Paso de código con parametrización de comportamiento

Un problema común en la ingeniería de software es que, independientemente de lo que se haga, **los requisitos del usuario cambian**. Por ejemplo, imaginemos una aplicación para ayudar a un agricultor a gestionar su inventario. El agricultor podría querer una función para encontrar todas las manzanas verdes. Pero al día siguiente podría decir: «*En realidad, también quiero encontrar todas las manzanas que pesen más de 150 g*». Dos días después, el agricultor añade: «*Sería genial poder encontrar todas las manzanas verdes que pesen más de 150 g*». ¿Cómo se pueden gestionar estos requisitos cambiantes? Lo ideal sería minimizar el esfuerzo de ingeniería. Además, las nuevas funcionalidades similares deberían ser fáciles de implementar y mantener a largo plazo.

La *parametrización del comportamiento* es un patrón de desarrollo de software que permite gestionar cambios frecuentes en los requisitos. En resumen, consiste en tomar un bloque de código y ponerlo a disposición sin ejecutarlo. Este bloque de código puede ser llamado posteriormente por otras partes del programa, lo que significa que se puede aplazar su ejecución. Por ejemplo, se podría pasar el bloque de código como argumento a otro método que lo ejecutará más tarde. Como resultado, el comportamiento del método se parametriza en función de ese bloque de código. Por ejemplo, si se procesa una colección, se podría escribir un método que:

- Realice *una acción* para cada elemento de la lista.
- Realice *otra acción* al finalizar el procesamiento de la lista.
- Realice *otra acción* si se produce un error.

A esto se refiere la *parametrización del comportamiento*. He aquí una analogía: tu compañero de piso sabe cómo ir al supermercado y volver a casa. Puedes pedirle que compre una lista de productos como pan, queso y vino. Esto equivale a llamar a un método `goAndBuy` pasándole una lista de productos como argumento. Pero un día estás en la oficina y necesitas que haga algo que nunca ha hecho antes: recoger un paquete en la oficina de correos. Debes darle una lista de instrucciones: ir a la oficina de correos, usar este número de referencia, hablar con el gerente y recoger el paquete. Podrías enviarle la lista de instrucciones por correo electrónico, y cuando la reciba, podrá seguirlas. Ahora has hecho algo un poco más avanzado, equivalente a un método `goAndDo`, que puede ejecutar varios comportamientos nuevos como argumentos.

Comenzaremos este capítulo mostrándote un ejemplo de cómo puedes adaptar tu código para que sea más flexible ante los cambios en los requisitos. Partiendo de este conocimiento, mostraremos cómo usar la parametrización de comportamiento en varios ejemplos prácticos. Por ejemplo, es posible que ya hayas usado el patrón de parametrización de comportamiento, utilizando clases e interfaces existentes en la API de Java para ordenar una `List`, filtrar nombres de archivos o indicarle a un `Thread` que ejecute un bloque de código o incluso que gestione eventos de la interfaz gráfica. Pronto te darás cuenta de que este patrón suele ser muy extenso en Java. Las expresiones lambda en Java 8 y versiones posteriores abordan el problema de la verbosidad. En el capítulo 3, mostraremos cómo construir expresiones lambda, dónde usarlas y cómo puedes hacer que tu código sea más conciso al adoptarlas.

2.1 Adaptación a los requisitos cambiantes

Escribir código que se adapte a los requisitos cambiantes es difícil. Analicemos un ejemplo que iremos mejorando gradualmente, mostrando algunas buenas prácticas para que tu código sea más flexible. En el contexto de una aplicación de inventario agrícola, debes implementar una funcionalidad para filtrar las manzanas *verdes* de una lista. Parece fácil, ¿verdad?

2.1.1 Primer intento: filtrar manzanas verdes

Supongamos, como en el capítulo 1, que disponemos de una enumeración `Color` para representar los diferentes colores de una manzana:

```
enum Color { RED, GREEN }
```

Una primera solución podría ser la siguiente:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();          <-- Una lista acumulativa
    for(Apple apple: inventory){                         para manzanas
        if( GREEN.equals(apple.getColor() ) {     <-- Selecciona únicamente
            result.add(apple);                          manzanas verdes.
        }
    }
    return result;
}
```

La línea resaltada muestra la condición necesaria para seleccionar las manzanas verdes. Podemos asumir que disponemos de una enumeración `Color` con un conjunto de colores, como `GREEN`. Pero ahora el agricultor cambia de opinión y quiere filtrar también las manzanas *red*. ¿Qué podemos hacer? Una solución sencilla sería duplicar el método, renombrarlo como `filterRedApples` y modificar la condición *if* para que coincida con las manzanas rojas. Sin embargo, este enfoque no funciona bien si el agricultor quiere varios colores. Un buen principio es el siguiente: cuando nos encontremos escribiendo código casi repetido, intentemos abstraerlo.

2.1.2 Segundo intento: parametrizar el color

¿Cómo podemos evitamos duplicar la mayor parte del código de `filterGreenApples` para crear `filterRedApples`? Para parametrizar el color y ser más flexible ante estos cambios, puedes añadir un parámetro a tu método:

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
Color color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( apple.getColor().equals(color) ) {
            result.add(apple);
        }
    }
    return result;
}
```

Ahora puedes complacer al granjero e invocar tu método de la siguiente manera:

```
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
…
```

Fácil, ¿verdad? Compliquemos un poco el ejemplo. El agricultor te dice: «*Sería genial poder diferenciar entre manzanas ligeras y pesadas. Las manzanas pesadas suelen pesar más de 150 g*».

Como ingeniero de software, te das cuenta de antemano de que el agricultor podría querer variar el peso. Así que creas el siguiente método para gestionar los distintos pesos mediante un parámetro adicional:

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
int weight) {
    List<Apple> result = new ArrayList<>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ) {
            result.add(apple);
        }
    }
    return result;
}
```

Esta es una buena solución, pero fíjate en que tienes que duplicar gran parte de la implementación para recorrer el inventario y aplicar los criterios de filtrado a cada manzana. Esto resulta algo decepcionante porque incumple el principio DRY (*no te repitas*) de la ingeniería de software. ¿Qué pasa si quieres modificar el recorrido del filtro para mejorar el rendimiento? Ahora tienes que modificar la implementación de *todos* tus métodos en lugar de solo uno. Esto supone un gran esfuerzo de ingeniería.

Podrías combinar el color y el peso en un solo método, llamado `filter`. Pero aun así necesitarías una forma de diferenciar qué atributo quieres usar para filtrar. Podrías añadir una bandera para diferenciar entre consultas de color y peso. (¡Pero nunca lo hagas! Explicaremos por qué en breve).

2.1.3 Tercer intento: filtrar con todos los atributos que se te ocurran

Un intento poco elegante de combinar todos los atributos podría ser el siguiente:

```
public static List<Apple> filterApples(List<Apple> inventory, Color color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( (flag && apple.getColor().equals(color)) ||
             (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

Una forma fea de seleccionar el color o el peso.

Podrías usarlo así (pero es poco elegante):

```
List<Apple> greenApples = filterApples(inventory, GREEN, 0, true);
List<Apple> heavyApples = filterApples(inventory, null, 150, false);
…
```

Esta solución es pésima. Para empezar, el código del cliente es terrible. ¿Qué significan `true` y `false`? Además, esta solución no se adapta bien a los cambios en los requisitos. ¿Qué ocurre si el agricultor solicita filtrar por diferentes atributos de una manzana, como su tamaño, forma, origen, etc.? ¿Y si solicita consultas más complejas que combinen atributos, como manzanas verdes que también sean pesadas? Tendría que usar varios métodos de `filter` duplicados o un método extremadamente complejo. Hasta ahora, ha parametrizado el método `filterApples` *con valores* como una String, un `Integer`, un tipo enumerado o un `boolean`. Esto puede funcionar bien para ciertos problemas bien definidos. Pero en este caso, necesita una mejor manera de indicarle al método *filterApples* los criterios de selección de manzanas. En la siguiente sección, describimos cómo usar la *parametrización de comportamiento* para lograr esa flexibilidad.

2.2 Behavior parameterization

Como viste en la sección anterior, necesitas una mejor manera que agregar muchos parámetros para manejar los requisitos cambiantes. Retrocedamos un poco y busquemos un mejor nivel de abstracción. Una posible solución es modelar tus criterios de selección: estás trabajando con manzanas y devolviendo un valor `boolean` basado en algunos atributos de `Apple`. Por ejemplo, ¿es verde? ¿Pesa más de 150 g? A esto lo llamamos *predicado* (una función que devuelve un valor booleano). Por lo tanto, definamos una interfaz para *modelar los criterios de selección*:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

Ahora puede declarar múltiples implementaciones de `ApplePredicate` para representar diferentes criterios de selección, como se muestra a continuación (y se ilustra en la figura 2.1):

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
```

Seleccionar solo manzanas pesadas

Seleccionar solo manzanas verdes

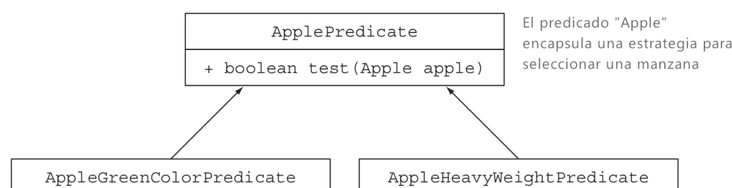

Figura 2.1 Diferentes estrategias para seleccionar una manzana

Estos criterios se pueden interpretar como diferentes comportamientos del método de filtrado. Lo que acabas de hacer está relacionado con el patrón de diseño de estrategia (ver http://en.wikipedia.org/wiki/Strategy_pattern), que permite definir una familia de algoritmos, encapsular cada algoritmo (denominado estrategia) y seleccionar un algoritmo en tiempo de ejecución. En este caso, la familia de algoritmos es `ApplePredicate` y las diferentes estrategias son `AppleHeavyWeightPredicate` y `AppleGreenColorPredicate`.

Pero, ¿cómo se pueden utilizar las diferentes implementaciones de `ApplePredicate`? El método `filterApples` debe aceptar objetos `ApplePredicate` para comprobar una condición en una manzana. Esto es lo que significa la parametrización del comportamiento: la capacidad de indicarle a un método que acepte múltiples comportamientos (o estrategias) como parámetros y que los utilice internamente para lograr diferentes resultados.

Para lograr esto en el ejemplo actual, se añade un parámetro al método `filterApples` para que acepte un objeto `ApplePredicate`. Esto ofrece una gran ventaja para la ingeniería de software: ahora se puede separar la lógica de iterar la colección dentro del método `filterApples` del comportamiento que se desea aplicar a cada elemento de la colección (en este caso, un predicado).

2.2.1 Cuarto intento: filtrado por criterios abstractos

Nuestro método de `filter` modificado, que utiliza un `ApplePredicate`, tiene este aspecto:

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if(p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

El predicado p encapsula la condición para probar una manzana.

PASO DE CÓDIGO/COMPORTAMIENTO

Vale la pena hacer una pequeña pausa para celebrar. Este código es mucho más flexible que nuestro primer intento, ¡y a la vez es fácil de leer y usar! Ahora puedes crear diferentes objetos `ApplePredicate` y pasarlos al método `filterApples`. ¡Flexibilidad total! Por ejemplo, si el agricultor te pide que encuentres todas las manzanas rojas que pesen más de 150 g, solo tienes que crear una clase que implemente el `ApplePredicate` correspondiente. Tu código ahora es lo suficientemente flexible para cualquier cambio en los requisitos relacionados con los atributos de la manzana.

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
        public boolean test(Apple apple){
                return RED.equals(apple.getColor())
                       && apple.getWeight() > 150;
        }
}
List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

¡Has logrado algo genial! El comportamiento del método `filterApples` depende del código que le pasas a través del objeto `ApplePredicate`. ¡Has parametrizado el comportamiento del método `filterApples`!

Ten en cuenta que, en el ejemplo anterior, el único código relevante es la implementación del método de prueba, como se ilustra en la figura 2.2; esto es lo que define los nuevos comportamientos del método `filterApples`. Desafortunadamente, dado que el método `filterApples` solo acepta objetos, debes encapsular ese código dentro de un objeto `ApplePredicate`. Lo que estás haciendo es similar a pasar código en línea, ya que estás pasando una expresión booleana a través de un objeto que implementa el método de prueba. Verás en la sección 2.3 que, mediante el uso de expresiones lambda, puedes pasar directamente la expresión `RED.equals(apple.getColor()) && apple.getWeight() >150` al método `filterApples` sin tener que definir varias clases `ApplePredicate`. Esto elimina la verbosidad innecesaria.

Pasa una estrategia al método `filter`: filtra las manzanas usando la expresión booleana encapsulada en el objeto `ApplePredicate`. Para encapsular este fragmento de código, se incluye una gran cantidad de código repetitivo (en negrita).

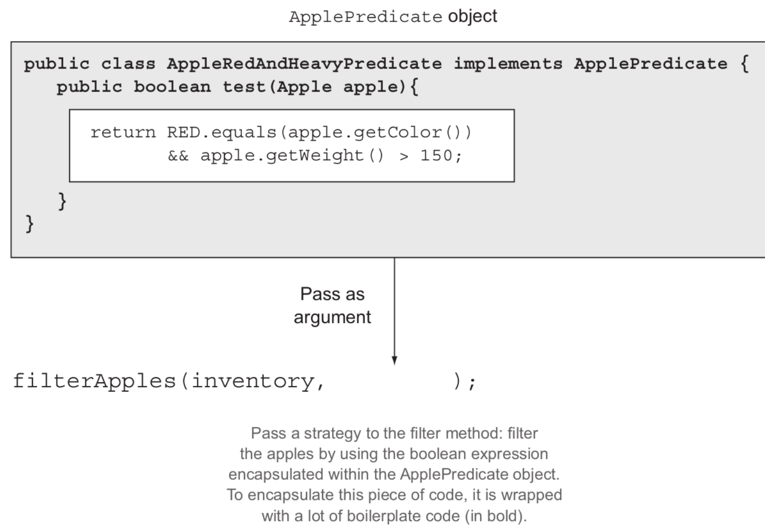

Figura 2.2 Parametrización del comportamiento de `filterApples` y paso de diferentes estrategias de filtrado

MÚLTIPLES COMPORTAMIENTOS, UN PARÁMETRO

Como explicamos anteriormente, la parametrización del comportamiento es muy útil porque permite separar la lógica de iterar la colección a filtrar del comportamiento que se aplica a cada elemento de dicha colección. En consecuencia, puedes reutilizar el mismo método y asignarle diferentes comportamientos para lograr distintos objetivos, como se ilustra en la figura 2.3. Por eso, la *parametrización del comportamiento* es un concepto útil que deberías tener en tu conjunto de herramientas para crear API flexibles.

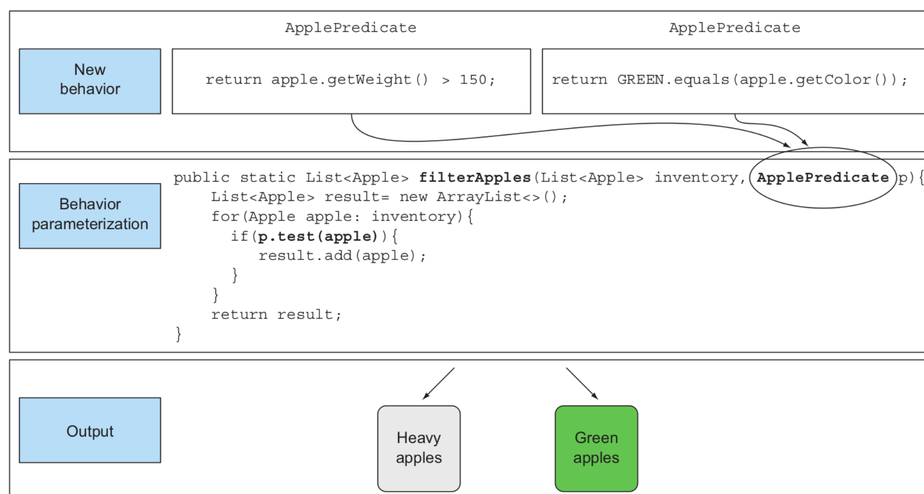

Figura 2.3 Parametrización del comportamiento de ‘filterApples‘ y diferentes estrategias de filtrado.

Para asegurarte de que te sientes cómodo con la parametrización del comportamiento, ¡intenta resolver el cuestionario 2.1!

Cuestionario 2.1: Escribe un método `prettyPrintApple` flexible.

Escribe un método `prettyPrintApple` que reciba una lista de manzanas y que pueda parametrizarse con varias formas de generar una cadena de texto a partir de una manzana (similar a varios métodos `toString` personalizados). Por ejemplo, podrías indicarle a tu método `prettyPrintApple` que imprima solo el peso de cada manzana. Además, podrías indicarle que imprima cada manzana individualmente e indique si es pesada o ligera. La solución es similar a los ejemplos de filtrado que hemos visto hasta ahora. Para ayudarte a empezar, te proporcionamos un esquema básico del método `prettyPrintApple`:

```
public static void prettyPrintApple(List<Apple> inventory, ???) {
    for(Apple apple: inventory) {
        String output = ???.???(apple);
        System.out.println(output);
    }
}
```

Respuesta:

Primero, necesitas una forma de representar un comportamiento que reciba una `Apple` y devuelva una cadena de texto formateada. Hiciste algo similar al crear la interfaz `Apple-Predicate`:

```
public interface AppleFormatter {
    String accept(Apple a);
}
```

Ahora puedes representar múltiples comportamientos de formato implementando la interfaz `Apple-Formatter`:

```
public class AppleFancyFormatter implements AppleFormatter {
    public String accept(Apple apple) {
        String characteristic = apple.getWeight() > 150 ? "heavy" :
            "light";
        return "A " + characteristic +
               " " + apple.getColor() +" apple";
    }
}
public class AppleSimpleFormatter implements AppleFormatter {
    public String accept(Apple apple) {
        return "An apple of " + apple.getWeight() + "g";
    }
}
```

Finalmente, debes indicarle a tu método `prettyPrintApple` que acepte objetos `AppleFormatter` y los use internamente. Puedes hacerlo agregando un parámetro a `pretty-PrintApple`:

```
public static void prettyPrintApple(List<Apple> inventory,
                                    AppleFormatter formatter) {
  for(Apple apple: inventory) {
    String output = formatter.accept(apple);
    System.out.println(output);
  }
}
```

¡Listo! Ahora puedes pasar múltiples comportamientos a tu método `prettyPrintApple`. Para ello, instancia implementaciones de `AppleFormatter` y pásalas como argumentos a `prettyPrintApple`:

```
prettyPrintApple(inventory, new AppleFancyFormatter());
```

Esto producirá una salida similar a esta:

```
A light green apple
A heavy red apple
…
```

O prueba esto:

```
prettyPrintApple(inventory, new AppleSimpleFormatter());
```

Esto producirá una salida similar a esta:

```
An apple of 80g
An apple of 155g
…
```

Has visto que puedes abstraer el comportamiento y hacer que tu código se adapte a los cambios en los requisitos, pero el proceso es engorroso porque necesitas declarar varias clases que solo se instancian una vez. Veamos cómo mejorarlo.

2.3 Abordando la verbosidad

Todos sabemos que una característica o concepto que sea engorroso de usar se evitará. Actualmente, cuando quieres pasar un nuevo comportamiento a tu método `filterApples`, te ves obligado a declarar varias clases que implementan la interfaz `ApplePredicate` y luego instanciar varios objetos `ApplePredicate` que se asignan solo una vez, como se muestra en el siguiente listado que resume lo que has visto hasta ahora. ¡Esto implica mucha verbosidad y es un proceso que consume mucho tiempo!

Listado 2.1 Parametrización del comportamiento: filtrado de manzanas con predicados

```
public class AppleHeavyWeightPredicate implements ApplePredicate {    <- Seleccionar manzanas pesadas
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {    <- Seleccionar manzanas verdes
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
public class FilteringApples {
    public static void main(String...args) {
        List<Apple> inventory = Arrays.asList(new Apple(80, GREEN),
                                              new Apple(155, GREEN),
                                              new Apple(120, RED));
        List<Apple> heavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());    <- Resultados en una lista conteniendo una manzana de 155
        List<Apple> greenApples =
            filterApples(inventory, new AppleGreenColorPredicate());    <- Resultados en una lista conteniendo dos manzanas verdes
    }
    public static List<Apple> filterApples(List<Apple> inventory,
                                           ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for (Apple apple : inventory) {
            if (p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }
}
```

Esto supone una sobrecarga innecesaria. ¿Se puede mejorar? Java cuenta con mecanismos llamados clases *anónimas*, que permiten declarar e instanciar una clase al mismo tiempo. Te permiten mejorar tu código un paso más, haciéndolo un poco más conciso. Pero no son del todo satisfactorios. La sección 2.3.3 anticipa el próximo capítulo con una breve introducción sobre cómo las expresiones lambda pueden hacer que tu código sea más legible.

2.3.1 Clases anónimas

Las *clases anónimas* son similares a las clases locales (una clase definida en un bloque) con las que ya estás familiarizado en Java. Sin embargo, las clases anónimas no tienen nombre. Permiten declarar e instanciar una clase simultáneamente. En resumen, permiten crear implementaciones ad hoc.

2.3.2 Quinto intento: uso de una clase anónima

El siguiente código muestra cómo reescribir el ejemplo de filtrado creando un objeto que implementa `ApplePredicate` mediante una clase anónima:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {    <- Parametriza el comportamiento del método filterApples con una clase anónima.
    public boolean test(Apple apple){
        return RED.equals(apple.getColor());
    }
});
```

Las clases anónimas se utilizan con frecuencia en aplicaciones con interfaz gráfica de usuario (GUI) para crear objetos de manejo de eventos. No queremos revivir recuerdos desagradables de Swing, pero el siguiente es un patrón común que se observa en la práctica (aquí usando la API de JavaFX, una plataforma de interfaz de usuario moderna para Java):

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
});
```

Pero las clases anónimas aún no son suficientes. Primero, suelen ser voluminosos porque ocupan mucho espacio, como se muestra en el código en negrita que utiliza los mismos dos ejemplos anteriores:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {    <-┐
    public boolean test(Apple a){                                            │
        return RED.equals(a.getColor());                         Mucho código
    }                                                               repetitivo
});                                                                          │
button.setOnAction(new EventHandler<ActionEvent>() {                     <-┘
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
```

Segundo, a muchos programadores les resulta confuso usarlos. Por ejemplo, el cuestionario 2.2 muestra un clásico acertijo de Java que suele pillar desprevenidos a la mayoría de los programadores. ¡Inténtalo!

Cuestionario 2.2: **Acertijo de clases anónimas**

¿Cuál será el resultado al ejecutar este código: 4, 5, 6 o 42?

```
public class MeaningOfThis {
    public final int value = 4;
    public void doIt() {
        int value = 6;
        Runnable r = new Runnable() {
                public final int value = 5;
                public void run(){
                    int value = 10;
                    System.out.println(this.value);
                }
            };
            r.run();
        }
        public static void main(String...args) {
            MeaningOfThis m = new MeaningOfThis();
            m.doIt();         <-┐
        }                       │¿Cuál es el resultado de esta
}                               │línea?
```

Respuesta:

La respuesta es `5`, porque *this* se refiere al objeto *Runnable* que lo contiene, no a la clase *MeaningOfThis*.

La verbosidad en general es perjudicial; desincentiva el uso de una característica del lenguaje porque escribir y mantener código extenso lleva mucho tiempo, ¡y no es agradable de leer! Un buen código debe ser fácil de comprender a simple vista. Si bien las clases anónimas abordan en cierta medida la verbosidad asociada con la declaración de múltiples clases concretas para una interfaz, aún resultan insatisfactorias. En el contexto de pasar un fragmento de código simple (por ejemplo, una expresión `boolean` que representa un criterio de selección), todavía es necesario crear un objeto e implementar explícitamente un método para definir un nuevo comportamiento (por ejemplo, el método *test* para *Predicate* o el método *handle* para `EventHandler`).

Idealmente, nos gustaría animar a los programadores a usar el patrón de parametrización de comportamiento, ya que, como acabamos de ver, hace que el código sea más adaptable a los cambios en los requisitos. En el capítulo 3, verás que los diseñadores del lenguaje Java 8 resolvieron este problema introduciendo las expresiones lambda, una forma más concisa de pasar código. Basta de intriga; aquí tienes un breve adelanto de cómo las expresiones lambda pueden ayudarte en tu búsqueda de un código limpio.

2.3.3 Sexto intento: uso de una expresión lambda

El código anterior se puede reescribir de la siguiente manera en Java 8 usando una expresión lambda:

```
List<Apple> result =
  filterApples(inventory, (Apple apple) -> RED.equals(apple.getColor()));
```

Hay que reconocer que este código se ve mucho más limpio que nuestros intentos anteriores. Es genial porque empieza a parecerse mucho más al enunciado del problema. Ya hemos resuelto el problema de la verbosidad. La figura 2.4 resume nuestro progreso hasta ahora.

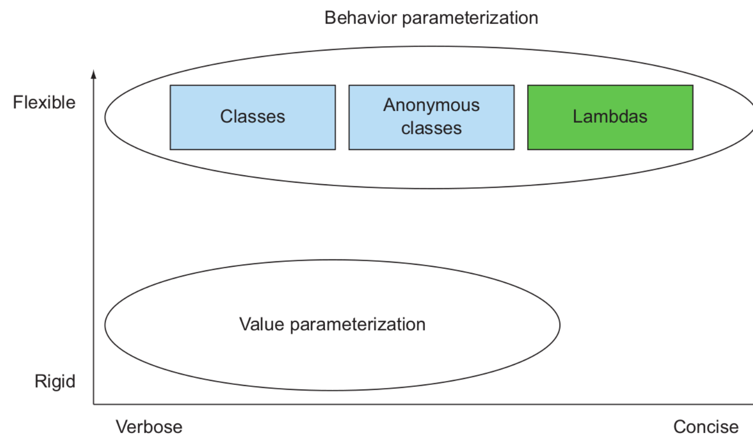

Figura 2.4 Parametrización del comportamiento frente a parametrización del valor

2.3.4 Séptimo intento: abstracción sobre el tipo Lista

Hay un paso más que puedes dar en tu camino hacia la abstracción. Por el momento, el método `filterApples` solo funciona para `Apple`. Pero también puedes abstraerte sobre el tipo Lista para ir más allá del dominio del problema que estás considerando, como se muestra:

```
public interface Predicate<T> {
    boolean test(T t);
}
public static <T> List<T> filter(List<T> list, Predicate<T> p) {   <- Introduce un parámetro de tipo T
    List<T> result = new ArrayList<>();
    for(T e: list) {
        if(p.test(e)) {
            result.add(e);
        }
    }
    return result;
}
```

¡Ahora puedes usar el método `filter` con una `List` de plátanos, naranjas, `Integer` o `String`! Aquí tienes un ejemplo, usando expresiones lambda:

```
List<Apple> redApples =
  filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers =
  filter(numbers, (Integer i) -> i % 2 == 0);
```

¿No es genial? ¡Has logrado encontrar el equilibrio perfecto entre flexibilidad y concisión, algo que no era posible antes de Java 8!

2.4 Ejemplos prácticos

Ahora has visto que la parametrización del comportamiento es un patrón útil para adaptarse fácilmente a los requisitos cambiantes. Este patrón te permite encapsular un comportamiento (un fragmento de código) y parametrizar el comportamiento de los métodos pasando y usando estos comportamientos que creas (por ejemplo, diferentes predicados para una `Apple`). Como mencionamos anteriormente, este enfoque es similar al patrón de diseño Strategy. Es posible que ya lo hayas utilizado en la práctica. Muchos métodos de la API de Java pueden parametrizarse con diferentes comportamientos. Estos métodos se suelen usar junto con clases anónimas. A continuación, mostramos cuatro ejemplos que te ayudarán a comprender mejor cómo pasar código: ordenar con un Comparator, ejecutar un bloque de código con un `Runnable`, devolver un resultado de una tarea usando un `Callable` y gestionar eventos de la interfaz gráfica de usuario (GUI).

2.4.1 Ordenación con un Comparator

Ordenar una colección es una tarea de programación recurrente. Por ejemplo, supongamos que un agricultor le pide que ordene el inventario de manzanas según su peso. O tal vez cambia de opinión y quiere que las ordene por color. ¿Le suena familiar? Sí, necesita una forma de representar y usar diferentes comportamientos de ordenación para adaptarse fácilmente a los cambios en los requisitos.

A partir de Java 8, la clase *List* incluye un método `sort` (también puede usar `Collections.sort`). El comportamiento de `sort` se puede parametrizar usando un objeto `java.util.Comparator`, que tiene la siguiente interfaz:

```
// java.util.Comparator
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Por lo tanto, puede crear diferentes comportamientos para el método `sort` creando una implementación específica de `Comparator`. Por ejemplo, puede usarlo para ordenar el inventario por peso creciente usando una clase anónima:

```
inventory.sort(new Comparator<Apple>() {
public int compare(Apple a1, Apple a2) {
return a1.getWeight().compareTo(a2.getWeight());
}
});
```

Si el agricultor cambia de opinión sobre cómo ordenar las manzanas, puede crear un `Comparator` específico que se ajuste al nuevo requisito y pasarlo al método `sort`. Los detalles internos de cómo ordenar se abstraen. Con una expresión lambda se vería así:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

De nuevo, no se preocupen por esta nueva sintaxis por ahora; el próximo capítulo explica en detalle cómo escribir y usar expresiones lambda.

2.4.2 Ejecución de un bloque de código con Runnable

Los *hilos* de Java permiten ejecutar un bloque de código simultáneamente con el resto del programa. Pero, ¿cómo indicarle a un hilo qué bloque de código debe ejecutar? Varios hilos pueden ejecutar código diferente. Lo que se necesita es una forma de representar un fragmento de código que se ejecutará posteriormente. Hasta Java 8, solo se podían pasar objetos al constructor de `Thread`, por lo que el patrón de uso típico, aunque poco práctico, consistía en pasar una clase anónima con un método ‘`run`‘ que devolvía ‘`void`‘ (sin resultado). Estas clases anónimas implementan la interfaz `Runnable`.

En Java, se puede usar la interfaz `Runnable` para representar un bloque de código que se ejecutará; tenga en cuenta que el código devuelve `void` (sin resultado):

```
// java.lang.Runnable
public interface Runnable {
    void run();
}
```

Esta interfaz permite crear hilos con el comportamiento deseado, como se muestra a continuación:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello world");
    }
});
```

Desde Java 8, se puede usar una expresión lambda, por lo que la llamada a Thread se vería así:

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

2.4.3 Devolución de un resultado mediante Callable

Quizás esté familiarizado con la abstracción `ExecutorService`, introducida en Java 5. La interfaz `ExecutorService` desacopla el envío y la ejecución de tareas. La ventaja de usar `ExecutorService`,

en comparación con el uso de hilos y Runnable, radica en que permite enviar una tarea a un grupo de hilos y almacenar su resultado en un Future. Si esto le resulta desconocido, no se preocupe; retomaremos este tema en capítulos posteriores al analizar la concurrencia con mayor detalle. Por ahora, basta con saber que la interfaz Callable se utiliza para modelar una tarea que devuelve un resultado. Puede considerarse como una versión mejorada de `Runnable`.

```
// java.util.concurrent.Callable
public interface Callable<V> {
    V call();
}
```

Puedes usarlo de la siguiente manera, enviando una tarea a un servicio ejecutor. Aquí se devuelve el nombre del hilo responsable de ejecutar la tarea:

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<String> threadName = executorService.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return Thread.currentThread().getName();
    }
});
```

Usando una expresión lambda, este código se simplifica a lo siguiente:

```
Future<String> threadName = executorService.submit(
                     () -> Thread.currentThread().getName());
```

2.4.4 Manejo de eventos de la interfaz gráfica de usuario (GUI)

Un patrón típico en la programación de GUI es realizar una acción en respuesta a un evento específico, como hacer clic o pasar el cursor sobre un texto. Por ejemplo, si el usuario hace clic en el botón *Enviar*, es posible que desee mostrar una ventana emergente o registrar la acción en un archivo. Nuevamente, necesita una forma de gestionar los cambios; debe poder realizar cualquier respuesta. En JavaFX, puede usar un `EventHandler` para representar una respuesta a un evento pasándolo a `setOnAction`:

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

Aquí, el comportamiento del método `setOnAction` se parametriza con objetos *EventHandler*. Con una expresión lambda, se vería así:

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

Resumen

- La parametrización del comportamiento es la capacidad de un método para tomar múltiples comportamientos diferentes como parámetros y usarlos internamente para lograr distintos comportamientos.
- La parametrización del comportamiento permite que su código sea más adaptable a los requisitos cambiantes y ahorra esfuerzos de ingeniería en el futuro.
- Pasar código es una forma de proporcionar nuevos comportamientos como argumentos a un método. Pero resultaba demasiado extenso antes de Java 8. Las clases anónimas ayudaron un poco antes de Java 8 a eliminar la verbosidad asociada con la declaración de múltiples clases concretas para una interfaz que solo se necesitan una vez.
- La API de Java contiene muchos métodos que se pueden parametrizar con diferentes comportamientos, que incluyen ordenación, subprocesos y manejo de la interfaz gráfica de usuario (GUI).