3

PROGRAMACIÓN POR CONTRATO

3.1 ASERCIONES

Es habitual, y no por ello deja de ser problemático, no prestar la debida atención a la semántica del software que se desarrolla. Con frecuencia, se intenta implementar rápidamente un comportamiento sin haberlo definido con exactitud antes. Vamos a volver a las raíces pero sin perder de vista lo aprendido hasta ahora.

Como ya estudiamos en el primer capítulo, implementar un tipo abstracto de datos implica simular la semántica de dicho tipo abstracto de datos mediante la semántica de una clase. He aquí muy resumido en qué consiste el desarrollo orientado a objetos. Ahora bien, damos por supuesto que la semántica del tipo abstracto de datos está bien definida por su especificación matemática, la cual no es materia de estudio en esta obra. De hecho, en la práctica de la vida real, es improbable que tengamos a nuestra disposición la especificación matemática de un tipo abstracto de datos para, comenzando a partir de ahí, desarrollar nuestro software. Lo habitual es encontrarnos con una descripción informal en lenguaje natural del tipo abstracto de datos o, lo que por desgracia es más frecuente de lo que debería ser, sólo un conocimiento intuitivo del tipo abstracto de datos del que necesitamos realizar la implementación. Por tanto, es necesario un método que permita especificar correcta y formalmente la semántica de una clase, en base a la información que se nos haya proporcionado de cuál debe ser su comportamiento.

Este método es la Programación por Contrato. El término original en inglés es *Design by Contract* y fue inventado y registrado por Bertrand Meyer, creador del lenguaje de programación Eiffel. La Programación por Contrato no es ni más ni menos que la aplicación práctica de conceptos y teorías de la ciencia de la computación.

Para prestar la atención debida a la semántica del software, tenemos que tener en cuenta que lo que nos interesa de verdad de un elemento software (sea una clase, un método o una única instrucción) es que sea correcto. Pero lo primero que habría que preguntarse es qué significa que un elemento software sea correcto. Para considerar significativa la pregunta, se necesita tener no sólo el elemento software, sino también una descripción precisa de su semántica, o sea, de lo que se supone que este elemento software debe hacer, es decir, una especificación. Por tanto, un elemento software no es correcto o incorrecto por sí mismo, sino que es correcto o incorrecto con respecto a cierta especificación.

Otra forma de verlo. Podríamos decir que la descripción de lo que un elemento software hace realmente es su semántica real; la descripción de lo que un elemento software debería hacer es su semántica intencional o simplemente semántica. Cuando ambas coinciden, el elemento software es correcto.

Estrictamente hablando, no se debería discutir sobre si un elemento software es o no correcto, sino sobre si es consistente con su especificación. No obstante, se utiliza el término bien aceptado de corrección pero no se aplica a un elemento software aislado: Se aplica al par formado por un elemento software y su especificación. Así pues, la corrección es un concepto relativo.

Vamos a aprender la forma de expresar la especificación de un elemento software con el fin de establecer la corrección del mismo. Para expresar dicha especificación usaremos aserciones.

Sea **A** un cierto elemento software de un programa. Una **fórmula de corrección**, también llamada **Tripleta de Hoare**, es una expresión de la forma:

```
{P} A {Q}
```

Que denota la propiedad siguiente, que puede ser cierta o no:

Una ejecución de **A** que comience en un estado de ejecución en el que se cumple **P** terminará en un estado de ejecución en el que se cumple **Q**.

Un **estado de ejecución** de un programa es el conjunto formado por los Valores de todas las variables alcanzables de dicho programa en un momento dado de su ejecución.

P y **Q aserciones**, **P** es la **precondición** y **Q** es la **postcondición**. Una Aserción es una expresión booleana que involucra algunas variables de un programa y que establece una propiedad lógica que dichas variables deben satisfacer en uno o más estados de ejecución.

Si un estado de ejecución cumple una aserción, se dice que dicho estado de ejecución satisface dicha aserción.

Una aserción **A** implica otra aserción **B** o, lo que es lo mismo, una aserción **B** es consecuencia lógica de otra aserción **A** si y sólo si todo estado de ejecución que **A** también satisface **B**. De forma semejante, dos aserciones son equivalentes si las satisfacen los mismos estados de ejecución.

En lugar de hablar de los estados de ejecución que satisfacen una aserción, podemos hablar de forma completamente equivalente de los estados de ejecución definidos por una aserción, de modo tal que toda aserción define un conjunto de estados de ejecución formado por todos aquellos estados que la satisfacen.

De este modo, la propiedad denotada por la fórmula de corrección, que puede ser cierta o no, es interpretada del siguiente modo:

Una ejecución **A** que comience en un estado de ejecución definido por **P** terminará en un estado de ejecución definido por **Q**.

Cuanto mayor sea el conjunto de estados que define una aserción, diremos que más débil es. A la inversa, cuanto menor sea el conjunto de estados que define, diremos que más fuerte es.

Se dice que la aserción **A** es más fuerte que la aserción **B** o, lo que es lo mismo, una aserción **B** es más débil que otra aserción **A** si y sólo si el conjunto de estados definidos por **A** está incluido en el conjunto de estados definido por **B**. Obviamente, la aserción **A** es más fuerte que la aserción **B** si y sólo si **A** implica **B**; o, dicho de otro modo, **B** es más débil **A** si **B** es consecuencia lógica de **A**.

Por todo ello, la aserción **Falso** es la más fuerte posible ya que define el conjunto vacío de estados o, lo que es equivalente, implica cualquier otra aserción; de forma semejante, la aserción **Verdadero** es la más débil posible ya que define el conjunto de todos los estados posibles o, lo que es equivalente, es consecuencia lógica de cualquier otra aserción.

Este repaso breve a conceptos básicos de la ciencia de la computación nos va a resultar útil. La Programación por Contrato recoge estos conceptos y utiliza las aserciones como una forma de reintroducir en la clase las propiedades semánticas del tipo abstracto de datos subyacente. Las aserciones así pasan, de ser una mera construcción teórica para guiar el razonamiento en el desarrollo del software, a tener también una aplicación práctica.

La Programación por Contrato aplica de forma práctica las aserciones definiendo, en primer lugar, una precondición y una postcondición para cada método

de la clase. Llegado este punto, puede que nos preguntemos qué sentido tiene la palabra contrato en la Programación por Contrato. Quizá sorprenda la respuesta. **Una tripleta de Hoare** donde **A** es un método, **es un contrato** de ese método que lo vincula con quienes lo invocan. Pero veámoslo con calma.

El primer uso de las aserciones es la especificación semántica de los métodos. Un método no es solamente un trozo de código; como implementación de alguna operación de la especificación de un tipo abstracto de datos, debería realizar alguna tarea útil. Es necesario expresar esta tarea con precisión, como ayuda para el diseño -no se puede aspirar a asegurar que un método sea correcto a menos que se haya especificado lo que se supone que hace- y como ayuda para la comprensión posterior del texto del código fuente. Para especificar la tarea se necesita algo más que el nombre del método, que sólo da una idea imprecisa de lo que hace. Se puede especificar la tarea que lleva a cabo un método mediante dos aserciones asociadas al método, como ya sabemos: Una precondición y una postcondición.

La precondición establece las propiedades que se tienen que cumplir cada vez que se llama al método; es decir, expresa las restricciones bajo las que un método funcionará correctamente. La precondición se aplica a todas las llamadas al método, tanto desde dentro de la clase como desde los clientes. Un sistema correcto nunca ejecutará una llamada en un estado que no satisfaga la precondición del método al que se llama.

La postcondición establece las propiedades que debe garantizar el método cuando retorne; es decir, expresa propiedades del estado resultante de la ejecución de un método. La presencia de una postcondición en un método expresa una garantía por parte de quien implementa el método de que éste producirá un estado en el que se satisfacen ciertas propiedades si se supone que ha sido invocado satisfaciéndose la precondición.

Por tanto, definir una precondición y una postcondición para un método es una forma de definir un contrato que vincula al método con quienes lo llaman.

Al asociar una precondición **pre** y una postcondición **post** con un método **m**, la clase le está diciendo a sus clientes:

Si usted me promete llamar a **m** con **pre** satisfecho, entonces yo le prometo entregar un estado final en el que **post** es satisfecho.

Un par precondición-postcondición de un método describe el contrato que el método -es decir, el proveedor de un cierto servicio- define para los que lo llaman -los clientes del servicio-. Un buen contrato entraña tanto obligaciones como

beneficios para ambas partes, ya que lo que es una obligación para uno se convierte en un beneficio para el otro. Así:

- La precondición compromete al cliente: Define las condiciones bajo las cuales es legítima la llamada a un método. Es una obligación para el cliente y un beneficio para el proveedor.
- La postcondición compromete al proveedor: Define las condiciones que debe asegurar el método al retornar. Esto es un beneficio para el cliente y una obligación para el proveedor.

Los beneficios son para el cliente la garantía de que ciertas propiedades se van a cumplir después de la llamada, y para el proveedor que se puede suponer que determinados presupuestos se van a cumplir cada vez que se llame al método.

Las obligaciones para el cliente son satisfacer los requisitos que establecen las precondiciones, y para el proveedor cumplir con la tarea que establece la postcondición.

La precondición es un beneficio para el proveedor porque, si el cliente no cumple con su parte, es decir, si la llamada no satisface la precondición, entonces la clase no está obligada a cumplir la postcondición. En este caso, es lícito para el método hacer lo que quiera: Retornar cualquier valor, caer en un bucle infinito sin retornar valor alguno o incluso acabar con la ejecución de una manera drástica lanzando una excepción.

La primera ventaja de esta convención es que simplifica considerablemente el estilo de programación. Habiendo especificado como precondición las restricciones bajo las cuales se puede llamar a un método, el que desarrolla la clase puede suponer, cuando escribe el cuerpo del método, que las restricciones se satisfacen, por lo que no es necesario verificar éstas en el cuerpo del método; de hecho, la Programación por Contrato va más allá: No solamente es innecesario, sino que es inaceptable. Es lo que se conoce como **principio de no redundancia**:

- Bajo ninguna circunstancia debe el cuerpo del método verificar el cumplimiento de su precondición.

El principio de no redundancia realmente indica que, para cualquier condición de consistencia que pudiera hacer peligrar el funcionamiento apropiado de un método, se le debe asignar el asegurar esta condición a una sola de las dos partes del contrato. ¿A cuál de las dos? La respuesta puede variar, y es en parte un asunto de estilo de diseño. Hay dos posibilidades:

- O se le asigna la responsabilidad a los clientes, en cuyo caso la condición aparecerá como parte de la precondición del método, y recibe el nombre de **diseño exigente de las precondiciones**.

- O se le pasa al proveedor en cuyo caso la condición aparecerá en una instrucción condicional en el cuerpo del método, y recibe el nombre de **diseño tolerante de las precondiciones**.

¿Cuál es el mejor estilo? Hasta cierto punto es un asunto de criterio personal, al contrario que el principio de no redundancia, que es absoluto cuando establece que nunca es aceptable tratar una condición de corrección desde ambos lados, el del cliente y el del proveedor.

En nuestro caso, consideramos más apropiado el diseño exigente de las precondiciones, ya que el autor de un método no tiene que ser más listo que los clientes; si no está seguro de lo que tiene que hacer el método en una cierta situación anormal, debe excluir ésta explícitamente en la precondición.

Ahora bien, el diseño exigente de las precondiciones sólo es aplicable si las precondiciones se mantienen razonables. ¿Qué es lo que significa "razonable" para la precondición de un método? A continuación se da una caracterización más precisa, mediante el **principio de la precondición razonable**:

La precondición de todo método en el diseño exigente de las precondiciones debe satisfacer los siguientes requisitos:

- La precondición aparece en la documentación oficial distribuida a los autores de los módulos clientes.

- Es posible justificar la necesidad de una precondición exclusivamente en términos de la especificación.

El segundo requisito excluye restricciones que sean sólo para conveniencia del proveedor al implementar el método. Ahora bien, algunas restricciones pueden surgir debido a la forma general de implementación que se seleccione. Por ejemplo, utilizar un array estático para implementar las pilas tendrá como consecuencia que el método **push()** (que apila un elemento en la cima de la pila) tenga una precondición que exija que la pila no esté llena. Pero un caso como éste no viola el principio porque la naturaleza acotada de dicha implementación se hace parte de la especificación: La clase no se anunciará para representar pilas de cualquier tamaño, sino sólo pilas de una capacidad máxima finita.

El principio de no redundancia es contrario a lo que se conoce a menudo con el nombre de programación defensiva. Ésta establece que para obtener software

fiable hay que diseñar componentes que se protejan a sí mismos todo lo posible. Es mejor comprobar demasiado, según este enfoque, que demasiado poco: Uno nunca es demasiado cuidadoso cuando tiene que tratar con extraños. Una comprobación redundante puede no ayudar, pero al menos no hace daño.

La Programación por Contrato proviene de la observación opuesta: Las comprobaciones redundantes pueden hacer daño. Si se restringe el panorama al estrecho mundo de un único método, entonces puede parecer que dicho método es más robusto con una comprobación adicional que sin ella. Pero el mundo de un sistema software no está restringido a un único método: Contiene una multitud de métodos en una multitud de clases. Para obtener sistemas software fiables, se debe pasar de este enfoque tan microscópico a uno macroscópico que considere la arquitectura completa.

Si se considera esta visión global, la sencillez se convierte en un criterio crucial. Como ya se sabe, la complejidad es el mayor enemigo de la calidad. Cuando se tiene esto en cuenta, las posibles comprobaciones redundantes ¡ya no parecen tan inocuas! Extrapoladas a miles de métodos de un sistema software de tamaño medio -o decenas o cientos de miles de métodos en uno muy grande-, una comprobación adicional basada en una única instrucción condicional, que es inocua a primera vista, comienza a parecer un monstruo de complejidad inútil.

La Programación por Contrato invita a identificar las condiciones de consistencia que son necesarias para el funcionamiento correcto de cada cooperación cliente-proveedor -es decir, cada contrato- y a especificar, para cada una de estas condiciones, de quién es la responsabilidad de asegurar la misma, del cliente o del proveedor. Una vez que se ha tomado una decisión, hay que basarse en ella: Si en una precondición aparece un requisito de corrección, lo cual indica que el requisito es parte de la responsabilidad del cliente, no debe haber una comprobación correspondiente en el método; y, si no está en una precondición, entonces el método debe comprobar el requisito.

Es importante entender que los contratos son una herramienta excelente para mejorar la corrección y seguridad de nuestro software, pero no dejan de ser precisamente eso, una herramienta más que complementa a otras. A veces puede ocurrir que en un proyecto no es posible utilizarlos. Y no hay que perder de vista que el desarrollo de software tiene sus propias reglas: Reglas debidas al hecho de que se ejecuta en máquinas reales que están limitadas en espacio de memoria y en velocidad de ejecución. Hay momentos en los que es necesario tomar decisiones de ingeniería y evaluar que es mejor no ir más allá en la especificación de las clases para no afectar en demasía al rendimiento de la ejecución del programa. Porque, efectivamente, por si alguien se lo había preguntado, comprobar que se cumplen los contratos de la

clase durante la ejecución del programa supone una penalización en el rendimiento del mismo.

En particular, en la programación de videojuegos, donde se busca maximizar el rendimiento de la aplicación que se está ejecutando, es buena idea limitar qué contratos son aquellos que se verifican en tiempo de ejecución. Generalmente, sólo se comprueban las precondiciones.

Existen una serie de heurísticas que guían en el descubrimiento de las aserciones de una clase:

- Dividir métodos en órdenes y consultas. Las consultas devuelven un resultado pero no cambian las propiedades visibles del objeto. Las órdenes pueden cambiar el objeto pero no devuelven un resultado. Seguir este principio, que ya conocemos, es importante porque permite usar consultas en aserciones sin temor de que puedan cambiar el estado del objeto.

- Separar las consultas básicas de las consultas derivadas. Las consultas derivadas pueden ser especificadas en términos de las consultas básicas.

- Para cada consulta derivada, escribir una postcondición que especifique qué resultado será devuelto, en términos de una o más consultas básicas. Así, si sabemos los valores de las consultas básicas, también sabemos los valores de las consultas derivadas.

- Para cada orden, escribir una postcondición que especifique el valor de cada consulta básica. Así sabemos el efecto total visible de cada orden.

- Para cada consulta y cada orden, decidir una precondición apropiada.

- Escribir invariantes para definir propiedades invariables de los objetos. Hay que concentrarse en propiedades que ayudan al lector de la clase a construirse un modelo conceptual apropiado de la abstracción que la clase representa.

- Añadir restricciones físicas donde sea apropiado. Una de las restricciones más habituales es exigir que las variables no sean nulas.

- Intentar que las consultas usadas en las precondiciones sean poco costosas de calcular.

- Restringir atributos usando el invariante. Cuando una consulta derivada es implementada como un atributo, puede ser restringida a ser consistente con otras consultas mediante una aserción en el invariante de la clase.

Realmente, los contratos establecen supuestos que deben cumplirse para que el software se ejecute correctamente. Hacen explícito lo que está implícitamente codificado en el sistema software. Sin contratos, estos supuestos son usualmente indicados como comentarios.

Las precondiciones y las postcondiciones describen las propiedades semánticas de los métodos individuales. También es necesario expresar las propiedades semánticas globales de todas las instancias de una clase, que deben ser preservadas por todos los métodos. Tales propiedades semánticas constituyen el **invariante de la clase** y capturan las restricciones de integridad que caracterizan a una clase.

Si bien se comprueba que se verifica la precondición a la entrada del método y que se verifica la postcondición a la salida del mismo, un invariante de una clase es una aserción que debe satisfacer cada instancia de la clase en todos los momentos estables. Los momentos estables de una instancia de una clase son aquellos en los que el objeto es observable desde fuera, en el sentido de que un cliente le puede aplicar una característica. Estos momentos estables son:

- El estado resultante de la creación del objeto.
- El estado inmediatamente antes y después de una llamada ejecutada por un cliente a un método público de la clase de ese objeto.

Por ello, si fuéramos exhaustivos en comprobar que se cumplen todos los contratos de una clase, habría que verificar el cumplimiento del invariante de clase en los siguientes casos:

- A la salida de los métodos constructores.
- A la entrada y salida de los métodos públicos y de los métodos con visibilidad de paquete.

Para que una precondición pueda ser satisfecha por un cliente, no debe utilizar características de la clase que estén ocultas a los clientes, es decir, que sean privadas. Tenemos así la **regla de disponibilidad de la precondición**:

- Toda característica que aparezca en la precondición de un método debe estar disponible para cualquier cliente para el que esté disponible dicho método.

Sin embargo, no hay una regla similar para las postcondiciones o los invariantes de clase. No es un error el que algunas cláusulas de una postcondición o del invariante de clase se refieran a atributos o métodos privados. Esto significa

simplemente que se están expresando propiedades del efecto del método que no son utilizables directamente por los clientes. Se llaman **propiedades de implementación**.

La ocultación de la información no versa sobre ocultar el código fuente al cliente -es decir, al autor de las clases cliente-, sino sobre que éste no pueda escribir su código basándose en características privadas, incluyendo entre ellas la representación interna de la clase. Por ello, las precondiciones deben usar siempre características públicas, que son visibles para el cliente, de modo que éste pueda comprobar que se cumplen. Sin embargo, las postcondiciones y los invariantes no tienen que cumplir este requisito: En este caso no es un error incluir características privadas porque ello no causa problemas al cliente. Aunque el cliente pueda leer las cláusulas de las postcondiciones y del invariante que muestran propiedades de implementación de la clase, del mismo modo que aunque pudiera leer el código fuente completo de la clase, no se viola el principio de ocultación de la información porque al cliente no se le permite escribir una clase cliente basándose en esas características privadas, incluida entre ellas la representación interna de la clase, por la simple razón de que no puede acceder a ellas.

Es importante hacer notar que los contratos que aquí se discuten ocurren entre un método -el proveedor- y otro método -el cliente que lo llama-. En la obtención de información del mundo externo -de entradas de usuario, de una red, de sensores, de un sistema de ficheros, etc. -, no se puede basar uno en precondiciones. En este contexto es útil el diseño tolerante de las precondiciones y en este caso no hay sustituto para las estructuras de control condicionales habituales.

Debe quedar claro que las aserciones no son estructuras de control para manejar casos especiales. Si se quiere escribir un método que calcule la raíz cuadrada de un número real y trate los argumentos negativos de cierta manera y los argumentos no negativos de otra, entonces no se necesita una precondición sino las clásicas estructuras de control condicionales.

Las aserciones son otra cosa. Expresan las condiciones de corrección. Si dicho método que calcula la raíz cuadrada de un número real tiene una precondición que exige que el argumento sea mayor o igual que cero, una llamada a dicho método con un argumento negativo no es un caso especial: Es un error simple y llano. Y dada la visión de las aserciones como contrato, podemos enunciar la **regla de violación de las aserciones**:

- La violación en tiempo de ejecución de una aserción es la manifestación de un error en el software.
- La violación de una precondición es la manifestación de un error en el cliente.

- La violación de una postcondición es la manifestación de un error en el proveedor.
- La violación de un invariante es la manifestación de un error en el proveedor.

La herencia de contratos funciona de la misma manera para la implementación de interfaces como para la herencia de clases. La clase que implementa una interfaz hereda los contratos de ésta y la clase hijo hereda los contratos de la clase padre. Más aún, los contratos se pueden refinar. Y, a diferencia del mecanismo de redefinición de métodos, los contratos no se reemplazan: La clase que implementa una interfaz combina sus contratos con los de ésta y la clase hijo combina sus contratos con los de la clase padre. Veamos cuáles son las reglas de combinación.

La **regla de herencia del invariante** establece que el invariante de la clase padre de una clase se aplica por igual a la clase hija; de manera análoga, el invariante de la interfaz implementada por una clase se aplica por igual a dicha clase. Los invariantes de la clase padre y de la interfaz se añaden a los de la propia clase mediante el operador lógico **AND**. Si en una clase no hay invariante, se considerará que tiene por invariante la cláusula **True**.

Cuando se implementa o bien se redefine un método, se pueden conservar las aserciones originales, pero también se podría:

- Reemplazar la precondición por una más débil.
- Reemplazar la postcondición por una más fuerte.

El primer caso significa ser más generoso que el método original, es decir, aceptar más casos. Esto no puede causar daño alguno al cliente que satisfacía la precondición original antes de la llamada.

El segundo caso implica hacer más cosas de las que se prometía; lo cual tampoco puede causar daño al cliente que se basaba en que la postcondición original se cumplía después de la llamada.

La **regla de herencia de las aserciones** establece que una implementación o redefinición de un método sólo puede reemplazar la precondición original por una igual o más débil y la postcondición original por una igual o más fuerte.

La regla expresa que la nueva versión debe aceptar todas las llamadas que eran aceptadas por el original y debe garantizar al menos lo mismo que garantizaba el original. Esta nueva versión puede -aunque no tiene por qué hacerlo- aceptar más casos o dar garantías más fuertes.

Resumiendo, los casos posibles en la herencia de contratos:

- Las precondiciones son combinadas mediante el operador lógico **OR**.
- Las postcondiciones y los invariantes de clase son combinados mediante el operador lógico **AND**.

Una vez que ya sabemos qué son las precondiciones, las postcondiciones y los invariantes, podemos definir lo que significa que una clase sea correcta. La base de la respuesta ya ha sido estudiada: Una clase, como cualquier otro elemento software, es correcta o incorrecta no por sí misma sino con respecto a una especificación. Introduciendo las precondiciones, postcondiciones e invariantes se tiene una forma de incluir la especificación en el propio texto de la clase. Esto proporciona una base para precisar la corrección: La clase es correcta si y sólo si su implementación, según se da en los cuerpos de los métodos, es consistente con las precondiciones, las postcondiciones y los invariantes.

Existen varias bibliotecas en Java que implementan la Programación por Contrato. No obstante, están obsoletas o son difíciles de configurar.Por ello, vamos a emplear una forma muy simple, pero efectiva, de llevar a término la Programación por Contrato. Tenemos a nuestra disposición una clase final que dispone de métodos sencillos pero eficaces para determinar si una aserción se cumple:

```
package org.jomaveger.lang.dbc;
import org.jomaveger.lang.dbc.exceptions.ContractViolationException;
public final class Contract {

   public static final String PRECONDITION_ERROR = "Precondicion Fallida.";
   public static final String POSTCONDITION_ERROR = "Postcondicion Fallida.";
   public static final String CLASS_INVARIANT_ERROR = "Invariante de Clase Falli-
do.";
   public static final String CHECK_ERROR = "Asercion Fallida.";
   public static void require(boolean condition) {
      if (!condition) {
         throw new ContractViolationException(PRECONDITION_ERROR);
      }
   }

   public static void require(boolean condition, String description) {
      if (!condition) {
         throw new ContractViolationException(PRECONDITION_ERROR + descrip-
tion);
      }
   }

   public static void ensure(boolean condition) {
      if (!condition) {
         throw new ContractViolationException(POSTCONDITION_ERROR);
      }
```

```
    }

    public static void ensure(boolean condition, String description) {
        if (!condition) {
            throw new ContractViolationException(POSTCONDITION_ERROR + descrip-
tion);
        }
    }

    public static void invariant(boolean condition) {
        if (!condition) {
            throw new ContractViolationException(CLASS_INVARIANT_ERROR);
        }
    }

    public static void invariant(boolean condition, String description) {
        if (!condition) {
            throw new ContractViolationException(CLASS_INVARIANT_ERROR + descrip-
tion);
        }
    }

    public static void check(boolean condition) {
        if (!condition) {
            throw new ContractViolationException(CHECK_ERROR);
        }
    }

    public static void check(boolean condition, String description) {
        if (!condition) {
            throw new ContractViolationException(CHECK_ERROR + description);
        }
    }
}
```

Los contratos se escriben en el código fuente Java mediante expresiones lógicas válidas en el lenguaje Java. Se verifica el cumplimiento de los contratos de la siguiente manera:

- Las precondiciones de métodos mediante la invocación del método estático **Contract.require()**.
- Las postcondiciones de métodos mediante la invocación del método estático **Contract.ensure()**.
- Los invariantes de clase mediante la invocación del método estático **Contract.invariant()**.

Se pueden escribir contratos tanto para clases como para interfaces.

Cualquier otro tipo de aserción que se desee verificar se valida a través del método estático **Contract.check()**.

Las precondiciones y las postcondiciones son evaluadas en el contexto del método al que están unidas. En las precondiciones y las postcondiciones, se puede acceder a los valores de los parámetros de los métodos.

3.2 ROBUSTEZ

La robustez es la capacidad de un sistema software de reaccionar adecuadamente ante situaciones excepcionales.

Un sistema software no sólo ha de ser correcto sino también robusto.

En un sistema software, es recomendable emplear un diseño exigente de las precondiciones para la comunicación que tiene lugar entre un método -el proveedor- y otro método -el cliente que lo llama-.

Aunque estudiaremos las pilas como estructuras de datos más adelante, supongamos que tenemos definida una interfaz genérica **IStack< T >**, que define las operaciones que se pueden aplicar a una pila cualquiera, y una clase **LinkedStack< T >** que implementa dicha interfaz. Consideremos, por ejemplo, el método **peek()** de las pilas, que desapila el elemento que se encuentra en la cima de la pila. Este método tiene como precondición que la pila no puede estar vacía, puesto que no tiene sentido intentar eliminar el elemento de la cima de la pila si no hay elemento alguno en ella.

Nótese que un cliente de la clase **LinkedStack** no debe llamar a **peek()** cuando la pila está vacía. La explicación teórica es que el comportamiento de **peek()** en una pila vacía está indefinido por definición, desde el mismo momento en que no se cumple la precondición. La explicación práctica es que el control de aserciones puede no estar activado cuando se ejecuta una aplicación -aunque debería estarlo-, pero sin duda el código debe funcionar de la misma manera tanto si está activado el control de aserciones como si no lo está; además, la gestión de excepciones nunca debe emplearse como una estructura de control condicional. Por eso, no hay que escribir código como éste:

```
IStack<Integer> stack = new LinkedStack<>();
try {
    stack.peek();
} catch (PreconditionError e) {
    // Hacer lo que sea en caso de que la pila esté vacía.
}
```

En lugar del código anterior, habría que escribir código como el siguiente:

```
IStack<Integer> stack = new LinkedStack<>();
if (stack.isEmpty()) {
    // Hacer lo que sea en caso de que la pila esté vacía.
} else {
    stack.peek();
}
```

Por supuesto, puede ocurrir que la comunicación de la que hemos hablado entre el método proveedor y el método cliente no termine con la satisfacción del contrato.

Una llamada a un método tiene éxito si termina su ejecución en un estado en el que satisface el contrato del método. Fracasa, en cambio, si no tiene éxito.

Una excepción es un suceso en tiempo de ejcución que puede causar que un método fracase.

Una excepción puede ocurrir durante la ejecución de un método **m** como resultado de alguna de las situaciones siguientes:

1. Intentar hacer una llamada **a.f(...)** y encontrar que **a** es nulo.
2. Ejecutar una operación que produce una condición anormal detectada por el hardware o el sistema operativo.
3. Invocar a un método que fracasa.
4. Encontrar que la precondición **m** no se cumple al tratar de entrar en **m**.
5. Encontrar que la postcondición de **m** no se cumple al salir de **m**.
6. Encontrar que el invariante de **m** no se cumple al entrar o salir de **m**.
7. Ejecutar una instrucción que pida explícitamente elevar o lanzar una excepción.

El sistema de control de aserciones que hemos diseñado define la siguiente excepción no comprobada:

- **org.jomaveger.lang.dbc.exceptions.ContractViolationException**, que se lanza cuando se detecta que un contrato se ha violado porque una aserción no se cumple.

```
package org.jomaveger.lang.dbc.exceptions;
public class ContractViolationException extends RuntimeException {
   public ContractViolationException(String s) {
      super(s);
   }
}
```

El caso **2** es resultado de señales que el sistema operativo envía a una aplicación cuando detecta un suceso anormal, tal como un fallo en una operación aritmética -por ejemplo, desbordamiento-, un intento de asignar memoria cuando no hay memoria disponible, un intento de acceder a un fichero que no existe en el disco duro o un intento de obtener datos de la red cuando el cable de red está desconectado.

El caso **3** surge cuando un método fracasa, como resultado de una excepción ocurrida durante su propia ejecución y de la cual no fue capaz de recuperarse. Como resultado, podemos concluir lo siguiente:

- Un fracaso de un método causa una excepción en quien lo llama.

Los casos **4, 5 y 6** sólo pueden ocurrir si está activado el control en tiempo de ejecución de las aserciones.

El caso **7** supone que el software puede incluir llamadas de la forma **throw**, cuyo único objetivo es elevar o lanzar una excepción.

Una llamada a un método fracasa si y sólo si ocurre una excepción durante la ejecución del mismo y, al mismo tiempo, no se puede recuperar de dicha excepción.

Cuando se desarrollan componentes software que tratan con fuentes externas de error como, por ejemplo, interfaces gráficas de usuario, módulos de comunicación de red o sistemas de ficheros, no hay forma de estar seguro de que los usuarios de un sistema software introducirán siempre valores correctos, que la conexión de red nunca se caerá o bien que los ficheros siempre van a existir y a estar disponibles para ser leídos. En estos casos, es recomendable usar un diseño tolerante de las precondiciones.

Ciertamente, no tiene sentido establecer como precondición que *el fichero existe y puede ser abierto para lectura*. ¿Cómo podría el cliente verificar esta precondición? Se podría argumentar que el cliente podría verificar que el fichero existe y, de ser así, intentar abrir el fichero para lectura. El problema es, no obstante, no la posibilidad de comprobar la precondición, sino el hecho de que durante la diferencia de tiempo entre la comprobación del cliente y la llamada al método del proveedor, el fichero puede haber sido borrado o bloqueado por otro proceso.

Mediante el diseño tolerante de las precondiciones, es el método proveedor -y no el cliente- el que comprobará si el fichero existe y puede ser abierto para lectura; de no ser así, lanzará una excepción.

Como norma, un bloque **catch** debe garantizar siempre que el método deja su objeto en un estado consistente, es decir, en un estado en el que se cumple el invariante de la clase.

Ahora bien, ¿qué tipo de excepciones debemos lanzar? ¿Comprobadas o no comprobadas? La polémica entre ambos tipos de excepciones existe desde hace mucho tiempo. De hecho, las excepciones comprobadas presentan una serie de problemas, tanto teóricos como prácticos:

- En cualquier código de una aplicación real, la mayoría de bloques **catch** de las excepciones comprobadas son un sin sentido, dado que se limitan a escribir un mensaje de error o bien a relanzar la excepción que más tarde será reportada como un error.

- En la mayoría de los casos, cuando una excepción comprobada es lanzada, no hay posibilidad de recuperarse. Podemos mostrar un mensaje de error al usuario y registrar el problema en el fichero de trazas de la aplicación de modo que nos aseguremos que esa excepción no ocurre de nuevo. Dado que la mayoría de las excepciones son errores en nuestro código y todo lo que hacemos es registrar el problema en un fichero de trazas, no tiene sentido estar obligados a relanzar las excepciones una y otra vez.

- Es fácil ignorar una excepción comprobada relanzándola como una instancia de una excepción no comprobada.

- Es fácil ignorar una excepción comprobada silenciándola, es decir, escribiendo un bloque **catch** vacío.

- Es muy fácil caer en la tentación de usar las excepciones comprobadas como si fueran simplemente valores de retorno alternativos por los cuales el cliente tuviera que preguntar al proveedor.

Por todo ello, como recomendación de diseño, es mejor crear y lanzar excepciones no comprobadas.

Algunas recomendaciones en el empleo de excepciones serían las siguientes:

- No debemos silenciar el tratamiento de una excepción; es decir, no debemos crear un manejador de excepciones vacío. Si no sabemos cómo tratar una determinada excepción, debe propagarse al cliente; en este

caso, si va a ser propagada y se trata de una excepción comprobada, será relanzada como una no comprobada que conceptualmente sea de más alto nivel.

- Si el tratamiento de la excepción consiste en registrar y notificar el error -lo que ocurre en la mayoría de casos-, entonces conviene primero anotarlo en el fichero de trazas de la aplicación y después notificarlo al usuario final a través de la interfaz de usuario.

- Al crear una excepción con el fin de lanzarla, es conveniente establecer el mensaje de error personalizado que explica por qué se va a lanzar esa excepción. Tal mensaje de error puede ser consultado a través del método **getMessage()** de la clase **Throwable**.

- El método **printStackTrace()** de la clase **Throwable** es útil para depurar porque escribe en la consola estándar de salida la traza con la pila de llamadas que ha provacado una excepción. Es posible tener también dicha traza almacenada en una cadena de caracteres, por ejemplo para guardarla en un fichero de trazas:

```
public static String getStackTrace(Throwable throwable) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    throwable.printStackTrace(pw);
    return sw.toString();
}
```

- Existe un truco muy útil que se puede utilizar en caso de necesitar propagar una excepción comprobada. Si no se desea relanzarla como una no comprobada que conceptualmente sea de más alto nivel, pero tampoco se quiere declarar en la cabecera del método, he aquí otra opción:

```
public static RuntimeException softenException(Exception e) {
    return checkednessRemover(e);
}
private static <T extends Exception> T checkednessRemover(Exception e)
throws T {
    throw (T) e;
}
```

En primer lugar, la derivación genérica asocia el parámetro genérico formal **T** con **RuntimeException**, ya que es el tipo de retorno del método **softenException()**. Esto significa que la expresión **throws T** se convierte en **throws RuntimeException**,

que el compilador interpreta como si no se lanzaran excepciones -ya que **RuntimeException** es la clase padre de todas las excepciones no comprobadas-.

No obstante, la sentencia **throw (T)e;** teóricamente debería evaluar a **throw (RuntimeException)e;**. Si **e** fuera, por ejemplo, **NoSuchFileException**, lo lógico sería esperar que esa sentencia resultara en una excepción **ClassCastException**. Pero, por la manera en la que trabajan los genéricos en Java, la información de tipo es eliminada por el compilador. Por tanto, el código de bytes generado para la máquina virtual de Java realmente es **throw (Exception)e;**, que es correcto.

Este truco muestra que las excepciones comprobadas son puramente una característica del compilador. No existe una verificación en tiempo de ejecución de excepciones comprobadas.

La siguiente clase engloba todas estas utilidades:

```
package org.jomaveger.lang.exceptions;
import java.io.PrintWriter;
import java.io.StringWriter;
public final class ExceptionUtils {
   private ExceptionUtils() {
   }

   public static String getStackTrace(Throwable throwable) {
      StringWriter sw = new StringWriter();
      PrintWriter pw = new PrintWriter(sw);
      throwable.printStackTrace(pw);
      return sw.toString();
   }

   public static String getExpandedMessage(Throwable throwable) {
      StringBuilder string = new StringBuilder();
      string.append(throwable.getClass().getName() + "[");
      string.append(throwable.getMessage());
      string.append("]");
      string.append("\n");
      string.append(ExceptionUtils.getStackTrace(throwable));
      return string.toString();
   }

   public static RuntimeException softenException(Exception e) {
       return checkednessRemover(e);
   }

   private static <T extends Exception> T checkednessRemover(Exception e) throws
T {
       throw (T) e;
   }
}
```

3.3 PRUEBAS UNITARIAS

La realización de las pruebas es una tarea fundamental del desarrollo de cualquier sistema software. Las pruebas nos ayudan, como desarrolladores, a crear código de calidad y sencillo de mantener. Aunque es cierto que esta tarea puede ralentizar en cierto modo el desarrollo, nos aseguramos un código donde los posibles errores se habrán reducido a prácticamente cero antes de la puesta en producción.

Otra característica importante es que nos permiten evolucionar nuestras aplicaciones de forma segura. Si las pruebas pasan con éxito al modificar una clase existente o añadir una nueva, ofrecen la certeza de no haber roto ninguna funcionalidad; en caso contrario, nos avisan de que nuestros cambios han alterado el funcionamiento de la aplicación de manera inesperada.

Existen diferentes tipos de pruebas:

- Una **prueba unitaria** es una sección de código software automatizado que invoca a una unidad de trabajo del sistema software para después verificar una serie de condiciones sobre el comportamiento de dicha unidad de trabajo.

  Una **unidad de trabajo** es un caso de uso lógico-funcional del sistema software, considerado de manera aislada, que puede ser invocado mediante una interfaz pública. Una unidad de trabajo puede corresponderse con un único método, o bien con una clase completa o incluso con múltiples clases que trabajan juntas para lograr un único propósito lógico que puede ser verificado.

- Una **prueba de integración** es una sección de código software automatizado que verifica que dos o más unidades de trabajo del sistema software están correctamente integradas. Por ejemplo, se podría probar el uso de una biblioteca externa dentro de uno de nuestros métodos.

- Una **prueba funcional** es una sección de código software automatizado que verifica el sistema software completo desde el punto de vista del usuario final.

Nos vamos a concentrar en las pruebas unitarias. Para realizarlas, vamos a utilizar la biblioteca [JUnit](https://junit.org/junit4/).

Para que las pruebas unitarias sean útiles y efectivas, deben cumplir una serie de propiedades cuyas iniciales en inglés forman la palabra **FIRST**.

- **F** de **Fast**: Una prueba unitaria debe ser rápida, ejecutarse rápidamente.

- **I** de **Isolated**: Una prueba unitaria debe estar aislada y tener una única razón para fallar. Por tanto, debe ser independiente de factores externos y también de cualquier otra prueba. Es decir, la prueba no debe hablar con base de datos alguna, ni debe utilizar la red, tampoco el sistema de ficheros ni debe requerir modificación alguna del entorno para ser ejecutada. Si hiciera falta acceder a una base de datos, por ejemplo, se deben utilizar dobles de prueba para lograr el aislamiento necesario.
- **R** de **Repeatable**: Una prueba unitaria debe ser repetible y se deben obtener los mismos resultados cada vez que se ejecuta. Así, por ejemplo, no se admiten números aleatorios en pruebas unitarias.
- **S** de **Self-Verifying**: Una prueba unitaria debe ser verificable por sí misma, pasa o falla inequívocamente.
- **T** de **Timely**: Una prueba unitaria debe ser oportuna. No se debe esperar a tener el código de producción terminado para empezar a escribir las pruebas unitarias.

Hay quienes enfrentan la programación por contrato a las pruebas unitarias, cuando realmente se trata de herramientas complementarias:

- Los contratos capturan la semántica general de un método, mientras que las pruebas unitarias verifican una ruta de ejecución específica.
- Los contratos están escritos en el código de producción, mientras que las pruebas unitarias dan un ejemplo de uso correcto en código que no es de producción.
- Los contratos son impuestos durante la ejecución viva y poco fiable del código, mientras que las pruebas unitarias ejercitan el código en un entorno de pruebas seguro. Por tanto, es recomendable dejar los contratos activos, incluso en producción.
- La programación por contrato previene los defectos en el código. Las pruebas unitarias detectan los defectos del código.

El concepto fundamental en JUnit es el caso de prueba.

Un **caso de prueba** es una clase que dispone de métodos para probar una unidad de trabajo.

Normalmente, una unidad de trabajo se corresponde con una clase. Por tanto, para cada clase que quisiéramos probar, definiríamos su correspondiente clase de caso de prueba.

A la hora de implementar las pruebas con JUnit, deberemos seguir una serie de buenas prácticas que se detallan a continuación:

- La clase de prueba se llamará igual que la clase a probar, pero con el sufijo **Test**. Por ejemplo, si queremos probar la clase **MiClase**, la clase de prueba se llamará **MiClaseTest**.
- La clase de prueba se ubicará en el mismo paquete en el que estaba la clase probada. Si **MiClase** está en el paquete **es.jomaveger.examples.chapter3**, **MiClaseTest** pertenecerá e ese mismo paquete. De esta forma, nos aseguramos tener acceso a todos los miembros de la clase a probar que tienen visibilidad protegida y de paquete.
- Mezclar clases reales de la aplicación con clases que sólo nos servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no queremos renunciar a poner la clase de prueba en el mismo paquete que la clase probada. Para solucionar este problema, lo que se hará es crear las clases de prueba en un directorio de fuentes diferente. Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado **main** dentro de la carpeta **src**, los fuentes de pruebas irán en un directorio dentro de **src**.
- Los métodos de prueba (aquellos anotados con la **@Test** de JUnit) tendrán como nombre el mismo que el del método probado, pero con prefijo **test** y con sufijo una descripción de lo que se está probando.
- Aunque dentro de un método de prueba podemos poner tantos **assert** como queramos, es recomendable crear un método de prueba diferente por cada caso de prueba que tengamos. Por ello, podemos tener varios métodos de prueba dedicados a probar la distinta casuística de un único método.

Una clase de prueba es una clase común en la que los métodos de prueba son marcados con la **@Test**. Esta anotación indica al ejecutor de pruebas unitarias de JUnit que el método representa una prueba unitaria y que, por tanto, debería ser ejecutado. En estos métodos de prueba, llamaremos al método probado pasándole los parámetros de entrada establecidos para cada caso de prueba y comprobaremos si el resultado obtenido es igual al resultado esperado.

Es importante saber que JUnit no garantiza que las pruebas unitarias se ejecutan en un orden determinado. Aunque pueda pensarse que el orden de ejecución de los métodos de prueba sería el mismo con el que han sido declarados, no es así. Por ello, las pruebas unitarias deben ser independientes entre sí.

Para comprobar si el resultado obtenido coincide con el resultado esperado, utilizaremos los métodos estáticos de aseveración **assert** de la biblioteca JUnit, que se encuentran en la clase **org.junit.Assert**. Estos métodos de aseveración nos permiten comprobar en una prueba si una condición se cumple y, por tanto, la prueba pasa o, por el contrario, la condición no se cumple y la prueba falla. En los métodos de aseveración con dos parámetros, el primero siempre es el resultado esperado y el segundo es el resultado real obtenido. Existen multitud de variantes de estos métodos; por ejemplo:

- El método **assertArrayEquals()**, que compara si dos arrays son iguales entre sí. En otras palabras, si los dos arrays contienen el mismo número de elementos y contienen además los mismos elementos y en el mismo orden.
- El método **assertEquals()**, que compara si dos objetos son iguales entre sí.
- Los métodos **assertTrue()** y **assertFalse()**, que comprueban si el valor de una variable o expresión son, respectivamente, **true** o **false**.
- Los métodos **assertNull()** y **assertNotNull()**, que comprueban si el valor de una variable o expresión son, respectivamente, **null** o diferente de **null**.
- Los métodos **assertSame()** y **assertNotSame()**, que comprueban, respectivamente, si dos referencias a objetos apuntan al mismo objeto o no.

En algunos casos de prueba, lo que se espera como salida no es que el método nos devuelva un determinado valor, sino que se produzca una excepción. Con JUnit también se puede comprobar en una prueba que nuestra aplicación lanza las excepciones esperadas. Para ello, se agrega el atributo **expected**, indicando qué excepción se espera, a la anotación **@Test**.

Ya hemos indicado que, en una clase de prueba, JUnit sólo ejecuta los métodos que poseen la anotación **@Test**; otros métodos que haya en la misma clase sin esa anotación no son ejecutados por el ejecutor de pruebas unitarias, a excepción

de cuatro métodos que pueden existir anotados con sendas diferentes anotaciones predefinidas que podemos utilizar si nos resultan útiles:

- El método con la anotación **@Before**, que se ejecuta justo antes de la ejecución de cada uno de los métodos de prueba.
- El método con la anotación **@After**, que se ejecuta justo después de la ejecución de cada uno de los métodos de prueba.
- El método con la anotación **@BeforeClass**, que se ejecuta antes de que se ejecute método de prueba alguno.
- El método con la anotación **@AfterClass**, que se ejecuta después de que se ejecuten todos los métodos de prueba.

Definir los casos de prueba para una unidad de trabajo no es una tarea que se realice al azar o por ciencia infusa. El origen de estos casos de prueba está en los contratos que se han definido para la unidad de trabajo que se desea probar. No debemos olvidar que, en realidad, una prueba no deja de ser un cliente de dicha unidad de trabajo.

3.4 DISEÑO DE ALGORITMOS ITERATIVOS

Aunque parezca lo contrario, es sumamente difícil de desarrollar de forma correcta un bucle. Algunos de los problemas típicos que se presentan cuando se trabaja con instrucciones iterativas son los siguientes:

- Llevar a cabo una iteración de más o de menos.
- Manejar incorrectamente los casos límite, tales como las estructuras vacías; por ejemplo, un bucle puede trabajar bien con un array grande pero fallar cuando el array tiene cero o un único elemento.
- No llegar a terminar en algunos casos: Es lo que se conoce como el síndrome del bucle infinito.

El cálculo de un bucle tiene los siguientes componentes:

- Un invariante del bucle, que no debe confundirse con el invariante de clase para la clase que lo encierra. El invariante del bucle es una aserción que describe todos los estados por los que atraviesa el cálculo realizado por

el bucle, observados justo antes de evaluar la condición de terminación del bucle.

- Una función de cota, que depende de las variables del cuerpo del bucle.

El invariante se satisface antes de la primera iteración, después de cada una de ellas y, en particular, después de la última, es decir, a la terminación del bucle. El invariante del bucle es a la vez precondición y postcondición del bucle, ya que éste modifica el estado pero no las relaciones entre las variables. En general, el invariante del bucle es consecuencia lógica de la precondición; además, el invariante del bucle es una generalización de la postcondición, es decir, incluye a la postcondición como un caso especial.

La función de cota es mayor o igual que cero mientras no se cumpla la condición de salida del bucle. Además, cada ejecución del cuerpo del bucle decrementa la función de cota.

Por ejemplo, queremos diseñar un método que, dado un vector de enteros, nos indique si está ordenado o no de manera creciente. Una primera aproximación a la implementación del método sería escribir una especificación del siguiente modo:

```
public static boolean isSorted(int[] v) {
       Contract.require(v != null);
         ...
         Contract.ensure(isSorted == ArrayUtils.isSorted(v));
       return isSorted;
    }
```

donde **ArrayUtils** es una clase que pertenece a un paquete que contiene una gran cantidad de clases de utilidad, proporcionado por la fundación Apache. Aquí, **ArrayUtils** se utiliza como método auxiliar en la especificación de la condición de corrección del método, no como ayuda en la implementación del mismo.

Parece claro que necesitaremos una variable booleana **isSorted**, iniciada a **true** al comienzo del método, lo que indica que la propiedad de ordenación se cumple. Sólo tiene sentido que sigamos examinando el vector mientras continúe cumpliéndose la propiedad de ordenación.

También necesitamos una variable de tipo entero **i** que tenga la función de índice para recorrer el vector. El diseño del método puede basarse en la idea de recorrer el vector de izquierda a derecha en un bucle, comprobando si los elementos están ordenados de forma no decreciente. Dado que se compara un elemento y su

siguiente, el índice empezará a recorrer el vector desde el elemento que está en la posición cero hasta el elemento que está en la posición **(v.length - 2)**.

La función de cota es una expresión numérica que es siempre mayor o igual que cero, aunque su valor se decremente cada vez que se ejecuta una iteración del bucle. En este ejemplo, la función de cota es la siguiente, puesto que siempre se va a cumplir que:

```
v.length - 1 - i >= 0;
```

Por otro lado, el invariante del bucle, aserción que se va a cumplir después de cada iteración, es simplemente:

```
i >= 0 && (i < v.length) && (isSorted || (v[i-1] > v[i]))
```

Dos apuntes sobre el invariante:

- Se podría pensar que el segundo término debería ser

  ```
  (i < v.length - 1)
  ```

  Sin embargo, esta opción no funciona porque, en la última iteración, el índice **i** se incrementa por última vez, aunque no vaya a ejecutarse el bucle de nuevo, pudiendo alcanzar **i** el valor **v.length - 1**. Por tanto, fortalecer el invariante del bucle de esta manera es contraproducente.

- El invariante del bucle también tiene que cumplirse antes de la primera iteración. Alguien podría objetar que, si **i** está iniciado a cero, la condición **(v[i-1] > v[i])** fallaría necesariamente si se ejecuta antes de haber empezado el bucle. Lo que ocurre es que el operador || en Java se ejecuta en cortocircuito y, como **isSorted** es **true** antes de ejecutarse el bucle, Java ya no evalúa la siguiente condición del operador ||, por lo que no se produciría ningún fallo.

Basándonos en todas estas explicaciones, quedaría el siguiente método:

```
package org.jomaveger.examples.chapter3;
import org.apache.commons.lang3.ArrayUtils;
import org.jomaveger.lang.dbc.Contract;
public class Loop {
   public static boolean isSorted(int[] v) {
```

```
        Contract.require(v != null);

        int i = 0;
        boolean isSorted = true;
        while (i < v.length - 1 && isSorted) {

            Contract.check(v.length - 1 - i >= 0);

            isSorted = v[i] <= v[i+1];
            i++;

            Contract.check(i >= 0 && (i < v.length) && (isSorted || (v[i-1] >
v[i])));
        }


        Contract.ensure(isSorted == ArrayUtils.isSorted(v));
        return isSorted;
    }
}
```

La clase de prueba para este método sería la siguiente:

```
package org.jomaveger.examples.chapter3;
import static org.junit.Assert.*;
import org.jomaveger.lang.dbc.exceptions.ContractViolationException;
import org.junit.Test;
public class LoopTest {
   @Test
   public void testIsSortedTrue() {
      int[] array = {5, 7, 9, 13, 15};
      assertTrue(Loop.isSorted(array));
   }

   @Test
   public void testIsSortedFalse() {
      int[] array = {5, 7, 25, 13, 15};
      assertFalse(Loop.isSorted(array));
   }

   @Test(expected = ContractViolationException.class)
   public void testIsSortedPreconditionError() {
      int[] array = null;
      assertTrue(Loop.isSorted(array));
   }
}
```

Sea otro ejemplo en el que queremos diseñar un método tal que, dado un vector de enteros y un número entero, nos devuelva el índice en el vector resultado de aplicar en el mismo la búsqueda binaria de dicho número entero. Seguimos operando en la misma clase **Loop** y partimos de la siguiente especificación:

```
public static int binarySearch(int[] v, int e) {
       Contract.require(v != null && Loop.isSorted(v));

       int k;
         ...
         Contract.ensure((k >= -1) && (k <= v.length -1) && checkOrderingPost(v,
e, k));
       return k;
   }
```

Nótese que la precondición exige que el vector esté ordenado.

Asimismo, la postcondición permite devolver el valor -1, que no corresponde a ninguna posición del vector; es decir, la postcondición garantiza que las posiciones estrictamente superiores a **k**, si existen, no contienen el valor **e**. Esta afirmación se cumple para todo el vector en el caso de que **k** tenga el valor -1. Por tanto, el método no contesta a la pregunta de si el elemento está o no en el vector. En su lugar, si el elemento está en el vector, el resultado **k** es el índice de la posición del vector en la que se encuentra dicho elemento; ahora bien, si el elemento no está en el vector, el resultado **k + 1** marca la posición en la que debería estar el elemento si apareciera en el vector o, dicho de otra forma, el punto en que habría que hacerle sitio para insertarlo en él manteniendo la ordenación. Estas ideas se reúnen en el método **checkOrderingPost()** que se emplea para especificar la postcondición:

```
private static boolean checkOrderingPost(int[] v, int e, int k) {
   boolean ordering = true;
   int j = 0;
   while (j <= k && ordering) {
      ordering = v[j] <= e;
      j++;
   }
   j = k + 1;
   ordering = true;
   while (j < v.length && ordering) {
      ordering = e < v[j];
      j++;
   }
   return ordering;
}
```

Para escribir las expresiones booleanas que conforman las aserciones contamos con la posibilidad de incluir llamadas a métodos auxiliares. No obstante, hay un riesgo. Tan pronto como se permiten métodos en las aserciones, se introducen elementos potencialmente imperativos en un mundo esencialmente declarativo y se corre el riesgo de modificar accidentalmente el estado del objeto. Por ello, cualquier método que se utilice en una aserción no debe causar ningún cambio permanente en el estado del objeto.

Un método auxiliar que sólo se usa en la construcción de aserciones carece a su vez de ellas. No tiene sentido verificar a los guardias a la vez que están inspeccionando las credenciales de los visitantes. Cualquier comprobación se llevará a cabo con anterioridad.

Un método que se emplea únicamente en postcondiciones o el invariante no va a ser usado directamente por el cliente de la clase, por lo que su ámbito es privado.

Dicho esto, pasemos al desarrollo.

Para generalizar o debilitar la postcondición, de modo que obtengamos un invariante de bucle, incluiremos en ella dos índices: Uno de ellos será virtualmente idéntico a **k** pero de nombre **i**, y el otro aparecerá en sustitución de la expresión **k + 1** pero de nombre **d**. El dominio del primero será el mismo que el de **k**, y el dominio del segundo el mismo que el de **k + 1**.

Cuando termine de ejecutarse el bucle, debe ser posible que se cumpla la postcondición, por lo que exigimos que se cumpla:

```
i + 1 == d
```

Por ello, elegimos como condición de ejecución del bucle

```
i + 1 != d
```

Esto implica que en el invariante se cumple también la condición:

```
i + 1 <= d
```

Lo que nos permite elegir como función de cota la expresión

```
d - i
```

Para decidir qué posición del vector evaluar, el algoritmo de búsqueda binaria utiliza un valor próximo al punto medio:

```
(i + d) / 2
```

El método de búsqueda binaria quedaría finalmente de la siguiente forma:

```
public static int binarySearch(int[] v, int e) {
   Contract.require(v != null && Loop.isSorted(v));

   int k;
   int i = -1;
   int d = v.length;

   while (i + 1 != d) {

      Contract.check(d - i >= 0);

      int m = (i + d) / 2;

      if (v[m] <= e) i = m;
      else d = m;

      Contract.check((i >= -1) && (i <= v.length -1) && (d >= 0) && (d <=
v.length)
         && (i + 1 <= d) && checkOrderingInv(v, e, i, d));
   }

   k = i;
   Contract.ensure((k >= -1) && (k <= v.length -1) && checkOrderingPost(v, e,
k));
   return k;
}
```

Donde el método auxiliar **checkOrderingInv()** es la versión de **checkOrderingPost()** ajustada a **i** y **d** según el debilitamiento realizado de la postcondición:

```
private static boolean checkOrderingInv(int[] v, int e, int i, int d) {
   boolean ordering = true;
   int j = 0;
   while (j <= i && ordering) {
      ordering = v[j] <= e;
      j++;
   }
   j = d;
   ordering = true;
   while (j < v.length && ordering) {
      ordering = e < v[j];
      j++;
   }
   return ordering;
}
```

Las pruebas realizadas de la búsqueda binaria se han incluido también en la clase **LoopTest**:

```
@Test
public void testBinarySearch1() {
   int[] array = {5, 7, 9, 13, 15};
   assertEquals(0, Loop.binarySearch(array, 5));
}

@Test(expected = ContractViolationException.class)
public void testBinarySearchPreconditionError() {
   int[] array = {5, 7, 25, 13, 15};
   assertEquals(0, Loop.binarySearch(array, 5));
}

@Test
public void testBinarySearch2() {
   int[] array = {5, 7, 9, 13, 15};
   assertEquals(4, Loop.binarySearch(array, 15));
}

@Test
public void testBinarySearch3() {
   int[] array = {5, 7, 9, 13, 15};
   assertEquals(1, Loop.binarySearch(array, 8));
}
```

3.5 DISEÑO DE ALGORITMOS RECURSIVOS

Un método es recursivo si su cuerpo incluye llamadas a sí mismo. Con el objetivo de no invocarse a sí mismo infinitas veces, el diseño de todo método recursivo se basa en realizar un análisis de sus argumentos de entrada para distinguir entre:

- Caso Directo: El argumento de entrada es tal que el resultado puede calcularse directamente de forma sencilla. Un método recursivo puede tener uno o más casos directos.
- Caso Recursivo: El resultado se calcula mediante llamadas recursivas que realiza el método a sí mismo, teniendo en cuenta que se sabe cómo calcular a partir de los argumentos de entrada otros argumentos de entrada más pequeños, y sabemos además cómo calcular el resultado para los argumentos de entrada originales **suponiendo** conocido el resultado para dichos argumentos de entrada más pequeños. Un método recursivo puede tener uno o más casos recursivos.

Cada llamada recursiva de un método recursivo puede acceder a su correspondiente copia de sus variables locales y argumentos de entrada, no pudiendo acceder a los valores de la copia de las llamadas recursivas anteriores. Al final de la

ejecución de la llamada recursiva, se destruyen esos valores temporales; ahora bien, mientras está pendiente de terminar una llamada recursiva, sus llamadas anteriores también están pendientes de terminar.

A diferencia de lo que ocurre con las variables locales y los argumentos de entrada, los atributos están asociados a los objetos y a las clases, y todas las llamadas recursivas actúan sobre esas mismas variables y sus valores.

Un método recursivo tiene recursión simple o lineal si cada caso recursivo realiza exactamente una llamada recursiva.

Un método recursivo simple tiene recursión no final si el resultado de la llamada recursiva se combina en una expresión para dar lugar al resultado final del método recursivo.

Un método recursivo simple tiene recursión final (en inglés *tail recursion*) si el resultado que es devuelto es el resultado de la ejecución de la última llamada recursiva. Los métodos recursivos finales tienen la interesante propiedad de que pueden ser traducidos de manera directa a soluciones iterativas.

Un método recursivo tiene recursión múltiple si, al menos en un caso recursivo, se realizan al menos dos llamadas recursivas.

Un ejemplo clásico de método recursivo simple no final sería el cálculo del factorial de un número entero positivo:

```
package org.jomaveger.examples.chapter3;
import java.math.BigInteger;
public final class Recursion {
    public Recursion() {
    }
    public static BigInteger factorialRecursive(Integer x) {
        Contract.require(x >= 0);
        if (x == 0) {
            return BigInteger.ONE;
        } else {
            return factorialRecursive(x - 1).multiply(BigInteger.valueOf(x));
        }
    }
}
```

Las pruebas para este método serían las siguientes:

```
package org.jomaveger.examples.chapter3;
import static org.junit.Assert.*;
import java.math.BigInteger;
import org.jomaveger.lang.dbc.exceptions.ContractViolationException;
```

```
import org.junit.Test;
public class RecursionTest {
   @Test
   public void testFactorialRecursive0() {
      assertEquals(BigInteger.valueOf(1), Recursion.factorialRecursive(0));
   }

   @Test(expected = ContractViolationException.class)
   public void testFactorialRecursiveMinus1() {
      assertEquals(1, Recursion.factorialRecursive(-1));
   }
   @Test
   public void testFactorialRecursive5() {
      assertEquals(BigInteger.valueOf(120), Recursion.factorialRecursive(5));
   }
}
```

Un ejemplo clásico de método recursivo final sería el cálculo del máximo común divisor de dos números enteros positivos mayores que cero por el método de Euclides:

```
//class Recursion...
public static Integer gcd(Integer x, Integer y) {
    Contract.require(x > 0 && y > 0);
    Integer m = 1;
    if (x == y) {
        m = x;
    }
    else if (x > y) {
        m = gcd(x - y, y);
    }
    else if (x < y) {
        m = gcd(x, y - x);
    }
    return m;
}
```

Posibles pruebas para este método serían las siguientes:

```
//class RecursionTest...
@Test
public void testGcd() {
   assertEquals(new Integer(6), Recursion.gcd(12, 18));
}

@Test(expected = ContractViolationException.class)
public void testGcdNegative() {
   assertEquals(new Integer(1), Recursion.gcd(-1, 3));
}
```

Un ejemplo clásico de método recursivo múltiple sería el cálculo del número de Fibonacci dentro de la sucesión de Fibonacci de un número entero positivo:

```
//class Recursion...
public static BigInteger fibonacci(Integer x) {
    Contract.require(x >= 0);
    if (x == 0 || x == 1) {
        return BigInteger.valueOf(x);
    }
    return fibonacci(x - 1).add(fibonacci(x - 2));
}
```

Las pruebas para este método serían las siguientes:

```
//class RecursionTest...
@Test
public void testFibonacci() {
   assertEquals(BigInteger.valueOf(55), Recursion.fibonacci(10));
}

@Test(expected = ContractViolationException.class)
public void testFibonacciNegative() {
   assertEquals(new Integer(1), Recursion.fibonacci(-1));
}
```

Podemos estudiar cómo sería una posible versión recursiva de la búsqueda binaria. Así pues, dado un vector **v** de números enteros ordenado crecientemente y un número entero, se ha de indicar si este número se halla en el vector y en qué posición, o bien si no se halla y, en ese caso, en qué posición habría de insertarse para mantener la propiedad de ordenación del vector. Como es lógico, el rango de las posibles posiciones de inserción del número entero se extiende desde cero hasta **v.length**, ya que insertar un elemento quiere decir aumentar en uno los elementos del vector **v**.

Dado que queremos devolver dos elementos de información, creamos la siguiente clase para devolver el resultado del método:

```
package org.jomaveger.examples.chapter3;
public class SearchResult {

   public boolean found;
   public int index;
}
```

Partimos entonces de la siguiente especificación, situada en la clase **Recursion**:

```
public static SearchResult binarySearch(int[] v, int e) {
   Contract.require(Loop.isSorted(v));

   SearchResult result = ...;

   Contract.ensure((!result.found ||
    (result.index >= 0 && (result.index <= v.length - 1) && (e == v[result.
index])))
      && (result.found ||
    (result.index >= 0 && (result.index <= v.length) && checkOrdering(v, e, re-
sult.index))));

   return result;
}
```

Se ha utilizado la implicación lógica en forma disyuntiva en la postcondición, gracias a la siguiente ley de equivalencia lógica:

```
p -> q = !p v q
```

Así, la postcondición dice lo siguiente: Si el número entero buscado ha sido encontrado, entonces el índice resultado marca la posición en el vector en la que se encuentra. A la vez, si el número entero buscado no se ha encontrado, el índice resultado se encuentra entre cero y **v.length**, cumpliéndose que todos los elementos del vector situados antes del índice son menores que el número buscado, y también que el número buscado es menor que todos los elementos del vector situados después del índice resultado. Esta última propiedad se verifica en el método **checkOrdering()**:

```
//class Recursion...
private static boolean checkOrdering(int[] v, int e, int index) {
   boolean ordering = true;
   int j = 0;
   while (j <= index - 1 && ordering) {
      ordering = v[j] < e;
      j++;
   }
   j = index;
   ordering = true;
   while (j < v.length && ordering) {
      ordering = e < v[j];
      j++;
   }
   return ordering;
}
```

Para poder llamar recursivamente al método, hemos de generalizarlo, permitiendo que busque en cualquier sección del vector. Nuestro método quedaría así entonces:

```
//class Recursion...
public static SearchResult binarySearch(int[] v, int e) {
   Contract.require(Loop.isSorted(v));

   SearchResult result = gBinarySearch(v, e, 0, v.length);

   Contract.ensure((!result.found ||
    (result.index >= 0 && (result.index <= v.length - 1) && (e == v[result.
index])))
      && (result.found ||
    (result.index >= 0 && (result.index <= v.length) && checkOrdering(v, e, re-
sult.index))));

   return result;
}
```

Donde **gBinarySearch()** es el método más general implementado recursivamente. La cabecera del método es la siguiente:

```
public static SearchResult gBinarySearch(int[] v, int e, int c, int f);
```

El caso directo consiste en examinar secciones vacías, de modo que **c > f**. El caso recursivo sería, por tanto, cuando la sección del vector que vamos a examinar no está vacía, es decir, **c <= f**. En este último caso, lo que hacemos es evaluar el elemento que se halla en la posición media del vector. Si no es el buscado, ocurrirá que el número que buscamos es mayor o menor que el elemento investigado del vector. Aprovechando la propiedad de que el vector está ordenado, se descarta, en el primer caso, la sección izquierda y la búsqueda prosigue en la mitad derecha; en el segundo caso, se descarta la sección derecha y la búsqued prosigue en la mitad izquierda.

El método recursivo quedaría finalmente del siguiente modo:

```
//class Recursion...
public static SearchResult gBinarySearch(int[] v, int e, int c, int f) {
   SearchResult result = new SearchResult();

   if (c > f) {
      result.found = false;
      result.index = c;
   }
   if (c <= f) {
```

```
        int m = (c + f) / 2;
        if (e < v[m]) return gBinarySearch(v, e, c, m - 1);
        if (e == v[m]) {
            result.found = true;
            result.index = m;
        }
        if (e > v[m]) return gBinarySearch(v, e, m + 1, f);
    }

    return result;
}
```

Las pruebas de este método serían las siguientes:

```
//class RecursionTest
@Test
public void testBinarySearch1() {
    int[] array = {5, 7, 9, 13, 15};
    SearchResult result = Recursion.binarySearch(array, 5);
    assertEquals(0, result.index);
    assertEquals(true, result.found);
}

@Test(expected = ContractViolationException.class)
public void testBinarySearchPreconditionError() {
    int[] array = {5, 7, 25, 13, 15};
    assertEquals(null, Recursion.binarySearch(array, 5));
}

@Test
public void testBinarySearch2() {
    int[] array = {5, 7, 9, 13, 15};
    SearchResult result = Recursion.binarySearch(array, 15);
    assertEquals(4, result.index);
    assertEquals(true, result.found);
}

@Test
public void testBinarySearch3() {
    int[] array = {5, 7, 9, 13, 15};
    SearchResult result = Recursion.binarySearch(array, 8);
    assertEquals(2, result.index);
    assertEquals(false, result.found);
}
```

Si observamos la traza de las llamadas de un método recursivo, podemos comprobar que, para que la ejecución del método pueda recorrer el camino de vuelta, en algún lugar se deberán almacenar los valores de los parámetros de entrada y de las variables locales de las llamadas recursivas anteriores. Este lugar es una zona de memoria conocida como la pila de ejecución. Debido a que en la ejecución de

un método recursivo podemos tener que realizar muchas llamadas hasta alcanzar los casos directos, puede darse la situación de que se llene la pila de ejecución y se produzca un error de desbordamiento de pila (en inglés *Stack Overflow*). Siempre que se desarrolla un método recursivo existe la posibilidad de que, durante su ejecución, pueda llegar a ocurrir un desbordamiento de la pila de ejecución. La ocurrencia o no de este hecho depende finalmente de las características de los datos de entrada: Cuanto más grande sea el tamaño de los datos de entrada, más fácil es que tenga lugar un desbordamiento de la pila.

Por tanto, si en una determinada aplicación se conoce que los datos de entrada son de tamaño moderado, entonces no haría falta optimizar la recursión. En caso de que los datos de entrada tuvieran un tamaño grande, sería conveniente, para evitar sorpresas desagradables, realizar algún tipo de optimización sobre el diseño inicial recursivo, que con frecuencia suele ser más evidente.

Antes comentamos que los métodos recursivos finales tienen la interesante propiedad de que pueden ser traducidos de manera directa a soluciones iterativas. En particular, estas soluciones iterativas son más eficientes. Hay compiladores que incorporan entre sus características la optimización de la recursión final (en inglés conocida como *tail call elimination* o bien *tail call optimization*): Transforman automáticamente un método recursivo final a iterativo. Muchos lenguajes de programación, incluyendo Java, no incluyen esta optimización automática, y es necesario hacerla de manera manual; es decir, el programador debe convertir él mismo el método recursivo final en un método iterativo.

Así, por ejemplo, si necesitamos optimizar el método recursivo final que calcula el máximo común divisor, podemos convertirlo a un método iterativo de una manera directa:

```
//class Recursion...
public static Integer gcdIter(Integer x, Integer y) {
    Integer m = 1;
    while (x != y) {
        if (x > y) {
            x = x - y;
        } else if (x < y) {
            y = y - x;
        }
    }
    m = x;
    return m;
}
```

Si es necesario optimizar un método recursivo simple no final como el factorial, primero es necesario convertirlo en un método recursivo final. Para ello,

se utiliza una técnica de generalización (también llamada *técnica de inmersión*), de forma que se añade al método un nuevo parámetro donde se va acumulando el resultado a medida que se construye. Particularizando este nuevo parámetro al valor 1, se obtiene el comportamiento original del método. Por tanto, la versión recursiva final del factorial sería la siguiente:

```
//class Recursion...
public static BigInteger acuFact(BigInteger a, Integer n) {
    BigInteger r = BigInteger.ONE;
    if (n == 0) {
        r = a;
    } else if (n > 0) {
        r = acuFact(a.multiply(BigInteger.valueOf(n)), n - 1);
    }
    return r;
}
public static BigInteger fact(Integer n) {
    return Recursion.acuFact(BigInteger.ONE, n);
}
```

Para generar la solución iterativa, el parámetro acumulador **a** se convierte en una variable local inicializada a 1:

```
//class Recursion...
public static BigInteger factIter(Integer n) {
    BigInteger r;
    BigInteger a = BigInteger.ONE;
    while (n > 0) {
        a = a.multiply(BigInteger.valueOf(n));
        n = n - 1;
    }
    r = a;
    return r;
}
```

Si es necesario optimizar un método recursivo múltiple, como es el caso del método que calcula el número de Fibonacci, podemos emplear también la técnica de generalización para transformarlo primero en un método recursivo final. Dado que Fibonacci realiza dos llamadas recursivas, necesitamos dos parámetros acumuladores. Por tanto, la versión recursiva final de Fibonacci sería la siguiente:

```
//class Recursion...
public static BigInteger acuFib(BigInteger acc1, BigInteger acc2, BigInteger x)
{
    if (x.equals(BigInteger.ZERO)) {
        return BigInteger.ZERO;
    } else if (x.equals(BigInteger.ONE)) {
```

```
            return acc1.add(acc2);
        } else {
            return acuFib(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE));
        }
    }
    public static BigInteger fib(Integer x) {
        return acuFib(BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(x));
    }
```

Para generar la solución iterativa, los dos parámetros acumuladores se convierten en sendas variables locales:

```
    //class Recursion...
    public static BigInteger fibIter(Integer x) {
        BigInteger r;
        BigInteger acc1 = BigInteger.ONE;
        BigInteger acc2 = BigInteger.ZERO;
        BigInteger temp;
        while (x > 1) {
            temp = acc2;
            acc2 = acc1.add(acc2);
            acc1 = temp;
            x = x - 1;
        }
        r = acc1.add(acc2);
        return r;
    }
```