Detección de objetos espurios en generación automática de entradas

Simón Gutiérrez Brida, Pablo Ponzio, Valeria Bengolea, y Nazareno Aguirre

Universidad Nacional de Río Cuarto,
Río Cuarto, Argentina
{sgutierrez,pponzio,vbengolea,naguirre}@dc.exa.unrc.edu.ar

Resumen En el contexto de testing, el uso de entradas generadas de manera automática, sobre las cuales se va a ejecutar el programa bajo prueba, es una práctica cada vez más común. En el caso particular de tipos de datos complejos, como los encontrados en programación orientada a objetos, una de las técnicas utilizadas se basa en el uso de mecanismos de reflexión provistos por el lenguaje, y especificaciones para evitar la generación de entradas inválidas. Sin embargo, cuando las especificaciones son débiles pueden llevar a construir objetos espurios, es decir, inválidos o no construibles por la API asociada al tipo del objeto. Estos objetos pueden llevar a falsos negativos: tests que fallan cuando no existe un bug e incrementan el trabajo del tester que deberá filtrar los tests que efectivamente evidencian un bug de aquellos que fallan solo por una entrada inválida; y falsos positivos: tests que deberían fallar pero que no lo hacen debido a que la entrada inválida enmascara el bug. En este trabajo evidenciaremos el problema mediante un ejemplo y delinearemos los componentes y pasos necesarios para construir una técnica que detecte cuando un objeto es inválido con respecto al API.

1. Introducción

Garantizar que un producto de software realiza correctamente las tareas para las cuales fue desarrollado corresponde a una de las tareas más importantes en ingeniería de software. Corresponde a un problema indecidible que puede ser atacado mediante enfoques analíticos (ej.: cálculo de precondición más debil) no completamente automatizables, o por un enfoque empírico el cual puede ser completamente automatizado.

Testing es uno de los enfoques más utilizados para garantizar que un producto de software realiza correctamente las tareas para las cuales fue desarrollado [6,12,8]. Siendo el testing un enfoque empírico, las garantías que ofrece son débiles en comparación con enfoques analíticos. Aunque dentro de sus ventajas se encuentra su facilidad de uso y su escalabilidad, este enfoque consiste en ejecutar el programa bajo evaluación sobre un conjunto (finito) de entradas y contrastar el comportamiento del programa obtenido con el esperado [1]. Evidentemente cómo se elige el conjunto de entradas sobre las cuales evaluar al programa, tiene un gran impacto en la garantía de que el programa es correcto, cuando los tests pasan.

Generar tests de manera automática es actualmente un área importante de investigación en ingeniería de software. Para comprender las dificultades de este problema es necesario considerar las tres partes que constituyen un test. **Preparación**, la combinación de los argumentos del programa a evaluar y el estado del sistema sobre el que se va a ejecutar al mismo; **Ejecución**, la ejecución del programa sobre el escenario anterior; **Evaluación**, la evaluación del comportamiento obtenido de la ejecución del programa, incluyendo resultados y cambios de estado del sistema, con respecto al comportamiento esperado. En este trabajo nos centraremos en la generación automática de entradas.

Para tipos de datos básicos como tipos primitivos, arreglos de tipos primitivos, y algunas estructuras de datos, la generación automática de valores es relativamente simple. Sin embargo, ciertas estructuras de datos requieren combinación de métodos con determinados valores y en un orden particular para producir una instancia específica de dicha estructura. Un caso simple es una lista simplemente encadenada con cache de nodos (*NodeCachingLinkedList* [2]). Esta estructura funciona como una lista simplemente encadenada con la diferencia de que al eliminar un elemento de la lista, el nodo asociado se guarda en una cache en lugar de dejar que el *garbage collector* lo elimine de la memoria, o para evitar la llamada de métodos de destrucción como en *C++*, luego al insertar un elemento se reutilizan los nodos almacenados en la cache. Para esta estructura, producir una instancia de una lista vacía con dos nodos en la cache, requiere llamar dos veces al método *insertar* y luego dos veces al método *eliminar*. En la sección 2 se mencionarán las dos metodologías más utilizadas para generar entradas para tipos complejos de datos.

Una de las técnicas utilizadas para generar entradas de tipos complejos se basa en el uso de mecanismos de reflexión e invariantes de representación. Cuando estos invariantes son débiles, es decir, fallan en considerar aspectos cruciales sobre la correcta implementación de un tipo, estas técnicas pueden llevar a la construcción de entradas inválidas. Nuestra propuesta es mostrar en detalles este problema y sus efectos negativos en el contexto de testing, y esbozar una técnica que permita la detección de estas entradas inválidas.

2. Preliminares

Como mencionamos, la generación automática de entradas para datos complejos no es una tarea trivial. Los dos enfoques utilizados actualmente para la generación de entradas para estructuras de datos complejas son:

Generación basada en reflexión y especificaciones, es utilizada en lenguajes que proveen los mecanismos apropiados para modificar valores de un objeto en tiempo de ejecución, ignorando los modificadores de acceso al mismo (ej.: modificar directamente el valor de un campo *private* en *Java*). No utilizar la *API* para construir las entradas, lleva a que sea posible generar instancias inválidas, por esto es que este tipo de generación requiere que se provea una especificación del invariante del tipo de la entrada, comúnmente provista por un método *repOK* que no toma parámetros y retorna si una instancia satisfa-

ce o no el invariante del tipo. Herramientas basadas en este enfoque incluyen *Korat* [3] para generación de entradas, y *TACO* [5] como herramienta de verificación automática de especificaciones que utiliza reflexión y especificaciones para construir contraejemplos[1].

Generación basada en secuencias de métodos, que utiliza la *API* del tipo de la entrada a generar para construir instancias particulares de la misma. Este enfoque tiene la ventaja de no generar instancias inválidas si la implementación es correcta, mientras que la generación basada en reflexión puede utilizarse incluso si los métodos de la *API* están incorrectamente implementados o directamente no están provistos. *EvoSuite* [4] y *Randoop* [11] son ejemplos de herramientas que generan entradas[2] mediante el uso de secuencias de métodos.

3. Objetos espurios en generación automática de entradas

En este trabajo nos vamos a centrar en la generación de entradas de tipos de datos complejos basada en reflexión y especificaciones, cuando éstas últimas son débiles, más precisamente, cuando las instancias generadas no pueden ser construidas mediante los métodos asociados al tipo de la entrada (*API*).

Es necesario, antes de continuar, introducir la noción de invariante de representación y que significa que éste sea débil: *Un invariante de representación establece las propiedades que debe cumplir un objeto para ser considerado válido con respecto al tipo abstracto que representa*[3] [9].

Si la especificación utilizada es muy débil habrá instancias de un tipo concreto que sean erróneamente consideradas como representaciones válidas del tipo abstracto. Por ejemplo, que el invariante de *NodeCachingList* no exija que un nodo no pertenezca al mismo tiempo a la cache y a la lista.

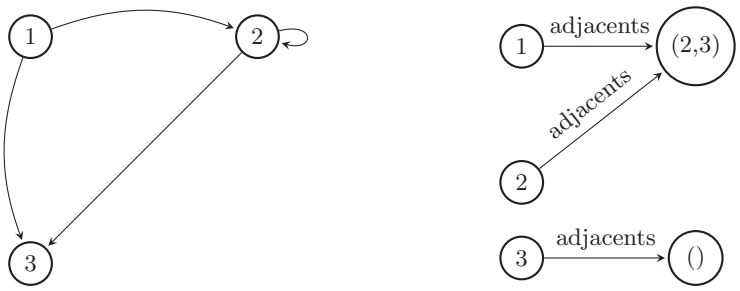
A modo de motivación, consideremos un grafo dirigido *DirectedGraph* implementado mediante listas de adyacencias. La implementación se basa en un arreglo de vértices donde cada uno posee un identificador y una lista de vértices adyacentes. El grafo de la Figura 1a debería estar representado por un arreglo de 3 vértices con, etiqueta 1 y la lista *L1(2,3)* para el primero, etiqueta 2 y lista *L2(2,3)* para el segundo, y finalmente etiqueta 3 y lista *L3()* para el tercero. Si bien los métodos de grafos sobre listas de adyacencias deberían asegurar la representación para la instancia anterior, una construcción basada en reflexión podría utilizar la misma lista para los vértices 1 y 2, tal como se muestra en la Figura 1b, si la especificación del invariante no verifica que cada vértice debe estar asociado con una lista única. Al utilizar *Korat* para generar instancias de *DirectedGraph* una de las obtenidas es justamente la que acabamos de presentar.

[1] Entradas particulares que cumplen con la precondición del programa a evaluar pero causan que se viole la postcondición del mismo.

[2] Como parte de generación automática de tests

[3] Desde otra perspectiva, un invariante de representación define el subconjunto de todas las instancias de un tipo concreto que representan a un tipo abstracto.

[4] A modo de simplificación solo se muestran los vértices del grafo con sus correspondientes listas de adyacentes.



(a) Grafo dirigido con tres nodos.

(b) Instancia inválida representando el grafo en la Figura 1a.[4]

Figura 1: Ejemplo de objeto espurio.

En el contexto de testing, la segunda instancia podría llevar a falsos negativos, es decir, un test que falla porque la entrada es inválida; y a falsos positivos cuando existe un bug pero no es detectado porque la entrada inválida hace que éste no sea visible. Consideremos la API de *DirectedGraph* de la Figura 2 y los dos tests de la Figura 3.

El primer test de la Figura 3 falla porque al usar la misma lista de adyacentes para los vértices 1 y 2, al conectar 1 con 1, también se conecta 2 con 1. El segundo test utiliza la misma instancia del anterior, y el método *disconnectVertex* que está incorrectamente implementado, sin embargo el test pasa porque al estar compartida la lista de adyacentes entre los vértices 1 y 2, eliminar el vértice 3 de los adyacentes del vértice 1 logra el mismo efecto para el vértice 2.

4. Detectando objetos espurios

Dado un objeto generado mediante reflexión, y la API asociada al tipo del mismo, detectar si este objeto es inválido consiste en verificar que éste no pertenece al conjunto de todos los objetos construibles mediante la API. Considerando que generar todo el conjunto de objetos construibles resulta imposible o sumamente ineficiente en el mejor caso, es necesario considerar el siguiente problema:

> *Dado un objeto y una API, buscar si existe una secuencia de operaciones provistas por la misma que permita construir el objeto.*

Esto es consistente con un problema de búsqueda, en el sentido general ya que el problema consiste en buscar una solución en un conjunto de soluciones candidatas, a continuación analizaremos todos los subproblemas que requieren ser resueltos antes de poder desarrollar una técnica para detectar si un objeto generado mediante reflexión es inválido (no construible mediante la API) o si es posible su construcción, en cuyo caso se debería proveer una secuencia de operaciones que lo construye.

```
public class DirectedGraph {
  /*
  * Los vértices del grafo, cada Vertex tiene una
  * etiqueta asociada y una lista de adyacentes.
  */
  private Vertex[] vertices;

  /**
  * Crea un nuevo grafo con n vértices
  */
  public DirectedGraph(int n) {...}

  /**
  * Conecta el vértice a con el vértice b
  */
  public void addEdge(int a, int b) {...}

  /**
  * Remueve la conexión del vértice a al b
  * (no remueve la conexión inversa)
  */
  public void removeEdge(int a, int b) {...}

  /**
  * Remueve todas las conexiones hacia un nodo
  */
  public void disconnectVertex(int a) {
    for (Vertex v : vertices) {
      if (connected(v.label, a)) {
        removeEdge(v.label, a);
        break; //bug
      }
    }
  }

  /**
  * Retorna true si y solo si existe un camino
  * desde el vértice a al vértice b
  */
  public boolean connected(int a, int b) {...}

}
```

Figura 2: Implementación de la clase *DirectedGraph*.

4.1. Representación

Representar a cada elemento en el espacio de búsqueda directamente como un objeto construido mediante la API es posible pero lleva acarreado varios problemas: dificultad en obtener la secuencia de operaciones que lo construye, para obtener a la misma sería necesario llevar un historial de que operaciones (y con que valores) fueron aplicadas; y dificultad en la implementación de enfoques asociados a algoritmos genéticos, teniendo en cuenta que los elementos dentro de estos algoritmos se deben representar en forma de genes para las operaciones de *crossover* en donde los genes de dos candidatos son *entrecruzados* para generar un nuevo candidato [10]. A partir de estas observaciones creemos que la mejor representación sería utilizar la secuencia de operaciones (junto a sus argumentos) que construye cada objeto en el espacio de búsqueda.

```
@Test
public void addEdgeTest() {
  //Falso negativo, el test falla pero no existe un bug
  DirectedGraph graph = invalidInstance();
  graph.addEdge(1,1);
  assertTrue(graph.connected(1,1));
  assertFalse(graph.conneted(2,1)); //falla
}

@Test
public void disconnectNodeTest() {
  //Falso positivo, el test pasa cuando existe un bug
  DirectedGraph graph = invalidInstance();
  graph.disconnectVertex(3);
  assertFalse(graph.connected(1,3)); //no falla
  assertFalse(graph.connected(2,3)); //no falla
  assertFalse(graph.connected(3,3)); //no falla
}
```

Figura 3: Dos tests para *DirectedGraph* en donde el tester piensa utilizar el grafo de la Figura 1a pero obtiene la representación inválida de la Figura 1b.

4.2. Exhaustividad del espacio de búsqueda

El problema tal como fue planteado anteriormente requiere acotar el espacio de búsqueda, dentro de las cotas provistas es posible considerar de manera exhaustiva a todas las secuencias de operaciones a analizar, o puede considerarse un subconjunto. La ventaja de no hacer un análisis exhaustivo es principalmente la disminución en los recursos requeridos (tiempo y memoria principalmente), la desventaja es que no ser exhaustivo puede llevar a no encontrar una secuencia de operaciones que existe dentro de las cotas consideradas, aumentando las chances de dar una respuesta negativa y que se utilicen cotas más grandes que las necesarias para intentar encontrar una secuencia que construya al objeto buscado. Esto lleva a una observación importante, si el objeto no pudo ser construido no necesariamente significa que éste sea inválido, solo que dentro del espacio de candidatos evaluado, no existe una secuencia de operaciones que lo construya.

A modo de ejemplo consideremos un tipo *List* con solo el método *add(int)*, que agrega un valor al final de la lista, y la instancia *(3,3)*. Las cotas que se deben establecer son la longitud máxima de las secuencias y que valores se van a utilizar (en este caso por el método *add(int)*). Definir que valores asignar a estas cotas no es un problema trivial, cotas muy restrictivas llevan a no poder encontrar la secuencia que construye el objeto, incluso cuando ésta existe; mientras que cotas muy relajadas tienen asociadas un crecimiento del espacio de búsqueda, considerando una solución por búsqueda basada en recorrido de árboles (*Depth First Search*, *Breadth First Search*, *Best First Search*, etc), el tamaño del espacio de búsqueda está dado por la formula $\sum_{i=0}^{h} m^i$, con h la cota establecida para el tamaño máximo de las secuencias de operaciones a considerar, y m por cuantas operaciones y valores utilizar. Específicamente para el ejemplo de *List*, h es el tamaño máximo de las listas, y m la cantidad de valores a utilizar para el método

add(int). En la Figura 4 se muestra el árbol de búsqueda para el ejemplo de anterior con las cotas mencionadas.

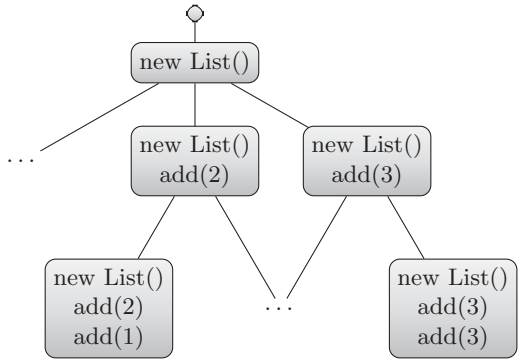


Figura 4: Búsqueda para la lista *(3,3)* dentro de un espacio acotado de todas las listas de tamaño 0 a 3 con valores a insertar del 1 al 3 y solo utilizando el constructor vacío y el método *add(int)*.

4.3. Operaciones redundantes y observadoras

Métodos observadores: No todas las operaciones asociadas a un tipo realizan modificaciones al estado, muchas permiten observar datos del objeto (ya sean datos computados a partir del estado, ej.: *List#contains(int)*, o acceso de lectura a valores específicos, ej.: *List#size()*). Estas operaciones nunca deberían ser necesarias para construir un objeto particular, y a su vez incrementan el espacio de búsqueda.

Detectar cuando una operación es observadora corresponde a un problema indecidible, aunque es posible dar soluciones aproximadas. En [7] los autores presentan un sistema de tipos junto a verificación estática para identificar métodos observadores (llamados también *puros*).

Operaciones redundantes: No es raro encontrar APIs que definen operaciones similares donde una se puede definir en base a la otra, como por ejemplo las operaciones de *add(int)* y *add(int, int)* de *List* que agregan un elemento al final de la lista o en un índice particular respectivamente, es posible definir la primera a partir de la segunda como *add(i) = add(i, size())*. Este tipo de operaciones no son equivalentes pero generan un conjunto de secuencias representando al mismo objeto lo que a su vez lleva a incrementar el espacio de búsqueda. Si bien este también es un problema indecidible, resulta incluso más complejo dar una solución aproximada.

4.4. Evaluación de un candidato

Finalmente un punto muy importante en todo problema de búsqueda es poder determinar cuando la solución es finalmente encontrada. Para el problema planteado en este trabajo, esta evaluación deberá constatar si el objeto de entrada fue construido por alguna secuencia. La comparación se deberá hacer entre el objeto a buscar y el objeto generado por la secuencia bajo evaluación. Comúnmente funciones como *equals* o *compareTo* podrían ser utilizadas pero tal como la funciones de hash o de representación (al estilo *toString()* en *Java*) muchas veces ignoran ciertos valores de una estructura, o utiliza valores que no hacen a la misma, particularmente en *Java* el hash por defecto contiene valores aleatorios, esto lleva a que objetos distintos puedan ser considerados iguales o que objetos iguales puedan ser considerados distintos, ambos casos son perjudiciales para el proceso de búsqueda. También es necesario considerar que estas funciones pueden no estar provistas, o aplicarlas sobre un objeto inválido puede llevar a excepciones.

5. Delineamientos para detectar un objeto espurio

En base a lo discutido en la sección 4 delinearemos a continuación una técnica para detectar objetos espurios. El objetivo es dar un punto de inicio para el desarrollo de una técnica eficaz, siempre dentro del contexto de búsqueda acotada, y eficiente.

Nuestro primer enfoque se basa la generación exhaustiva acotada de objetos mediante la API, para la construcción del espacio de búsqueda. El resultado de esta etapa es un conjunto de objetos, la ventaja de este enfoque inicial es que es posible cambiar la herramienta de generación del espacio de búsqueda sin tener que modificar la búsqueda en el espacio de candidatos. La evaluación de candidatos, es decir, dado un candidato c evaluar si se cumple $equal(c, o)$ donde o representa el objeto objetivo, requiere una representación de los objetos a comparar, anteriormente en 4.1 se mencionó el uso de funciones provistas por el mismo objeto (*equals*, *compareTo*, *hashCode*, y *toString*), todas con problemas asociados. Creemos que una representación basada en recorrer la estructura de un objeto siguiendo un orden determinado de los campos (como recorrido Depth first search) permite independizarse de la existencia y correcta implementación de los métodos anteriores evitando que información necesaria no sea considerada e información irrelevante (como el hash por defecto de cada objeto) lo sea.

La generación exhaustiva acotada basada exclusivamente en el uso de operaciones de la API presenta dos problemas importantes: pueden existir varias secuencias de operaciones que generen el mismo objeto; y acotar el tamaño de las secuencias generadas no necesariamente acota de la misma forma a la estructura, a modo de ejemplo consideremos una *NodeCachingList*, para construir la lista que tiene un solo elemento y dos nodos en la cache es necesario 6 operaciones: la construcción de la lista, agregar 3 elementos y eliminar 2; mientras que la estructura solo contiene 2 nodos en la cache y 1 en la lista. Por lo tanto es necesario que la herramienta permita utilizar cotas sobre la estructura. Obtener

automáticamente las cotas a utilizar representa un subproblema muy complejo de resolver, nuestro enfoque considera que las cotas a utilizar son provistas por el usuario. A su vez, la exhaustividad de la generación estará basada en los objetos a generar, en lugar de las secuencias a generar. Para esto consideraremos todas las relaciones entre campos y valores de acuerdo a las cotas provistas y el objetivo consiste en "cubrir" todas las relaciones durante la generación del espacio de candidatos.

Finalmente la búsqueda es inicialmente lineal, tal como se puede apreciar en el algoritmo de la Figura 5, en donde *generateCandidates* representa la herramienta de generación exhaustiva, basada exclusivamente en el uso de operaciones de la API, y *fieldBasedRepresentation* es el método que retorna la representación discutida anteriormente.

Para retornar una secuencia de operaciones que construye a un objeto solo es necesario considerar los candidatos como listas de operaciones y disponer de un método que retorne un objeto a partir de estas operaciones.

```
isValid(Object o, Bounds b) {
  List<Object> candidates = generateCandidates(b);
  Representation oRep = fieldBasedRepresentation(o);
  for (Object c : candidates) {
    Representation cRep = fieldBasedRepresentation(c);
    if (oRep.equals(cRep)) {
      return true;
    }
  }
  return false;
}
```

Figura 5: Algoritmo inicial para detección de objetos inválidos.

6. Conclusiones

Hemos presentado el problema de objetos inválidos en la generación automática de entradas basada en mecanismos de reflexión y especificaciones. Inicialmente presentamos un ejemplo de un tipo para el cual la herramienta *Korat* genera un objeto inválido. Luego presentamos el problema de detección de objetos inválidos como un problema de búsqueda detallando que subproblemas surgen. Finalmente dimos un esbozo de una técnica para detectar estos casos. Es importante destacar que el ejemplo mostrado fue satisfactoriamente detectado con una implementación inicial del algoritmo propuesto. Sin embargo es claro ver que este enfoque no escala al considerar objetos "más grandes" que requieren aumentar las cotas utilizadas y por ende realizar la búsqueda sobre un espacio de candidatos mucho mayor. Esto abre una vía de investigación para resolver este problema de manera más eficiente, considerando el objetivo final

de obtener la secuencia que construye (si existe dentro de las cotas provistas) un objeto particular.

Referencias

1. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.
2. Apache Software Foundation. Commons collection. `https://commons.apache.org/proper/commons-collections`.
3. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.
4. Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.
5. Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.
6. Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall, 1991.
7. Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: checking and inference of reference immutability and method purity. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 879–896. ACM, 2012.
8. Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag, Berlin, Heidelberg, 2nd edition, 1997.
9. Barbara Liskov and John V. Guttag. *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
10. Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.
11. Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 815–816. ACM, 2007.
12. Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.