Robert C. Martin

La artesanía del código limpio

Disciplinas, estándares y ética

Prólogo de **Stacia Heimgartner Viscardi**, CST y Agile Mentor

ANAYA MULTIMEDIA

Índice de contenidos

Agradecimientos ... 6
Sobre el autor ... 6

Prólogo ... 14

Prefacio ... 18

Sobre el término "artesanía" ... 18
Sobre el Único camino verdadero ... 19
Introducción al libro ... 19
Para usted ... 19
Para la sociedad ... 20
La estructura de este libro ... 21
Nota para jefes ... 22
Vídeos ... 22
Imágenes y referencias ... 23

1. Artesanía ... 24

Parte I. Las disciplinas ... 32

Programación Extrema ... 34
El círculo de la vida ... 35
Desarrollo guiado por pruebas ... 35
La refactorización ... 36
Diseño simple ... 37
Programación colaborativa ... 38
Pruebas de aceptación ... 38

2. Desarrollo guiado por pruebas (TDD) ... 40

Panorama general ... 41
Software ... 43
Las tres leyes del TDD ... 44
La cuarta ley ... 52
Conceptos básicos ... 54
Ejemplos simples ... 54
Stack ... 55
Factores primos ... 68
La partida de bolos ... 75
Conclusión ... 89

3. TDD avanzado ... 90

Ordenamiento 1 ... 91
Ordenamiento 2 ... 95
Atascarse ... 101
Arrange, *Act*, *Assert* ... 107
Llega el BDD ... 108
Máquina de estados finitos ... 109
Otra vez el BDD ... 111
Dobles de pruebas ... 111
Dummy ... 113
Stub ... 116
Spy ... 119
Mock ... 121
Fake ... 123
El principio de incertidumbre del TDD ... 125
Londres frente a Chicago ... 135
El problema de la certeza ... 136
Londres ... 137
Chicago ... 137
Síntesis ... 138
Arquitectura ... 139
Conclusión ... 141

4. Diseño de pruebas 142

Probar bases de datos 143
Probar GUI 145
Entrada de GUI 147
Patrones de pruebas 148
Test-specific subclass 148
Self-Shunt 149
Humble Object 150
Diseño de pruebas 153
El problema de las pruebas frágiles 153
La correspondencia uno a uno 153
Romper la correspondencia 155
El videoclub 156
Especificidad frente a generalidad 171
Premisa de prioridad de transformación 172
{} → Nulo 174
Nulo → Constante 174
Constante → Variable 175
Incondicional → Selección 175
Valor → Lista 176
Sentencia → Recursividad 176
Selección → Iteración 177
Valor → Valor mutado 177
Ejemplo: Fibonacci 178
La premisa de prioridad de transformación 181
Conclusión 182

5. Refactorización 184

¿Qué es la refactorización? 185
El kit de herramientas básico 186
Cambiar nombre 187
Extraer método 187
Extraer variable 189
Extraer campo 190
El cubo de Rubik 200
Las disciplinas 200
Pruebas 201
Pruebas rápidas 201
Romper todas las correspondencias uno a uno profundas 201

Refactorice continuamente 201
Refactorice sin piedad 202
¡Que sigan pasándose las pruebas! 202
Déjese una salida 203
Conclusión 203

6. Diseño simple 204

YAGNI 207
Cubierto por pruebas 208
Cobertura 209
Un objetivo asintótico 211
¿Diseño? 211
Pero hay más 212
Maximizar la expresión 212
La abstracción subyacente 213
Pruebas: la otra mitad del problema 214
Minimizar la duplicación 215
Duplicación accidental 216
Minimizar el tamaño 217
Diseño simple 217

7. Programación colaborativa 218

8. Pruebas de aceptación 222

La disciplina 224
La construcción continua 225

Parte II. Los estándares 226

Su nuevo director de tecnología 227

9. Productividad 228

Nunca enviaremos c**a 229
Adaptabilidad barata 231
Siempre estaremos listos 232
Productividad estable 233

10. Calidad 236

Mejora continua 237
Competencia sin miedo 237
Calidad extrema 239

No le echamos el muerto a QA ... 239
La enfermedad de QA ... 240
QA no encontrará nada ... 240
Automatización de pruebas ... 241
Pruebas automatizadas e interfaces de usuario ... 242
Probar la interfaz de usuario ... 242

11. Valor ... 244

Nos cubrimos unos a otros ... 245
Estimaciones honestas ... 246
Debe decir "no" ... 248
Aprendizaje agresivo continuo ... 249
Enseñanza ... 250

Parte III. La ética ... 252

El primer programador ... 253
Setenta y cinco años ... 254
Empollones y salvadores ... 257
Modelos a imitar y villanos ... 259
Gobernamos el mundo ... 260
Catástrofes ... 261
El juramento ... 263

12. Daño ... 264

Primero, no cause daño ... 265
No causar daño a la sociedad ... 266
No causar daño a la función ... 267
No causar daño a la estructura ... 269
Soft ... 271
Pruebas ... 272
El mejor trabajo ... 273
Hacerlo bien ... 274
¿Qué es una buena estructura? ... 275
La matriz de Eisenhower ... 276
Programadores como partes interesadas ... 278
Lo mejor ... 279
Prueba repetible ... 281
Dijkstra ... 281
Demostrar la corrección ... 282
Programación estructurada ... 284

Descomposición funcional ... 286
Desarrollo guiado por pruebas ... 287

13. Integridad ... 290

Ciclos pequeños ... 291
La historia del control del código fuente ... 291
Git ... 296
Ciclos cortos ... 297
Integración continua ... 297
Ramas frente a *toggles* ... 298
Despliegue continuo ... 300
Construcción continua ... 301
Mejora incansable ... 302
Cobertura de las pruebas ... 302
Prueba de mutaciones ... 303
Estabilidad semántica ... 303
Limpieza ... 304
Creaciones ... 304
Mantener una productividad alta ... 305
Viscosidad ... 305
Gestionar las distracciones ... 308
Gestión del tiempo ... 310

14. Trabajo en equipo ... 312

Trabajar como un equipo ... 313
Oficina abierta/virtual ... 313
Estimar de forma honesta y justa ... 314
Mentiras ... 315
Honestidad, exactitud, precisión ... 316
Historia 1: Vectores ... 317
Historia 2: pCCU ... 319
La lección ... 320
Exactitud ... 320
Precisión ... 322
Agregación ... 323
Honestidad ... 324
Respeto ... 326
Nunca deje de aprender ... 326

Índice alfabético ... 328

Prefacio

Antes de empezar, hay dos asuntos que debemos tratar para garantizar que usted, querido lector, entiende el marco de referencia en el que se presenta este libro.

Sobre el término "artesanía"

El comienzo del siglo XXI ha estado marcado por cierta controversia respecto al lenguaje. Quienes pertenecemos a la industria del software hemos protagonizado parte de esta controversia. Un término que se ha señalado como un fracaso a la hora de ser inclusivo es "artesano".

He pensado bastante en este asunto y he hablado con muchas personas con opiniones distintas, y he llegado a la conclusión de que no hay un término mejor para utilizar en el contexto de este libro.

Se barajaron alternativas a "artesano", como "persona artesana", "oficial de artesanía" y "artífice", entre otras, pero ninguno de esos términos tiene el peso histórico de "artesano", y dicho peso es importante para este mensaje.

"Artesano" nos hace pensar en una persona muy habilidosa y dotada en una actividad concreta, alguien que está a gusto con sus herramientas y su oficio, que se enorgullece de su trabajo y que hace que podamos esperar que se comporte con la dignidad y profesionalidad de su vocación.

Puede que haya quien no esté de acuerdo con mi decisión. Entiendo por qué podría darse el caso. Solo espero que nadie lo interprete como un intento de ser excluyente de ninguna manera, ya que esa no es en absoluto mi intención.

Sobre el Único camino verdadero

Cuando lea *La artesanía del código limpio*, puede que tenga la sensación de que este es el "Único camino verdadero" a la artesanía. Puede que sea así para mí, pero no tiene por qué serlo para usted. Le ofrezco este libro como ejemplo de mi camino. Por supuesto, usted tendrá que elegir el suyo.

¿Necesitaremos al final un "Único camino verdadero"? No lo sé. Quizá. Como leerá en estas páginas, la presión para que haya una definición estricta de una profesión del software está aumentando. Puede que seamos capaces de admitir muchos caminos diferentes, dependiendo de la criticidad del software que se está creando. Pero, como verá en este libro, puede que no sea tan fácil separar el software crítico del que no lo es.

De una cosa estoy seguro. Los días de los "Jueces"[1] se han acabado. Ya no basta con que un programador haga lo que es correcto a su parecer. Habrá disciplinas, estándares y ética. La decisión que tenemos hoy ante nosotros es si los definiremos nosotros, los programadores, o si nos los impondrán aquellos que no nos conocen.

Introducción al libro

Este libro está escrito para programadores y para jefes de programadores. Pero, en otro sentido, está escrito para toda la sociedad humana, porque somos nosotros, los programadores, los que nos hemos encontrado sin darnos cuenta en el mismísimo eje de esa sociedad.

Para usted

Si es un programador con muchos años de experiencia, es probable que conozca la satisfacción por desplegar un sistema que funcione. Se siente cierto orgullo por haber sido parte de ese logro. Está orgulloso de que su sistema salga al mundo.

Pero ¿está orgulloso de la forma en que consiguió que ese sistema saliese al mundo? ¿Es el orgullo de haber acabado o el orgullo por la calidad del trabajo? ¿Está orgulloso de que el sistema se haya desplegado o de la forma en que ha construido ese sistema?

Cuando llega a casa después de un duro día escribiendo código, ¿se mira en el espejo y dice "buen trabajo hoy" o tiene que darse una ducha?

1. Referencia al Libro de los Jueces del Antiguo Testamento.

de la disciplina o solo de los elementos arbitrarios. No se deje confundir por los elementos arbitrarios. Mantenga su atención sobre los elementos esenciales. Una vez que haya internalizado la esencia de cada disciplina, lo más probable es que la forma arbitraria pierda importancia.

Por ejemplo, en 1861, Ignaz Semmelweis publicó sus hallazgos derivados de la aplicación de la disciplina del lavado de manos para los médicos. Los resultados de su investigación fueron asombrosos. Fue capaz de demostrar que, cuando los médicos se lavaban las manos meticulosamente con hipoclorito cálcico antes de examinar a mujeres embarazadas, la tasa de muertes de esas mujeres a causa de la subsiguiente sepsis pasaba de ser una de cada diez a prácticamente cero.

Pero los médicos de la época no separaron la esencia de lo arbitrario cuando repasaron la disciplina propuesta por Semmelweis. El hipoclorito cálcico era la parte arbitraria. Les espantaba la inconveniencia de lavarse con lejía, así que rechazaron la evidencia de la naturaleza esencial del lavado de manos.

Pasaron muchas décadas hasta que los médicos empezaron a lavarse las manos.

PROGRAMACIÓN EXTREMA

En 1970, Winston Royce publicó el artículo que convirtió el proceso de desarrollo en cascada en el modelo dominante. Se necesitaron casi 30 años para enmendar ese error.

Para 1995, los expertos en software habían empezado a considerar un enfoque diferente, más progresivo. Se presentaron procesos como Scrum, el desarrollo basado en funcionalidades, el método de desarrollo de sistemas dinámicos (DSDM) y las metodologías Crystal, pero, en general, hubo muy pocos cambios en la industria.

Después, en 1999, Kent Beck publicó el libro *Una explicación de la Programación Extrema* (Addison-Wesley). La Programación Extrema (XP) se cimentaba en las ideas de aquellos procesos anteriores, pero añadía algo nuevo: las prácticas de ingeniería.

El entusiasmo por la XP creció de forma exponencial entre 1999 y 2001. Ese entusiasmo fue lo que engendró y guio la revolución del desarrollo ágil. Hasta el día de hoy, la XP sigue siendo el mejor definido y el más completo de todos los métodos ágiles. Las prácticas de ingeniería en su núcleo son el foco de atención de esta sección sobre las disciplinas.

EL CÍRCULO DE LA VIDA

La figura I.1 muestra el círculo de la vida de Ron Jeffries, que recoge las prácticas de la XP. Las disciplinas que tratamos en este libro son las cuatro del centro y la del extremo izquierdo.

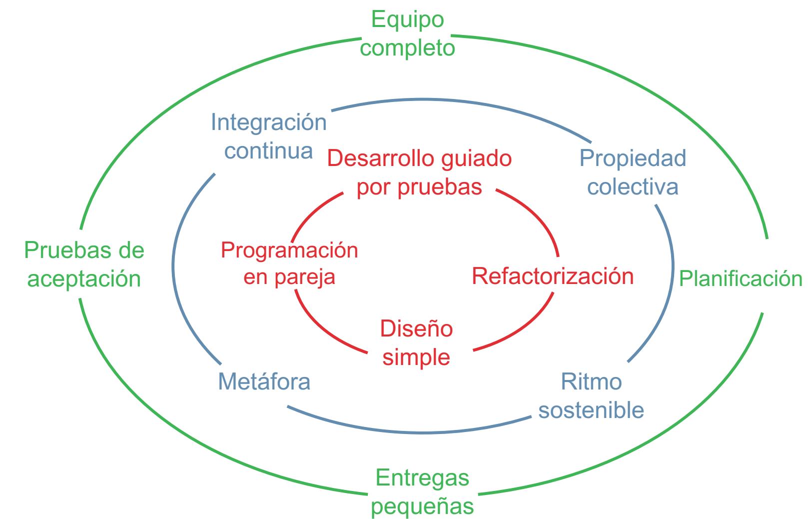

Figura I.1. El círculo de la vida; las prácticas de la XP.

Las cuatro del centro son las prácticas de ingeniería de la XP: desarrollo guiado por pruebas (*Test-driven development*, TDD), refactorización, diseño simple y programación en pareja (que denominaremos programación colaborativa). La práctica en el extremo izquierdo, las pruebas de aceptación, es la más técnica y centrada en la ingeniería de las prácticas de empresa de la XP.

Estas cinco prácticas están entre las disciplinas fundamentales de la artesanía de software.

DESARROLLO GUIADO POR PRUEBAS

El desarrollo guiado por pruebas es la disciplina que actúa como eje. Sin él, las otras disciplinas son imposibles o impotentes. Por esa razón, las dos siguientes secciones que describen el TDD representan casi la mitad de las páginas de este libro y son muy técnicas. Esta organización puede parecer mal equilibrada. De hecho, a mí también me lo parece, y le di muchas vueltas a qué hacer al respecto.

No, eso no ha sido una errata. Quería que quedase muy, muy claro. ¡Con esa *suite* de pruebas, puede limpiar el código de forma segura!

Y, si puede limpiar el código de manera segura, limpiará el código, y también lo hará el resto del equipo, porque a nadie le gusta el desorden.

La regla del *boy scout*

Si tiene esa *suite* de pruebas a la que confiaría su vida profesional, entonces puede seguir de manera segura esta sencilla directriz:

Libere el código más limpio de lo que estaba cuando lo reservó.

Imagine que todo el mundo hiciese eso. Antes de liberar el código, hicieron en él un pequeño acto de amabilidad. Lo limpiaron un poquito.

Imagine que cada vez que se liberase el código, este estuviese un poco más limpio. Imagine que nadie liberase nunca el código peor de lo que estaba, sino siempre mejor.

¿Cómo sería mantener un sistema así? ¿Qué pasaría con las estimaciones y los plazos si el sistema estuviese cada vez más limpio con el tiempo? ¿Cómo de largas serían sus listas de fallos? ¿Necesitaría una base de datos automatizada para mantener esas listas de fallos?

Esa es la razón

Mantener el código limpio. Limpiar continuamente el código. Por eso practicamos el desarrollo guiado por pruebas, para poder estar orgullosos del trabajo que hacemos.

Para poder mirar el código y saber que está limpio. Para que sepamos que cada vez que tocamos ese código, se vuelve mejor de lo que era antes. Y para poder llegar a casa por la noche, mirarnos en el espejo y sonreír, sabiendo que hoy hemos hecho un buen trabajo.

La cuarta ley

Hablaré mucho más de la refactorización en capítulos posteriores. Por ahora, quiero afirmar que la refactorización es la cuarta ley del TDD. A partir de las tres primeras leyes, es fácil ver que el ciclo del TDD implica escribir una cantidad muy pequeña de código de prueba que falla y, después, escribir

una cantidad muy pequeña de código de producción que pasa la prueba que fallaba. Podríamos imaginar un semáforo que alterna entre el rojo y el verde cada pocos segundos.

Pero, si permitiésemos que el ciclo continuase de esa forma, entonces el código de pruebas y el de producción se degradarían con rapidez. ¿Por qué? Porque a los humanos no se nos da bien hacer dos cosas a la vez. Si nos centramos en escribir una prueba que falle, es poco probable que sea una prueba bien escrita. Si nos centramos en escribir código de producción que pase la prueba, es poco probable que sea un buen código de producción. Si nos centramos en el comportamiento que queremos, no nos centraremos en la estructura que queremos.

No se engañe. No puede hacer las dos cosas al mismo tiempo. Ya es bastante difícil conseguir que el código se comporte de la forma que queremos. Es demasiado difícil escribirlo para que se comporte bien y tenga la estructura correcta. Así, seguimos el consejo de Kent Beck:

Primero haz que funcione. Luego hazlo bien.

Por tanto, añadimos una nueva ley a las tres leyes del TDD: la ley de la refactorización. Primero, escribimos una pequeña cantidad de código de prueba que falle. Después, escribimos una pequeña cantidad de código de producción que pase. Luego, limpiamos el desorden que acabamos de crear.

El semáforo tiene un nuevo color: rojo → verde → refactorizar (figura 2.1).

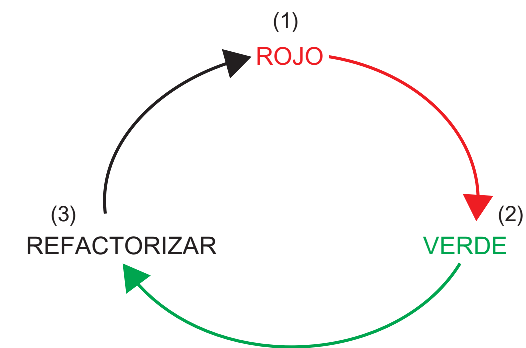

Figura 2.1. Rojo → verde → refactorizar.

Es probable que haya oído hablar de la refactorización y, como ya he dicho, le dedicaremos mucho tiempo en futuros capítulos. De momento, deje que desmienta algunos mitos e ideas equivocadas:

La primera prueba que falla será el caso degradado de una matriz vacía:

```
public class SortTest {

   @Test
   public void sorted() throws Exception {
      assertEquals(asList(), sort(asList()));
   }
   private List<Integer> sort(List<Integer> list) {
      return null;
   }
}
```

Obviamente, esto falla, pero es fácil hacer que la prueba se pase:

```
private List<Integer> sort(List<Integer> list) {
   return new ArrayList<>();
}
```

Subiendo un escalón más en la degradación, probamos con una lista que tiene un entero:

```
assertEquals(asList(1), sort(asList(1)));
```

Esto falla con claridad, pero podemos hacer que se pase haciendo el código de producción un poco más general:

```
private List<Integer> sort(List<Integer> list) {
   return list;
}
```

Es una monada, ¿a que sí? Y ya hemos visto este truco antes, en el ejemplo de los factores primos en el capítulo 2. Parece relativamente común que las dos primeras pruebas en un problema dado se resuelvan devolviendo la respuesta más degradada, seguida del argumento de entrada.

La siguiente prueba es trivial porque ya se pasa: dos elementos en orden. Podría argumentarse que ni siquiera deberíamos tener que escribirla porque no es una prueba que falla, pero es agradable ver cómo se pasan estas pruebas.

```
assertEquals(asList(1, 2), sort(asList(1, 2)));
```

Si invertimos el orden de la matriz de entrada, la prueba fallará: dos elementos no están en orden.

```
assertEquals(asList(1, 2), sort(asList(2, 1)));
```


Para hacer que se pase, vamos a tener que hacer algo un poco inteligente. Si la matriz de entrada tiene más de un elemento en ella y los dos primeros elementos de la matriz no están en orden, deberíamos intercambiarlos:

```
private List<Integer> sort(List<Integer> list) {
   if (list.size() > 1) {
      if (list.get(0) > list.get(1)) {
         int first = list.get(0);
         int second = list.get(1);
         list.set(0, second);
         list.set(1, first);
      }
   }
   return list;
}
```

Quizá vea adónde nos dirigimos con esto. Si es así, no destripe la sorpresa para los demás. Y recuerde también este momento; volveremos sobre ello en la siguiente sección.

Las dos siguientes pruebas ya se pasan. En la primera, la matriz de entrada ya está en orden. En la segunda, los dos primeros elementos no están en orden, y nuestra solución actual los intercambia.

```
assertEquals(asList(1, 2, 3), sort(asList(1, 2, 3)));
assertEquals(asList(1, 2, 3), sort(asList(2, 1, 3)));
```

La siguiente prueba que falla es la de tres elementos con los dos segundos desordenados:

```
assertEquals(asList(1, 2, 3), sort(asList(2, 3, 1)));
```

Conseguimos que se pase poniendo nuestro algoritmo de comparación e intercambio en un bucle que recorre hacia abajo la longitud de la lista:

```
private List<Integer> sort(List<Integer> list) {
   if (list.size() > 1) {
      for (int firstIndex=0; firstIndex < list.size()-1; firstIndex++) {
         int secondIndex = firstIndex + 1;
         if (list.get(firstIndex) > list.get(secondIndex)) {
            int first = list.get(firstIndex);
            int second = list.get(secondIndex);
            list.set(firstIndex, second);
            list.set(secondIndex, first);
         }
      }
   }
   return list;
}
```

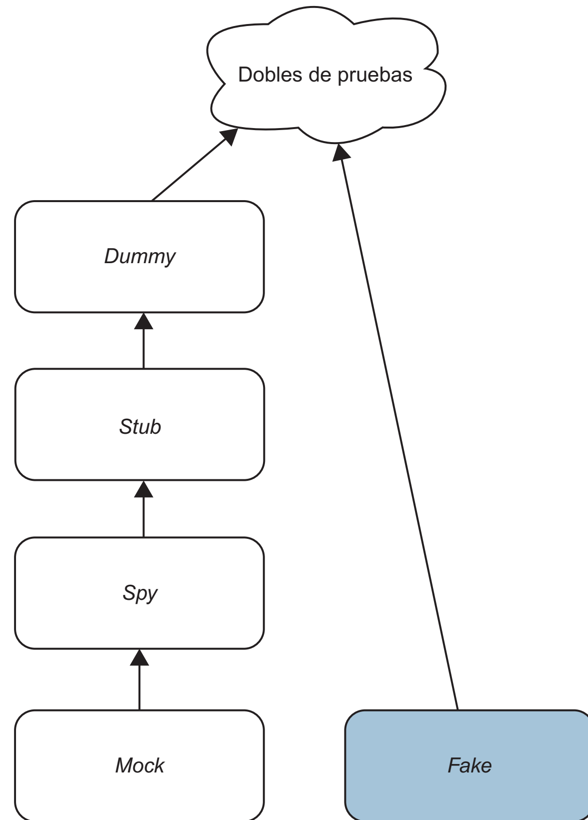

Figura 3.7. The *fake*.

¿Cómo probamos ese sistema?

Creamos un *fake*. El *fake* que construimos era un COLT cuya interfaz de conmutación se sustituía por un simulador. Ese simulador fingía marcar líneas de teléfono y fingía medirlas. Después, informaba de resultados brutos enlatados según el número de teléfono que se le pidiese probar.

El *fake* nos permitía probar el software de comunicación, control y análisis del SAC sin tener que instalar un auténtico COLT en la central de conmutación de una compañía telefónica real o incluso tener que instalar hardware de conmutación real y líneas de teléfono "reales".

Hoy en día, un *fake* es un doble de pruebas que implementa algún tipo de reglas de negocio rudimentarias de manera que las pruebas que usan ese *fake* puedan seleccionar cómo se comporta este. Quizá un ejemplo sirva mejor como explicación:


```
@Test
public void badPasswordAttempt_loginFails() throws Exception {
   Authenticator authenticator = new FakeAuthenticator();
   LoginDialog dialog = new LoginDialog(authenticator);
   dialog.show();
   boolean success = dialog.submit("user", "bad password");
   assertFalse(success);
}
@Test
public void goodPasswordAttempt_loginSucceeds() throws Exception {
   Authenticator authenticator = new FakeAuthenticator();
   LoginDialog dialog = new LoginDialog(authenticator);
   dialog.show();
   boolean success = dialog.submit("user", "good password");
   assertTrue(success);
}
```

Estas dos pruebas usan el mismo `FakeAuthorizer`, pero le pasan una contraseña diferente. Las pruebas esperan que `bad password` falle el intento de inicio de sesión y que `good password` tenga éxito.

El código para `FakeAuthenticator` debería ser fácil de visualizar:

```
public class FakeAuthenticator implements Authenticator {
   public Boolean authenticate(String username, String password)
   {
      return (username.equals("user") &&
            password.equals("good password"));
   }
}
```

El problema con los *fakes* es que, a medida que la aplicación crezca, siempre habrá más condiciones que probar. Como consecuencia, los *fakes* tienden a crecer con cada nueva condición probada. Al final, pueden ser tan grandes y complejos que necesiten sus propias pruebas.

Yo escribo *fakes* con muy poca frecuencia porque no me fío de que no crezcan.

El principio de incertidumbre del TDD

Mockear o no *mockear*, esa es la cuestión. Bueno, no. En realidad, la cuestión es cuándo *mockear*.

Hay dos escuelas de pensamiento a este respecto: la de Londres y la de Chicago, de las que hablaremos al final de este capítulo. Pero, antes de entrar en eso, tenemos que definir por qué esto es un problema, en primer lugar. Es un problema por el principio de incertidumbre del TDD.

```
        reporter = new NewCasesReporter();
    }

    @Test
    public void countyReport() throws Exception {
        String report = reporter.makeReport("" +
            "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
            "c2, s2, 2, 2, 2, 2, 2, 2, 2, 7");
        assertEquals("" +
                "County     State     Avg New Cases\n" +
                "======     =====     =============\n" +
                "c1         s1        1.86\n" +
                "c2         s2        2.71\n\n" +
                "s1 cases: 14\n" +
                "s2 cases: 21\n" +
                "Total Cases: 35\n",
            report);
    }

    @Test
    public void stateWithTwoCounties() throws Exception {
        String report = reporter.makeReport("" +
            "c1, s1, 1, 1, 1, 1, 1, 1, 1, 7\n" +
            "c2, s1, 2, 2, 2, 2, 2, 2, 2, 7");
        assertEquals("" +
                "County     State     Avg New Cases\n" +
                "======     =====     =============\n" +
                "c1         s1        1.86\n" +
                "c2         s1        2.71\n\n" +
                "s1 cases: 35\n" +
                "Total Cases: 35\n",
            report);
    }

    @Test
    public void statesWithShortLines() throws Exception {
        String report = reporter.makeReport("" +
            "c1, s1, 1, 1, 1, 1, 7\n" +
            "c2, s2, 7\n");
        assertEquals("" +
                "County     State     Avg New Cases\n" +
                "======     =====     =============\n" +
                "c1         s1        2.20\n" +
                "c2         s2        7.00\n\n" +
                "s1 cases: 11\n" +
                "s2 cases: 7\n" +
                "Total Cases: 18\n",
            report);
    }
}
```


Las pruebas nos dan una buena idea de lo que está haciendo el programa. La entrada es una cadena CSV. Cada línea representa un condado y tiene una lista del número de casos nuevos de COVID por día. La salida es un informe que muestra la media móvil de siete días de casos nuevos por condado y proporciona algunos totales para cada estado, junto con un total general.

Está claro que no interesa extraer métodos de esta función grande y horrible. Empezaremos por el bucle de la parte superior. Ese bucle hace todos los cálculos para todos los países, así que probablemente deberíamos ponerle un nombre como `calculateCounties`. Sin embargo, seleccionar ese bucle e intentar extraer un método produce el cuadro de diálogo que se muestra en la figura 5.1.

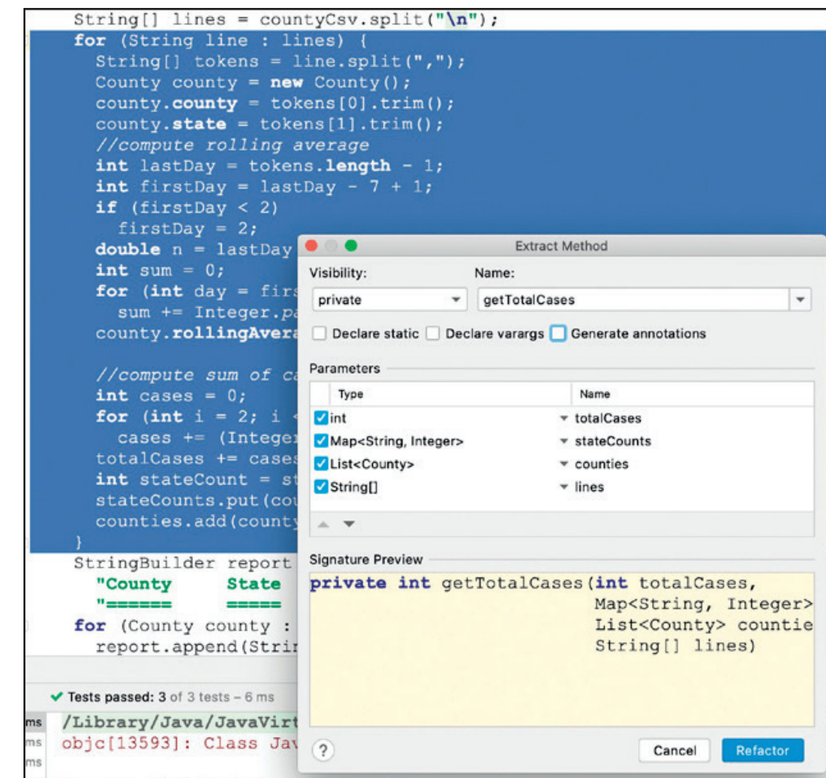

Figura 5.1. Cuadro de diálogo para extraer métodos.

El IDE quiere poner a la función el nombre `getTotalCases`. Hay que reconocérselo a los autores del IDE: se han esforzado mucho para intentar sugerir nombres. El IDE se ha decidido por ese nombre porque el código después del bucle necesita el número de casos nuevos y no hay forma de obtenerlo si esta nueva función no lo devuelve.

Debe decir "no"

Espero que diga "no" cuando la respuesta sea "no".

Una de las cosas más importantes que puede decir un programador es "¡No!". Si se dice en el momento adecuado, en el contexto adecuado, esta respuesta puede ahorrar a su jefe cantidades enormes de dinero y evitar fracasos horribles y mucha vergüenza.

Eso no da carta blanca para ir por ahí diciendo que no a todo. Somos ingenieros: nuestro trabajo es encontrar un camino hacia el sí. Pero, a veces, el sí no es una opción. Somos los únicos que podemos determinar esto. Los únicos que lo sabemos. Por tanto, depende de nosotros decir que no cuando la respuesta es realmente "no".

Supongamos que nuestro jefe nos pide que acabemos algo para el viernes. Después de pensarlo el tiempo necesario, nos damos cuenta de que no hay opciones razonables de completar la tarea para el viernes. Debemos hablar de nuevo con nuestro jefe y decirle "no". Sería inteligente decirle también que podemos tenerlo acabado para el martes siguiente, pero debemos mantenernos firmes en cuanto al hecho de que tenerlo el viernes no es factible.

A los directores no suele gustarles oír "no". Insisten. Puede que se enfrenten a nosotros. Quizá nos griten. La confrontación emocional es una de las herramientas que emplean algunos directores.

No debe ceder ante eso. Si la respuesta es no, debe aferrarse a ella y no sucumbir a la presión.

Y tenga mucho cuidado con la maniobra del "¿Puedes al menos intentarlo?". Parece razonable que nos pidan que lo intentemos, ¿no? Pero no es razonable en absoluto, porque ya estamos intentándolo. No hay nada nuevo que podamos hacer para convertir ese "no" en un "sí", así que decir que lo intentaremos es mentira, sin más.

Espero que, si la respuesta es "no", diga "no".

Aprendizaje agresivo continuo

La industria del software es muy dinámica. Podemos debatir acerca de si debería ser así, pero no podemos discutir que es así. Lo es y, por tanto, debemos ser aprendices agresivos de forma continuada.

Es probable que el lenguaje que está utilizando hoy no sea el mismo que esté utilizando dentro de 5 años. Es probable que el *framework* que está utilizando hoy no sea el que esté utilizando el año que viene. Para prepararse para esos cambios, preste atención a todo lo que está cambiando a su alrededor.

A menudo, a los programadores se nos ha aconsejado[1] aprender un lenguaje nuevo cada año. Es un buen consejo. Además, elija un lenguaje que tenga un estilo con el que no esté familiarizado. Si nunca ha escrito código en un lenguaje de tipado dinámico, aprenda uno. Si nunca ha escrito código en un lenguaje declarativo, aprenda uno. Si nunca ha escrito Lisp, Prolog o Forth, apréndalos.

1. David Thomas y Andrew Hunt, *The Pragmatic Programmer: From Journey to Mastery* (Addison-Wesley, 2020).

Al tomar esta decisión, consultó con su jefe, Adriaan van Wijngaarden. A Dijkstra le preocupaba que nadie hubiese identificado una disciplina o ciencia de la programación y que, por tanto, nadie le tomaría en serio. Su jefe respondió que Dijkstra podría ser una de las personas que la convirtieran en una ciencia.

Para alcanzar ese objetivo, a Dijkstra le interesaba la idea de que el software era un sistema formal, un tipo de matemáticas. Concluyó que el software podía convertirse en una estructura matemática, algo como los elementos de Euclides, un sistema de postulados, demostraciones, teoremas y lemas. Por tanto, se dedicó a crear el lenguaje y la disciplina de las demostraciones de software.

DEMOSTRAR LA CORRECCIÓN

Dijkstra se dio cuenta de que solo había tres técnicas que podíamos usar para demostrar la corrección de un algoritmo: enumeración, inducción y abstracción. La enumeración se utiliza para demostrar que dos sentencias en una secuencia o dos sentencias seleccionadas por una expresión booleana son correctas. La inducción se usa para demostrar que un bucle es correcto. La abstracción se emplea para descomponer sentencias en porciones probables más pequeñas.

Si suena difícil, lo es.

Como ejemplo de lo difícil que es, he incluido un programa Java simple para calcular el resto de un número entero (véase la figura 12.2), y la demostración escrita a mano de ese algoritmo (véase la figura 12.3).[2]

```
public static int remainder(int numerator, int denominator) (
  assert(numerator > 0 && denominator > 0);
  int r = numerator;
  int dd = denominator;
  while(dd<=r)
    dd *= 2;
  while(dd != denominator) {
    dd /= 2;
    if(dd <= r)
      r -= dd;
  }
  return r;
}
```

Figura 12.2. Un programa Java simple.

2. Esta es una traducción a Java de una muestra del trabajo de Dijkstra.

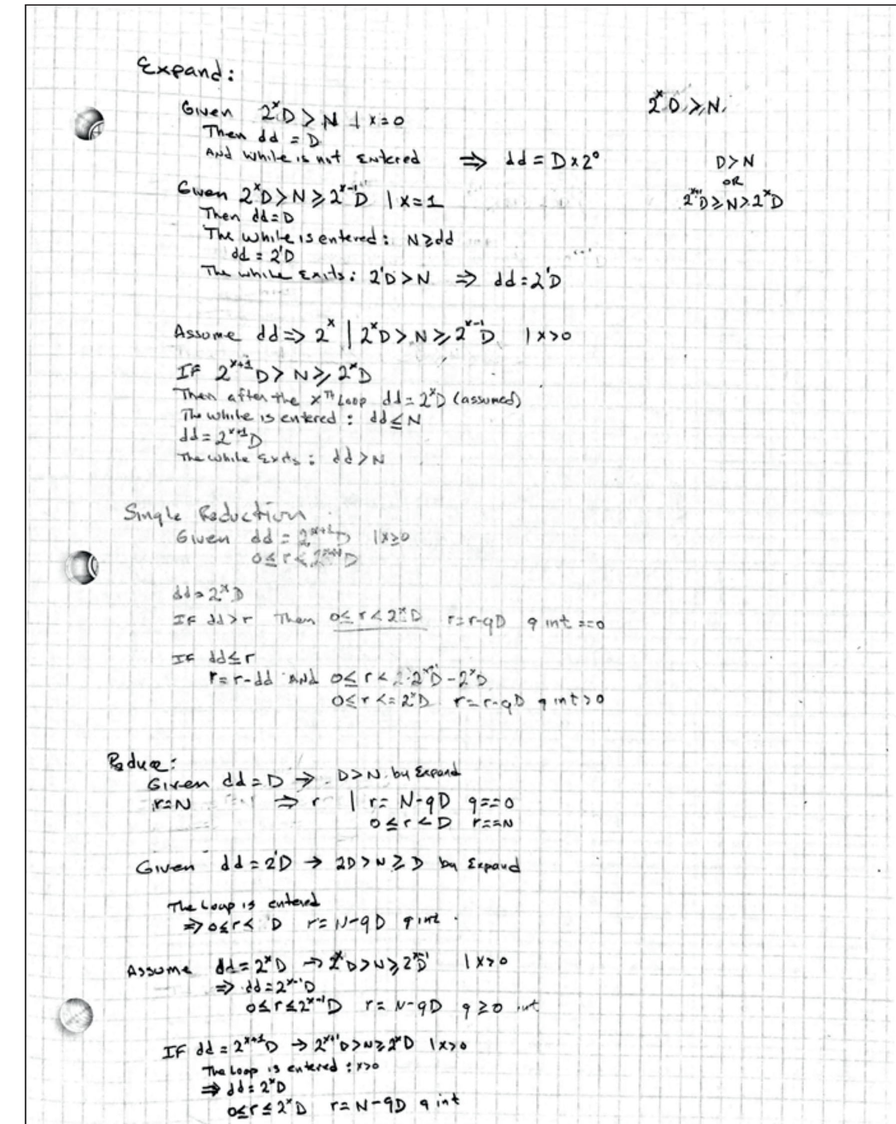

Figura 12.3. Demostración escrita a mano del algoritmo.

Creo que ve el problema con este enfoque. De hecho, es algo de lo que Dijkstra se quejó con amargura:

> *Por supuesto, no me atrevería a sugerir (¡al menos en el presente!) que es deber del programador proporcionar una demostración así cada vez que escribe un simple bucle en su programa. Si fuese el caso, nunca escribiría ningún programa de ningún tamaño en absoluto.*

13

Integridad

Varias promesas del juramento están relacionadas con la integridad.

Ciclos pequeños

Promesa 4. Realizaré liberaciones pequeñas y frecuentes para no obstaculizar el progreso de otros.

Realizar liberaciones pequeñas significa solo cambiar una cantidad pequeña de código para cada liberación. Puede que el sistema sea grande, pero los cambios graduales en ese sistema son pequeños.

La historia del control del código fuente

Volvamos un momento a los sesenta. ¿Cómo es el control del código fuente cuando ese código fuente se perfora en un mazo de tarjetas? (véase la figura 13.1).

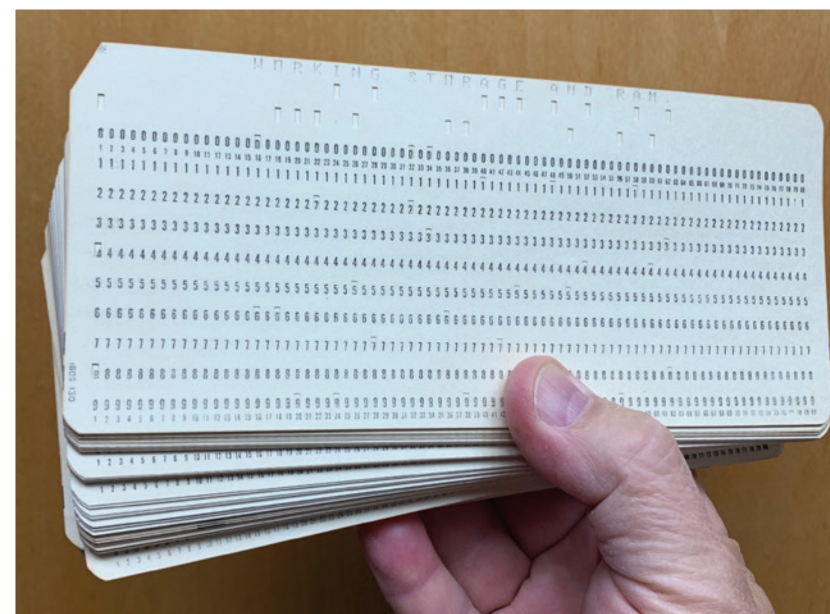

Figura 13.1. Tarjeta perforada.

El código fuente no se almacena en un disco. No está "en el ordenador". Su código fuente está, literalmente, en sus manos.

¿Qué es el sistema de control de versiones del código fuente? Es el cajón de su escritorio.

Verá, este cliente era muy pequeño. Su instalación era, literalmente, la configuración más pequeña posible para un conmutador digital. Es más, resultaba (por casualidad) que la configuración de su equipo eliminaba prácticamente toda la complejidad que el CCU/CMU resolvía.

Resumiendo, puse en marcha una unidad única para un fin especial en dos semanas. La llamamos el pCCU.

La lección

Estas dos historias son ejemplos de la oscilación enorme que pueden tener las estimaciones. Por una parte, subestimé la vectorización de los chips por un factor de seis. Por otra parte, encontramos la solución para el CCU/CMU en una vigésima parte del tiempo esperado.

Aquí es donde entra en juego la honestidad. Porque, honestamente, cuando las cosas van mal, pueden ir muy muy mal y, cuando van bien, a veces pueden ir muy muy bien.

Esto hace que la estimación sea todo un desafío.

Exactitud

A estas alturas, debería estar claro que una estimación para un proyecto no puede ser una fecha. Una sola fecha es demasiado precisa para un proceso que puede desviarse por un factor de 6 o, incluso, de 20.

Las estimaciones no son fechas, son rangos. Son distribuciones de probabilidad.

Las distribuciones de probabilidad tienen una media y una anchura, a veces denominada desviación estándar o sigma.

Vamos a fijarnos primero en la media.

Encontrar el tiempo de completitud medio esperado para una tarea compleja es cuestión de sumar todos los tiempos de completitud medios para todas las subtareas. Y, por supuesto, es algo recursivo. Las subtareas pueden estimarse sumando todos los tiempos de las subsubtareas. Esto crea un árbol de tareas que se conoce a menudo como estructura de desglose de trabajo (EDT).

Vale, eso está muy bien. El problema, sin embargo, es que no se nos da muy bien identificar todas las subtareas y subsubtareas y subsubsubtareas.

Por lo general, pasamos por alto algunas. Como la mitad, o así.

Lo compensamos multiplicando la suma por dos. A veces por tres. O, a veces, incluso más.

Kirk: *¿Cuánto tiempo necesitamos para las reparaciones?*

Scotty: *Ocho semanas, señor, pero, como no disponemos de ocho, tendremos que hacerlo en dos.*

Kirk: *Señor Scott, ¿divide usted siempre los presupuestos de sus reparaciones entre cuatro?*

Scotty: *¡Naturalmente, señor! Si no, echaría al traste mi reputación como técnico milagroso.*[3]

Este *fudge factor* de 2, 3, incluso 4, suena a trampa. Y, por supuesto, lo es. Pero también lo es el propio acto de estimar.

Solo hay una manera real de determinar cuánto tiempo va a llevarnos algo, y es hacerlo. Cualquier otro mecanismo es hacer trampa.

Así pues, asumámoslo, vamos a hacer trampa. Vamos a utilizar la EDT y, después, a multiplicar por un F, y ese F está entre 2 y 4, dependiendo de nuestra confianza y nuestra productividad. Eso nos dará nuestro tiempo medio para completarlo.

Los directores nos preguntarán cómo hemos llegado a esas estimaciones y tendremos que decírselo. Y, cuando les hablemos de ese *fudge factor*, van a pedirnos que lo reduzcamos dedicando más tiempo a la EDT.

Eso es justo, y deberíamos estar dispuestos a hacerlo. Sin embargo, también deberíamos advertirles que el coste de desarrollar una EDT completa es equivalente al coste de la tarea en sí. De hecho, para cuando hayamos desarrollado la EDT completa, también habremos completado el proyecto, porque la única manera de enumerar de verdad todas las tareas es ejecutar las tareas que conocemos en orden para descubrir el resto de forma recursiva.

Por tanto, asegúrese de poner su esfuerzo para la estimación en una caja de tiempo e informe a sus directores de que perfeccionar el *fudge factor* va a resultar caro.

Existen muchas técnicas para estimar las subtareas en las hojas del árbol de la EDT. Podría utilizar puntos función o una métrica de complejidad similar, pero a mí siempre me ha parecido que estas tareas se estiman mejor por puro instinto.

Por lo general, lo hago comparando las tareas con otras que ya haya completado. Si creo que una tarea es el doble de difícil, multiplico ese tiempo por dos.

---

3. *Star Trek III: En busca de Spock*, dirigida por Leonard Nimoy (Paramount Pictures, 1984).

Ingeniería de software/Programación

La artesanía del código limpio

Cómo escribir código del que se sienta orgulloso… todos los días

En ***La artesanía del código limpio***, el legendario Robert C. Martin («Uncle Bob») ha escrito los principios que definen la profesión (y el arte) del desarrollo de software. Reúne las disciplinas, los estándares y la ética que necesita para entregar un software sólido y efectivo y para estar orgulloso de todo el software que escribe.

Robert, autor del superventas *Código limpio*, ofrece una perspectiva pragmática, técnica y prescriptiva de las disciplinas que forman los cimientos de la artesanía de software. Explica los estándares, mostrando cómo las expectativas que el mundo tiene sobre los desarrolladores difieren a menudo de las que tienen ellos mismos, y nos ayuda a sincronizarlas. Termina con la ética de la profesión del programador, describiendo las promesas fundamentales que todos los desarrolladores deberían hacer a sus colegas, a sus usuarios y, sobre todo, a sí mismos.

«… un recordatorio oportuno y humilde de la complejidad cada vez mayor de nuestro mundo de la programación y de cómo debemos al legado de la humanidad, y a nosotros mismos, practicar un desarrollo ético. Tómese su tiempo para leer ***La artesanía del código limpio****. […] Tenga este libro siempre a mano. Deje que se convierta en un viejo amigo (su tío Bob, su guía) a medida que avanza por este mundo con curiosidad y valor».*

— Stacia Heimgartner Viscardi, CST y Agile Mentor

Con las aportaciones de Uncle Bob, todos los programadores y sus directores pueden entregar de manera consistente código que genera confianza en vez de socavarla, confianza entre los usuarios y entre las sociedades que dependen del software para su supervivencia.

- ***Avanzar hacia la «Estrella Polar» de la verdadera artesanía de software: el estado de saber cómo programar bien.***
- ***Orientación práctica y específica para aplicar cinco disciplinas esenciales: desarrollo guiado por pruebas, refactorización, diseño simple, programación colaborativa y pruebas de aceptación.***
- ***Cómo los desarrolladores y los equipos pueden fomentar la productividad, la calidad y el valor.***
- ***El verdadero significado de la integridad y el trabajo en equipo entre programadores, y diez promesas específicas que todo profesional del software debería hacer.***

Robert C. Martin («Uncle Bob»), programador profesional desde 1970, es cofundador de *cleancoders.com* para ofrecer formación en línea a desarrolladores de software, y fundador de Uncle Bob Consulting LLC para proporcionar asesoramiento, formación y servicios de desarrollo de habilidades a empresas importantes de todo el mundo. Antiguo redactor jefe de *C++ Report*, fue el primer presidente de la Alianza Ágil. Entre los *best sellers* de Martin se incluyen *Código limpio, El limpiador de código, Arquitectura limpia* y *Desarrollo ágil esencial*.

Imagen de cubierta: © Tom Cross/iStockphoto LP/Getty Images

ISBN 978-84-415-4499-4
2315169
9 788441 544994
ANAYA MULTIMEDIA
www.anayamultimedia.es