JUnit

EN ACCIÓN

Vincent Massol

con Ted Husted

Dotación

JUnit en acción

VICENTE MASSOL
con TED HUSTED

MANN I NG
Greenwich
(74 ° w. De largo)

Para obtener información en línea y solicitar este y otros libros de Manning, visite www.manning.com. El editor ofrece descuentos en este libro cuando se pide en cantidad. Para obtener más información, póngase en contacto:

Departamento de ventas especiales
Publicaciones Manning Co.
209 Bruce Park Avenue Fax: (203) 661-9018
Greenwich, CT 06830 correo electrónico: orders@manning.com

Publicaciones Manning Co. Corrector de estilo: Tiffany Taylor
209 Bruce Park Avenue Tipografista: Tony Roberts
Greenwich, CT 06830 Diseñador de portada: Leslie Haimes

ISBN 1-930110-99-5

Impreso en los Estados Unidos de América.

1 2 3 4 5 6 7 8 9 10 - VHG - 06 05 04 03

contenido

prefacio

Hasta la fecha, las pruebas siguen siendo la mejor solución que ha encontrado la humanidad para ofrecer software que funcione. Este libro es la suma de cuatro años de investigación y práctica en el campo de las pruebas. La práctica proviene de mi experiencia en consultoría de TI, primero en Octo Technology y luego en Pivolis; la investigación proviene de mi participación en el desarrollo de código abierto durante la noche y los fines de semana.

Desde mis primeros días de programación en 1982, he estado interesado en escribir herramientas para ayudar a los desarrolladores a escribir mejor código y desarrollar más rápidamente. Este interés me ha llevado a dominios como la tutoría de software y la mejora de la calidad. En estos días, estoy configurando plataformas de construcción continua y trabajando en las mejores prácticas de desarrollo, las cuales requieren conjuntos sólidos de pruebas. Cuanto más cerca estén estas pruebas de la actividad de codificación, más rápido obtendrá comentarios sobre su código; de ahí mi interés en las pruebas unitarias, que están tan cerca de la codificación que ahora es una parte tan importante del desarrollo como el código que se está escribiendo.

Esta experiencia me llevó a involucrarme en proyectos de código abierto relacionados con la calidad del software:

- Cactus para pruebas unitarias de componentes J2EE (http://jakarta.apache.org/cactus/)
- Simulacros de objetos para realizar pruebas unitarias en cualquier código (http://www.mockobjects.com/)
- Gump para compilaciones continuas (http://jakarta.apache.org/gump/)
- Maven para compilaciones y compilaciones continuas (http://maven.apache.org/)

- Prueba de concepto de prueba de patrones para el uso de programación orientada a aspectos (AOP) para verificar la arquitectura y las reglas de diseño (http://patterntesting.sf.net/). [1]

JUnit en acción es la conclusión lógica de esta participación.

Nadie quiere escribir código descuidado. Todos queremos escribir código que funcione, código del que podamos estar orgullosos. Pero a menudo nos distraemos de nuestras buenas intenciones. ¿Con qué frecuencia ha escuchado esto: “Queríamos escribir pruebas, pero estábamos bajo presión y no teníamos tiempo suficiente para hacerlo”; o, "Empezamos a escribir pruebas unitarias, pero después de dos semanas nuestro impulso cayó y con el tiempo dejamos de escribirlas".

Este libro le brindará las herramientas y técnicas que necesita para escribir código de calidad con gusto. Demuestra de forma práctica cómo utilizar las herramientas de forma eficaz, evitando los errores habituales. Le permitirá escribir código que funcione. Le ayudará a introducir las pruebas unitarias en su actividad de desarrollo diaria y a desarrollar un ritmo para escribir código robusto.

Sobre todo, este libro le mostrará cómo controlar la entropía de su software en lugar de ser controlado por él. Recuerdo algunos versos del escritor latino Lucrecio, quien, en 94-55 a. C. escribió en su *Sobre la naturaleza de las cosas (* Te ahorraré el texto original en latín):

> Es hermoso, cuando los vientos agitan las olas del gran mar, contemplar desde la tierra los grandes esfuerzos de otra persona; no porque sea un placer agradable que alguien esté en dificultades, sino porque es encantador darse cuenta de los problemas que usted mismo se ha librado.

Esta es exactamente la sensación que experimentará cuando sepa que está armado con un buen conjunto de pruebas. Verá a otros luchando y estará agradecido de tener pruebas para evitar que cualquier persona (incluido usted mismo) cause estragos en su aplicación.

Vincent Massol
Richeville (cerca de París), Francia

[1] Por mucho que quisiera, no he incluido un capítulo sobre código de prueba unitaria usando un marco AOP. Los frameworks AOP existentes son todavía jóvenes, y escribir pruebas unitarias con ellos conduce a un código detallado. Mi predicción es que los marcos especializados de AOP / pruebas unitarias aparecerán en un futuro muy cercano, y ciertamente los cubriré en una segunda edición. Consulte la siguiente entrada en mi blog sobre pruebas unitarias de un EJB con JUnit y AspectJ: http://blogs.codehaus.org/people/ vmassol / archives / 000138.html.

expresiones de gratitud

Este libro tardó un año en realizarse. Estoy eternamente agradecido con mi esposa, MarieAlbane, y mis hijos, Pierre-Olivier y Jean. Durante ese año, aceptaron que pasaba al menos la mitad de mi tiempo libre escribiendo el libro en lugar de estar con ellos. Tuve que prometer que no escribiría otro libro... por un tiempo....

Gracias a Ted Husted, quien dio un paso al frente y ayudó a que la primera parte del libro fuera más legible mejorando mi inglés, reorganizando los capítulos y agregando un poco de perejil aquí y allá donde había hecho atajos.

JUnit en acción no existiría sin Kent Beck y Erich Gamma, los autores de JUnit. Les agradezco su inspiración; y más especialmente, agradezco a Erich, que accedió a leer el manuscrito mientras estaba bajo presión para entregar Eclipse 2.1 y que presentó una bonita cita para el libro.

Una vez más, el libro no sería lo que es sin Tim Mackinnon y Steve Freeman, los creadores originales de la estrategia de prueba unitaria de objetos simulados, que es el tema de gran parte de este libro. Les agradezco que me hayan presentado a objetos simulados mientras bebo cerveza (¡para mí era jugo de arándano!) En el London Extreme Tuesday Club.

La calidad de este libro no sería la misma sin los revisores. Muchas gracias a Mats Henricson, Bob McWhirter, Erik Hatcher, William Brogden, Brendan Humphreys, Robin Goldsmith, Scott Stirling, Shane Mingins y Dorothy Graham. Me gustaría expresar un agradecimiento especial a Ilja Preuß, Kim Topley, Roger D. Cornejo y JB Rainsberger, quienes brindaron comentarios de revisión extremadamente completos y brindaron excelentes sugerencias.

Con este primer libro he descubierto el mundo de la edición. Me ha impresionado mucho el profesionalismo y la obsesión de Manning por la perfección. Siempre que pensaba que el libro estaba terminado y podía relajarme, ¡tenía que pasar por otra fase de verificaciones de algún tipo! Muchas gracias al editor Marjan Bace por su continua confianza a pesar de que seguí presionando la fecha de entrega. La editora de desarrollo Marilyn Smith fue un ejemplo de capacidad de respuesta, devolviéndome capítulos corregidos y sugerencias solo unas horas después de haber enviado los capítulos. La editora de estilo Tiffany Taylor corrigió una cantidad increíble de errores (casi no pude reconocer mis capítulos después de que Tiffany los revisara). Tony Roberts tuvo la difícil tarea de componer el libro y respaldar mis numerosas solicitudes; gracias, Tony.

Por último, pero no menos importante, un gran agradecimiento a Francois Hisquin, CEO de Octo Technology y Pivolis, las dos empresas para las que he estado trabajando mientras escribía. *JUnit en acción,* ¡por dejarme escribir algunas partes del libro durante el día!

sobre este libro

JUnit en acción es un libro de instrucciones basado en ejemplos sobre aplicaciones Java de pruebas unitarias, incluidas las aplicaciones J2EE, que utilizan el marco JUnit y sus extensiones. Este libro está dirigido a lectores que sean arquitectos de software, desarrolladores, miembros de equipos de prueba, gerentes de desarrollo, programadores extremos o cualquier persona que practique cualquier metodología ágil.

JUnit en acción se trata de resolver problemas difíciles del mundo real, como pruebas unitarias de aplicaciones heredadas, escribir pruebas reales para objetos reales, emplear métricas de prueba, automatizar pruebas, probar de forma aislada y más.

Características especiales

A lo largo del libro aparecen varias características especiales.

Mejores prácticas

La comunidad JUnit ya ha adoptado varias mejores prácticas. Cuando se introducen en el libro, un cuadro de llamada resume las mejores prácticas.

Patrones de diseño en acción

El marco JUnit pone en funcionamiento varios patrones de diseño conocidos. Cuando analizamos por primera vez un componente que hace un buen uso de un patrón de diseño, un cuadro de llamada define el patrón y señala su uso en el marco JUnit.

Directorio de software

A lo largo del libro, cubrimos cómo usar extensiones y herramientas con JUnit. Para su comodidad, las referencias a todos estos paquetes de software se han recopilado en un directorio en la sección de referencias al final de este libro. También se proporciona una bibliografía de otros libros que mencionamos en la sección de referencias.

Mapa vial

El libro está dividido en tres partes. La parte 1 es "JUnit destilado". Aquí, le presentamos las pruebas unitarias en general y JUnit en particular. La Parte 2, “Estrategias de prueba”, investiga diferentes formas de probar los objetos complejos que se encuentran en aplicaciones profesionales. La Parte 3, “Prueba de componentes”, explora estrategias para probar subsistemas comunes como servlets, filtros, páginas de JavaServer, bases de datos e incluso EJB.

Parte 1: JUnit destilado

El capítulo 1 explica cómo crear una prueba para un objeto simple. Presentamos los beneficios, la filosofía y la tecnología de las pruebas unitarias a lo largo del camino. A medida que las pruebas se vuelven más sofisticadas, presentamos JUnit como la solución para crear mejores pruebas.

El capítulo 2 profundiza en las clases, el ciclo de vida y la arquitectura de JUnit. Analizamos más de cerca las clases principales y el ciclo de vida general de JUnit. Para poner todo en contexto, miramos varias pruebas de ejemplo, como las que escribirías para tus propias clases.

El Capítulo 3 presenta un caso de prueba sofisticado para mostrar cómo funciona JUnit con componentes más grandes. El tema del estudio de caso es un componente que se encuentra en muchas aplicaciones: un controlador. Introducimos el código del estudio de caso, identificamos qué código probar y luego mostramos cómo probarlo. Una vez que sabemos que el código funciona como se esperaba, creamos pruebas para condiciones excepcionales, para asegurarnos de que el código se comporte bien incluso cuando las cosas van mal.

El Capítulo 4 analiza los diversos tipos de pruebas de software, el papel que desempeñan en el ciclo de vida de una aplicación, cómo diseñar para la capacidad de prueba y cómo practicar el desarrollo de prueba primero.

El Capítulo 5 explora las diversas formas en que puede integrar JUnit en su entorno de desarrollo. Buscamos automatizar JUnit con Ant, Maven y Eclipse.

Parte 2: Estrategias de prueba

El capítulo 6 describe cómo realizar pruebas unitarias utilizando stubs. Introduce una aplicación de muestra que se conecta a un servidor web y demuestra cómo realizar una prueba unitaria del método que llama a la URL remota mediante una técnica de código auxiliar.

El capítulo 7 muestra una técnica llamada objetos simulados que le permite probar un código de forma aislada de los objetos de dominio circundantes. Este capítulo continúa

con la aplicación de muestra (abrir una conexión HTTP a un servidor web); muestra cómo escribir pruebas unitarias para la aplicación y destaca las diferencias entre stubs y objetos simulados.

El Capítulo 8 demuestra otra técnica que es útil para realizar pruebas unitarias de componentes J2EE: las pruebas en contenedor. Este capítulo cubre cómo usar Cactus para ejecutar pruebas unitarias desde dentro del contenedor. Además, explicamos los pros y los contras de usar un enfoque dentro del contenedor versus un enfoque de objetos simulados, y cuándo usar cada uno.

Parte 3: Prueba de componentes

El Capítulo 9 muestra cómo realizar pruebas unitarias de servlets y filtros utilizando tanto el enfoque de objetos simulados como el enfoque dentro del contenedor. Destaca cómo se complementan entre sí y brinda estrategias sobre cuándo usarlos.

El capítulo 10 nos lleva al mundo de las JSP y taglibs de prueba unitaria. Muestra cómo utilizar los objetos simulados y las estrategias en el contenedor.

El capítulo 11 toca un tema difícil pero crucial: las aplicaciones de prueba unitaria que llaman a las bases de datos mediante JDBC. También demuestra cómo realizar pruebas unitarias del código de la base de datos de forma aislada de la base de datos.

El Capítulo 12 investiga cómo realizar pruebas unitarias de todo tipo de EJB utilizando objetos simulados, casos de prueba JUnit puros y Cactus.

Código

El código fuente de los ejemplos de este libro se ha donado a Apache Software Foundation. Está disponible en SourceForge (http://sourceforge.net/ projects / junitbook /). También se proporciona un enlace al código fuente desde la página web del libro en http://www.manning.com/massol. Consulte el apéndice A para obtener detalles sobre cómo está organizado el código fuente y los requisitos de la versión del software.

Los listados de código Java que presentamos tienen las palabras clave Java mostradas en negrita para que el código sea más legible. Además, cuando resaltamos cambios en una nueva lista, los cambios se muestran en negrita para llamar la atención sobre ellos. En ese caso, las palabras clave de Java se muestran en una fuente de código estándar que no está en negrita. A menudo, aparecen números y anotaciones en el código. Estos números se refieren a la discusión de la parte del código que sigue directamente a la lista.

En el texto, se utiliza una fuente monotipo para indicar código (JSP, Java y HTML), así como métodos Java, nombres de etiquetas JSP y la mayoría de los demás identificadores de código fuente:

- Es posible que una referencia a un método en el texto no incluya la firma porque puede haber más de una forma de la llamada al método.

- Una referencia a un elemento XML o etiqueta JSP en el texto generalmente no incluye las llaves ni los atributos.

Referencias

Las referencias bibliográficas se indican en notas a pie de página o en el cuerpo del texto. Los detalles completos de la publicación y / o las URL se proporcionan en la sección de referencias al final de este libro. Las URL de los sitios web se proporcionan en el texto del libro y tienen referencias cruzadas en el índice.

Autor en línea

Compra de *JUnit en acción* incluye acceso gratuito a un foro web privado administrado por Manning Publications, donde puede hacer comentarios sobre el libro, hacer preguntas técnicas y recibir ayuda del autor y de otros usuarios. Para acceder al foro y suscribirse a él, dirija su navegador web a http://www.manning.com/massol. Esta página proporciona información sobre cómo ingresar al foro una vez que esté registrado, qué tipo de ayuda está disponible y las reglas de conducta en el foro.

El compromiso de Manning con nuestros lectores es proporcionar un lugar donde pueda tener lugar un diálogo significativo entre los lectores individuales y entre los lectores y el autor. No es un compromiso con una cantidad específica de participación por parte del autor, cuya contribución al AO sigue siendo voluntaria (y no remunerada). Le sugerimos que intente hacerle al autor algunas preguntas desafiantes para que no se pierda su interés.

El foro Author Online y los archivos de discusiones anteriores serán accesibles desde el sitio web del editor siempre que el libro esté impreso.

Sobre los autores

Vincent Massol es el creador del marco Jakarta Cactus. También es un miembro activo de los equipos de desarrollo de Maven, Gump y MockObjects. Después de haber pasado cuatro años como arquitecto técnico en varios proyectos importantes (principalmente J2EE), Vincent es ahora el cofundador y CTO de Pivolis, una empresa especializada en aplicar metodologías ágiles al desarrollo de software offshore. Vincent, consultor y conferencista durante el día y desarrollador de código abierto durante la noche, vive actualmente en París, Francia. Puede ser contactado a través de su blog en http://blogs.codehaus.org/people/vmassol/.

Ted Husted es un miembro activo del equipo de desarrollo de Struts, gerente del Foro JGuru Struts y el autor principal de *Struts en acción.* [2] Como consultor, conferenciante y formador, Ted ha trabajado con equipos de desarrollo de Java en todo Estados Unidos. El último proyecto de desarrollo de Ted utilizó el desarrollo basado en pruebas en todo momento y está disponible como código abierto (http://sourceforge.net/projects/ wqdata /). Ted vive en Fairport, Nueva York, con su esposa, dos hijos, cuatro computadoras y un gato envejecido.

[2] Ted Husted, Cedric Dumoulin, George Franciscus y David Winterfeldt, *Struts en acción* (Greenwich, CT: Manning, 2002).

sobre el titulo

De Manning *en acción* Los libros combinan una descripción general con ejemplos prácticos para fomentar el aprendizaje. *y* recordando. La ciencia cognitiva nos dice que recordamos mejor a través del descubrimiento y la exploración. En Manning, pensamos en la exploración como "jugar". Cada vez que los informáticos crean una nueva aplicación, creemos que juegan con nuevos conceptos y nuevas técnicas, para ver si pueden hacer que el próximo programa sea mejor que el anterior. Un elemento esencial de una *en acción* libro es que se basa en ejemplos. *En acción* Los libros animan al lector a jugar con código nuevo y explorar nuevas ideas. En Manning, estamos convencidos de que el aprendizaje permanente se obtiene a través de la exploración, el juego y, lo más importante, *intercambio* lo que hemos descubierto con otros. La gente aprende mejor *en acción.*

Hay otra razón, más mundana, para el título de este libro: Nuestros lectores están ocupados. Usan libros para hacer un trabajo o resolver un problema. Necesitan libros que les permitan entrar y salir fácilmente, libros que les ayuden *en acción.*

Los libros de esta serie están diseñados para estos lectores "impacientes". Puedes empezar a leer un *en acción* reserve en cualquier momento, para aprender exactamente lo que necesita cuando lo necesita.

sobre la ilustración de la portada

La figura de la portada de *JUnit en acción* es un "Burco de Alpeo", tomado de un compendio español de costumbres de vestimenta regional publicado por primera vez en Madrid en 1799. La portada del libro dice:

> *Coleccion general de los Trages que usan actualmente todas las Nacionas del Mundo desubierto, dibujados y grabados con la mayor exactitud por RMVAR Obra muy util y en special para los que tienen el viajero universal*

que traducimos, lo más literalmente posible, así:

> *Colección general de trajes que se utilizan actualmente en las naciones del mundo conocido, diseñados e impresos con gran exactitud por RMVAR. Esta obra es muy útil sobre todo para quienes se pretenden ser viajeros universales.*

Aunque no se sabe nada de los diseñadores, grabadores y trabajadores que colorearon a mano esta ilustración, la “exactitud” de su ejecución es evidente en este dibujo, que es solo uno de los muchos de esta colorida colección. Su diversidad habla vívidamente de la singularidad e individualidad de las ciudades y regiones del mundo hace apenas 200 años. Este fue un momento en el que los códigos de vestimenta de dos regiones separadas por unas pocas docenas de millas identificaban a las personas únicamente como pertenecientes a una u otra. La colección da vida a una sensación de aislamiento y distanciamiento de ese período ‹y de cualquier otro período histórico excepto nuestro propio presente hipercinético. Los códigos de vestimenta han cambiado desde entonces y la diversidad por regiones, tan rica en ese momento, se ha desvanecido. Ahora a menudo es difícil distinguir al habitante de un continente de otro. Quizás,

tratando de verlo con optimismo, hemos cambiado una diversidad cultural y visual por una vida personal más variada. O una vida intelectual y técnica más variada e interesante.

En Manning celebramos la inventiva, la iniciativa y, sí, la diversión del negocio de las computadoras con portadas de libros basadas en la rica diversidad de la vida regional de hace dos siglos, revivida por las imágenes de esta colección.

En el momento de la publicación, no pudimos descifrar el significado de la leyenda "Burco de Alpeo", pero lo mantendremos informado sobre nuestro progreso en el *JUnit en acción* Página web. El primer lector que obtenga la traducción correcta recibirá una copia gratuita de otro libro de Manning de su elección. Realice publicaciones en el foro Author Online en www.manning.com/massol.

Parte 1

JUnit destilado

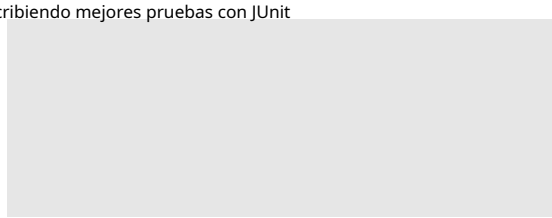
En la parte 1, ¡te infectarás de prueba! A través de un ejemplo simple, el capítulo 1 le enseñará qué es el marco JUnit y qué problemas resuelve. El Capítulo 2 lo llevará en un recorrido de descubrimiento de las clases principales de JUnit y cómo usarlas mejor. En el capítulo 3, practicarás tu nuevo conocimiento de JUnit en un ejemplo del mundo real. También aprenderá cómo configurar un proyecto JUnit y cómo ejecutar las pruebas unitarias. El Capítulo 4 da un paso atrás y explica por qué las pruebas unitarias son importantes y cómo encajan en el ecosistema de pruebas global. También presenta la metodología de desarrollo basado en pruebas y proporciona orientación sobre cómo medir la cobertura de la prueba. El Capítulo 5 demuestra cómo automatizar las pruebas unitarias utilizando tres herramientas populares: Eclipse, Ant y Maven.

Al final de la parte 1, tendrá un buen conocimiento general de JUnit, cómo escribir pruebas unitarias y cómo ejecutarlas fácilmente. Estará listo para comenzar a aprender sobre las diferentes estrategias necesarias para realizar pruebas unitarias de aplicaciones completas: stubs, objetos simulados y pruebas en el contenedor.

1 JUnit Jumpstart

Este capítulo cubre

- Escribir pruebas sencillas a mano
- Instalación de JUnit y ejecución de pruebas
- Escribiendo mejores pruebas con JUnit



Nunca en el campo del desarrollo de software tantos debieron tanto a tan pocas líneas de código.

- Martin Fowler

Se prueba todo el código.

Durante el desarrollo, lo primero que hacemos es ejecutar la "prueba de aceptación" de nuestro propio programador. Codificamos, compilamos y ejecutamos. Y cuando corremos, probamos. La "prueba" puede simplemente hacer clic en un botón para ver si aparece el menú esperado. Pero, aún así, todos los días codificamos, compilamos, ejecutamos ... y *probamos.*

Cuando realizamos pruebas, a menudo encontramos problemas, especialmente en la primera ejecución. Así que codificamos, compilamos, ejecutamos y probamos nuevamente.

La mayoría de nosotros desarrollamos rápidamente un patrón para nuestras pruebas informales: agregamos un registro, vemos un registro, editamos un registro y eliminamos un registro. Ejecutar un pequeño conjunto de pruebas como este a mano es bastante fácil de hacer; así que lo hacemos. *Una y otra vez.*

A algunos programadores les gusta hacer este tipo de pruebas repetitivas. Puede ser un descanso agradable de la reflexión profunda y la codificación dura. Y cuando nuestras pequeñas pruebas de click-through finalmente tienen éxito, hay una sensación real de logro: *¡Eureka! ¡Lo encontré!*

A otros programadores no les gusta este tipo de trabajo repetitivo. En lugar de ejecutar la prueba a mano, prefieren crear un pequeño programa que ejecute la prueba automáticamente. El código de prueba de juego es una cosa; ejecutar pruebas automatizadas es otra.

Si es un desarrollador de “prueba de juegos”, este libro es para usted. ¡Le mostraremos cómo crear pruebas automatizadas puede ser fácil, efectivo e incluso divertido!

Si ya está "infectado de prueba", ¡este libro también es para usted! Cubrimos los conceptos básicos en la parte 1 y luego pasamos a los problemas difíciles de la vida real en las partes 2 y 3.

1.1 Demostrar que funciona

Algunos desarrolladores sienten que las pruebas automatizadas son una parte esencial del proceso de desarrollo: un componente no puede ser *probado* para trabajar hasta que pase una serie completa de pruebas. De hecho, dos desarrolladores sintieron que este tipo de "pruebas unitarias" era tan importante que merecía su propio marco. En 1997, Erich Gamma y Kent Beck crearon una unidad de prueba simple pero efectiva *estructura* para Java, llamado JUnit. El trabajo siguió el diseño de un marco anterior que Kent Beck creó para Smalltalk, llamado SUnit.

DEFINICIÓN *estructura* —Un marco es una aplicación semi-completa.[1] Un marco proporciona una estructura común reutilizable que se puede compartir entre aplicaciones. Los desarrolladores incorporan el marco en su propia aplicación y lo amplían para satisfacer sus necesidades específicas. Los marcos se diferencian de los conjuntos de herramientas al proporcionar una estructura coherente, en lugar de un simple conjunto de clases de servicios públicos.

Si reconoce esos nombres, es por una buena razón. Erich Gamma es bien conocido como uno de los "Gang of Four" que nos dio el ahora clásico *Patrones de diseño* libro.[2] Kent Beck es igualmente conocido por su trabajo pionero en la disciplina del software conocida como Programación Extrema (http://www.extremeprogramming.org).

JUnit (junit.org) es un software de código abierto, publicado bajo la Common Public License Versión 1.0 de IBM y alojado en SourceForge. La Licencia Pública Común es favorable a los negocios: las personas pueden distribuir JUnit con productos comerciales sin mucha burocracia o restricciones.

JUnit se convirtió rápidamente en el marco estándar de facto para desarrollar pruebas unitarias en Java. De hecho, el modelo de prueba subyacente, conocido como xUnit, está en camino de convertirse en el marco estándar para *ningún* idioma. Hay marcos de xUnit disponibles para ASP, C ++, C #, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk y Visual Basic, ¡solo por nombrar algunos!

Por supuesto, el equipo de JUnit no inventó las pruebas de software ni siquiera la prueba unitaria. Originalmente, el término *prueba de unidad* describió una prueba que examinó el comportamiento de un solo *unidad de trabajo.*

Con el tiempo, el uso del término *prueba de unidad* ampliado. Por ejemplo, IEEE ha definido la prueba unitaria como "Prueba de unidades individuales de hardware o software *o grupos de unidades relacionadas* " (énfasis añadido).[3]

En este libro usamos el término *prueba de unidad* en el sentido más estricto de una prueba que examina una sola unidad aislada de otras unidades. Nos centramos en el tipo de prueba pequeña e incremental que los programadores aplican a su propio código. A veces estos se llaman *pruebas de programador* para diferenciarlos de las pruebas de garantía de calidad o las pruebas de clientes (http://c2.com/cgi/wiki?ProgrammerTest).

[1] Ralph Johnson y Brian Foote, "Diseño de clases reutilizables", *Revista de programación orientada a objetos* 1.5 (junio / julio de 1988): 22–35; http://www.laputan.org/drc/drc.html.

[2] Erich Gamma y col., *Patrones de diseño (* Reading, MA: Addison-Wesley, 1995).

[3] *Diccionario informático estándar IEEE: una compilación de glosarios informáticos estándar IEEE (* Nueva York: IEEE, 1990).

Aquí hay una descripción genérica de una prueba unitaria típica desde nuestra perspectiva: "Confirme que el método acepta el rango esperado de entrada y que el método devuelve el valor esperado para cada entrada de prueba".

Esta descripción nos pide que probemos el comportamiento de un método a través de su interfaz. Si le damos valor *X,* ¿Devolverá valor? *y?* Si le damos valor *z* en su lugar, ¿lanzará la excepción adecuada?

DEFINICIÓN *prueba de unidad* —Una prueba unitaria examina el comportamiento de una *unidad de trabajo.* Dentro de una aplicación Java, la "unidad de trabajo distinta" es a menudo (pero no siempre) un método único. Por el contrario, *pruebas de integración* y *prueba de aceptacion* examinar cómo interactúan varios componentes. A *unidad de trabajo* es una tarea que no depende directamente de la realización de ninguna otra tarea.

Las pruebas unitarias a menudo se enfocan en probar si un método sigue los términos de su *Contrato API.* Al igual que un contrato escrito por personas que aceptan intercambiar ciertos bienes o servicios en condiciones específicas, un contrato API se considera un acuerdo formal realizado por la interfaz de un método. Un método requiere que quienes llaman proporcionen objetos o valores específicos y, a cambio, devolverán ciertos objetos o valores. Si el contrato no se puede cumplir, entonces el método arroja una excepción para significar que el contrato no se puede mantener. Si un método no funciona como se esperaba, entonces decimos que el método ha roto su contrato.

DEFINICIÓN *Contrato API* —Una vista de una interfaz de programación de aplicaciones (API) como un acuerdo formal entre la persona que llama y la persona que llama. A menudo, las pruebas unitarias ayudan a definir el contrato de API al demostrar el comportamiento esperado. La noción de un contrato API se deriva de la práctica de *Diseño por contrato,* popularizado por el lenguaje de programación Eiffel (http://archive.eiffel.com/doc/manuals/technology/contract).

En este capítulo, veremos cómo crear una prueba unitaria para una clase simple desde cero. Comenzaremos escribiendo algunas pruebas manualmente, para que pueda ver cómo *usado* para hacer cosas. Luego, implementaremos JUnit para mostrarle cómo las herramientas adecuadas pueden simplificar la vida.

1.2 Empezando desde cero

Digamos que acaba de escribir el Calculadora clase que se muestra en el listado 1.1.

```
clase pública Calculadora
{
    público doble agregar( doble numero 1, doble Número 2) {

        regreso número1 + número2;
    }
}
```

Aunque no se muestra la documentación, el propósito previsto del Calculadora 's agregar (doble, doble) el método es tomar dos doble sy devuelve la suma como doble. El compilador puede decirle que se compila, pero también debe asegurarse de que funcione en tiempo de ejecución. Un principio fundamental de las pruebas unitarias es: "Cualquier función del programa sin una prueba automatizada simplemente no existe". [4] El agregar El método representa una característica fundamental de la calculadora. Tiene un código que supuestamente implementa la función. Lo que falta es una prueba automatizada que demuestre que su implementación funciona.

¿Pero no es el método de adición "demasiado simple para posiblemente romper"?

La implementación actual de la agregar El método es demasiado simple para romperlo. Si agregar fuera un método de utilidad menor, es posible que no lo pruebe directamente. En ese caso, si agregar fallaron, luego las pruebas de los métodos que usaban agregar fallaría. El agregar El método se probaría indirectamente, pero de todos modos.

En el contexto del programa de la calculadora, agregar no es solo un método, es un *función del programa.* Para tener confianza en el programa, la mayoría de los desarrolladores esperaría que hubiera una prueba automatizada para la función de agregar, sin importar cuán simple parezca la implementación.

En algunos casos, puede probar las características del programa mediante pruebas funcionales automáticas o pruebas de aceptación automáticas. Para obtener más información sobre las pruebas de software en general, consulte el capítulo 4.

Sin embargo, probar algo en este punto parece problemático. Ni siquiera tiene una interfaz de usuario con la que ingresar un par de doble s. Podrías escribir un pequeño programa de línea de comandos que esperara a que escribieras dos doble valores y luego muestra el resultado. Por supuesto, también estaría probando su propia capacidad para escribir un número y agregar el resultado nosotros mismos. Esto es mucho más de lo que quieres hacer. Solo desea saber si esta "unidad de trabajo" en realidad sumará dos doble sy devuelva la suma correcta. ¡No necesariamente quiere probar si los programadores pueden escribir números!

[4] Kent Beck, *Explicación de la programación extrema: Acepte el cambio (* Reading, MA: Addison-Wesley, 1999).

Mientras tanto, si va a hacer el esfuerzo de probar su trabajo, también debe tratar de preservar ese esfuerzo. Es bueno saber que el agregar (doble, doble) El método funcionó cuando lo escribiste. Pero lo que realmente desea saber es si el método funciona cuando envía el resto de la aplicación.

Como se muestra en la figura 1.1, si juntamos estos dos requisitos, se nos ocurre la idea de escribir un programa de prueba simple para el método. El programa de prueba podría pasar valores conocidos al método y ver si el resultado coincide con nuestras expectativas.

Figura 1.1 Justificación de JUnit: Unir los dos requisitos de prueba da una idea para un programa de prueba simple.

También puede ejecutar el programa nuevamente más tarde para asegurarse de que el método continúe funcionando a medida que la aplicación crece.

Entonces, ¿cuál es el programa de prueba más simple que podrías escribir? ¿Qué tal lo simple? TestCalculator programa que se muestra en el listado 1.2?

```
clase pública TestCalculator
{
    vacío estático público main (String [] args) {

        Calculadora calculadora = nuevo Calculadora();
        doble resultado = calculadora.add (10,50);
        si ( resultado! = 60) {

            System.out.println ("Mal resultado:" + resultado);
        }
    }
}
```

La primera TestCalculator es realmente simple! Crea una instancia de Calculadora, le pasa dos números y comprueba el resultado. Si el resultado no cumple con sus expectativas, imprime un mensaje en la salida estándar.

Si compila y ejecuta este programa ahora, la prueba pasará silenciosamente y todo parecerá estar bien. Pero, ¿qué pasa si cambias el código para que falle? Tendrá que observar atentamente la pantalla para ver si aparece el mensaje de error. Puede que no tenga que proporcionar la entrada, pero aún está probando su propia capacidad para monitorear la salida del programa. ¡Quieres probar el código, no tú mismo!

La forma convencional de manejar las condiciones de error en Java es lanzar una excepción. Dado que fallar la prueba es una condición de error, intentemos lanzar una excepción en su lugar.

Mientras tanto, es posible que también desee ejecutar pruebas para otros Calculadora métodos que aún no has escrito, como sustraer o multiplicar. Pasar a un diseño más modular facilitaría la captura y el manejo de excepciones y facilitaría la ampliación del programa de prueba más adelante. El Listado 1.3 muestra una TestCalculator programa.

```
clase pública TestCalculator
{
    int privado nbErrors = 0;

    vacío público testAdd ()
    {
        Calculadora calculadora = nuevo Calculadora();
        doble resultado = calculadora.add (10, 50);
        si ( resultado! = 60) {

            tirar nuevo RuntimeException ("Mal resultado:" + resultado);
        }
    }

    vacío estático público main (String [] args) {

        TestCalculator test = nuevo TestCalculator ();
        intentar
        {
            test.testAdd ();
        }
        captura ( Desechable e)
        {
            test.nbErrors ++;
            e.printStackTrace ();
        }

        si ( test.nbErrors> 0) {

            tirar nuevo RuntimeException ("Hubo" + test.nbErrors
                + "error (s)");
        }
    }
}
```

B

C

Trabajando desde el listado 1.3, en B mueve la prueba a su propio método. Ahora es más fácil concentrarse en lo que hace la prueba. También puede agregar más métodos con más unidades pruebas más tarde, sin hacer que el bloque principal sea más difícil de mantener. A C, tu cambias el principal bloque para imprimir un seguimiento de la pila cuando se produce un error y luego, si hay cualquier error, para lanzar una excepción de resumen al final.

1.3 Comprensión de los marcos de pruebas unitarias

Existen varias prácticas recomendadas que deben seguir los marcos de pruebas unitarias. Estas mejoras aparentemente menores en el TestCalculator El programa destaca tres reglas que (en nuestra experiencia) todos los marcos de pruebas unitarias deben observar:

- Cada prueba unitaria debe ejecutarse independientemente de todas las demás pruebas unitarias.
- Los errores deben detectarse y notificarse prueba a prueba.
- Debe ser fácil definir qué pruebas unitarias se ejecutarán.

El programa de prueba "ligeramente mejor" se acerca a seguir estas reglas, pero aún se queda corto. Por ejemplo, para que cada prueba unitaria sea realmente independiente, cada una debe ejecutarse en una instancia de cargador de clases diferente.

Agregar una clase también es solo un poco mejor. Ahora puede agregar nuevas pruebas unitarias agregando un nuevo método y luego agregando un trata de atraparlo bloquear a principal.

Definitivamente un paso adelante, pero aún por debajo de lo que querrías en un *verdadero* suite de pruebas unitarias. El problema más obvio es que trata de atraparlo Se sabe que los bloques son pesadillas de mantenimiento. ¡Podría dejar fácilmente una prueba de unidad y nunca saber que no se estaba ejecutando!

Sería bueno si pudiera agregar nuevos métodos de prueba y terminar con eso. Pero, ¿cómo sabría el programa qué métodos ejecutar?

Bueno, podría tener un procedimiento de registro simple. Un método de registro al menos haría un inventario de las pruebas que se están ejecutando.

Otro enfoque sería utilizar Java *reflexión* y *introspección* capacidades. Un programa podría verse a sí mismo y decidir ejecutar los métodos que se nombran de cierta manera, como los que comienzan con las letras *prueba,* por ejemplo.

Facilitar la adición de pruebas (la tercera regla de nuestra lista anterior) suena como otra buena regla para un marco de pruebas unitarias.

El código de soporte para realizar esta regla (mediante registro o introspección) no sería trivial, pero valdría la pena. Habría mucho trabajo por adelantado, pero ese esfuerzo valdrá la pena cada vez que agregue una nueva prueba.

Felizmente, el equipo JUnit te ha ahorrado la molestia. El marco JUnit ya admite métodos de registro o introspección. También admite el uso de una *cargador de clases* instancia para cada prueba, e informa todos los errores caso por caso.

Ahora que tiene una mejor idea de por qué necesita marcos de prueba unitarios, configuremos JUnit y veámoslo en acción.

1.4 Configuración de JUnit

JUnit viene en forma de archivo jar (junit.jar). Para usar JUnit para escribir las pruebas de su aplicación, simplemente deberá agregar el junit jar a la ruta de clase de compilación de su proyecto y a su ruta de clase de ejecución cuando ejecute las pruebas.

Descarguemos ahora JUnit (JUnit 3.8.1 o más reciente [5]) distribución, que contiene varias muestras de prueba que ejecutará para familiarizarse con la ejecución de pruebas JUnit. Sigue estos pasos:

1. Descargue la última versión de JUnit de junit.org, a la que se hace referencia en el paso 2 como http://junit.zip.
2. Descomprime el junit.zip archivo de distribución a un directorio en su sistema informático (por ejemplo, C:\ en Windows o / optar/ en UNIX).
3. Debajo de este directorio, descomprimir creará un subdirectorio para la distribución JUnit que descargó (por ejemplo, C: \ junit3.8.1 en Windows o / opt / junit.3.8.1 en UNIX).

Ahora está listo para ejecutar las pruebas proporcionadas con la distribución JUnit. JUnit viene completo con programas Java que puede usar para ver el resultado de una prueba. Hay un *gráfico,* Corredor de prueba basado en swing (figura 1.2), así como un *textual* corredor de pruebas (figura 1.3) que se puede utilizar desde la línea de comandos.

Para ejecutar el corredor de prueba gráfico, abra un shell en C: \ junit3.8.1 en Windows o en / opt / junit3.8.1 en UNIX y escriba el comando apropiado:

Ventanas:

```
java -cp junit.jar ;. junit.swingui.TestRunner junit.samples.AllTests
```

UNIX:

```
java -cp junit.jar :. junit.swingui.TestRunner junit.samples.AllTests
```

Para ejecutar el corredor de prueba de texto, abra un shell en C: \ junit3.8.1 en Windows o en /opt/junit3.8.1 en UNIX y escriba el comando apropiado:

Ventanas:

```
java -cp junit.jar ;. junit.textui.TestRunner junit.samples.AllTests
```

UNIX:

```
java -cp junit.jar :. junit.textui.TestRunner junit.samples.AllTests
```

[5] Las versiones anteriores de JUnit no funcionarán con todo nuestro código de muestra.

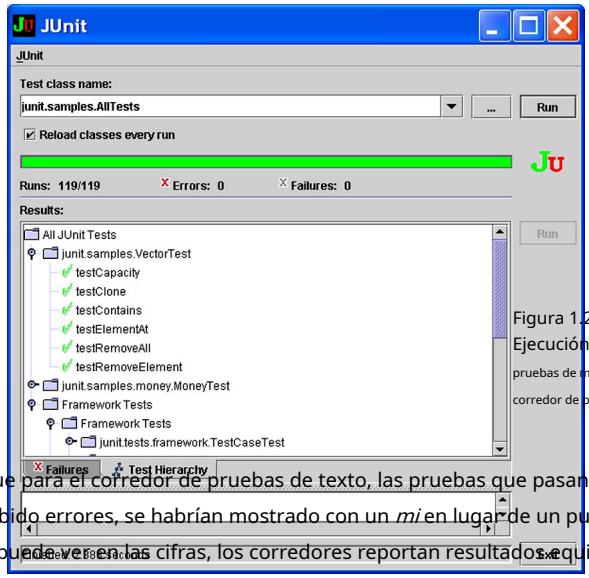


Figura 1.2
Ejecución del JUnit
pruebas de muestra de distribución utilizando el
corredor de prueba gráfico Swing

Observe que para el corredor de pruebas de texto, las pruebas que pasan se muestran con un punto. Si hubiera habido errores, se habrían mostrado con un *mi* en lugar de un punto.

Como puede ver en las cifras, los corredores reportan resultados equivalentes. El corredor de prueba textual es más fácil de ejecutar, especialmente en trabajos por lotes, aunque el corredor de prueba gráfico puede proporcionar más detalles.

El ejecutor de pruebas gráfico también usa su propia instancia de cargador de clases (un cargador de clases de recarga). Esto hace que sea más fácil de usar de forma interactiva, porque puede volver a cargar las clases (después de cambiarlas) y ejecutar rápidamente la prueba nuevamente sin reiniciar el corredor de prueba.

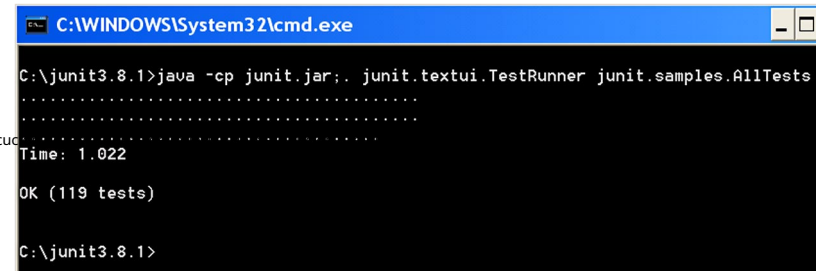


Figura 1.3 Ejecuc

En el capítulo 5, “Automatización de JUnit”, analizamos la ejecución de pruebas utilizando la herramienta de compilación Ant y desde entornos de desarrollo integrados, como Eclipse.

1.5 Prueba con JUnit

JUnit tiene muchas características que hacen que las pruebas sean más fáciles de escribir y ejecutar. Verá estas características en funcionamiento a lo largo de este libro:

- Interfaces alternativas, o corredores de prueba, para mostrar el resultado de sus pruebas. Los corredores de prueba de línea de comandos, AWT y Swing se incluyen en la distribución JUnit.
- Cargadores de clases separados para cada prueba unitaria para evitar efectos secundarios.
- Métodos estándar de inicialización y recuperación de recursos (configuración y lágrima-Abajo).
- Una variedad de métodos de afirmación para facilitar la verificación de los resultados de sus pruebas.
- Integración con herramientas populares como Ant y Maven, e IDE populares como Eclipse, IntelliJ y JBuilder.

Sin más preámbulos, pasemos al listado 1.4 y veamos qué es lo simple Calculadora La prueba se ve como cuando se escribe con JUnit.

```
importar junit.framework.TestCase;

clase pública TestCalculator se extiende Caso de prueba {

    vacío público testAdd ()
    {
        Calculadora calculadora = nuevo Calculadora();
        doble resultado = calculadora.add (10, 50); asertEquals (60,
        resultado, 0);
    }
}
```

B

C

D

mi

F

Bastante simple, ¿no? Vamos a dividirlo por números.

En el listado 1.4 en B, comienza extendiendo la clase de prueba desde el JUnit estándar junit.framework.TestCase. Esta clase base incluye el código de marco que JUnit necesita para ejecutar las pruebas automáticamente.

A C, simplemente asegúrese de que el nombre del método siga el patrón prueba ***XXX ().*** Observar esta convención de nomenclatura deja claro al marco

que el método es una prueba unitaria y que se puede ejecutar automáticamente. Siguiendo el prueba ***XXX*** la convención de nomenclatura no es un requisito estricto, pero se recomienda encarecidamente como práctica recomendada.

A D, comienza la prueba creando una instancia del Calculadora clase (el "objeto bajo prueba"), y en mi, como antes, ejecuta la prueba llamando al método a probar, pasándole dos valores conocidos.

A F, ¡el marco JUnit comienza a brillar! Para comprobar el resultado de la prueba, llame a un asertEquals método, que heredaste de la base Caso de prueba. El Javadoc para el asertEquals el método es:

```
/ **
  * Afirma que dos dobles son iguales con respecto a un delta. Si el
  * el valor esperado es infinito, entonces se ignora el valor delta.
  * /

vacío público estático asertEquals ( doble previsto, doble real,
      doble delta)
```

En el listado 1.4, pasó asertEquals estos parámetros:

- esperado = 60
- actual = resultado
- delta = 0

Como le pasó a la calculadora los valores 10 y 50, le dice asertEquals esperar que la suma sea 60. (Pasas 0 ya que estás sumando números enteros, por lo que no hay delta). calculadora objeto, metió el valor de retorno en un local doble nombrada resultado. Entonces, pasas esa variable a asertEquals para comparar con el valor esperado de 60.

Lo que nos lleva a lo misterioso delta parámetro. Muy a menudo, el delta El parámetro puede ser cero y puede ignorarlo con seguridad. Entra en juego con cálculos que no siempre son precisos, que incluyen muchos cálculos de punto flotante. El delta proporciona un factor más / menos. Entonces si el real está dentro del rango (esperado-delta) y (esperado + delta), la prueba aún pasará.

Si desea ingresar el programa de prueba del listado 1.4 en su editor de texto o IDE, puede probarlo usando el corredor de prueba gráfico. Supongamos que ha introducido el código de los listados 1.1 y 1.4 en el C: \ junitbook \ jumpstart directorio (/ opt / junitbook / jumpstart en UNIX). Primero compilemos el código abriendo un indicador de shell en ese directorio y escribiendo lo siguiente (asumiremos que tiene el ejecutable javac en su SENDERO):

Objetivos de diseño de JUnit

El equipo de JUnit ha definido tres objetivos discretos para el marco:

- El marco debe ayudarnos a escribir pruebas útiles.
- El marco debe ayudarnos a crear pruebas que conserven su valor a lo largo del tiempo.
- El marco debe ayudarnos a reducir el costo de escribir pruebas al reutilizar el código.

En el listado 1.4, intentamos mostrar lo fácil que puede ser escribir pruebas con JUnit. Volveremos a los otros objetivos en el capítulo 2.

Ventanas:

javac -cp .. \ .. \ junit3.8.1 \ junit.jar * .java

UNIX:

javac -cp ../../junit3.8.1/junit.jar * .java

Ahora está listo para iniciar el corredor de prueba de Swing, escribiendo lo siguiente:

Ventanas:

java -cp.; .. \ .. \ junit3.8.1 \ junit.jar

→ junit.swingui.TestRunner TestCalculator

UNIX:

java -cp.: ../../ junit3.8.1 / junit.jar

→ junit.swingui.TestRunner TestCalculator

El resultado de la prueba se muestra en la figura 1.4.

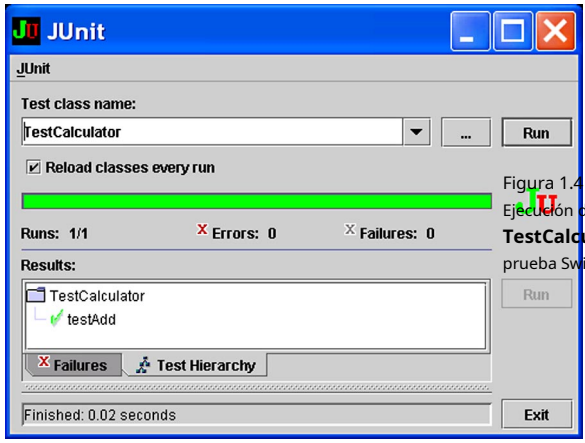


Figura 1.4
Ejecución de la primera prueba JUnit **TestCalculator** usando el corredor de prueba Swing

Lo notable del JUnit TestCalculator clase en el listado 1.4 es que el código es tan fácil de escribir como el primero TestCalculator programa en el listado 1.2, pero ahora puede ejecutar la prueba automáticamente a través del marco JUnit.

NOTA Si mantiene pruebas escritas antes de JUnit versión 3.8.1, necesitará agregar un constructor, como este:

TestCalculator público (nombre de cadena) {super (nombre); }

Ya no es necesario con JUnit 3.8 y posteriores.

1.6 Resumen

Cada desarrollador realiza algún tipo de prueba para ver si el nuevo código realmente funciona. Los desarrolladores que utilizan pruebas unitarias automáticas pueden repetir estas pruebas bajo demanda para asegurarse de que el código siga funcionando más tarde.

Las pruebas unitarias simples no son difíciles de escribir a mano, pero a medida que las pruebas se vuelven más complejas, la escritura y el mantenimiento de las pruebas pueden volverse más difíciles. JUnit es un marco de pruebas unitarias que facilita la creación, ejecución y revisión de pruebas unitarias.

En este capítulo, arañamos la superficie de JUnit al instalar el marco y realizar una prueba sencilla. Por supuesto, JUnit tiene mucho más que ofrecer.

En el capítulo 2, examinamos más de cerca las clases del marco JUnit y cómo funcionan juntas para hacer que las pruebas unitarias sean eficientes y efectivas. (¡Sin mencionar simplemente diversión!)

2 Explorando JUnit

Este capítulo cubre

- Usando las clases principales de JUnit
- Comprender el ciclo de vida de JUnit

Los errores son los portales del descubrimiento.

- James Joyce

En el capítulo 1, decidimos que necesitamos un programa de prueba automático para poder replicar nuestras pruebas. A medida que agregamos nuevas clases, a menudo queremos hacer cambios en las clases bajo prueba. Por supuesto, la experiencia nos ha enseñado que a veces las clases interactúan de formas inesperadas. Entonces, realmente nos gustaría seguir ejecutando todas nuestras pruebas en todas nuestras clases, ya sea que hayan cambiado o no. Pero, ¿cómo podemos ejecutar varios casos de prueba? ¿Y qué usamos para ejecutar todas estas pruebas?

En este capítulo, veremos cómo JUnit proporciona la funcionalidad para responder esas preguntas. Comenzaremos con una descripción general de las clases principales de JUnit. TestCase, TestSuite, y BaseTestRunner. Luego, veremos más de cerca a los corredores de prueba y Banco de pruebas, antes de que volvamos a visitar a nuestro viejo amigo Caso de prueba. Finalmente, examinaremos cómo funcionan juntas las clases principales.

2.1 Explorando el núcleo JUnit

El TestCalculator El programa del capítulo 1, que se muestra en el listado 2.1, puede ejecutar un solo caso de prueba bajo demanda. Como puede ver, para crear un solo caso de prueba, puede extender el Caso de prueba clase.

```
importar junit.framework.TestCase;

clase pública TestCalculator se extiende Caso de prueba {

    vacío público testAdd ()
    {
        Calculadora calculadora = nuevo Calculadora();
        doble resultado = calculadora.add (10, 50); asertEquals (60,
        resultado, 0);
    }
}
```

Cuando necesita escribir más casos de prueba, crea más Caso de prueba objetos. Cuando necesita ejecutar varios Caso de prueba objetos a la vez, crea otro objeto llamado un Banco de pruebas. Para ejecutar un Banco de pruebas, usas un TestRunner, como lo hiciste por un single Caso de prueba objeto del capítulo anterior. La figura 2.1 muestra este trío en acción.

Estas tres clases son la columna vertebral del marco JUnit. Una vez que entiendas como TestCase, TestSuite, y BaseTestRunner trabajar, serás capaz de

Figura 2.1 Los miembros del trío JUnit trabajan juntos para generar un resultado de prueba.

DEFINICIÓN ***Caso de prueba (*** *o caso de prueba)* —Una clase que amplía JUnit Caso de prueba clase. Contiene una o más pruebas representadas por prueba ***XXX*** métodos. Un caso de prueba se utiliza para agrupar pruebas que ejercen comportamientos comunes. En el resto de este libro, cuando mencionamos un *prueba,* nos referimos a un prueba ***XXX*** método; y cuando mencionamos un *caso de prueba,* nos referimos a una clase que se extiende Caso de prueba —Es decir, un conjunto de pruebas.

Banco de pruebas (*o suite de pruebas)* —Un grupo de pruebas. Un conjunto de pruebas es una forma conveniente de agrupar las pruebas relacionadas. Por ejemplo, si no define un conjunto de pruebas para un Caso de prueba, JUnit proporciona automáticamente un conjunto de pruebas que incluye todas las pruebas que se encuentran en el Caso de prueba (más sobre eso más tarde).

TestRunner (*o corredor de prueba)* —Un lanzador de suites de prueba. JUnit proporciona varios corredores de prueba que puede utilizar para ejecutar sus pruebas. No hay TestRunner interfaz, solo una BaseTestRunner que todos los corredores de prueba extienden. Así cuando escribimos TestRunner en realidad nos referimos a cualquier clase de corredor de pruebas que se extienda BaseTestRunner.

escriba las pruebas que necesite. Diariamente, solo necesita escribir casos de prueba. Las otras clases trabajan detrás de escena para dar vida a tus pruebas. Estas tres clases trabajan en estrecha colaboración con otras cuatro clases para crear el marco principal de JUnit. La Tabla 2.1 resume las responsabilidades de las siete clases principales.

Tabla 2.1 Las siete clases e interfaces principales de JUnit (las interfaces se indican en cursiva)

Clase / ***interfaz***	Responsabilidades	Introducido en ...
Afirmar	Un método de aserción es silencioso cuando su proposición tiene éxito, pero lanza una excepción si la proposición falla.	Sección 2.6.2
Resultado de la prueba	A Resultado de la prueba recopila cualquier error o falla que ocurra durante una prueba.	Sección 2.4
Prueba	A ***Prueba*** se puede ejecutar y pasar un Resultado de la prueba.	Sección 2.3.2
TestListener	A ***TestListener*** está informado de los eventos que ocurren durante una prueba, incluso cuándo comienza y termina la prueba, junto con cualquier error o falla.	Sección 2.5

Continúa en la siguiente página

Tabla 2.1 Las siete clases e interfaces principales de JUnit (las interfaces se indican en cursiva) (*continuado)*

Clase / ***interfaz***	Responsabilidades	Introducido en ...
Caso de prueba	A Caso de prueba define un entorno (o *accesorio)* que se puede utilizar para ejecutar varias pruebas.	Sección 2.1
Banco de pruebas	A Banco de pruebas ejecuta una colección de casos de prueba, que pueden incluir otros conjuntos de pruebas. Es un compuesto de ***Prueba** s.*	Sección 2.3
BaseTestRunner	Un corredor de pruebas es una interfaz de usuario para iniciar pruebas. BaseTestRunner es la superclase para todos los corredores de pruebas.	Sección 2.2.2

La Figura 2.2 muestra las relaciones entre las siete clases principales de JUnit. Verá cómo estas clases e interfaces centrales funcionan juntas en este capítulo y en todo el libro.

2.2 Lanzamiento de pruebas con corredores de prueba

Escribir pruebas puede ser divertido, pero ¿qué pasa con el duro trabajo de ejecutarlas? Cuando escribe pruebas por primera vez, desea que se ejecuten de la forma más rápida y sencilla posible. Debería poder hacer que las pruebas formen parte del ciclo de desarrollo: *código: ejecutar: prueba: código (* o *prueba: código: ejecutar: prueba* si está inclinado a probar primero). Hay IDE y compiladores para crear y ejecutar aplicaciones rápidamente, pero ¿qué puede usar para ejecutar las pruebas?

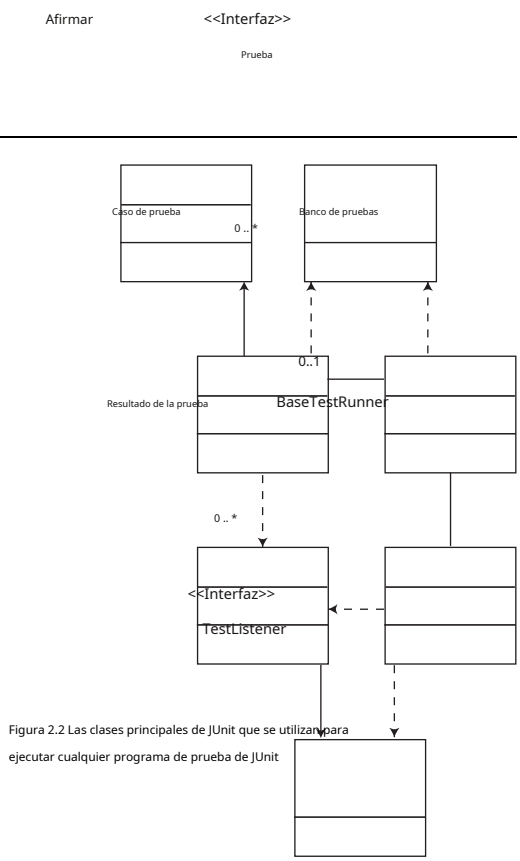


Figura 2.2 Las clases principales de JUnit que se utilizan para ejecutar cualquier programa de prueba de JUnit

2.2.1 Seleccionar un corredor de prueba

Para que la ejecución de pruebas sea lo más rápida y sencilla posible, JUnit proporciona una selección de corredores de prueba. Los corredores de prueba están diseñados para ejecutar sus pruebas y proporcionarle estadísticas sobre el resultado. Debido a que están diseñados específicamente para este propósito, los corredores de prueba pueden ser muy fáciles de usar. La figura 2.3 muestra el corredor de pruebas Swing en acción.

El indicador de progreso que se encuentra en la pantalla es la famosa barra verde JUnit. *Mantenga la barra verde para mantener limpio el código* es el lema de JUnit.

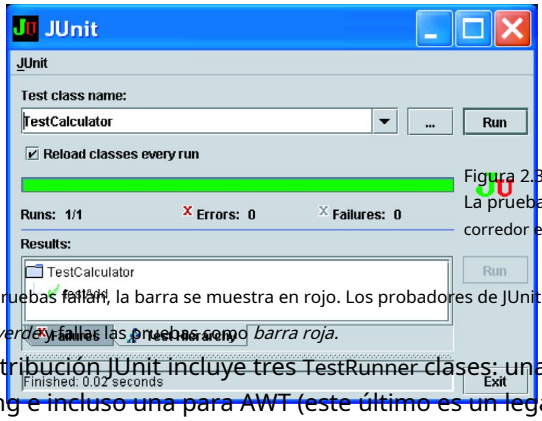


Figura 2.3
La prueba gráfica corredor en acción

Cuando las pruebas fallan, la barra se muestra en rojo. Los probadores de JUnit tienden a referirse a pasar las pruebas como *barra verde* y fallar las pruebas como *barra roja.*

La distribución JUnit incluye tres TestRunner clases: una para la consola de texto, otra para Swing e incluso una para AWT (este último es un legado que pocas personas todavía usan).

2.2.2 *Definición de su propio corredor de pruebas*

A diferencia de otros elementos del marco JUnit, no hay TestRunner interfaz. En cambio, los distintos corredores de prueba incluidos con JUnit se extienden BaseTestRunner. Si necesita escribir su propio corredor de prueba por cualquier motivo, también puede ampliar esta clase usted mismo. Por ejemplo, el marco de Cactus que discutiremos en capítulos posteriores se extiende BaseTestRunner para crear un ServletTestRunner que puede ejecutar pruebas JUnit desde un navegador.

2.3 Composición de pruebas con TestSuite

Las cosas simples deben ser simples... y las cosas complejas deben ser posibles. Suponga que compila el programa de prueba de calculadora simple del listado 2.1 y se lo entrega a un corredor de prueba gráfico, como este:

```
> Java junit.swingui.TestRunner TestCalculator
```

Debería funcionar bien, asumiendo que tiene la ruta de clases correcta. (Consulte la figura 2.3 para ver el corredor de pruebas Swing en acción). En conjunto, esto parece simple, al menos en lo que respecta a ejecutar un solo caso de prueba.

Pero, ¿qué sucede cuando desea ejecutar varios casos de prueba? ¿O solo algunos de sus casos de prueba? ¿Cómo se pueden agrupar los casos de prueba?

Entre los Caso de prueba y el TestRunner, parece que necesita algún tipo de contenedor que pueda recopilar varias pruebas juntas y ejecutarlas como un conjunto. Pero, al facilitar la ejecución de varios casos, no querrá que sea más difícil ejecutar un solo caso de prueba.

La respuesta de JUnit a este acertijo es la Banco de pruebas. El Banco de pruebas está diseñado para ejecutar uno o más casos de prueba. El corredor de pruebas lanza el Banco de pruebas; qué casos de prueba ejecutar depende de la Banco de pruebas.

2.3.1 Ejecutando la suite automática

Quizás se pregunte cómo logró ejecutar el ejemplo al final del capítulo 1, cuando no definió un Banco de pruebas. Para simplificar las cosas simples, el ejecutor de pruebas crea automáticamente un Banco de pruebas si no proporciona uno propio. (*¡Dulce!)*

El valor por defecto Banco de pruebas escanea su clase de prueba en busca de métodos que comiencen con los caracteres *prueba.* Internamente, el valor predeterminado Banco de pruebas crea una instancia de tu Caso de prueba para cada prueba ***XXX*** método. El nombre del método que se invoca se pasa como Caso de prueba constructor, de modo que cada instancia tenga una identidad única.

Para el TestCalculator en el listado 2.1, el valor predeterminado Banco de pruebas podría representarse en un código como este:

```
público estático Banco de pruebas() {

    volver nuevo TestSuite (TestCalculator. clase);
}
```

Y esto es nuevamente equivalente a lo siguiente:

```
público estático Banco de pruebas() {

    Suite TestSuite = nuevo Banco de pruebas(); suite.addTest ( nuevo TestCalculator
    ("testAdd"));
    regreso suite;
}
```

NOTA Para utilizar este formulario, el hipotético TestCalculator la clase necesitaría para definir el constructor apropiado, así:

```
TestCalculator público (nombre de cadena) {super (nombre); }
```

JUnit 3.8 hizo que este constructor sea opcional, por lo que no es parte del código fuente del original. TestCalculator clase. La mayoría de los desarrolladores ahora confían en el Banco de pruebas y rara vez crean suites manuales, por lo que pueden omitir este constructor.

Si agregaste otra prueba, como testSubtract, el valor por defecto Banco de pruebas lo incluiría automáticamente también, ahorrándole la molestia de mantener otro bloque de pelusa:

```
público estático Banco de pruebas() {

        Suite TestSuite = nuevo Banco de pruebas(); suite.addTest ( nuevo TestCalculator
        ("testAdd"));
        suite.addTest ( nuevo TestCalculator ("testSubstract"));
        regreso suite;
}
```

Este es un código trivial y sería fácil de copiar, pegar y editar, pero ¿por qué molestarse con tanto trabajo cuando JUnit puede hacerlo por usted? Lo más importante es que el conjunto de pruebas automático asegura que no olvide agregar alguna prueba al conjunto de pruebas.

2.3.2 *Desarrollando su propia suite de pruebas*

El valor por defecto Banco de pruebas contribuye en gran medida a mantener las cosas simples simples. Pero, ¿qué sucede cuando la suite predeterminada no satisface sus necesidades? Es posible que desee combinar suites de varios paquetes diferentes como parte de una suite principal. Si está trabajando en una nueva función, a medida que realiza cambios, es posible que desee ejecutar un pequeño conjunto de pruebas relevantes.

Hay muchas circunstancias en las que es posible que desee ejecutar varios conjuntos o pruebas seleccionadas dentro de un conjunto. Incluso el marco JUnit tiene un caso especial: para probar la función de suite automática, ¡el marco necesita construir su propia suite para comparar!

Si comprueba el Javadoc para Banco de pruebas y Caso de prueba, notará que ambos implementan el Prueba interfaz, que se muestra en el listado 2.2.

```
paquete junit.framework;

Prueba de interfaz pública {
        public abstract int countTestCases ();
        ejecución vacía abstracta pública (resultado de TestResult);
}
```

Si es un gran triunfador y también busca el Javadoc para Banco de pruebas, probablemente notarás que el addTest la firma no especifica un Caso de prueba tipo — cualquier viejo Prueba servirá.

La capacidad de agregar tanto conjuntos de pruebas como casos de prueba a un conjunto simplifica la creación de conjuntos de especialidades, así como un agregado. TestAll clase para su aplicación.

Objetivos de diseño de JUnit

La combinación simple pero efectiva de un TestRunner con un Banco de pruebas facilita la ejecución de todas sus pruebas todos los días. Al mismo tiempo, puede seleccionar un subconjunto de pruebas que se relacionen con el esfuerzo de desarrollo actual. Esto habla del segundo objetivo de diseño de JUnit:

El marco debe crear pruebas que conserven su valor a lo largo del tiempo.

Cuando continúa ejecutando sus pruebas, minimiza su inversión en pruebas y maximiza el retorno de esa inversión.

Normalmente, el TestAll la clase es solo una estática suite método que registra lo que sea Prueba objetos (Caso de prueba objetos o Banco de pruebas objetos) su aplicación debe ejecutarse de forma regular. El Listado 2.3 muestra un típico TestAll clase.

2.3 Una clase típica de TestAll

```
importar junit.framework.Test;
importar junit.framework.TestSuite;
importar junitbook.sampling.TestDefaultController;

clase pública TestAll
{
        público estático Banco de pruebas() {

                Suite TestSuite = nuevo TestSuite ("Todas las pruebas de la parte 1"); suite.addTestSuite
                (TestCalculator. clase);
                suite.addTestSuite (TestDefaultController. clase);
                        // si TestDefaultController tuviera un método de suite
                        // (o métodos de suite alternativos) también puede usar
                        // suite.addTestSuite (TestDefaultController.suite ());
                regreso suite;
        }
}
```

B Crear un suite método para llamar a todas sus otras pruebas o suites. Dar el Banco

C de pruebas una leyenda para ayudar a identificarlo más tarde.

D Llama addTestSuite para agregar lo que sea Caso de prueba objetos o Banco de pruebas objetos que desea ejecutar juntos. Funciona para ambos tipos porque el addTestSuite el método acepta un Prueba objeto como parámetro, y Caso de prueba y Banco de pruebas ambos implementan el Prueba interfaz.

En el capítulo 5, analizamos varias técnicas para automatizar tareas como ésta, de modo que no tenga que crear y mantener una TestAll clase. Por supuesto, es posible que aún desee crear suites especializadas para subconjuntos discretos de sus pruebas.

Patrones de diseño en acción: Composite y Command

Patrón compuesto. "Componga objetos en estructuras de árbol para representar jerarquías completas. Composite permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme ". [1] El uso de JUnit del Prueba La interfaz para ejecutar una sola prueba o conjuntos de conjuntos de conjuntos de pruebas es un ejemplo del patrón compuesto. Cuando agrega un objeto a un Banco de pruebas, realmente estás agregando un ***Prueba,*** no simplemente un Prueba ***Caso.*** Porque ambos Banco de pruebas y Caso de prueba implementar Prueba,
puede agregar cualquiera a una suite. Si el Prueba es un Caso de prueba, se ejecuta la prueba única. Si el Prueba es un Banco de pruebas, luego se ejecuta un grupo de pruebas (que podrían incluir
otro Banco de pruebas s).

Patrón de comando. "Encapsular una solicitud como un objeto, lo que le permite parametrizar clientes con diferentes solicitudes, solicitudes de cola o de registro, y admitir operaciones que se pueden deshacer". [2] El uso de la Prueba interfaz para proporcionar un correr El método es un ejemplo del patrón Command.

2.4 Recopilación de parámetros con TestResult

Newton nos enseñó que por cada acción hay una reacción igual y opuesta. Asimismo, para cada Banco de pruebas, hay un Resultado de la prueba.

A Resultado de la prueba recopila los resultados de la ejecución de un Caso de prueba. Si todas sus pruebas siempre tuvieran éxito, ¿cuál sería el punto de ejecutarlas? Entonces, Resultado de la prueba almacena los detalles de todas sus pruebas, aprobadas o reprobadas.

El TestCalculator El programa (listado 2.1) incluye una línea que dice

```
asertEquals (60, resultado, 0);
```

Si el resultado no fuera igual a 60, JUnit crearía un TestFailure objeto para ser almacenado en el Resultado de la prueba.

El TestRunner usa el Resultado de la prueba para informar el resultado de sus pruebas. Si no hay TestFailure s en el Resultado de la prueba colección, entonces el código está limpio y la barra se vuelve verde. Si hay fallas, el TestRunner informa el fallo

[1] Erich Gamma y col., *Patrones de diseño (* Reading, MA: Addison-Wesley, 1995).

[2] Ibídem.