
23              new Regex(@"^(\s*)([+|\-|/|*])(\s+)");
24          Regex endsWithOperator =
25              new Regex(@"(\s+)([+|\-|/|*])(\s*)$");
26
27          string exp = expressions[index];
28          exp = exp.Trim();
29          if (startsWithOperator.IsMatch(exp) ||
30               endsWithOperator.IsMatch(exp))
31          {
32              splitByOperator(expressions, exp, index);
33              return true;
34          }
35          return false;
36      }
37
38      private void splitByOperator(List<string> expressions,
39          string inputExpression, int position)
40      {
41          string[] nextExps =
42                   Regex.Split(inputExpression, @"([+|\-|/|*])");
43          int j = position;
44          expressions.RemoveAt(j);
45          foreach (string subExp in nextExps)
46          {
47              expressions.Insert(j, subExp.Trim());
48              j++;
49          }
50      }
51
52      public bool IsEmptyExpressionThenRemove(List<string> expressions,
53                                              int index)
54      {
55          if (expressions[index].Length == 0)
56          {
57             expressions.RemoveAt(index);
58             return true;
59          }
60          return false;
61      }
62  }
```

Nótese que también extraje el método `splitByOperator` como resultado de otra refactorización. La hice en dos pasos aunque haya pegado el código una sola vez (todo sea por ahorrar papel).

Me costó trabajo decidir qué nombre ponerle a los métodos y al final el que hemos puesto denota claramente que cada método hace dos cosas. Está indicando que estamos violando el principio de una única responsabilidad. Tratemos de mejorar el diseño. Puesto que tenemos una clase que gestiona expresiones regulares (`ExpressionValidator`) tiene sentido que la

pregunta de si la expresión contiene un número y un operador pase a estar ahí:

```
public class ExpressionFixer
{
    ExpressionValidator _validator;

    public ExpressionFixer(ExpressionValidator validator)
    {
        _validator = validator;
    }

    public void FixExpressions(List<string> expressions)
    {
        bool listHasChanged = true;
        while (listHasChanged)
        {
            listHasChanged = false;
            for (int i = 0; i < expressions.Count; i++)
            {
                if (_validator.IsNumberAndOperator(
                          expressions[i]))
                {
                    splitByOperator(expressions,
                                    expressions[i], i);
                    listHasChanged = true;
                    break;
                }
                if (expressions[i]. Length == 0)
                {
                    expressions.RemoveAt(i);
                    listHasChanged = true;
                    break;
                }
            }
        }
    }

    private void splitByOperator(List<MathExpression> expressions,
        string inputExpression, int position)
    {
        string[] nextExps =
                  Regex.Split(inputExpression, @"([+|\-|/|*])");
        int j = position;
        expressions.RemoveAt(j);
        foreach (string subExp in nextExps)
        {
            expressions.Insert(j, new MathExpression(subExp.Trim()));
            j++;
        }
    }
}

public class ExpressionValidator
{
    public bool IsExpressionValid(string expression)
    {
        Regex fullRegex = new Regex(
             @"^-{0,1}\d+((\s+)[+|\-|/|*](\s+)-{0,1}\d+)+$");
```

```
57          Regex singleOperator = new Regex(@"^[+|\-|/|*]$");
58          Regex singleNumber = new Regex(@"^\d+$");
59          return (fullRegex.IsMatch(expression, 0) ||
60                  singleOperator.IsMatch(expression, 0) ||
61                  singleNumber.IsMatch(expression, 0));
62      }
63
64      public bool IsNumberAndOperator(string expression)
65      {
66          Regex startsWithOperator =
67              new Regex(@"^(\s*)([+|\-|/|*])(\s+)");
68          Regex endsWithOperator =
69              new Regex(@"(\s+)([+|\-|/|*])(\s*)$");
70
71          string exp = expression;
72          if (startsWithOperator.IsMatch(exp) ||
73               endsWithOperator.IsMatch(exp))
74              return true;
75          return false;
76      }
77  }
```

Ahora las responsabilidades están bien repartidas y ningún nombre de método suena extraño. El código fuente debe poderse entender fácilmente al leerlo y para eso es fundamental que los nombres describan con total precisión qué hacen los métodos.

Algo sigue sin encajar del todo. ¿`ExpressionValidator` es realmente un validador? Más bien es un clase de consulta de expresiones regulares del dominio. El que valida es lexer. Buen momento para hacer el cambio:

```
1   public class ExpressionFixer
2   {
3       MathRegex _mathRegex;
4
5       public ExpressionFixer(MathRegex mathRegex)
6       {
7           _mathRegex = mathRegex;
8       }
9       ...
10  }
11
12  public class MathRegex
13  {
14      public bool IsExpressionValid(string expression)
15      {
16         ...
17      }
18
19      public bool IsNumberAndOperator(string expression)
20      {
21         ...
22      }
23  }
```

Hemos renombrado `ExpressionValidator` por `MathRegex`. Mucho mejor ahora.

El test pasa y así también el anterior. Luz verde en toda la batería de tests. Revisamos la libreta:

```
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
▪ "(2 + 2", produce una excepción
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

Si un paréntesis abierto no encuentra correspondencia con otro cerrado, entonces excepción:

```
1  [Test]
2  public void ThrowExceptionOnOpenParenthesis()
3  {
4      try
5      {
6          List<string> expressions =
7              _lexer.GetExpressions("(2␣+␣3␣*␣1");
8          Assert.Fail("Exception␣didn't␣arise!");
9      }
10     catch (InvalidOperationException)
11     { }
12 }
```

Se arregla con dos líneas (26 y 27):

9.30: MathLexer

```
1  public List<string> GetExpressions(string expression)
2  {
3      List<string> expressions =
4          new List<string>();
5      Stack<int> parenthesis = new Stack<int>();
6      int index = 1;
7      expressions.Add(String.Empty);
8      foreach (char ch in expression)
9      {
10         if (ch == '(')
11         {
12             parenthesis.Push(index);
13             index++;
14             expressions.Add(String.Empty);
15         }
16         else if (ch == ')')
17         {
18             index = parenthesis.Pop();
19         }
20         else
21         {
22             expressions[index -1] +=
23                ch.ToString();
24         }
25     }
26     if (parenthesis.Count > 0)
27         throw new
```

```
28              InvalidOperationException("Parenthesis␣do␣not␣match");
29
30      _fixer.FixExpressions(expressions);
31      return expressions;
32  }
```

Aprovecho también para mover algunos tests a sus clases ya que `MathLexer` ha crecido bastante.

El código de `GetExpressions` me está empezando a hacer daño a la vista. Vamos a ver cómo nos las apañamos para refactorizarlo:

9.31: MathLexer

```
1   public List<string> GetExpressions(string expression)
2   {
3       List<string> totalExpressionsFound =
4           new List<string>();
5
6       int openedParenthesis = 0;
7       getExpressions(expression, 0,
8                      String.Empty, totalExpressionsFound,
9                      ref openedParenthesis);
10      if (openedParenthesis != 0)
11          throw new
12              InvalidOperationException("Parenthesis␣do␣not␣match");
13      _fixer.FixExpressions(totalExpressionsFound);
14      return totalExpressionsFound;
15  }
16
17  /// <summary>
18  /// Returns the position where the close parenthesis is found or
19  /// the position of the last char in the string.
20  /// Also populates the list of expressions along the way
21  /// </summary>
22  private int getExpressions(string fullInputExpression,
23      int subExpressionStartIndex,
24      string subExpressionUnderConstruction,
25      List<string> totalSubexpressionsFound,
26      ref int openedParanthesis)
27  {
28      for (int currentIndex = subExpressionStartIndex;
29          currentIndex < fullInputExpression.Length;
30          currentIndex++)
31      {
32          char currentChar = fullInputExpression[currentIndex];
33
34          if (currentChar == OPEN_SUBEXPRESSION)
35          {
36              openedParanthesis++;
37              int closePosition = getExpressions(
38                                      fullInputExpression,
39                                      currentIndex+ 1,
40                                      String.Empty,
41                                      totalSubexpressionsFound,
42                                      ref openedParanthesis);
43              currentIndex = closePosition;
44          }
45          else if (currentChar == CLOSE_SUBEXPRESSION)
```

```
46          {
47              totalSubexpressionsFound.Add(
48                      subExpressionUnderConstruction);
49              openedParanthesis--;
50              return currentIndex;
51          }
52          else
53          {
54              subExpressionUnderConstruction +=
55                    fullInputExpression[currentIndex].ToString();
56          }
57      }
58      totalSubexpressionsFound.Add(subExpressionUnderConstruction);
59      return fullInputExpression.Length;
60  }
```

Después de un rato dándole vueltas no encuentro una manera de deshacerme de ese bloque if-else. Si utilizo polimorfismo tengo la sensación de que se va a liar todavía más. Ahora la función es más natural, tiene la naturaleza recursiva de las propias expresiones con paréntesis anidados pero también es compleja. Lo que no me gusta nada es devolver un valor en la función recursiva que en algunos casos no utilizo: en la línea 7 del listado estoy invocándola sin utilizar para nada su valor de retorno. Esta es la razón por la que he creído conveniente añadir un comentario a la función, para sopesar la poca claridad de su código. Si pudiera cambiar eso me daría por satisfecho de momento.

Reintento:

9.32: MathLexer

```
1   public List<string> GetExpressions(string expression)
2   {
3       List<string> totalExpressionsFound =
4           new List<string>();
5
6       int openedParenthesis = 0;
7       int startSearchingAt = 0;
8       getExpressions(expression, ref startSearchingAt,
9                      String.Empty, totalExpressionsFound,
10                     ref openedParenthesis);
11      if (openedParenthesis != 0)
12          throw new
13              InvalidOperationException("Parenthesis␣do␣not␣match");
14      _fixer.FixExpressions(totalExpressionsFound);
15      return totalExpressionsFound;
16  }
17
18  private void getExpressions(string fullInputExpression,
19      ref int subExpressionStartIndex,
20      string subExpressionUnderConstruction,
21      List<string> totalSubexpressionsFound,
22      ref int openedParanthesis)
23  {
24      for (int currentIndex = subExpressionStartIndex;
25          currentIndex < fullInputExpression.Length;
26          currentIndex++)
```

```
27      {
28          char currentChar = fullInputExpression[currentIndex];
29          if (currentChar == OPEN_SUBEXPRESSION)
30          {
31              openedParanthesis++;
32              subExpressionStartIndex = currentIndex +1;
33              getExpressions(fullInputExpression,
34                             ref subExpressionStartIndex,
35                             String.Empty,
36                             totalSubexpressionsFound,
37                             ref openedParanthesis);
38              currentIndex = subExpressionStartIndex;
39          }
40          else if (currentChar == CLOSE_SUBEXPRESSION)
41          {
42              totalSubexpressionsFound.Add(
43                      subExpressionUnderConstruction);
44              subExpressionStartIndex = currentIndex;
45              openedParanthesis--;
46              return;
47          }
48          else
49          {
50              subExpressionUnderConstruction +=
51                  fullInputExpression[currentIndex].ToString();
52          }
53      }
54      totalSubexpressionsFound.Add(subExpressionUnderConstruction);
55      subExpressionStartIndex = subExpressionUnderConstruction.Length;
56  }
```

Ahora todos los tests están pasando y el código pinta algo mejor. Seguramente más adelante tengamos oportunidad de afinarlo.

Volvamos a la libreta:

```
# Aceptación - "(2 + 2) * (3 + 1)", devuelve 16
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

Voy a ver qué tal se maneja `MathLexer` con la expresión que tiene dos paréntesis:

```
1  [Test]
2  public void GetExpressionWithTwoGroups()
3  {
4      List<string> expressions =
5          _lexer.GetExpressions("(2␣+␣2)␣*␣(3␣+␣1)");
6      failIfOtherSubExpressionThan(
7          expressions, "3␣+␣1", "2␣+␣2", "*");
8  }
```

Vaya no funciona. Devuelve las subcadenas que queremos pero se han colado algunos espacios en blanco delante y detrás del asterisco. Escribimos otro test de grano más fino para solucionarlo:

```
1  [TestFixture]
2  public class ExpressionFixerTests
3  {
4      ExpressionFixer _fixer;
5      List<string> _expressions;
6
7      [SetUp]
8      public void SetUp()
9      {
10         _fixer = new ExpressionFixer();
11         _expressions = new List<string>();
12     }
13
14     [Test]
15     public void SplitExpressionWhenOperatorAtTheEnd()
16     {
17         _expressions.Add("2␣+");
18         _fixer.FixExpressions(_expressions);
19         Assert.Contains("2", _expressions);
20         Assert.Contains("+", _expressions);
21     }
22
23     [Test]
24     public void Trim()
25     {
26         _expressions.Add("␣*␣");
27         _fixer.FixExpressions(_expressions);
28         Assert.AreEqual("*", _expressions[0]);
29     }
```

Luz roja. Ya podemos arreglarlo:

9.33: EspressionFixer

```
1  public void FixExpressions(List<string> expressions)
2  {
3      bool listHasChanged = true;
4      while (listHasChanged)
5      {
6          listHasChanged = false;
7          for (int i = 0; i < expressions.Count; i++)
8          {
9              expressions[i] = expressions[i].Trim();
10             if (_mathRegex.IsNumberAndOperator(
11                     expressions[i]))
12             {
13                 splitByOperator(expressions,
14                                 expressions[i], i);
15                 listHasChanged = true;
16                 break;
17             }
18             if (expressions[i]. Length == 0)
19             {
20                 expressions.RemoveAt(i);
21                 listHasChanged = true;
22                 break;
23             }
24         }
```

```
25      }
26  }
```

Ahora funciona el test actual y el anterior (arreglado con línea 9). Excelente.

Hora de resolver una expresión que tiene paréntesis:

```
1   [Test]
2   public void ProcessAcceptanceExpressionWithParenthesis()
3   {
4       Assert.AreEqual(16,
5           _parser.ProcessExpression(
6            "(2␣+␣2)␣*␣(3␣+␣1)"));
7   }
```

Se lanza una excepción porque `parser` está intentando extraer de `lexer` los tokens en lugar de las expresiones[2]. Acabamos de diseñar para que se extraigan las expresiones primero y luego los tokens de cada una, como si tuviéramos dos niveles. Voy a adaptar el código:

9.34: MathParser

```
1   public int ProcessExpression(string expression)
2   {
3       List<string> subExpressions =
4           _lexer.GetExpressions(expression);
5       String flatExpression = String.Empty;
6       foreach (string subExp in subExpressions)
7       {
8           if (isSubExpression(subExp))
9               flatExpression += resolveSimpleExpression(subExp);
10          else
11              flatExpression += "␣" + subExp + "␣";
12      }
13      return resolveSimpleExpression(flatExpression);
14  }
15
16  private bool isSubExpression(string exp)
17  {
18      Regex operatorRegex = new Regex(@"[+|\-|/|*]");
19      Regex numberRegex = new Regex(@"\d+");
20      if (numberRegex.IsMatch(exp) &&
21          operatorRegex.IsMatch(exp))
22          return true;
23      return false;
24  }
25
26  private int resolveSimpleExpression(string expression)
27  {
28      List<MathToken> mathExp = _lexer.GetTokens(expression);
29      while (mathExp.Count > 1)
30      {
31          MathOperator op = GetMaxPrecedence(mathExp);
32
33          int firstNumber, secondNumber;
34          firstNumber = mathExp[op.Index - 1].IntValue;
```

[2]Ver listado 11.22 en página 155

```
35          secondNumber = mathExp[op.Index + 1].IntValue;
36          int result = op.Resolve(firstNumber,
37                                  secondNumber, _calcProxy);
38          replaceTokensWithResult(mathExp, op.Index, result);
39      }
40      return mathExp[0].IntValue;
41  }
```

El método `resolveSimpleExpression` ya lo teníamos escrito antes pero con el nombre `ProcessExpression`. El algoritmo aplana las expresiones hasta llegar a una expresión simple de las que sabe resolver. He escrito mucho código para implementar el SUT, no he sido estricto con la regla del código mínimo y esto me ha hecho anotar en la libreta que me gustaría escribir algunos tests para `isSubExpression` más adelante. Si no lo anoto en la libreta seguro que me olvido. El test sigue fallando porque el orden en que llegan las subexpresiones del lexer está cambiado. Está llegando `"2 + 2"` `"3 + 1"` y `"*"`. Para resolverlo tendré que escribir un nuevo test para lexer que exija que el orden de aparición de las subexpresiones se mantenga:

```
1   [Test]
2   public void GetSeveralParenthesisExpressionsInOrder()
3   {
4       List<string> expressions =
5           _lexer.GetExpressions("2␣+␣2)␣*␣(3␣+␣1)");
6       foreach (string exp in expressions)
7           Console.Out.WriteLine("x:" + exp + ".");
8       Assert.AreEqual("2␣+␣2", expressions[0]);
9       Assert.AreEqual("*", expressions[1]);
10      Assert.AreEqual("3␣+␣1", expressions[2]);
11  }
```

Tal como tenemos la función en el lexer puedo hacer que se registre el orden de aparición de cada subexpresión y luego ordenar si me ayudo de una nueva clase (`MathExpression`):

9.35: MathLexer

```
1   public List<MathExpression> GetExpressions(string expression)
2   {
3       List<MathExpression> totalExpressionsFound =
4           new List<MathExpression>();
5
6       int openedParenthesis = 0;
7       int startSearchingAt = 0;
8       getExpressions(expression, ref startSearchingAt,
9                      new MathExpression(String.Empty),
10                     totalExpressionsFound,
11                     ref openedParenthesis);
12      if (openedParenthesis != 0)
13          throw new
14              InvalidOperationException("Parenthesis␣do␣not␣match");
15      _fixer.FixExpressions(totalExpressionsFound);
16      return totalExpressionsFound;
17  }
```

```
18
19 private void getExpressions(string fullInputExpression,
20     ref int subExpressionStartIndex,
21     MathExpression subExpressionUnderConstruction,
22     List<MathExpression> totalSubexpressionsFound,
23     ref int openedParanthesis)
24 {
25     for (int currentIndex = subExpressionStartIndex;
26         currentIndex < fullInputExpression.Length;
27         currentIndex++)
28     {
29         char currentChar = fullInputExpression[currentIndex];
30         if (currentChar == OPEN_SUBEXPRESSION)
31         {
32             openedParanthesis++;
33             subExpressionStartIndex = currentIndex +1;
34             getExpressions(fullInputExpression,
35                            ref subExpressionStartIndex,
36                            new MathExpression(String.Empty,
37                                subExpressionStartIndex),
38                            totalSubexpressionsFound,
39                            ref openedParanthesis);
40             currentIndex = subExpressionStartIndex;
41         }
42         else if (currentChar == CLOSE_SUBEXPRESSION)
43         {
44             totalSubexpressionsFound.Add(
45                    subExpressionUnderConstruction);
46             subExpressionStartIndex = currentIndex;
47             openedParanthesis--;
48             return;
49         }
50         else
51         {
52             subExpressionUnderConstruction.Expression +=
53                 fullInputExpression[currentIndex].ToString();
54             if (subExpressionUnderConstruction.Order == -1)
55                 subExpressionUnderConstruction.Order =
56                                                 currentIndex;
57         }
58     }
59     totalSubexpressionsFound.Add(subExpressionUnderConstruction);
60     subExpressionStartIndex =
61         subExpressionUnderConstruction.Expression.Length;
62 }
```

9.36: MathExpression

```
1  public class MathExpression
2  {
3      private string _expression;
4      private int _order;
5
6      public MathExpression(string expression)
7      {
8          _expression = expression;
9          _order = -1;
10     }
```

```
11
12      public MathExpression(string expression, int order)
13      {
14          _expression = expression;
15          _order = order;
16      }
17
18      public string Expression
19      {
20          get { return _expression; }
21          set { _expression = value; }
22      }
23
24      public int Order
25      {
26          get { return _order; }
27          set { _order = value; }
28      }
29  }
```

He cambiado las cadenas por un objeto sencillo llamado `MathExpression` que guarda la posición en que aparece la subexpresión dentro de la expresión de entrada. Solamente me falta ordenar las subexpresiones para hacer pasar el test:

9.37: MathLexer

```
1   public List<MathExpression> GetExpressions(string expression)
2   {
3       List<MathExpression> totalExpressionsFound =
4           new List<MathExpression>();
5
6       int openedParenthesis = 0;
7       int startSearchingAt = 0;
8       getExpressions(expression, ref startSearchingAt,
9                      new MathExpression(String.Empty),
10                     totalExpressionsFound,
11                     ref openedParenthesis);
12      if (openedParenthesis != 0)
13          throw new
14              InvalidOperationException("Parenthesis␣do␣not␣match");
15      _fixer.FixExpressions(totalExpressionsFound);
16      bubbleSortExpressions(totalExpressionsFound);
17      return totalExpressionsFound;
18  }
19
20  private void bubbleSortExpressions(
21                      List<MathExpression> subExpressions)
22  {
23      for (int i = 0; i < subExpressions.Count; i++)
24      {
25          for (int j = 0; j < subExpressions.Count -1; j++)
26          {
27              MathExpression exp1 = subExpressions[j];
28              MathExpression exp2 = subExpressions[j + 1];
29              if (exp2.Order < exp1.Order)
30              {
31                  subExpressions[j] = exp2;
```

```
32                  subExpressions[j + 1] = exp1;
33              }
34
35          }
36      }
37  }
```

Una ordenación sencilla bastará por ahora. Luego hago un spike para ver si .Net me ordena la lista (lo anoto en la libreta). El último test ya pasa y todos los demás también, con la excepción de `ProcessAcceptanceExpressionWithParenthesis`, que todavía tiene un pequeño bug por falta de adaptar el código a la nueva clase `MathExpression`.

Corrijo:

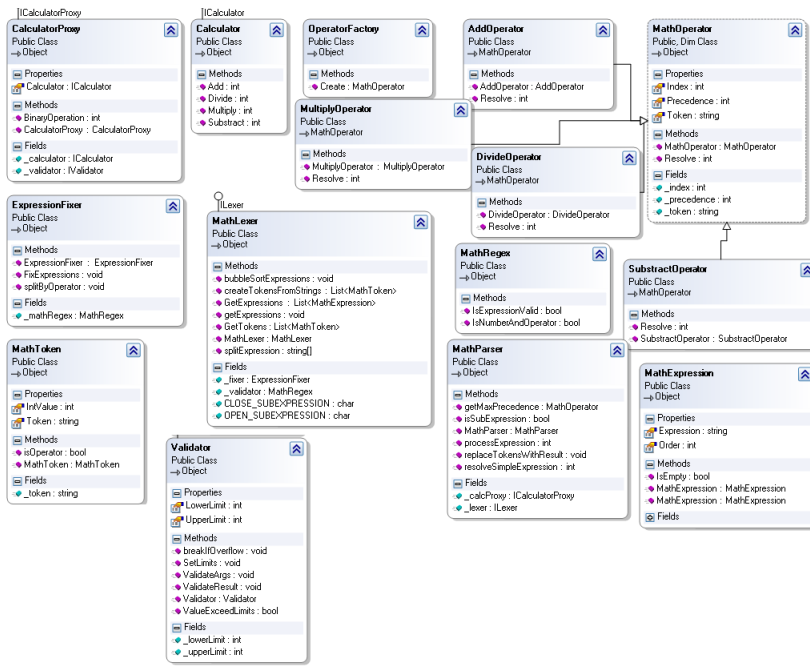
9.38: parte de MathParser

```
1   public int ProcessExpression(string expression)
2   {
3       List<MathExpression> subExpressions =
4           _lexer.GetExpressions(expression);
5       String flatExpression = String.Empty;
6       foreach (MathExpression subExp in subExpressions)
7       {
8           if (isSubExpression(subExp.Expression))
9               flatExpression +=
10                  resolveSimpleExpression(subExp.Expression);
11          else
12              flatExpression += "␣" + subExp.Expression + "␣";
13      }
14      return resolveSimpleExpression(flatExpression);
15  }
16
17  private bool isSubExpression(string exp)
18  {
19      Regex operatorRegex = new Regex(@"[+|\-|/|*]");
20      Regex numberRegex = new Regex(@"\d+");
21      if (numberRegex.IsMatch(exp) &&
22          operatorRegex.IsMatch(exp))
23      {
24          Console.Out.WriteLine("YES:" + exp);
25          return true;
26      }
27      return false;
28  }
29
30  private int resolveSimpleExpression(string expression)
31  {
32      List<MathToken> mathExp = _lexer.GetTokens(expression);
33      while (mathExp.Count > 1)
34      {
35          MathOperator op = GetMaxPrecedence(mathExp);
36
37          int firstNumber, secondNumber;
38          firstNumber = mathExp[op.Index - 1].IntValue;
39          secondNumber = mathExp[op.Index + 1].IntValue;
40          int result = op.Resolve(firstNumber,
41                                  secondNumber, _calcProxy);
```

```
42          replaceTokensWithResult(mathExp, op.Index, result);
43      }
44      return mathExp[0].IntValue;
45  }
```

¡Ya funciona! ¿Hay algo que refactorizar ahora? Sí, hay unos cuantos métodos que no están en el sitio más adecuado.

Para recordar al lector cómo está ahora mismo nuestro diagrama de clases emergente veamos la siguiente figura:




Movamos `isSubExpression` a nuestro `MathRegex`. Vaya, en ese caso necesito inyectar `MathRegex` en `MathParser` aunque ya estaba inyectado en `MathLexer`. Eso de que parser necesite de lexer y que ambos tengan inyectado al que entiende de expresiones regulares, pinta mal. ¿Para qué estamos usando lexer? Para extraer elementos de la expresión de entrada. ¿Y parser? Para encontrarles el sentido a los elementos. Entonces, ¿a quién corresponde la validación de la expresión que está haciendo ahora lexer? ¡A parser! De acuerdo, lo primero que muevo es la validación de expresiones:

```
1  public class MathParser
2  {
3      Lexer _lexer;
4      MathRegex _mathRegex;
```

```
5       CalculatorProxy _calcProxy;
6
7       public MathParser(Lexer lexer, CalculatorProxy calcProxy,
8           MathRegex mathRegex)
9       {
10          _lexer = lexer;
11          _mathRegex = mathRegex;
12          _calcProxy = calcProxy;
13      }
14
15      public MathOperator GetMaxPrecedence(List<MathToken> tokens)
16      {
17          int precedence = 0;
18          MathOperator maxPrecedenceOperator = null;
19
20          int index = -1;
21          foreach (MathToken token in tokens)
22          {
23              index++;
24              if (token.isOperator())
25              {
26                  MathOperator op = OperatorFactory.Create(token);
27                  if (op.Precedence >= precedence)
28                  {
29                      precedence = op.Precedence;
30                      maxPrecedenceOperator = op;
31                      maxPrecedenceOperator.Index = index;
32                  }
33              }
34          }
35          return maxPrecedenceOperator;
36      }
37
38      public int ProcessExpression(string expression)
39      {
40          List<MathExpression> subExpressions =
41              _lexer.GetExpressions(expression);
42          String flatExpression = String.Empty;
43          foreach (MathExpression subExp in subExpressions)
44          {
45              if (_mathRegex.IsSubExpression(subExp.Expression))
46                  flatExpression +=
47                        resolveSimpleExpression(subExp.Expression);
48              else
49                  flatExpression += "␣" + subExp.Expression + "␣";
50          }
51          return resolveSimpleExpression(flatExpression);
52      }
53
54      private int resolveSimpleExpression(string expression)
55      {
56          if (!_mathRegex.IsExpressionValid(expression))
57              throw new InvalidOperationException(expression);
58
59          List<MathToken> mathExp = _lexer.GetTokens(expression);
60          while (mathExp.Count > 1)
61          {
62              MathOperator op = GetMaxPrecedence(mathExp);
63
```

```
64              int firstNumber, secondNumber;
65              firstNumber = mathExp[op.Index - 1].IntValue;
66              secondNumber = mathExp[op.Index + 1].IntValue;
67              int result = op.Resolve(firstNumber,
68                                      secondNumber, _calcProxy);
69              replaceTokensWithResult(mathExp, op.Index, result);
70          }
71          return mathExp[0].IntValue;
72      }
73
74      private void replaceTokensWithResult(List<MathToken> tokens,
75          int indexOfOperator, int result)
76      {
77          tokens[indexOfOperator - 1] =
78                new MathToken(result.ToString());
79          tokens.RemoveAt(indexOfOperator);
80          tokens.RemoveAt(indexOfOperator);
81      }
82
83  }
84
85  public class MathLexer : Lexer
86  {
87      ExpressionFixer _fixer;
88      static char OPEN_SUBEXPRESSION = '(';
89      static char CLOSE_SUBEXPRESSION = ')';
90
91      public MathLexer(ExpressionFixer fixer)
92      {
93          _fixer = fixer;
94      }
95
96      public List<MathToken> GetTokens(string expression)
97      {
98          string[] items = splitExpression(expression);
99          return createTokensFromStrings(items);
100     }
101
102     private string[] splitExpression(string expression)
103     {
104         return expression.Split((new char[] { ' ', '\t' }),
105             StringSplitOptions.RemoveEmptyEntries);
106     }
107
108     private List<MathToken> createTokensFromStrings(string[] items)
109     {
110         List<MathToken> tokens = new List<MathToken>();
111         foreach (String item in items)
112         {
113             tokens.Add(new MathToken(item));
114         }
115         return tokens;
116     }
117
118     public List<MathExpression> GetExpressions(string expression)
119     {
120         List<MathExpression> totalExpressionsFound =
121             new List<MathExpression>();
122
```

```
123         int openedParenthesis = 0;
124         int startSearchingAt = 0;
125         getExpressions(expression, ref startSearchingAt,
126                        new MathExpression(String.Empty),
127                        totalExpressionsFound,
128                        ref openedParenthesis);
129         if (openedParenthesis != 0)
130             throw new
131                 InvalidOperationException("Parenthesis␣do␣not␣match");
132         _fixer.FixExpressions(totalExpressionsFound);
133         bubbleSortExpressions(totalExpressionsFound);
134         return totalExpressionsFound;
135     }
136 
137     private void bubbleSortExpressions(
138                  List<MathExpression> subExpressions)
139     {
140         for (int i = 0; i < subExpressions.Count; i++)
141         {
142             for (int j = 0; j < subExpressions.Count -1; j++)
143             {
144                 MathExpression exp1 = subExpressions[j];
145                 MathExpression exp2 = subExpressions[j + 1];
146                 if (exp2.Order < exp1.Order)
147                 {
148                     subExpressions[j] = exp2;
149                     subExpressions[j + 1] = exp1;
150                 }
151 
152             }
153         }
154     }
155 
156     private void getExpressions(string fullInputExpression,
157         ref int subExpressionStartIndex,
158         MathExpression subExpressionUnderConstruction,
159         List<MathExpression> totalSubexpressionsFound,
160         ref int openedParanthesis)
161     {
162         for (int currentIndex = subExpressionStartIndex;
163             currentIndex < fullInputExpression.Length;
164             currentIndex++)
165         {
166             char currentChar = fullInputExpression[currentIndex];
167             if (currentChar == OPEN_SUBEXPRESSION)
168             {
169                 openedParanthesis++;
170                 subExpressionStartIndex = currentIndex +1;
171                 getExpressions(fullInputExpression,
172                                ref subExpressionStartIndex,
173                                new MathExpression(String.Empty,
174                                    subExpressionStartIndex),
175                                totalSubexpressionsFound,
176                                ref openedParanthesis);
177                 currentIndex = subExpressionStartIndex;
178             }
179             else if (currentChar == CLOSE_SUBEXPRESSION)
180             {
181                 totalSubexpressionsFound.Add(
```

```
182                     subExpressionUnderConstruction);
183                 subExpressionStartIndex = currentIndex;
184                 openedParanthesis--;
185                 return;
186             }
187             else
188             {
189                 subExpressionUnderConstruction.Expression +=
190                     fullInputExpression[currentIndex].ToString();
191                 if (subExpressionUnderConstruction.Order == -1)
192                     subExpressionUnderConstruction.Order =
193                                                     currentIndex;
194             }
195         }
196         totalSubexpressionsFound.Add(subExpressionUnderConstruction);
197         subExpressionStartIndex =
198             subExpressionUnderConstruction.Expression.Length;
199     }
200 }
201
202 public class MathRegex
203 {
204     public bool IsExpressionValid(string expression)
205     {
206         Regex fullRegex = new Regex(
207               @"^-{0,1}\d+((\s+)[+|\-|/|*](\s+)-{0,1}\d+)+$");
208         Regex singleOperator = new Regex(@"^[+|\-|/|*]$");
209         Regex singleNumber = new Regex(@"^\d+$");
210         return (fullRegex.IsMatch(expression, 0) ||
211                 singleOperator.IsMatch(expression, 0) ||
212                 singleNumber.IsMatch(expression, 0));
213     }
214
215     public bool IsNumberAndOperator(string expression)
216     {
217         Regex startsWithOperator =
218             new Regex(@"^(\s*)([+|\-|/|*])(\s+)");
219         Regex endsWithOperator =
220             new Regex(@"(\s+)([+|\-|/|*])(\s*)$");
221
222         string exp = expression;
223         if (startsWithOperator.IsMatch(exp) ||
224              endsWithOperator.IsMatch(exp))
225             return true;
226         return false;
227     }
228
229     public bool IsSubExpression(string expression)
230     {
231         Regex operatorRegex = new Regex(@"[+|\-|/|*]");
232         Regex numberRegex = new Regex(@"\d+");
233         if (numberRegex.IsMatch(expression) &&
234             operatorRegex.IsMatch(expression))
235             return true;
236         return false;
237     }
238 }
```

He corregido los tests para que compilen después de los cambios y falla uno.

Aquel que le dice al lexer que ante una expresión inválida lance una excepción. Se debe a que ya lexer no valida expresiones sino parser. Entonces tenemos que mover el test a parser. Sin embargo, el método `resolveSimpleExpression` de parser es privado y no lo podemos testear directamente. Me empieza a parecer que parser tiene demasiadas responsabilidades. Está bien que entienda la expresión pero prefiero que sea otra clase la que resuelva las operaciones:

```
1  public class Resolver
2  {
3      MathRegex _mathRegex;
4      Lexer _lexer;
5      CalculatorProxy _calcProxy;
6
7      public Resolver(MathRegex mathRegex, Lexer lexer,
8          CalculatorProxy calcProxy)
9      {
10         _mathRegex = mathRegex;
11         _lexer = lexer;
12         _calcProxy = calcProxy;
13     }
14
15     public MathOperator GetMaxPrecedence(List<MathToken> tokens)
16     {
17         int precedence = 0;
18         MathOperator maxPrecedenceOperator = null;
19
20         int index = -1;
21         foreach (MathToken token in tokens)
22         {
23             index++;
24             if (token.isOperator())
25             {
26                 MathOperator op = OperatorFactory.Create(token);
27                 if (op.Precedence >= precedence)
28                 {
29                     precedence = op.Precedence;
30                     maxPrecedenceOperator = op;
31                     maxPrecedenceOperator.Index = index;
32                 }
33             }
34         }
35         return maxPrecedenceOperator;
36     }
37
38     public int ResolveSimpleExpression(string expression)
39     {
40         if (!_mathRegex.IsExpressionValid(expression))
41             throw new InvalidOperationException(expression);
42
43         List<MathToken> mathExp = _lexer.GetTokens(expression);
44         while (mathExp.Count > 1)
45         {
46             MathOperator op = GetMaxPrecedence(mathExp);
47
48             int firstNumber, secondNumber;
```

```
49              firstNumber = mathExp[op.Index - 1].IntValue;
50              secondNumber = mathExp[op.Index + 1].IntValue;
51              int result = op.Resolve(firstNumber,
52                                      secondNumber, _calcProxy);
53              replaceTokensWithResult(mathExp, op.Index, result);
54          }
55          return mathExp[0].IntValue;
56      }
57
58      private void replaceTokensWithResult(List<MathToken> tokens,
59          int indexOfOperator, int result)
60      {
61          tokens[indexOfOperator - 1] =
62                  new MathToken(result.ToString());
63          tokens.RemoveAt(indexOfOperator);
64          tokens.RemoveAt(indexOfOperator);
65      }
66  }
67
68  public class MathParser
69  {
70      Lexer _lexer;
71      MathRegex _mathRegex;
72      Resolver _resolver;
73
74      public MathParser(Lexer lexer, MathRegex mathRegex,
75          Resolver resolver)
76      {
77          _lexer = lexer;
78          _resolver = resolver;
79          _mathRegex = mathRegex;
80      }
81
82      public int ProcessExpression(string expression)
83      {
84          List<MathExpression> subExpressions =
85              _lexer.GetExpressions(expression);
86          String flatExpression = String.Empty;
87          foreach (MathExpression subExp in subExpressions)
88          {
89              if (_mathRegex.IsSubExpression(subExp.Expression))
90                  flatExpression +=
91                      _resolver.ResolveSimpleExpression(
92                              subExp.Expression);
93              else
94                  flatExpression += "␣" + subExp.Expression + "␣";
95          }
96          return _resolver.ResolveSimpleExpression(flatExpression);
97      }
98  }
```

Tuve que mover un test de sitio y modificar las llamadas a los constructores puesto que han cambiado. Ahora hay demasiadas inyecciones y dependencias. Demasiadas clases necesitan a `MathRegex`. Es hora de que se convierta en una serie de métodos estáticos[3] y deje de ser una dependencia inyectada.

[3]Usar métodos estáticos es casi siempre una mala idea, al final del capítulo se vuelve

9.39: Resolver

```
public int ResolveSimpleExpression(string expression)
{
    if (!MathRegex.IsExpressionValid(expression))
        throw new InvalidOperationException(expression);

    List<MathToken> mathExp = _lexer.GetTokens(expression);
    while (mathExp.Count > 1)
    {
        MathOperator op = GetMaxPrecedence(mathExp);

        int firstNumber, secondNumber;
        firstNumber = mathExp[op.Index - 1].IntValue;
        secondNumber = mathExp[op.Index + 1].IntValue;
        int result = op.Resolve(firstNumber,
                                secondNumber, _calcProxy);
        replaceTokensWithResult(mathExp, op.Index, result);
    }
    return mathExp[0].IntValue;
}
```

Por cierto, quería añadir un test para las subexpresiones:

9.40: MathRegexTests

```
[Test]
public void IsSubExpression()
{
    Assert.IsTrue(MathRegex.IsSubExpression("2␣+␣2"));
}
```

Luz verde. Este test me simplificará la búsqueda de posibles defectos.

Veamos el diagrama de clases resultante:

Los métodos que empiezan en mayúscula son públicos y los que empiezan en minúscula privados.

¿Queda algo más por mejorar? Si quisiéramos ser fieles a la teoría de compiladores entonces la función `GetExpressions` de lexer se movería a parser ya que en ella se hace la validación de paréntesis y se le busca sentido a las expresiones. Como no estoy diseñando un compilador ni siguiendo el método tradicional de construcción de una herramienta de análisis de código, no me importa que mis clases no coincidan exactamente con la teoría.

Por otra parte, habíamos hecho un método de ordenación que probablemente no sea necesario ya que .Net seguramente lo resuelve. En el momento de escribirlo me resultó más rápido utilizar el conocido método de la burbuja que hacer un spike. Ahora acabo de hacer el spike y veo que sólo con implementar la interfaz `IComparable` en `MathExpression` ya puedo utilizar el método `Sort` de las listas:

9.41: MathExpression

a hablar sobre esto

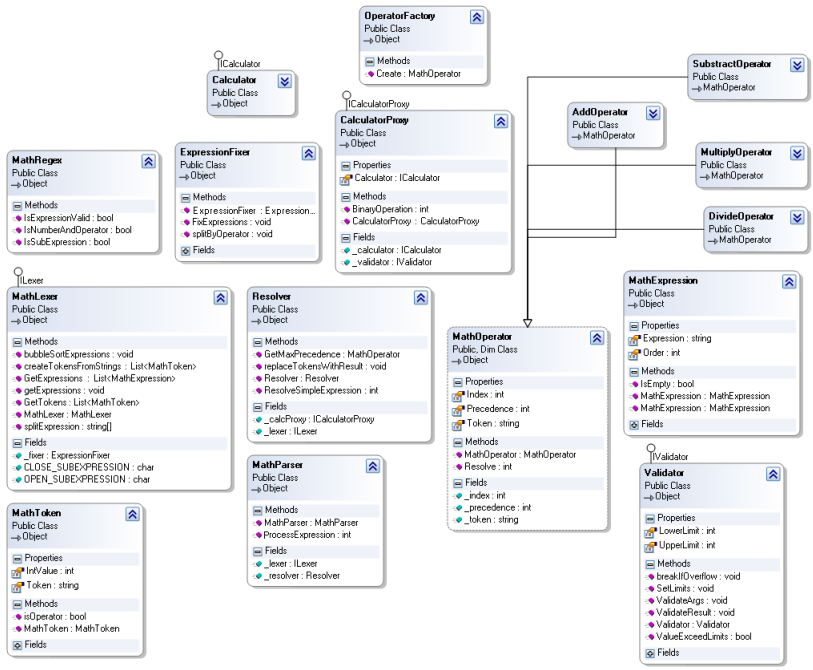



Figura 9.1: Diagrama de clases actual

```
1  public class MathExpression  : IComparable
2  {
3      private string _expression;
4      private int _order;
5
6      public MathExpression(string expression)
7      {
8          _expression = expression;
9          _order = -1;
10     }
11
12     public MathExpression(string expression, int order)
13     {
14         _expression = expression;
15         _order = order;
16     }
17
18     public string Expression
19     {
20         get { return _expression; }
21         set { _expression = value; }
22     }
23
24     public int Order
25     {
26         get { return _order; }
27         set { _order = value; }
```

```
28      }
29
30      public bool IsEmpty()
31      {
32          return _expression.Length == 0;
33      }
34
35      public int CompareTo(Object obj)
36      {
37         MathExpression exp = (MathExpression)obj;
38         return _order.CompareTo(exp.Order);
39      }
40  }
```

9.42: MathLexer

```
1   public List<MathExpression> GetExpressions(string expression)
2   {
3       List<MathExpression> totalExpressionsFound =
4           new List<MathExpression>();
5
6       int openedParenthesis = 0;
7       int startSearchingAt = 0;
8       getExpressions(expression, ref startSearchingAt,
9                      new MathExpression(String.Empty),
10                     totalExpressionsFound,
11                     ref openedParenthesis);
12      if (openedParenthesis != 0)
13          throw new
14             InvalidOperationException("Parenthesis␣do␣not␣match");
15      _fixer.FixExpressions(totalExpressionsFound);
16      totalExpressionsFound.Sort();  // ---> Ordenacion
17      return totalExpressionsFound;
18  }
```

El método de ordenación de la burbuja fue eliminado por completo.

Añadí el método `IsEmpty` a `MathExpression` para encapsular esta característica que se usaba aquí (línea 18):

9.43: ExpressionFixer

```
1   public void FixExpressions(List<MathExpression> expressions)
2   {
3       bool listHasChanged = true;
4       while (listHasChanged)
5       {
6           listHasChanged = false;
7           for (int i = 0; i < expressions.Count; i++)
8           {
9               expressions[i].Expression =
10                     expressions[i].Expression.Trim();
11              if (MathRegex.IsNumberAndOperator(
12                        expressions[i].Expression))
13              {
14                  splitByOperator(expressions,
15                                  expressions[i].Expression, i);
16                  listHasChanged = true;
```

```
17                  break;
18              }
19              if (expressions[i].IsEmpty())
20              {
21                  expressions.RemoveAt(i);
22                  listHasChanged = true;
23                  break;
24              }
25          }
26      }
27  }
```

Por último nos había quedado pendiente estudiar la relación entre las clases `MathToken` y `MathOperator`. La verdad es que el método `GetMaxPrecedence` de `Resolver` está haciendo demasiadas cosas. No sólo busca la precedencia sino que devuelve un objeto operador. Al leer el código me da la sensación de que ese metodo debería ser privado o mejor, estar en otra clase. Voy a refactorizar:

```
1   public class MathToken
2   {
3       protected int _precedence = 0;
4       protected string _token = String.Empty;
5       protected int _index = -1;
6
7       public MathToken(string token)
8       {
9           _token = token;
10      }
11
12      public MathToken(int precedence)
13      {
14          _precedence = precedence;
15      }
16
17      public int Index
18      {
19          get { return _index; }
20          set { _index = value; }
21      }
22
23      public string Token
24      {
25          get { return _token; }
26      }
27
28      public int Precedence
29      {
30          get { return _precedence; }
31      }
32
33      // Eliminado metodo IsOperator
34  }
35
36  public abstract class MathOperator : MathToken
37  {
38      public MathOperator(int precedence)
```

```
39          : base(precedence)
40      { }
41
42      public abstract int Resolve(int a, int b,
43                                  CalculatorProxy calcProxy);
44  }
45
46  public class MathNumber : MathToken
47  {
48      public MathNumber()
49          : base(0)
50      {}
51
52      public MathNumber(string token)
53              : base (token)
54      {}
55
56      public int IntValue
57      {
58          get { return Int32.Parse(_token); }
59      }
60
61      public static int GetTokenValue(string token)
62      {
63          return Int32.Parse(token);
64      }
65  }
66
67  public class Resolver
68  {
69      Lexer _lexer;
70      CalculatorProxy _calcProxy;
71
72      public Resolver(Lexer lexer,
73          CalculatorProxy calcProxy)
74      {
75          _lexer = lexer;
76          _calcProxy = calcProxy;
77      }
78
79      public MathToken GetMaxPrecedence(List<MathToken> tokens)
80      {
81          int precedence = 0;
82          MathToken maxPrecedenceToken = null;
83
84          int index = -1;
85          foreach (MathToken token in tokens)
86          {
87              index++;
88              if (token.Precedence >= precedence)
89              {
90                  precedence = token.Precedence;
91                  maxPrecedenceToken = token;
92                  maxPrecedenceToken.Index = index;
93              }
94          }
95          return maxPrecedenceToken;
96      }
97
```

```
98      public int ResolveSimpleExpression(string expression)
99      {
100         if (!MathRegex.IsExpressionValid(expression))
101             throw new InvalidOperationException(expression);
102
103         List<MathToken> mathExp = _lexer.GetTokens(expression);
104         while (mathExp.Count > 1)
105         {
106             MathToken token = GetMaxPrecedence(mathExp);
107             MathOperator op = OperatorFactory.Create(token);
108             int firstNumber, secondNumber;
109             firstNumber =
110                 MathNumber.GetTokenValue(
111                                 mathExp[op.Index - 1].Token);
112             secondNumber =
113                 MathNumber.GetTokenValue(
114                                 mathExp[op.Index + 1].Token);
115             int result = op.Resolve(firstNumber,
116                                     secondNumber, _calcProxy);
117             replaceTokensWithResult(mathExp, op.Index, result);
118         }
119         return MathNumber.GetTokenValue(mathExp[0].Token);
120     }
121
122     private void replaceTokensWithResult(List<MathToken> tokens,
123         int indexOfOperator, int result)
124     {
125         tokens[indexOfOperator - 1] =
126               new MathToken(result.ToString());
127         tokens.RemoveAt(indexOfOperator);
128         tokens.RemoveAt(indexOfOperator);
129     }
130 }
```

Al ejecutar los tests veo que todos los del parser fallan porque `OperatorFactory.Create` está intentando crear operadores a partir de tokens que no lo son. Lexer está construyendo todos los tokens con precedencia cero, por lo que no hay distinción:

9.44: MathLexer

```
1  private List<MathToken> createTokensFromStrings(string[] items)
2  {
3      List<MathToken> tokens = new List<MathToken>();
4      foreach (String item in items)
5      {
6          tokens.Add(new MathToken(item));
7      }
8      return tokens;
9  }
```

Si este método crease tokens de tipo operador o número tendríamos el problema resuelto. Voy a pedirle a `MathRegex` que me responda si los tokens son operadores o números:

9.45: MathRegexTests

```
[Test]
public void IsNumber()
{
    Assert.IsTrue(MathRegex.IsNumber("22"));
}
```

Ahora el SUT:

9.46: MathRegex

```
public static bool IsNumber(string token)
{
    Regex exactNumber = new Regex(@"^\d+$");
    return exactNumber.IsMatch(token, 0);
}
```

No necesito un test para comprobar que el token es un número si tiene espacios delante o detrás porque la función va a ser usada cuando ya se han filtrado los espacios.

¿Es operador?

```
[Test]
public void IsOperator()
{
    string operators = "*+/-";
    foreach(char op in operators)
        Assert.IsTrue(MathRegex.IsOperator(op.ToString()));
}
```

9.47: MathRegex

```
public static bool IsOperator(string token)
{
    Regex exactOperator = new Regex(@"^[*|\-|/|+]$");
    return exactOperator.IsMatch(token, 0);
}
```

¿Hay código duplicado en `MathRegex`? Sí, se repiten a menudo partes de expresiones regulares. Refactorizo:

```
public class MathRegex
{
    public static string operators = @"[*|\-|/|+]";

    public static bool IsExpressionValid(string expression)
    {
        Regex fullRegex = new Regex(@"^-{0,1}\d+((\s+)" +
            operators + @"(\s+)-{0,1}\d+)+$");
        return (fullRegex.IsMatch(expression, 0) ||
                IsNumber(expression) ||
                IsOperator(expression));
    }

    public static bool IsNumberAndOperator(string expression)
    {
        Regex startsWithOperator =
```

```
17              new Regex(@"^(\s*)(" + operators + @")(\s+)");
18          Regex endsWithOperator =
19              new Regex(@"(\s+)(" + operators + @")(\s*)$");
20
21          string exp = expression;
22          if (startsWithOperator.IsMatch(exp) ||
23               endsWithOperator.IsMatch(exp))
24              return true;
25          return false;
26      }
27
28      public static bool IsSubExpression(string expression)
29      {
30          Regex operatorRegex = new Regex(operators);
31          Regex numberRegex = new Regex(@"\d+");
32          if (numberRegex.IsMatch(expression) &&
33              operatorRegex.IsMatch(expression))
34              return true;
35          return false;
36      }
37
38      public static bool IsNumber(string token)
39      {
40          return IsExactMatch(token, @"\d+");
41      }
42
43      public static bool IsOperator(string token)
44      {
45          return IsExactMatch(token, operators);
46      }
47
48      public static bool IsExactMatch(string token, string regex)
49      {
50          Regex exactRegex = new Regex(@"^" + regex + "$");
51          return exactRegex.IsMatch(token, 0);
52      }
```

¿Por dónde íbamos? Ejecuto toda la batería de tests y veo que lexer está devolviendo tokens genéricos. Ahora que ya sabemos distinguir números de operadores podemos hacer que lexer construya los objetos adecuadamente:

9.48: LexerTests

```
1  [Test]
2  public void GetTokensRightSubclasses()
3  {
4      List<MathToken> tokens =
5          _lexer.GetTokens("2␣+␣2");
6      Assert.IsTrue(tokens[0] is MathNumber);
7      Assert.IsTrue(tokens[1] is MathOperator);
8  }
```

SUT:

9.49: MathLexer

```
1  private List<MathToken> createTokensFromStrings(string[] items)
2  {
```

```
3       List<MathToken> tokens = new List<MathToken>();
4       foreach (String item in items)
5       {
6           if (MathRegex.IsOperator(item))
7               tokens.Add(OperatorFactory.Create(item));
8           else
9               tokens.Add(new MathNumber(item));
10      }
11      return tokens;
12  }
```

El test pasa. ¿Pasan todos los tests?. No, aún faltaba adaptar `replaceTokensWithResult` de `Resolver` para que devuelva un `MathNumber` en lugar de un token genérico. Ya pasan todos los tests menos uno que discutiremos un poco más adelante[4]. El código queda así:

9.50: Resolver

```
1   public class Resolver
2   {
3       Lexer _lexer;
4       CalculatorProxy _calcProxy;
5
6       public Resolver(Lexer lexer,
7           CalculatorProxy calcProxy)
8       {
9           _lexer = lexer;
10          _calcProxy = calcProxy;
11      }
12
13      public MathToken GetMaxPrecedence(List<MathToken> tokens)
14      {
15          int precedence = 0;
16          MathToken maxPrecedenceToken = null;
17
18          int index = -1;
19          foreach (MathToken token in tokens)
20          {
21              index++;
22              if (token.Precedence >= precedence)
23              {
24                  precedence = token.Precedence;
25                  maxPrecedenceToken = token;
26                  maxPrecedenceToken.Index = index;
27              }
28          }
29          return maxPrecedenceToken;
30      }
31
32      public int ResolveSimpleExpression(string expression)
33      {
34          if (!MathRegex.IsExpressionValid(expression))
35              throw new InvalidOperationException(expression);
36
37          List<MathToken> mathExp = _lexer.GetTokens(expression);
```

[4]Lo pospongo por fines docentes

```
38          while (mathExp.Count > 1)
39          {
40              MathToken token = GetMaxPrecedence(mathExp);
41              MathOperator op = (MathOperator)token;
42              int firstNumber, secondNumber;
43              firstNumber =
44                   ((MathNumber)mathExp[op.Index - 1]).IntValue;
45              secondNumber =
46                   ((MathNumber)mathExp[op.Index + 1]).IntValue;
47              int result = op.Resolve(firstNumber,
48                                      secondNumber, _calcProxy);
49              replaceTokensWithResult(mathExp, op.Index, result);
50          }
51          return ((MathNumber)mathExp[0]).IntValue;
52      }
53
54      private void replaceTokensWithResult(List<MathToken> tokens,
55          int indexOfOperator, int result)
56      {
57          tokens[indexOfOperator - 1] = new MathNumber(result.ToString());
58          tokens.RemoveAt(indexOfOperator);
59          tokens.RemoveAt(indexOfOperator);
60      }
61  }
```

¿Movemos el cálculo de precedencia a una clase específica?:

```
1   public interface TokenPrecedence
2   {
3       MathToken GetMaxPrecedence(List<MathToken> tokens);
4   }
5
6   public class Precedence : TokenPrecedence
7   {
8       public MathToken GetMaxPrecedence(List<MathToken> tokens)
9       {
10          int precedence = 0;
11          MathToken maxPrecedenceToken = null;
12
13          int index = -1;
14          foreach (MathToken token in tokens)
15          {
16              index++;
17              if (token.Precedence >= precedence)
18              {
19                  precedence = token.Precedence;
20                  maxPrecedenceToken = token;
21                  maxPrecedenceToken.Index = index;
22              }
23          }
24          return maxPrecedenceToken;
25      }
26  }
27
28  public class Resolver
29  {
30      Lexer _lexer;
31      CalculatorProxy _calcProxy;
32      TokenPrecedence _precedence;
```

```
33
34      public Resolver(Lexer lexer,
35          CalculatorProxy calcProxy, TokenPrecedence precedence)
36      {
37          _lexer = lexer;
38          _calcProxy = calcProxy;
39          _precedence = precedence;
40      }
41
42      public int ResolveSimpleExpression(string expression)
43      {
44          if (!MathRegex.IsExpressionValid(expression))
45              throw new InvalidOperationException(expression);
46
47          List<MathToken> mathExp = _lexer.GetTokens(expression);
48          while (mathExp.Count > 1)
49          {
50              MathToken token = _precedence.GetMaxPrecedence(mathExp);
51          ...
52      ...
53      }
54  }
```

Hubo que rectificar un poquito los tests para que compilasen pero fue cuestión de un minuto.

Ahora ejecuto todos los tests y falla uno que había quedado por ahí pendiente: `ParserWorksWithLexer`[5]. El motivo es que se están haciendo varias llamadas al `lexer` y el mock estricto dice que sólo le habían avisado de una. Para corregir el test podría partir en dos, ya que hay dos llamadas. Una la simularía tipo stub y la otra tipo mock y viceversa. Pero es demasiado trabajo. A estas alturas el test no merece la pena porque toda la lógica de negocio de parser utiliza a lexer. Se ha convertido en una depedencia difícil de eludir, con lo cual, un test que comprueba que se usa, es poco importante. Simplemente lo elimino. Ya tenemos todos los tests en verde.

Repasemos el diagrama de clases:

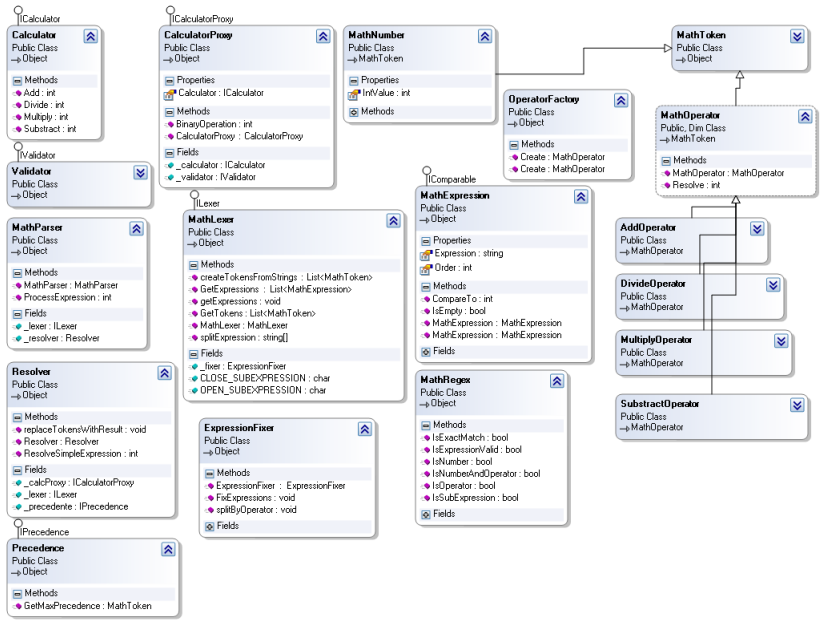
¿Qué sensación da el diagrama y el código? Personalmente me gusta como ha quedado el diagrama pero hay código que da un poco de mal olor. Se trata de `ResolveSimpleExpression` ya que está violando el Principio de Sustitución de Liskov[6]. Generalmente los *typecasts* son un indicador de algo que no se está haciendo del todo bien. Recordemos cómo quedó el método:

9.51: Resolver

```
1  public int ResolveSimpleExpression(string expression)
2  {
3      if (!MathRegex.IsExpressionValid(expression))
4          throw new InvalidOperationException(expression);
5
```

[5] página 148

[6] Ver Capítulo 7 en la página 111




```
6       List<MathToken> mathExp = _lexer.GetTokens(expression);
7       while (mathExp.Count > 1)
8       {
9           MathToken token = _precedence.GetMaxPrecedence(mathExp);
10          MathOperator op = (MathOperator)token;
11          int firstNumber, secondNumber;
12          firstNumber =
13                  ((MathNumber)mathExp[op.Index - 1]).IntValue;
14          secondNumber =
15                  ((MathNumber)mathExp[op.Index + 1]).IntValue;
16          int result = op.Resolve(firstNumber,
17                                  secondNumber, _calcProxy);
18          replaceTokensWithResult(mathExp, op.Index, result);
19      }
20      return ((MathNumber)mathExp[0]).IntValue;
21  }
```

Si refactorizo de manera que `Resolve` queda en la clase base (*Pull up method*), podría tratar a todos los tokens por igual:

9.52: Resolver

```
1   public int ResolveSimpleExpression(string expression)
2   {
3       if (!MathRegex.IsExpressionValid(expression))
4           throw new InvalidOperationException(expression);
5
6       List<MathToken> mathExp = _lexer.GetTokens(expression);
7       while (mathExp.Count > 1)
8       {
```

```
        MathOperator op = _precedence.GetMaxPrecedence(mathExp);
        int firstNumber = mathExp[op.Index - 1].Resolve();
        int secondNumber = mathExp[op.Index + 1].Resolve();
        op.CalcProxy = _calcProxy;
        int result = op.Resolve(firstNumber, secondNumber);
        replaceTokensWithResult(mathExp, op.Index, result);
    }
    return mathExp[0].Resolve();
}
```

```
public abstract class MathToken
{
    protected int _precedence = 0;
    protected string _token = String.Empty;
    protected int _index = -1;

    public MathToken(string token)
    {
        _token = token;
    }

    public MathToken(int precedence)
    {
        _precedence = precedence;
    }

    public int Index
    {...}

    public string Token
    {...}

    public int Precedence
    {...}

    public abstract int Resolve();
}

public class MathNumber : MathToken
{
    public MathNumber()
        : base(0)
    {}

    public MathNumber(string token)
        : base (token)
    {}

    public int IntValue
    {
        get { return Int32.Parse(_token); }
    }

    public override int Resolve()
    {
        return IntValue;
    }
}
```

```
49
50  public abstract class MathOperator : MathToken
51  {
52      protected int _firstNumber;
53      protected int _secondNumber;
54      protected CalculatorProxy _calcProxy;
55
56      public MathOperator(int precedence)
57          : base(precedence)
58      { }
59
60      public int FirstNumber
61      {
62          get { return _firstNumber; }
63          set { _firstNumber = value; }
64      }
65
66      public int SecondNumber
67      {
68          get { return _secondNumber; }
69          set { _secondNumber = value; }
70      }
71
72      public CalculatorProxy CalcProxy
73      {
74          get { return _calcProxy; }
75          set { _calcProxy = value; }
76      }
77
78      public override int Resolve()
79      {
80          return Resolve(_firstNumber, _secondNumber);
81      }
82
83      public abstract int Resolve(int a, int b);
84  }
85
86  public interface TokenPrecedence
87  {
88      MathOperator GetMaxPrecedence(List<MathToken> tokens);
89  }
90
91  public class Precedence : TokenPrecedence
92  {
93      public MathOperator GetMaxPrecedence(List<MathToken> tokens)
94      {
95          ...
96          return (MathOperator)maxPrecedenceToken;
97      }
98  }
```

Nótese que `CalProxy` ha sido movido dentro del operador. Al convertir `MathToken` en abstracta nos aseguramos que nadie crea instancias de esa clase. Al fin y al cabo queremos trabajar con instancias concretas.

No hemos respetado el principio de Liskov del todo. La clase que gestiona precedencia sigue necesitando una conversión explícita. Vamos a darle una

vuelta de tuerca más:

9.53: Resolver

```
public int ResolveSimpleExpression(string expression)
{
    if (!MathRegex.IsExpressionValid(expression))
        throw new InvalidOperationException(expression);

    List<MathToken> mathExp = _lexer.GetTokens(expression);
    while (mathExp.Count > 1)
    {
        MathToken op = _precedence.GetMaxPrecedence(mathExp);
        op.PreviousToken = mathExp[op.Index - 1];
        op.NextToken = mathExp[op.Index + 1];
        int result = op.Resolve();
        replaceTokensWithResult(mathExp, op.Index, result);
    }
    return mathExp[0].Resolve();
}
```

```
public abstract class MathToken
{
    protected int _precedence = 0;
    protected string _token = String.Empty;
    protected int _index = -1;
    protected MathToken _previousToken, _nextToken;


    public MathToken(string token)
    {
        _token = token;
    }

    public MathToken(int precedence)
    {
        _precedence = precedence;
    }

    public MathToken PreviousToken
    {
        get { return _previousToken; }
        set { _previousToken = value; }
    }

    public MathToken NextToken
    {
        get { return _nextToken; }
        set { _nextToken = value; }
    }

    public int Index
    {
        get { return _index; }
        set { _index = value; }
    }

    public string Token
```

```
38      {
39          get { return _token; }
40      }
41
42      public int Precedence
43      {
44          get { return _precedence; }
45      }
46
47      public abstract int Resolve();
48  }
49
50  public abstract class MathOperator : MathToken
51  {
52      protected CalculatorProxy _calcProxy;
53
54      public MathOperator(int precedence)
55          : base(precedence)
56      {
57          _calcProxy = new CalcProxy(
58              new Validator(-100, 100), new Calculator());
59      }
60
61      public CalculatorProxy CalcProxy
62      {
63          get { return _calcProxy; }
64          set { _calcProxy = value; }
65      }
66
67      public override int Resolve()
68      {
69          return Resolve(_previousToken.Resolve(), _nextToken.Resolve());
70      }
71
72      public abstract int Resolve(int a, int b);
73  }
```

`Resolver` ya no necesita ningún `CalculatorProxy`. El constructor de `MathOperator` crea el proxy. Llegado este punto no dejaría que el constructor crease la instancia del colaborador porque sino, perdemos la inversión del control. Es el momento perfecto para introducir un contenedor de inyección de dependencias tipo `Castle.Windsor`. En ese caso dentro del constructor se haría una llamada al contenedor para pedirle una instancia del colaborador de manera que éste la inyecte.

9.54: Supuesto MathOperator

```
1  public MathOperator(int precedence)
2          : base(precedence)
3  {
4      WindsorContainer container =
5                new WindsorContainer(new XmlInterpreter());
6      _calProxy = container.Resolve<CalculatorProxy>(
7                                          "simpleProxy");
8  }
```

Los contenedores de inyección de dependencias se configuran a través de un fichero o bien mediante código, de tal manera que si existen varias clases que implementan una interfaz, es decir, varias candidatas a ser inyectadas como dependencias, sólo tenemos que modificar la configuración para reemplazar una dependencia por otra, no tenemos que tocar el código fuente. Por tanto el método `Resolve` del contenedor buscará qué clase concreta hemos configurado para ser instanciada cuando usamos como parámetro la interfaz `CalculatorProxy`.

Hemos podido eliminar la conversión de tipos en la función que obtiene la máxima precedencia y ahora el código tiene mucha mejor pinta.

Lo último que me gustaría refactorizar es la función recursiva `getExpressions` de `MathLexer`. Tiene demasiados parámetros. Una función no debería tener más de uno o dos parámetros, a lo sumo tres. Voy a utilizar un objeto para agrupar datos y funcionalidad:

9.55: MathLexer

```
1  public List<MathExpression> GetExpressions(string expression)
2  {
3      int openedParenthesis = 0;
4      ExpressionBuilder expBuilder =
5          ExpressionBuilder.Create();
6      expBuilder.InputText = expression;
7      getExpressions(expBuilder, ref openedParenthesis);
8      if (openedParenthesis != 0)
9          throw new
10             InvalidOperationException("Parenthesis␣do␣not␣match");
11     _fixer.FixExpressions(expBuilder.AllExpressions);
12     expBuilder.AllExpressions.Sort();
13     return expBuilder.AllExpressions;
14 }
15
16 private void getExpressions(ExpressionBuilder expBuilder,
17     ref int openedParanthesis)
18 {
19     while(expBuilder.ThereAreMoreChars())
20     {
21         char currentChar = expBuilder.GetCurrentChar();
22         if (currentChar == OPEN_SUBEXPRESSION)
23         {
24             openedParanthesis++;
25             getExpressions(expBuilder.ProcessNewSubExpression(),
26                            ref openedParanthesis);
27         }
28         else if (currentChar == CLOSE_SUBEXPRESSION)
29         {
30             expBuilder.SubExpressionEndFound();
31             openedParanthesis--;
32             return;
33         }
34         else
35             expBuilder.AddSubExpressionChar();
36     }
37     expBuilder.SubExpressionEndFound();
```

```
}
```

```
public class ExpressionBuilder
{
    private static string _inputText;
    private static int _currentIndex = 0;
    private static List<MathExpression> _allExpressions;
    private MathExpression _subExpression;

    private ExpressionBuilder() { }

    public static ExpressionBuilder Create()
    {
        ExpressionBuilder builder = new ExpressionBuilder();
        builder.AllExpressions = new List<MathExpression>();
        builder.CurrentIndex = 0;
        builder.InputText = String.Empty;
        builder.SubExpression = new MathExpression(String.Empty);
        return builder;
    }

    public ExpressionBuilder ProcessNewSubExpression()
    {
        ExpressionBuilder builder = new ExpressionBuilder();
        builder.InputText = _inputText;
        builder.SubExpression = new MathExpression(String.Empty);
        updateIndex();
        return builder;
    }

    public bool ThereAreMoreChars()
    {
        return _currentIndex < MaxLength;
    }

    public void AddSubExpressionChar()
    {
        _subExpression.Expression +=
                    _inputText[_currentIndex].ToString();
        if (_subExpression.Order == -1)
            _subExpression.Order = _currentIndex;
        updateIndex();
    }

    public void SubExpressionEndFound()
    {
        _allExpressions.Add(_subExpression);
        updateIndex();
    }

    public string InputText
    {
        get { return _inputText; }
        set { _inputText = value; }
    }

    public char GetCurrentChar()
    {
```

```
57          return _inputText[_currentIndex];
58      }
59
60      public int MaxLength
61      {
62          get { return _inputText.Length; }
63      }
64
65      public int CurrentIndex
66      {
67          get { return _currentIndex; }
68          set { _currentIndex = value; }
69      }
70
71      public List<MathExpression> AllExpressions
72      {
73          get { return _allExpressions; }
74          set { _allExpressions = value; }
75      }
76
77      public MathExpression SubExpression
78      {
79          get { return _subExpression; }
80          set { _subExpression = value; }
81      }
82
83      private void updateIndex()
84      {
85          _currentIndex++;
86      }
87  }
```

Refactorización hecha, código más claro y casi todos los tests pasando. El test `ParserWorksWithCalcProxy` se ha roto ya que el parser no recibe ningún proxy. Es momento de eliminarlo. Nos ha prestado buen servicio hasta aquí pero ya terminó su misión. Tenemos toda la batería de test en verde y la refactorización terminada por el momento.

Mientras resolvía los últimos tests me vino a la mente un ejemplo más rebuscado que apunté en la libreta y que ahora expreso de forma ejecutable:

9.56: LexerTests

```
1  [Test]
2  public void GetComplexNestedExpressions()
3  {
4      List<MathExpression> expressions =
5          _lexer.GetExpressions("((2␣+␣2)␣+␣1)␣␣*␣(3␣+␣1)");
6      failIfOtherSubExpressionThan(
7          expressions, "3␣+␣1", "2␣+␣2", "+", "*", "1");
8  }
```

Luz verde sin tocar el SUT. ¡Estupendo!. ¿Será parser capaz de operar esa lista de subexpresiones?

```
1  [Test]
2  public void ProcessComplexNestedExpressions()
```

```
3 {
4     Assert.AreEqual(20,
5         _parser.ProcessExpression(
6         "((2␣+␣2)␣+␣1)␣␣*␣(3␣+␣1)"));
7 }
```

Pues no, no puede. Se nos está perdiendo la precedencia de las operaciones con paréntesis anidados. `MathParser` tiene que colaborar más con `MathLexer` y tal vez ir resolviendo subexpresiones al tiempo que se van encontrando. Hay que afinar más `MathParser`. Sin embargo, no lo voy a hacer yo. Lo dejo abierto como ejercicio para el lector, junto con los demás ejemplos pendientes de nuestra libreta:

```
# Aceptación - "((2 + 2) + 1) * (3 + 1)", devuelve 20
# Aceptación - "3 / 2", produce ERROR
# Aceptación - "2 + -2", devuelve 0
```

Hemos llegado al final del segundo capítulo práctico. A lo largo de este capítulo hemos resuelto un problema clásico de la teoría de compiladores pero de manera emergente, haciendo TDD. Ciertamente, para un problema que lleva tantos años resuelto como es el de los analizadores de código, hubiera buscado algún libro de *Alfred V. Aho* o algún colega suyo y habría utilizado directamente los algoritmos que ya están más que inventados. Pero el fin de este libro es docente; es mostrar con ejemplos como se hace un desarrollo dirigido por ejemplos, valga la redundancia.

Como habrá observado el código va mutando, va adaptándose a los requisitos de una manera orgánica, incremental. A menudo en mis cursos de TDD, cuando llegamos al final de la implementación del ejemplo alguien dice... *"si hubiera sabido todos esos detalles lo hubiese implementado de otra manera"*. A esa persona le parece que el diagrama de clases que resulta no se corresponde con el modelado mental de conceptos que tiene en la cabeza y le cuesta admitir que las clases no necesariamente tienen una correspondencia con esos conceptos que manejamos los humanos. Para esa frase hay dos respuestas. La primera es que si conociésemos absolutamente el 100 % de los detalles de un programa y además fuesen inamovibles, entonces haríamos programas igual que hacemos edificios: es una situación utópica. La segunda es que aun conociendo todos los ejemplos posibles, nadie tiene capacidad para escribir del tirón el código perfecto que los resuelve todos. Es más humano y por tanto más productivo progresar paso a paso.

También me comentan a veces que se modifica mucho código con las refactorizaciones y que eso da sensación de pérdida de tiempo. ¿Hacer software de calidad es perder el tiempo? Realmente no lleva tanto tiempo refactorizar; típicamente es cuestión de minutos. Luego se ganan horas, días y hasta meses

cuando hay que mantener el software. Al disponer de una batería de tests de tanta calidad como la que surge de hacer TDD bien, modificar el código es una actividad con la que se disfruta. Todo lo contrario a la estresante tarea de hacer modificaciones sobre código sin tests.

Lo ideal es refactorizar un poquito a cada vez, respetar con conciencia el último paso del algoritmo TDD que dice que debemos eliminar el código duplicado y si es posible, mejorar el código también.

El código que tenemos hasta ahora no es perfecto y mi objetivo no es que lo sea. A pesar de que no tiene mal aspecto es posible hacer mejoras todavía. Lo que pretendo mostrar es que el diseño emergente genera código que está preparado para el cambio. Código fácil de mantener, con una cierta calidad. No es decisivo que en la etapa de refactorización lleguemos a un código impoluto, nadie tiene tiempo de entretenerse eternamente a retocar un bloque de código pero desde luego hay que cumplir unos mínimos. Para mí esos mínimos son los principios S.O.L.I.D, aunque no siempre es necesario que se cumplan completamente cada vez que se refactoriza: si la experiencia nos avisa de que tendremos que hacer cambios en el futuro cercano (en los próximos tests) que nos darán oportunidad de seguir evolucionando el código, podemos esperar a terminar de refactorizar. La artesanía sigue jugando un papel muy importante en esta técnica de desarrollo.

Debo reconocer que el hecho de que `MathRegex` sea un conjunto de métodos tipo `static` es una mala solución. Lo mejor sería dividir en clases cuando estén claras las distintas responsabilidades que tiene y así arreglar el problema de que tantas otras clases dependiesen de la misma entidad. Generalmente los métodos estáticos rompen con la orientación a objetos y nos hacen volver a la programación funcional. Se deben de evitar a toda costa porque se nos hace imposible probar cuestiones como la interacción entre objetos, puesto que un conjunto de métodos estáticos no constituyen un objeto. Sin embargo en el punto en que estamos no he visto con claridad cómo refactorizar y he decidido preservar esa solución, aunque en la libreta me apunto que debo cambiarla tan pronto como sepa hacerlo.

En el próximo capítulo afrontaremos ejemplos que nos exigirán dobles de prueba y terminaremos integrando nuestro código con el resto del sistema para visitar todos los escenarios que se presentan en el desarrollo de aplicaciones "empresariales".

Capítulo 10

# Fin del proyecto - Test de Integración

Hasta ahora nos hemos movido por la lógica de negocio mediante ejemplos en forma de test unitario. Hemos respetado las propiedades de rápidez, inocuidad, claridad y atomicidad, aunque la granularidad de algunos tests no fuese mínima en algún caso. En todo momento hemos trabajado con variables en memoria, no ha habido acceso al sistema de ficheros ni al sistema de gestión de base de datos. Hemos trabajado en el escenario ideal para comprender de qué trata el diseño emergente. Si ha seguido los capítulos anteriores escribiendo código frente a la pantalla, a la par que leía el libro, ya habrá comprendido de qué va TDD.

La utilidad mas común de una aplicación es la manipulación de datos, algo que todavía no sabemos hacer de manera emergente. Por ello vamos a dar un giro a la implementación de nuestro problema tratando criterios de aceptación que requieren que nuestros objetos se integren con otros objetos externos, que sí tienen acceso a una base de datos.

Atendiendo a los requisitos que se exponen al comienzo del capítulo 8, no necesitamos acceder a ningún sistema de ficheros para desarrollar nuestra solución sino sólo a una base de datos. No obstante, para mostrar en detalle como podemos hacer integración a la TDD, imaginemos que el usuario nos ha pedido que la lista de alumnos del juego se guarde en un fichero en disco. Más adelante trabajaremos con la base de datos. Nuestra libreta contiene lo siguiente:

```
# Aceptación:
  Ruta y nombre del fichero de datos son especificados por el usuario
  Fichero datos = C:datos.txt
  Fichero datos = /home/jmb/datos.txt
# Aceptación: Un usuario tiene nick y clave de acceso:
  nick: adri, clave: pantera
# Aceptación: Crear, modificar y borrar usuarios
# Aceptación: La clave de acceso está encriptada
  clave plana: pantera, hash MD5: 2d58b0ac72f929ca9ad3238ade9eab69
# Aceptación: Si los usuarios que existen son Fran y Yeray
  El listado de usuarios del sistema devuelve [(0, Fran), (1, Yeray)]
▪ Si usuarios del sistema son Esteban y Eladio, fichero contendrá:
  # Begin
  0:Esteban:2d58b0ac72f929ca9ad3238ade9eab69
  1:Eladio:58e53d1324eef6265fdb97b08ed9aadf
  # End
▪ Si añadimos el usuario Alberto al fichero anterior:
  # Begin
  0:Esteban:2d58b0ac72f929ca9ad3238ade9eab69
  1:Eladio:58e53d1324eef6265fdb97b08ed9aadf
  2:Alberto:5ad4a96c5dc0eae3d613e507f3c9ab01
  # End
▪ Las expresiones introducidas por el usuario 1 se guardan junto con
  su resultado
  "2 + 2", 4, Usuario(1, Alberto)
  "2 + 1", 3, Usuario(1, Alberto)
# Aceptación:
  Obtener todas las expresiones introducidas por el usuario Alberto
  ([Usuario(1, Alberto), "2 + 2", 4], [Usuario(1, Alberto), "2 + 1", 3])
```

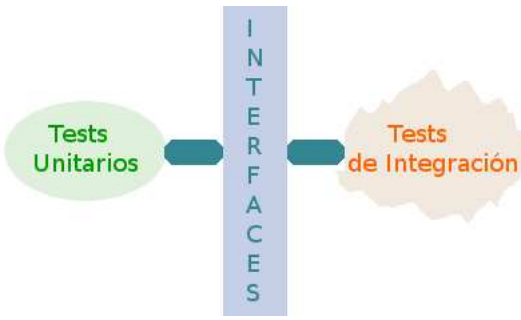
El criterio de aceptación de que los usuarios se pueden modificar y borrar no está bien escrito, no llega a haber ejemplos. Sabemos que al crearlo sus atributos se guardan en el fichero de datos porque se tiene que listar el nick y el identificador de usuario pero de momento, el cliente no nos ha contado cómo se modifican y se borran usuarios. Lo dejaremos así para empezar cuanto antes con la implementación. Como podrá observar, en la libreta vuelve a haber tests de aceptación y tests de desarrollo. Los tests de desarrollo son una propuesta que responde a cómo queremos implementar los tests de aceptación, los cuales no llegan a adentrarse en cómo sino en qué.

Empieza a oler a integración porque la definición de los ejemplos se ayuda

de muchos datos de contexto (*fixtures*), en este caso de un fichero y también de una base de datos. ¿Por dónde empezamos a trabajar? Lo primero es delimitar la frontera entre nuestros objetos y los objetos de terceros.

## 10.1. La frontera entre tests unitarios y tests de integración

Como hemos visto, trabajar en el ámbito de los tests unitarios es rápido y productivo. Cuanta más lógica de negocio podamos mantener controlada bajo nuestra batería de tests unitarios, más fácil será detectar y corregir problemas, además de ampliar funcionalidad. Sin embargo, llega un punto en que inevitablemente tenemos que romper las reglas de los tests unitarios y modificar el estado del sistema mediante una escritura en disco. La mejor forma de hacerlo es estableciendo una frontera a modo de contrato entre las dos partes: la que se limita a trabajar con datos en memoria y la que modifica el estado del sistema. Gráficamente pienso en ello como la parte a la izquierda y la parte a la derecha.




Tal frontera se delimita mediante interfaces en lenguajes como Java y C#. Una interfaz es un contrato al fin y al cabo. En el caso de lenguajes interpretados no se necesitan interfaces, simplemente habrá clases.

Para que la parte izquierda contenga el mayor volumen de lógica de negocio posible, tenemos que pensar que la parte derecha ya está implementada. Tenemos que diseñar la frontera y pensar que ya hay alguna clase que implementa ese contrato y que es capaz de recibir y enviar datos al sistema. Es decir, nos mantenemos en el ámbito de los tests unitarios considerando que las clases que modifican el estado del sistema ya existen, utilizando mocks y stubs para hablar con ellas, sin que realmente toquen el sistema.

Una vez llegamos ahí, vamos a por los tests de integración, que probarán que las clases efectivamente se integran bien con el sistema externo, escribiendo y leyendo datos. Para ejecutar los tests de integración necesitaremos un entorno de preproducción que se pueda montar y desmontar antes y después de la ejecución de la batería de tests de integración, ya que dichos tests podrían dejar el sistema en un estado inconsistente.

Aunque la plataforma sobre la que trabajemos nos ofrezca ya interfaces que nos puedan parecer válidas para establecer la frontera, se recomienda definir nuestra propia interfaz independiente. Luego, la clase que implemente la interfaz frontera podrá hacer uso de todas las herramientas que la plataforma le ofrezca.

Vamos a verlo con ejemplos mediante la implementación de nuestra solución. Hay un test de aceptación que dice que crearemos el fichero de datos de usuario en la ruta especificada por alguien. En los tests de desarrollo se aprecia que el fichero de usuarios contiene una línea por cada usuario, con unos delimitadores de comienzo y fin de fichero y unos delimitadores de campos para los atributos de los usuarios. Por un lado veo la necesidad de crear un fichero y por otra la de leerlo e interpretarlo. Empiezo a resolver el problema por la parte izquierda:

```
1  [TestFixture]
2  public class UserManagementTests
3  {
4      [Test]
5      public void ConfigUsersFile()
6      {
7          string filePath = "/home/carlosble/data.txt";
8          FileHandler handlerMock =
9              MockRepository.GenerateMock<FileHandler>();
10         handlerMock.Expect(
11                 x => x.CreateFile(filePath)).Return(
12                                     new UserFile());
13
14         UsersStorageManager manager =
15                 new UsersStorageManager(handlerMock);
16         manager.SetUsersFile(filePath);
17
18         handlerMock.VerifyAllExpectations();
19     }
20 }
```

Mi planteamiento ha sido el siguiente: Quiero una clase para gestión de datos de usuarios que tenga un método para definir cuál es el fichero donde se guardarán los datos. Existirá una interfaz `FileHandler` con un método para crear el fichero, que devolverá un objeto de tipo `UserFile`. Estoy obligando a que mi SUT colabore con el manejador de ficheros. Nada más terminar de escribir el test me he preguntado qué pasaría si hay algún problema con la creación del fichero (ej: insuficiencia de permisos) y lo he apuntado en la

libreta.

El subconjunto de próximas tareas de la libreta es el siguiente:

- *Crear un fichero en la ruta especificada*
- *Resolver casos en que no se puede crear*
  - *Permisos insuficientes*
  - *Ruta no existe*
- *Leer un fichero de texto línea a línea*

Voy a ocuparme de que el test pase lo antes posible, como siempre:

```
1  public class UsersStorageManager
2  {
3      FileHandler _handler;
4
5      public UsersStorageManager(FileHandler handler)
6      {
7          _handler = handler;
8      }
9
10     public void SetUsersFile(string path)
11     {
12         _handler.CreateFile(path);
13     }
14 }
```

Luz verde.

Acabamos de definir una frontera mediante la interfaz `FileHandler`. Ahora podemos ir desarrollando todos los casos que nos vayan haciendo falta, como por ejemplo el de que no haya permisos para crear el fichero. Simularemos tal caso diciéndole a Rhino.Mocks que lance una excepción por falta de permisos y nos ocuparemos de que nuestra clase gestora responda adecuadamente. El juego consiste en utilizar los dobles[1] para que produzcan un determinado comportamiento y nosotros podamos programar nuestra lógica de negocio acorde a ese comportamiento.

El beneficio que obtenemos es doblemente bueno. Por un lado estamos trabajando con tests unitarios y por otro, resulta que ante posibles defectos en las clases que implementen `FileHandler`, nuestro test no se romperá. El SUT está bien aislado.

¿Cómo implementamos los tests de integración para la clase que cumple con el contrato `FileHandler`? Para esta funcionalidad me parece bien seguir usando NUnit. Lo primero que se me viene a la cabeza antes de empezar es que tendremos que cuidarnos de saber si estamos en un entorno MS Windows, POSIX o algún otro para evitar problemas con las rutas. Lo apunto en

[1] Mocks, Stubs, etc

la libreta. Voy a crear un nuevo proyecto dentro de la solución para los tests de integración (otra DLL) de modo que no se ejecuten cada vez que lance los tests unitarios. La libreta dice:

- *Crear un fichero en la ruta especificada*
- *Resolver casos en que no se puede crear*

  *Permisos insuficientes*

  *Ruta no existe*
- *Leer un fichero de texto línea a línea*

  Integración:
- *Crear el fichero en Windows*
- *Crear el fichero en Ubuntu*
- *Crear el fichero en MacOS*

Voy con el primero de ellos:

```
1  namespace IntegrationTests
2  {
3      [TestFixture]
4      public class FileHandlerTests
5      {
6          [Test]
7          public void CreateFileWithWindowsPath()
8          {
9              string path = @"c:\data.txt";
10             UserFileHandler handler =
11                 new UserFileHandler();
12
13             handler.CreateFile(path);
14
15             if (!File.Exists(path))
16             {
17                 Assert.Fail("File was not created");
18             }
19         }
20     }
21 }
```

He creado una clase `UserFileHandler` que implementa la interfaz `FileHandler`. Llamo a su método de crear fichero y luego compruebo que el fichero existe mediante la API que me ofrece .Net. Hay que cuidarse de no hacer la comprobación mediante el mismo SUT cuando se trata de integración. Es decir, supongamos que el SUT tiene un metodo tipo `IsFileOpened`. Sería absurdo invocarlo para hacer la comprobación de que el fichero se ha creado ya que eso no garantiza que efectivamente el sistema de ficheros se ha modificado. Por este motivo a veces la etapa de afirmación de un test de integración se puede llegar a complicar bastante. Vamos a hacer que el primer test pase:

```
public class UserFileHandler : FileHandler
{
    public DataFile CreateFile(string path)
    {
        File.Create(path);
        return null;
    }
}
```

Luz verde. Me llama la atención que no estamos haciendo nada con el fichero. No estamos creando ningún `DataFile`. Voy a aumentar un poco más la libreta:

- *Crear un fichero en la ruta especificada*
- *El que implemente DataFile contiene un FileStream con acceso al fichero creado*
- *Resolver casos en que no se puede crear*
  *Permisos insuficientes*
  *Ruta no existe*
- *Leer un fichero de texto línea a línea*
  Integración:
- *Crear el fichero en Windows*
- *Crear el fichero en Ubuntu*
- *Crear el fichero en MacOS*

El método `File.Create` devuelve un `FileStream`. Todo eso es parte de la API de .Net. `FileStream` tiene un campo `Handle` que es una referencia al fichero en disco. En base a esto voy a modificar mi test de integración:

```
[Test]
public void CreateFileWithWindowsPath()
{
    string path = @"c:\data.txt";
    UserFileHandler handler =
        new UserFileHandler();

    DataFile dataFile = handler.CreateFile(path);
    if (!File.Exists(path))
    {
        Assert.Fail("File was not created");
    }
    Assert.IsNotNull(dataFile.Stream);
}
```

Vamos a buscar la luz verde:

```
public class UserFileHandler : FileHandler<UserFile>
{
    public UserFile CreateFile(string path)
```

```
    {
        FileStream stream = File.Create(path);
        UserFile userFile = new UserFile();
        userFile.Stream = stream;
        return userFile;
    }
}
```

El test ya pasa. He decidido introducir genéricos en el diseño lo cual me ha llevado a hacer algunas modificaciones:

```
public interface FileHandler<T>
    where T: DataFile
{
    T CreateFile(string path);
}

public interface DataFile
{
    FileStream Stream { get; set; }
}

public class UserFile: DataFile
{
    FileStream _stream = null;

    public FileStream Stream
    {
        get { return _stream; }
        set { _stream = value; }
    }
}
```

```
namespace UnitTests
{
    [TestFixture]
    public class UserManagementTests
    {
        [Test]
        public void ConfigUsersFile()
        {
            string filePath = "/home/carlosble/data.txt";
            FileHandler<UserFile> handlerMock =
                MockRepository.GenerateMock<FileHandler<UserFile>>();
            handlerMock.Expect(
                x => x.CreateFile(filePath)).Return(new UserFile());

            UsersStorageManager manager =
                new UsersStorageManager(handlerMock);
            manager.SetUsersFile(filePath);

            handlerMock.VerifyAllExpectations();
        }
    }
}
```

```
namespace IntegrationTests
```

```
2  {
3      [TestFixture]
4      public class FileHandlerTests
5      {
6          [Test]
7          public void CreateFileWithWindowsPath()
8          {
9              string path = @"c:\data.txt";
10             UserFileHandler handler =
11                 new UserFileHandler();
12
13             DataFile dataFile = handler.CreateFile(path);
14             if (!File.Exists(path))
15             {
16                 Assert.Fail("File was not created");
17             }
18             Assert.IsNotNull(dataFile.Stream);
19         }
20     }
21 }
```

Los genéricos funcionan prácticamente igual en C# que en Java, aunque su sintaxis me parece más clara en C#. No es importante que sea experto en genéricos si comprende el código que he escrito. Si quiere leer más sobre genéricos en .Net, hace tiempo escribí un artículo en español sobre ello[2].

Bien, volvemos a tener los dos tests pasando, el unitario y el de integración. Si nos paramos a pensarlo bien, quizás deberíamos de separar los tests de integración en distintas DLL. Una DLL debería contener los tests que son para sistemas MS Windows, otra los que son para Linux y así con cada sistema. De eso trata la integración. Me plantearía utilizar máquinas virtuales para lanzar cada batería de tests. ¿Se lo había planteado?

Afortunadamente .Net se basa en un estándar que implementa también el framework Mono[3], por lo que no tenemos que preocuparnos de que el sistema se comporte distinto en Windows que en Linux. Tan solo tenemos que mirar la API de `File` para ver, qué posibles excepciones se pueden producir y escribir tests que se ejecuten con el mismo resultado en cualquier plataforma. Vamos a arreglar nuestro test de integración para que funcione también en Linux y MacOS.

```
1  namespace IntegrationTests
2  {
3      [TestFixture]
4      public class FileHandlerTests
5      {
6          private string getPlatformPath()
7          {
8              System.OperatingSystem osInfo =
9                  System.Environment.OSVersion;
10             string path = String.Empty;
```

[2]http://www.carlosble.com/?p=257

[3]http://www.mono-project.com

```
11              switch (osInfo.Platform)
12              {
13                  case System.PlatformID.Unix:
14                  {
15                      path = "/tmp/data.txt";
16                      break;
17                  }
18                  case System.PlatformID.MacOSX:
19                  {
20                      path = "/tmp/data.txt";
21                      break;
22                  }
23                  default:
24                  {
25                      path = @"C:\data.txt";
26                      break;
27                  }
28              }
29              return path;
30          }
31
32          [Test]
33          public void CreateFileMultiPlatform()
34          {
35
36              string path = getPlatformPath();
37              UserFileHandler handler =
38                  new UserFileHandler();
39
40              DataFile dataFile = handler.CreateFile(path);
41              if (!File.Exists(path))
42              {
43                  Assert.Fail("File was not created");
44              }
45              Assert.IsNotNull(dataFile.Stream);
46          }
47      }
48  }
```

Perfecto, ya tenemos todas las plataformas que nos interesan por ahora cubiertas. Ya no importa que algunos de los desarrolladores del equipo trabajen con Ubuntu y otros con Windows 7. Todo esto es igual de aplicable a Java. ¿Por dónde vamos?

- *Crear un fichero en la ruta especificada*
- *Resolver casos en que no se puede crear*

  *Permisos insuficientes*

  *Ruta no existe*
- *Leer un fichero de texto línea a línea*

Avancemos un poco más en los casos de posibles excepciones creando el fichero. Primero siempre prefiero abordar el test unitario antes que el de

integración:

10.1: UnitTests

```
[Test]
[ExpectedException(typeof(DirectoryNotFoundException))]
public void TryCreateFileWhenDirectoryNotFound()
{
    FileHandler<UserFile> handlerMock =
        MockRepository.GenerateStub<FileHandler<UserFile>>();
    handlerMock.Expect(
        x => x.CreateFile("")).Throw(
                new DirectoryNotFoundException());

    UsersStorageManager manager =
        new UsersStorageManager(handlerMock);
    manager.SetUsersFile("");
}
```

Le he dicho a Rhino.Mocks que lance una excepción cuando se invoque a `CreateFile`. El comportamiento que estoy expresando es que `UserStorageManager` no va a capturar la excepción sino a dejarla pasar. Nótese que estoy utilizando un stub y no un mock, puesto que la verificación de la llamada al colaborador ya está hecha en el test anterior. No olvide que cada test se centra en una única característica del SUT.

Vamos a la parte de integración:

10.2: IntegrationTests

```
[Test]
[ExpectedException(typeof(DirectoryNotFoundException))]
public void CreateFileDirectoryNotFound()
{
    string path = new NotFoundPath().GetPlatformPath();
    UserFileHandler handler =
        new UserFileHandler();

    DataFile dataFile = handler.CreateFile(path);
}
```

En realidad he copiado el test despúes de su refactorización. Estamos en verde sin tocar el SUT. El código tras la refactorización que he aplicado, es el siguiente:

10.3: IntegrationTests

```
public abstract class MultiPlatform
{
    public abstract string GetPOSIXpath();

    public abstract string GetWindowsPath();

    public string GetPlatformPath()
    {
        System.OperatingSystem osInfo =
```

```
10              System.Environment.OSVersion;
11          string path = String.Empty;
12          switch (osInfo.Platform)
13          {
14              case System.PlatformID.Unix:
15                  {
16                      path = GetPOSIXpath();
17                      break;
18                  }
19              case System.PlatformID.MacOSX:
20                  {
21                      path = GetPOSIXpath();
22                      break;
23                  }
24              default:
25                  {
26                      path = GetWindowsPath();
27                      break;
28                  }
29          }
30          return path;
31      }
32  }
33
34  public class NotFoundPath : MultiPlatform
35  {
36
37      public override string GetPOSIXpath()
38      {
39          return "/asdflwiejawseras/data.txt";
40      }
41
42      public override string GetWindowsPath()
43      {
44          return @"C:\asdfalsdfkwjerasdfas\data.txt";
45      }
46  }
47
48  public class EasyPath : MultiPlatform
49  {
50
51      public override string GetPOSIXpath()
52      {
53          return "/tmp/data.txt";
54      }
55
56      public override string GetWindowsPath()
57      {
58          return @"C:\data.txt";
59      }
60  }
61
62  [TestFixture]
63  public class FileHandlerTests
64  {
65      [Test]
66      public void CreateFileMultiPlatform()
67      {
68          string path = new EasyPath().GetPlatformPath();
```

```
69          UserFileHandler handler =
70              new UserFileHandler();
71
72          DataFile dataFile = handler.CreateFile(path);
73          if (!File.Exists(path))
74          {
75              Assert.Fail("File was not created");
76          }
77          Assert.IsNotNull(dataFile.Stream);
78      }
79
80      [Test]
81      [ExpectedException(typeof(DirectoryNotFoundException))]
82      public void CreateFileDirectoryNotFound()
83      {
84          string path = new NotFoundPath().GetPlatformPath();
85          UserFileHandler handler =
86              new UserFileHandler();
87
88          DataFile dataFile = handler.CreateFile(path);
89      }
90  }
```

Para las demás excepciones que lance el sistema, tales como `UnauthorizedAccessException` o `ArgumentNullException`, no voy a escribir más tests puesto que sé que el comportamiento de mis clases es el mismo, al no capturar las excepciones. Si decido más adelante utilizar `UserStorageManager` desde otra clase y capturar las posibles excepciones para construir un mensaje amigable para el usuario, entonces escribiré los tests correspondientes en su momento y lugar.

Nos queda trabajar en la lectura del fichero línea a línea. ¿Qué es lo primero que haremos? Escribir un test unitario que utilice un colaborador que cumple una interfaz (contrato), la cual contiene las llamadas a los métodos que nos devuelven datos. Al trabajar primero en el test unitario nos cuidamos mucho de que la interfaz que definimos como contrato para hablar con el exterior, nos resulte lo más cómoda posible. Es todo lo contrario que en el desarrollo clásico, cuando diseñamos la interfaz y luego al usarla tenemos que hacer malabares para adaptarnos a ella. Esta parte queda como ejercicio pendiente para el lector.

## 10.2. Diseño emergente con un ORM

Vamos a irnos directamente al acceso a base de datos. Para ello asumiremos que estamos trabajando con algún ORM[4] como Hibernate, ActiveRecord, Django ORM o similar.

[4] http://es.wikipedia.org/wiki/Mapeo_objeto-relacional

Las dos formas más extendidas de manejar datos mediante un ORM son las siguientes:

- Un DAO con métodos de persistencia + modelos anémicos
- Un modelo con atributos para datos y métodos de persistencia

Ambas tienen sus ventajas e inconvenientes. La primera opción se compone de un DAO[5] que tiene métodos como `save`, `create` y `delete`, cuyos parámetros son modelos. Estos modelos son clases sin métodos funcionales pero con atributos designados para alojar datos, tales como `Name`, `Address`, `Phone`, etc. Martin Fowler les llamó modelos anémicos por la ausencia de lógica de negocio en ellos.

La segunda opción es la mezcla de los dos objetos anteriores en uno solo que unifica lógica de persistencia y campos de datos.

Por revisar los distintos casos que se nos pueden plantear a la hora de aplicar TDD, vamos a abandonar el ejemplo que veníamos desarrollando desde el capítulo 8 y a enfocarnos sobre ejemplos puntuales. Siento decirle que no vamos a terminar de implementar todo lo que el cliente nos había pedido en el capítulo 8. Toda esa larga historia que el cliente nos contó, era sólo para que el lector no pensase que nuestra aplicación de ejemplo no es suficientemente "empresarial".

En Hibernate y NHibernate el patrón que se aplica viene siendo el del DAO más los modelos. El DAO es un objeto que cumple la interfaz `Session` en el caso de Hibernate e `ISession` en el caso de NHibernate. Con JPA[6] los modelos tienen anotaciones para expresar relaciones entre ellos o tipos de datos. Exactamente lo mismo que ocurre en .Net con *Castle.ActiveRecord*[7]. En lo sucesivo hablaré de Hibernate para referirme a la versión Java y a la versión .Net indistintamente.

Si nos casamos con Hibernate como ORM, entonces la frontera que delimita nuestros tests unitarios y nuestros tests de integración puede ser la propia interfaz `Session`. Ya sabemos que mediante un framework de mocks podemos simular que este DAO invoca a sus métodos de persistencia, con lo cual utilizamos modelos en los tests unitarios sin ningún problema. Veamos un pseudo-código de un test unitario:

```
1 [Test]
2 public void UserManagerCreatesNewUser()
3 {
4       UserModel user = new UserModel();
5       ISession sessionMock =
```

[5]Data Access Object
[6]Java Persistence API
[7]Proyecto CastleProject

```
6                 MockRepository.GenerateMock<ISession>();
7         sessionMock.Expect(
8                   x => x.Save(user)).IgnoreArguments();
9
10        UserManager manager = new UserManager(sessionMock);
11        manager.SaveNewUser('Angel', 'clave1234');
12
13        sessionMock.VerifyAllExpectations();
14  }
```

Estamos diciendo que nuestro `UserManager` es capaz de recibir un nick y una clave y guardar un registro en la base de datos. He especificado que se ignoren los argumentos porque únicamente me preocupa que se salve el registro, nada más. Para continuar ahora probando la parte de integración, podría tomar el modelo `UserModel`, grabarlo en base de datos y a continuación pedirle al ORM que lo lea desde la base de datos para afirmar que se guardó bien.

Si se fija bien verá que el formato es idéntico al del acceso a ficheros del principio del capítulo. Si lo comprende le valdrá para todo tipo de tests que se aproximan a la frontera de nuestras clases con el sistema externo.

¿Cómo abordamos los tests unitarios con ORMs que unifican modelo y DAO? Fácil. Nos inventamos el DAO nosotros mismos a través de una interfaz que hace de frontera. En nuestra lógica de negocio, en lugar de llamar directamente a los métodos de persistencia del modelo, invocamos al DAO pasando el modelo. Como el modelo incluye toda la lógica de persistencia, el código del DAO es super sencillo, se limita a invocar al modelo. Pseudocódigo:

```
1  public class DAO()
2  {
3          public void Save(Model model)
4          {
5              model.Save();
6          }
7  }
```

Habrá casos en los que quizás podamos trabajar directamente con el modelo si hablamos de operaciones que no acceden a la base de datos. No se trata de adoptar el patrón del DAO para todos los casos de manera general. Donde sea que quiera establecer un límite entre lógica de negocio diseñada con tests unitarios, lo utilizaré y donde no, aprovecharé la potencia de un modelo que incorpora lógica de persistencia.

Para los tests de integración conviene trabajar con una base de datos diferente a la de producción porque si su ejecución falla, la base de datos puede quedar inconsistente. Muchos frameworks ofrecen la posibilidad de crear una base de datos al vuelo cuando lanzamos los tests de integración y destruirla al terminar. Además entre un test y otro se encargan de vaciar las tablas para obligar a que los tests sean independientes entre sí.

Hay que tener cuidado de no abusar de estos frameworks, evitando que se creen y se destruyan bases de datos cuando tan sólo estamos escribiendo tests unitarios. Esto ocurre con el framework de tests de Django[8] por ejemplo, que siempre crea la base de datos al vuelo incluso aunque no accedamos a ella. Mi recomendación en este caso es utilizar el mecanismo de tests de Django para los tests de integración y utilizar pyunit para los tests unitarios. La razón es que los tests unitarios deben ser rápidos y crear la base de datos consume más de 5 segundos. Demasiado tiempo.

### 10.2.1. Diseñando relaciones entre modelos

Típicamente cuando llega el momento de diseñar modelos y empezar a trazar relaciones entre unos y otros[9], me gusta también hacerlo con un test primero. Supongamos que mi modelo `User` tiene una relación de uno a muchos con mi modelo `Group`, es decir, que un grupo tiene uno o más usuarios. Si mi objetivo es trabajar con modelos, entonces me ahorro la parte de tests unitarios y salto directamente a tests de integración. Probar una relación tan sencilla como esta mediante un test unitario no vale la pena, por lo menos en este momento. En lugar de escribir ambas clases a priori, diseño su API desde un test de integración para asegurarme que es así como los quiero utilizar:

```
1  [Test]
2  public void UserBelongsInManyGroups()
3  {
4       Group group = Group();
5       group.Name = "Curris";
6       group.Save();
7
8       UserModel user = new UserModel();
9       user.name = "curri1";
10      user.Group = group;
11      user.Save();
12
13      UserModel user = new UserModel();
14      user.name = "curri2";
15      user.Group = group;
16      users.Save();
17
18      Group reloaded = Group.FindAll();
19      Assert.AreEqual(2, reloaded.Count);
20      ...
21 }
```

Además de que el test me ayuda a encontrar la API que busco, me puede servir para probar que estoy trabajando bien con el framework ORM en caso de que esté utilizando mecanismos complejos tipo caché.

[8] Desarrollo web con Python (http://www.djangoproject.com/)
[9] OneToMany, ManyToMany

Es importante tener cuidado de no hacer tests para probar que el framework que estamos usando funciona. Es decir, si escojo Hibernate no me pongo a escribir tests para comprobar que funciona. Igual que tampoco pruebo que .Net funciona. Cuando nos iniciamos en el desarrollo de tests de integración es común caer en la trampa de lanzarse a escribir tests para todas las herramientas que estamos utilizando. Tenemos que partir de la base de que el código de terceros, que utilizamos es robusto. Hay que confiar en ello y asumir que funciona tal cual diga su documentación. La próxima vez que escoja una herramienta de terceros asegúrese de que dispone de una buena batería de tests que al menos cumple un mínimo de calidad.

El diseño emergente para los modelos no es algo que haga siempre. Si se trata de unos modelos que ya he diseñado mil veces como por ejemplo usuarios y grupos, no hago un test primero. Sin embargo en muchas otras ocasiones, sí que me ayuda a clarificar lo que necesito. Hago TDD cuando por ejemplo tengo que leer datos de un fichero de entrada y guardarlos en base de datos a través de modelos o simplemente cuando no entiendo las relaciones entre entidades sin ver (y escribir) un ejemplo primero.

## 10.3. La unificación de las piezas del sistema

Ya casi tenemos todas las piezas del sistema diseñadas pero en lo que a tests se refiere todavía nos queda un área que cubrir. Son los tests de sistema. Es decir, tests de integración que van de extremo a extremo de la aplicación. En el `UserManager` de antes, tenemos tests que cubren su lógica de negocio y tests que cubren la integración de nuestra clase que accede a datos con el sistema exterior. Sin embargo no hay ningún bloque de código que se ocupe de inyectar el objeto `ISession` en `UserManager` para operar en producción. Para esto escribimos un nuevo test de integración que cubre todo el área. Para este ejemplo concreto escribiría el test antes que el SUT al estilo TDD porque todavía es un conjunto de objetos reducido. Sin embargo, cuando el área a cubrir es mayor, puede ser un antipatrón seguir haciendo TDD. A veces conviene escribir tests a posteriori. Encuentro que es así cuando se trata de integrar la interfaz de usuario con el resto del sistema. Intento por todos los medios dejar la interfaz gráfica para el final del todo, cuando lo demás está terminado y sólo hay que colocar la carcasa. Es una buena manera de evitar que el diseño gráfico contamine la lógica de negocio y al mismo tiempo produce código menos acoplado y más fácil de mantener. En general, TDD para integrar interfaces de usuario no es una práctica común.

Los tests de sistema para aplicaciones web se pueden llevar a cabo con

herramientas como Selenium[10]. Soy de la opinión de que no se debe abusar de los tests que atacan a la interfaz de usuario porque son extremádamente frágiles. Ante cualquier cambio se rompen y cuesta mucho mantenerlos. Si nuestra batería de tests unitarios contempla la mayor parte de la lógica de negocio y nuestros tests de integración de bajo nivel (como los que hemos hecho en este capítulo) contienen las partes clave, me limitaría a escribir unos cuantos tests de sistema para comprobar, grosso modo, que todas las piezas encajan pero no intentaría hacer tests para toda la funcionalidad del sistema atacando a la interfaz de usuario.

Cuando queremos refactorizar un código legado (que no tiene tests automáticos) entonces es buena idea blindar la aplicación con tests de sistema antes de tocar código, porque así seremos avisados de los destrozos que hagamos. Por desgracia escribir todos esos tests es un trabajo de usar y tirar. Una vez refactorizado el código y creados sus correspondientes tests unitarios y de integración, habrá cambios en los niveles superiores que romperán los tests de sistema y dejará de valer la pena mantenerlos. Cuando sea posible será mejor opción escribir tests unitarios si el código lo permite. Michael Feathers lo describe en su libro[6].

[10] http://seleniumhq.org/

Capítulo 11

# La solución en versión Python

Al igual que en los capítulos anteriores, vamos a escribir la Supercalculadora (al menos una versión reducida) siguiendo el desarrollo orientado por pruebas pero, esta vez, en Python. Para la escritura del código vamos a seguir el estándar de estilo PEP8 [1], una guía de estilo muy conocida en el mundillo Python. Además, usaremos el clásico y estándar `pyunit` para las pruebas unitarias aunque también se podría utilizar el sencillo `doctests` o el potente `nosetests`[2] (el cual recomiendo) que es un poderoso *framework* para test unitarios que nos hará la vida mucho más fácil.

El objetivo de este capítulo no es más que mostrar al lector que hacer TDD con un lenguaje interpretado como Python es igual que con un lenguaje compilado o híbrido. Otro punto a destacar es la elección de los tests unitarios y cómo ésto puede cambiar el diseño de la solución. Esto no es más que la visualización del llamado *diseño emergente* el cual "sale" de los tests, por tanto, si la elección de pruebas es distinta en distintos puntos (o el orden) el diseño puede variar (aunque el resultado debe permanecer igual).

Una aclaración más antes de empezar. Es posible que le sorprenda después de leer los capítulos anteriores que el código esté escrito en español en lugar de inglés. Después de varias discusiones sobre el tema, el bueno de Carlos ha aceptado a que así fuera. Al contrario que él, creo que el código no tiene por qué estar escrito en inglés. Estará escrito en esa lengua por todo aquel que se sienta cómodo escribiendo en inglés pero me parece contraproducente el hacerlo "por deber" cuando se tiene otra lengua con la cual se sienta uno más a gusto. Por supuesto, huelga decir, que el contexto influye y que si estamos en una organización internacional, en un trabajo donde hay otros desarrolladores que sólo hablan inglés, lo lógico (y debido puesto que el

[1] http://www.python.org/dev/peps/pep-0008/
[2] http://code.google.com/p/python-nose/

idioma internacional *de facto*) es hacerlo en inglés ya que el código debe ser entendido por todo el mundo (por cierto, que sirva como nota que en este mundillo, todo el mundo debería ser capaz de manejarse sin problema en inglés). En este caso, ya que el libro está enfocado a lectores hispanos y para evitar que alguien pueda no entender correctamente el código en inglés, hemos decidido dar una "nota de color" en este capítulo escribiendo el código en español.

Sin más dilación, vamos a ponernos a ello resumiendo en una frase qué es lo que el cliente nos pedía (para obtener la información completa, ver Capítulo 8 en la página 121):

> *Una calculadora de aritmética básica de números enteros*

Para seguir la misma filosofía que el capítulos anteriores, vamos a usar los mismos tests de aceptación que en el caso de la Supercalculadora en C# aunque, como veremos más tarde, quizá tomemos decisiones distintas a lo hora de elegir los test unitarios o el orden en que los implementaremos. Para no duplicar, no nos vamos a meter a analizar los tests de aceptación en este capítulo. En caso de que quiera revisar los motivos de por qué estos tests y no otros, por favor, vuelva a releer los capítulos anteriores. Para refrescar la memoria, a continuación se muestran los tests de aceptación elegidos al inicio del proyecto:

- "2 + 2", devuelve 4
- "5 + 4 * 2 / 2", devuelve 9
- "3 / 2", produce el mensaje ERROR
- "* * 4 - 2": produce el mensaje ERROR
- "* 4 5 - 2": produce el mensaje ERROR
- "* 4 5 - 2 -": produce el mensaje ERROR
- "*45-2-": produce el mensaje ERROR

¿Recuerda cual era nuestra forma de trabajo en capítulos anteriores?, sí, una muy sencilla: un editor de textos por un lado para llevar las notas (al que llamábamos *libreta*) y un IDE por otro para escribir el código. De nuevo vamos a seguir esta forma de trabajar que aunque sencilla, es muy poderosa. Sin embargo, vamos a simplificar aún más ya que en lugar de un IDE vamos a usar otro *editor* de textos, Emacs[3] con el modo *python-mode*[4] (vaya, ahora podemos usar el mismo programa como *libreta*, como *editor* de código y también como editor de LaTeX, que no se nos olvide).

[3] Emacs 23.1 - ftp://ftp.gnu.org/gnu/emacs/windows/emacs-23.1-bin-i386.zip
[4] http://www.rwdev.eu/articles/emacspyeng

A por ello, entonces. Lo primero, creamos un directorio donde vamos a trabajar, por ejemplo "supercalculadora_python". Y dentro de este directorio creamos nuestro primer fichero que, por supuesto, será de pruebas: `ut_supercalculadora.py`. Vamos a empezar con los mismos test unitarios procedentes del primer test de aceptación. Estos son los siguientes:

```
# Aceptación - "2 + 2", devuelve 4
▪  Sumar 2 al número 2, devuelve 4
▪  La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce el mensaje ERROR
# Aceptación - "* * 4 - 2": produce el mensaje ERROR
# Aceptación -"* 4 5 - 2": produce el mensaje ERROR
# Aceptación -"* 4 5 - 2 : produce el mensaje ERROR
# Aceptación -"*45-2-": produce el mensaje ERROR
```

Vamos a por el primer test de desarrollo:

```
1  import unittest
2  import supercalculadora
3
4  class TestsSupercalculadora(unittest.TestCase):
5      def test_sumar_2_y_2(self):
6          calc = supercalculadora.Supercalculadora()
7          self.failUnlessEqual(4, calc.sumar(2, 2))
8
9  if __name__ == "__main__":
10     unittest.main()
```

El código no compila porque todavía no hemos creado ni el fichero `supercalculadora` ni la clase `Supercalculadora`. Sin embargo ya hemos diseñado algo: el nombre de la clase, su constructor y su primer método (nombre, parámetros que recibe y el resultado que devuelve). El diseño, como vemos, emerge de la prueba. A continuación escribimos el mínimo código posible para que el test pase al igual que en capítulos anteriores. Devolver 4, que es el resultado correcto, es la implementación mínima en este caso.

11.1: supercalculadora.py

```
1  class Supercalculadora:
2      def sumar(self, a, b):
3          return 4
```

Muy bien, ahora habría que refactorizar pero estamos tan al principio que todavía no hay mucho que hacer así que vamos a seguir con la siguiente prueba unitaria sobre el mismo tema.

```
1      def test_sumar_5_y_7(self):
2          self.failUnlessEqual(12, self.calc.sumar(5, 7))
```

Ejecutamos (`ctrl+c ctrl+c`) en Emacs y observamos que falla (en una ventana distinta dentro del editor):

```
.F
======================================================================
FAIL: test_sumar_5_y_7 (__main__.TestsSupercalculadora)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<stdin>", line 11, in test_sumar_5_y_7
AssertionError: 12 != 4

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Vamos entonces a cambiar la implementación que sería mínima para este test:

```
class Supercalculadora:
    def sumar(self, a, b):
        return 12
```

Sin embargo, aunque este es el código mínimo que hace pasar la segunda prueba, hace fallar la primera (es un fallo que podríamos haber obviado pero por motivos didácticos lo incluimos como primer ejemplo ya que en ejemplos más complejos no es fácil ver el fallo de pruebas anteriores) por lo tanto debemos buscar la implementación mínima que hace pasar todas las pruebas. En este caso, sería:

```
class Supercalculadora:
    def sumar(self, a, b):
        return a + b
```

Ahora ya tenemos la luz verde ("`Ran 2 tests in 0.000s OK`" en nuestro caso) así que pasemos al paso de refactorización. Esta vez, sí que hay cosas que hacer ya que como vemos en las pruebas, la línea `calc = supercalculadora.Supercalculadora()` está duplicada. Subsanémoslo creando un método `setUp` (las mayúsculas y minúsculas son importantes) donde movemos la duplicidad de las pruebas (por consistencia, añadimos también el método `tearDown` aunque no hará nada ya que es el contrario a `setUp`)

```
class TestsSupercalculadora(unittest.TestCase):
    def setUp(self):
        self.calc = supercalculadora.Supercalculadora()

    def tearDown(self):
        pass

    def test_sumar_2_y_2(self):
        self.failUnlessEqual(4, self.calc.sumar(2, 2))

    def test_sumar_5_y_7(self):
```

```
12          self.failUnlessEqual(12, self.calc.sumar(5, 7))
```

Voy a añadir un último test que pruebe que el orden en los sumandos no altera el resultado (propiedad conmutativa), además de añadirlo en la libreta ya que puede ser de importancia para futuras operaciones. También, como puede apreciar, marcaremos con **HECHAS** las cuestiones resueltas de la libreta por claridad.

```
# Aceptación - "2 + 2", devuelve 4
  - Sumar 2 al número 2, devuelve 4 - ¡HECHO!
  - La propiedad conmutativa se cumple
  - La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce el mensaje ERROR
# Aceptación - "* * 4 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2 : produce el mensaje ERROR
# Aceptación - "*45-2-": produce el mensaje ERROR
```

Vamos con ello con un tests:

```
1       def test_sumar_propiedad_conmutativa(self):
2           self.failUnlessEqual(self.calc.sumar(5, 7),
3                                self.calc.sumar(7, 5))
```

La prueba pasa sin necesidad de tocar nada. ¡Genial!, la propiedad conmutativa se cumple. Hasta el momento y para recapitular, tenemos los siguientes tests:

```
1   class TestsSupercalculadora(unittest.TestCase):
2       def setUp(self):
3           self.calc = supercalculadora.Supercalculadora()
4
5       def tearDown(self):
6           pass
7
8       def test_sumar_2_y_2(self):
9           self.failUnlessEqual(4, self.calc.sumar(2, 2))
10
11      def test_sumar_5_y_7(self):
12          self.failUnlessEqual(12, self.calc.sumar(5, 7))
13
14      def test_sumar_propiedad_conmutativa(self):
15          self.failUnlessEqual(self.calc.sumar(5, 7),
16                               self.calc.sumar(7, 5))
```

Todo bien hasta el momento. ¡Fenomenal!

```
# Aceptación - "2 + 2", devuelve 4
  - Sumar 2 al número 2, devuelve 4 - ¡HECHO!
  - La propiedad conmutativa se cumple - ¡HECHO!
  - La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce el mensaje ERROR
# Aceptación - "* * 4 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2 : produce el mensaje ERROR
# Aceptación - "*45-2-": produce el mensaje ERROR
```

Hemos actualizado la libreta con el último resultado. Recordad que es muy importante que la libreta esté en todo momento actualizada y refleje el estado actual de nuestro desarrollo.

En este punto, paremos un momento y miremos si hay algo que refactorizar. Vemos que el código es bastante limpio así que reflexionamos sobre cómo seguir.

Hemos hecho la suma pero, sin embargo, no tenemos ningún test de aceptación sobre la resta que parece algo distinta. En el caso anterior deberíamos verificar el orden ya que será importante (no se cumple la propiedad conmutativa). Además, en ciertos casos el número devuelto podría ser negativo y eso no lo habíamos contemplado. ¿Debe ser un error o un número negativo es correcto?, obviamente este caso es trivial y queremos que los valores negativos sean aceptados (un número negativo es un entero y eso es lo que había pedido el cliente, ¿no?) pero, por si acaso, ¡preguntémosle al cliente!. Es mejor estar seguros de que quiere trabajar con negativos y que cuando dijo "entero" realmente quería decir "entero" y no "natural".

Añadamos en la libreta un test de aceptación y unas cuantas pruebas unitarias para este nuevo caso.

```
# Aceptación - "2 + 2", devuelve 4
  - Sumar 2 al número 2, devuelve 4 - ¡HECHO!
  - La propiedad conmutativa se cumple - ¡HECHO!
  - La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
# Aceptación - "5 - 3", devuelve 2
  - Restar 5 al número 3, devuelve 2
  - Restar 2 al número 3, devuelve -1
  - La propiedad conmutativa no se cumple
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce el mensaje ERROR
# Aceptación - "* * 4 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2 : produce el mensaje ERROR
# Aceptación - "*45-2-": produce el mensaje ERROR
  - Operaciones con números negativos
```

Podríamos seguir con el último test unitario que tenemos en la suma, el de los operadores y operandos. Sin embargo, parece "algo distinto" y sabemos que requiere algo más que cambios en `Supercalculadora`, por eso decidimos posponerlo un poco y pasarnos a la resta. Escribimos el primer test para la resta:

```
1 ...
2     def test_restar_5_y_3(self):
3         self.failUnlessEqual(2, self.calc.restar(5, 3))
```

Por supuesto, falla (la operación resta no existe todavía), así que vamos a ponernos con la implementación como ya hemos hecho anteriormente.

```
1 class Supercalculadora:
2 ...
3     def restar(self, a, b):
4         return 2
```

¡Pasa de nuevo!. ¿Refactorizamos?, parece que todavía el código (tanto de las pruebas como el resto) es limpio, así que vamos a seguir con otra prueba.

```
1 ...
2     def test_restar_2_y_3(self):
3         self.failUnlessEqual(-1, self.calc.restar(2, 3))
```

Luz roja de nuevo. Vamos a arreglarlo de tal modo que tanto esta como la prueba anterior (y por supuesto todas las demás que formen o no parte de la resta) pasen.

11.2: supercalculadora.py

```
class Supercalculadora:
    def sumar(self, a, b):
        return a + b

    def restar(self, a, b):
        return a - b
```

Otra vez en verde. Y seguimos con la siguiente prueba ya que al igual que antes no vemos necesidad de refactorizar (¡pero lo mantenemos en mente!)

```
...
    def test_restar_no_propiedad_conmutativa(self):
        self.failIfEqual(self.calc.restar(5, 3),
                         self.calc.restar(3, 5))
```

Sigue funcionando sin tocar nada... pero, al escribir esto se nos vienen a la cabeza otra serie de preguntas (actualizamos, a la vez, la libreta).

```
# Aceptación - "2 + 2", devuelve 4
  - Sumar 2 al número 2, devuelve 4 - ¡HECHO!
  - La propiedad conmutativa se cumple - ¡HECHO!
  - La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
# Aceptación - "5 - 3", devuelve 2
  - Restar 5 al número 3, devuelve 2 - ¡HECHO!
  - Restar 2 al número 3, devuelve -1 - ¡HECHO!
  - La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", produce el mensaje ERROR
# Aceptación - "* * 4 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2": produce el mensaje ERROR
# Aceptación - "* 4 5 - 2 : produce el mensaje ERROR
# Aceptación - "*45-2-": produce el mensaje ERROR
  - ¿Qué número es el más grande que se puede manejar?, ¿y el más
    pequeño?
```

Las nuevas cuestiones sobre límites numéricos atañen a todas las operaciones de la calculadora y no sabemos qué debemos hacer en esta disyuntiva. Lo primero es clarificar qué comportamiento debe tener el software preguntando al cliente. De ahí podemos sacar los nuevos test de aceptación para las dudas que tenemos.

No vamos a ir por ese camino como hemos hecho en los capítulos anteriores. Recuerde que podíamos recorrer el "árbol de tests" en profundidad o en amplitud. Anteriormente lo hemos hecho en amplitud así que esta vez vamos a coger el otro camino y hacerlo en profundidad. Vamos a ir más lejos

en el camino ya andado y no moviéndonos a otras partes del árbol (los nuevos tests que aparecerían sobre el número máximo y mínimo, por ejemplo). Vamos a seguir con el test de aceptación que ya íbamos manejando.

Ahora bien, parece que vamos a cambiar de tercio porque el test unitario que habíamos definido al principio en la suma es más un test de un *parser* que de la calculadora en sí. Nos ponemos con él en el mismo sitio donde estabamos escribiendo los test unitarios anteriormente con algunas decisiones de diseño ya en la cabeza, por ejemplo, que la expresión será una cadena de caracteres. Por claridad, ponemos todas la pruebas hasta ahora:

11.3: ut_supercalculadora.py

```
import unittest
import supercalculadora
import expr_aritmetica

class TestsSupercalculadora(unittest.TestCase):
    def setUp(self):
        self.calc = calculadora.Calculadora()

    def tearDown(self):
        pass

    def test_sumar_2_y_2(self):
        self.failUnlessEqual(4, self.calc.sumar(2, 2))

    def test_sumar_5_y_7(self):
        self.failUnlessEqual(12, self.calc.sumar(5, 7))

    def test_sumar_propiedad_conmutativa(self):
        self.failUnlessEqual(self.calc.sumar(5, 7),
                             self.calc.sumar(7, 5))

    def test_restar_5_y_3(self):
        self.failUnlessEqual(2, self.calc.restar(5, 3))

    def test_restar_2_y_3(self):
        self.failUnlessEqual(-1, self.calc.restar(2, 3))

    def test_restar_no_propiedad_conmutativa(self):
        self.failIfEqual(self.calc.restar(5, 3),
                         self.calc.restar(3, 5))

    def test_extraer_operandos_y_operadores_en_2_mas_2(self):
        expresion = expr_aritmetica.ExprAritmetica()
        self.failUnless({'Operandos': [2, 2], 'Operadores': ['+']},
                        expresion.parse("2␣+␣2"))
```

Estamos en rojo porque el archivo "expr_aritmetica.py" ni siquiera existe. Lo corregimos pero también falla porque la clase `ExprAritmetica` tampoco existe. Lo corregimos igualmente y ahora falta el método `parse` que creamos con la mínima implementación posible. Sería algo así:

11.4: expr_aritmetica.py

```
class ExprAritmetica:
    def parse(self, exp):
        return {'Operandos': [2, 2], 'Operadores': ['+']}
```

Por fin pasa las pruebas y, como es habitual, pensamos en el siguiente paso, la refactorización.

Ummm, hay muchas cosas que refactorizar. Empecemos por el nombre de la clase Supercalculadora (y su fichero). Parece ser ahora que tenemos dos "módulos" que son parte de un programa que será la supercalculadora, la clase supercalculadora debería ser, en realidad, calculadora puesto que es la clase encargada de calcular (supercalculadora será la aplicación en sí o, como mucho, la clase principal). Cambiemos el nombre del fichero (por `calculadora.py`) y de la clase.

Sigamos por los tests. Tenemos los tests de la clase Calculadora y de la clase `ExprAritmetica` en la misma clase de tests llamada `TestsSupercalculadora`. Aunque ambos son tests de la aplicación Supercalculadora creo que es necesario tenerlos separados en tests para Calculadora y tests para ExprAritmetica para que todo quede más claro. Para alcanzar mayor claridad aún, vamos a separarlos también en dos ficheros de tests, uno para cada "módulo" al igual que la implementación. En el archivo principal creamos una *suite* donde recogemos todos las pruebas de los distintos "módulos".

Tenemos entonces:

11.5: ut_supercalculadora.py

```
import unittest
import ut_calculadora
import ut_expr_aritmetica

if __name__ == "__main__":
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(
                    ut_calculadora.TestsCalculadora))
    suite.addTest(unittest.makeSuite(
                    ut_expr_aritmetica.TestsExprAritmetica))
    unittest.TextTestRunner(verbosity=3).run(suite)
```

11.6: ut_calculadora.py

```
import unittest
import calculadora

class TestsCalculadora(unittest.TestCase):
    def setUp(self):
        self.calc = calculadora.Calculadora()

    def tearDown(self):
        pass

```

```
11      def test_sumar_2_y_2(self):
12          self.failUnlessEqual(4, self.calc.sumar(2, 2))
13
14      def test_sumar_5_y_7(self):
15          self.failUnlessEqual(12, self.calc.sumar(5, 7))
16
17      def test_sumar_propiedad_conmutativa(self):
18          self.failUnlessEqual(self.calc.sumar(5, 7),
19                               self.calc.sumar(7, 5))
20
21      def test_restar_5_y_3(self):
22          self.failUnlessEqual(2, self.calc.restar(5, 3))
23
24      def test_restar_2_y_3(self):
25          self.failUnlessEqual(-1, self.calc.restar(2, 3))
26
27      def test_restar_no_propiedad_conmutativa(self):
28          self.failIfEqual(self.calc.restar(5, 3),
29                           self.calc.restar(3, 5))
```

11.7: ut_expr_aritmetica.py

```
1  import unittest
2  import expr_aritmetica
3
4  class TestsExprAritmetica(unittest.TestCase):
5      def test_extraer_operandos_y_operadores_en_2_mas_2(self):
6          expresion = expr_aritmetica.ExprAritmetica()
7          self.failUnlessEqual({'Operandos': [2, 2],
8                                'Operadores': ['+']},
9                               expresion.parse("2␣+␣2"))
```

Sabemos que para `ExprAritmetica` tenemos que implementar mucho más pero por ahora lo vamos a dejar hasta que salga más funcionalidad en las otras pruebas de aceptación. En este punto, actualicemos la libreta y sigamos con la siguiente prueba de aceptación. Ésta es un poco compleja tocando muchos puntos a la vez así que mejor la aplazamos y vamos a pensar qué podemos hacer ahora.

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0
- Restar 2 al número -5, devuelve -7
- Restar -2 al número -7, devuelve -5
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", devuelve un error
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

Nos ponemos con los números negativos puesto que parece más relacionado con el mismo tema, las operaciones. Como siempre, escribimos la prueba primero (para abreviar, pongo todas las pruebas juntas aunque primero se escribe una, luego el código mínimo para que pase, luego se refactoriza, luego otra prueba...). En esta ocasión, hemos escrito las pruebas en dos tests (para suma y para resta puesto que son dos comportamientos distintos en nuestra aplicación) dando nombres que reflejen con claridad la intención de los tests.

```
1 ...
2     def test_sumar_numeros_negativos(self):
3         self.failUnlessEqual(0, self.calc.sumar(2, -2))
4
5     def test_restar_numeros_negativos(self):
6         self.failUnlessEqual(-7, self.calc.restar(-5, 2))
7         self.failUnlessEqual(-5, self.calc.restar(-7, -2))
```

Esto ha sido fácil, la implementación que teníamos ya soportaba los números negativos por los que el test pasa sin necesidad de hacer ninguna modificación en el código. Esto es bueno ya que tenemos más tests que verifican el funcionamiento de la implementación pero malo a la vez porque no hemos avanzado con nuestro software. Debemos seguir escribiendo pruebas unitarias que nos hagan implementar más la funcionalidad del software que estamos

escribiendo.

Actualicemos la libreta, escojamos por dónde seguir y pensemos en nuevos test de aceptación y/o unitarios si hiciera falta.

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0 - ¡HECHO!
- Restar 2 al número -5, devuelve -7 - ¡HECHO!
- Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", devuelve un error
- Dividir 2 entre 2 da 1
- Dividir 10 entre 5 da 2
- Dividir 10 entre -5 da -2
- Dividir -10 entre -5 da 2
- Dividir 3 entre 2 lanza una excepción
- Dividir 3 entre 0 lanza una excepción
- La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/'
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

Vemos que hemos añadido algunos test unitarios al caso de la división. Vamos a seguir con esta prueba de aceptación ya que parece una buena forma de seguir después de la suma y la resta.

Empezamos con las pruebas:

```
1 ...
2     def test_division_exacta(self):
3         self.failUnlessEqual(1, self.calc.dividir(2, 2))
4         self.failUnlessEqual(2, self.calc.dividir(10, 5))
5
```

```
6       def test_division_exacta_negativa(self):
7           self.failUnlessEqual(-2, self.calc.dividir(10, -5))
8           self.failUnlessEqual(2, self.calc.dividir(-10, -5))
```

Recuerde que es un test a la vez, esto es sólo para abreviar. Es decir, las pruebas anteriores se harían en tres pasos de "prueba - código - refactorización". El código después de estas pruebas, sería (recuerde que la primera aproximación sería nada más que `return 1`)

```
1  ...
2      def dividir(self, a, b):
3          return a / b
```

Seguimos con el test de división no entera:

```
1  ...
2      def test_division_no_entera_da_excepcion(self):
3          self.failUnlessRaises(ValueError, self.calc.dividir, 3, 2)
```

Falla, así que escribimos la mímima implementación para hacer pasar las pruebas:

```
1  ...
2      def dividir(self, a, b):
3          if a % b != 0:
4              raise ValueError
5          else:
6              return a / b
```

Parace que no hace falta refactorizar, así que seguimos con el siguiente.

```
1  ...
2      def test_division_por_0(self):
3          self.failUnlessRaises(ZeroDivisionError,
4                                self.calc.dividir, 3, 0)
```

Pasa sin necesidad de tocar el código ya que `a / b` lanza una excepción directamente. ¿Hace falta refactorizar?, parece que todavía no. Ahora pasamos a un nuevo test unitario sobre las expresiones aritméticas que promete ser un poco más complejo de resolver. Vamos primero con el test y luego con la implementación, como siempre.

```
1  ...
2  class TestsExprAritmetica(unittest.TestCase):
3  ...
4     def test_extraer_operandos_y_operadores_en_10_entre_menos_5(self):
5        expresion = expr_aritmetica.ExprAritmetica()
6        self.failUnlessEqual({'Operandos': [10, -5],
7                              'Operadores': ['/']},
8                             expresion.parse("10␣/␣-5"))
```

Y el código más simple que se me ha ocurrido para pasar ambas pruebas es el siguiente:

```
import string

class ExprAritmetica:
    def parse(self, exp):
        operandos = []
        operadores = []
        tokens = string.split(exp)
        for token in tokens:
            try:
                operandos.append(string.atoi(token))
            except ValueError:
                operadores.append(token)
        return {'operandos': operandos, 'operadores': operadores}
```

Un último paso antes de actualizar la libreta. La refactorización. El código parece limpio pero aunque es simple, las excepciones se usan fuera del flujo normal de comportamiento (hay lógica en el `except`) y esto, en general, no es bueno (aunque en un caso tan simple no importe mucho). Lo mejor es que lo arreglemos junto con los tests de la clase `ExprAritmetica` donde vemos que hay algo de duplicidad.

11.8: expr_aritmetica.py

```
    def __es_numero__(self, cadena):
        try:
            string.atoi(cadena)
            return True
        except ValueError:
            return False

    def parse(self, exp):
        operandos = []
        operadores = []
        tokens = exp.split()
        for token in tokens:
            if self.__es_numero__(token):
                operandos.append(string.atoi(token))
            else:
                operadores.append(token)
        return {'operandos': operandos, 'operadores': operadores}
```

Una vez más y como en los casos anteriores, movemos la duplicidad al método `setUp` y creamos su contrario `tearDown` vacío.

11.9: ut_expr_aritmetica.py

```
import unittest
import expr_aritmetica

class TestsExprAritmetica(unittest.TestCase):
   def setUp(self):
       self.expresion = expr_aritmetica.ExprAritmetica()

   def tearDown(self):
       pass
```

```
10
11      def test_extraer_operandos_y_operadores_en_2_mas_2(self):
12          self.failUnlessEqual({'operandos': [2, 2],
13                                'operadores': ['+']},
14                               self.expresion.parse("2␣+␣2"))
15
16      def test_extraer_operandos_y_operadores_en_10_entre_menos_5(self):
17          self.failUnlessEqual({'operandos': [10, -5],
18                                'operadores': ['/']},
19                               self.expresion.parse("10␣/␣-5"))
```

Después de todos estos cambios y de las nuevas pruebas de aceptación que hemos creado, debemos actualizar la libreta:

```
# Aceptación - "2 + 2", devuelve 4
▪ Sumar 2 al número 2, devuelve 4 - ¡HECHO!
▪ La propiedad conmutativa se cumple - ¡HECHO!
▪ La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  - ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
▪ Restar 5 al número 3, devuelve 2 - ¡HECHO!
▪ Restar 2 al número 3, devuelve -1 - ¡HECHO!
▪ La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
▪ Sumar 2 al número -2, devuelve 0 - ¡HECHO!
▪ Restar 2 al número -5, devuelve -7 - ¡HECHO!
▪ Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
# Aceptación - "3 / 2", devuelve un error
▪ Dividir 2 entre 2 da 1 - ¡HECHO!
▪ Dividir 10 entre 5 da 2 - ¡HECHO!
▪ Dividir 10 entre -5 da -2 - ¡HECHO!
▪ Dividir -10 entre -5 da 2 - ¡HECHO!
▪ Dividir 3 entre 2 lanza una excepción - ¡HECHO!
▪ Dividir 3 entre 0 lanza una excepción - ¡HECHO!
▪ La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/' - ¡HECHO!
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
▪ ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

Vamos a movernos un poco más en la `ExprAritmetica`. Hagámos algu-

nas pruebas más.

```
def test_extraer_operandos_y_operadores_expr_sin_ptesis(self):
    self.failUnlessEqual({'operandos': [5, 4, 2, 2],
                          'operadores': ['+', '*', '/']},
                         self.expresion.parse("5␣+␣4␣*␣2␣/␣2"))
```

Vaya, ¡qué sorpresa!. Nuestro parser funciona para expresiones más complejas sin paréntesis. Pongamos al día la libreta y pensemos en cómo seguir adelante.

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  - ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0 - ¡HECHO!
- Restar 2 al número -5, devuelve -7 - ¡HECHO!
- Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
- "5 + 4 * 2 / 2", devuelve 9
- Operandos son '5', '4', '2' y '2' y operadores '+', '*', '/' -
  ¡HECHO!
# Aceptación - "3 / 2", devuelve un error
- Dividir 2 entre 2 da 1 - ¡HECHO!
- Dividir 10 entre 5 da 2 - ¡HECHO!
- Dividir 10 entre -5 da -2 - ¡HECHO!
- Dividir -10 entre -5 da 2 - ¡HECHO!
- Dividir 3 entre 2 lanza una excepción - ¡HECHO!
- Dividir 3 entre 0 lanza una excepción - ¡HECHO!
- La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/' - ¡HECHO!
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

En este momento tenemos un poco de la calculadora y un poco de la

ExprAritmetica. Vamos a dar un giro y en vez de seguir, vamos a integrar estas dos partes. La clase principal será `Supercalculadora` que usará `Calculadora` para calcular el resultado de las operaciones y `ExprAritmetica` para evaluar las expresiones. Decidimos de antemano que a la calculadora le vamos a pasar expresiones aritméticas en forma de cadena de caracteres para que las calcule. Vamos a ver qué diseño sacamos siguiendo TDD...

Renombramos el fichero actual `ut_supercalculadora.py` a `ut_main.py` y creamos de nuevo el fichero `ut_supercalculadora.py`. Comenzamos con las pruebas. La primera será la más básica de la suma. En este punto decidimos que `ExprAritmetica` será pasada como parámetro en el constructor de `Supercalculadora` ya que tiene un comportamiento muy definido.

11.10: ut_supercalculadora.py

```
1  import supercalculadora
2  import ut_calculadora
3  import ut_expr_aritmetica
4
5  class TestsSupercalculadora(unittest.TestCase):
6      def test_sumar(self):
7          sc = supercalculadora.Supercalculadora(
8                   expr_aritmetica.ExprAritmetica())
9          self.failUnlessEqual("4", sc.calcular("2␣+␣2"))
```

El test falla. El fichero `supercalculadora.py` no existe. Una vez creado sigue en rojo porque la clase `Supercalculadora` no existe y, posteriormente, el método `calcular` tampoco. Corrigiendo paso por paso llegamos a una implementación final que será la siguiente:

11.11: supercalculadora.py

```
1  import expr_aritmetica
2  import calculadora
3
4  class Supercalculadora:
5      def __init__(self, parser):
6          self.calc = calculadora.Calculadora()
7          self.parser = parser
8
9      def calcular(self, expresion):
10         expr_descompuesta = self.parser.parse(expresion)
11         if expr_descompuesta['operadores'][0] == '+':
12            return str(self.calc.sumar(
13                           expr_descompuesta['operandos'][0],
14                           expr_descompuesta['operandos'][1]))
```

Ahora tenemos la luz verde de nuevo pero nos empezamos a dar cuenta de que `parse` va a ser difícil de utilizar si queremos operar correctamente con la precedencia de operadores. De todas formas, vayamos paso a paso y creemos más pruebas.

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  - ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0 - ¡HECHO!
- Restar 2 al número -5, devuelve -7 - ¡HECHO!
- Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
- "5 + 4 * 2 / 2", devuelve 9
- "5 + 4 - 3", devuelve 6
- "5 + 4 / 2 - 4", devuelve 3
- Operandos son '5', '4', '2' y '2' y operadores '+', '*', '/' -
  ¡HECHO!
# Aceptación - "3 / 2", devuelve un error
- Dividir 2 entre 2 da 1 - ¡HECHO!
- Dividir 10 entre 5 da 2 - ¡HECHO!
- Dividir 10 entre -5 da -2 - ¡HECHO!
- Dividir -10 entre -5 da 2 - ¡HECHO!
- Dividir 3 entre 2 lanza una excepción - ¡HECHO!
- Dividir 3 entre 0 lanza una excepción - ¡HECHO!
- La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/' - ¡HECHO!
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

Vayamos con el test más sencillo 5 + 4 - 3 pero antes, creemos otros incluso más sencillos aún:

```
1 ...
2     def test_restar(self):
3         sc = supercalculadora.Supercalculadora(
4                   expr_aritmetica.ExprAritmetica())
5         self.failUnlessEqual("0", sc.calcular("2␣-␣2"))
```

Falla como era de esperar así que lo arreglamos con la implementación mínima.

```
...
    def calcular(self, expresion):
        expr_descompuesta = self.parser.parse(expresion)
        if expr_descompuesta['operadores'][0] == '+':
           return str(self.calc.sumar(
                           expr_descompuesta['operandos'][0],
                           expr_descompuesta['operandos'][1]))
        elif expr_descompuesta['operadores'][0] == '-':
           return str(self.calc.restar(
                           expr_descompuesta['operandos'][0],
                           expr_descompuesta['operandos'][1]))
```

Y ahora si que nos ponemos con el test que habíamos identificado antes:

```
...
    def test_expresion_compleja_sin_parentesis_sin_precedencia(self):
        sc = supercalculadora.Supercalculadora(
                 expr_aritmetica.ExprAritmetica())
        self.failUnlessEqual("6", sc.calcular("5␣+␣4␣-␣3"))
```

Falla, así que nos ponemos con la implementación.

```
...
    def calcular(self, expresion):
        expr_descompuesta = self.parser.parse(expresion)
        res = 0
        for i in range(len(expr_descompuesta['operadores'])):
            if i == 0:
                res = expr_descompuesta['operandos'][0]
            if expr_descompuesta['operadores'][i] == '+':
                   res = self.calc.sumar(
                             res,
                             expr_descompuesta['operandos'][i + 1])
            if expr_descompuesta['operadores'][i] == '-':
                   res = self.calc.restar(
                             res,
                             expr_descompuesta['operandos'][i + 1])
        return str(res)
```

Otra vez funciona pero sigo preocupado por la precedencia de operadores. Creemos la prueba para esto y veamos como funciona.

```
...
    def test_expresion_compleja_sin_parentesis_con_precedencia(self):
        sc = supercalculadora.Supercalculadora(
                 expr_aritmetica.ExprAritmetica())
        self.failUnlessEqual("3", sc.calcular("5␣+␣4␣/␣2␣-␣4"))
```

Falla, lo que nos temíamos. Ahora tenemos que pensar cómo solucionamos este problema. Una primera idea es buscar los operadores más prioritarios y hacer la operación y así ir poco a poco simplificando la expresión. Lo hacemos pasar de la manera más sencilla (y fea... realmente fea) que se me ha ocurrido:

```
def calcular(self, expresion):
    expr_descompuesta = self.parser.parse(expresion)

    try:
        i = expr_descompuesta['operadores'].index('/')
        res_intermedio = self.calc.dividir(
                            expr_descompuesta['operandos'][i],
                            expr_descompuesta['operandos'][i + 1])
        expr_descompuesta = {'operandos':
                            [expr_descompuesta['operandos'][0],
                            res_intermedio,
                            expr_descompuesta['operandos'][3]],
                            'operadores':
                            [expr_descompuesta['operadores'][0],
                            expr_descompuesta['operadores'][2]]}
    except ValueError:
        pass

    res = 0
    for i in range(len(expr_descompuesta['operadores'])):
        if i == 0:
            res = expr_descompuesta['operandos'][0]
        if expr_descompuesta['operadores'][i] == '+':
               res = self.calc.sumar(
                           res, expr_descompuesta['operandos'][i + 1])
        if expr_descompuesta['operadores'][i] == '-':
               res = self.calc.restar(
                           res, expr_descompuesta['operandos'][i + 1])
    return str(res)
```

Da realmente miedo... Vamos a refactorizar primero y luego a añadir más pruebas para este caso ya que sabemos que hay muchos más casos que hay que probar. Movamos la simplificación de la expresión (evaluación intermedia) a otro método llamado simplificar. No es que sea mucho mejor pero es algo más legible. Con los nuevos tests, el código irá tomando mejor forma... o eso espero. Empezamos con los tests, luego con el código.

11.12: ut_supercalculadora.py

```
class TestsSupercalculadora(unittest.TestCase):
    def setUp(self):
        self.sc = supercalculadora.Supercalculadora(
                        expr_aritmetica.ExprAritmetica())

    def tearDown(self):
        pass

    def test_sumar(self):
        self.failUnlessEqual("4", self.sc.calcular("2␣+␣2"))

    def test_restar(self):
        self.failUnlessEqual("0", self.sc.calcular("2␣-␣2"))

    def test_expresion_compleja_sin_parentesis_sin_precedencia(self):
        self.failUnlessEqual("6", self.sc.calcular("5␣+␣4␣-␣3"))

```

```
18      def test_expresion_compleja_sin_parentesis_con_precedencia(self):
19          self.failUnlessEqual("3", self.sc.calcular("5␣+␣4␣/␣2␣-␣4"))
```

11.13: supercalculadora.py

```
1
2   def simplificar(self, expr_descompuesta):
3       expr_simplificada = {}
4       try:
5           i = expr_descompuesta['operadores'].index('/')
6           res_intermedio = self.calc.dividir(
7                                expr_descompuesta['operandos'][i],
8                                expr_descompuesta['operandos'][i + 1])
9           expr_simplificada = {'operandos':
10                               [expr_descompuesta['operandos'][0],
11                               res_intermedio,
12                               expr_descompuesta['operandos'][3]],
13                               'operadores':
14                               [expr_descompuesta['operadores'][0],
15                               expr_descompuesta['operadores'][2]]}
16      except ValueError:
17          expr_simplificada = expr_descompuesta
18
19      return expr_simplificada
20
21  def calcular(self, expresion):
22      expr_simplificada = self.simplificar(
23                              self.parser.parse(expresion))
24
25      res = 0
26      for i in range(len(expr_simplificada['operadores'])):
27          if i == 0:
28              res = expr_simplificada['operandos'][0]
29          if expr_simplificada['operadores'][i] == '+':
30                 res = self.calc.sumar(
31                             res, expr_simplificada['operandos'][i + 1])
32          if expr_simplificada['operadores'][i] == '-':
33                 res = self.calc.restar(
34                             res, expr_simplificada['operandos'][i + 1])
35      return str(res)
```

Seguimos con un nuevo test algo más complejo en el mismo área para mejorar nuestra implementación.

```
1   ...
2       def test_expresion_compleja_sin_parentesis_con_precedencia(self):
3           self.failUnlessEqual("3", self.sc.calcular("5␣+␣4␣/␣2␣-␣4"))
4           self.failUnlessEqual("-1", self.sc.calcular("4␣/␣2␣-␣3"))
```

La implementación para cumplir ambos tests ha quedado como sigue:

```
1
2   def simplificar(self, expr_descompuesta):
3       expr_simplificada = {}
4       try:
```

```
        i = expr_descompuesta['operadores'].index('/')
        res_intermedio = self.calc.dividir(
                              expr_descompuesta['operandos'][i],
                              expr_descompuesta['operandos'][i + 1])
        expr_simplificada = expr_descompuesta
        expr_simplificada['operadores'].pop(i)
        expr_simplificada['operandos'].pop(i)
        expr_simplificada['operandos'].pop(i)
        expr_simplificada['operandos'].insert(i, res_intermedio)
    except ValueError:
        expr_simplificada = expr_descompuesta

    return expr_simplificada

def calcular(self, expresion):
    expr_simplificada = self.simplificar(
                             self.parser.parse(expresion))

    res = 0
    for i in range(len(expr_simplificada['operadores'])):
        if i == 0:
            res = expr_simplificada['operandos'][0]
        if expr_simplificada['operadores'][i] == '+':
               res = self.calc.sumar(
                         res,
                         expr_simplificada['operandos'][i + 1])
        if expr_simplificada['operadores'][i] == '-':
               res = self.calc.restar(
                         res,
                         expr_simplificada['operandos'][i + 1])
    return str(res)
```

Y en el paso de refactorización podemos mejorar mucho las cosas moviendo la lógica a `simplificar` mientras resolvemos la expresión de una manera recursiva. Quizá, incluso, creemos algún otro método donde podamos mover alguna lógica dentro del algoritmo que estamos creando. Como tenemos pruebas unitarias será fácil hacer cambios más grandes en la refactorización sin que rompamos el funcionamiento del programa. En caso de que lo hagamos sin darnos cuenta, las pruebas unitarias nos alertarán de los errores cometidos y así los corregiremos fácilmente.

Aprovechamos la ocasión también para repasar los nombres de todos los métodos y mejorarlos para que todos muestren claramente la intención de la funcionalidad y de las pruebas.

11.14: supercalculadora.py

```

def __operar__(self, expr_descompuesta):
    i = None
    res_intermedio = 0
    if '/' in expr_descompuesta['operadores']:
        i = expr_descompuesta['operadores'].index('/')
        res_intermedio = self.calc.dividir(
```

```
8                               expr_descompuesta['operandos'][i],
9                               expr_descompuesta['operandos'][i + 1])
10      elif '-' in expr_descompuesta['operadores']:
11          i = expr_descompuesta['operadores'].index('-')
12          res_intermedio = self.calc.restar(
13                              expr_descompuesta['operandos'][i],
14                              expr_descompuesta['operandos'][i + 1])
15      elif '+' in expr_descompuesta['operadores']:
16          i = expr_descompuesta['operadores'].index('+')
17          res_intermedio = self.calc.sumar(
18                              expr_descompuesta['operandos'][i],
19                              expr_descompuesta['operandos'][i + 1])
20      else:
21          # Es un error, tenemos que decidir que hacer en los test
22          # siguientes
23          # Forzamos el error para que no haya problemas luego
24          assert False
25
26      return (i, res_intermedio)
27
28  def __simplificar__(self, expr_descompuesta):
29      if expr_descompuesta['operadores'] == []:
30          return expr_descompuesta
31
32      (i, res_intermedio) = self.__operar__(expr_descompuesta)
33      expr_simplificada = expr_descompuesta
34      expr_simplificada['operadores'].pop(i)
35      expr_simplificada['operandos'].pop(i)
36      expr_simplificada['operandos'].pop(i)
37      expr_simplificada['operandos'].insert(i, res_intermedio)
38
39      return self.__simplificar__(expr_simplificada)
40
41  def calcular(self, expresion):
42      return str(self.__simplificar__(
43                      self.parser.parse(expresion))['operandos'][0]
```

Creamos un par de test unitarios más para estar seguros de que la implementación funciona para casos más complejos o, en el peor de los casos, para cambiarla hasta que todas las pruebas pasen.

11.15: ut_supercalculadora.py

```
1   ...
2       def test_expresion_compleja_sin_parentesis_con_precedencia(self):
3           self.failUnlessEqual("3", self.sc.calcular("5␣+␣4␣/␣2␣-␣4"))
4           self.failUnlessEqual("-1", self.sc.calcular("4␣/␣2␣-␣3"))
5           self.failUnlessEqual("1", self.sc.calcular(
6                                       "4␣/␣2␣-␣3␣+␣1␣+␣6␣/␣3␣-␣1"))
7           self.failUnlessEqual("-8",
8                               self.sc.calcular(
9                                   "4␣/␣-2␣+␣3␣+␣-1␣+␣-6␣/␣-3␣-␣10"))
```

¡Qué suerte!, ¡luz verde de nuevo sin que hayamos tenido que cambiar nada! Eso nos da pie para seguir con otras nuevas pruebas para seguir incrementando la funcionalidad poco a poco. Pero, antes de eso, es el momento de actualizar la libreta otra vez más.

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  - ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0 - ¡HECHO!
- Restar 2 al número -5, devuelve -7 - ¡HECHO!
- Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
- "5 + 4 * 2 / 2", devuelve 9
- "2 - 2", devuelve 0 - ¡HECHO!
- "5 + 4 - 3", devuelve 6 - ¡HECHO!
- "5 + 4 / 2 - 4", devuelve 3 - ¡HECHO!
- "4 / 2 - 3 + 1 + 6 / 3 - 1", devuelve 1 - ¡HECHO!
- "4 / -2 + 3 + -1 + -6 / -3 - 10", devuelve -8 - ¡HECHO!
- Operandos son '5', '4', '2' y '2' y operadores '+', '*', '/' -
  ¡HECHO!
# Aceptación - "3 / 2", devuelve un error
- Dividir 2 entre 2 da 1 - ¡HECHO!
- Dividir 10 entre 5 da 2 - ¡HECHO!
- Dividir 10 entre -5 da -2 - ¡HECHO!
- Dividir -10 entre -5 da 2 - ¡HECHO!
- Dividir 3 entre 2 lanza una excepción - ¡HECHO!
- Dividir 3 entre 0 lanza una excepción - ¡HECHO!
- La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/' - ¡HECHO!
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande que se puede manejar?, ¿y el más
  pequeño?
```

Ya sólo nos queda una operación, la multiplicación. Vamos ir con ella poniendo nuestra atención en el test de aceptación 5 + 4 * 2 / 2. Antes de ponernos con esta expresión, vamos a ir a por algo más sencillo como 4

`* 2`, `-4 * 2`, `4 * -2` y `-4 * -2`.

Como siempre, seguimos la forma de trabajar TDD empezando por un test, haciendo la implementación, refactorizando, después tomando otro test, etc., repitiendo el algoritmo TDD paso a paso.

11.16: ut_calculadora.py

```
...
    def test_multiplicar_simple(self):
        self.failUnlessEqual(8, self.calc.multiplicar(4, 2))

    def test_multiplicar_negativa(self):
        self.failUnlessEqual(-8, self.calc.multiplicar(-4, 2))
        self.failUnlessEqual(-8, self.calc.multiplicar(4, -2))
        self.failUnlessEqual(8, self.calc.multiplicar(-4, -2))
```

11.17: calculadora.py

```
class Calculadora:
    def sumar(self, a, b):
        return a + b

    def restar(self, a, b):
        return a - b

    def multiplicar(self, a, b):
        return a * b

    def dividir(self, a, b):
        if a % b != 0:
            raise ValueError
        else:
            return a / b
```

Actualizamos la libreta antes de centrarnos en la expresión más compleja con suma, multiplicación y división.

# Aceptación - *"2 + 2", devuelve 4*

- *Sumar 2 al número 2, devuelve 4* - ¡HECHO!
- *La propiedad conmutativa se cumple* - ¡HECHO!
- *La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'* - ¡HECHO!

# Aceptación - *"5 - 3", devuelve 2*

- *Restar 5 al número 3, devuelve 2* - ¡HECHO!
- *Restar 2 al número 3, devuelve -1* - ¡HECHO!
- *La propiedad conmutativa no se cumple* - ¡HECHO!

# Aceptación - *"2 + -2", devuelve 0*

- *Sumar 2 al número -2, devuelve 0* - ¡HECHO!
- *Restar 2 al número -5, devuelve -7* - ¡HECHO!
- *Restar -2 al número -7, devuelve -5* - ¡HECHO!

# Aceptación - *"5 + 4 * 2 / 2", devuelve 9*

- *"5 + 4 * 2 / 2", devuelve 9*
- *"4 * 2", devuelve 8* - ¡HECHO!
- *4 * 2", devuelve -8* - ¡HECHO!
- *"4 * -2", devuelve -8* - ¡HECHO!
- *4 * -2", devuelve 8* - ¡HECHO!
- *"2 - 2", devuelve 0* - ¡HECHO!
- *"5 + 4 - 3", devuelve 6* - ¡HECHO!
- *"5 + 4 / 2 - 4", devuelve 3* - ¡HECHO!
- *"4 / 2 - 3 + 1 + 6 / 3 - 1", devuelve 1* - ¡HECHO!
- *"4 / -2 + 3 + -1 + -6 / -3 - 10", devuelve -8* - ¡HECHO!
- *Operandos son '5', '4', '2' y '2' y operadores '+', '*', '/'* - ¡HECHO!

# Aceptación - *"3 / 2", devuelve un error*

- *Dividir 2 entre 2 da 1* - ¡HECHO!
- *Dividir 10 entre 5 da 2* - ¡HECHO!
- *Dividir 10 entre -5 da -2* - ¡HECHO!
- *Dividir -10 entre -5 da 2* - ¡HECHO!
- *Dividir 3 entre 2 lanza una excepción* - ¡HECHO!
- *Dividir 3 entre 0 lanza una excepción* - ¡HECHO!
- *La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y '/'* - ¡HECHO!

# Aceptación - *"* * 4 - 2": devuelve un error*

# Aceptación - *"* 4 5 - 2": devuelve un error*

# Aceptación - *"* 4 5 - 2 : devuelve un error*

# Aceptación - *"*45-2-": devuelve un error*

- *¿Qué número es el más grande que se puede manejar?, ¿y el más pequeño?*

Ahora nos ponemos con la expresión pendiente como parte de la supercalculadora. Vamos con ello con el test que será parte de "expresión compleja con precedencia sin paréntesis".

11.18: ut_supercalculadora.py

```
1     def test_expresion_compleja_sin_parentesis_con_precedencia(self):
2         ...
3         self.failUnlessEqual("9", self.sc.calcular("5␣+␣4␣*␣2␣/␣2"))
```

Falla puesto que se va por la rama que no está implementada en `__operar__`. Nos ponemos con ello.

11.19: supercalculadora.py

```
1  def __operar__(self, expr_descompuesta):
2      i = None
3      res_intermedio = 0
4      if '/' in expr_descompuesta['operadores']:
5          i = expr_descompuesta['operadores'].index('/')
6          res_intermedio = self.calc.dividir(
7                                  expr_descompuesta['operandos'][i],
8                                  expr_descompuesta['operandos'][i + 1])
9      elif '*' in expr_descompuesta['operadores']:
10         i = expr_descompuesta['operadores'].index('*')
11         res_intermedio = self.calc.multiplicar(
12                                 expr_descompuesta['operandos'][i],
13                                 expr_descompuesta['operandos'][i + 1])
14
15     elif '-' in expr_descompuesta['operadores']:
16         i = expr_descompuesta['operadores'].index('-')
17         res_intermedio = self.calc.restar(
18                                 expr_descompuesta['operandos'][i],
19                                 expr_descompuesta['operandos'][i + 1])
20     elif '+' in expr_descompuesta['operadores']:
21         i = expr_descompuesta['operadores'].index('+')
22         res_intermedio = self.calc.sumar(
23                                 expr_descompuesta['operandos'][i],
24                                 expr_descompuesta['operandos'][i + 1])
25     else:
26         # Es un error, tenemos que decidir que hacer en los test
27         # siguientes
28         # Forzamos el error para que no haya problemas luego
29         assert False
30
31     return (i, res_intermedio)
```

Bien, ahora pasa y no hay mucho que refactorizar. Sin embargo no me siento muy seguro de que esta implementación sea correcta. Al fin y al cabo, tenemos precedencia y la división puede lanzar fácilmente una excepción si el resultado no es exacto... Voy a creear una prueba más que me de más confianza. Por ejemplo, `4 - -3 * 2 / 3 + 5`

11.20: ut_supercalculadora.py

```
...
    def test_expresion_compleja_sin_parentesis_con_precedencia(self):
        ...
        self.failUnlessEqual("11",
                             self.sc.calcular("4␣-␣-3␣*␣2␣/␣3␣+␣5"))
```

Vaya, ¡falla!. La división lanza una excepción, lo que nos temíamos. Si lo pensamos, tiene sentido ya que si damos prioridad a la división antes que a la multiplicación la división podría ser no natural cuando lo sería si multiplicamos primero. Vamos a cambiar la implementación y ver si hacemos pasar todos los tests.

11.21: supercalculadora.py

```
...
class Supercalculadora:
    def __init__(self, parser):
        self.calc = calculadora.Calculadora()
        self.parser = parser

    def __operar__(self, expr_descompuesta):
        i = None
        res_intermedio = 0
        if '*' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('*')
            res_intermedio = self.calc.multiplicar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '/' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('/')
            res_intermedio = self.calc.dividir(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '-' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('-')
            res_intermedio = self.calc.restar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '+' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('+')
            res_intermedio = self.calc.sumar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        else:
            # Es un error, tenemos que decidir que hacer en los test
            # siguientes
            # Forzamos el error para que no haya problemas luego
            assert False

        return (i, res_intermedio)
...
```

¡Ahora pasa!. Es el turno de la refactorización. El código no ha cambiado mucho y es aceptable, sin embargo, tengo mis dudas con respecto a los tests.

Tenemos un test con muchas pruebas dentro y quizá es tiempo de revisarlo y ver si podemos separarlo un poco.

11.22: ut_supercalculadora.py

```
def test_expresion_compleja_sin_parentesis_con_precedencia(self):
    self.failUnlessEqual("3", self.sc.calcular("5␣+␣4␣/␣2␣-␣4"))
    self.failUnlessEqual("-1", self.sc.calcular("4␣/␣2␣-␣3"))
    self.failUnlessEqual("1", self.sc.calcular(
                                  "4␣/␣2␣-␣3␣+␣1␣+␣6␣/␣3␣-␣1"))
    self.failUnlessEqual("-8",
                         self.sc.calcular(
                             "4␣/␣-2␣+␣3␣+␣-1␣+␣-6␣/␣-3␣-␣10"))
    self.failUnlessEqual("9", self.sc.calcular("5␣+␣4␣*␣2␣/␣2"))

def test_expr_compleja_todas_operaciones_sin_parentesis(self):
    self.failUnlessEqual("11",
                         self.sc.calcular("4␣-␣-3␣*␣2␣/␣3␣+␣5"))
```

Hemos sacado las expresiones (en este caso sólo una, por desgracia) que utilizan todas las operaciones sin paréntesis y son más propensas a dar errores en otra test que prueba expecíficamente este caso.

Actualizamos la libreta una última vez:

```
# Aceptación - "2 + 2", devuelve 4
- Sumar 2 al número 2, devuelve 4 - ¡HECHO!
- La propiedad conmutativa se cumple - ¡HECHO!
- La cadena "2 + 2", tiene dos números y un operador: '2', '2' y '+'
  - ¡HECHO!
# Aceptación - "5 - 3", devuelve 2
- Restar 5 al número 3, devuelve 2 - ¡HECHO!
- Restar 2 al número 3, devuelve -1 - ¡HECHO!
- La propiedad conmutativa no se cumple - ¡HECHO!
# Aceptación - "2 + -2", devuelve 0
- Sumar 2 al número -2, devuelve 0 - ¡HECHO!
- Restar 2 al número -5, devuelve -7 - ¡HECHO!
- Restar -2 al número -7, devuelve -5 - ¡HECHO!
# Aceptación - "5 + 4 * 2 / 2", devuelve 9
- "5 + 4 * 2 / 2", devuelve 9 - ¡HECHO!
- "4 - -3 * 2 / 3 + 5", devuelve 11 - ¡HECHO!
- "4 * 2", devuelve 8 - ¡HECHO!
- 4 * 2", devuelve -8 - ¡HECHO!
- "4 * -2", devuelve -8 - ¡HECHO!
- 4 * -2", devuelve 8 - ¡HECHO!
- "2 - 2", devuelve 0 - ¡HECHO!
- "5 + 4 - 3", devuelve 6 - ¡HECHO!
- "5 + 4 / 2 - 4", devuelve 3 - ¡HECHO!
- "4 / 2 - 3 + 1 + 6 / 3 - 1", devuelve 1 - ¡HECHO!
- "4 / -2 + 3 + -1 + -6 / -3 - 10", devuelve -8 - ¡HECHO!
- Operandos son '5', '4', '2' y '2' y operadores '+', '*', '/' -
  ¡HECHO!
# Aceptación - "3 / 2", devuelve un error
- Dividir 2 entre 2 da 1 - ¡HECHO!
- Dividir 10 entre 5 da 2 - ¡HECHO!
- Dividir 10 entre -5 da -2 - ¡HECHO!
- Dividir -10 entre -5 da 2 - ¡HECHO!
- Dividir 3 entre 2 lanza una excepción - ¡HECHO!
- Dividir 3 entre 0 lanza una excepción - ¡HECHO!
- La cadena "10 / -5", tiene dos números y un operador: '10', '-5' y
  '/' - ¡HECHO!
# Aceptación - "* * 4 - 2": devuelve un error
# Aceptación - "* 4 5 - 2": devuelve un error
# Aceptación - "* 4 5 - 2 : devuelve un error
# Aceptación - "*45-2-": devuelve un error
- ¿Qué número es el más grande permitido?, ¿y el más pequeño?
```

Para terminar, vamos a crear una última prueba con un *stub* y vamos a cambiar nuestro diseño en base a esto sin necesidad de implementar nada (o casi nada). No voy a explicar el uso de mocks y stubs ya que se ha explicado[5] y verá que es igual nada más que cambiando la sintaxis del lenguaje. Simplemente quiero mostrar una posible solución en Python usando *pymox*[6] para que vea que en este lenguaje se puede lograr lo mismo que en .NET y Java.

Pensemos ahora en las expresiones aritméticas. Por ahora suponemos que están bien y no tenemos ningún validador que detecte errores. En algún momento lo necesitaremos pero todavía es pronto para ponernos a ello. Sin embargo, queremos ver cómo se comportaría nuestro código si tuviéramos ese validador y que éste nos dijese que una expresión es invalida. Esto lo podemos hacer con un *stub*.

Vamos a crear un test usando *pymox* que simule la respuesta de `validar` cada vez que calculamos una expresión aritmética. Para esto, obviamente, necesitamos tomar ciertas decisiones en el diseño, como por ejemplo que la clase `Supercalculadora` recibirá un parámetro más que será el validador. Suponemos también que el validador va a responder con un booleano si la expresión es válida o no. En este punto, vamos a probar sólo que la expresión es inválida pero, como ya hemos dicho, tenemos que comprobar que el método es llamado.

El test quedaría como sigue:

11.23: ut_supercalculadora.py

```
1  ...
2     def test_validador_expresion_invalida_stub(self):
3           validador_stub =
4              validador_expr_aritmetica.ValidadorExprAritmetica()
5           validar_mock = mox.Mox()
6           validar_mock.StubOutWithMock(validador_stub, 'validar')
7           validador_stub.validar("2␣^␣3").AndReturn(False)
8           validar_mock.ReplayAll()
9           sc = supercalculadora.Supercalculadora(
10                        exp_aritmetica.ExprAritmetica(),
11                                validador_stub)
12          self.failUnlessRaises(SyntaxError, sc.calcular, "2␣^␣3")
13          validar_mock.UnsetStubs()
14          validar_mock.VerifyAll()
```

Esto falla puesto que el `Supercalculadora` sólo recibe un parámetro y no dos en su constructor y también porque no tenemos ni siquiera un esqueleto de la clase `Validador`. Pongámonos con las implementación corrigiendo poco a poco los errores.

[5] Ver Capítulo 6 en la página 95

[6] http://code.google.com/p/pymox/

11.24: supercalculadora.py

```
...
class Supercalculadora:
    def __init__(self, parser, validador):
        self.calc = calculadora.Calculadora()
        self.parser = parser
        self.validador = validador
...
```

11.25: validador_expr_aritmetica.py

```
class ValidadorExprAritmetica:
    def validar (self, expresion):
        True
```

Por último, en `ut_supercalculadora.py` tenemos que importar el nuevo fichero de validación.

En este punto vemos que la prueba aún falla porque la excepción por validación no se produce. Además, vemos que las demás pruebas se han roto ya que hemos cambiado el constructor (hemos añadido el validador como un nuevo argumento). Corrijamos primero el error de la última prueba y después pasemos a corregir (mejor dicho a actualizar) todas las demás.

11.26: supercalculadora.py

```
...
    def calcular(self, expresion):
        if not self.validador.validar(expresion):
            raise SyntaxError("La␣expresion␣no␣es␣valida")

        return str(self.__simplificar__(
                        self.parser.parse(expresion))['operandos'][0])
```

La prueba pasa ahora así que pongámonos a corregir las otras pruebas en el paso de refactorización. Como la llamada al constructor está en el `setUp` y este es llamado por todas las pruebas, solamente tenemos que cambiar el constructor aquí y todas las pruebas pasarán ya que vamos a utilizar la "implementación" real que siempre devuelve "True".

11.27: ut_supercalculadora.py

```
import mox
import unittest
import validador_expr_aritmetica as validador
import exp_aritmetica
import supercalculadora

class TestsSupercalculadora(unittest.TestCase):
    def setUp(self):
        self.sc = supercalculadora.Supercalculadora(
                        exp_aritmetica.ExprAritmetica(),
                        validador.ValidadorExprAritmetica())

```

```
13      def tearDown(self):
14          pass
15
16      def test_sumar(self):
17          self.failUnlessEqual("4", self.sc.calcular("2␣+␣2"))
18
19      def test_restar(self):
20          self.failUnlessEqual("0", self.sc.calcular("2␣-␣2"))
21
22      def test_expresion_compleja_sin_parentesis_sin_precedencia(self):
23          self.failUnlessEqual("6", self.sc.calcular("5␣+␣4␣-␣3"))
24
25      def test_expresion_compleja_sin_parentesis_con_precedencia(self):
26          self.failUnlessEqual("3", self.sc.calcular("5␣+␣4␣/␣2␣-␣4"))
27          self.failUnlessEqual("-1", self.sc.calcular("4␣/␣2␣-␣3"))
28          self.failUnlessEqual("1", self.sc.calcular(
29                                          "4␣/␣2␣-␣3␣+␣1␣+␣6␣/␣3␣-␣1"))
30          self.failUnlessEqual("-8",
31                               self.sc.calcular(
32                                   "4␣/␣-2␣+␣3␣+␣-1␣+␣-6␣/␣-3␣-␣10"))
33          self.failUnlessEqual("9", self.sc.calcular("5␣+␣4␣*␣2␣/␣2"))
34
35      def test_expr_compleja_todas_operaciones_sin_parentesis(self):
36          self.failUnlessEqual("11",
37                               self.sc.calcular("4␣-␣-3␣*␣2␣/␣3␣+␣5"))
```

Ahora es el momento de actualizar la libreta y hacer un nuevo `commit` al repositorio. Sí, digo "nuevo" porque durante este ejercicio deberíamos haber subido al repositorio el código (y las pruebas, que no olvidemos son parte del código) frecuentemente, por ejemplo cada vez que teníamos una nueva funcionalidad en verde después de unas cuantas pruebas.

Para clarificar todo el código, una vez más, aquí están todos los ficheros que hemos creado (menos las ya mencionados `ut_supercalculadora.py` y `validador_expr_aritmetica.py`).

11.28: ut_main.py

```
1  import unittest
2  import ut_calculadora
3  import ut_supercalculadora
4  import ut_expr_aritmetica
5
6  if __name__ == "__main__":
7      suite = unittest.TestSuite()
8      suite.addTest(unittest.makeSuite(
9                          ut_calculadora.TestsCalculadora))
10     suite.addTest(unittest.makeSuite(
11                         ut_expr_aritmetica.TestsExprAritmetica))
12     suite.addTest(unittest.makeSuite(
13                         ut_supercalculadora.TestsSupercalculadora))
14     unittest.TextTestRunner(verbosity=3).run(suite)
```

11.29: ut_calculadora.py

```
1  import unittest
```

```
2  import calculadora
3
4  class TestsCalculadora(unittest.TestCase):
5      def setUp(self):
6          self.calc = calculadora.Calculadora()
7
8      def tearDown(self):
9          pass
10
11     def test_sumar_numeros_iguales(self):
12         self.failUnlessEqual(4, self.calc.sumar(2, 2))
13
14     def test_sumar_numeros_distintos(self):
15         self.failUnlessEqual(12, self.calc.sumar(5, 7))
16
17     def test_sumar_propiedad_conmutativa(self):
18         self.failUnlessEqual(self.calc.sumar(5, 7),
19                              self.calc.sumar(7, 5))
20
21     def test_sumar_numeros_negativos(self):
22         self.failUnlessEqual(0, self.calc.sumar(2, -2))
23
24     def test_resta_positiva_numeros_distintos(self):
25         self.failUnlessEqual(2, self.calc.restar(5, 3))
26
27     def test_resta_negativa_numeros_distintos(self):
28         self.failUnlessEqual(-1, self.calc.restar(2, 3))
29
30     def test_restar_numeros_negativos(self):
31         self.failUnlessEqual(-7, self.calc.restar(-5, 2))
32         self.failUnlessEqual(-5, self.calc.restar(-7, -2))
33
34     def test_restar_no_propiedad_conmutativa(self):
35         self.failIfEqual(self.calc.restar(5, 3),
36                          self.calc.restar(3, 5))
37
38     def test_division_exacta(self):
39         self.failUnlessEqual(1, self.calc.dividir(2, 2))
40         self.failUnlessEqual(2, self.calc.dividir(10, 5))
41
42     def test_division_exacta_numeros_negativos(self):
43         self.failUnlessEqual(-2, self.calc.dividir(10, -5))
44         self.failUnlessEqual(2, self.calc.dividir(-10, -5))
45
46     def test_division_no_entera_da_excepcion(self):
47         self.failUnlessRaises(ValueError,
48                               self.calc.dividir, 3, 2)
49
50     def test_division_por_0(self):
51         self.failUnlessRaises(ZeroDivisionError,
52                               self.calc.dividir, 3, 0)
53
54     def test_multiplicar_simple(self):
55         self.failUnlessEqual(8, self.calc.multiplicar(4, 2))
56
57     def test_multiplicar_negativa(self):
58         self.failUnlessEqual(-8, self.calc.multiplicar(-4, 2))
59         self.failUnlessEqual(-8, self.calc.multiplicar(4, -2))
60         self.failUnlessEqual(8, self.calc.multiplicar(-4, -2))
```

11.30: ut_expr_aritmetica.py

```
import unittest
import expr_aritmetica

class TestsExprAritmetica(unittest.TestCase):
    def setUp(self):
        self.expresion = expr_aritmetica.ExprAritmetica()

    def tearDown(self):
        pass

    def test_extraer_operandos_y_operadores_en_2_mas_2(self):
        self.failUnlessEqual({'operandos': [2, 2],
                              'operadores': ['+']},
                             self.expresion.parse("2 + 2"))

    def test_extraer_operandos_y_operadores_expr_sin_ptesis(self):
        self.failUnlessEqual({'operandos': [5, 4, 2, 2],
                              'operadores': ['+', '*', '/']},
                             self.expresion.parse("5 + 4 * 2 / 2"))
```

11.31: supercalculadora.py

```
import exp_aritmetica
import calculadora

class Supercalculadora:
    def __init__(self, parser, validador):
        self.calc = calculadora.Calculadora()
        self.parser = parser
        self.validador = validador

    def __operar__(self, expr_descompuesta):
        i = None
        res_intermedio = 0
        if '*' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('*')
            res_intermedio = self.calc.multiplicar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '/' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('/')
            res_intermedio = self.calc.dividir(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '-' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('-')
            res_intermedio = self.calc.restar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
        elif '+' in expr_descompuesta['operadores']:
            i = expr_descompuesta['operadores'].index('+')
            res_intermedio = self.calc.sumar(
                             expr_descompuesta['operandos'][i],
                             expr_descompuesta['operandos'][i + 1])
```

```
33          else:
34              # Es un error, tenemos que decidir que hacer en los test
35              # siguientes
36              # Forzamos el error para que no haya problemas luego
37              assert False
38
39          return (i, res_intermedio)
40
41
42      def __simplificar__(self, expr_descompuesta):
43          if expr_descompuesta['operadores'] == []:
44              return expr_descompuesta
45
46          (i, res_intermedio) = self.__operar__(expr_descompuesta)
47
48          expr_simplificada = expr_descompuesta
49          expr_simplificada['operadores'].pop(i)
50          expr_simplificada['operandos'].pop(i)
51          expr_simplificada['operandos'].pop(i)
52          expr_simplificada['operandos'].insert(i, res_intermedio)
53
54          return self.__simplificar__(expr_simplificada)
55
56      def calcular(self, expresion):
57          if not self.validador.validar(expresion):
58              raise SyntaxError("La␣expresion␣no␣es␣valida")
59
60          return str(self.__simplificar__(
61                         self.parser.parse(expresion))['operandos'][0])
```

11.32: calculadora.py

```
1   class Calculadora:
2       def sumar(self, a, b):
3           return a + b
4
5       def restar(self, a, b):
6           return a - b
7
8       def multiplicar(self, a, b):
9           return a * b
10
11      def dividir(self, a, b):
12          if a % b != 0:
13              raise ValueError
14          else:
15              return a / b
```

11.33: expr_aritmetica.py

```
1   import string
2
3   class ExprAritmetica:
4       def __es_numero__(self, cadena):
5           try:
6               string.atoi(cadena)
7               return True
```

```
8           except ValueError:
9               return False
10
11      def parse(self, exp):
12          operandos = []
13          operadores = []
14          tokens = exp.split()
15          for token in tokens:
16              if self.__es_numero__(token):
17                  operandos.append(string.atoi(token))
18              else:
19                  operadores.append(token)
20          return {'operandos': operandos, 'operadores': operadores}
```

Lo último antes de acabar este capítulo, la salida después de correr todos los tests:

```
1  test_division_exacta
2      (ut_calculadora.TestsCalculadora) ... ok
3  test_division_exacta_numeros_negativos
4      (ut_calculadora.TestsCalculadora) ... ok
5  test_division_no_entera_da_excepcion
6      (ut_calculadora.TestsCalculadora) ... ok
7  test_division_por_0
8      (ut_calculadora.TestsCalculadora) ... ok
9  test_multiplicar_negativa
10     (ut_calculadora.TestsCalculadora) ... ok
11 test_multiplicar_simple
12     (ut_calculadora.TestsCalculadora) ... ok
13 test_resta_negativa_numeros_distintos
14     (ut_calculadora.TestsCalculadora) ... ok
15 test_resta_positiva_numeros_distintos
16     (ut_calculadora.TestsCalculadora) ... ok
17 test_restar_no_propiedad_conmutativa
18     (ut_calculadora.TestsCalculadora) ... ok
19 test_restar_numeros_negativos
20     (ut_calculadora.TestsCalculadora) ... ok
21 test_sumar_numeros_distintos
22     (ut_calculadora.TestsCalculadora) ... ok
23 test_sumar_numeros_iguales
24     (ut_calculadora.TestsCalculadora) ... ok
25 test_sumar_numeros_negativos
26     (ut_calculadora.TestsCalculadora) ... ok
27 test_sumar_propiedad_conmutativa
28     (ut_calculadora.TestsCalculadora) ... ok
29 test_extraer_operandos_y_operadores_en_2_mas_2
30     (ut_exp_aritmetica.TestsExpAritmetica) ... ok
31 test_extraer_operandos_y_operadores_expr_sin_ptesis
32     (ut_exp_aritmetica.TestsExpAritmetica) ... ok
33 test_expresion_compleja_sin_parentesis_con_precedencia
34     (ut_supercalculadora.TestsSupercalculadora) ... ok
35 test_expresion_compleja_sin_parentesis_sin_precedencia
36     (ut_supercalculadora.TestsSupercalculadora) ... ok
37 test_expr_compleja_todas_operaciones_sin_parentesis
38     (ut_supercalculadora.TestsSupercalculadora) ... ok
39 test_restar
40     (ut_supercalculadora.TestsSupercalculadora) ... ok
41 test_sumar
```

```
42      (ut_supercalculadora.TestsSupercalculadora) ... ok
43  test_validador_expresion_invalida_mock
44      (ut_supercalculadora.TestsSupercalculadora) ... ok
45
46  ----------------------------------------------------------------------
47  Ran 22 tests in 0.000s
48
49  OK
```

En este punto, tenemos una calculadora que hace todas las operaciones y lanza excepciones si hay error como habíamos diseñado. Además, tenemos una funcionalidad básica para manejar expresiones, por ahora, sin paréntesis y está preparada para seguir con el validador de expresiones.

Merece la pena mencionar que aunque la calculadora sea capaz de lanzar excepciones en caso de error y de que tengamos test unitarios para ello, no hemos hecho lo mismo a nivel de aplicación (para `Supercalculadora`). Es decir, ahora mismo la clase principal no sabe qué hacer (aunque debería mostrar un error) si la `Calculadora`, por ejemplo, lanzase una excepción. Esto hay que tratarlo con nuevos tests unitarios a nivel de `Supercalculadora` y con el test de aceptación que tenemos para este caso.

Todavía hay mucho trabajo que hacer para completar la Supercalculadora, muchas más pruebas y casos. El diseño cambiará cuando haya más pruebas (podemos intuir que expresión aritmética tendrá que usar un parser "de verdad" y quizá un *tokenizer*, etcétera) y lo que tengamos será, probablemente, bastante distinto a lo que hemos llegado ahora. Sin embargo, esperamos que este ejemplo haya mostrado como hacer TDD para crear la aplicación y le haya ayudado a entender como usar TDD con Python.

Capítulo 12

# Antipatrones y Errores comunes

Hay una amplia gama de antipatrones en que podemos incurrir cuando estamos practicando TDD. Este capítulo no pretende cubrirlos todos ni mucho menos, sino dar un pequeño repaso a los más comunes. Antes de pasar a ver antipatrones, es interesante citar los errores típicos que cometemos cuando empezamos a practicar TDD por primera vez.

## Errores del principiante

### El nombre del test no es suficientemente descriptivo

Recordemos que el nombre de un método y de sus parámetros son su mejor documentación. En el caso de un test, su nombre debe expresar con total claridad la intención del mismo.

### No sabemos qué es lo que queremos que haga el SUT

Nos hemos lanzado a escribir un test pero no sabemos en realidad qué es lo que el código bajo prueba tiene que hacer. En algunas ocasiones, lo resolvemos hablando con el dueño de producto y, en otras, hablando con otros desarrolladores. Tenga en cuenta que está tomando decisiones de diseño al escribir el test y que la programación por parejas o las revisiones de código nos ayudan en la toma de decisiones.

### No sabemos quién es el SUT y quién es el colaborador

En los tests de validación de interacción, pensamos que el colaborador, aquel que representamos mediante un doble, es el SUT. Antes de utilizar dobles de prueba tenemos que estar completamente seguros de quién es

el SUT y quién es el colaborador y qué es lo que queremos comprobar. En general, comprobamos que el SUT habla con el colaborador, o bien le decimos al colaborador que, si le hablan, responda algo que le decimos.

### Un mismo método de test está haciendo múltiples afirmaciones

Cuando practicamos TDD correctamente, apenas tenemos que usar el depurador. Cuando un test falla, lo encontramos directamente y lo corregimos en dos minutos. Para que esto sea así, cada método debe probar una única funcionalidad del SUT. A veces utilizamos varias afirmaciones (asserts) en el mismo test, pero sólo si giran en torno a la misma funcionalidad. Un método de test raramente excede las 10 líneas de código.

### Los test unitarios no están separados de los de integración

Los tests unitarios se ejecutan frecuentemente. De hecho, se ejecutan continuamente cuando practicamos TDD. Así que tenemos que conseguir que se ejecuten en menos de un segundo. Esta es la razón fundamental para tenerlos separados de los tests de integración

### Rápido, Inocuo, Atómico, Independiente

Si rompe alguna de sus propiedades, entonces no es un test unitario. Pregúntese si sus tests cumplen las reglas y, en caso de que no sea así, piense si con un poco más de esfuerzo puede conseguir que lo hagan.

### Se nos olvida refactorizar

No sólo por tener una gran batería de tests, el código ya es más fácil de mantener. Si el código no está limpio, será muy costoso modificarlo, y también sus tests. No olvide buscar y corregir código duplicado después de hacer pasar cada test. El código de los tests debe estar tan limpio como el código de producción.

### Confundir un mock con un stub

Cuando queremos que un objeto falso devuelva una respuesta programada en caso de que se le llame, usamos un stub. Cuando queremos confirmar que efectivamente la llamada a un método se produce, usamos un mock. Un mock es más restrictivo que un stub y puede aumentar la fragilidad del test. No obstante, en muchas ocasiones la mejor solución pasa por usar un mock[1].

[1] Revise el Capítulo 6 en la página 95 para más información

### No eliminamos código muerto

A veces, tras cambios en la lógica de negocio, queda código en desuso. Puede ser código de producción o pueden ser tests. Puesto que disponemos de un sistema de control de versiones que nos permite volver atrás si alguna vez volviese a hacer falta el código, debemos eliminarlo de la versión en producción. El código muerto induce a errores antes o después. Se suele menospreciar cuando se trata de tests pero, como ha visto, el código de los tests es tan importante como el código que prueban.

## Antipatrones

James Carr[2] recopiló una lista de antipatrones ayudado por la comunidad TDD que traduje en mi blog[3] y que ahora añado a esta sección. Los nombres que les pusieron tienen un carácter cómico y no son en absoluto oficiales pero su contenido dice mucho. Algunos de ellos ya están recogidos en los errores comentados arriba.

### El Mentiroso

Un test completo que cumple todas sus afirmaciones (asserts) y parece ser válido pero que cuando se inspecciona más de cerca, muestra que realmente no está probando su cometido en absoluto.

### Setup Excesivo

Es un test que requiere un montón de trabajo para ser configurado. A veces se usan varios cientos de líneas de código para configurar el entorno de dicho test, con varios objetos involucrados, lo cual nos impide saber qué es lo que se está probando debido a tanto "ruido".

### El Gigante

Aunque prueba correctamente el objeto en cuestión, puede contener miles de líneas y probar muchísimos casos de uso. Esto puede ser un indicador de que el sistema que estamos probando es un Objeto Dios[4]

[2] http://blog.james-carr.org
[3] La traducción ha sido mejorada para el libro porque en el blog está bastante mal
[4] http://en.wikipedia.org/wiki/God_object

### El Imitador

A veces, usar mocks puede estar bien y ser práctico pero otras, el desarrollador se puede perder imitando los objetos colaboradores. En este caso un test contiene tantos mocks, stubs y/o falsificaciones, que el SUT ni siquiera se está probando. En su lugar estamos probando lo que los mocks están devolviendo.

### El Inspector

Viola la encapsulación en un intento de conseguir el 100 % de cobertura de código y por ello sabe tanto del objeto a prueba que, cualquier intento de refactorizarlo, rompe el test.

### Sobras Abundantes

Es el caso en que un test crea datos que se guardan en algún lugar y otro test los reutiliza para sus propios fines. Si el "generador" de los datos se ejecuta después, o no se llega a ejecutar, el test que usa esos datos falla por completo.

### El Héroe Local

Depende del entorno de desarrollo específico en que fue escrito para poder ejecutarse. El resultado es que el test pasa en dicho entorno pero falla en cualquier otro sitio. Un ejemplo típico es poner rutas que son específicas de una persona, como una referencia a un fichero en su escritorio.

### El Cotilla Quisquilloso

Compara la salida completa de la función que se prueba, cuando en realidad sólo está interesado en pequeñas partes de ella. Esto se traduce en que el test tiene que ser continuamente mantenido a pesar de que los cambios sean insignificantes. Este es endémico de los tests de aplicaciones web. Ejemplo, comparar todo un html de salida cuando solo se necesita saber si el *title* es correcto.

### El Cazador Secreto

A primera vista parece no estar haciendo ninguna prueba por falta de afirmaciones (asserts). El test está en verdad confiando en que se lanzará una excepción en caso de que ocurra algún accidente desafortunado y que el framework de tests la capturará reportando el fracaso.

### El Escaqueado

Un test que hace muchas pruebas sobre efectos colaterales (presumiblemente fáciles de hacer) pero que nunca prueba el auténtico comportamiento deseado. A veces puede encontrarse en tests de acceso a base de datos, donde el método a prueba se llama, después el test selecciona datos de la base de datos y hace afirmaciones sobre el resultado. En lugar de comprobar que el método hace lo que debe, se está comprobando que dicho método no alteró ciertos datos o, lo que es lo mismo, que no causó daños.

### El Bocazas

Un test o batería de tests que llenan la consola con mensajes de diagnóstico, de log, de depuración, y demás forraje, incluso cuando los tests pasan. A veces, durante la creación de un test, es necesario mostrar salida por pantalla, y lo que ocurre en este caso es que, cuando se termina, se deja ahí aunque ya no haga falta, en lugar de limpiarlo.

### El Cazador Hambriento

Captura excepciones y no tiene en cuenta sus trazas, a veces reemplazándolas con un mensaje menos informativo, pero otras incluso registrando el suceso en un log y dejando el test pasar.

### El Secuenciador

Un test unitario que depende de que aparezcan, en el mismo orden, elementos de una lista sin ordenar.

### Dependencia Oculta

Un primo hermano del Héroe Local, un test que requiere que existan ciertos datos en alguna parte antes de correr. Si los datos no se rellenaron, el test falla sin dejar apenas explicación, forzando al desarrollador a indagar por acres de código para encontrar qué datos se suponía que debía haber.

### El Enumerador

Una batería de tests donde cada test es simplemente un nombre seguido de un número, ej, test1, test2, test3. Esto supone que la misión del test no queda clara y la única forma de averiguarlo es leer todo el test y rezar para que el código sea claro.

### El Extraño

Un test que ni siquiera pertenece a la clase de la cual es parte. Está en realidad probando otro objeto (X), muy probablemente usado por el que se está probando en la clase actual (objeto Y), pero saltándose la interacción que hay entre ambos, donde el objecto X debía funcionar en base a la salida de Y, y no directamente. También conocido como La Distancia Relativa.

### El Evangelista de los Sistemas Operativos

Confía en que un sistema operativo específico se está usando para ejecutarse. Un buen ejemplo sería un test que usa la secuencia de nueva línea de Windows en la afirmación (assert), rompiéndose cuando corre bajo Linux.

### El que Siempre Funciona

Se escribió para pasar en lugar de para fallar primero. Como desafortunado efecto colateral, sucede que el test siempre funciona, aunque debiese fallar.

### El Libre Albedrío

En lugar de escribir un nuevo test unitario para probar una nueva funcionalidad, se añade una nueva afirmación (assert) dentro de un test existente.

### El Unico

Una combinación de varios antipatrones, particularmente El Libre Albedrío y El Gigante. Es un sólo test unitario que contiene el conjunto entero de pruebas de toda la funcionalidad que tiene un objeto. Una indicación común de eso es que el test tiene el mismo nombre que su clase y contiene múltiples líneas de setup y afirmaciones.

### El Macho Chillón

Debido a recursos compartidos puede ver los datos resultantes de otro test y puede hacerlo fallar incluso aunque el sistema a prueba sea perfectamente válido. Esto se ha visto comúnmente en fitnesse, donde el uso de variables de clase estáticas, usadas para guardar colecciones, no se limpiaban adecuadamente después de la ejecución, a menudo repercutiendo de manera inesperada en otros tests. También conocido como El huésped no invitado.

### El Escabador Lento

Un test que se ejecuta de una forma increíblemente lenta. Cuando los desarrolladores lo lanzan, les da tiempo a ir al servicio, tomar café, o peor, dejarlo corriendo y marcharse a casa al terminar el día.

### Notas finales

Por último yo añadiría como antipatrón el hecho de ponerle comentarios a un test. Para mí, si un test necesita comentarios, es que no está bien escrito.

No se tome a pecho cada uno de los patrones ni los errores. Como en tantas otras áreas, las reglas tienen sus excepciones. El objetivo es que le sirvan para identificar "malos olores" en su práctica con TDD.

Apéndice A

# Integración Continua (CI)

## A.1. Introducción

### ¿Qué es la integración continua?

En palabras de Martin Fowler, entendemos la integración continua como: "Una práctica del desarrollo de software donde los miembros del equipo integran su trabajo con frecuencia: normalmente, cada persona integra de forma diaria, conduciendo a múltiples integraciones por día. Cada integración es comprobada por una construcción automática (incluyendo las pruebas) para detectar errores de integración tan rápido como sea posible. Muchos equipos encuentran que este enfoque conduce a la reducción significativa de problemas de integración y permite a un equipo desarrollar software cohesivo más rápidamente"

Muchos asocian la integración continua (utilizaré IC para referirme al término en adelante) con el uso de herramientas como CruiseControl[1] o Hudson[2], sin embargo, la IC puede practicarse sin el uso de estas, aunque con una mayor disciplina. En otras palabras, IC es mucho más que la utilización de una herramienta.

En algunos proyectos, la integración se lleva a cabo como un evento (cada lunes integramos nuestro código...), la práctica de la IC elimina esta forma de ver la integración, ya que forma parte de nuestro trabajo diario.

La IC encaja muy bien con prácticas como TDD dado que se centran en disponer de una buena batería de pruebas y en evolucionar el código realizando pequeños cambios a cada vez. Aunque, en realidad, la metodología de desarrollo no es determinante siempre y cuando se cumplan una serie de

[1] http://cruisecontrol.sourceforge.net/
[2] http://hudson-ci.org/

buenas prácticas.

La IC es independiente del tipo de metodología de gestión (ágil o predictiva), sin embargo, por los beneficios que aporta, proporciona gran valor a las metodologías ágiles, ayudando a tener un producto funcional en todo momento.

Los beneficios de hacer IC son varios, sin embargo, se entenderán y apreciarán mejor una vez que conozcamos las prácticas que conllevan.

### Conceptos

**Construcción (Build)**: una construcción implica algo más que compilar, podría consistir en compilar, ejecutar pruebas, usar herramientas de análisis de código[3], desplegar... entre otras cosas. Un build puede ser entendido como el proceso de convertir el código fuente en software que funcione.

**Scripts de construcción (build script)** : se trata de un conjunto de scripts que son utilizados para compilar, testear, realizar análisis del código o desplegar software. Podemos tener scripts de construcciones sin tener que implementar IC, sin embargo, para practicar IC son vitales.

### Empezando con IC

Ya se conoce de las bondades de un SCV (Sistema de Control de Versiones) para nuestro código fuente. La verdad, es que es difícil pensar en proyectos que no utilicen alguna herramienta de este tipo. Para realizar IC también es vital disponer de un repositorio centralizado cuya localización sea conocida por los miembros del equipo. Será nuestra base para realizar las construcciones cada vez que un desarrollador suba sus cambios.

El repositorio debe contener todo lo necesario para que nuestro proyecto pueda ser construido de forma automática, ya sean scripts, librerías de terceros, ficheros de configuración, etc.

### ¿Cómo es la vida con integración continua?

Imaginemos que decidimos agregar una pequeña funcionalidad a nuestro software. Comenzamos descargando el código actual del repositorio a nuestra máquina local, a esta copia la llamaremos copia local o working copy. En nuestra copia local, añadimos o modificamos el código necesario para realizar la funcionalidad elegida (no nos olvidemos de las pruebas asociadas). A continuación, debemos realizar una construcción de manera automática, esto podría consistir en compilar y ejecutar una batería de pruebas. Si hemos tenido éxito, lo siguiente que pensaremos es que ya estamos listos para subir

[3] http://pmd.sourceforge.net/

nuestros cambios al repositorio, sin embargo, otros desarrolladores han podido subir sus cambios mientras nosotros realizábamos nuestra tarea. Por lo tanto debemos bajarnos los cambios del repositorio, resolver los conflictos si los hubiera y lanzar de nuevo la construcción automática para verificar que todo ha sido integrado correctamente. Finalmente, podemos subir nuestros cambios al repositorio.

Pero nuestro trabajo no acaba aquí. Debemos construir una última vez pero, en este caso, en una "máquina de integración" basándonos en el código actual del repositorio (con nuestra nueva funcionalidad). Entre otras cosas, podría haber ocurrido que nos hayamos olvidado de subir un fichero y el repositorio no haya sido actualizado correctamente (este problema no es extraño). Si todo ha ido bien en la máquina de integración, hemos acabado nuestro trabajo. En caso contrario, debemos arreglar tan pronto como sea posible los problemas que hayamos podido ocasionar en el repositorio. De esta forma, disponemos de una base estable en el repositorio del cual cada desarrollador partirá para realizar su trabajo diario.

Llegados a este punto, es posible que piense que es un proceso muy latoso. No obstante, para su tranquilidad, comentaré que esta última parte en la máquina de integración podría ser ejecutada automáticamente por un servidor de IC como CruiseControl, Hudson, etc, al detectar que ha habido cambios en el repositorio. No se desespere, la integración continua promueve automatizar todo el proceso lo máximo posible para aumentar nuestra productividad. Más adelante entraremos en detalle, ahora he querido simplificar el proceso sin nuevos conceptos.

Ya tenemos lo básico para empezar a trabajar con IC. Nuestro proyecto en un repositorio centralizado, toda la información necesaria en él para construir el proyecto y unas nociones sobre la manera de trabajar. Pero esto es sólo la punta del iceberg, a continuación se pasará a detallar una serie de prácticas para realizar IC de forma efectiva.

## A.2. Prácticas de integración continua

### A.2.1. Automatizar la construcción

Nuestro trabajo, en parte, consiste en automatizar procesos para nuestros queridos usuarios. Sin embargo, a veces nos olvidamos de que una parte de nuestras tareas podrían ser automatizadas, concretamente, las tareas necesarias para obtener nuestro software a partir de ese montón de código fuente. Pero, cuando hablamos de construir nuestro software a partir de lo que existe en el repositorio, ¿a que nos referimos?. Construir significa mucho más que compilar, nuestra construcción podría descargar las últimas fuentes del

trunk[4], compilar, ejecutar pruebas automáticas, generar documentación o un esquema de base de datos si interaccionamos con un SGBD[5], etc. Iniciar una construcción debería ser tan fácil para cualquier desarrollador como lanzar un único comando desde una consola:

**freyes@dev:/home/project$build_now**

Existen herramientas de scripts de construcción libres que nos facilitan la labor, muchas son usadas en diversos proyectos open-source. Algunas de las más conocidas son Ant[6], NAnt[7], Maven[8], MSBuild[9], Rake[10], etc.

**Buenas prácticas**

- Divide el script de construcción en diferentes comandos para que cualquiera pueda lanzar una parte de forma aislada (por ejemplo, lanzar las pruebas), sin que pierda tiempo en realizar el proceso completamente.
- Normalmente, algunos desarrolladores usan un IDE para la construcción, estás herramientas son de gran utilidad para nuestra productividad pero es esencial poder construir nuestro software sin IDE alguno. Nos deben facilitar la vida pero no debemos caer en la dependencia absoluta.
- El proceso de construcción debería ir tan rápido como se pueda. Nadie lanzará el build para comprobar que la integración ha ido correctamente si la construcción es lenta. Dedica algo de tiempo para que sea lo más eficiente posible. Sin embargo, no todo es posible en el mundo $real^{TM}$... En este caso, podríamos plantearnos dividir la construcción en varias etapas. Una de las etapas, de cara al trabajo de cada desarrollador, podría compilar y ejecutar las pruebas unitarias. Las etapas que llevaran más tiempo podrían ser lanzadas únicamente en la máquina de integración (algunas incluso de forma paralela).

### A.2.2. Los test forman parte de la construcción

Ya se ha hecho hincapié, aunque sea de manera indirecta, en que los tests forman parte de la construcción. Sin embargo, no está de más reafirmar la

[4] http://svnbook.red-bean.com/nightly/en/svn-book.html#svn.branchmerge.maint.layout
[5] http://es.wikipedia.org/wiki/Sistema_de_gesti %C3 %B3n_de_bases_de_datos
[6] http://ant.apache.org/
[7] http://nant.sourceforge.net/
[8] http://maven.apache.org/
[9] http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx
[10] http://rake.rubyforge.org/

importancia de estos en el proceso. Es muy difícil tener una larga batería de test que prueben todas las partes del proyecto (100 % de cobertura) o que todas estas sean perfectas. Pero, como bien dice Martin Fowler: "Pruebas imperfectas, que corren frecuentemente, son mucho mejores que pruebas perfectas que nunca se han escrito". Aunque esto no supone que debamos dejar de mejorar nuestras habilidades para desarrollar pruebas de mejor calidad.

Es necesario automatizar la ejecución de las pruebas para que formen parte de la construcción. También es vital que el tiempo de ejecución de las pruebas sea corto (tanto en la máquina del desarrollador, como en la máquina de integración). Además, si se produce una larga demora notificando a las partes interesadas sobre lo que ha ocurrido en la construcción y los desarrolladores se centran en otras actividades, se pierde uno de los principales beneficios de la IC.

Tener pruebas que se ejecutan rápidamente es lo preferible pero no siempre podemos conseguirlo. Existen diferentes tipos de pruebas (unitarias, integración, sistema...) y todas son importantes para nuestro software pero el tiempo de ejecución suele ser más largo en unas que en otras (como podría ser el caso de las pruebas de sistemas). Llegados a este punto, si nuestra construcción se demora bastante (XP recomienda 10 minutos) podríamos ejecutar sólo las pruebas más rápidas (como suelen ser las unitarias) cada vez que el repositorio sea modificado y lanzar las restantes a intervalos.

Uno de los beneficios de la IC es que reduce los riesgos cuando llevemos nuestro sistema al entorno de producción pero, para ello, es necesario que las pruebas se ejecuten en un entorno lo más parecido al de producción. Cada diferencia es un riesgo más que no podremos verificar hasta la fase de instalación en producción. El uso de máquinas virtuales[11] para configurar estos entornos es una opción bastante acertada.

### Buenas prácticas

- Estructura el proyecto por cada tipo de test (test/unit <-> test/integration <-> test/system) así podrás ejecutar un grupo de manera independiente sin muchas complicaciones u otros ficheros de configuración
- Escribe también pruebas para los defectos/bugs que encuentres
- Realiza una comprobación (assert) por test. Además de dejar claro el objetivo del test, reducimos el ciclo para que los test pasen. Recuerda que si un assert falla, el test termina en ese punto sin dar información de las siguientes comprobaciones.

[11] http://es.wikipedia.org/wiki/M%C3%A1quina_virtual

### A.2.3. Subir los cambios de manera frecuente

Es una de las prácticas principales de la IC (implícita en la definición). Uno de los beneficios de realizar cambios frecuentemente en el repositorio es que nos fuerza a dividir nuestra tarea en otras más pequeñas, lo que proporciona una sensación de avance. Otro de los beneficios es que es más fácil detectar errores, puesto que en ese tiempo no habremos podido escribir mucho código, sólo habrá unos cuántos lugares donde el problema estará escondido.

En IC hay una máxima que debe ser cumplida por todo el equipo cuando se plantea subir cambios al repositorio: "Nunca se debe subir código que no funciona". Esto puede ser desde código que no compila hasta código que no pasa las pruebas. Para prevenir que ocurra esto, el desarrollador debe realizar una construcción en su entorno local. Puede ser que más adelante encontremos problemas en la máquina de integración pero, seguir esta máxima, produce un repositorio más estable.

Tampoco deberíamos partir de código del repositorio cuya construcción ha fallado. Esto podría llevarnos a duplicar esfuerzos (varios desarrolladores solucionando el mismo problema) o a pensar que nuestros cambios han sido lo que han provocado el problema, lo que supone una gran pérdida de tiempo hasta que lo descubramos.

Es responsabilidad de quien haya roto la construcción del repositorio, arreglarlo lo más rápidamente posible. Para ello, no deberíamos usar trucos para llevar a cabo nuestro objetivo, como eliminar o comentar el código de las pruebas que no pasan. Si el problema no es fácil de resolver y puede llevar tiempo, quizás debamos plantearnos revertir los cambios en el respositorio y solucionarlo tranquilamente en nuestro entorno local, de manera que no interfiramos en el trabajo del resto del equipo.

### A.2.4. Construir en una máquina de integración

Cuando utilizamos una máquina de integración donde realizamos las construcciones a partir del repositorio, reducimos las suposiciones sobre el entorno y la configuración y ayudamos a prevenir los problemas del tipo "¡En mi máquina funciona!"[12]. Hay 2 maneras de realizar esta tarea: ejecutar la construcción automática de forma manual o utilizar un servidor de IC.

La operación de forma manual es muy simple, el desarrollador que sube los cambios al repositorio, advierte al equipo para que no haya interferencias, se dirige a la máquina de integración y allí realiza la construcción a partir del repositorio.

Usando un servidor de IC, cada vez que alguien sube sus cambios al

[12] http://www.codinghorror.com/blog/archives/000818.html

repositorio, se realiza la construcción de manera automática, notificando del resultado del proceso (por e-mail, jabber, etc).

A priori puede parecer que el enfoque manual es una pérdida de tiempo, sin embargo, existen buenos argumentos[13] en contra del uso de servidores de IC.

### A.2.5. Todo el mundo puede ver lo que está pasando

Uno de los beneficios de la integración continua es que aporta claridad en el proyecto, cualquiera puede ver el estado del repositorio ojeando la máquina de integración. Pero lo correcto sería que cualquiera, desde el lugar donde se encuentre, pueda ver el estado del proyecto. Aquí es donde marca la diferencia el uso de un servidor de integración continua con respecto a realizar la integración de forma manual. Los servidores de IC, como por ejemplo Hudson, aportan bastante información respecto a la evolución del proyecto (cuántos builds se han fallado, cuántos han pasado, gráficos sobre la evolución, etc) y de los problemas concretos que ha producido cada cambio en el sistema. La mayoría de los servidores de IC proporcionan plugins o extensiones que enriquecen la información de nuestros proyectos (cobertura de las pruebas, generación automática de documentación a partir del código fuente, etc).

Además, los servidores de IC son capaces de comunicar por diferentes mecanismos lo que está ocurriendo en el servidor. Entre los mecanismos menos ostentosos se encuentra el envío de emails o sms al móvil de los interesados, los más frikies pueden hacer uso de lamparas de lava[14], semáforos[15], ambient orb[16], emisión de sonidos en un altavoz con el resultado de la última construcción, Nabaztag[17], etc.

También proporciona gran valor que cada persona, independientemente de su cargo o labor en el proyecto, pueda obtener el último ejecutable y ser capaz de arrancarlo sin muchos suplicios. Aunque a muchos les pueda parecer arriesgado este enfoque, entre los beneficios aportados se encuentran:

- Aumento del feedback a lo largo del proyecto entre todos los integrantes implicados en el mismo. Las buenas ideas diferencian nuestro software y las críticas constructivas nos hacen mejorar.
- Menor número de interrupciones para el equipo de desarrollo cada vez que alguien (ajeno al desarrollo diario) nos pida ver nuestro software funcionando. Ahí está, ¡pruébalo tú mismo!

[13] http://jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html
[14] http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBu
[15] http://wiki.hudson-ci.org/pages/viewpage.action?pageId=38633731
[16] http://weblogs.java.net/blog/kohsuke/archive/2006/11/diyorb_my_own_e.html
[17] http://wiki.hudson-ci.org/display/HUDSON/Nabaztag+Plugin

- Incremento del conocimiento sobre lo que hace nuestro software, qué funcionalidades cubre en cada momento y qué carencias tiene, a expensas de angustiosos y largos documentos de ánalisis

### A.2.6. Automatizar el despliegue

Todas las prácticas anteriores reducen el riesgo al llegar a la fase de despliegue (automatizar la construcción, automatizar las pruebas, tener un entorno lo más parecido al entorno de producción, etc). Si somos capaces de desplegar software en cualquier momento, estaremos aportando un valor inmenso a nuestros usuarios. Algunas personas afirman que, sin un despliegue exitoso, el software realmente no existe, pues nadie, sin ser el equipo de desarrollo, lo ha visto funcionando. El éxito del despliegue pasa por eliminar el mayor número de errores mediante la automatización.

Tener software que funciona en cualquier momento es un beneficio enorme, por no decir totalmente necesario, si usamos metodologías ágiles como Scrum, Crystal, etc, donde se producen pequeñas entregas al cliente de manera incremental a lo largo del proyecto.

#### Buenas prácticas

- Identifica cada despliegue en el repositorio. En subversion son conocidos como TAGs.
- Todas las pruebas deben ejecutarse y pasar correctamente para realizar el despliegue.
- Si el despliegue no ha sido correcto, añade capacidades para realizar una vuelta atrás (roll back) y dejar la última versión que funcionó.

## A.3. IC para reducir riesgos

La IC ayuda a identificar y reducir los riesgos existentes a lo largo del desarrollo de un proyecto. Pero, ¿de qué tipo de riesgos estamos hablando?. Podemos apreciar que la integración continua intenta minimizar el riesgo en los siguientes escenarios

### Despliegues demasiado largos y costosos

No es difícil encontrar proyectos que realizan la construcción del software en determinados entornos sólo cuando faltan unos pocos días para la entrega del mismo. Lo toman como algo eventual, una etapa que se realiza exclusivamente al final. Esto se suele traducir casi siempre en entregas intensas y

dolorosas para los miembros del equipo y, sobre todo, en una fase difícil de estimar. Algunos de los escenarios que hacen dura esta fase:

- No se ha podido verificar que toda la información para la construcción se encuentra disponible en el repositorio, y no en los equipos de los desarrolladores. La famosa frase "En mi equipo funciona" suele ser reflejo de este tipo de problemas
- Ausencia de una construcción automática

### Descubrir defectos demasiado tarde

Definir pruebas para nuestro software garantiza algo de calidad pero tan importante como 'hacerlas' es 'ejecutarlas' a diario con el fin de que saquen a relucir los defectos que podamos haber introducido.

### Falta de visibilidad del proyecto

¿Cuántas pruebas pasan con éxito?. ¿Seguimos los estándares planteados para el proyecto? ¿Cuándo se construyo el último ejecutable que funcionaba correctamente? Este tipo de preguntas suelen ser difíciles de responder si son hechas en cualquier momento sin previo aviso.

### Baja calidad del software

Desarrollar código de baja calidad suele producir un elevado coste en el tiempo, esta afirmación puede parecer gratuita para muchas personas pero el programador que haya regresado a modificar o leer esa parte del código que no huele bien[18], no la verá como tal. Disponer de herramientas que nos ayuden a analizar el código, pudiendo generar informes sobre asuntos tales como código duplicado, etc. suele ser de gran utilidad para identificar estos problemas y solventarlos antes de que nuestro proyecto se convierta en un enorme pantano.

## A.4. Conclusión

La integración continua es una práctica que ayuda a los equipos a mejorar la calidad en el ciclo de desarrollo. Reduce riesgos al integrar diariamente y genera software que puede ser desplegado en cualquier momento y lugar.

[18] http://en.wikipedia.org/wiki/Code_smell

Además, aporta mayor visibilidad sobre el estado del proyecto. Dado los beneficios que proporciona, vale la pena sacrificar algo de tiempo para que forme parte de nuestra caja de herramientas.

# Bibliografía


[1] Gojko Adzic. *Test Driven .NET Development with FitNesse*. Neuri Limited, 2008.

[2] Gojko Adzic. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. Neuri Limited, 2009.

[3] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[4] Bennington. *Ingeniería del Software: Un enfoque práctico (3ra Edición)*. Roger S. Presuman, 1956.

[5] Mike Cohn. *User Stories Applied*. Addison-Wesley Professional, 2004.

[6] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] Henrik Kniberg. *SCRUM y XP desde las Trincheras*. InfoQ, 2006.

[9] Lasse Koskela. *Test Driven*. Manning, 2007.

[10] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[11] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[12] Gerard Meszaros. *xUnit Test Patterns*. Addison-Wesley, 2007.

[13] Roberto Canales Mora. *Informática Profesional. Las reglas no escritas para triunfar en la empresa*. Starbook Editorial, 2009.

[14] Steve Freeman Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[15] Juan Palacio. *Flexibilidad con SCRUM*. Lulu.com, 2007.

[16] J. B. Rainsberg. *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications, 2004.