Enfoque para Pruebas de Unidad Basado en la Generación Aleatoria de Objetos

Tesis presentada para obtener
el grado de Magister en
Ingeniería de Software

Facultad de Informática
Universidad Nacional de La Plata

Febrero 2014

Autor: Lic. Pablo Andrés Barrientos

Director: Dr. Claudia Pons

Agradecimientos

Quisiera mencionar en primer lugar al amor de mi vida: Tere. Tenerla al lado mío cada día de mi vida es saber que puedo contar con alguien que me quiere por sobre todas las cosas. Por suerte nos tenemos el uno al otro para ir aprendiendo a vivir. Hemos superado juntos momentos difíciles, esos donde todo parece tirar para abajo, y tambien hemos compartido muchos momentos felices que es la única forma en que se puede compartir la felicidad.

También quiero agradecer a toda mi familia: mi viejo Ruben, mi vieja María Teresa, mis hermandas Anabelia, Guillermina y María Sabina, por el apoyo a la distancia para seguir progresando mientras la vida transcurre. También a mis sobrinos que tanto alegran tan solo por existir y poder escucharlos y verlos crecer.

Quisiera agradecer a Claudia Pons, la persona que elegí para ser directora de este trabajo de tesis, y que siempre confió en mis criterios y decisiones, dando consejos y ofreciendo su ayuda cuando cuando fueron necesarios.

Es grato mencionar a los amigos y colegas que me ayudaron desinteresadamente a completar el trabajo de tesis, colaborando en las tareas de comparación, que son una de las bases de sustento del trabajo realizado. Son además excelentes e invaluables personas. Ellos son: Mandy Siu, Victor Toledo, Mariano Pilotto, David Vara y Andres Sayago.

Agradezco finalmente a quienes me inspiraron, estuvieron a mi lado y/o me dieron fuerzas (quizás sin saberlo) durante mi carrera y mi vida. A mis amigos de siempre: Eduardo, Juan Pablo, David G., Mariano, Valeria, Eugenia y Esteban; y también a los nuevos que encontré durante este tiempo y que tanto me ayudaron. Especialmente a Suzanne-Gabrielle Gill y Mark Gill, que tan desinteresadamente nos ayudaron a que “la gran mudanza” y la adaptación no fueran cuesta arriba.

Pablo A. Barrientos
La Plata, 02 de Febrero de 2014

Este documento fue procesado usando el sistema de macros LaTeX 2ε.

Prefacio

El testing del software es una tarea crucial y a la vez muy desafiante dentro del proceso de desarrollo de software. El testing permite encontrar errores y problemas del software contra la especificación del mismo y cumple un rol fundamental en el aseguramiento de la calidad del producto.

Entre los tipos de pruebas que se pueden realizar al software están las pruebas de unidad, carga, integración y funcionales. Cada una de ellas tiene distintos objetivos y son realizadas en diferentes etapas del desarrollo del software. En el primer tipo mencionado, se desarrollan pruebas a componentes individuales de un sistema de software. Los desarrolladores especifican y codifican pruebas para cubrir todos o al menos una parte significativa de los posibles estados/configuraciones del artefacto o unidad de software, para simular el entorno del componente y descubrir la presencia de errores o "*bugs*". Dado que escribir todas esas pruebas de forma manual es costoso, las pruebas de unidad son generalmente realizadas de manera ineficiente o simplemente dejadas de lado. El panorama es aún peor, más allá del esfuerzo, porque el testing no puede ser usado para probar la usencia de errores en el software sino tan solo la presencia. Por eso es necesario atacar el problema desde diferentes enfoques, cada uno teniendo sus fortalezas y ventajas.

Actualmente existen muchas técnicas para hacer testing de software, y la mayoría de ellos se basan en la automatización de pasos o caminos de ejecución, con valores fijos o componentes predefinidos (*hard-coded*) o estáticos, y condiciones específicas. En este trabajo de maestría, se presenta un enfoque para pruebas de unidad en la programación orientada a objetos, basado en la generación de objetos de manera aleatoria. El fundamento básico de este enfoque propuesto es el testing aleatorio. También se presenta una herramienta de testing de unidad que usa el enfoque dicho, y que fue escrita en un lenguaje orientado a objetos de amplia difusión.

El testing aleatorio (*RT* o *random testing*) como técnica no es nueva. Tampoco lo es la generación de valores aleatorios para pruebas. En el paradigma funcional, existe una herramienta muy conocida para probar especificaciones sobre funciones llamada *QuickCheck*. Ésta herramienta (escrita en

Haskell) y sus ideas subyacentes son usadas como fundamento para la herramienta creada en este trabajo. La herramienta desarrollada en el presente trabajo cubre además características que existen en el paradigma orientado a objetos de manera inherente, tales como el estado de los objetos (en particular los objetos *singleton* con estado), clases abstractas e interfaces, que no existen en la programación funcional pura.

La constribución de este trabajo de maestría es la presentación de una forma alternativa de realizar tests de unidad en la programación orientada a objetos (POO), basada en un trabajo anterior para el paradigma funcional. También se presenta una herramienta llamada *YAQC4J* que plasma esas ideas en un lenguaje orientado a objetos de amplia difusión. Finalmente se incluyen ejemplos que ilustran el uso de la herramienta, y se presenta una comparación con herramientas existentes que han intentado implementar el enfoque de testing. Este trabajo está dirigido a los desarrolladores de software interesados en conocer soluciones alternativas para el testing de unidad, y al mismo tiempo una forma complementaria a las ya existentes para pruebas de unidad.

Keywords: Pruebas de unidad, testing aleatorio, QuickCheck, generación aleatoria de objetos, herramienta de testing, testing de software

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

"Software testing can be used to show the presence of bugs, but never to show their absence"
Edsger W. Dijkstra [Dij72]

Los sistemas actuales incluyen requerimientos de gran y creciente complejidad. La alta complejidad implica que un sistema puede tener potencialmente infinitas combinaciones de entradas y salidas resultantes. Además en muchos casos es muy difícil cubrir todos los posibles caminos dentro del código fuente de estos sistemas de forma manual o usando las herramientas actuales de testing.

El tipo de pruebas usado mayormente en el desarrollo de software es el de pruebas de unidad, donde se trata de descubrir errores nuevos o resultantes de modificaciones a los componentes lógicos más pequeños ya existentes. Esta opción es la más económica ya que el costo de encontrar errores a medida que se avanza en las etapas de un proyecto es cada vez mayor [BB01]. Además es necesario asegurar cierto grado de correctitud al nivel más pequeño para poder proceder con mayor confianza hacia pruebas de integración funcional y funcionales.

En el caso de la programación orientada a objetos, el componente funcional más chico es la clase y en particular, cada uno de sus métodos. Las pruebas de unidad se centran por lo tanto en estos componentes. Las pruebas de unidad en la programación orientada a objetos fueron popularizadas por Ducasse [Duc, Ken] con su herramienta SUnit en el lenguaje Smalltalk, que inspiró luego muchas otras herramientas similares para otros lenguajes como JUnit [TLMG10] para Java.

En las pruebas de unidad de JUnit (y SUnit), el desarrollador escribe una clase de test que contiene varios métodos de test y métodos que permiten crear y luego destruir un contexto general para cada uno de los tests. La ma-

yoría de los métodos dentro de la clase de test tienen el objetivo de verificar una o más post condiciones de la ejecución de un método. Es necesario entonces escribir tantos métodos de tests como posibles caminos existan dentro del método a testear, para verificar que las postcondiciones se cumplen para cada unos de esos caminos. Y para que un método recorra un camino particular de ejecución, será necesario que los colaboradores del objeto que recibe el mensaje y los parámetros recibidos tengan una determinada *configuración*.

Cada método de test se realiza codificando tres secciones principales: *arrange*, *act*, y *assert*. En la sección de *arrange*, el objeto que se desea testear es creado, inicializado y se crean los parámetros que se enviarán en el llamado al método a testear. Lo normal es que el desarrollador escriba el código necesario para crear los objetos que se necesitan tanto para crear el objeto, como los parámetros para invocar el método a probar, o incluso aquellos que se necesita inyectar como dependencias. Todos estos objetos tendrán valores fijos en cada ejecución del test. También muchas veces se crean *objetos mock*[1] [FMPW04] como alternativa. En la sección de *act*, simplemente el método/comportamiento es invocado con sus correspondientes parámetros. Y finalmente, en la sección de *assert*, se verifican las post-condiciones (estado del objeto, valores de salida o excepciones que deben producirse). En este punto es importante notar que muchos de los métodos dentro de la clase de test compartirán las secciones de *act*, y *assert*, y que sólo variarán en la sección de *arrange*.

Este enfoque de pruebas de unidad se toma para cubrir todos los caminos dentro de un método. Sin embargo, muchas veces debido a la complejidad del comportamiento y/o del objeto en sí, o simplemente por la falta de tiempo, la tarea de escribir tests de unidad no se realiza de manera completa. Es decir, no se cubren todos los caminos y no se llega a una cobertura del código cervcana al 100 %. Como resultado, muchos caminos no son ejecutados. En el peor de los casos, los caminos no cubiertos son los más usados en tiempo de ejecución, por lo que las pruebas escritas son bastante inútiles en términos de la cobertura real del código.

Para finalizar, es importante nombrar las características que una prueba de unidad en JUnit debe cumplir:

- Automatic: las pruebas deben poder correrse de manera automática, sin más intervención humana que la de iniciarlo y ver el resultado.
- Thorough: los tests deben probar todo aquello que es problable que falle. Aunque es una propiedad subjetiva, algunas métricas como la

[1] Pseudoobjetos. Son objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar al objeto real en la prueba

cobertura de los caminos de ejecución pueden dar una medida de la completitud de las pruebas.

- Repeatable: las pruebas deben poder correrse infinitas veces, en cualquier order, y producir los mismos resultados cada vez.

- Independent: los tests deben ser independientes entre sí. No se puede dar el caso que un test dependa de que otro corra antes para ser correcto. La prueba debe centrarse además en un comportamiento y condiciones particulares.

- Professional: significa que las pruebas deben respetar el estándar de calidad del código real, teniendo un buen diseño y factorización.

- Fast: las pruebas de unidad deben ser rápidas. Se espera correr las pruebas varias veces por día, por lo cual una prueba no debe durar más que unos pocos segundos.

- Isolated: las pruebas de unidad no deben depender de un contexto externo a ella, como ser la disponibilidad de un archivo o de conexión a una red.

En este trabajo se propone un enfoque *complementario* al descripto, basado en la idea de *random testing* (RT) o testing aleatorio [Ham94]. El testing aleatorio no es una técnica nueva sino que fue propuesta ya hace mucho tiempo atrás [DN84, BM83]. Se ha demostrado que puede ser una técnica efectiva en la búsqueda de errores en el software [FM00, AHJW06] y por lo tanto una técnica alternativa para encontrar defectos en el software.

Para poder entender la utilidad del testing aleatorio, podemos analizar cualquier software que reciba dos enteros como entrada. La cardinalidad del conjunto de entradas posibles para ese software sería:

$$2^{32} * 2^{32} = 2^{64}$$

Sería imposible escribir esa cantidad de casos de prueba, con lo cual la opción es escribir casos de prueba para determinadas entradas decididas por el desarrollador. En el testing aleatorio en cambio, se generan objetos o valores de prueba aleatoriamente que se utilizan como entrada para los métodos de test. Además cada prueba es corrida varias veces para tratar de falsificar la especificación (o propiedad) de la unidad de software. Algunas herramientas no sólo incluyen la generación de los datos de entrada, sino que además generan las postcondiciones e invariantes. Ese tipo de herramientas se detallan en el Capítulo 7. El testing aleatorio puede dar una valuación empírica de la

calidad del software indicando, por ejemplo, que *el software testeado no falla más de una vez en 10.000 ejecuciones*, o que el software testeado tiene un tiempo promedio de falla mayor a 10.000 horas.

En este trabajo se presenta la herramienta *YAQC4J* (**Y**et **A**nother **Q**uick **C**heck for **J**ava), para la generación aleatoria de objetos de prueba y ejecución de pruebas de unidad bajo la filosofía de random testing. Esta herramienta tiene como objetivo verificar fácilmente las especificaciones sobre los métodos de una clase, para detectar errores a un costo bajo. El enfoque propuesto de testing no es un reemplazo de otras técnicas de pruebas de unidad, pero sí un complemento útil de las mismas y que cubra sus deficiencias (al igual que otras herramientas de testing aleatorio [CPO+11]). En este enfoque, las pruebas de unidad de software orientado a objetos tienen una sección de *arrange* extremadamente corta o nula. Como consecuencia, cada método de test se asemeja a una especificación funcional del comportamiento, ya que el desarrollador debe enfocarse principalmente en definir el oráculo (la salida esperada).

YAQC4J fue escrita en Java y está basada en las ideas detrás de *QuickCheck* , pero ataca además los problemas generados de la migración de las ideas originales a un entorno orientado a objetos (ver Capítulo 4, donde se describen dichos problemas). En este sentido, no existe herramienta similar que haya sido exitosa en la traducción de los conceptos de *QuickCheck* al paradigma de objetos, y haya atacado además los problemas inherentes de dicho paradigma de la forma en que se ha realizado, y esta es la mayor contribución técnica de este trabajo.

Es creencia del autor que así como no existe una *bala de plata* para el desarrollo del software [Bro87], tampoco lo hay para el testing. Sin embargo el problema puede ser atacado con diferentes enfoques, y el testing aleatorio es uno de ellos.

Capítulo 2

Antecedentes

En el paradigma funcional, Claessen y Hughes [CH00] crearon una herramienta (una librería de combinadores) para formular y probar propiedades en los programas escritos en Haskell [HHJW07]. Esa herramienta, llamada *QuickCheck* , ha sido traducida a muchos otros lenguajes funcionales como Erlang [AHJW06], Curry [CF08] y Scheme [sch] debido a su enorme éxito y aceptación. *QuickCheck* ha sido también traducida a lenguajes orientados a objetos como Java [JCh, Qui] y C++ [qcp].

La herramienta que se presenta en el presente trabajo fue escrita en Java y está basada en las ideas detrás de *QuickCheck* , pero ataca además los problemas generados de la migración de las ideas originales a un entorno orientado a objetos (ver Capítulo 4, donde se describen dichos problemas). En este sentido, no existe herramienta similar que haya sido exitosa en la traducción de los conceptos de *QuickCheck* al paradigma de objetos, y haya atacado además los problemas inherentes de dicho paradigma de la forma en que se ha realizado, y esa es otra de las contribuciones de este trabajo.

A continuación, se presenta una breve descripción de *QuickCheck* , para entender por un lado la filosofía del testing aleatorio que se traduce a la programacióon orientada a objetos con la herramienta, y a su vez la herramienta desarrollada en el presente trabajo.

QuickCheck fue escrito en Haskell como una herramienta (y enfoque) de testing. Con esta herramienta se escriben especificaciones para probar diferentes propiedades sobre una función de un tipo dado. Las pruebas se generan específicamente para intentar falsificar propiedades, invocando la función varias veces. Por ejemplo, la propiedad testreverse, fue definida de la siguiente manera:

```
testreverse xs = reverse (reverse xs) == xs
```

Dicha propiedad que habla de las listas de cualquier tipo, establece que

la lista resultante de revertir dos veces los elementos de cualquier lista es igual a la lista original. En términos funcionales, aplicar la función reverse sobre una lista y luego aplicar la misma funcion sobre la lista resultante debe retornar una lista igual a la lista original. Ésta propiedad debe valer para cualquier lista xs. *QuickCheck* trata de falsificar la propiedad probando la especificación varias veces (por defecto lo hace 100 veces) con listas generadas aleatoriamente. Para generar estas listas, se restringe a que el argumento xs sea de la clase Arbitrary (clase en el sentido que se le da en Haskell). Instancias de Arbitrary proveen todo lo que la herramienta necesita para generar valores aleatorios, y se define de la siguiente manera:

```
class Arbitrary a where
  arbitrary   :: Gen a
  coarbitrary :: a -> Gen b -> Gen b
```

La función coarbitrary se usa para generar funciones aleatorias sobre el tipo. Como este trabajo no trabaja sobre el concepto de funciones, ya que se trata de una implementación en el paradigma orientado a objetos, su explicación se descarta.

La función que debe implementarse para obtener valores aleatorios de un tipo dado es arbitrary. Para poder definir esta función para cualquier tipo que se desee, *QuickCheck* provee varios combinadores (funciones) que podrían usarse para crear generadores (Gen a). Algunos de ellos son:

- elements:: [a] → Gen a, que dada una lista de elementos de un tipo, elige aleatoriamente un elemento de la lista y lo retorna.
- choose:: Random a ⇒ (a, a) → Gen a, que dado un rango de valores de un tipo, elige aleatoriamente uno de ellos y lo retorna.
- oneof:: [Gen a] → Gen a, que dada una lista de generadores, elige aleatoriamente uno de ellos y lo retorna.
- frequency[(Int, Gen a)] → Gen a, que dada una lista de frecuencias y generadores, elige uno de ellos basado en la frecuencia que ocurre y lo retorna.

Para ilustrar con un ejemplo, si se tiene el tipo de datos:

```
data TernaryLogic = Yes | No | Unknown
```

Se puede definir que ese tipo de datos implementa la clase Arbitraty, definiendo la función arbitrary y creando un generador usando el combinador choose de la siguiente manera:

```
instance Arbitrary TernaryLogic where
  arbitrary = do n <- choose (0, 2) :: Gen Int
                 return $ case n of
                             0 -> Yes
                             1 -> No
                             2 -> Unknown
```

Una vez que una propiedad está especificada, puede ser testeada usando la función `quickCheck`. Por ejemplo, al correr esa función sobre la propiedad testreverse:

```
quickCheck testreverse
```

la herramienta retornará el siguiente resultado:

```
OK, passed 100 tests.
```

indicando que la propiedad fue verificada 100 veces con valores generados aleatoriamente, y pasó satisfactoriamente para todos los casos.

Por el contrario, cuando la propiedad falla, *QuickCheck* muestra un contra ejemplo. Si se definiera:

```
prop_RevId xs = reverse xs == xs
  where types = xs::[Int]
```

al evaluar la propiedad (`quickCheck prop_RevId`), podría indicar, por ejemplo:

```
Falsifiable, after 1 tests:
[-3,15]
```

Los generadores tienen un parámetro implícito que controla el tamaño de los objetos generados. Los generadores de valores aleatorios interpretan ese párametro de manera distinta segun el caso. Algunos de ellos lo ignoran y otros lo toman como límite superior. En el ejemplo del generador de listas, lo interpreta como el tamaño máximo de las listas. El desarrollador puede controlar el valor de ese parámetro de ser necesario, y para ello puede usar la función `resize :: Int -> Gen a -> Gen a`. Luego, `resize n g` invoca al generador `g` con `n` como su parámetro de tamaño.

QuickCheck provee también combinadores para clasificar los valores generados aleatoriamente. El combinador classify toma un predicado y un String que funciona como etiqueta. Al final de la ejecución de los test, genera un reporte con el porcentaje de valores que pudieron ser clasificados dentro de esa categoría o label. Por ejemplo, si definiéramos:

```
prop_Insert x xs =
ordered xs ==>
classify (ordered (x:xs)) "at-head"$
  classify (ordered (xs++[x])) "at-tail"$
ordered (insert x xs)
  where types = x::Int
```

El resultado de testear esa propiedad de la función `insert` sería:

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
58% at-head, at-tail.
22% at-tail.
4% at-head.
```

Notar que si las categorías o clasificadores no son disjuntos, un mismo valor podría ser clasificado dentro de la misma categoría.

Si se quisieran clasificaciones disjuntas (particiones del tipo / grupo de valores), existe el combinador collect, que permite clasificar los valores generados a partir de una expresión dada. Por ejemplo:

```
prop_Insert x xs =
ordered xs ==> collect (length xs)$
           ordered (insert x xs)
      where types = x::Int
```

hace una partición del conjunto de valores generados a partir de la longitud de la lista. Luego de ejecutado el test, se reportan los porcentajes de valores para cada longitud de lista.

```
Main> quickCheck prop_Insert
OK, passed 100 tests.
58% 0.
26% 1.
13% 2.
3% 3.
```

Otra característica de *QuickCheck* es la posibilidad de restringir los valores generados aleatoriamente para una propiedad. Por ejemplo, si quisiéramos indicar que la inserción sobre una lista ordenada no debería desordenar la lista resultante, entonces debemos restringir a que la lista original esté ordenada. Para ello se escribe:

```
prop_Insert x xs = ordered xs ==> ordered (insert x xs)
```

QuickCheck descarta los casos de listas generadas aleatoriamente que no satisfacen el antecedente del operador `==>` (también conocido como suchAs), e intenta generar nuevos valores hasta que llegue a probar la propiedad en 100 valores distintos (o falle la propiedad con una de ellas). Para evitar caer en un loop infinito porque la restricción no se puede cumplir con los valores generados aleatoriamente, se establece un límite de cantidad de intentos.

En el caso visto, podría aparecer un mensaje similar a:

`Arguments exhausted after 97 tests.`

si la herramienta pudo generar 97 listas que cumplieran la restricción pero no pudo generar otro valor que la cumpliera y superó el límite de intentos.

Por último, existen propiedades cuantificadas, que tienen la forma:

```
forAll <generator> $ \<pattern> -> <property>
```

El combinador `forAll` permite el uso de generadores específicos (que corresponde al argumento `<generator>`) para los tipos de datos. Esto permite controlar qué tipo de valores se generan para la propiedad, y puede ser muy útil para evitar el problema del valores que no cumplen la restricción impuesta por el operador `==>`. En el caso de la propiedad `prop_Insert`, si `orderedList` es un generador de listas ordenadas, podemos escribir:

```
prop_Insert2 x = forAll orderedList $
                      \xs -> ordered (insert x xs)
  where types = x::Int
```

Ésta es una importante estrategia que el desarrollador debe utilizar si quiere que sus tests sean eficientes en la generación de datos útiles.

Las características de *QuickCheck* que se mencionaron en los párrafos anteriores han sido traducidas en la herramienta que se presenta en este trabajo, como se explicará en la Section 3. Para más detalles sobre *QuickCheck* para Haskell se recomienda leer el trabajo original de Claessen [CH00], dado que este trabajo no tiene como objetivo describirlo en profundidad sino los aspectos más sobresalientes y esenciales.

Capítulo 3

Fundamentos y descripción de la herramienta

JUnit [TLMG10][junb] es el conocido framework de testing usado en Java para escribir pruebas de unidad. JUnit es bastante simple en términos de su API. Como fue mencionado en el Capítulo 1, está basado en el framework de Smalltalk llamado SUnit [Duc][Ken].

El framework tiene un ciclo de ejecución donde se corre la clase de test método a método sin un orden predefinido y aleatorio. Los métodos de test son aquellos anotados con la anotación `@Test` o los que tienen el prefijo `test` en versiones anteriores. Cada vez que un test se ejecuta, se ejecutan previamente los métodos que inicializan el contexto general, que son aquellos anotados con `@Before`, o el método `setUp` en versiones anteriores de JUnit. Al finalizar la ejecución del método de test, se destruye el contexto general, ejecutando los metodos anotados con `@After`, o el método (`tearDown` en versiones anteriores. En la Figura 3.1 se muestra el ciclo de ejecución típico de JUnit para una clase de test. También provee la posibilidad de ejecutar un sólo método particular dentro de la clase de test, pero siempre respetando el ciclo que la figura Figura 3.1.

Cada método representa una *configuración* especial; un estado y contexto especial en el que el objeto es testeado. Cada método de test manda un único mensaje al objeto bajo prueba (sección *act*) que se corresponde con el método o comportamiento de la clase a testear. Previamente el objeto receptor debe ser creado, y también deben ser creados los objetos que son argumento del mensaje (sección *arrange*). El desarrollador de las pruebas tiene dos alternativas en JUnit: inicializar el objeto y su contexto en el método `setUp` (o aquellos anotados con `@Before`), o bien escribir las líneas de código necesarias para la creación del objeto y su contexto dentro del mismo método de test. En la Figura 3.2 se muestra un test sencillo en JUnit. En el mismo,

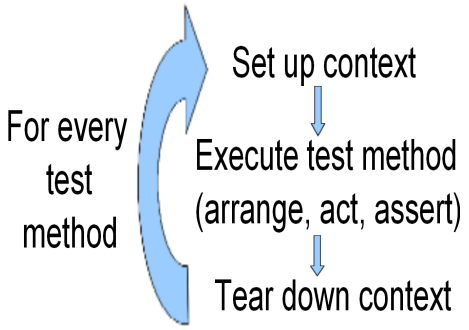

Figura 3.1: Ciclo de ejecución de métodos de test en JUnit

los métodos que inician y destruyen el contexto general no hacen nada y sólo están de ejemplo. Además se indica explícitamente qué *test runner* se va a utilizar, ya que JUnit permite ejecutar los tests con otro *runner* que no sea el estándard.

```
@RunWith(org.junit.runners.BlockJUnit4ClassRunner.class)
public class TestClass {
  @Before
  public void init() {
     // set general context or basic objects to be tested
  }

  @After
  public void destroy() {
     // clean up the context
  }

  @Test
  public void testReverse() {
    final String str = "some hardcoded string";
    StringBuffer buff = new StringBuffer(str);
    assertEquals(str, buff.reverse().reverse().toString());
  }
}
```

Figura 3.2: Ejemplo sencillo de test en JUnit

Idealmente, se debería probar el método reverse para cualquier argumento de modo de asegurar alta calidad del software. Sin embargo eso no es posible

en JUnit.

El enfoque propuesto en este trabajo, junto con la herramienta desarrollada (*YAQC4J*), entra en este punto para proveer la creación aleatoria de objetos. En este nuevo enfoque, el desarrollador se centra en probar el comportamiento del objeto y no en el código que se necesita anteriormente para que el objeto esté listo para poder recibir el mensaje. Incluso no debe escribir tantos métodos como *configuraciones* posibles pueda haber (que pueden ser muchas, incluso infinitas). La herramienta corre cada método de test un número determinado de veces (que es configurable), y cada vez provee objetos generados aleatoriamente diferentes. Para indicar que un objeto es generado aleatoriamente, se usan tests parametrizados [TS05] con anotaciones opcionales de configuración para cada parámetro del método de test. Sólo en el caso que no se especifique la necesidad de crear objetos aleatorios se correrá el test una sola vez. Ésta es la filosofía básica tomada de *QuickCheck* . Ahora los tests no son *configuraciones* sino *especificaciones* sobre el comportamiento de un objeto.

Para correr un test de la forma en que *QuickCheck* lo hace, cada método se corre más de una vez. Con ese objetivo se creó un *runner* especial que forma parte del *core* de la herramienta, llamado QCCheckRunner. Este runner es subclase de uno de los runners existentes de JUnit. Esto implica que la herramienta se sentará sobre la infraestructura que provee JUnit para las pruebas de unidad, y que, por ejemplo, podrá ser posible usar la herramienta en entornos de programación con soporte para JUnit como ser Eclipse [eclb], Netbeans[net] o IntelliJ[int]. El runner tendrá en cuenta los parámetros de cada test, y si uno de ellos no tiene parámetros, interpretará que el test es un clásico ejemplo de testing de JUnit, donde no hay que generar objetos aleatorios. En ese caso correrá el test sólo una vez.

El *runner* maneja el ciclo de vida de los tests basado en metadata dada por la anotación @Configuration, y genera objetos aleatorios para cada argumento de los métodos de tests. Con *YAQC4J* , el desarrollador sólo necesita indicar qué objetos deben ser generador aleatoriamente poniéndolos como parámetros del método de test. La herramienta ejecuta el método varias veces y en cada oportunidad genera objetos aleatorios, y ejecuta además los métodos que inicializan y destruyen el contexto general. En la Figura 3.3 se muestra el típico ciclo de ejecución de *YAQC4J* .

En la Figura 3.4 se muestra un ejemplo sencillo de test con *YAQC4J* , donde se especifica el *runner* con la anotación de JUnit @RunWith y además el método de test recibe un String generado aleatoriamente.

El desarrollador puede además crear generadores aleatorios de instancias para una clase particular. De la misma forma que en *QuickCheck* se usaba el combinador forAll para indicar el uso de generadores particulares para

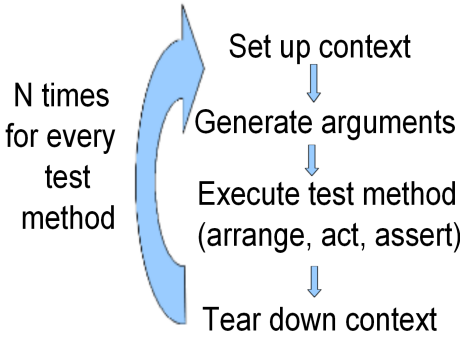

Figura 3.3: Ciclo de ejecución de métodos de test en *YAQC4J*

```
@RunWith(QCCheckRunner.class)
public class SimpleTest {
  @Test
  public void testReverse(final String str) {
        StringBuffer buff = new StringBuffer(str);
        assertEquals(str, buff.reverse().reverse().toString());
  }
}
```

Figura 3.4: Ejemplo sencillo de test en *YAQC4J*

un test, en *YAQC4J* se puede indicar qué generador se puede usar en cada método de prueba o en la clase de test. Esta meta información del método de prueba se especifica con la anotación @Generator a nivel de clase o método de test. En la Figura 3.5 se puede ver un ejemplo de metadata junto con la anotación @Generator. En este ejemplo se pueden ver además otros elementos útiles. En la anotación @Configuration se puede indicar los tamaños mínimos y máximos que toman los generadores para crear objetos con las propiedades minsize y maxsize. El parámetro maxsize es similar al párametro implicito de los generadores de *QuickCheck* descripto en el Capítulo 2. Como ocurre en esa herramienta, la interpretación del tamaño del objeto depende exclusivamente de cada generador. También se puede especificar la distribución (distribution) aleatoria que se usa para elegir valores aleatorios. La generación de objetos aleatorios y el papel que juegan estos parámetros se explicarán con más detalle más adelante.

YAQC4J provee algunos generadores para tipos estándares y algunos basados en los generadores de *QuickCheck* . Algunos de los generadores que incluye son:

```
@RunWith(QCCheckRunner.class)
@Configuration(maxsize = 50)
@Generator(generator = IntegerGen.class, klass = int.class)
public class SimpleTest {
    @Test
    @Configuration(distribution=
                        InvertedNormalDistribution.class)
    public void classConfigurationTest(final int i) {
        assertTrue("@Configuration for class (or int generator)
                        failed", i >= -50 && i <= 50);
    }

    @Test
    @Configuration(maxsize = 5, minsize = -5)
    public void methodConfigurationTest(final int i) {
        assertTrue("@Configuration for methods (or "
        + "in generator) failed", i >= -5 && i <= 5);
    }
}
```

Figura 3.5: Metadata en tests

- ArrayGen, que genera arreglos de algún tipo dado al construirse. Incluye la generación de arreglos de tipos primitivos.
- ElementGen, FrequencyGen y OneOfGen, que son las contrapartes en *YAQC4J* de los generadores element, frequency y oneOf de *QuickCheck* respectivamente.
- DateGen y CalendarGen, que generan instancias de java.util.Date y de java.util.Calendar respectivamente.
- ListGen, MapGen, QueueGen y SetGen, que generan instancias de java.util.List, java.util.Map, java.util.Queue y java.util.Set con objetos de algun tipo especificado.
- StringGen, ShortGen, LongGen, IntegerGen, FloatGen, DoubleGen, CharacterGen, ByteGen y BooleanGen, que generan instancias de las clases estandares respectivas del paquete java.lang.
- BigIntegerGen y BigDecimalGen, que generan instancias de java.math.BigInteger y de java.math.BigDecimal respectivamente.

YAQC4J provee además la posibilidad de definir generadores de objetos particulares para una clase dada. Los nuevos generadores deben implementar la interface Gen<T>, siendo T la clase del las instancias generadas aleatoriamente. En la Figura 3.6 se puede observar que sólo es necesario implementar un método (el que se usa para generar objetos del tipo T), que recibe tres parámetros:

- `Distribution`, que representa la distribución aleatoria usada para generar los objetos. Por defecto se usa la probabilidad uniforme basada en la clase java.util.Random, pero también se dispone de distribución uniforme, normal, normal inversa, negativa, y normal positiva. Adicionalmente, el desarrollador podría crear y usar su propia Distribution, subclasificando dicha clase.

- minsize, que puede ser usada como límite inferior para objetos de tipos específicos como números, árboles, Strings, etc.

- maxsize, que puede ser usada como límite superior para objetos de tipos específicos como coleccciones, mapas, etc.

```
public interface Gen<T> {
     T arbitrary(Distribution random,
                 long minsize, long maxsize);
}
```

Figura 3.6: La interfaz de los generadores

Los argumentos antes detallados pueden ser especificados como parte de la metadata de los tests. En la Figura 3.5 se puede ver un ejemplo de uso de metadata junto con la anotación @Generator para usar un generador específico. Nuevamente se puede ver que la metadata de configuración se puede usar tanto a nivel de clase como a nivel de método.

Se pueden definir muchos generadores para la misma clase. Los desarrolladores pueden escribir tantas ocurrencias de la anotación @Generator como les sea necesario, agrupándolas bajo la anotación @UseGenerators. Cuando la herramienta tiene que generar un objeto de determinado tipo, seleccionará aleatoriamente uno de los generadores disponibles (aquellos definidos en el método, en la clase y los que por defecto se registran para los tipos básicos de java). Ésta selección de un generador se hace cada vez que el método es ejecutado.

Muchas veces podría requerirse una precondición o restricción sobre un objeto generado de forma aleatoria. En *QuickCheck* se tiene el combinador ==>, y en *YAQC4J* se tiene el método Implication.imply(boolean condition) con ese objetivo. Como fue discutido en la Sección 2, es conveniente que el desarrollador cree un generador particular que considere la restricción sobre los valores que espera. De todas maneras, *YAQC4J* provee soporte para Implication.imply(boolean condition), de forma tal que verifica que se cumpla la restricción y, de no cumplirse, trata de generar un nuevo objeto aleatoriamente que la cumpla. Dado que la generación de un objeto aleatorio que satisfaga la condición del imply puede no ocurrir rápidamente, se puede establecer un límite máximo de fallas toleradas antes que la prueba se considere fallida. Éste valor máximo de fallas se establece como metadata en la propiedad maxArgumentsFails de la anotación @Configuration y tiene un valor de 100 por defecto. Adicionalmente, también se puede especificar en @Configuration el número de veces que un método de tests se tratará de correr mediante la propiedad test, que tiene 100 como valor default.

Como fue mencionado antes, *YAQC4J* provee diferentes generadores para muchos de los tipos estándares de Java, incluyendo tipos primitivos y enumerativos (generador EnumGen). También provee muchos generadores que son equivalentes a los combinadores de *QuickCheck* (oneof, element, frequency, etc). Por otra parte, se proveen generadores sólo presentes en la programación orientada a objetos:

- NullGen, que genera valores nulos (null) con determinada probabilidad.
- IdentityGen, que retorna siempre un mismo objeto. Este generador no tiene cuidado del estado del objeto, con lo cual existe un riesgo enorme sobre las propiedades de repetibilidad e independencia de los tests si no se usa cuidadosamente. Una modificación hecha sobre el objeto en una ejecución persiste a través de las ejecuciones posteriores.
- CloneGen, que retorna un clon de un objeto original.

Finalmente, un generador llamado TransformerGenerator permite mapear un objeto generado aleatoriamente a otro de otro tipo. Éste generador es bastante útil, y fue usado para definir el generador de java.util.Calendar usando el generador de java.util.Date, que había sido definido previamente.

Un ejemplo de un generador predefinido se puede observar en la Figura 3.7.

En la Figura 3.8 se puede ver un ejemplo de uso de *YAQC4J* para crear un test abstracto parametrizado (genérico) para testear propiedades que cualquier clase de Java debe cumplir al redefinir los métodos equals y hashCode.

```
public final class ShortGen implements Gen<Short> {
    public Short arbitrary(final Distribution random,
              final long minsize, final long maxsize) {
       long maxSize = Math.min(maxsize, Short.MAX_VALUE);
       long minSize = Math.max(minsize, Short.MIN_VALUE);
       return Short.valueOf((short)
              Arbitrary.choose(random, minSize, maxSize));
    }
}
```

Figura 3.7: Generador para la clase `java.math.BigInteger`

Es importante destacar que la herramienta soporta tipos genéricos en los tests, y que por lo tanto este tipo de tests pueden servir para cualquier clase, siempre que cumpla con las restricciones de la especificación del tipo genérico.

YAQC4J fue testeado con JUnit y también se usó la herramienta misma para auto testearla. Se han escrito casos de prueba que usan las clases básicas de la herramienta para testearse a sí mismas y al resto del código (principalmente los generatores definidos). Se encontraron algunos errores, uno de los cuales forzó un cambio de diseño de la herramienta. El generador de ArrayList fallaba en determinados casos, cuando aleatoriamente seleccionaba el constructor que recibe como parámetro el tamaño de la lista, y generaba como argumento un número negativo. Se tuvo que sacar el generador de enteros como generador por defecto, y se definió un generador de enteros positivos PositiveIntegerGen que se puede usar cuando se generan listas y se puede ver en la Figura 3.9. También se definieron generadores de valores positivos para Long, Double y Float.

Algo importante a destacar es que la herramienta no es intrusiva en el código del desarrollador. Es decir, no es necesario agregar anotaciones ni modificar código fuente del negocio para usar la herramienta. Ésto es una diferencia respecto de *QuickCheck* , donde se debía especificar explícitamente que el tipo debía ser de la clase Arbitrary. Finalmente, la herramienta tiene muy pocas dependencias con otros frameworks y librerías, y aquellos que fueron usados para desarrollar *YAQC4J* son populares y han sido usados y testeados por un largo tiempo. Como la herramienta se sienta sobre la infraestructura de JUnit, también puede ser usada en entornos de desarrollo que soportan JUnit como Eclipse o IntelliJ.

```
@RunWith(QCCheckRunner.class)
public class EqualsHashProperties<T> {
    public EqualsHashProperties() {
        Arbitrary.registerConstructorGeneratorFor((Class<?>)
            Arbitrary.getSuperclassTypeParameter(this.getClass()));
    }

    @Test
    public void reflexiveEquals(final T a) {
        imply(a != null);
        assertEquals(a, a);
    }

    @Test
    public void symmetricEquals(final T a, final T b) {
        imply(a != null && b != null);
        imply(a.equals(b));
        assertEquals(b, a);
    }

    @Test
    public void transitiveEquals(final T a, final T b, final T c) {
        imply(a != null && b != null && c != null);
        imply(a.equals(b));
        imply(b.equals(c));
        assertEquals(a, c);
    }

    @Test
    public void equalsHaveSameHashcode(final T a, final T b) {
        imply(a != null && b != null && a.equals(b));
        assertEquals(a.hashCode(), b.hashCode());
    }

    @Test
    public void hashcodeIsTheSame(final T a, final int times) {
        imply(a != null);
        int hc = a.hashCode();
        for (int i = 0; i < times; i++) {
            assertTrue(hc == a.hashCode());
        }
    }
}
```

Figura 3.8: Test abstracto y genérico para los métodos equals y hashcode

```
public class PositiveIntegerGen implements Gen<Integer> {
    private Gen<Integer> INT_GEN = new IntegerGen();

    public Integer arbitrary(final Distribution random,
           final long minsize, final long maxsize) {
        if (maxsize < 0){
          throw new GenerationError(
           "Cannot generate positive integers with maxsize = "
            + maxsize);
        }
        long max = Math.min(maxsize, Long.MAX_VALUE);
        return INT_GEN.arbitrary(random,
               Math.max(0, minsize), max);
    }
}
```

Figura 3.9: Generador de números enteros positivos

Capítulo 4

Características especiales en la POO

La programación funcional y la orientada a objetos tienen diferencias muy significativas. El lenguaje de implementación orientado a objetos evidencia esas diferencias respecto del lenguaje original en que fue escrito *QuickCheck* (Haskell). Java tiene efectos laterales, jerarquía de clases y herencia decomportamiento y estructura, interfaces, clases abstractas, e incluso objetos que son únicos (*singletons*) en el entorno o máquina virtual y muchas veces mutables (*stateful*).

En ésta sección se explica cómo se manejaron estas situaciones especiales anteriormente mencionadas, y se presentan algunos generadores especiales o comportamientos particulares de la herramienta.

4.1 Singletons

El patrón de diseño *singleton* [GHJV95] es usado para implementar el concepto matemático de singleton, e implica que una clase tiene sólo una instancia para la máquina virtual donde existe. Como consecuencia de ésto, la instancia es compartida por todos los objetos que la referencian y usan. El problema con este patrón de diseño ocurre cuando la única instancia tiene estado, pues ese estado es compartido. En el contexto de pruebas de unidad, el objeto singleton se comparte entre todos los métodos de test que lo tienen como parámetro o lo referencian en la misma ejecución. Ésto atenta directamente contra muchas de las propiedades deseables de las pruebas de unidad (por ejemplo, independencia – ver Capítulo 1), ya que puede llevar a resultados inesperados si uno de los tests modifica el estado del singleton y luego otro test lo usa. Para solucionar este problema, se crearon dos generadores

diferentes de objetos singletons para cada test, descriptos a continuación:

- SingletonGen<T>, es un generador abstracto genérico muy simple que accede en tiempo de ejecución a la variable de clase (típicamente llamada instance) que referencia a la instancia única de la clase y la setea en null. De ésta manera, la clase creará una nueva instancia cuando se le solicite la *única* instancia que genera. La variable de clase se asume que es instance, pero se puede especificar otro nombre de variable si se crea una subclases. También asume que el método es getInstance(), pero permite redefinirlo al ser subclasificada. Esta clase y la que se describe a continuación, son subclases de AbstractSingletonGen<T>, que provee la lógica necesaria para invocar al método de clase que retorna el singleton. Ambas implementaciones se pueden ver en la Figura 4.1 y la Figura 4.2.

```
public abstract class AbstractSingletonGen<T>
                                        implements Gen<T> {
    public static final String DEFAULT_METHOD_NAME =
                                            "getInstance";
    private Class<? extends T> className = null;
    private String singletonMethod;

    public AbstractSingletonGen() {
        this(DEFAULT_METHOD_NAME);
    }

    public AbstractSingletonGen(final String
                                    singletonMethodName) {
         this.setClassName((Class<? extends T>) Arbitrary
            .getSuperclassTypeParameter(this.getClass()));
         this.setSingletonMethod(singletonMethodName);
   }
}
```

Figura 4.1: Clase abstracta para generadores de singletons

- StatefulSerializableSingletonGen, cuando la única instancia de la clase es *stateful* e implementa la interfaz java.io.Serializable, se puede definir un generador que provee diferentes instancias (con nuevos estados) cada vez que un test lo necesita. El generador se basa en la capacidad de serializar el objeto para obtener una nueva instancia. Éste

```
public abstract class SingletonGen<T>
                        extends AbstractSingletonGen<T> {
    public static final String DEFAULT_FIELD_NAME = "instance";
    private String fieldName;

    public SingletonGen() {
        super();
        this.setFieldName(DEFAULT_FIELD_NAME);
    }

    public SingletonGen(final String singletonMethodName,
                        final String singletonField) {
        super(singletonMethodName);
        this.setFieldName(singletonField);
    }

    public final T arbitrary(final Distribution random,
                      final long minsize, final long maxsize) {
        Class<T> cls;
        Method method = null;
        try {
            cls = (Class<T>) Class.forName(
                  this.getClassName().getName(),
                  true, this.getClassName().getClassLoader());
            Field instance = cls.getDeclaredField(
                  this.getFieldName());
            instance.setAccessible(true);
            instance.set(cls, null);
            method = cls.getMethod(this.getSingletonMethod(),
                        new Class[0]);
            return (T) method.invoke(null, new Object[0]);
        } catch (ClassNotFoundException e) {
            throw new GenerationError(e);
        } catch (...){
        }
    }
}
```

Figura 4.2: Clase para generadores de singletons

generador debe usarse con mucho cuidado, y se recomienda en general usar el generador SingletonGen<T>, ya que provee la funcionalidad

para singletons *stateful* y *stateless*. Parte de la implementación de este generador se puede ver en la Figura 4.3.

```
public abstract class StatefulSerializableSingletonGen
             <T extends Serializable>
        extends AbstractSingletonGen<T> {

    public StatefulSerializableSingletonGen() {
         super();
    }

    public StatefulSerializableSingletonGen(
                         final String singletonMethodName) {
         super(singletonMethodName);
    }

    public final T arbitrary(final Distribution random,
                     final long minsize, final long maxsize) {
        Class<T> cls;
        try {
            Method method = null;
            cls = (Class<T>) Class.forName(
                     this.getClassName().getName(), true,
                     this.getClassName().getClassLoader());
            method = cls.getMethod(this.getSingletonMethod(),
                     new Class[0]);
            Object o = method.invoke(null, new Object[0]);
            return new CloneGen<T>((T) o).arbitrary(random,
                     minsize, maxsize);
        } catch (ClassNotFoundException e) {
             throw new GenerationError(e);
        } catch (SecurityException e) {
        }
    }
}
```

Figura 4.3: Clase para generadores de singletons stateful serializables

Para facilitar el uso de la herramienta, se definió una anotación especial para indicar que un parámetro es singleton, y que se debe usar un generador adecuado. Si un parámetro de test tiene la anotación IsSingleton, se creará una instancia particular de SingletonGen asociada al tipo del parámetro. El efecto

es el mismo que declarar un generador subclase de SingletonGen a nivel del método de test. La anotación tiene dos parámetros opcionales y con valores por defecto (fieldName y singletonMethodName), que se corresponden con los parámetros que necesita el generador.

4.2 Clases abstractas y jerarquías

Uno de los objetivos de la herramienta es simplificar la cantidad de código que los desarrolladores deben escribir para especificar sus tests. Eso favorecería la adopción de este enfoque para realizar tests de unidad. Con dicho propósito en mente, cuando un desarrollador no especifica o no tiene un generador definido para algún tipo, la herramienta intenta crear una instancia buscando generadores (si es que hay definidos) para alguna de sus subclases dentro de la jerarquía. Si existen generadores, elije aleatoriamente uno de ellos. De la misma manera se procede cuando la clase es abstracta. El soporte de ésta lógica está en parte en el runner de *YAQC4J* , y en parte en el generador ClassHierarchyOrInterfaceGen<T> cuya implementación puede observarse en la Figura 4.4. Si a pesar de todo no se encuentra un generador para un tipo, la herramienta usa un generador basado en los constructores de la clase, que será explicado en la Sección 4.4.

El generador de instancias de la interface java.util.Serializable, por ejemplo, se definió usando éste generador simplemente subclasificando el generador mencionado.

4.3 Interfaces

Cuando un argumento de un test es una interface, se tienen dos opciones: la primera es buscar en el *classpath* aquellas clases que implementan esa interface. Luego se buscan generadores de instancias de una de esas clases, para después seleccionar uno de manera aleatoria. La otra opción que provee la herramienta es generar un objeto proxy [GHJV95] que implementa la interface. El generador encargado de ese comportamiento es InterfaceInstancesGen<T>. Cuando se pide generar un objeto a este tipo de generador, éste crea un objeto capaz de recibir los mensajes de la interfaz, y retorna objetos de los tipos esperados. Los objetos que se retornan son generados de manera aleatoria si es posible y necesario. Esto significa que el método retornará un objeto si el tipo de retorno no es void y si el tipo tiene un generador. Si el tipo del objeto que se retorna es el mismo que la interfaz, se retorna null para evitar posibles loops infinitos.

```
public abstract class ClassHierarchyOrInterfaceGen<T> implements Gen<T> {
    private Class<T> subjectClass;

    public ClassHierarchyOrInterfaceGen() {
      Class<T> clazz = (Class<T>) Arbitrary.getSuperclassTypeParameter(this.getClass());
      if (clazz.isAnnotation() || clazz.isEnum() || clazz.isArray() ||
          clazz.isPrimitive()) {
         throw new IllegalArgumentException(
             "The constructor argument must be a class or interface");
      }
      this.setSubjectClass(clazz);
    }

    private Class<T> getSubjectClass() {
      return this.subjectClass;
    }

    public T arbitrary(final Distribution random, final long minsize, final long maxsize) {
      List<Class<?>> classes = null;
      List<Gen<?>> assignableGenerators = null;
      try {
         classes = ClassSearcher.getAssignableClasses(this.getSubjectClass());
      } catch (Exception e) {
          throw new GenerationError(e);
      }
      if (classes != null && !classes.isEmpty()) {
         assignableGenerators = new ArrayList<Gen<?>>();
           for (Class<?> assignableClass : classes) {
              if (!Modifier.isAbstract(assignableClass.getModifiers())) {
                  Set<Gen<?>> set = Arbitrary.getGeneratorsOf(assignableClass);
                  if (!set.isEmpty()) {
                        assignableGenerators.addAll(set);
                  }
               }
           }
           if (assignableGenerators.size() > 0) {
               int index = (int) Arbitrary.choose(random, 0,
                                                  assignableGenerators.size() - 1);
               return (T) assignableGenerators.get(index).arbitrary(random, minsize,
                                                                    maxsize);
           } else {
               throw new GenerationError("No generators available for classes " +
                    "in the class hierarchy");
           }
      } else {
         throw new GenerationError("No subclasses defined for this class");
      }
   }
}
```

Figura 4.4: Generador para jerarquías, clases abstractas e interfaces

En la Figura 4.5 se puede ver parte de la implementación del comportamiento descripto para la clase InterfaceInstancesGen:

```
public class InterfaceInstancesGen<T> implements Gen<T> {
    private Class<T> interfaze;
    private Distribution random;
    private long minsize;
    private long maxsize;

    public class InternalInvocationHandler implements InvocationHandler {
        public Object invoke(final Object proxy, final Method method,
                             final Object[] args) throws Throwable {
            if (method.getReturnType().equals(Void.TYPE) ||
                method.getReturnType().equals(
                        InterfaceInstancesGen.this.getInterfaze())) {
                return null;
            }
            Gen<T> generator = (Gen<T>) Arbitrary.getGeneratorFor(
                    InterfaceInstancesGen.this.getRandom(),
                    method.getReturnType());
            if (generator == null) {
                return null;
            } else {
                 return generator.arbitrary(
                        InterfaceInstancesGen.this.getRandom(),
                        InterfaceInstancesGen.this.getMinsize(),
                        InterfaceInstancesGen.this.getMaxsize());
            }
        }
   }

   public InterfaceInstancesGen(final Class<T> interfaze) {
        if (!interfaze.isInterface()) {
            throw new IllegalArgumentException(
                         "The given argument is not an interface.");
        }
        this.setInterfaze(interfaze);
   }

   public T arbitrary(final Distribution random, final long minsize,
              final long maxsize) {
        this.setMaxsize(maxsize);
        this.setMinsize(minsize);
        this.setRandom(random);
        InvocationHandler handler = new InternalInvocationHandler();
        return (T) Proxy.newProxyInstance(this.interfaze.getClassLoader(),
              new Class[] { this.getInterfaze() }, handler);
   }
}
```

Figura 4.5: Implementación de generador de proxy de interfaces

4.4 Generador basado en los constructores

Como fue mencionado anteriormente, se definió un generador genérico especial llamado ConstructorBasedGen<T> que crea instancias de cualquier clase basado en los constructores (públicos) que la clase tiene. El generador busca los constructores, selecciona uno aleatoriamente, y crea una instancia pasándole los argumentos al constructor, generados también aleatoriamente.

La implamentación de este generador puede verse en la Figura 4.6.

```
public class ConstructorBasedGen<T> implements Gen<T> {
    private Class<T> subjectClass;

    public ConstructorBasedGen(final Class<T> aClass) {
        if (aClass.isAnnotation()) {
            throw new IllegalArgumentException(...);
        }
        if (aClass.isEnum()) {
            throw new IllegalArgumentException(...);
        }
        if (aClass.isArray()) {
              throw new IllegalArgumentException(...);
        }
        this.setSubjectClass(aClass);
    }

    private Object[] generateArguments(final Distribution random,
            final Class<?>[] argumentsTypes) {
        Object[] arguments = new Object[argumentsTypes.length];
        for (int i = 0; i < argumentsTypes.length; i++) {
            Class<?> aClass = argumentsTypes[i];
            arguments[i] = Arbitrary.getInstanceOf(random, aClass);
        }
        return arguments;
    }

    public final T arbitrary(final Distribution random, final long minsize,
           final long maxsize) {
        Constructor<T>[] constructors = (Constructor<T>[]) this
             .getSubjectClass().getConstructors();
        if (constructors.length == 0) {
            if (this.getSubjectClass().isInterface()) {
                return new InterfaceInstancesGen<T>(
                    this.getSubjectClass()).arbitrary(random, minsize, maxsize);
            } else {
                 throw new GenerationError(this.getSubjectClass().toString()
                    + " has no constructor.");
            }
        }
        int index = (int) Arbitrary.choose(random, 0, constructors.length - 1);
        Constructor<?> constructor = constructors[index];
        Class<?>[] classes = constructor.getParameterTypes();
        Object[] arguments = this.generateArguments(random, classes);
        try {
             return (T) constructor.newInstance(arguments);
        } catch (IllegalArgumentException e) {
            ...
        }
    }
}
```

Figura 4.6: Implementación de generador basado en constructores

4.5 Generadores de transformación

Existen dos generadores interesantes que se basan en la transformación (generación) de objetos a partir de otro objeto generado aleatoriamente. Uno de ellos es TransformerGen<T, U>, que es abstracto y cuyo objetivo es facilitar la creación de nuevos generadores para un cierto tipo U, sabiendo que existe otro tipo T del cual se pueden obtener instancias. Como fue mencionado en la Sección 3, el generador para java.util.Calendar se definió subclasificando éste generador y usando el generador de java.util.Date (ver Figura 6.4).

El otro generador basado en la idea de transformación es

MethodTransformerGen<T>, cuya idea es *aplicar* n veces (n elegido aleatoriamente) un método definido en el transformador mismo sobre una instancia generada aleatoriamente del tipo T. La implementación es sencilla y puede verse en la Figura 4.7.

```
public abstract class MethodTransformerGen<T> implements Gen<T> {
    private Gen<T> originGenerator;

    public MethodTransformerGen(final Gen<T> originGenerator) {
        this.originGenerator = originGenerator;
    }

    public T arbitrary(Distribution random, long minsize,
                       long maxsize) {
        long times = Arbitrary.choose(random, minsize, maxsize);
        T origin = originGenerator.arbitrary(random, minsize,
                       maxsize);
        for (int i = 0; i < times; i++) {
           origin = this.transform(origin);
        }
        return origin;
    }

    protected abstract T transform(T object);
}
```

Figura 4.7: Implementación de un generador de transformación

Capítulo 5

Clasificación de objetos y aleatoriedad

En el capítulo 2 se mencionó que *QuickCheck* tiene dos combinadores para clasificar los valores generados aleatoriamente: collect y classify. Ambos combinadores fueron traducidos a la herramienta como parte de la propuesta de traducir el enfoque de testing aleatorio a la programación orientada a objetos. Esta característica es una de las tantas que sólo ha sido desarrollada en *YAQC4J* , como parte de la traducción del enfoque de testing en el paradigma funcional al paradigma orientado a objetos.

A continuación se muestran ejemplos de dos tests donde se usan anotaciones para indicar el uso de esos clasificadores.

```
@Configuration(tests = 300)
public class SampleCollector {

@Test
@Configuration(maxsize=9, minsize=-9)
@Generator(klass = Integer.class, generator = IntegerGen.class)
public void sample1(
            @Classify(name="sampleIntegers",
               classifiers = {EvenClassifier.class,
                              OddClassifier.class })
               final Integer a,
            @Collect(name="partitionInt",
               collector=IntegerPartitionCollector.class)
               final Integer b) {
assertEquals(a, a);
}
```

```
@Test
@Generator(generator=SimpleCollectionGen.class,
               klass=List.class)
@Configuration(maxsize=200)
public void sample2(
@Classify(name="sampleLists",
          classifiers = {EmptyListClassifier.class,
                         NonEmptyListClassifier.class,
                         SingletonListClassifier.class })
  final List<Integer> list) {
         assertTrue(list.size() >= 0);
}
}
```

En el primer test se generan números aleatorios entre -9 y 9, usando el generador de enteros provisto por la herramienta: IntegerGen. El test en sí no verifica nada interesante porque el objetivo es mostrar las anotaciones a nivel del primer argumento. La anotación @Classify recibe un nombre y un arreglo de clases que son los clasificadores a usar. Los clasificadores son subclases de la clase abstracta Classifier. Ésta clase define un método sencillo que representa el predicado que el objeto generado aleatoriamente debe cumplir para entrar dentro de la clasificación:

```
public abstract boolean classify(T object);
```

En el segundo argumento del test, se desea observar la distribución de la *partición identidad*, es decir, cuántas veces ocurre cada elemento dentro del conjunto total de objetos generados aletoriamente. Para ello se crea una subclase de IdentityPartitionCollector, una clase abstracta que particiona el conjunto de objetos por el simple criterio de identidad. La superclase de esta clase es Collector, que implementa la lógica de recolección de los datos y que tiene el método abstracto:

```
public abstract K getCategoryFor(T t);
```

El segundo método de test simplemente usa clasificadores para listas. El resultado de ejecutar este test es el siguiente:

```
Results for collector "partitionInt":
  0: 37 occurrences (12.333333%)
  1: 11 occurrences (3.6666667%)
```

```
  2: 8 occurrences (2.6666667%)
  3: 19 occurrences (6.3333335%)
  4: 12 occurrences (4.0%)
  5: 14 occurrences (4.6666665%)
  6: 13 occurrences (4.3333335%)
  7: 16 occurrences (5.3333335%)
  8: 15 occurrences (5.0%)
  9: 18 occurrences (6.0%)
  -9: 18 occurrences (6.0%)
  -8: 15 occurrences (5.0%)
  -7: 16 occurrences (5.3333335%)
  -6: 13 occurrences (4.3333335%)
  -5: 14 occurrences (4.6666665%)
  -4: 11 occurrences (3.6666667%)
  -3: 19 occurrences (6.3333335%)
  -2: 13 occurrences (4.3333335%)
  -1: 18 occurrences (6.0%)

Results for collector "sampleIntegers":
  even: 142 occurrences (47.333332%)
  odd: 158 occurrences (52.666668%)

OK. Passed 300 tests.
Results for collector "sampleLists":
  singleton: 0 occurrences (0.0%)
  empty: 2 occurrences (0.6666667%)
  non-empty: 298 occurrences (99.333336%)

OK. Passed 300 tests.
```

Estas funciones nos van a permitir verificar si la distribución de las clases o particiones es la deseada en base a los criterios que se definan, y cambiar la forma en que se generan esos valores para obtener mejores distribuciones de valores y objetos.

En este punto es interesante mencionar una técnica llamada*Adaptive Random Testing*[CKMT10, CLOM08, CLM05], que intenta mejorar la búsqueda de objetos de prueba que podrían fallar. La técnica se basa en la idea que los objetos que generan fallas se tienden a agrupar o clusterizar en regiones contiguas denominadas *regiones de falla* [Amm88, SM07]. Por ejemplo, en una función que toma dos enteros `x` e `y`, si hay una falla en un condicional tal como `if(x>=0 && y>=x)`, entonces la región contigua de falla se localiza

cuando `x` e `y` son positivos e `y` es mayor a `x`. Por eso, para generar nuevos valores u objetos tiene en cuenta los generados anteriormente y se trata de probar con valores *más diversos*, es decir, de otras regiones posibles de falla. Por supuesto, se pierde un poco de aleatoriedad en la generación de nuevos objetos, pero se intenta maximizar la cobertura de regiones o particiones dentro del universo de valores posibles.

La definición de *contiguo* en esta técnica no es absolutamente clara, y en general se basa en cálculos matemáticos sobre el valor u objeto. La primera debilidad de esta técnica se encuentra cuando se intenta definir el término *contiguo* para estructuras complejas como árboles, heaps o grafos. Lamentablemente, la mayor cantidad de trabajo escrito se realizó sobre el dominio de los números, donde se puede usar la distancia euclideana como medida de distancia. Para estructuras complejas, es difícil tomar una medida adecuada.

Aunque la técnica parece intuitivamente mejor que la generación aleatoria, se ha demostrado que sólo es eficaz para pequeñas cantidades de valores/objetos. Es además ineficiente en tiempo y espacio para encontrar la primera falla y su *F-measure*[1] no es buena [AB11]. La mayor crítica a esta técnica es que los resultados que muestran su superioridad por sobre el testing aleatorio (RT) son sobre casos específicos y no reales.

Los objetos generados por *YAQC4J* son en general objetos "frescos", es decir, objetos que no han sido modificados luego de creados. Sin embargo, cada vez que corre un test, algunos de esos objetos son modificados y pueden ser casos interesantes de entrada para otros tests. Con la idea de tener objetos en ese tipo de estados y enriquecer la entrada, se podría implementar un *pool* de objetos [CPO+11] que contenga objetos asociados a tipos. Los tests podrían tener un parámetro opcional que indique la probabilidad de crear nuevos objetos o tomarlos del *pool*. También debería considerar el reuso de objetos polimórficos y un tratamiento especial para objetos singleton. Sin embargo, esa implementación queda fuera del alcance de este trabajo, ya que es una variación del enfoque original propuesto por *QuickCheck*, que implica además analizar si el impacto de ese cambio es realmente significativo.

[1] F-measure es una métrica que se refiere a la cantidad de tests que se necesitan correr para encontrar la primera falla

Capítulo 6

Ejemplos

En este capítulo se presentan algunos ejemplos de código donde se usa *YAQC4J* . El capítulo tiene dos secciones. La primera contiene ejemplos de código usando la propia herramienta, y la segunda sección contiene ejemplos de uso en proyectos de código abierto.

6.1 Ejemplos en la herramienta

El primer ejemplo está basado en el test EqualsHashProperties<T> descripto en el Capítulo 3. Para crear un test que prueba las propiedades de los métodos equals y hashcode para la clase java.lang.Integer, se define el caso de prueba como se muestra en la Figura 6.1.

```
@Configuration(maxsize = 25, tests = 1,
               maxArgumentsFails = 10000)
@Generator(klass = int.class, generator = IntegerGen.class)
public class EqualsHashPropertiesInteger extends
EqualsHashProperties<Integer> {
}
```

Figura 6.1: Propiedades de equals y hash para java.lang.Integer

En el extracto de código mostrado en la Figura 6.2 el test no recibe parámetros. Sin embargo en ese test, se usa la clase Arbitrary, que no fue mencionada anteriormente, pero que juega un rol importante en la herramienta, y a la vez provee métodos para obtener algún generador para cualquier clase. En el ejemplo dado, se le solicita un generador para la clase java.util.Date, para ser usado como generador de los elementos de las listas creadas por el

generador ListGen<Date>. El generador de listas da una instanca de lista con un tamaño aleatorio entre 0 y 10. Se agregó la anotación @Configuration para indicar que el test debe correrse 20 veces, ya que por defecto el test se corre una sola vez si no recibe parámetros.

```
@Test
@Configuration(tests = 20)
public void testDateList() {
  ListGen<Date> listGen =
       new ListGen<Date>((Gen<Date>)
          Arbitrary.getGeneratorFor(
            Arbitrary.defaultDistribution(), Date.class));
  List<Date> list = listGen.arbitrary(
    Arbitrary.defaultDistribution(), 0, 10);
  assertTrue(list.size() < 11);
}
```

Figura 6.2: Ejemplo de test usando distintos generadores predefinidos

Finalmente, en el código de la Figura 6.3, hay un test que no tiene generadores definidos de manera explícita para la clase Person. Sin embargo, la subclase Employee tiene definido y asociado un generador en el contexto del método de test. En este caso, la herramienta automáticamente selecciona y usa ese generador para crear instancias aleatorias de las clase Person.

```
@RunWith(QCCheckRunner.class)
@Configuration(maxsize = 50)
public class HierarchySample {
    @Test
    @Generator(generator = EmployeeGen.class,
               klass = Employee.class)
    public void personIsNotNull(final Person p) {
        assertNotNull(p);
    }
}
```

Figura 6.3: Uso de generadores de la jerarquía de una clase

Finalmente, en la Figura 6.4 se muestra como ejemplo de definición de generador al generador de java.util.Calendar, donde se usa el generador de java.util.Date.

```
public class CalendarGen
           extends TransformerGen<Date, Calendar> {

    public CalendarGen() {
       super(new DateGen());
    }

    @Override
    protected Calendar transform(final Date date) {
        Calendar calendar = Calendar.getInstance();
       calendar.setTimeInMillis(date.getTime());
       return calendar;
    }

}
```

Figura 6.4: Generador para la clase java.util.Calendar

6.2 Ejemplos en proyectos open source

Para esta sección del capítulo se seleccionaron proyectos de código abierto que permitan validar que la herramienta es realmente útil y usable en proyectos reales. Para cada caso, se mejoraron los escenarios de pruebas usando las características de *YAQC4J* . Luego se agregaron generadores particulares y se agregaron pruebas de unidad para un proyecto que no poseía ninguna.

También se seleccionó una implementación de una estructura de datos y luego de reescribir los tests con *YAQC4J* , se verificó que la implementación no cumplía con la especificación. Finalmente se reescribieron los tests de una librería de código abierto para manejo de fechas y tiempo que es conocida por su precisión. No se detectaron más errores con los nuevos tests, pero se realizó una pequeña comparación de las respectivas implementaciones de tests de unidad.

6.3 JStock

JStock[jst] (v1.0.5y) es una librería de código abierto disponible en sourceforge que permite llevar un control de las inversiones en acciones. En este caso encontramos un test escrito con JUnit test en la clase TSTSearchEngineTest llamado testSearchAll. La clase permite realizar búsquedas eficientes de ítems almacenados, a partir de un prefijo. Un ejemplo de los muchos que

se escribieron para testear el comportamiento se muestra en la Figura 6.5:

```
public void testSearchAll() {
    TSTSearchEngine<Name> engine = new TSTSearchEngine<Name>();
    engine.put(new Name("Mr Cheok"));   // <--
    engine.put(new Name("miss Lim"));
    engine.put(new Name("mRM"));        // <--
    engine.put(new Name("mr H"));       // <--
    engine.put(new Name("ABCDEFG"));
    assertEquals(3, engine.searchAll("MR").size());
}
```

Figura 6.5: Prueba de unidad en JStock

Como puede observarse, los valores usados en el motor de búsqueda están *hard-coded*. Esto implica que tan sólo algunas combinaciones son probadas. Se escribió un nuevo test usando *YAQC4J* , cuya intención es similar al test dado, pero que provee al método estado y valores aleatorios. El test resultante tiene 58 líneas de código, incluyendo las líneas de código para dos generadores que fue necesario codificar. Al ejecutar el nuevo tests, se comprobó que la implementación de la clase contenía bugs, incluso para prefijos chicos y pocos valores almacenados, retornando siempre una cantidad muy inferior de resultados a los esperados o nada. La Figura 6.6 muestra la parte más significativa del test y el código de los generadores.

La intención del desarrollador de la librería al escribir el test original no era trivial. Se trataba de poner a la instancia del motor de búsqueda en un estado tal que retornada la cantidad de ítems esperados. Para ello se debían proveer ítems que incluyeran el prefijo (para ser encontrados por el motor) e ítems que no lo contuvieran. Ese trabajo fue realizado en nuestro test por un generador especial (TSTSearchEngineStringGen) que agrega el prefijo en una posición aleatoria de cada string. Además el motor de búsqueda es *case-insensitive* (no da importancia a las mayúsculas y minúsculas), con lo cual otro generador (AlterUpperAndLowerCaseStringGen) se encargaba de alterar el prefijo de forma que estuviera presente en mayúsculas y minúsculas cambiadas aleatoriamente.

También se escribieron tests para la clase org.yccheok.jstock.engine.Duration, sin considerar los tests escritos para la misma. Se pudieron encontrar dos bugs que no habían sido encontrados con los tests existentes, uno en el método getTodayDurationByYears y otro en el método getTodayDurationByDays cuando el parámetro (int era superior a 500).

```
@Test
@Configuration(maxsize=300)
public void qcTestSearchAll(String... strings) {
    String substring = new StringGen().arbitrary(
                       Arbitrary.defaultDistribution(), 1, 20);
    int numOccurrences = new PositiveIntegerGen().arbitrary(
          Arbitrary.defaultDistribution(), 0, strings.length);
    Gen<String[]> gen = new TSTSearchEngineStringGen(substring, numOccurrences, strings);
    String[] inputStrings = gen.arbitrary(Arbitrary.defaultDistribution(),
                            Integer.MIN_VALUE, Integer.MAX_VALUE);
    TSTSearchEngine<Name> engine = new TSTSearchEngine<Name>();
    for (String input : inputStrings) {
         if (!StringUtils.isEmpty(input)){
              engine.put(new Name(input));
         }
    }
    assertEquals(numOccurrences, engine.searchAll(substring).size());
}

//generators
public class TSTSearchEngineStringGen implements Gen<String[]> {
   <...>
   public String[] arbitrary(Distribution random,
                             long minsize, long maxsize) {
      String[] res = new String[strings.length];
      AlterUpperAndLowerCaseStringGen alterGen =
         new AlterUpperAndLowerCaseStringGen(substring);
      for (int i = 0; i < numOccurrences; i++){
         String original = strings[i];
         int pos = (int) Arbitrary.choose(random, 0,
                   original.length());
         res[i] = original.substring(0,pos) +
            alterGen.arbitrary(random,minsize,maxsize)
           + original.substring(pos,original.length());
      }
      return res;
   }
}

public class AlterUpperAndLowerCaseStringGen implements Gen<String> {
   <...>
   public String arbitrary(Distribution distribution,
                           long minsize, long maxsize) {
      StringBuilder result = new StringBuilder();
      for (int i = 0; i < str.length(); i++) {
         BooleanGen bgen = new BooleanGen();
         boolean lower = bgen.arbitrary(distribution, 0L, 0L);
         result.append(lower ? toLowerCase(str.charAt(i)) :
                               toUpperCase(str.charAt(i)));
      }
      return result.toString();
   }
}
```

Figura 6.6: Método de test para TSTSearchEngine.searchAll con *YAQC4J*

También se tomaron métricas del código y se compararon con la clase existente para verificar si había algún beneficio adicional. La Figura 6.7 muestra

que se obtuvo mejor cobertura con menor cantidad de código.

	DurationTest	YAQC4JDurationTest
ELOC	110	106
Branch coverage	62.5 %	75 %
Inst. coverage	71.3 %	95.1 %
Errores encontrados	0	2

Figura 6.7: Métricas para test de JUnit y *YAQC4J* para clase Duration

6.4 TuxGuitar

TuxGuitar[tux] es una aplicación de código abierto para escribir composiciones musicales (tablaturas de guitarra).

En los archivos fuente publicados por el equipo de TuxGuitar no hay pruebas de unidad. Sin embargo hay propiedades interesantes que se pueden probar en el modelo de dominio. Por ejemplo, existe una propiedad de las notas musicales que establece que: "*Para cualquier nota, si tiene un punto inmediatamente después, la duración se incrementa un cincuenta por ciento*". La Figura 6.8 muestra esta relación visualmente.

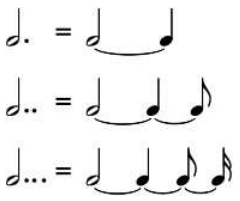

Figura 6.8: Ejemplo de notas con puntillo

Dado que se quería probar las duraciones, se debió crear un generador para la clase TGDuration. El generador escrito, llamado TGDurationGen, se muestra en la Figura 6.9. El generador usa ElementGen para seleccionar una duración aleatoriamente entre las existentes (siete).

Una vez definido este generador, se procedió a escribir las pruebas usando las anotaciones @Configuration y `@Generator`. La versión final se muestra en la Figura 6.10.

Se puede observar que no se pusieron valores ni objetos *hard-coded* en el test. Éste es un caso en el que queda muy evidente que las pruebas de unidad usando YAQJ4J no son *configuraciones* o *fixtures* sino *especificaciones* de propiedades.

```
public class TGDurationGen implements Gen<TGDuration> {
    @Override
    public TGDuration arbitrary(Distribution random,
           long minsize, long maxsize) {
        TGDuration duration = new TGFactory().newDuration();
        ElementGen<Integer> durations =
              new ElementGen<Integer>(
                  TGDuration.WHOLE, TGDuration.HALF,
                  TGDuration.QUARTER, TGDuration.EIGHTH,
                  TGDuration.SIXTEENTH,
                  TGDuration.THIRTY_SECOND,
                  TGDuration.SIXTY_FOURTH) {};
        duration.setValue(durations.arbitrary(random,
                                    minsize, maxsize));
        return duration;
    }
}
```

Figura 6.9: Generador de instancias de TGDuration

```
@RunWith(ar.edu.unlp.yaqc4j.runners.QCCheckRunner.class)
public class TGDurationTest {
    @Test
    @Generator(generator=TGDurationGen.class,
               klass=TGDuration.class)
    @Configuration(minsize= 1, maxsize=7)
    public void testDottedDuration(TGDuration duration) {
        long oldTime = duration.getTime();
        duration.setDotted(true);
        assertEquals(oldTime + (oldTime / 2),
                     duration.getTime());
    }
}
```

Figura 6.10: Propiedad para notas con puntillo

6.5 Red-black trees

Los árboles red-black [Bay72, Hin99] son una estructura de datos usada para búsquedas eficientes de tiempo logarítmico ($O(logn)$). La estructura es

un árbol binario de búsqueda donde sus nodos se colorean de rojo o negro para poder balancearse automáticamente cuando se inserta o borra un nodo.

Se tomó la implementación de código abierto de Red-Black trees de [rbt] para realizar pruebas de unidad. Lo que se hizo en este caso fue reescribir los tests hechos por el autor para probar la clase PersistentRedBlackTreeSet, usando *YAQC4J* . El resultado es que de los cinco tests originales, cuando son traducidos sólo dos son exitosos luego de varias ejecuciones. El resto fueron corridos varias veces y fallan en la primera ejecución. Es decir, la implementación es errónea y los tests escritos por el autor no verifican que el código cumple con la especificación de esta estructura. Por ejemplo, en el primer test, se insertaban números del 1 al 1000 de forma desordenada, y se probaba que luego se sacaban de forma ordenada. En los tests usando *YAQC4J* , la cantidad de elementos a insertar y los elementos en sí se generan aleatoriamente. Los tests nuevos revelaron que luego de cierto número de extracciones, el valor siguiente era menor que el anterior, o que ciertos números directamente no se insertaban en el árbol. En este ejemplo al menos, se verifica que el enfoque del presente trabajo de maestría es más efectivo en la búsqueda de errores que el enfoque tradicional de pruebas de unidad. En la figura 6.11 se muestra un ejemplo de test original y reescrito en *YAQC4J* . El resto de los tests se adjuntan como material al presente trabajo.

6.6 Joda-Time

Joda-time [jod] es una librería de Java muy conocida que implementa clases y funcionalidades para manejar fechas y horas. Ésta librería pretende atacar los defectos y falencias del manejo que se hace la versión estándar de Java con las fechas y horas.

La librería tiene muchos tests de unidad y son bastante amplios. Se tomó como ejemplo representativo de los test la clase de test TestDateTime_Constructors, que verifica el correcto funcionamiento de los constructores de la clase DateTime. Se reescribieron todos los tests en *YAQC4J* en la clase TestDateTime, y el resultado es que *YAQC4J* no pudo encontrar nuevos defectos en la implementación de DateTime. Sin embargo es interesante observar algunas métricas capturadas en el Cuadro 6.1, tomadas con el plugin de eclipse Metrics [met].

De todas estas métricas, se destaca la cantidad de código escrita, para la cual *YAQC4J* es ligeramente inferior. Sin embargo, se escribieron dos generadores (ChronologyGenerator y DateTimeZoneGenerator) que generan instancias aleatorias para las clases Chronology y DateTimeZone respectivamente, y que se usan como parámetros de algunos constructores de DateTime. Entre

Métrica	TestDateTime_Constructors	TestDateTime
Overriden methods	2	0
Attributes	7	0
Static attributes	2	0
Method LOC	356	317
Total LOC	528	483
Number of methods	64	40

Cuadro 6.1: Comparación entre TestDateTime_Constructors yTestDateTime

ambos generadores suman 5 líneas de código más.

La cobertura de código por branches (medida con el plugin de Eclipse Cobertura [eco]) en el test original es del 91.1 %, mientras que para el test con *YAQC4J* es de 91.6 %. Estos resultados similares puede que se deban a que los tests escritos con *YAQC4J* estaban basados en los test originales y recorrían los caminos internos de la clase. Sin embargo hay que destacar que la entrada para las pruebas con *YAQC4J* son más diversos.

6.7 Apache Commons Math

Otro caso similar que demuestra una de las ventajas que tiene el enfoque, fue el realizado con la librería de Apache Commons Math (v.3.2) [apa]. En este caso se tomó la clase Complex y se redefinieron los tests con *YAQC4J* . El resultado es que la cobertura de branches pasó del 91.2 % al 94.1 %, y el número total de líneas del test más los generadores (se escribieron 2, uno para generar doubles y otro para generar Complex) pasó de 1038 a 992. La clave del aumento en la cobertura estuvo en que los generadores generaban valores extremos (negativo infinito, positivo infinito, cero, NaN) con una probabilidad del 1 % y construían instancias de complejos y reales que no fueron capturados por los tests originales. Por último, se destaca que los oráculos son más precisos y específicos. En los tests originales, los oráculos (aserciones) constaban de valores fijos, mientras que en la nueva versión son resultado de un cálculo, que corresponde con la definición matemática. Una comparación sencilla se puede realizar viendo la Figura 6.12.

```
// test original
public void testPersistantInsert() {
PersistentRedBlackTreeSet<Integer> tree =
new PersistentRedBlackTreeSet<Integer>(INTEGER_COMP);

ArrayList<Integer> shuffledIntegers = getShuffledIntegers(1000);

// add shuffled integers to the tree
for (Integer integer : shuffledIntegers) {
            tree = tree.insert(integer);
}

assertTrue(tree.size() == 1000);

int count = 0;
// check they are iterated in the correct order
for (Integer integer : tree) {
            assertEquals(integer.intValue(), count++);
}
}

// test reescrito
@Test
@Generator(generator = PositiveIntegerGen.class, klass = int.class)
@Configuration(minsize = 0, maxsize = 1000)
@Ignore
public void testPersistantInsert(int size) {
        PersistentRedBlackTreeSet<Integer> tree =
            new PersistentRedBlackTreeSet<Integer>(INTEGER_COMP);
        IntegerGen intGen = new IntegerGen();
        // add shuffled integers to the tree
        for (int i = 0; i < size; i++) {
            int num = intGen.arbitrary(new NormalDistribution(), 0,
                        Integer.MAX_VALUE);
            tree = tree.insert(num);
        }
        assertEquals(tree.size(), size);
        // check they are iterated in the correct order
        int value = Integer.MIN_VALUE;
        for (Integer integer : tree) {
            assertTrue("Extracting " + integer.intValue()
                + " and previous element was " + value,
                  integer.intValue() >= value);
            value = integer;
        }
    }
```

Figura 6.11: Prueba de unidad original y prueba con el enfoque de *YAQC4J*

```
//Original (JUnit)
@Test
public void testConstructor() {
    Complex z = new Complex(3.0, 4.0);
    Assert.assertEquals(3.0, z.getReal(), 1.0e-5);
    Assert.assertEquals(4.0, z.getImaginary(), 1.0e-5);
}

@Test
public void testConstructorNaN() {
    Complex z = new Complex(3.0, Double.NaN);
    Assert.assertTrue(z.isNaN());

    z = new Complex(nan, 4.0);
    Assert.assertTrue(z.isNaN());

    z = new Complex(3.0, 4.0);
    Assert.assertFalse(z.isNaN());
}

//YAQC4JComplexTest
@Test
public void testConstructor(Double r, Double i) {
    Complex z = new Complex(r, i);
    Assert.assertEquals(r, z.getReal(), 1.0e-5);
    Assert.assertEquals(i, z.getImaginary(), 1.0e-5);
    Assert.assertEquals(Double.isNaN(z.getImaginary()) ||
                        Double.isNaN(z.getReal()), z.isNaN());
    Assert.assertEquals(!Double.isNaN(z.getReal()) &&
                        !Double.isNaN(z.getImaginary()), !z.isNaN());
}
```

Figura 6.12: Fragmento de test de *Commons Math* con *YAQC4J* y *J*Unit

Capítulo 7

Trabajo relacionado

7.1 Herramientas basadas en QuickCheck

Como se mencionó anteriormente, *YAQC4J* no es la única herramienta que existe para la generación automática de objetos de prueba intentando seguir la filosofía de *QuickCheck* . Algunas otras herramientas ya existían previamente. Como la herramienta desarrollada en este trabajo intenta cubrir muchas características que las demás no han cubierto, el primer paso es presentarlas para luego realizar una comparación.

7.1.1 JCheck

JCheck[JCh] es un proyecto de código abierto (bajo *Common Public License 1.0*) en sourceforge [sou], que provee generadores básicos para las clases java del paquete java.lang y para la mayoría de los tipos primitivos. También tiene un runner especial para reusar la infraestructura de JUnit, el operador imply, anotaciones para configurar el entorno de los test (número de tets o el tamaño de un parámetro), y la posibilidad de definir generadores nuevos. Sin embargo, esta herrammienta usa sólo un tipo de distribución, la uniforme, al usar la clase de Java java.util.Random en los generadores. Por otra parte, *YAQC4J* tiene un mecanismo más flexible y potente para controlar la aleatoriedad de los objetos generados (ver las características en el Capítulo 3). La herramienta cuenta con un pequeño tutorial y ha estado en fase de desarrollo inicial por mucho tiempo, por lo cual se supone que no se va a desarrollar y mejorar más.

7.1.2 QuickCheck for Java

Esta herramienta[Qui] publicada en *java.net* y *https://bitbucket.org* bajo licencia *Apache License 2.0*, tiene muchas características comunes con la desarrollada en el presente trabajo:

- generadores de Array, Enum, Clone y Null
- generadores originales de *QuickCheck* (`frequency`, `oneOf`, etc)
- generadores de transformación
- diferentes distributiones
- tamaño mínimo y máximo para los objetos aleatorios
- *runner* de JUnit

Sin embargo, en *YAQC4J* los objetos y valores generados aleatoriamente llegan como parámetro del método de test, mientras que en *QuickCheck for java* los datos son provistos por métodos helpers, invocados por el usuario en la implementación del test. Esto puede verse como un efecto indeseado, ya que uno de los objetivos de la generación aleatoria de objetos de prueba es no tener que preocuparse por escribir código relacionado a la sección *arrange* en el test. La definición de los tests en *YAQC4J* es más concisa y legible en *YAQC4J* . La herramienta cuenta con bastante documentación para su uso.

7.1.3 QC4J

QC4J([qc4]) es otra herramienta publicada en sourceforge con licencia *Apache License 2.0* que provee pocos generadores (para boolean, byte, double, float, int, long y short), pero provee generadores base para definir nuevos. Provee un mecanismo potente para mapear clases con sus correspondientes generadores. También provee tests parametrizados. Quizás la mayor desventaja de esta herramienta es que no provee integración con JUnit. Tiene su propio *runner*, basado en un método main(). Al igual que JCheck, QC4J solamente usa la distribución uniforme (java.util.Random). Finalmente, ésta herramienta no posee un mecanismo para parametrizar el número de veces que un test se corre, y el tamaño de los objetos generados.

7.1.4 JUnit-QuickCheck

JUnit-QuickCheck [juna] es una librería para testing escrita para JUnit, y cuyo principal logro es proveer valores aleatorios a las teorías (*theories*) de JUnit [Saf07]. Las teorías son un elemento relativamente nuevo en JUnit que permite parametrizar los tests con *data points*, que son básicamente fuentes de datos de cierto tipo. La idea de las teorías es proveer un universo acotado dentro del cual se prueba el comportamiento implementado (de ahí también proviene su nombre).

La herramienta permite, con configuraciones adicionales, que las teorías sean alimentadas con datos generados aleatoriamente. Provee generación de elementos para tipos básicos y colecciones, y también la posibilidad de crear y usar nuevos generadores. Permite elegir la *fuente de aleatoriedad* (subclase de java.util.Random) y también tiene un operador para limitar los objetos válidos a usar. Reusa el runner de teorías de JUnit para proveer el ciclo de ejecución para las pruebas. La herramienta es reciente al momento de escribir este trabajo y está en fase inicial, por lo que se espera que evolucione con el tiempo e incluya otras características.

7.2 Otras herramientas de testing aleatorio

Además de las herramientas mencionadas anteriormente, existen otras que usan la filosofía de testing aleatorio pero que no siguen el enfoque específico de *QuickCheck*. En el presente trabajo se mencionan las más importantes junto con sus características, pero se descarta una comparación con *YAQC4J* principalmente por la enorme diversidad de herramientas que hay, y por tener un enfoque diferente de testing aleatorio. Las herramientas detalladas en esta sección, se destacan por tener un enfoque sistemático de búsqueda de valores de entrada, la generación automática de casos de prueba y oráculos (aserciones), o algunos por tener un proceso totalmente automático sin intervención del desarrollador. Sin embargo, y aunque parezca obvio, no hay evidencia de que esos enfoques realmente ayuden al desarrollador en la práctica, ya que las métricas que se usan normalmente no involucran al usuario y algunas herramientas demandan un esfuerzo significativo del usuario para entender y corregir los tests generados [FSM+13].

7.2.1 JCrasher

JCrasher [CS04, jcr] es una herramienta de testing escrita para Java, que examina el código fuente de las clases para generar casos de prueba. La herramienta crea caminos de ejecución que van creando instancias de

diferentes clases para verificar el comportamiento de métodos públicos. La herramienta analiza el software estáticamente e intenta buscar fallas sin intervención o conocimiento previo del desarrollador, con lo cual la hace una opción económicamente atractiva.

La herramienta genera tests para cada combinación posible de parámetros, y para ello genera un grafo de parámetros y lo recorre (por una cuestión práctica, descarta los métodos que no retornan valores – null). Los métodos con mayor cantidad de argumentos son testeados más frecuentemente, asumiendo que son los que más tiempo de ejecución consumen.

La búsqueda de test cases se centra en excepciones en tiempo de ejecución no contempladas por el desarrollador, como puede ser: variables no inicializadas (NullPointerException), conversión de tipo ilegal (ClassCastException), división por cero o acceso ilegal a posiciones de arreglos. Tiene además heurísticas para detectar *falsos errores*. Para asegurar que cada test es independiente del otro, usa diferentes *class loaders*, con lo cual cada vez que una clase es requerida para el test, su estado está limpio y libre de interferencia. El resultado de la ejecuión de la herramienta es un reporte de errores encontrados y los test de JUnit para esos casos, que podrían ser integrados permanentemente a los tests de unidad del software en construcción. Lamentablemente no se registran casos de uso de la herramienta en entornos reales de desarrollo, por lo que sólo ha tenido un uso acotado o académico.

7.2.2 Eclat

Eclat [PE05, ecla] es una herramienta de testing que utiliza el código fuente y ejemplos de uso de la clase, para producir pruebas de unidad con nuevos valores de entrada seleccionados. Los valores de entrada de la clase de test usan la clase de una forma distinta a la dada, con el objetivo de encontrar errores. La herramienta implementa de esta manera una técnica automática de testing aleatorio.

El primer paso de esta herramienta es generar un modelo operacional del software en base al ejemplo de uso dado como entrada (tests y casos de uso correctos). Luego se producen valores posibles de entrada usando valores iniciales y modificaciones sobre esos valore usando diferentes técnicas. Se van agregando los valores generados en un pool y se clasifican de acuerdo a lo que producen al darse como input a la clase. Finalmente se reducen los casos de acuerdo a la falla producida y su clasificación, que sirven para general la salida de la herramienta. El resultado es un conjunto de tests que evidencian posibles fallas. El desarrollador debe decidir si son casos de falla o falsos positivos.

En el paper que presenta la herramienta [PE05] se realiza una compara-

ción con JCrasher (ver 7.2.1) ya que el enfoque y objetivo de ambas herramientas es similar. En cuanto a la efectividad en la búsqueda de errores en el software, Eclat presenta mejores resultados.

7.2.3 Randoop

Randoop [PE07, ran] es una mejora sobre Eclat, que usa feedback-directed random test generation, una técnica explicada en otro trabajo de su pincipal autor [PLEB07]. La herramienta existe para Java, y existe además una versión para .Net [PLB08].

Randoop toma como input un conjunto de clases a testear, un límite de tiempo, y opcionalmente un conjunto de verificadores de *constraints* que se agregar a los que la herramienta ya tiene. La herramienta produce casos de prueba donde se produjeron violaciones a *constraints* de la clase, y tests que no fallan y que pueden ser usados para testing de regresión. Para la generación de casos de prueba, genera secuencias incrementales de invocaciones a métodos, creando los argumentos necesarios para poder hacer esas llamadas y reusándolos en otros tests. De acuerdo al resultado de la ejecución, las secuencias se clasifican y minimizan (se remueven las secuencias redundantes) para general la salida.

Debido a un efecto *plateau* en la cantidad de errores que detecta la herramienta con su estrategia de testing, y a evidencias de que la distribución puede contribuye a ese efecto, los autores piensan que usar distribución adaptativa puede mejorar la tasa de errores encontrados. La idea sería modificar la distribución aleatoria usada a medida que transcurre el tiempo.

Randoop pudo detectar errores en la versión 1.6 del Java Development Kit de IBM al ejecutar tests de regresión sobre colecciones. Sin embargo también se reportaron errores descubiertos con otras herramientas, que Randoom no pudo detectar.

7.2.4 Jartege

Java Random Test Generator (Jartege) [Ori05] permite la generación aleatoria de casos de prueba en Java para clases especificadas en JML (Java Modeling Language). La herramienta se inspira en otra herramienta: JML-JUnit [CL02], que produce casos de prueba a partir de la especificación hecha con JML. En Jartege, los tests generados consisten en secuencias de llamados a constructores y métodos que generan errores. La herramienta permite definir generadores de objetos que contienen valores aleatorios y también permite reusar objetos creados anteriormente para evitar el *overhead* asociado. Se puede controlar la aleatoriedad de las clases usadas definiendo un *perfil*

operacional que permite indicar qué métodos deben ser más intensivamente testeados. Los métodos tienen un peso asociado, y a mayor valor asociado, mayor uso del mismo en los tests.

El código producido por la herramienta es una clase con un método main() con la ejecución de todos los métodos de tests que produjeron errores, y que puede ser ejecutada posteriormente.

La mayor limitación de la herramienta es probablemente el uso obligatorio de JML para producir los casos de prueba. Muchas otras herramientas mencionadas en este capítulo no tienen esa fuerte imposición.

7.2.5 RUTE-J

Randomized **U**nit **T**esting **E**ngine for **J**ava (RUTE-J) [AHLL06, AHL+06] es una técnica distinta de testing aleatorio. El testing se realiza subclasificando la clase TestfragmentedCollection, que tiene métodos denominados *test fragments*, que son usados por la herramienta para que corra test de *longitud* X. La herramienta selecciona de manera aleatoria X test fragments uno a uno y genera un caso de prueba. Si se produce un error, la herramienta lo reporta y se puede crear un test de JUnit o de la misma herramienta a partir de la secuencia de test fragments ejecutados. La herramienta viene con una UI que facilita la tarea de escribir los fragmentos de código, correr los test y generar tests de unidad a partir de ellos.

7.2.6 AutoTest

AutoTest [MCLL07, CLOM08] es una herramienta escrita en Eiffel, que explota una de las características destacadas del leguaje que son los contratos (*contracts*). Al igual que Jartege en Java (ver 7.2.4), las precondiciones, postcondiciones e invariantes sobre las unidades a testear son usadas para generar casos de prueba que verifican el comportamiento esperado (oráculos). Esto hace que el proceso de testing aleatorio sea mucho más automático que con otras herramientas.

AutoTest puede ser ejecutada por consola indicando ciertos parámetros de ejecución: tiempo de ejecución, parametros de generacion de nuevos valores o reuso de existentes, distribución aleatoria, y las clases a ser testeadas. AutoTest usa una combinación de estrategias para incrementar la posibilidad de encontrar violaciones a los contratos. Entre las estrategias se incluye el *adaptive random testing*[CLM04] (mencionado en el Capítulo 5). En este caso, los autores definen que la distancia entre objetos (un criterio para seleccionar valores candidatos nuevos de entrada) se basa en:

- la distancia entre los tipos, basado en la longitud del camino de un tipo a otro en el grafo de herencia.
- la distancia entre los tipos primitivos que conforman su estado (variables)
- la distancia calculada recursivamente para los objetos referenciados en sus variables (estado).

La herramienta genera objetos y los retiene durante la ejecución de los test mediante un pool de objetos para reusarlos posteriormente (al igual que lo hace Eclat – ver 7.2.2). Para la creación de nuevas instancias, reusa objetos que son pasados como argumento a los constructores, o bien decide crear nuevos en base a los parámetros del programa y estado del pool. Un punto débil es que no soporta polimorfismo al no analizar las jerarquías de herencia y métodos polimórficos.

Los autores de la herramienta no tienen evidencia suficiente para demostrar la superioridad del enfoque frente a otros. Sin embargo sostienen que la herramienta puede mejorar su actividad si se ejecuta con diferentes parámetros. El esfuerzo de ellos está centrado en hacer de la técnica algo práctico y usable, que complemente otras técnicas de testing [CPO+11]. Existen varias actualizaciones del software desde su publicación inicial, muchas de ellas enfocadas en mejorar la generación de objetos de forma eficiente y efectiva.

7.2.7 Pex

Pex [TDH08] produce *test suites* con un alto porcentaje de cobertura en programas escritos en .Net. La herramienta analiza los componentes a ser testeados usando ejecución simbólica y dinámica [GKS05, BOP00]. Ésta técnica consiste en ejecutar el componente bajo prueba usando variables simbólicas como entrada, que permiten tracear los caminos posibles de ejecución. Pex analiza el comportamiento del programa monitoreando esas trazas de ejecución. Pex construye una fórmula que representa las condiciones bajo las cuales un camino de ejecución se alcanza, y usa un *constraint solver* para generar nuevos valores de entrada que cubran nuevos caminos no explorados. Como consecuencia, se aumenta el grado de cobertura de los tests generados.

Pex puede generar los test analizando los bytecodes de los componentes. Para ello instrumenta (es decir, modifica las instrucciones) los bytecodes y analiza el comportamiento del software mediante un intérprete. Pex trata de usar las variables y *setters* públicos de los objetos para setear su estado, pero usa un enfoque similar a *YAQC4J* cuando debe generar nuevos objetos, usando los constructores públicos de la clase.

7.2.8 EvoSuite

EvoSuite [FA11] es otra herramienta de generación de casos de prueba para Java que incluye aserciones y un alto grado de cobertura. La herramienta está disponible libremente como plugin de Elipse o para ser ejecutado en consola.

El trabajo de Fraser et. al. se centra en poder producir tests con un oráculo (es decir, una forma de verificar que la salida es la esperada) adecuado. La herramienta deja al desarrollador decidir si los errores encontrados son casos de falla o de falsos positivos. Las aserciones que genera la herrmienta reflejan el comportamiento observado y no el que se espera.

EvoSuite usa un enfoque de búsqueda de casos de falla que integra *hibrid search* [HM10], ejecución simbólica y dinámica (ver 9subsec:pex) y *testability transformation*[HHH$^+$04], entre otras técnicas. La idea central de la herramienta es mutar casos de prueba agregando, borrando o cambiando sentencias y parámetros individuales a los casos existentes, obteniendo además un alto grado de cobertura de los tests.

7.2.9 TestFul

TestFul [BLM10, tes] es un plugin de Eclipse que sigue un enfoque de generación de casos de pruebas con búsqueda sistemática. La herramienta trabaja a nivel de clase y métodos, primero para poner al objeto en un estado útil y luego para ejercitar los caminos de ejecución de los métodos.

TestFul crea un conjunto de variables para todos los tipos referenciados por la clase, considerando incluso instancias de distintas clases para un mismo tipo, para permitir mayor polimorfismo. Luego se generan algunos tests que evolucionan usando técnicas de búsqueda que intentan maximizar la cobertura de sentencias y ramas de ejecución. Cuando un test genera una cobertura mayor, el algoritmo evolutivo (basado en JMetal [jme]) que trabaja a nivel de la clase usa el test como punto de partida para alcanzar niveles de cobertura mayor.

Lamentablemente, la herramienta es según sus propios autores, un prototipo académico, y no tiene actualizaciones desde hace unos ya varios años.

7.2.10 eToc

eToc [Ton04] usa algoritmos genéticos para producir de manera automática casos de prueba para clases. Los casos de prueba son descriptos como cromosomas a los que se les puede aplicar mutaciones con el objetivo de aumentas la cobertura. Cada cromosoma tiene una secuencia de sentencias

(creación de objetos, invocación a métodos y en particular el método que se quiere testear al final de la secuencia) y un conjunto de parámetros de entrada.

El algoritmo genera nuevos casos de prueba hasta alcanzar un límite máximo de cobertura o de tiempo. Cada vez que se genera un nuevo caso de prueba, se analiza su resultado para agregarlo o no al conjunto resultado. Al final se quitan los casos redundantes para minimizar el tamaño del conjunto final de casos de prueba. Para seleccionar los cromosomas candidatos se calcula una función de aptitud (basada en la cantidad de controles que se ejercitan durante la ejecución del test) y se aplican nuevas mutaciones. Entre las mutaciones que se aplican están: el cambio de valores de entrada, cambio de constructores e inserción o remoción de una invocación a un método. Si bien no se dan detalles profundos sobre la forma en que se generan nuevas instancias de clases, la herramienta permite indicar qué clase concreta se debe usar cuando un método recibe un parámetro cuyo tipo es una clase abstracta o interface. El código resultante (tests de JUnit) debe ser editado por el desarrollador para agregar las aserciones necesarias (definición del *oráculo*) y para hacer el còdigo más legible opcionalmente.

7.2.11 YETI

York Extensible Testing Infrastructure (YETI) [yet] es una herramienta de código abierto cuyo código principal fue escrito en Java pero que soporta otros lenguajes como .Net [OT10]. YETI provee un fuerte desacomplamiento entre las estrategias de testing y código *core*, gracias a un meta-modelo independiente del lenguaje de implementación. Esto permite que el motor de la herramienta sea independiente del lenguaje del código a testear. Para testear código de otro lenguaje es necesario extender algunas clases de Java e implementar los *bindings* que hagan falta. En el caso de .Net, la implementación de esos bindings se hace a través de sockets [OT10].

YETI se ejecuta desde la línea de comandos con argumentos que indican, por ejemplo, la duración de la *sesión de testing*, la estrategia de aleatoriedad e información de tracing entre otros. La herramienta tiene además una interfaz visual que permite modificar algunos parámetros mientras los tests se ejecutan, que es algo único respecto de todas las demás herramientas. El desarrollador puede modificar el porcentaje de valores nulos (null), número de variables nuevas (ya que reusa objetos de un pool existente) y el máximo número de variables por tipo mientras los tests se están corriendo. También se puede ver la evolución de los errores encontrados en el tiempo, la cantidad de nuevas instancias creadas y el número total de llamadas. Las estrategias de aleatoriedad por defecto son cuatro:

- Pure random: que permite modificar el número de valores nulos y valores nuevos creados
- Random+: que genera valores *interesantes* para los tipos (aunque no se define el significado del término para objetos).
- Random decreasing: que es la estrategia Random+ donde los parámetros van decreciendo de 100 a 0 porciento.
- Random periodic: que es la estrategia Random+ donde los parámetros van creciendo y decreciendo periódicamente.

Capítulo 8

Comparación con otras herramientas

En esta capítulo se comparan las diferentes herramientas existentes en la acutalidad, presentadas en la primera parte del Capítulo 7, con la herramienta desarrollada en este trabajo.

8.1 Metodología

La comparación se divide en las siguientes áreas:

- características
- soporte al usuario final
- esfuerzo del usuario final usando la herramienta
- efectividad

En el área de características, se comparan aquellas propiedades o características de cada herramientas y se verifica si están o no presentes. Se toma *QuickCheck* como referencia, pero también se toman en consideración los aspectos referentes al paradigma orientado a objetos. Se indica si cumple o no con el ciclo de ejecución de *QuickCheck* ; esto es, si cada test se ejecuta múltiples veces con diferentes objetos generados aleatoriamente. Se considera si la herramienta provee los combinadores que permiten crear nuevos generadores: oneOf, choose, etc. También si se proveen generadores para los tipos primitivos (valores), objetos de los paquetes estandard de Java, singletons, clases abstractas, interfaces, jerarquías y generador basado en los constructores de una clase. Se analiza si la herramienta permite clasificar los valores

generados, permite configurar la distribución aleatoria, y los valores mínimos y máximos pasados a los constructores. Finalmente, si la herramienta provee generadores adicionales, como ser generador para enumerativos y clones, y el número de dependencias con otras librerías.

El soporte al usuario final se refiere a aquellos elementos que permiten al usuario final entender la herramienta para usarla. Incluye el entorno de desarrollo, la documentación que hay disponible, ejemplos y licenciamiento. Se incluye a JUnit como referencia.

El esfuerzo del usuario final da una idea del esfuerzo que el desarrollador necesita para implementar casos de prueba usando la herramienta (facilidad de uso). Se comparan entre sí las diferentes herramientas y con el framework base JUnit. Para evaluar el esfuerzo de usuario final se incluye métricas como: comprensibilidad (inteligibilidad, facilidad de uso, reporte y debugging), esfuerzo (tiempo de codificación) y tiempo de ejecución, y finalmente tamaño (líneas de código efectivas - ELOC) y métricas de diseño (número de clases, métodos, variables y palabras) de los componentes desarrollados como pruebas de unidad.

La efectividad refiere a la capacidad de la herramienta para detectar errores en el software, y los recursos usados para tal fin. También entra en este área la cobertura de código realizada durante los tests. Dentro de éste área también se toma en cuenta el número de bugs encontrados.

La primera condición para ser capaz de comparar las herramientas es disponer de la misma funcionalidad a probar para todas ellas. En este trabajo se siguieron las siguientes reglas:

- respetar el diseño unificado y los detalles de implementación de la herramienta original, descripta en el Capítulo 2;
- preparar los ejemplos para ser compilados y corridos desde el mismo entorno de desarrollo: Eclipse.
- proveer a los desarrolladores toda la documentación de cada una de las herramientas y darles una introducción del enfoque de testing propuesto.
- seleccionar casos de estudio y casos de prueba para ser codificados y que sirvan para comparar las herramientas de manera objetiva.

8.2 Métricas

En esta sección se definen cada una de las métricas y métodos de medición a utilizar en la comparación:

8.2.1 Comprensibilidad

La comprensibilidad es una métrica subjetiva, por lo cual para evaluarla se solicitó la colaboración de diferentes desarrolladores familiarizados con pruebas de unidad pero con distinto nivel y bagaje cultural. A cada persona se le dio un proyecto Eclipse para Java 1.6 con las clases a testear, las librerías a usar y toda la documentación disponible, y se le solicitó escribir las pruebas de unidad en las diferentes herramientas. Luego de ello, se solicitó que completaran una encuesta sencilla que captura la valoración personal respecto de: inteligibilidad, facilidad de uso, reporte, debugging y valoración general, con una escala de 1 (muy mala) a 5 (muy buena). La encuesta se puede ver en el anexo A del presente trabajo.

En total hubo cinco desarrolladores Java profesionales de distintos países y con distinta experiencia en programación orientada a objetos y en particular con JUnit: Hong Kong (China) / 12 años de experiencia, Colombia / 7 años de experiencia, Perú / 2 años de experiencia y Argentina / 7 y 4 años de experiencia. Ninguno de los desarrolladores se conoce entre sí y no se permitió interacción alguna tampoco para evitar cualquier tipo de desviación en cada uno de los resultados.

8.2.2 Esfuerzo

- Tiempo de codificación. Permite tener una medida de cuan fácil de aprender y usar es cada herramienta. Como parte de las tareas solicitadas a cada desarrollador, se le pidió medir el tiempo utilizado en codificar cada uno de los tests en cada una de las herramientas. El tiempo contempla toda ejecución preliminar que haya hecho el usuario durante la codificación. También se pidió separar el tiempo que se utilizó en desarrollar los generadores que se necesitaron.

8.2.3 Tiempo

- Tiempo de ejecución. Permite tener una medida del tiempo usado para ejecutar los tests en cada herramienta. Para medir el tiempo de ejecución, se tomaron cada una de las pruebas escritas por los desarrolladores y se corrieron en una misma computadora. Las características de la computadora donde se corrieron los tests son:
  - Dell XPS L501X
  - Procesador: Intel Core i5 M460 @ 2.53GHz.
  - Memoria RAM: 4Gb.

- Sistema operativo: Window 7 Home Premium, 64bits.
- JDK v1.6.0_23
- RAM reservada 64Mb

Dentro de las condiciones que se quisieron establecer, cada método de test debía ser corrido un número fijo de veces, y cada herramienta debía respetar ese número. Sin embargo, JUnit corre sólo una vez cada método de test, *Junit-QuickCheck* y *QuickCheck for Java* generan por defecto 100 y 200 instancias/valores respectivamente para cada parametro de cada método. Además para *JCheck* y *YAQC4J* un método podría ejecutarse más de 100 veces si el operador imply indica que no se cumple la condición, forzando a la herramienta a generar nuevos valores. Por lo tanto, fijar el número de ejecuciones no fue posible. Se tomó la decisión de dejar el número de ejecuciones al valor por defecto de cada herramienta y analizar esos resultados junto a los resultados relacionados a la efectividad.

Para medir el tiempo de ejecución se usó la herramienta Perf4J [per], que permite calcular y mostrar estadísticas de performance de código Java. Se tomó cada uno de los ejemplos escritos por los desarrolladores y se creó una clase que ejecute los tests desde un simple método main. El método fue anotado con @Profiled para que la herramienta pudiera tomar el tiempo de ejecución total. Además se anotó cada uno de los métodos de test para obtener el tiempo de ejecución y el número de ejecuciones. Esta estategia fue una forma ligeramente intrusiva pero rápida de poder realizar la medición de performance para cada una de las implementaciones.

Perf4J contiene soporte para *rodear* métodos con el código necesario para medir su performance mediante *aspectos* [KH01a, KH01b] basados en la herramienta AspectJ [asp]. El mecanismo usado específicamente es el *run-time weaving*, que significa que el código que rodea cada método es injectado cuando se carga la clase en tiempo de ejecución. Como el código necesario para ejecutar las clases de test es absolutamente minimal, el overhead de tiempo que podría tener es despreciable. Para lograr el *weaving*, cada test se ejecuta indicando a la virtual machine de Java que use el agente de *aspect weaving* (-javaagent:./lib/aspectjweaver-1.6.1.jar).

Se midió la performance de cada uno de los métodos de tests, y en general el tiempo de ejecución fue siempre menor de 5 milisegundos. Como en este análisis se debe tener en cuenta también el tiempo que cada herramienta consume en generar los objetos aleatorios, el análisis de los tiempos de ejecución de los métodos de tests fue descartado.

8.2.4 Tamaño

- Líneas efectivas de código (ELOCs). Proporciona un valor de la cantidad de código que contiene cada test. La definición de línea de código fuente es ambigua. Su significado varía de un lenguaje de programación a otro, pero también dentro de un mismo lenguaje de programación. Para normalizar las líneas escritas, se usó el formateador estándar de Eclipse y luego se usó el plugin Metrics v1.3.6 [met] para obtener el valor de TLOC (total lines of code – líneas de código totales), que no considera líneas en blanco ni comentarios.

- Número de clases, métodos y atributos. Proporcionan otra dimensión del tamaño. Estas métricas pueden ser obtenidas con Metrics, por lo cual se procedió de la misma forma que con ELOCs.

8.2.5 Efectividad

- Cobertura de código[MM63]. Es una medida dada en porcentaje que describe el grado en que el código fuente fue testeado. Una cobertura del código del 100 % indica que todos los caminos de ejecución del método fueron visitados por los tests ejecutados. Para medir la cobertura de los tests, se usó el plugin de Eclipse E-Cobertura [eco], que reporta la cobertura de líneas de código y de *branches* de los métodos.

- Número de bugs encontrados. Se tomará un número fijo de ejecuciones de los tests para poder tomar esta métrica y poder realizar la comparación.

8.3 Casos de estudio seleccionados

Para realizar la comparación, se deben seleccionar casos de estudio. En esta sección se presentan los casos seleccionados y los fundamentos de la elección. Los casos de estudio forman parte del material adjunto al presente trabajo.

No existe literatura que recomiende estrategias o modelos de software para realizar testing. Algunos de los criterios elegidos fueron utilizados en otras evaluaciones y comparaciones de herramientas de testing aleatorio [FSM+13].

El primer criterio de búsqueda tenido en cuenta fue que el código fuera de acceso libre para poder usarlo y leer su código. Otro criterio de elección fue que sean ejemplos sencillos y/o con dominio fácil de entender, pero que no fueran triviales. En particular que tuvieran condicionales. Elegir casos

de lógica o dominio complejos sólo agregaba mayor tiempo preliminar para poder escribir los casos de prueba. También se buscaron casos de estudio que tuvieran casos de tests ya escritos, ya que una de las tareas es reescribir los tests con las distintas herramientas de testing. Se tomó en consideración que los tests se pudieran escribir usando la características más importante de las herramientas. También se buscaron herramientas con documentación (javadoc, comentarios en los métodos o documentación externa) de forma que fuera rápido pensar los tests a escribir. A continuación se describen los ejemplos seleccionados:

- IMoney [imo]

  Es un caso sencillo de clase que se usa para explicar la herramienta de testing SUnit y JUnit. Este caso de estudio se toma para validar que incluso en ejemplos sencillos, las herramientas pueden ser efectivas en el descubrimiento de bugs. Se pidió a los desarrolladores que reescriban cada uno de los tests pertenecientes a la clase de test MoneyTest, usando cada una de las herramientas.

- JDatePicker [jda]

  Es una librería que provee un componente *date picker* para Swing. Este caso de estudio se tomó por poder acceder al código fuente y por ser de un tamaño chico comparado con el resto de los proyectos. También es sencillo de entender. Se utilizó este ejemplo (versión 1.3.2) para escribir tests sobre los componentes base, ya que no tenía tests codificados previamente sobre ellos. Se le solicitó a los desarrolladores que escribieran tests para las clases que implementan el modelo del componente visual: UtilDateModel, UtilCalendarModel y SqlDateModel.

- javaGeom [jav]

  Es una librería para aplicaciones de geometría, que provee una API para crear y manipular figuras geométricas y realizar algunas operaciones entre ellas. JavaGeom (version 0.11.1) está en fase alfa y tiene algunos tests escritos. Se le pidió a los desarrolladores pruebas adicionales para la clase math.geom2d.Box2D, que ya tiene una clase de tests (Box2DTest). Con *YAQC4J* y demás herramientas se pueden crear más test pero se necesitan crear generadores específicos para las clases math.geom2d.Point2D y math.geom2d.Box2D.

8.4 Resultados y análisis

En esta sección se presentan los resultados de la comparación para cada una de las áreas mencionadas en la Sección 8.1 y las métricas detalladas en la Sección 8.2. Tanto los entornos originales como los modificados para tomar las distintas métricas se encuentran adjuntos en el CD que acompaña este trabajo.

Debido al espacio que ocupa cada tabla, se usarán los siguientes números en todas las tablas a continuación, para hacer referencia a las herramientas: 1 = *YAQC4J* , 2 = JCheck, 3 = QuickCheck for Java, 4 = QC4J, 5 = JUnit-QuickCheck, 6 = QuickCheck, 7 = JUnit.

8.4.1 Características

	1	2	3	4	6	5
Ciclo de ejecución	S	-	S	S	S	S
Combinadores para generadores	S	S	S	N	S	N
Generadores de valores primitivos	S	S	S	S	S	S
Clasificación de valores/primitivos	S	N	N	N	S	N
Operador *imply*	S	S	S	N	S	S
Generador de objetos estandar	S	S	S	N	N/D	N
Generadores para clases abstractas	S	N	N	N	N/D	N
Generadores para jerarquías	S	N	N	N	N/D	N
Generadores para interfaces	S	N	N	N	N/D	N
Generador basado en constructores	S	N	N	N	N/D	N
Generador(es) de singletons	S	N	N	N	N/D	N
Generadores adicionales	S	S	S	N	S	N
Configuración de tests	S	S	S	N	S	S
Distribución aleatoria configurable	S	N	S	N	N	S
Objetos random son parametro de test	S	S	N	S	N	S

En cuanto a las características de las herramientas, del cuadro anterior se puede observar que *YAQC4J* es la única herramienta que implementa todas las características detalladas, y que coinciden en parte con la herramienta original, y en parte con las características particulares del paradigma orientado a objetos como son los generadores de interfaces u objetos estandar. El

resto de las herramientas proveen implementaciones parciales, destacándose sólo la columna de QC for Java. QC4J es la única herramienta que no provee la posibilidad de configurar los tests, que es una parte esencial del enfoque de pruebas de unidad propuesto. Es importante resaltar que sólo la herramienta desarrollada tiene implementación de generadores para clases abstractas, jerarquías, interfaces, singletons y basado en constructores.

Otro aspecto a destacar es la clasificación de los objetos aleatoriamente generados, para la cual únicamente *YAQC4J* tiene una implementación completa de la característica.

8.4.2 Soporte al usuario final

	1	2	3	4	5	7
Entorno de desarrollo	*	*	*	own runner	*	*
Documentación	online/ javadocs	online	online/ javadocs	No	online	online
Ejemplo de uso	online	pocas online	online	online	online	online

* = cualquier entorno con interfaz gráfica y soporte para JUnit.

En la tabla anterior se puede observar que QC4J tiene un runner propio, que probablemente haya sido una de las causas de las notas negativas en el esfuerzo del usuario final, ya que presenta una forma distinta y no estandard de ejecutar las pruebas de unidad.

Las herramientas tienen en su mayoría alguna forma de documentación, ya sea *on-line* o a partir de sus javadocs y código fuente. Solamente QC4J no tiene ningún tipo de documentación de uso (a excepción de los ejemplos de uso). Esta debilidad sumada al runner propio, jugaron en contra en la evaluación de los usuarios en cuanto a la facilidad de uso (ver Subsección 8.4.3).

8.4.3 Esfuerzo del usuario final

A continuación se muestra un resumen de los resultados de las encuestas del esfuerzo del usuario final, que son los promedios de los resultados obtenidos. Los resultados completos se detallan en el Anexo B.

	1	2	3	4	5	7
Inteligibilidad (1 to 5)	4.2	3.2	2.8	2.8	3.6	4
Facilidad de uso (1 to 5)	4.2	3.4	2.2	2.8	2.8	4.2
Reporting (1 to 5)	4	2.8	2.8	3.8	3	3.6
Debugging (1 to 5)	2.8	3	2.2	3	3.6	4.2
Valoración general (1 to 10)	8.2	6.4	5	6.6	6.6	7.4

Desde el punto de vista del esfuerzo del usuario final, JUnit-Quickcheck y *YAQC4J* son las herramientas más fáciles de entender, pero la herramienta desarrollada en este trabajo es la más fácil de usar. Sólo JUnit (agregado como referencia) obtiene resultados similares. QuickCheck for Java resultó difícil de usar y la causa fueron los problemas ocurridos durante la ejecución de los tests. Al parecer la herramienta no fue bien correctamente testeada y tiene muchos errores de implementación. Este problema también impactó negativamente en otras dimensiones de la comparación. QuickCheck for Java también resultó difícil de entender debido a la forma en que los datos aleatorios deben ser generados (con loops).

En cuanto al reporte de resultados, se destacan *YAQC4J* y QC4J, que presentan además los resultados de manera similar a como lo hace la herramienta original, QuickCheck.

Un punto débil de *YAQC4J* parece ser la facilidad de debugging. El problema indicado por los desarrolladores fue que una vez que ocurre un error, la pila de ejecución refiere al código de la herramienta antes que al código testeado. Esta debilidad deberá ser subsanada en futuros *releases* de la herramienta.

Finalmente, en la valoración general de cada herramienta se destacan *YAQC4J* , QuickCheck for Java y JUnit-QuickCheck. Sin embargo, la herramienta desarrollada en este trabajo es claramente la que mejor calificaciones obtuvo en promedio.

A continuación se muestra un resumen de los resultados de los tiempos promedios requeridos para codificar las pruebas, y de ejecutarlas. Los resultados completos se detallan en el Anexo B.

IMoney	1	2	3	4	5	7
Codificación total (m)	107	247	260	99	194	N/D
Codificación generadores (m) (m)	7	16	5	15	0	N/D
Ejecución (ms)	571	897	963	321	6027	293

jDatePicker	1	2	3	4	5	7
Codificación total (m)	31	59	38	45	41	N/D
Codificación generadores (m)	7	19	8	22	6	1
Ejecución (ms)	480	993	544	332	1832	262

javaGeom	1	2	3	4	5	7
Codificación total (m)	32	46	46	36	47	N/D
Codificación generadores (m)	11	12	18	14	13	25
Ejecución (ms)	377	457	496	225	2262	243

De los datos obtenidos sobre el esfuerzo del usuario final, se puede observar que el tiempo usado para codificar generadores difiere de herramienta en herramienta pero *YAQC4J* consume la menor cantidad de tiempo en la gran mayoría de los casos. En algunos casos fue necesario tiempo extra para desarrollar nuevos generadores, y coincide con las herramientas que no tienen generadores para tipos estándares (JCheck) o la forma en que hay de definirlos no es intuitiva (QC4J). El tiempo para codificar tests es significativaente mayor en JCheck, y siempre por arriba del promedio para JUnit-QuickCheck. *YAQC4J* obtuvo también buenos tiempos en la mayoría de los casos.

En cuanto al tiempo de ejecución, se observaron buenos resultados en todas las herramientas, excepto por Junit-QuickCheck. Desafortunadamente, en esta herramienta y QuickCheck for Java el número de veces que se ejecuta cada test depende de la cantidad de datos que se deben generar aleatoriamente y no se puede setear un número determinado. Se usaron valores por defecto para las otras herramientas (100 veces) y se aproximó el número de ejecuciones para las dos anteriormente mencionadas. Se puede observar que JUnt-QuickCheck es la herramienta menos eficiente y que and *YAQC4J* y QC4J son por el contrario, las más eficientes.

A continuación se muestra un resumen de los resultados del análisis del tamaño de las pruebas escritas. Los resultados son promedios de las muestras tomadas y los valores obtenidos se detallan en el Anexo B.

IMoney	1	2	3	4	5	7
ELOC	269	279	371	231	277	157
# clases	2	2	2	3	1	1
# métodos	23	23	22	25	22	23
# variables	0	5	1	2	0	6

jDatePicker	1	2	3	4	5	7
ELOC	166	189	186	198	191	140
# clases	3	4	3	6	3	2
# métodos	9	10	9	14	10	8,6
# variables	1	1	1	1	1	1

javaGeom	1	2	3	4	5	7
ELOC	102	104	116	92	105	81
# clases	2	3	3	3	3	1
# métodos	6	7	7	8	8	6
# variables	0	0	2	1	0	1

Un análisis de las métricas de tamaño seleccionadas no muestra superioridad de ninguna herramienta sobre otra. Ninguna de las herramientas pudo testear la misma funcionalidad que JUnit usando menos líneas de código (algo que observó al menos con *YAQC4J* en los ejemplos de Capítulo 6). QuickCheck for Java siempre requirió más líneas de código para cubrir la misma funcionalidad, debido a la forma en que se deben escribir los test para generar los valores aleatorios (usando loops anidados). La variación en el número de clases, métodos y variables no permite llegar a ninguna conclusión.

8.4.4 Efectividad

A continuación se muestra un resumen de los resultados del estudio de la efectividad de cada herramienta para cada caso de estudio. Como antes, los resultados son promedios de las muestras tomadas. Los resultados completos se detallan en el Anexo B.

IMoney	Class	1	2	3	4	5	7
Inst. coverage(%)	Money	95.44	98.54	94	98.74	91.98	94
Branch coverage(%)	Money	82.94	88.78	70.3	91.26	70.04	68.8
Inst. coverage(%)	MoneyBag	87.8	86.76	85.93	88.72	88,24	86.8
Branch coverage(%)	MoneyBag	86.18	83.34	80.35	89.44	81.6	85.7
Nro bugs encontrados		0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Inst. coverage(%)	69,34	68,74	68,74	69,34	69,34	59,68
Branch coverage(%)	51,42	51,42	51,42	51,42	51,42	36,52
Nro bugs encontrados	1	0	0	0	1	0

Al recibir el primer resultado de uno de los desarrolladores, me informó que *YAQC4J* y *Junit-quickcheck* habían detectado un error en el método addYear de la clase SqlDateModel. El número de ejecuciones de cada test en todas las herramientas era el mismo: 100. El test correspondiente fallaba en la mayoría de los casos, en la primera o segunda ejecución. *YAQC4J* además, indicaba

en qué casos (con qué argumentos) fallaba: numeros grandes para años (e.g.: 701564927). El resto de las herramientas no detectaron este error.

javaGeom	1	2	3	4	5	7
Inst. coverage(%)	13,88	13,88	13,98	15,22	15	12,52
Branch coverage(%)	4,22	4,22	4,62	8,4	4,62	2,52
Nro bugs encontrados	0	0	0	0	0	0

Para concluir el analisis de los resultados, los datos obtenidos sobre efectividad muestran que la cobertura de *YAQC4J* alcanza mejores resultados en instrucciones y branches que JUnit y QuickCheck for Java. Sin embargo, los resultados son a veces inferiores a otras herramientas, y en particular QC4J siempre alcanza mejor cobertura. Uno de los objetivos a futuro es mejorar la cobertura siguiendo una búsqueda estratégica de valores, al igual que lo hacen las herramientas mencionadas en la Sección 7.2.

Respecto de los bugs encontrados, es destacable que sólo dos herramientas (*YAQC4J* y JUnit-Quickcheck) fueran capaces de encontrar bugg en el código de JDatePicker. El resto de las herramientas no pudieron reportar el error incluso con un número grande de ejecuciones.

8.5 Riesgos de validez

El número de desarrolladores que colaboraron para realizar la comparación puede no ser suficiente para poder generalizar los resultados. Sin embargo, todos los desarrolladores son profesionales con experiencia y bagaje cultural diverso, que no puede ser encontrado en otros experimentos realizados con herramientas de testing aleatorio.

Finalmente, debido a la falta de directrices o casos estandares para seleccionar los casos de estudio y ejemplos, los casos seleccionados pueden no cubrir todos los aspectos de las herramientas de testing. Por lo tanto, los casos de estudio pueden no ser lo suficientemente representativos para realizar la comparación, y más criterios de selección y casos de estudio podrían haber sido agregados.

Capítulo 9

Conclusiones y trabajo futuro

QuickCheck ha sido una herramienta exitosa dentro de la programación funcional. Muchas herramientas similares a ella han sido escritas para diferentes lenguajes. En la programación orientada a objetos existen algunos ejemplos de traducciones o herramientas similares a *QuickCheck* . Específicamente en Java, hay algunas herramientas escritas siguiendo las ideas de *QuickCheck* . En el presente trabajo se ha presentado el enfoque de testing con generación aleatoria de objetos, desarrollado una herramienta en el lenguaje Java que sigue las ideas de testing aleatorio que se plasman en *QuickCheck* . Se demostró con varios ejemplos, que los beneficios respecto del enfoque tradicional con JUnit son satisfactorios, aún cuando existen tests ya escritos. Se han seleccionado además las herramientas similares existentes hoy día y y se describieron en el Capítulo 7. También se realizó una comparación de las mismas contra la herramienta desarrollada en el Capítulo 8. Ninguna de las otras herramientas analizadas han atacado la impedancia que existe entre la programación funcional y la programación orientada a objetos para pruebas aleatorias de manera completa. Tampoco han atacado los problemas existentes en la programación orientada a objetos que no existen en la programación funcional. En la comparación realizada entre *YAQC4J* y estas herramientas en los aspectos que sí se pueden comparar, la herramienta ha demostrado ser superior. Sin embargo, en algunos aspectos la herramienta debe ser mejorada, y los planes se detallan más tarde. La mayor contribución de este trabajo es haber podido generar una herramienta completa funcionalmente, que además implementa las ideas de esta filosofía de manera correcta y genera beneficios cuando es comparada con la herramienta del enfoque tradicional de pruebas de unidad.

A pesar de que *YAQC4J* es una herramienta completa funcionalmente, existe mucho trabajo futuro por realizar. Existen muchas posibilidades que explorar tales como:

- Anotaciones a nivel de parámetros

  La herramienta implementada sigue la filosofía de *QuickCheck*, que genera valores aleatorios para una función dada y según el tipo de cada parámetro. Sin embargo la herramienta no permite configurar distintas estrategias de generación aleatoria para cada uno de los parámetros. Una modificación posible de *YAQC4J* que la diferenciaría filosóficamente de la herramienta original, pero que la enriquecería, sería permitir la anotación Configuration a nivel de cada parámetro. Ésto permitiría que los tests reciban, por ejemplo, Strings de distinto tamaño máximo, o generados con otra distribución aleatoria.

- Generación de objetos basado en búsqueda sistemática

  Muchas veces la herramienta no es capaz de generar objetos que cumplan la (pre) condición del operador Implication.imply. Por ejemplo, para testear que el método equals es transitivo (ver el método transitiveEquals en la Figura 3.8), tres objetos `A`, `B` y `C` deben satisfacer que `A = B` y que `A = B`. Esta condición es muy difícil de cumplir y la mayoría de las pruebas fallan. La solución pasa exclusivamente por el desarrollador, que debe crear un generador particular para ese método de test particular, de forma tal que los objetos generador aleatoriamente cumplan la condición. Una solución que provea al desarrollador cierta independencia de esta problemática y que genere objetos adecuados de una forma rápida es una posible línea de trabajo e investigación futura.

  En el Capítulo 7 se mencionaron distintas herramientas que usan enfoques de generación de tests basados en búsquedas más sistemáticas. *YAQC4J* podría extenderse para incluir algunas de las ideas implementadas en esas herramientas, en particular el uso de un pool de objetos reusables, combinado con ejecución dinámica simbólica para mejorar las métricas de cobertura. También se pueden buscar formas de optimizar el F-measure [NCLC12] a partir del estudio de optimizaciones sobre esa métrica.

- Soporte para clases genéricas

  A pesar que se ha provisto soporte para colecciones genéricas y algunas clases parametrizadas, se encontró que no es posible saber en *runtime* el tipo de los parámetros para clases en el fondo de una jerarquía. Éste problema podría ser resuelto con reificación de los generics de Java. Sin embargo, falta aún especificar e implementar en las máquinas virtuales del lenguaje (JVM - *Java Virtual Machine*) esta característica. Deberá esperarse una especificación estándar y una implementación de

la JVM que incluya la reificación de clases parametrizadas por parte del Java Community Process [jcp].

- Ejemplos incluidos en la herramienta

  Aunque la documentación generada para poder entender y usar la herramienta es suficiente, muchos más ejemplos podrían ser incluidos como parte de la herramienta o en una librería aparte. De esa manera, los usuarios podrían conocer con mayor profundidad la herramienta y podrían sacar mejor ventaja de ella.

- Generadores para tipos estándares de Java

  El lenguaje Java consiste de muchas clases estándares útiles, distribuidas en diferentes paquetes. YAQC4J provee generadores para los tipos más comunes usados. Sin embargo, se podrían agregar generadores para otras clases como ser java.io.File, java.net.Socket ó java.lang.Exception. Cabe recordar que a pesar que no se proveen generadores para esas clases, sí existe la posibilidad de que el desarrollador los codifique y use, implementando la interfaz Gen<T>). También existe un mecanismo por el cual se crean instancias de una clase basado en los constructores, en la jerarquía o para el caso de las interfaces, la búsqueda de clases que la implementen y que se encuentren en el *classpath*.

- Integración con entorno con Inversión de control (IoC)

  Sería interesante implementar una extensión de la herramienta original, que considere runners capaces de manejar IoC a nivel de inyección de dependencias. Un ejemplo de ésto es Spring, que provee clases de test y runners específicos para poder correr un test con un determinado contexto de ejecución (beans) y con la posibilidad incluso de usar una base de datos sin alterarla (se hace rollback de los cambios al finalizar el test).

- Implementación en Smalltalk (Pharo)

  El lenguaje de programación Java es uno de los más ampliamente usados en la industria. Smalltalk [Kay93] por otro lado, tiene menor popularidad en ese ambiente pero es usado muchas veces en cursos de programación orientada a objetos por la ausencia de información de los tipos y su sintaxis fácil de entender entre otras características. Un posible trabajo futuro podría ser implementar una versión de *QuickCheck* en Smalltalk Pharo [BDN+09]. La mayor dificultad y desafío en la implementación de este enfoque es que la mayoría de las herramientas de testing automático aleatorio se basan en la información estática del

código fuente (información de tipos). Un detalle de los desafíos que se deben afrontar para una implementación de este tipo de herramientas fue detallada por Ducasse [DOB11] cuando analizó la implementación de posibles bindings para YETI (ver 7.2.11). Si bien la implementación para YETI sigue otro enfoque, los problemas son muy similaes. Entre ellos se destacan la generación de nuevas instancias, la identificación de errores y el análisis de resultados.

Anexo A

Formato de encuestas

Name:

Surname:

Level of knowledge on programming(1-10):

Level of knowledge on Java(1-10):

Level of knowledge on unit testing(1-10):

Task description You will have to write different test cases for three different tools which are briefly described bellow:

- IMoney, is a simple example used many years ago to explain SUnit and JUnit. For this example, you'll have to rewrite all test methods in the class `MoneyTest` using JQueck, QC4J, quickCheck, YAQC4J and junit-quickcheck. Create one class per tool.
- JDatePicker, is a library that provides a Swing date picker component. Unfortunately, this library does not provide unit tests, so you will have to code tests for the classes `UtilDateModel`, `UtilCalendarModel` and `SqlDateModel`. Write the same unit tests using JUnit, JQueck, QC4J, quickCheck, YAQC4J and junit-quickcheck.
- javaGeom, is a library for geometric applications. It provides an API for creating and manipulating geometrical figures. You will have to provide additional unit tests for the class `math.geom2d.Box2D`. Take a look at the existing test cases in `Box2DTest` and also the classes `math.geom2d.Point2D` and `math.geom2d.Box2D` (you will have to

define generator(s) for the class `math.geom2d.Point2D`). Your task is to define test cases for methods: `contains(Point2D)`, `union(Box2D)`, `intersection(Box2D)`, `merge(Box2D)` and `clip(Box2D box)`.

Survey

	YAQC4J	**JCheck**	**QC 4 Java**	**QC4J**	**JUnit**
Time to code test cases (minutes)					
Time to code generators (minutes)					
Inteligibility (1-5)					
Ease of use (1-5)					
Reporting (1-5)					
Debugging (1-5)					
Overall value (1-10)					

Comments:

Anexo B

Mandy Siu

IMoney	1	2	3	4	5	7
Time to code test cases	58	179	92	61	122	0
Time to code generators	5	5	5	5	0	0
Execution time	807	2137	0	723	8781	293
ELOC	318	345	0	269	328	157
# classes	2	2	0	2	1	1
# methods	23	23	0	25	22	23
# variables	1	0	0	3	0	6
Branch coverage Money	93.8	87.5	0	100	68.8	68.8
Inst. Coverage Money	100	98.7	0	100	94	94
Branch coverage MoneyBag	90.5	83.3	0	90	83.3	85.7
Inst. Coverage MoneyBag	89.1	86.8	0	89.1	86.8	86.8
# bugs found	0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Time to code test cases	16	40	10	24	20	26
Time to code generators	0	0	0	5	0	5
Execution time	523	2322	584	522	1918	495
ELOC	211	194	197	219	186	208
# classes	3	3	3	6	3	4
# methods	6	6	6	12	6	7
# variables	0	0	0	1	0	0
Inst. coverage	62.5	59.5	59.5	62.5	62.5	62.5
Branch coverage	85.71	85.7	85.71	85.71	85.71	85.71
# bugs found	2	0	0	0	0	0

javaGeom	1	2	3	4	5	7
Time to code test cases	11	16	15	15	15	21
Time to code generators	5	5	5	5	5	5
Execution time	583	524	350	350	1769	289
ELOC	111	116	132	101	123	117
# classes	3	3	3	3	3	3
# methods	7	7	7	9	9	7
# variables	0	0	6	2	0	0
Inst. coverage	12.5	12.5	12.5	15.2	14.2	12.5
Branch coverage	4	4	4	7.1	4	4
# bugs found	0	0	0	0	0	0

Survey	1	2	3	4	5	7
Inteligibility (1-5)	4	3	3	3	4	3
Ease of use (1-5)	4	3	1	3	3	4
Reporting (1-5)	4	3	3	4	3	3
Debugging (1-5)	3	3	2	4	4	4
Overall value (1-10)	8	6	4	7	7	7

`Andres Sayago`

IMoney	1	2	3	4	5	7
Time to code test cases	120	308	353	111	202	0
Time to code generators	5	8	5	10	0	0
Execution time	495	867	3258	184	8694	293
ELOC	227	238	459	241	324	157
# classes	1	3	2	3	1	1
# methods	23	24	24	25	22	23
# variables	0	0	1	3	0	6
Branch coverage Money	93.8	93.8	68.8	100	68.8	68.8
Inst. Coverage Money	100	100	94	100	94	94
Branch coverage MoneyBag	83.3	88.1	83.3	90.5	83.3	85.7
Inst. Coverage MoneyBag	86.8	89.1	86.8	89.1	86.8	86.8
# bugs found	0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Time to code test cases	26	60	20	34	35	30
Time to code generators	0	20	0	23	0	0
Execution time	506	751	459	286	1397	185
ELOC	170	184	171	206	181	165
# classes	3	5	3	6	3	1
# methods	6	8	6	12	7	6
# variables	0	0	0	1	0	0
Inst. coverage	59.5	59.5	59.5	59.5	59.5	59.5
Branch coverage	85.7	85.7	85.7	85.7	85.7	85.7
# bugs found	0	0	0	0	0	0

javaGeom	1	2	3	4	5	7
Time to code test cases	30	32	30	28	36	40
Time to code generators	8	9	9	11	10	0
Execution time	352	306	232	204	1842	262
ELOC	98	96	122	90	101	79
# classes	3	3	3	3	3	1
# methods	7	7	7	7	9	5
# variables	0	0	2	2	0	0
Inst. coverage	12.5	12.5	12.5	14.5	14.2	11.2
Branch coverage	4	4	4	7.1	4	0.5
# bugs found	0	0	0	0	0	0

Survey	1	2	3	4	5	7
Inteligibility (1-5)	4	3	3	3	4	3
Ease of use (1-5)	4	3	1	3	3	4
Reporting (1-5)	4	3	3	4	3	3
Debugging (1-5)	3	3	2	4	4	4
Overall value (1-10)	8	6	4	7	7	7

`David Vara`

IMoney	1	2	3	4	5	7
Time to code test cases	170	338	391	152	305	0
Time to code generators	15	18	0	27	0	0
Execution time	607	866	276	166	6059	293
ELOC	247	266	389	259	289	157
# classes	2	2	1	4	1	1
# methods	23	23	22	26	22	23
# variables	0	24	2	0	0	6
Branch coverage Money	93.8	100	68.6	100	68.8	68.8
Inst. Coverage Money	98.7	100	94	100	94	94
Branch coverage MoneyBag	88.1	81	81	92.9	83.3	85.7
Inst. Coverage MoneyBag	89.1	86.2	86.2	89.1	86.8	86.8
# bugs found	0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Time to code test cases	40	85	47	60	57	52
Time to code generators	0	25	0	38	0	0
Execution time	544	851	336	304	1480	185
ELOC	170	179	171	205	199	165
# classes	3	4	3	7	4	3
# methods	6	7	6	12	7	6
# variables	0	0	0	1	0	0
Inst. coverage	59.5	59.5	59.5	59.5	59.5	59.5
Branch coverage	85.7	85.7	85.7	85.7	85.7	85.7
# bugs found	0	0	0	0	0	0

javaGeom	1	2	3	4	5	7
Time to code test cases	40	66	55	47	74	60
Time to code generators	12	18	16	25	21	0
Execution time	303	312	229	205	2659	246
ELOC	95	95	129	92	117	79
# classes	2	1	2	4	2	1
# methods	6	6	6	9	7	5
# variables	0	0	1	1	0	0
Inst. coverage	15.2	15.2	15.2	17.2	15.2	11.2
Branch coverage	4	4	4	7.1	4	0.5
# bugs found	0	0	0	0	0	0

Survey	1	2	3	4	5	7
Inteligibility (1-5)	4	2	3	2	4	4
Ease of use (1-5)	4	3	3	3	3	4
Reporting (1-5)	5	2	3	4	3	4
Debugging (1-5)	2	2	2	3	4	4
Overall value (1-10)	7	5	5	5	7	7

Mariano Pilotto

IMoney	1	2	3	4	5	7
Time to code test cases	150	350	380	121	282	0
Time to code generators	10	38	13	20	0	0
Execution time	481	244	230	218	3506	293
ELOC	246	239	351	209	253	157
# classes	3	3	3	3	3	1
# methods	24	24	24	28	26	23
# variables	0	0	0	4	0	6
Branch coverage Money	75	93.8	75	87.5	75	68.8
Inst. Coverage Money	94	100	94	98.7	83.9	94
Branch coverage MoneyBag	83.3	78.6	71.4	83.3	72.4	85.7
Inst. Coverage MoneyBag	87.2	84.9	83.9	87.2	94	86.8
# bugs found	0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Time to code test cases	40	81	61	60	45	30
Time to code generators	35	38	40	39	30	0
Execution time	318	242	946	196	2228	173
ELOC	204	208	213	198	210	126
# classes	6	7	6	7	6	3
# methods	9	10	9	12	12	6
# variables	0	0	0	0	0	0
Inst. coverage	82.6	82.6	82.6	82.6	82.6	82.6
Branch coverage	0	0	0	0	0	0
# bugs found	0	0	0	0	0	0

javaGeom	1	2	3	4	5	7
Time to code test cases	40	69	58	43	70	60
Time to code generators	20	20	20	20	15	0
Execution time	273	255	1440	180	1488	190
ELOC	102	108	113	118	112	67
# classes	3	3	3	5	3	1
# methods	7	7	7	9	9	5
# variables	0	0	0	0	0	0
Inst. coverage	16.7	16.7	17.2	16.7	18.4	12.1
Branch coverage	5.1	5.1	7.1	5.1	5.1	2
# bugs found	0	0	0	0	0	0

Survey	1	2	3	4	5	7
Inteligibility (1-5)	4	4	3	3	3	4
Ease of use (1-5)	4	4	3	2	3	4
Reporting (1-5)	3	3	3	3	3	3
Debugging (1-5)	3	3	3	2	3	3
Overall value (1-10)	8	7	7	7	7	8

`Victor Toledo`

IMoney	1	2	3	4	5	7
Time to code test cases	40	62	87	52	63	0
Time to code generators	0	10	0	15	0	0
Execution time	465	369	200	315	3093	293
ELOC	211	310	287	175	192	157
# classes	1	2	1	2	1	1
# methods	20	21	19	21	20	23
# variables	0	0	0	0	0	6
Branch coverage Money	93.8	68.8	68.8	68.8	68.8	68.8
Inst. Coverage Money	100	94	94	94	94	94
Branch coverage MoneyBag	85.7	85.7	85.7	90.5	85.7	85.7
Inst. Coverage MoneyBag	86.8	86.8	86.8	89.1	86.8	86.8
# bugs found	0	0	0	0	0	0

jDatePicker	1	2	3	4	5	7
Time to code test cases	32	31	51	45	48	62
Time to code generators	0	10	0	5	0	0
Execution time	752	798	396	353	2137	214
ELOC	153	179	219	163	180	124
# classes	1	2	1	2	1	1
# methods	19	20	19	20	18	19
# variables	4	4	4	4	4	5
Inst. coverage	82.6	82.6	82.6	82.6	82.6	82.6
Branch coverage	0	0	0	0	0	0
# bugs found	1	0	0	0	1	0

javaGeom	1	2	3	4	5	7
Time to code test cases	40	49	70	45	53	60
Time to code generators	10	9	40	11	14	0
Execution time	374	348	228	185	3601	226
ELOC	105	104	85	60	71	65
# classes	1	3	4	2	3	1
# methods	3	7	9	7	8	8
# variables	0	0	0	0	0	4
Inst. coverage	12.5	12.5	12.5	12.5	14.2	15.6
Branch coverage	4	4	4	4	4	5.6
# bugs found	0	0	0	0	0	0

Survey	1	2	3	4	5	7
Inteligibility (1-5)	4	4	2	3	3	4
Ease of use (1-5)	5	4	2	3	2	4
Reporting (1-5)	3	4	3	4	3	4
Debugging (1-5)	4	4	2	2	3	5
Overall value (1-10)	9	8	5	7	5	6

Bibliografía

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: an illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York, NY, USA, 2011. ACM.

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[AHL+06] James H. Andrews, Susmita Haldar, Yong Lei, Felix Chun, Hang Li, and Na B. Randomized unit testing: Tool support and best practices. Technical report, 2006.

[AHLL06] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 36–45, New York, NY, USA, 2006. ACM.

[Amm88] Paul Eric Ammann. *Data diversity: an approach to software fault tolerance*. PhD thesis, Charlottesville, VA, USA, 1988. AAI8904228.

[apa] Commons Math: The Apache Commons Mathematics Library
`http://commons.apache.org/proper/commons-math/`
Last accessed 31/01/2014.

[asp] AspectJ v1.6.1
`http://eclipse.org/aspectj/`
Last accessed 31/01/2014.

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[BB01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BLM10] L. Baresi, P.-L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for java. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 185–194, 2010.

[BM83] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, September 1983.

[BOP00] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. *SIGSOFT Softw. Eng. Notes*, 25(5):39–48, August 2000.

[Bro87] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[CF08] Jan Christiansen and Sebastian Fischer. Easycheck - test data for free. In *FLOPS*, pages 322–336, 2008.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, January 2010.

[CL02] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 231–255, London, UK, UK, 2002. Springer-Verlag.

[CLM04] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, *ASIAN*, volume 3321 of

Lecture Notes in Computer Science, pages 320–329. Springer, 2004.

[CLM05] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive random testing. In MichaelJ. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 320–329. Springer Berlin Heidelberg, 2005.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[CPO+11] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Softw. Test. Verif. Reliab.*, 21(1):3–28, March 2011.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[Dij72] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, July 1984.

[DOB11] Stéphane Ducasse, Manuel Oriol, and Alexandre Bergel. Challenges to support automated random testing for dynamically typed languages. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 9:1–9:6, New York, NY, USA, 2011. ACM.

[Duc] Stephane Ducasse. Sunit explained revisited.

[ecla] Eclat
`http://groups.csail.mit.edu/pag/eclat/`
Last accessed 31/01/2014.

[eclb] Eclipse IDE
`http://www.eclipse.org/`
Last accessed 31/01/2014.

[eco] Eclipse Plugin for Cobertura
`http://ecobertura.johoop.de`
Last accessed 31/01/2014.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[FM00] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[FMPW04] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 236–246, New York, NY, USA, 2004. ACM.

[FSM+13] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, New York, NY, USA, 2013. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[Ham94] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[HHH+04] Mark Harman, Lin Hu, Robert Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel, and Marc Roper. Testability transformation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30:3–16, 2004.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*, pages 1–55. ACM Press, 2007.

[Hin99] Ralf Hinze. Constructing red-black trees, 1999.

[HM10] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, March 2010.

[imo] JUnit Test Infected: Programmers Love Writing Tests
`http://junit.sourceforge.net/doc/testinfected/testing.htm`
Last accessed 31/01/2014.

[int] IntelliJ
`http://www.jetbrains.com/idea/`
Last accessed 31/01/2014.

[jav] javaGeom, A geometry library for Java
`http://sourceforge.net/projects/geom-java/`
Last accessed 31/01/2014.

[JCh] EasyCheck - Test Data for Free
`http://www.jcheck.org/`
Last accessed 31/01/2014.

[jcp] Java Community Process
`http://www.jcp.org/en/home/index`
Last accessed 31/01/2014.

[jcr] JCrasher
`http://ranger.uta.edu/~csallner/jcrasher/`
Last accessed 31/01/2014.

[jda] JDatePicker, Java Swing Date Picker
`http://sourceforge.net/projects/jdatepicker/`
Last accessed 31/01/2014.

[jme] JMetal - Metaheuristic Algorithms in Java
`http://jmetal.sourceforge.net/`
Last accessed 31/01/2014.

[jod] Joda Time. Java date and time API
`http://joda-time.sourceforge.net/`
Last accessed 31/01/2014.

[jst] JStock
`http://jstock.sourceforge.net/`
Last accessed 31/01/2014.

[juna] JUnit-QuickCheck
`https://github.com/pholser/junit-quickcheck/`
Last accessed 31/01/2014.

[junb] JUnit Test Infected: Programmers Love Writing Tests
`http://junit.sourceforge.net/doc/testinfected/testing.htm`
Last accessed 31/01/2014.

[Kay93] Alan C. Kay. The early history of smalltalk. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM.

[Ken] Kent Beck. Simple Smalltalk Testing: With Patterns
`http://www.xprogramming.com/testfram.htm`
Last accessed 31/01/2014.

[KH01a] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 313–, New York, NY, USA, 2001. ACM.

[KH01b] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–, September 2001.

[MCLL07] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 114–129, Berlin, Heidelberg, 2007. Springer-Verlag.

[met] Metrics
`http://metrics.sourceforge.net/`
Last accessed 31/01/2014.

[MM63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963.

[NCLC12] Ye Nan, Kian Ming Adam Chai, Wee Sun Lee, and Hai Leong Chieu. Optimizing f-measure: A tale of two approaches. *CoRR*, abs/1206.4625, 2012.

[net] Netbeans
`https://netbeans.org/`
Last accessed 31/01/2014.

[Ori05] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Proceedings of the First international conference on Quality of Software Architectures and Software Quality, and Proceedings of the Second International conference on Software Quality*, QoSA'05, pages 242–256, Berlin, Heidelberg, 2005. Springer-Verlag.

[OT10] Manuel Oriol and Sotirios Tassis. Testing .net code with yeti. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, pages 264–265, Washington, DC, USA, 2010. IEEE Computer Society.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: automatic generation and classification of test inputs. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[per] Perf4J v0.9.16
`http://perf4j.codehaus.org/`
Last accessed 31/01/2014.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 87–96, New York, NY, USA, 2008. ACM.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[qc4] QC4J
`http://sourceforge.net/projects/qc4j/`
Last accessed 31/01/2014.

[qcp] QuickCheck++
`http://software.legiasoft.com/quickcheck/`
Last accessed 31/01/2014.

[Qui] Quickcheck for Java
`http://java.net/projects/quickcheck/pages/home/`
Last accessed 31/01/2014.

[ran] Randoop
`https://code.google.com/p/randoop/`
Last accessed 31/01/2014.

[rbt] A Java implementation of persistent red-black trees open sourced
`http://edinburghhacklab.com/2011/07/a-java-implementation-of-persistent-red-black-trees-open-sourced/`
Last accessed 31/01/2014.

[Saf07] David Saff. Theory-infected: or how i learned to stop worrying and love universal quantification. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 846–847, New York, NY, USA, 2007. ACM.

[sch] Scheme-Check: Randomized Unit Testing for PLT Scheme
`http://c2.com/cgi/wiki?schemecheck`
Last accessed 31/01/2014.

[SM07] Christoph Schneckenburger and Johannes Mayer. Towards the determination of typical failure patterns. In *Fourth international*

workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting, SOQUA '07, pages 90–93, New York, NY, USA, 2007. ACM.

[sou] Sourceforge
`http://www.sourceforge.org/`
Last accessed 31/01/2014.

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[tes] TestFul, an Evolutionary Testing Framework for Java
`https://code.google.com/p/testful/`
Last accessed 31/01/2014.

[TLMG10] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action, Second Edition*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2010.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.

[tux] TuxGuitar: Music composition software
`http://tuxguitar.com.ar/`
Last accessed 31/01/2014.

[yet] Yeti site
`http://code.google.com/p/yeti-test/`
Last accessed 31/01/2014.