Patrones de diseño

Clases 12 a 14 del curso 6.170
2, 3 y 10 de octubre de 2001

1. Patrones de diseño

Un patrón de diseño es:

- una solución estándar para un problema común de programación
- una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios
- un proyecto o estructura de implementación que logra una finalidad determinada
- un lenguaje de programación de alto nivel
- una manera más práctica de describir ciertos aspectos de la organización de un programa
- conexiones entre componentes de programas
- la forma de un diagrama de objeto o de un modelo de objeto.

1.1 Ejemplos

Les vamos a presentar algunos ejemplos de patrones de diseño que ya conocen. A cada diseño de proyecto le sigue el problema que trata de resolver, la solución que aporta y las posibles desventajas asociadas. Un desarrollador debe buscar un equilibrio entre las ventajas y las desventajas a la hora de decidir que patrón utilizar. Lo normal es, como observará a menudo en la ciencia computacional y en otros campos, buscar el balance entre flexibilidad y rendimiento.

Encapsulación (ocultación de datos)

Problema: los campos externos pueden ser manipulados directamente a partir del código externo, lo que conduce a violaciones del invariante de representación o a dependencias indeseables que impiden modificaciones en la implementación.
Solución: esconda algunos componentes, permitiendo sólo accesos estilizados al objeto.
Desventajas: la interfaz no puede, eficientemente, facilitar todas las operaciones deseadas. El acceso indirecto puede reducir el rendimiento.

Subclase (herencia)

Problema: abstracciones similares poseen miembros similares (campos y métodos). Esta repetición es tediosa, propensa a errores y un quebradero de cabeza durante el mantenimiento.

Solución: herede miembros por defecto de una superclase, seleccione la implementación correcta a través de resoluciones sobre qué implementación debe ser ejecutada.
Desventajas: el código para una clase está muy dividido, con lo que, potencialmente, se reduce la comprensión. La introducción de resoluciones en tiempo de ejecución introduce *overhead* (procesamiento extra).

Iteración

Problema: los clientes que desean acceder a todos los miembros de una colección deben realizar un transversal especializado para cada estructura de datos, lo que introduce dependencias indeseables que impiden la ampliación del código a otras colecciones.
Solución: las implementaciones, realizadas con conocimiento de la representación, realizan transversales y registran el proceso de iteración. El cliente recibe los resultados a través de una interfaz estándar.
Desventajas: la implementación fija la orden de iteración, esto es, no está controlada en absoluto por el cliente.

Excepciones

Problema: los problemas que ocurren en una parte del código normalmente han de ser manipulados en otro lugar. El código no debe desordenarse con rutinas de manipulación de error, ni con valores de retorno para identificación de errores.
Solución: introducir estructuras de lenguaje para arrojar e interceptar excepciones.
Desventajas: es posible que el código pueda continuar aún desordenado. Puede ser difícil saber dónde será gestionada una excepción. Tal vez induzca a los programadores a utilizar excepciones para controlar el flujo normal de ejecución, que es confuso y por lo general ineficaz.

Estos patrones de diseño en concreto son tan importantes que ya vienen incorporados en Java. Otros vienen incluidos en otros lenguajes, tal vez algunos nunca lleguen a estar incorporados a ningún lenguaje, pero continúan siendo útiles.

1.2 Cuando (no) utilizar patrones de diseño

La primera regla de los patrones de diseño coincide con la primera regla de la optimización: retrasar. Del mismo modo que no es aconsejable optimizar prematuramente, no se deben utilizar patrones de diseño antes de tiempo. Seguramente sea mejor implementar algo primero y asegurarse de que funciona, para luego utilizar el patrón de diseño para mejorar las flaquezas; esto es cierto, sobre todo, cuando aún no ha identificado todos los detalles del proyecto (si comprende totalmente el dominio y el problema, tal vez sea razonable utilizar patrones desde el principio, de igual modo que tiene sentido utilizar los algoritmos más eficientes desde el comienzo en algunas aplicaciones).

Los patrones de diseño pueden incrementar o disminuir la capacidad de comprensión de un diseño o de una implementación, disminuirla al añadir accesos indirectos o aumentar la cantidad de código, disminuirla al regular la modularidad, separar mejor los conceptos y simplificar la descripción. Una vez que aprenda el vocabulario de los patrones de diseño le será más fácil y más rápido comunicarse con otros individuos que también lo conozcan. Por

ejemplo, es más fácil decir “ésta es una instancia del patrón *Visitor*” que “éste es un código que atraviesa una estructura y realiza llamadas de retorno, en tanto que algunos métodos deben estar presentes y son llamados de este modo y en este orden”.

La mayoría de las personas utiliza patrones de diseño cuando perciben un problema en su proyecto —algo que debería resultar sencillo no lo es — o su implementación— como por ejemplo, el rendimiento. Examine un código o un proyecto de esa naturaleza. ¿Cuáles son sus problemas, cuáles son sus compromisos? ¿Qué le gustaría realizar que, en la actualidad, es muy difícil lograr? A continuación, compruebe una referencia de patrón de diseño y busque los patrones que abordan los temas que le preocupan.

La referencia más utilizada en el tema de los patrones de diseño es el llamado libro de la “banda de los cuatro”, *Design Patterns: Elements of Reusable Object-Oriented Software* por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, Addison-Wesley, 1995. Los patrones de diseño son muy populares en la actualidad, por lo que no dejan de aparecer nuevos libros.

1.3 ¿Por qué preocuparse?

Si es usted un programador o un diseñador brillante, o dispone de mucho tiempo para acumular experiencia, tal vez pueda hallar o inventar muchos patrones de diseño. Sin embargo, esta no es una manera eficaz de utilizar su tiempo. Un patrón de diseño es el trabajo de una persona que ya se encontró con el problema anteriormente, intentó muchas soluciones posibles, y escogió y describió una de las mejores. Y esto es algo de lo que debería aprovecharse.

Los patrones de proyecto pueden parecerle abstractos a primera vista o, tal vez, no tenga la seguridad de que se ocupan del problema que le interesa. Comenzará a apreciarlos a medida que construya y modifique sistemas más grandes; tal vez durante su trabajo en el proyecto final Gizmoball.

2. Patrones de creación

2.1 Fábricas

Suponga que está escribiendo una clase para representar carreras de bicicletas. Una carrera se compone de muchas bicicletas (entre otros objetos, quizás).

```
class Race {

        Race createRace() {
                Frame frame1 = new Frame();
                Wheel frontWhee11 = new Wheel();
                Wheel rearWhee11 = new Wheel();
                Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
                Frame frame2 = new Frame();
                Wheel frontWhee12 = new Wheel();
                Wheel rearWhee12 = new Wheel();
                Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
                ...
        }
```

```
        ...
    }
```

Puede especificar la clase ***Race*** para otras carreras de bicicleta.

```
// carrera francesa
class TourDeFrance extends Race {
    Race createRace() {
        Frame frame1 = new RacingFrame();
        Wheel frontWhee11 = new Whee1700c();
        Wheel rearWhee11 = new Whee1700c();
        Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
        Frame frame2 = new RacingFrame();
        Wheel frontWhee12 = new Whee1700c();
        Wheel rearWhee12 = new Whee1700c();
        Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
        ...
    }
    ...
}

//carrera en tierra
class Cyclocross extends Race {

    Race createRace() {
        Frame frame1 = new MountainFrame();
        Wheel frontWhee11 = new Whee127in();
        Wheel rearWhee11 = new Whee127in();
        Bicycle bike1 = new Bicycle(frame1, frontWhee11, rearWhee11);
        Frame frame2 = new MountainFrame();
        Wheel frontWhee12 = new Whee127in();
        Wheel rearWhee12 = new Whee127in();
        Bicycle bike2 = new Bicycle(frame2, frontWhee12, rearWhee12);
        ...
    }
    ...
}
```

En las subclases, *createRace* devuelve un objeto ***Race*** porque el compilador Java impone que los métodos superpuestos tengan valores de retorno idénticos.

Por economía de espacio, los fragmentos de código anteriores omiten muchos otros métodos relacionados con las carreras de bicicleta, algunos de los cuales aparecen en todas las clases, en tanto que otros aparecen sólo en ciertas clases.

La repetición del código es tediosa y, en particular, no fuimos capaces de reutilizar el método *Race.createRace*. (Podemos observar la abstracción de creación de un único objeto ***Bicycle*** a través de una función; utilizaremos esta sin más discusión, como es obvio, por lo menos después de realizar el curso 6.001). Debe existir un método mejor. El patrón de diseño de fábrica nos proporciona la respuesta.

2.1.1 Método de fabrica

Un método de fábrica es el que fabrica objetos de un tipo determinado. Podemos añadir métodos de fábrica a *Race*:

```
class Race {

        Frame createFrame() { return new Frame(); }
        Wheel createWheel() { return new Wheel(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
        return new Bicycle(frame, front, rear);
}
// devuelve una bicicleta completa sin necesidad de ningún argumento
Bicycle completeBicycle() {
        Frame frame = createFrame();
        Wheel frontWheel = createWheel();
        Wheel rearWheel = createWheel();
        return createBicycle(frame, frontWheel, rearWheel);
}
Race createRace() {
        Bicycle bike1 = completeBicycle();
        Bicycle bike2 = completeBicycle();
        ...
        }
}
```

Ahora las subclases pueden reutilizar *createRace* e incluso *completeBicycle* sin ninguna alteración:

```
//carrera francesa
class TourDeFrance extends Race {

        Frame createFrame() { return new RacingFrame(); }
        Wheel createWheel() { return new Whee1700c(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
        }
}

class Cyclocross extends Race {

        Frame createFrame() { return new MountainFrame(); }
        Wheel createWheel() { return new Whee126inch(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
        }
}
```

Los métodos de creación se denominan *Factory methods* (métodos de fábrica).

2.1.2 Objeto de fábrica

Si existen muchos objetos para construir, la inclusión de los métodos de fábrica puede inflar el código haciéndolo difícil de modificar. Las subclases no pueden compartir fácilmente el mismo método de fábrica.

Un objeto de fábrica es un objeto que comprende métodos de fábrica.

```
class BicycleFactory {
        Frame createFrame() { return new Frame(); }
        Wheel createWheel() { return new Wheel(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear){
                return new Bicycle(frame, front, rear);
}
// devuelve una bicicleta completa sin necesidad de ningún argumento
Bicycle completeBicycle() {
        Frame frame = createFrame();
        Wheel frontWheel = createWheel();
        Wheel rearWheel = createWheel();
                return createBicycle(frame, frontWheel, rearWheel);
        }
}
class RacingBicycleFactory {
        Frame createFrame() { return new RacingFrame(); }
        Wheel createWheel() { return new Whee1700c(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
        }
}
class MountainBicycleFactory {
        Frame createFrame() { return new MountainFrame(); }
        Wheel createWheel() { return new Whee126inch(); }
        Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
                return new RacingBicycle(frame, front, rear);
        }
}
```

Los métodos ***Race*** utilizan objetos fábrica.

```
class Race {

BicycleFactory bfactory;

//constructor

Race () {

        bfactory = new BicycleFactory();
}
Race createRace() {
        Bicycle bike1 = bfactory.completeBicycle();
        Bicycle bike2 = bfactory.completeBicycle();
```

```
        ...
        }
}

class TourDeFrance extends Race {
        //constructor
        TourDeFrance() {
        bfactory = new RacingBicycleFactory();
        }
}
class Cyclocross extends Race {
        //constructor
        Cyclocross () {
        bfactory = new MountainBicycleFactory();
        }
}
```

En esta versión del código, el tipo de bicicleta está codificado en cada variedad de carrera. Hay un método más flexible que requiere una alteración en la forma en que los clientes llaman al constructor.

```
class Race {

BicycleFactory bfactory;

// constructor
Race(BicycleFactory bfactory) {
        this.bfactory = bfactory;
}
Race createRace() {
        Bicycle bike1 = bfactory.completeBicycle();
        Bicycle bike2 = bfactory.completeBicycle();
        ...
      }
}

class TourDeFrance extends Race {
        //constructor
        TourDeFrance(BicycleFactory bfactory) {
        this.bfactory = bfactory;
      }
}

class Cyclocross extends Race {
        // constructor
        Cyclocross(BicycleFactory bfactory) {
        this.bfactory = bfactory;
      }
}
```

Éste es el mecanismo más flexible de todos. Con él, un cliente puede controlar tanto el tipo de carrera como la bicicleta utilizada en ella, por ejemplo, por medio de una llamada del tipo

new TourDeFrance(new TricycleFactory())

Una razón por la que los métodos *Factory* son necesarios es la primera debilidad de los constructores de Java: siempre devuelven un objeto del tipo especificado. Ellos nunca pueden devolver un objeto de un subtipo, aunque exista una corrección de tipos (tanto en relación con el mecanismo de subtipos de Java como en relación con el verdadero comportamiento de la práctica de subtipo, tal y como se describirá en la clase 15).

2.1.3 El patrón prototipo

El patrón prototipo ofrece otra manera de construir los objetos de los tipos arbitrarios. En lugar de pasar un objeto ***BicycleFactory***, un objeto ***Bicycle*** es recibido como argumento. Su método *clone* es invocado para crear nuevos objetos ***Bicycle***; estamos construyendo copias del objeto ofrecido.

```
class Bicycle {
	Object clone() { ... }
}
class Frame {
	Object clone() { ... }
}
class Wheel {
	Object clone() { ... }
}
class RacingBicycle {
	Object clone() { ... }
}
class RacingFrame {
	Object clone() { ... }
}
class Whee1700c {
	Object clone() { ... }
}
class MountainBicycle {
	Object clone() { ... }
}
class MountainFrame {
Object clone() { ... }
}
class Whee126inch {
	Object clone() { ... }
}

class Race {

Bicycle bproto;
```

```
//constructor
Race(Bicycle bproto) {
        this.bproto = bproto;
}

Race createRace() {
        Bicycle bike1 = (Bicycle) bproto.clone();
        Bicycle bike2 = (Bicycle) bproto.clone();
        }
}
class TourDeFrance extends Race {
//constructor
        TourDeFrance(Bicycle bproto) {
        this.bproto = bproto;
        }
}

class Cyclocross extends Race {
          //constructor
          Cyclocross(Bicycle bproto) {
          this.bproto = bproto;
          }
}
```

Efectivamente, cada objeto es, en sí mismo, una fábrica especializada en construir objetos iguales a sí mismo. Los prototipos se utilizan de ordinario en lenguajes tipificados dinámicamente como Smalltalk, y menos frecuentemente en lenguajes tipificados estáticamente como C++ y Java.

No obstante, esta técnica tiene un coste: el código para crear objetos de una clase particular debe estar en algún lugar. Los métodos de fábrica colocan el código en métodos de cliente; los objetos de fábrica colocan el código en métodos de un objeto de fábrica y los prototipos colocan el código en métodos *clone*.

2.2 El patrón *Sharing*

Muchos otros patrones de diseño están relacionados con la creación de objetos en el sentido de que influyen sobre los constructores (y necesitan utilizar fábricas) y están relacionados con la estructura en el sentido de que especifican patrones *sharing* entre varios objetos.

2.2.1 El patrón singular

El patrón singular garantiza que, en todo momento, sólo existe un objeto de una clase particular. Tal vez desee utilizarlo para su clase ***Gym*** del proyecto Gym Manager, ya que los métodos de este patrón (como listas de espera para una máquina específica) son los más acertados para la gestión de una única ubicación. Un programa que instancia múltiples copias, probablemente tenga un error, pero la utilización del patrón singular hace que tales errores sean inofensivos.

```
class Gym {
```

```
        private static Gym theGym;
        //constructor
        private Gym() { ... }
        //método fábrica
        public static getGym() {
                if (theGym == null) {
                        theGym = new Gym();
                }
                return theGym;
        }
}
```

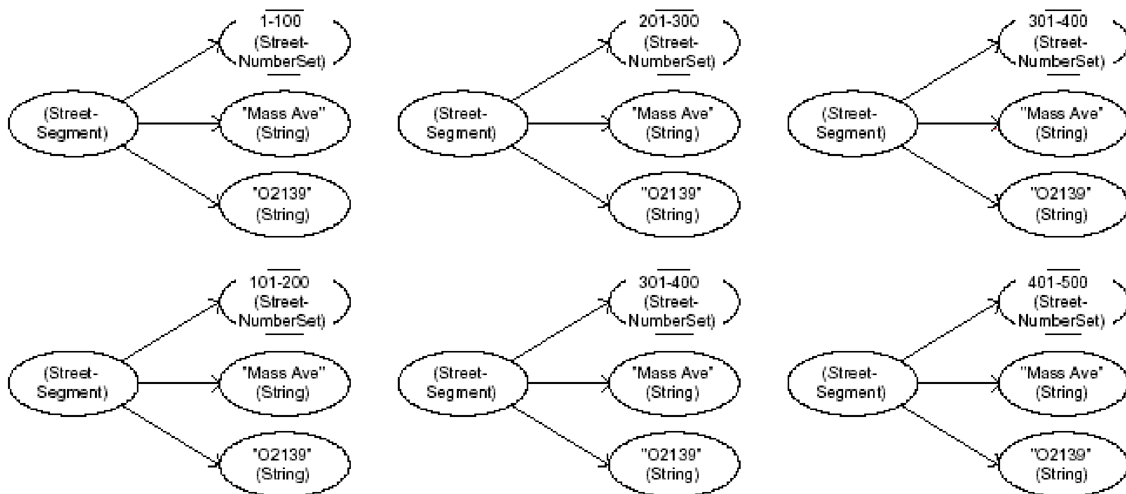
El patrón singular también es útil para objetos grandes y caros que no deben ser instanciados múltiples veces.

La razón por la que debe utilizarse un método de fábrica, en vez de un constructor, es la segunda debilidad de los constructores de Java: siempre devuelven un objeto nuevo, nunca un objeto ya existente.

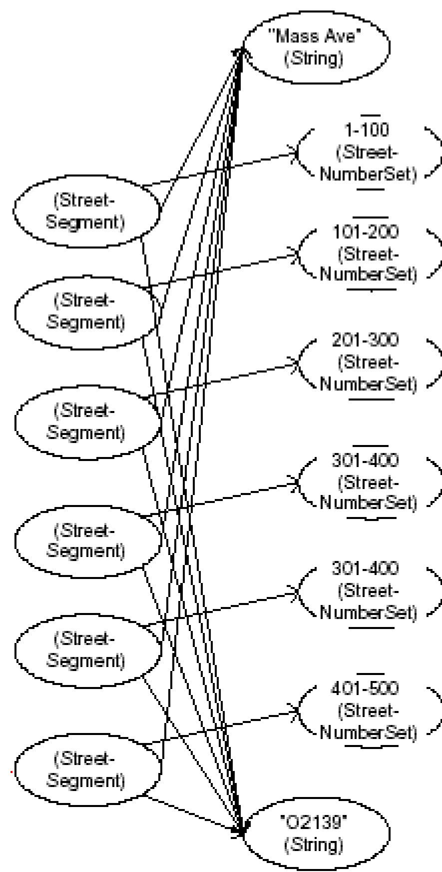
2.2.2 El patrón *Interning*

El patrón de diseño *Interning* reutiliza objetos inexistentes en vez de crear nuevos. Si un cliente solicita un objeto que es igual a uno ya existente, se devuelve el objeto existente. Esta técnica sólo funciona para los objetos inmutables.

Como ejemplo, la clase ***MapQuick*** representa una calle determinada mediante muchas clases ***StreetSegments***. Los objetos ***StreetSegments*** tendrán el mismo nombre de calle y el mismo código postal. Aquí representamos un posible diagrama de objeto (captura) para una parte de la calle.



Esta representación es correcta (por ejemplo, todos los pares de nombres de calle son considerados iguales a través del método *equals*), sin embargo, esto requiere una pérdida innecesaria de espacio. Una mejor configuración del sistema sería:



La diferencia en la utilización del espacio es substancial – tanto que es improbable que usted pueda leer, aunque se trate de una pequeña base de datos, en un objeto ***MapQuick*** en el que no aparezca este *sharing*. Por lo tanto, la implementación del objeto *StreetSegReader* que se le facilitó, realiza esta operación.

El patrón *Interning* posibilita la reutilización de objetos inmutables: en vez de crear un nuevo objeto, se reutiliza una representación canónica. Este patrón requiere una tabla de todos los objetos que han sido creados; si esta tabla contiene objetos iguales al objeto deseado, se devuelve esa versión del objeto existente. Por razones de rendimiento se utiliza una tabla *hash* (de dislocación), que asigna el contenido con los objetos (ya que la igualdad depende sólo de los contenidos).

Aquí se representa un fragmento de código que realiza la operación de *Interning* en *strings* (cadenas) que denominan nombres de segmentos:

```
HashMap segnames = new HashMap();

canonicalName(String n) {
    if (segnames.containsKey(n)) {
        return segnames.get(n);
    } else {
        segnames.put(n, n);
        return n;
    }
```

```
}
```

Las cadenas son un caso especial, pues la mejor representación de una secuencia de caracteres (el contenido) es la propia cadena; y terminamos con una tabla que asigna cadenas a cadenas. Esta estrategia es correcta en general: el código construye una representación no canónica, esta representación no canónica se asigna a la representación canónica y se devuelve esta última representación. No obstante, dependiendo de la cantidad de trabajo realizada por el constructor, puede ser más eficiente no construir la representación no canónica si no es necesario, en cuyo caso la tabla podría realizar la asignación del contenido (no del objeto) con la representación canónica. Por ejemplo, si estuviésemos realizando una operación de *Interning* sobre objetos de una clase denominada ***GeoPoints***, indexaríamos la tabla utilizando los parámetros de latitud y longitud.

El código de ejemplo anterior utiliza el mapa de las cadenas con las propias cadenas, pero no puede utilizar un objeto ***Set*** en lugar de un objeto ***Map***. La razón de esto es que la clase ***Set*** no posee una operación *get*, sólo una operación *contains*. La operación *contains* utiliza *equals* para realizar comparaciones. Por tanto, incluso si *myset.contains(mystring),* esto no significa que *mystring* sea un miembro, idéntico, de *myset*, y no hay modo conveniente de acceder al elemento de *myset* que corresponda (*equals*) a *mystring*.

La noción de tener sólo una versión de una cadena dada es tan importante que está incorporada en Java; *String.intern* devuelve la versión canónica de una cadena.

El texto de Liskov habla del patrón *Interning* en la sección 15.2.1, pero denomina la técnica 'flyweight' (o 'peso-mosca'), que es un término diferente de la terminología estándar en este campo.

2.2.3 *Flyweight*

El patrón Flyweight es una generalización del patrón *Interning*. (En el texto de Liskov, en la sección 15.2.1, titulada "Flyweight", habla del patrón *Interning* y dice que es un caso especial de *Flyweight*). El patrón *Interning* es aplicable sólo cuando un objeto es completamente inmutable. La forma más general del patrón *Flyweight* se puede utilizar cuando la mayor parte (no necesariamente todo) del objeto es inmutable.

Examine el caso de los radios (*spoke*) de la rueda de una bicicleta.

```
class Wheel {
...
FullSpoke[] spokes;
...
}

//más adelante definiremos una versión simplificada
//de esta clase, denominada "Spoke"
        class FullSpoke {
        int length;
        int diameter;
        boolean tapered;
        Metal material;
```

```
        float weight;
        float threading;
        boolean crimped;
        int location;        //localización en la cual el radio se encaja en el cubo y
    en el aro de la rueda
}
```

Normalmente hay de 32 a 36 radios por rueda (hasta 48 en una bicicleta tipo tándem). Sin embargo, existen apenas tres variedades diferentes de radio por bicicleta: uno para la rueda delantera y dos para la rueda trasera (ya que el cubo de la rueda de atrás no está centrado, de forma que son necesarios radios de longitudes diferentes). Preferiríamos asignar sólo tres objetos ***Spoke*** (o ***FullSpoke***) diferentes, en vez de uno por radio de bicicleta. No es aceptable tener un único objeto ***Spoke*** en la clase ***Wheel*** en vez de un *array*, no sólo por la falta de simetría de la rueda trasera, sino también porque podría sustituir un radio (después de una rotura, por ejemplo) por otro que tenga la misma longitud pero difiera en otras características. El patrón *Interning* no se puede utilizar, ya que los objetos no son idénticos: difieren en el campo *location*. En una carrera de bicicletas, con 10.000 bicicletas, es posible que apenas existan unas centenas de radios diferentes, pero millones de instancias de ellos; sería desastroso asignar millones de objetos ***Spoke***. Los objetos ***Spoke*** podrían ser compartidos entre las bicicletas diferentes (dos amigos con bicicletas idénticas podrían compartir el mismo radio número 22 de la rueda delantera), aun así no tendríamos una repartición representativa y, en cualquier evento, es más posible que existan radios semejantes en una bicicleta de los que hay entre varias bicicletas.

El primer paso para la utilización del patrón *Flyweight* es separar los estados intrínsecos de los estados extrínsecos. Los estados intrínsecos se mantienen en el objeto; los estados extrínsecos se mantienen fuera del objeto. Para que el patrón *Interning* sea posible, los estados intrínsecos deben ser inmutables y similares en los objetos.

Creemos una clase ***Spoke*** no dependiente de la propiedad de *location* para los estados intrínsecos:

```
class Spoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
}
```

Para añadir los estados extrínsecos, no es posible hacer lo siguiente:

```
class InstalledSpokeFull extends Spoke {
  int location;
}
```

porque esto es sólo una forma simplificada de la clase ***FullSpoke***; la clase ***InstalledSpokeFull*** consume la misma cantidad de memoria que ***FullSpoke*** ya que tienen los mismos campos. Otra posibilidad es:

```
class InstalledSpokeWrapper {
  Spoke s;
  int location;
}
```

Este es un ejemplo de un *wrapper* (del que trataremos en breve) que ahorra una buena cantidad de espacio porque los objetos ***Spoke*** se pueden compartir entre objetos ***InstalledSpokeWrapper***. No obstante, hay una solución que construye menos memoria.

Observe que la propiedad *location* de un radio dado es igual al valor del índice del objeto ***Spoke*** que representa este radio en el array *Wheel.spokes*:

```
class Wheel {
  ...
  Spoke[] spokes;
  ...
}
```

No es necesario en absoluto almacenar esta información (extrínseca). No obstante, algunas partes del código de cliente (en ***Wheel***) deben cambiarse, ya que los métodos de ***FullSpoke*** que utilizaban el campo *location* deben tener acceso a esta información.

Dada esta versión utilizando la clase ***FullSpoke***:

```
class FullSpoke {
  // tense el radio girando el engrasador el número
  // especificado de veces (turns)
  void tighten(int turns){
  ... location...
  }
}

class Wheel {
  FullSpoke[] spokes;

//el método debería tener el nombre "true",
//pero este nombre de identificador no es bueno
void align() {
  while (la rueda está mal alineada) {
    ... spokes[i].tighten(numturns) ...
            }
        }
}
```

La versión correspondiente con el patrón *Flyweight* es:

```
class Spoke {
  void tighten(int turns, int location) {
    ... location...
  }
}

class Wheel {
  FullSpoke[] spokes;

void align() {
  while (la rueda está mal alineada) {
    ... spokes[i].tighten(numturns, i) ...
    }
  }
}
```

La referencia a una clase ***Spoke*** como patrón *Interning* es mucho menos costosa para el sistema si se compara con una clase ***Spoke*** sin la utilización de ese patrón, se puede decir que esta nueva versión de la clase es más leve, pudiendo ser considerada peso-mosca (flyweight) en contraposición a su versión más pesada; el mismo principio se puede aplicar a la clase ***InstalledSpokeWrapper***, aunque su consumo extra de memoria es como mínimo tres veces mayor (o posiblemente más).

La misma técnica funciona en la clase ***FullSpoke*** si contiene un campo 'rueda' (*wheel*) que se refiere a la rueda en la que el radio representado por la clase está instalado; los métodos de la clase ***Wheel*** pueden, fácilmente, pasar el propio objeto ***Wheel*** para los métodos de la clase ***Spoke***.

El mismo truco funciona si *FullSpoke* contiene un campo *wheel* referido a la rueda en que se instala; los métodos *wheels* pueden convertir esto al método *Spoke*.

Si la clase ***FullSpoke*** también contiene un campo booleano denominado '*broken*' (quebrado), ¿cómo podría representarse? Se trata de otra información extrínseca, que no aparece en el programa explícitamente, de la misma forma que lo hacen las propiedades *location* y *wheel*. Esta información debe estar explícitamente almacenada en la clase ***Wheel***, probablemente como un *array* booleano, paralelo al *array* de objetos ***Spoke***. Esto es un tanto inadecuado —el código está comenzado a ponerse feo— pero es aceptable si la necesidad de ahorro de espacio es crítica. Sin embargo, si hay muchos campos semejantes al campo *broken*, el proyecto debe reconsiderarse.

Recuerde que el patrón *Flyweight* debe utilizarse únicamente después de que un análisis de sistema determine que la economía de espacio de la memoria es crítica para el rendimiento, esto es, la memoria es un cuello de botella (*bottleneck*) del programa. Al introducir estas construcciones en un programa, estamos complicando su código y aumentando las posibilidades de aparición de errores. El patrón *Interning* debe ponerse en práctica sólo en circunstancias muy limitadas.