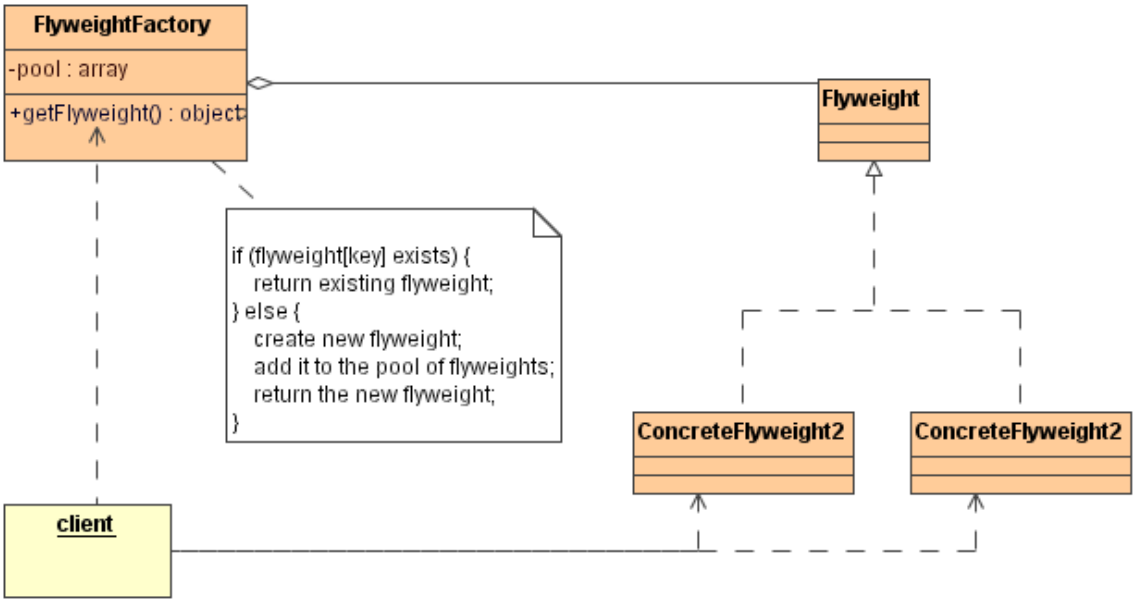
Flyweight

Diagrama de clases e interfaces



Intención

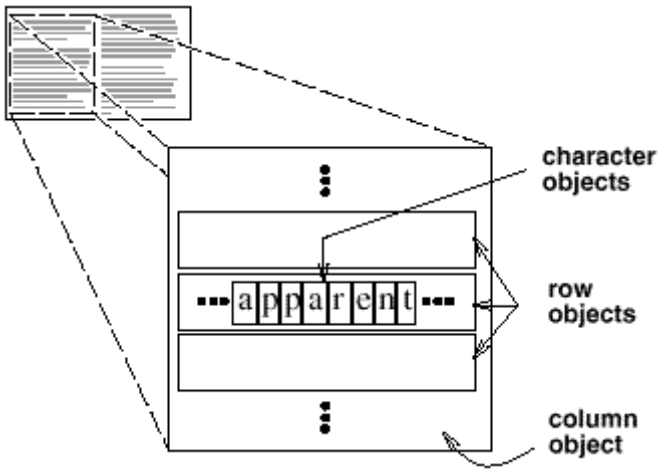
El patrón Flyweight promueve la compartición de recursos cuando es necesario gestionar eficientemente una gran cantidad de objetos. Sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos en los que parte de su información es idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos). La información común se extrae de los objetos para depositarla en una nueva clase que será compartida por el resto. Esta nueva clase recibe el nombre de Flyweight.

Motivación

Desde el punto de vista del Diseño Orientado a Objetos lo adecuado es modelar conceptos, diseñar clases a partir de estos conceptos y así poder crear instancias de estas clases. En otras palabras: encapsular los conceptos como objetos. De esta forma nos beneficiamos de las técnicas que nos proporciona la Orientación a Objetos: polimorfismo, reutilización, etc.

Lamentablemente, hay situaciones en las que modelarlo todo como un objeto sencillamente no es posible por cuestiones de rendimiento.

Supongamos el caso de un procesador de textos. Esta clase de aplicaciones habitualmente usa objetos para representar elementos embebidos como tablas e imágenes. En cambio, no utilizan objetos para representar los caracteres del documento, a pesar de que esto permitiría tratar de manera uniforme (y por tanto, simple) el formateo y el dibujado de tablas, figuras y caracteres. Es más, permitiría que la estructura de objetos de la aplicación imitase a la estructura física del documento:



El inconveniente de utilizar objetos para los caracteres es que tendría un coste prohibitivo para la aplicación. Incluso un documento de tamaño medio necesitará miles de objetos carácter, cuya suma de consumo de memoria produciría un rendimiento inaceptable.

En un escenario como el descrito es donde el patrón Flyweight puede desempeñar un papel excelente. Flyweight modela conceptos o entidades que son demasiado numerosos como para poder representarse individualmente mediante objetos.

En el caso del procesador de textos, se podría crear un objeto flyweight por cada carácter del alfabeto. Cada flyweight almacenaría el código del carácter, mientras que la posición del mismo en el documento (coordenada), su tipo de fuente y formato específico se determinaría por un algoritmo de presentación allí donde el carácter necesitara ser representado.

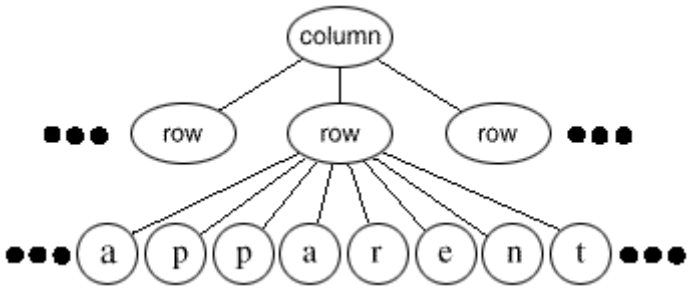
Un flyweight es un objeto compartido, aunque indistinguible de otros objetos no compartidos, que puede utilizarse simultáneamente en múltiples contextos y actuando de manera independiente en cada uno de ellos. Por ejemplo, podríamos tener el carácter ‘B’ formando parte del título del documento y simultáneamente apareciendo en catorce de los veintiséis párrafos de dicho documento.

Esto nos lleva a dos conceptos clave en el patrón Flyweight: estado intrínseco y estado extrínseco. El código de cada carácter es el estado intrínseco y el resto de información es el estado extrínseco.

Estado intrínseco: Es la información que se almacena en el flyweight. Es información que no varía, es fija, independiente del contexto del flyweight. Por tanto, esta información puede ser compartida por otros objetos.

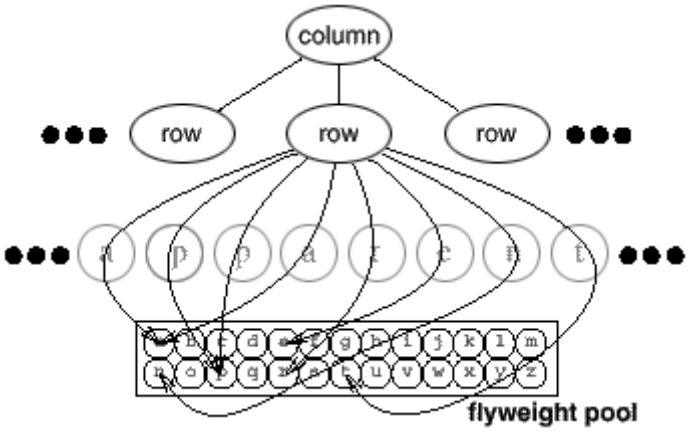
Estado extrínseco: Es la información variable, que depende del contexto del flyweight. Por tanto, no puede ser compartida. Las clases cliente son responsables de proporcionar esta información al Flyweight cuando quieren obtener una referencia a él.

A nivel lógico, tendríamos un objeto por cada aparición de un carácter en el documento:



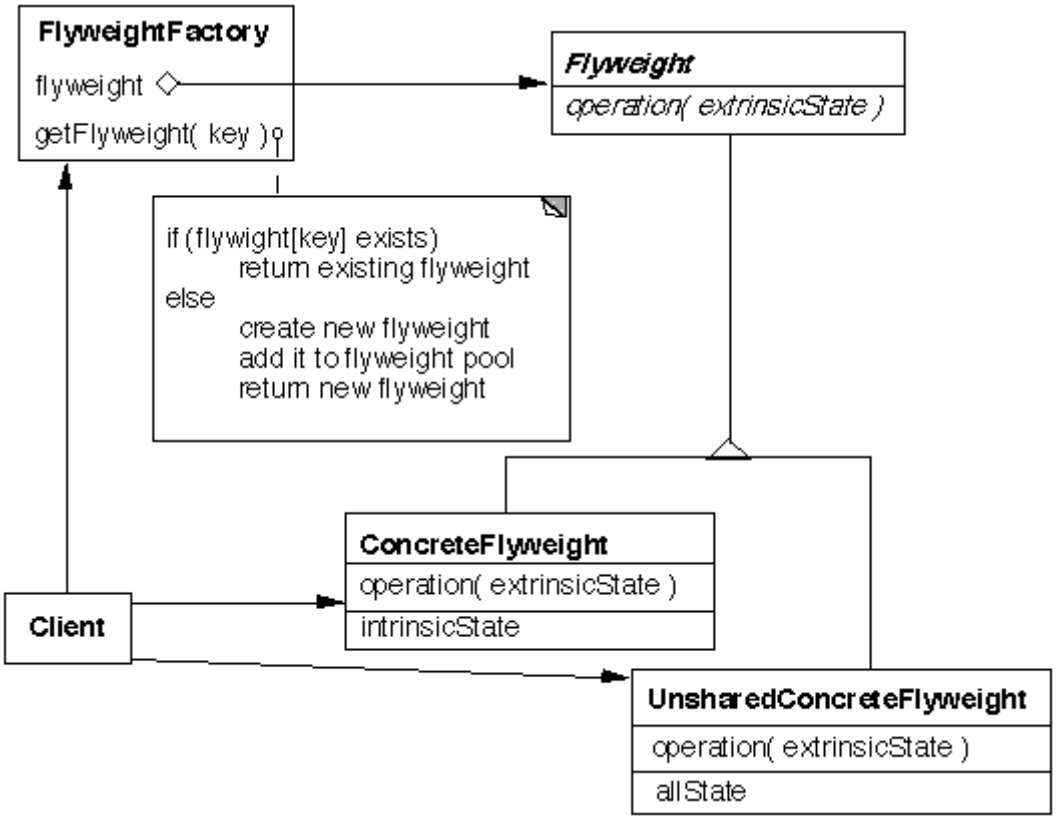
Notad en la figura que aparece dos veces el carácter ‘p’, como si se trataran de dos instancias diferentes.

Sin embargo, a nivel físico tendríamos un objeto compartido (flyweight) por cada carácter del alfabeto, y tal carácter aparecería arbitrariamente en diferentes partes del documento. Sin embargo, cada aparición de un carácter particular haría referencia a la misma instancia flyweight, la cual estaría almacenada en un pool compartido de objetos flyweight:



Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Fyweight:** Declara una interfaz a través de la cual los flyweights pueden recibir el estado extrínseco (por ejemplo, el tipo de fuente) y actuar sobre él.

- **ConcreteFlyweight:** Implementa la interfaz Flyweight y proporciona almacenamiento para el estado intrínseco (por ejemplo, el carácter ‘B’). Un objeto ConcreteFlyweight debe poder compartirse y la información que almacena debe ser intrínseca, por lo tanto, independiente del contexto.

- **UnsharedConcreteFlyweight:** La presencia de esta clase en nuestro diseño dependerá de la naturaleza de nuestra aplicación. No todas las subclases de Flyweight tienen que compartirse necesariamente. La interfaz Flyweight habilita la compartición pero no la fuerza. Es habitual que los objetos UnsharedConcreteFlyweight tengan objetos ConcreteFlyweight como hijos en algún nivel de la estructura de objetos. Por ejemplo, en el caso del procesador de textos podríamos tener las clases Row y Column, ambas con el rol UnsharedConcreteFlyweight y compuestas por objetos Caracter, con el rol ConcreteFlyweight.

- **FyweightFactory:** Crea y gestiona objetos Flyweight. Esta clase garantiza que los flyweights se comparten de manera adecuada. Cuando una clase cliente solicita un flyweight, la factoría le proporciona una instancia existente o crea una, en caso de no existir.

- **Client:** Mantiene una referencia al o a los flyweights. Calcula o almacena el estado extrínseco del o de los flyweights.

Aplicabilidad y Ejemplos

El patrón Flyweight es adecuado en las situaciones siguientes:

- La cantidad de objetos necesarios hace que el coste de almacenamiento resulte elevado y/o que un sistema sea difícil de manejar o de rendir adecuadamente.

- La mayor parte de la información de una clase es de tipo extrínseco (dimensión, tipo de fuente, posición, etc.) y por tanto, se puede compartir entre otros objetos.

- La aplicación no depende de la identidad de los objetos. Dado que los objetos flyweight se comparten, es absurdo hacerles test de identidad, ya que el test siempre retornará cierto. Siguiendo con el ejemplo del procesador de textos, supongamos que tenemos un flyweight, por ejemplo el carácter 'a', en negrita, formando parte del índice de un documento; tenemos otro flyweight que es el mismo carácter 'a', pero en cursiva y formando parte de la introducción del documento. En un caso como este, dado que el carácter 'a' se comparte, el test de identidad devolverá cierto, mientras que conceptualmente son objetos distintos.

Requisitos que deben satisfacerse a la hora de aplicar el patrón Flyweight:

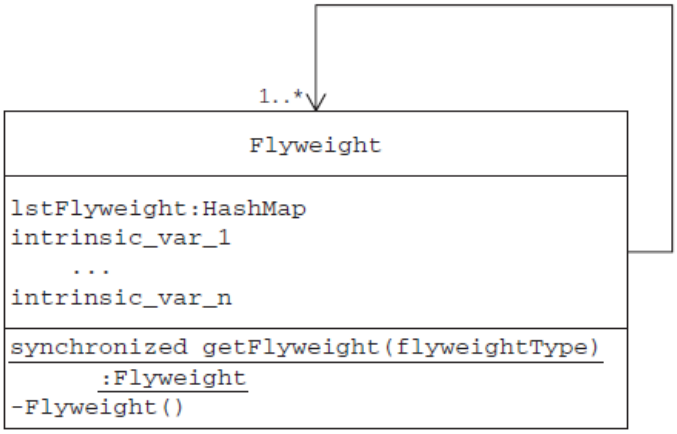
- Sólo puede existir un objeto de un tipo dado de flyweight y tal objeto debe ser compartido apropiadamente por los objetos que lo requieran. Por ejemplo, sólo podemos tener una objeto flyweight correspondiente al caracter 'a', lo cual parece bastante sensato.

- Las clases cliente no han de poder crear directamente instancias de objetos flyweight, ya que esto resultaría en un acoplamiento innecesario en los detalles de implementación del flyweight, además del peligro que supondría que el código cliente pudiera crear objetos sin ningún tipo de control. No obstante, han de tener una manera de obtener un objeto flyweight cuando lo necesiten. Como veremos, existen varias alternativas para solucionarlo.

Detalles de diseño

La clase Flyweight se diseña haciendo que su constructor sea privado. Esto sirve para evitar que el código cliente pueda crear objetos directamente invocando al constructor.

El flyweight suele contener una lista o un mapa donde almacenar una instancia y sólo una de cada tipo de flyweight que pueda existir en la aplicación. Continuando con el ejemplo del procesador de textos, tendríamos que la clase Flyweight podría almacenar en su lista/mapa tantos objetos diferentes como letras hay en el alfabeto, pero no más; sólo estaría permitido una instancia de cada letra. Esto puede ser visto como una variación del patrón Singleton.



Cuando un clase cliente necesita un objeto flyweight de un tipo dado, invoca al método estático getFlyweight(), indicándole el tipo de flyweight requerido como argumento. El método getFlyweight() tiene que estar sincronizado, ya que en otro caso se podrían producir problemas con otros hilos en ejecución (por ejemplo, que se crearan dos instancias de un mismo objeto flyweight).

Cuando el método getFlyweight() recibe la petición de devolver un determinado tipo flyweight, lo primero que hace es comprobar si la instancia ya existe en su lista/mapa (según la figura anterior, la estructura de almacenamiento sería un HashMap llamado lstFlyweight).

- Si ya existe, devuelve el objeto.
- Si no existe:
    - Crea una nueva instancia de sí mismo que se corresponda con el tipo de flyweight solicitado por el código cliente. Debido a que el método getFlyweight() reside en la propia clase Flyweight, puede acceder al constructor para crea la instancia, a pesar de que éste sea privado.
    - A continuación, la añade a la colección (al HashMap lstFlyweight). Con esto, el nuevo objeto queda cacheado, optimizando el acceso en posteriores solicitudes.
    - Finalmente, devuelve el objeto.

Notad que el diseño descrito para la clase Flyweight cumple con los requisitos comentados anteriormente, al principio de esta sección.

Si miramos la figura anterior, veremos las variables de instancia intrinsic_var_1,...,intrinsic_var_n. Estas variables almacenan el estado intrínseco de cada instancia (singleton) flyweight, es decir, la información que no varía bajo ninguna circunstancia.

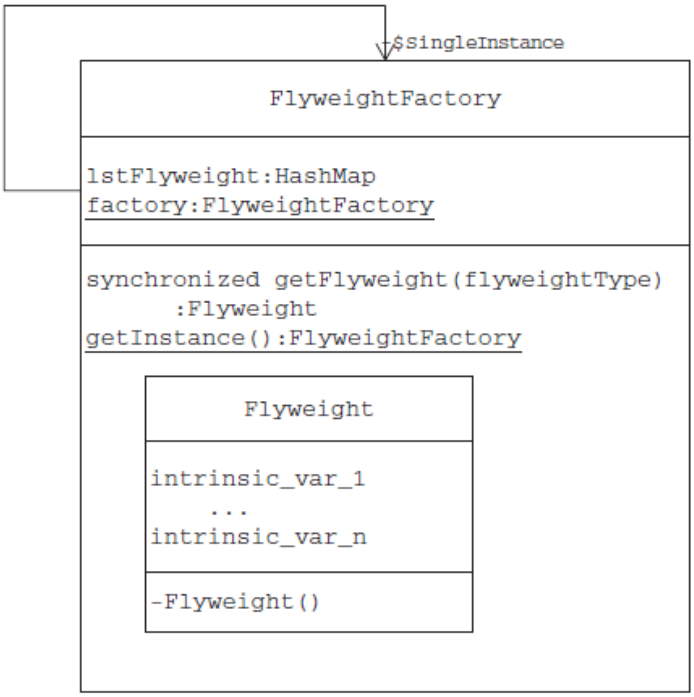
En general, un flyweight se diseña exclusivamente para representar el estado intrínseco de un objeto. No obstante, además de esto, el flyweight contiene las estructuras de datos necesarias (el HashMap) y el comportamiento adecuado (getFlyweight()) para mantener diferentes tipos de flyweights singleton's.

Diseño de alternativo: factoría de flyweight's

Como diseño alternativo al anterior, la responsabilidad de crear y mantener diferentes objetos singleton Flyweight puede traspasarse a una clase específica: la FlyweightFactory.

La clase Flyweight puede diseñarse como una clase interna de la clase FlyweightFactory. De esta manera, dado que la clase Flyweight declara su constructor privado, se impide que las clases cliente puedan crear instancias singleton, mediante invocación directa del constructor. En cambio, la clase FlyweightFactory, al ser la clase externa de Flyweight, puede invocar al constructor privado de ésta cuando lo necesite.

En este nuevo diseño, tanto en la estructura de datos (el HashMap lstFlyweight) como el comportamiento (método getFlyweight()), ambos relacionados con la creación y el mantenimiento de los objetos singleton flyweight, se han pasado desde la clase Flayweight a la clase FlyweightFactory. Ahora, la clase Flyweight se limita únicamente a representar el estado intrínseco de cada una de sus instancias.



Cuando una clase cliente necesite obtener una instancia de un tipo dado de flyweight, invocará al método getFlyweight() -ahora perteneciente a la instancia singleton FlyweightFactory- pasándole como argumento el tipo de flyweight que necesite. FlyweightFactory tiene la variable de instancia lstFlyweight (HashMap), que utiliza para almacenar a todos los objetos flyweight existentes.

La clase FlyweightFactory, como es habitual en cualquier otra factoría, se diseña como un Singleton. De esta manera se evita que el código cliente pueda crear más de una instancia de FlyweightFactory y, por consiguiente, más de una instancia de un tipo dado de flyweight.

Cuando la factoría devuelve el flyweight al código cliente, éste puede:

- Aproximación 1: Crear un objeto formado exclusivamente con los datos extrínsecos (información variable) y asociarle el flyweight recién obtenido. Esta aproximación todavía supone crear un gran número de objetos, aunque el diseño es más eficiente que no aplicar patrón alguno, ya que la información intrínseca (fija) no se encuentra duplicada en cada objeto, sino en el objeto flyweight compartido.
- Aproximación 2: Es un diseño mucho más óptimo que consisten en enviar la información extrínseca (variable) a un método del objeto Flyweight. Esta aproximación resulta en la creación de pocos objetos y sin duplicación.

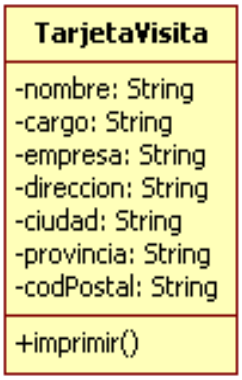
Ejemplo 1 – Tarjetas de visita (Sin aplicar el patrón Flyweight)

La aplicación que vamos a crear muestra los datos de las tarjetas de visita de todos los empleados de una empresa. Tal empresa es de un tamaño considerable, dividida físicamente en cuatro regiones: Norte, Sur, Este y Oeste.

Para nuestro caso, una típica tarjeta de visita presentará el siguiente aspecto:

```
<<Nombre del Empleado>>
      <<Cargo>>
<<Nombre de la empresa>>
<<Región, Direccion de las instalaciones>>
<<Ciudad>><<Provincia>><<Codigo Postal>>
```

Vamos a diseñar la aplicación comenzando por crear una clase TarjetaVisita. El método imprimir() nos servirá para mostrar los datos de la tarjeta:



El código para la clase queda como sigue (notad que el nombre de la clase refleja que no estamos aplicando el patrón):

TarjetaVisitaNoFlyweight.java

```
package estructurales.flyweight.aprox0;

public class TarjetaVisitaNoFlyweight {

      private String empresa;
      private String nombre;
      private String cargo;
      private String region;
      private String direccion;
      private String ciudad;
      private String provincia;
      private String codPostal;

      public TarjetaVisitaNoFlyweight(String empresa, String nombre,
```

```
                  String cargo, String region, String direccion,
                  String ciudad, String provincia, String codPostal)
      {
            this.empresa = empresa;
            this.nombre = nombre;
            this.cargo = cargo;
            this.direccion = direccion;
            this.region = region;
            this.ciudad = ciudad;
            this.provincia = provincia;
            this.codPostal = codPostal;
      }

      public void print() {
            System.out.println(nombre);
            System.out.println(cargo);
            System.out.println(region);
            System.out.println(empresa + "-" +
                        direccion + "-" +
                        ciudad + "-" +
                        provincia + "-" +
                        codPostal
            );
            System.out.println("----------------");
      }
}
```

Lo normal es que en una empresa grande haya miles de empleados, distribuidos por diferentes zonas geográficas, por lo que el programa puede necesitar llevar a cabo la creación de miles de objetos tarjeta.

Ahora veamos la clase cliente. Esta clase crea 12.000 objetos tarjeta y los imprime. Podría haber sido cualquier otro número, aunque este ya es adecuado para nuestros propósitos.

El programa lee la información sobre las tarjetas de visita desde un vector y por cada elemento del mismo crea un objeto tarjeta y lo imprime.

MainClientNoFlyweight.java

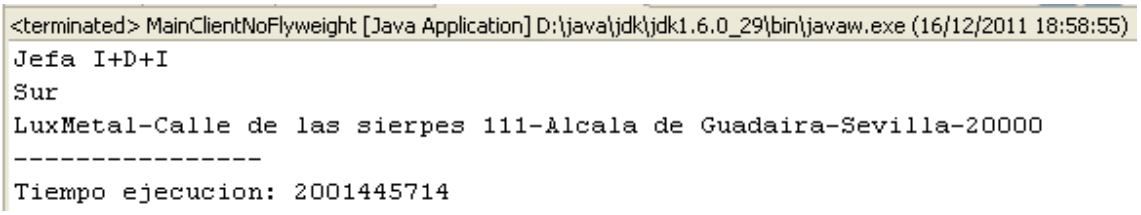
```
package estructurales.flyweight.aprox0;

import java.util.ArrayList;
import java.util.List;

public class MainClientNoFlyweight {
      private static int EMPRESA = 0, NOMBRE = 1, CARGO = 2, REGION =
3;
      private static int DIRECCION = 4, CIUDAD = 5, PROVINCIA = 6,
CODPOSTAL = 7;
```

```
	public static void main(String[] args) {
		long tiempoIni = System.nanoTime();
		List<String> empleados = inicializar();

		for (int i = 0; i < empleados.size(); i++) {
			String empleado = empleados.get(i);
			String partesEmpleado[] = empleado.split(",");

			String empresa = partesEmpleado[EMPRESA].trim();
			String nombre = partesEmpleado[NOMBRE].trim();
			String cargo = partesEmpleado[CARGO].trim();
			String region = partesEmpleado[REGION].trim();
			String direccion = partesEmpleado[DIRECCION].trim();
			String ciudad = partesEmpleado[CIUDAD].trim();
			String provincia = partesEmpleado[PROVINCIA].trim();
			String codPostal = partesEmpleado[CODPOSTAL].trim();

			// Construimos la tarjeta de visita
			TarjetaVisitaNoFlyweight tarjetaVisita =
				new TarjetaVisitaNoFlyweight(empresa, nombre,
cargo,
					region, direccion, ciudad,
provincia, codPostal);

			tarjetaVisita.print();
		}

		long tiempoFin = System.nanoTime();
		System.out.println("Tiempo ejecucion: " + (tiempoFin -
tiempoIni));
	}

	/*
	 * Por simplicidad, proporcionamos los valores directamente
	 * en lugar de obtenerlos de una base de datos o similar.
	 * El bucle tiene la intención de trabajar con más datos,
	 * para nuestro propósito no importa que se repitan.
	 */
	private static List<String> inicializar() {
		List<String> emp = new ArrayList<String>();
		for (int i=0; i<1000; i++) {
			emp.add("LuxMetal, Vicente Guash, Jefe de Ventas,
Norte, Calle Cantabria 456, Bilbao, Vizcaya, 10000");
			emp.add("LuxMetal, Antonio Martinez, Jefe Compras,
Sur, Calle de las sierpes 111, Alcala de Guadaira, Sevilla, 20000");
			emp.add("LuxMetal, Loreto Bermudez, Jefa RRHH, Norte,
Calle Cantabria 456, Bilbao, Vizcaya, 10000");
			emp.add("LuxMetal, Maria Caraz, Controller, Este,
Avenida del Mediterraneo s/n, Oropesa, Castellon, 30000");
			emp.add("LuxMetal, Sergio Fernandez, Jefe Informatica
de sistemas, Este, Avenida del Mediterraneo s/n, Oropesa, Castellon,
30000");
			emp.add("LuxMetal, Luisa Garcia, Jefa de
Contabilidad, Este, Avenida del Mediterraneo s/n, Oropesa, Castellon,
30000");
			emp.add("LuxMetal, Ramon Teruel, Jefe Calidad, Oeste,
Carretera de Portugal, Jerez de los Caballeros, Badajoz, 40000");
```

```
                emp.add("LuxMetal, Claudia Gutierrez, Jefa Marketing,
Oeste, Carretera de Portugal, Jerez de los Caballeros, Badajoz,
40000");
                emp.add("LuxMetal, Daniel Lopez, Secretario Gerencia,
Oeste, Carretera de Portugal, Jerez de los Caballeros, Badajoz,
40000");
                emp.add("LuxMetal, Carmela Pons, Jefe Oficina
Tecnica, Sur, Calle de las sierpes 111, Alcala de Guadaira, Sevilla,
20000");
                emp.add("LuxMetal, Angel Martinez, Director General,
Sur, Calle de las sierpes 111, Alcala de Guadaira, Sevilla, 20000");
                emp.add("LuxMetal, Beatriz Sanchez, Jefa I+D+I, Sur,
Calle de las sierpes 111, Alcala de Guadaira, Sevilla, 20000");
            }
            return emp;
      }

}
```

Se muestra el tiempo en nanosegundos que tarda el programa en ejecutarse. Más adelante compararemos este valor con el programa que utiliza el patrón Flyweight.

Salida (parte final del resultado):



Ejemplo 2 – Tarjetas de visita (aproximación 1)

En esta nueva aproximación se utiliza el patrón Flyweight.

De los datos de la tarjeta de visita podemos observar lo siguiente:

- El nombre y el cargo es una información que pertenece exclusivamente a cada empleado. Es una información que no se puede compartir, por lo tanto, la consideramos extrínseca.
- El nombre de la empresa es el mismo para todos los empleados. Por tanto, esta información se puede compartir, ya que es intrínseca (invariable).
- Todos los empleados que trabajan en una misma región lo hacen en las mismas instalaciones (oficinas). Por consiguiente, esta información se puede compartir, ya que es intrínseca (invariable).

La utilización del patrón Flyweight evita la duplicación innecesaria de datos. Una buena estrategia a seguir consistiría en compartir la información relativa a cada región entre todos los empleados de una misma región. Por tanto, decidimos crear cuatro tipos diferentes de objetos Flyweight, uno (y sólo uno por cada región). Notad que siendo puristas, estaremos duplicando el nombre de la empresa en cada uno de estos cuatro objetos.

En este diseño (aproximación 1), los datos extrínsecos (los que varían) se representarán como un objeto TarjetaVisita. Si tenemos mil empleados, tendremos mil objetos TarjetaVisita, pero sin información duplicada. Por tanto, tendremos que extraer de la clase TarjetaVisita los datos intrínsecos (constantes, luego comunes y susceptibles de ser compartidos) y utilizarlos para crear un objeto Flyweight.

Comencemos por crear una interfaz Flyweight, que representa la información intrínseca (compartida) de una tarjeta de visita. El código queda como sigue:

Flyweight.java

```
package estructurales.flyweight.aprox1;

public interface Flyweight {
	public String getEmpresa();
	public String getDireccion();
	public String getCiudad();
	public String getProvincia();
	public String getCodPostal();
}
```

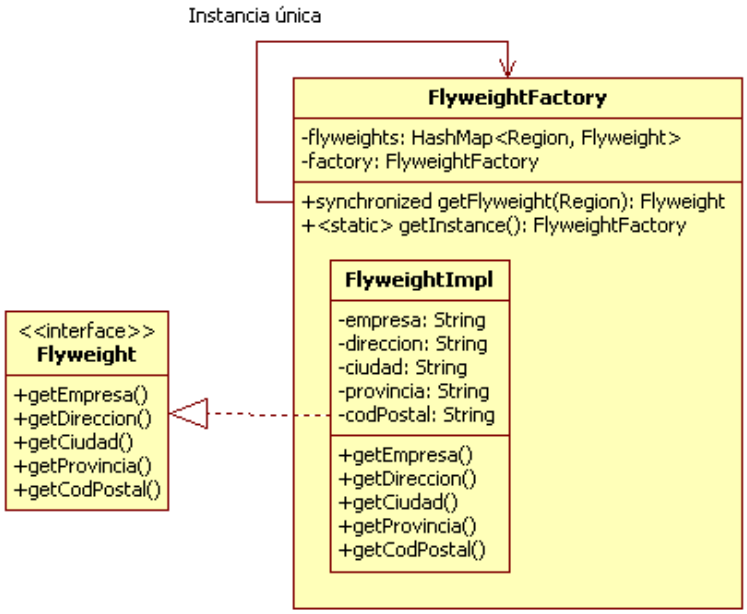
Por otro lado, necesitamos expresar los cuatro tipos de regiones de alguna manera. En lugar de utilizar Strings estáticos, una mejor opción es definir una enum:

Region.java

```
package estructurales.flyweight.aprox1;

public enum Region {
	Norte, Sur, Este, Oeste;
}
```

A continuación, vamos a crear la clase FlyweightFactory, cuya responsabilidad será crear y mantener una (y sólo una) instancia de cada tipo de objeto flyweight, teniendo en cuenta que tendremos cuatro tipos posibles, uno por cada región de la empresa.

Vamos a llamar FlyweightImpl a la clase que realmente va a implementar el Flyweight (en el diagrama general del patrón, esta sería la clase ConcreteFlyweight) y la vamos a definir como una clase anidada dentro de la clase FlyweightFactory. Primero vemos el diagrama UML y después el código:



FlyweightFactory.java (ojo que contiene a la clase FlyweightImpl)

```
package estructurales.flyweight.aprox1;

import java.util.HashMap;

/*
 * Factoría de objetos flyweight (es un Singleton).
 *
 * Encapsula los datos de una tarjeta de visita que
 * son constantes para todos los empleados de una
 * misma región.
 */
public class FlyweightFactory {
	private HashMap<Region, Flyweight> flyweights;

	// Instancia única de la factoria (Singleton)
	private static FlyweightFactory factory =
		new FlyweightFactory();

	public static FlyweightFactory getInstance() {
		return factory;
	}

	public synchronized Flyweight getFlyweight(Region region) {
		if (flyweights.get(region) == null) {
			Flyweight flyweight = new FlyweightImpl(region);
```

```
                flyweights.put(region, flyweight);
                return flyweight;
            } else {
                return (Flyweight) flyweights.get(region);
            }
        }

        // Constructor privado
        private FlyweightFactory() {
            flyweights = new HashMap<Region, Flyweight>();
        }

        // Clase FlyweightImpl
        private class FlyweightImpl implements Flyweight {
            // El nombre de la empresa es el mismo para
            // los cuatro tipos de objetos flyweight
            private static final String empresa = "LuxMetal";
            private String direccion, ciudad, provincia, codPostal;

            /*
             * Constructor. Los valores los proporcionamos directamente
             * pero tendrían que venir de una base de datos.
             */
            private FlyweightImpl(Region region) {
                if (region.equals(Region.Norte)) {
                    setValues("Calle Cantabria 456", "Bilbao",
                            "Vizcaya", "10000");
                }
                if (region.equals(Region.Sur)) {
                    setValues("Calle de las sierpes 111",
                            "Alcala de Guadaira", "Sevilla",
"20000");
                }
                if (region.equals(Region.Este)) {
                    setValues("Avenida del Mediterraneo s/n",
                            "Oropesa", "Castellon", "30000");
                }
                if (region.equals(Region.Oeste)) {
                    setValues("Carretera de Portugal",
                            "Jerez de los Caballeros",
"Badajoz", "40000");
                }
            }

            public String getEmpresa() { return empresa; }
            public String getDireccion() { return direccion; }
            public String getCiudad() { return ciudad; }
            public String getProvincia() { return provincia; }
            public String getCodPostal() { return codPostal; }

            private void setValues(String dir,
                    String ciu, String prov, String cp)
            {
                direccion = dir;
                ciudad = ciu;
                provincia = prov;
                codPostal = cp;
            }
```

```
        } // Final de FlyweightImpl
}
```

Comentarios sobre FlyweightFactory y FlyweightImpl:

- La clase FlyweightImpl se ha diseñado con un constructor privado para evitar que desde el exterior se pueda crear directamente un objeto de esta clase (esto sólo lo puede hacer la factoría).
- La clase FlyweightImpl se ha diseñado como una clase interna a FlyweightFactory para permitir a FlyweightFactory invocar a su constructor privado.
- FlyweightFactory tiene la responsabilidad de crear y gestionar diferentes instancias FlyweightImpl (sólo una por región). Para ello dispone de atributo de tipo HashMap, cuya clave es una enum que identifica la región a la que pertenece la instancia, y cuyo valor es la instancia FlyweightImpl que corresponde a esa región.
- Cuando un cliente solicita un objeto FlyweightImpl correspondiente a una región, la factoría comprueba el HashMap para ver si tal objeto ya fue creado. Si existe, lo devuelve. Si no existe, lo crea, lo guarda en el HashMap y lo devuelve.
- FlyweightFactory se ha diseñado como un Singleton para garantizar que nunca haya en el sistema dos instancias de una misma región.

Continuando con la tarjeta de visita:

Después de haber extraído la información común de TarjetaVisita y haberla trasladado a la clase FlyweightImpl, la clase TarjetaVisita todavía contiene datos extrínsecos (variables, luego no compartidos).

El constructor de TarjetaVisita, recibirá el nombre del empleado, el cargo que ostenta y un objeto Flyweight que contenga los datos intrínsecos del empleado, es decir la información que se corresponda con la región del empleado (será una de las cuatro posibles instancias flyweight existentes).

Como vemos, un objeto TarjetaVisita se construye a partir de datos extrínsecos e intrínsecos, pero de una manera que la información común de cada instancia de TarjetaVisita queda almacenada en una única instancia (flyweight) compartida:

El método print() muestra tanto los datos (extrínsecos) de TarjetaVisita como los (intrínsecos) de Flyweight.

Veamos el código para la clase:

TarjetaVisita.java

```
package estructurales.flyweight.aprox1;

public class TarjetaVisita {
	private String nombre;
	private String cargo;
	private Flyweight flyweight;

	public TarjetaVisita(String nombre, String cargo, Flyweight
flyweight) {
		this.nombre = nombre;
		this.cargo = cargo;
		this.flyweight = flyweight;
	}

	public void print() {
		System.out.println(nombre);
		System.out.println(cargo);
		System.out.println(flyweight.getEmpresa() + "-" +
				flyweight.getDireccion() + "-" +
				flyweight.getCiudad() + "-" +
				flyweight.getProvincia() + "-" +
				flyweight.getCodPostal()
		);
		System.out.println("----------------");
	}
}
```

Por último, vamos a crear una clase cliente que imprima las tarjetas de visita:

MainClient.java

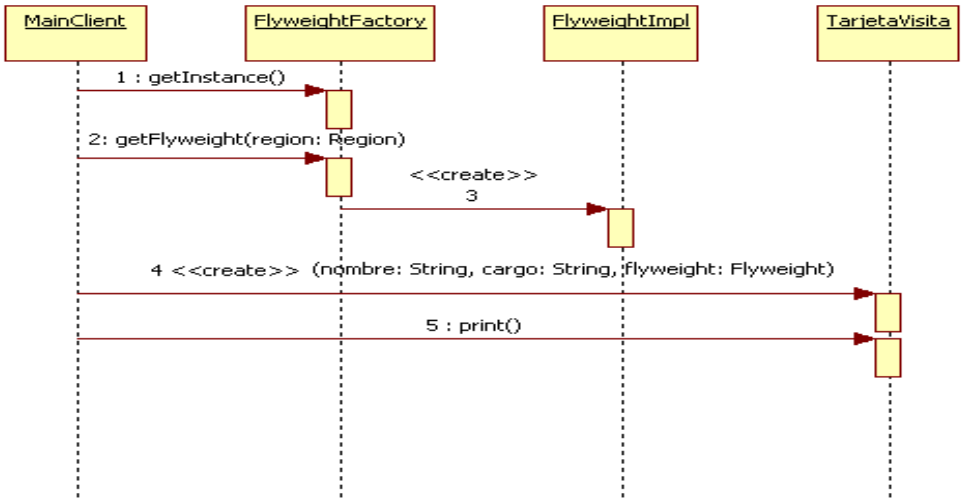
```
package estructurales.flyweight.aprox1.client;

import java.util.ArrayList;
import java.util.List;

import estructurales.flyweight.aprox1.Flyweight;
import estructurales.flyweight.aprox1.FlyweightFactory;
import estructurales.flyweight.aprox1.Region;
import estructurales.flyweight.aprox1.TarjetaVisita;

public class MainClient {
	private static int NOMBRE = 0, CARGO = 1, REGION = 2;
	private static Region region;

	public static void main(String[] args) throws Exception {
		long tiempoIni = System.nanoTime();
		List<String> empleados = inicializar();
		FlyweightFactory factory = FlyweightFactory.getInstance();

		for (int i = 0; i < empleados.size(); i++) {
			String empleado = empleados.get(i);
			String partesEmpleado[] = empleado.split(",");

			String nombreEmpleado =
partesEmpleado[NOMBRE].trim();
			String cargoEmpleado = partesEmpleado[CARGO].trim();

			String regionEmpleado =
partesEmpleado[REGION].trim();
			region = Region.valueOf(regionEmpleado);

			Flyweight flyweight = factory.getFlyweight(region);

			// Asociamos el flyweight con la tarjeta de visita

			TarjetaVisita tarjetaVisita =
				new TarjetaVisita(nombreEmpleado,
cargoEmpleado, flyweight);

			tarjetaVisita.print();
		}
		long tiempoFin = System.nanoTime();
		System.out.println("Tiempo ejecucion: " + (tiempoFin -
tiempoIni));
	}

	/*
	 * Por simplicidad, proporcionamos los valores directamente
	 * en lugar de obtenerlos de una base de datos o similar.
	 * El bucle tiene la intención de trabajar con más datos,
	 * para nuestro propósito no importa que se repitan.
	 */
	private static List<String> inicializar() {
		List<String> emp = new ArrayList<String>();
		for (int i=0; i<1000; i++) {
			emp.add("Vicente Guash, Jefe de Ventas, Norte");
```

```
                    emp.add("Antonio Martinez, Jefe Compras, Sur");
                    emp.add("Loreto Bermudez, Jefa RRHH, Norte");
                    emp.add("Maria Caraz, Controller, Este");
                    emp.add("Sergio Fernandez, Jefe Informatica de
sistemas, Este");
                    emp.add("Luisa Garcia, Jefa de Contabilidad, Este");
                    emp.add("Ramon Teruel, Jefe Calidad, Oeste");
                    emp.add("Claudia Gutierrez, Jefa Marketing, Oeste");
                    emp.add("Daniel Lopez, Secretario Gerencia, Oeste");
                    emp.add("Carmela Pons, Jefe Oficina Tecnica, Sur");
                    emp.add("Angel Martinez, Director General, Sur");
                    emp.add("Beatriz Sanchez, Jefa I+D+I, Sur");
            }
            return emp;
      }
}
```

Salida (parcial):

```
<terminated> MainClient (13) [Java Application] D:\java\jdk\jdk1.6.0_29\bin\javaw.exe (16/12/2011 18:59:30)
Beatriz Sanchez
Jefa I+D+I
LuxMetal-Calle de las sierpes 111-Alcala de Guadaira-Sevilla-20000
----------------
Tiempo ejecucion: 1624170111
```

El siguiente diagrama de secuencias muestra el flujo de mensajes durante la creación e impresión de las tarjetas de visita de cada empleado. Esta aproximación requiere la creación de un objeto TarjetaVisita por cada empleado y adicionalmente cuatro objetos Flyweight. El hecho de que los datos intrínsecos no se dupliquen en cada objeto TarjetaVisita resulta en un importante ahorro del uso de memoria.



2 : getFlyweight()

Si ahora comparamos los tiempos de ejecución de la aproximación sin patrón Flyweight con la que sí lo implementa, tenemos:

```
Sin Flyweight: 2001445714

Con Flyweight: 1624170111
```

Estos tiempos pueden variar en ejecuciones sucesivas, sobretodo en función de qué otros programas se estén ejecutando en ese momento en el ordenador y de qué tareas estén haciendo.

Lo importante es observar que cuántos más objetos se tengan que crear y más información se pueda compartir, más convincente resultará el uso del patrón, a pesar de presentar un diseño mucho más complicado que la aproximación sin patrón.

El resultado de la ejecución del programa puede parecernos muy simple para todo el esfuerzo que representa el diseño que hemos seguido. Pero no tenemos que dejarnos llevar por este pensamiento. Nuestro diseño escalará correctamente cuando tengamos que enfrentarnos a una situación que requiera crear miles de objetos y estos sean complejos, mientras que un diseño simplista no podrá siquiera ejecutarse o lo hará en pésimas condiciones.

Ejemplo 3 – Tarjetas de visita (aproximación 2)

¿Qué sucedería si en lugar de pasarle el flyweight al constructor de TarjetaVisita para que esta clase imprima las tarjetas, trasladamos el método print() a la clase Flyweight y desde el código cliente pasamos directamente los datos intrínsecos (variables) al nuevo método print() del Flyweight?

Respuesta:

Conseguiremos imprimir todas las tarjetas de visita y sólo habremos creados cuatro objetos, lo que supone un beneficio muy importante en ahorro de memoria. Por otro lado, podremos suprimir la clase TarjetaVisita, pues ya no aporta valor.

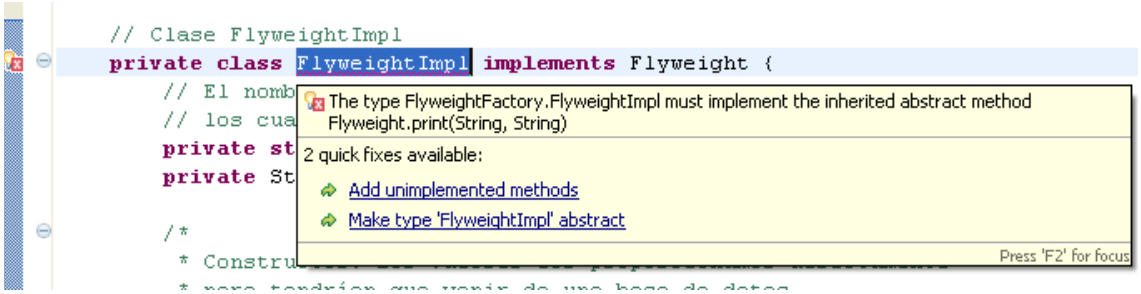
Veamos paso a paso los cambios necesarios:

Definimos el método print() en la interfaz Flyweight. Se requiere cambiar la signatura del método, ya que ahora queremos que el código cliente envíe los datos intrínsecos del empleado en curso:

Flyweight.java

```
package estructurales.flyweight.aprox2;

public interface Flyweight {
	public String getEmpresa();
	public String getDireccion();
	public String getCiudad();
	public String getProvincia();
	public String getCodPostal();
	public void print(String nombre, String cargo);
}
```

El cambio de signatura hace que ya no compile la clase anidada FlyweightImpl:



Por tanto, tendremos que implementar allí el método print().

En cuanto a la clase factoría: esta no cambia respecto a la aproximación vista anteriormente, aunque sí lo hace la clase anidada.

FlyweightFactory.java (código parcial)

```java
package estructurales.flyweight.aprox2;

import java.util.HashMap;
/*
 * Factoría de objetos flyweight (es un Singleton).
 *
 * Encapsula los datos de una tarjeta de visita que
 * son constantes para todos los empleados de una
 * misma región.
 * A traves del metodo print() recibe los datos de
 * la tarjeta de visita que son variables.
 */
public class FlyweightFactory {

	...

	// Clase FlyweightImpl
	private class FlyweightImpl implements Flyweight {

		private static final String empresa = "LuxMetal";
		private String direccion, ciudad, provincia, codPostal;

		// Constructor
		private FlyweightImpl(Region region) {
			...
		}

		public String getEmpresa() { return empresa; }
		public String getDireccion() { return direccion; }
		public String getCiudad() { return ciudad; }
		public String getProvincia() { return provincia; }
		public String getCodPostal() { return codPostal; }

		private void setValues(String dir,
				String ciu, String prov, String cp)
		{
			...
		}

		@Override
		public void print(String nombre, String cargo) {
			System.out.println(nombre);
			System.out.println(cargo);
			System.out.println(getEmpresa() + "-" +
					getDireccion() + "-" +
					getCiudad() + "-" +
					getProvincia() + "-" +
					getCodPostal()
			);
			System.out.println("----------------");

		}

	} // Final de FlyweightImpl
}
```

Por último, tenemos que hacer que el código cliente sea quien invoque al método print(), proporcionándole los dos argumentos que necesita:

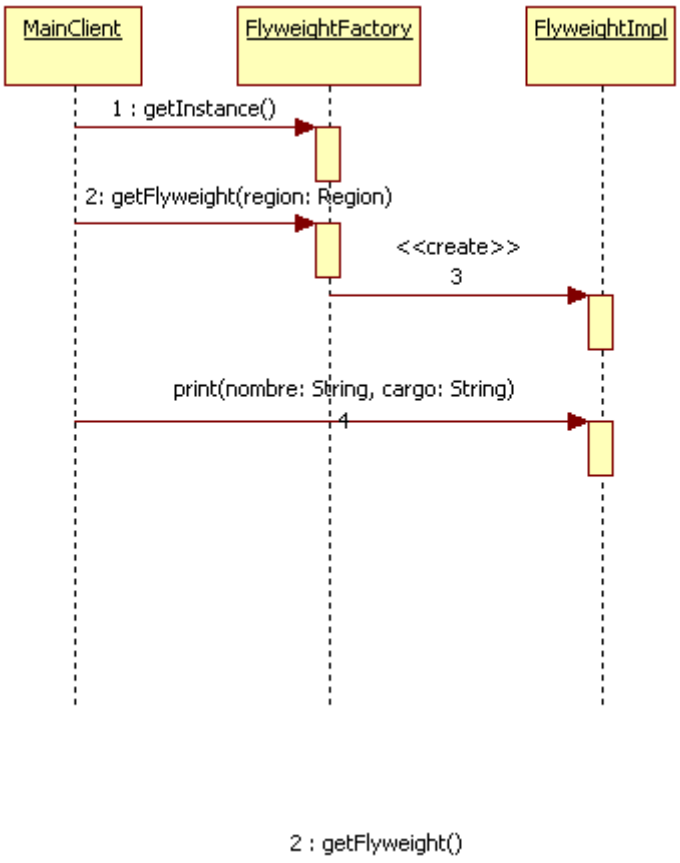
MainClient.java (código parcial)

```
package estructurales.flyweight.aprox2.client;

import java.util.ArrayList;
import java.util.List;

import estructurales.flyweight.aprox2.Flyweight;
import estructurales.flyweight.aprox2.FlyweightFactory;
import estructurales.flyweight.aprox2.Region;

public class MainClient {
	private static int NOMBRE = 0, CARGO = 1, REGION = 2;
	private static Region region;

	public static void main(String[] args) throws Exception {
		long tiempoIni = System.nanoTime();
		List<String> empleados = inicializar();
		FlyweightFactory factory = FlyweightFactory.getInstance();

		for (int i = 0; i < empleados.size(); i++) {
			String empleado = empleados.get(i);
			String partesEmpleado[] = empleado.split(",");

			String nombreEmpleado =
partesEmpleado[NOMBRE].trim();
			String cargoEmpleado = partesEmpleado[CARGO].trim();

			String regionEmpleado =
partesEmpleado[REGION].trim();
			region = Region.valueOf(regionEmpleado);

			Flyweight flyweight = factory.getFlyweight(region);

			// Pasar al flyweight los datos extrínsecos
			flyweight.print(nombreEmpleado, cargoEmpleado);
		}
		long tiempoFin = System.nanoTime();
		System.out.println("Tiempo ejecucion: " + (tiempoFin -
tiempoIni));
	}

	private static List<String> inicializar() {
		...
	}
}
```

Notad que la clase TarjetaVisita ya no es necesaria, ya que los datos extrínsecos los proporciona directamente la clase cliente al Flyweight, por lo que éste tiene toda la

información necesaria para imprimir las tarjetas. Por tanto, podemos eliminar del diseño a la clase TarjetaVisita.

El siguiente diagrama de secuencias muestra el flujo de mensajes para esta segunda aproximación.



Problemas específicos e implementación

Tiempo de ejecución

El patrón Flyweight puede hacer que una aplicación sea más lenta en ejecutarse que la misma aplicación sin utilizar el patrón. Este coste obedece a las operaciones de obtención de los datos extrínsecos: búsqueda, extracción y cálculo (cuando es necesario), sobre todo cuando tales datos se encuentran inicialmente encapsulados junto con la información intrínseca.

En cualquier caso, este inconveniente es un mal menor comparado con el beneficio que supone el ahorro de memoria, el cual aumenta conforme mayor es el número de objetos flyweight que se comparten y cuanto mayor son los datos que se comparten.

Inmutabilidad

Cuando múltiples objetos cliente comparten una misma instancia de otro objeto, por ejemplo un flyweight, nos encontramos con la situación de que si uno de ellos cambia el estado de objeto compartido, el resto de objetos apreciará ese cambio al acceder al objeto compartido. La manera más sencilla y habitual de evitar que los objetos cliente se interfieran indirectamente los unos a los otros, consiste en imposibilitar que puedan cambiar el estado del objeto compartido. Esto se logra diseñando el objeto compartido como un objeto inmutable, esto es, haciendo que una vez creado no pueda ser modificado. Un ejemplo de clase inmutable es la clase String del API de Java.

Patrones relacionados

Flyweight se suele combinar a veces con Composite para implementar estructuras jerárquicas tipo grafo cuyos nodos hoja (leaf) se comparten entre otros miembros de la estructura.

Patrones como State y Strategy a menudo se implementan como flyweights.