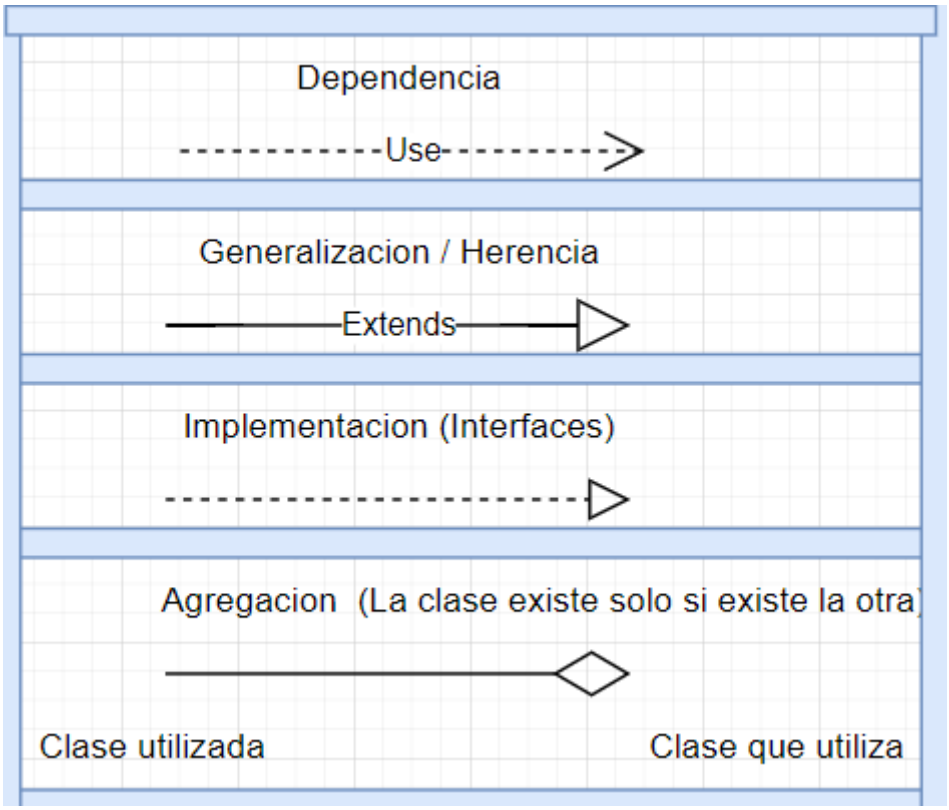
Contenido

UML



Excepciones – Java

```
1
2  public class ExeptionDividirPorCero extends Exception {
3
4      private static final long serialVersionUID = 1L;
5
6      public ExeptionDividirPorCero() {
7          super("Exeption Dividir por cero");
8      }
9      public ExeptionDividirPorCero(String str) {
10         super(str);
11     }
12
13 }
14
```

Luego nosotros dentro de la clase que lanzara este error, le agregamos donde queremos que lance el error **throw new ExcepcionDividirPorCero(“Este es mi mensaje personalizado”);**

Montículo – Cola de Prioridad

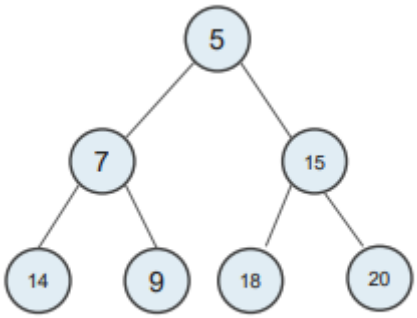
Se crea un vector donde la primera posición no se utiliza (la 0).

Existen solo 4 operaciones que se pueden realizar con un Montículo

- Crear la cola de prioridad
- Agregar un elemento
- Eliminar el primer elemento de la cola de prioridad

- Consultar el valor de la primera cola de prioridad

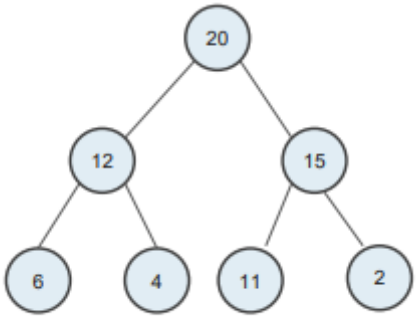
Existen Montículos de mínimo y Montículos de máximo



Montículo de mínimo

Acá el vector de la cola de prioridad quedaría así

Índice	0	1	2	3	4	5	6	7
Valor		5	7	15	14	9	18	20



Montículo de máximo

Acá el vector de la cola de prioridad quedaría así

Índice	0	1	2	3	4	5	6	7
Valor		20	12	15	6	4	11	2

Insertar un elemento

Imaginando la situación que queramos insertar un elemento en un Montículo de mínimo

Siguiendo con el ejemplo anterior, nosotros tendríamos un vector de esta manera

Índice	0	1	2	3	4	5	6	7
Valor		5	7	15	14	9	18	20

Insertamos el numero 6 a nuestro Montículo de mínimo

Índice	0	1	2	3	4	5	6	7	8
Valor		5	7	15	14	9	18	20	6

Siempre, cuando se inserta un elemento, se lo pone al final.

Y de ahí empezamos a evaluar.

Índice	0	1	2	3	4	5	6	7	8
Valor		5	7	15	14	9	18	20	6

Como lo insertamos en la posición 8, nos preguntamos si (8/2 = 4) la posición 4 (que es el padre dentro del montículo) contiene un valor menor al que queremos ingresar, de ser verdadero dejamos todo como esta, pues esta bien. Ahora, de no serlo, tendremos que intercambiar los valores y repetir el proceso hasta que esa pregunta de verdadero.

Índice	0	1	2	3	4	5	6	7	8
Valor		5	7	15	6	9	18	20	14

Estamos en la posición 4, nos preguntamos si la posición 2 (4/2 = 2) contiene un valor menor al que queremos ingresar (6). De ser verdadero, dejamos todo como esta; de ser falso, tendremos que intercambiar y repetir el proceso.

Índice	0	1	2	3	4	5	6	7	8
Valor		5	6	15	7	9	18	20	14

Estamos en la posición 2, nos preguntamos si la posición 1 (2/2 = 1) contiene un valor menor al que queremos ingresar (6). De ser verdadero, dejamos todo como esta; de ser falso, tendremos que intercambiar y repetir el proceso.

Finalmente, quedaría así el montículo de mínimo luego de agregar el 6.

Índice	0	1	2	3	4	5	6	7	8
Valor		5	6	15	7	9	18	20	14

El mismo procedimiento seria si estuviéramos trabajando con un montículo de máximo, pero con la diferencia de preguntar si es menor que, estaríamos preguntando si es mayor que.

Sacar elemento

Siempre se saca el primer elemento del vector. En este caso, estaríamos sacando al 5.

Índice	0	1	2	3	4	5	6	7	8
Valor		5	6	15	7	9	18	20	14

Luego de sacar al 5, intercambiamos al ultimo valor que contenga nuestro vector a la posición 1. Y comenzamos a evaluar a partir de ahí.

Índice	0	1	2	3	4	5	6	7
Valor		14	6	15	7	9	18	20

Como estamos en un montículo de mínimo, el valor del padre debe ser menor al de sus hijos. Por lo que tendremos que garantizar en este caso que 14 sea menor que 6 y que 15, en caso contrario, intercambiamos con el menor de ellos.

Si estuviéramos en un montículo de máximo, el valor del padre debe ser mayor al de sus hijos. Por lo que tendríamos que garantizar en este caso que 14 sea mayor que 6 y que 15, en caso contrario, intercambiaríamos con el mayor de ellos.

Índice	0	1	2	3	4	5	6	7
Valor		6	14	15	7	9	18	20

Ahora nos preguntamos lo mismo, ¿14 es menor que el valor que contienen sus hijos?

De ser verdadero dejamos todo como esta, pues ya sabemos que cumple con la condición de montículo. De ser falso, debemos intercambiarlo por el menor valor de sus hijos y seguir evaluando.

Índice	0	1	2	3	4	5	6	7
Valor		6	14	15	7	9	18	20

En este caso es falso, así que intercambiamos por el menor de ellos.

Índice	0	1	2	3	4	5	6	7
Valor		6	7	15	14	9	18	20

Seguimos evaluando, nos tocaría preguntar por la posición 8 y 9; pero como no existen dentro de nuestro vector, ya sabemos entonces que este vector cumple con la condición de montículo.

Algoritmos de Ordenamiento

	Estables	Sensibles
Insercion	X	X
Seleccion		
Burbujeo	X	X
Shell		
Quick		
Merge	X	

Burbujeo – O(n^2)

Se compara elemento a elemento de la siguiente manera.

Índice	0	1	2

Valor	7	6	3

Nos preguntamos si el 6 es menor que el 7.

De serlo intercambiamos, de no serlo dejamos como esta y continuamos

En este caso toca intercambiar.

Índice	0	1	2
Valor	6	7	3

Nos preguntamos si el 3 es menor que el 7.

De serlo intercambiamos, de no serlo dejamos como esta y continuamos

En este caso toca intercambiar.

Índice	0	1	2
Valor	6	3	7

Como no hay mas elementos a la derecha, se vuelve a empezar.

Se repite el mismo proceso hasta que en toda una pasada completa no hayan intercambios.

Índice	0	1	2
Valor	6	3	7

Nos preguntamos si el 3 es menor que el 6. Efectivamente, entonces intercambiamos.

Índice	0	1	2
Valor	3	6	7

Nos preguntamos, 6 es menor que 7. No, entonces dejamos como esta.

Como hubo un intercambio en esta ultima iteración, entonces volvemos a repetir el proceso.

Nos preguntamos, 3 es menor a 6. Si, entonces seguimos con el siguiente.

Índice	0	1	2
Valor	3	6	7

Nos preguntamos, 6 es menor a 7. Si entonces, seguimos con el siguiente.

Índice	0	1	2
Valor	3	6	7

No hay siguiente, no hubo intercambios en esta ultima pasada, entonces terminamos de ordenar el vector.

Inserción – O(n^2)

Este algoritmo primero marca al primer elemento del vector como ordenado. Y luego

Índice	0	1	2
Valor	7	6	3

Tomamos el primer elemento y lo comparamos con el siguiente.

Si es menor, entonces hacemos un intercambio. Si no, seguimos.

En este caso, toca hacer un intercambio

Índice	0	1	2
Valor	6	7	3

Ahora, nos preguntamos si 3 es menor a 7. Efectivamente, entonces hacemos un intercambio.

Índice	0	1	2
Valor	6	3	7

Como hay elementos a la izquierda pregunto si el 3 es menor que 6, de serlo continuo intercambiando. De no serlo dejo ahí.

Índice	0	1	2
Valor	3	6	7

En realidad, en código el intercambio solo se realiza una vez y es cuando no encuentra con quien comparar.

Aquí el código de Inserción.

```
int i, key, j;
for (i = 1; i < n; i++)
{
    key = arr[i];
    j = i - 1;

    // Move elements of arr[0..i-1],
    // that are greater than key, to one
    // position ahead of their
    // current position
    while (j >= 0 && arr[j] > key)
    {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
}
```

Selección – O(n^2)

Buscamos el menor elemento dentro del vector, y lo intercambiamos por lo que este en la posición 0.

Luego, buscamos el menor elemento dentro del vector, pero partiendo de la posición 1. Luego, intercambiamos por lo que este en la posición 1.

Luego, buscamos el menor elemento dentro del vector, pero partiendo de la posición 2. Luego, intercambiamos por lo que este en la posición 2.

Ejemplo:

Empezamos desde 0 hasta 3.

Índice	0	1	2
Valor	7	6	3

Encontramos el menor, que es 3. (posición 2).

Intercambiamos por lo que este en la posición 0.

Índice	0	1	2
Valor	3	6	7

Ahora, empezamos recorriendo desde la posición 1 hasta el final. Buscando el menor.

Encontramos que el menor es el 6, y ya se encuentra en la posición correcta, por lo que terminamos el algoritmo.

Shell – O((n*log(n))^2)

Comparación por saltos.

En cada iteración se reducen los saltos a la mitad de los que estaban antes.

Índice	0	1	2	3
Valor	7	6	3	5

Se toma la cantidad de elementos que contenga este vector y se lo divide entre 2. En este caso obtendríamos:

4/2 = 2

Tomamos la parte entera, se formaría 2 subgrupos de comparación.

Comparamos entonces el 7 con el 3.

Y en el otro subgrupo, el 6 con el 5.

Índice	0	1	2	3
Valor	7	6	3	5

7 < 3? No, entonces intercambiamos

6 < 5? No, entonces intercambiamos

Índice	0	1	2	3
Valor	3	5	7	6

Ahora, como se realizaron cambios, continuamos con el algoritmo. Sigue dividir la cantidad de subgrupos que habíamos obtenido la vez pasada entre 2 nuevamente.

Ahora, formaríamos 1 subgrupo de comparación. Es decir, comparamos todos con todos.

Índice	0	1	2	3
Valor	3	5	7	6

3 < 5? Si, no hacemos nada.

5<7? Si, no hacemos nada.

7 < 6? No, intercambiamos.

Índice	0	1	2	3
Valor	3	5	6	7

Como hubo intercambios, repetimos el proceso.

Al tener 1 de subgrupos, seguimos con 1 solo subgrupo de comparación.

3<5? Si, no hacemos nada.

5<6? Si, no hacemos nada.

6<7? Si, no hacemos nada.

Terminamos de iterar el vector y no hubo intercambios, entonces finaliza el algoritmo.

Quick-Sort – O(n^2)

Este algoritmo en promedio es n*log(n). Casi siempre es así.

Pero en el peor de los casos es n^2. El peor de los casos se da cuando el pivote termina en un extremo de la lista. Este peor caso suele ocurrir cuando se le pasan listas ordenadas o casi ordenadas.

Lo primero que hay que hacer es seleccionar un pivote dentro de nuestro vector.

Este pivote lograra dividir nuestro vector original en dos sub vectores.

Empezamos recorriendo el vector de la izquierda y nos preguntamos si el numero al que apuntamos es mayor que nuestro pivote, de serlo pasamos a revisar el sub vector de la derecha, si no llegara a ser mayor, tendremos que avanzar hasta que encontremos uno mayor o que nos pasemos de los límites del sub vector.

Con el sub vector de la derecha hacemos lo mismo, pero en vez de preguntarnos si es mayor que el pivote, al estar del lado derecho nos preguntaremos si es menor que nuestro pivote. Debido a que nosotros buscamos tener a los elementos mas grandes a la derecha y a los elementos más pequeños a la izquierda.

Cuando encontramos un elemento dentro del sub vector izquierdo, y un elemento dentro del sub vector derecho. Lo que hacemos es intercambiarlos. Y continuamos con el proceso hasta que se nos acabe el limite de nuestros sub vectores (hasta que hallamos recorrido todos los elementos de nuestros sub vectores).

Una vez que se termina de recorrer, se llama recursivamente a este algoritmo de modo que ahora se evaluaran a los dos sub vectores por separado de la misma manera que hicimos para el vector original.

Se tomara un pivote en cada sub vector y se evaluara de izq. a der.

Si lo miramos en forma práctica es muy fácil

Índice	0	1	2	3	4	5	6	7
Valor	5	3	7	6	8	4	9	6

Elegimos un pivote. En este caso el de la mitad vec[3].

El objetivo es llevar todos los menores a 6, para la izquierda y todos los mayores a 6 para la derecha.

5>6? No, entonces avanzamos.

3>6? No, entonces avanzamos.

7>6? Si, entonces marcamos este y pasamos a ver el sub vector derecho.

6<6? No, entonces avanzamos.

9<6? No, entonces avanzamos.

4<6? Si, entonces marcamos a este elemento y lo intercambiamos con el elemento marcado del sub vector izquierdo.

Índice	0	1	2	3	4	5	6	7
Valor	5	3	4	6	8	7	9	6

Continuamos…

Volvemos al sub vector izquierdo

6>6? No, entonces avanzamos.

Como ya nos pasamos del limite del sub vector izquierdo. Continuamos con el sub vector derecho.

8<6? No, entonces avanzamos.

6<6? No, entonces avanzamos.

Como ya nos fuimos del limite, ahora toca llamar de forma recursiva a estos sub vectores.

Subvector izquierdo

Índice	0	1	2
Valor	5	3	4

Tomariamos a sub_vec[1] como pivote y haríamos el mismo procedimiento.

Subvector derecho

Índice	0	1	2	3
Valor	8	7	9	6

Tomariamos a sub_vec[1] como pivote y haríamos el mismo procedimiento.

Merge-Sort – O(n*log(n))

Este algoritmo tiene un pseudocodigo similar a este

```
merge_sort(lista) {

if(lista.length > 1) {

merge_sort(sub_lista_izq);

merge_sort(sub_lista_der);

mezclar_listas(sub_lista_izq, sub_lista_der);

}

}
```

La lista se divide en dos partes de tamaños iguales.

Primero, se llama de forma recursiva a la funcion merge_sort pasandole la sub_lista_izq. Por lo cual la primer sub_lista que ordenaremos será la izq.

Seguiremos subdividiendo la lista hasta que lleguemos a tener una sublista de un solo elemento. Acá, tendremos nuestro caso base, y retornaremos.

Por lo que pasaremos a hacer un merge de la parte derecha, hasta que esta tenga solo un elemento. Ahí se mezclara de forma ordenada.

Esta mezcla será un retorno de alguna de las llamadas recursivas por lo que podrán seguir avanzando y así continuar con el procedimiento de todo el algoritmo de ordenamiento.

Programación Dinámica

La programación dinámica es aplicable para optimizar el tiempo de ejecución de un programa o algoritmo.

Lo que se busca es eliminar toda solución que se repite.

El clásico ejemplo es el Fibonacci.

```
public int fib (int n) {
   if (n < 2)
      return 1;
   return fib(n-1) + fib(n-2);
}
```

Esta es la típica función recursiva de Fibonacci.

Si aplicamos Programación Dinámica, podremos resolver cada solución una única vez, y esta guardarla en memoria para que cuando luego otra de estas secuencias recursivas tenga que obtenerla, solamente la pueda leer de memoria en vez de tener que perder tiempo de cómputo para resolver algo que ya había resuelto previamente.

Funciona muy parecido a una memoria cache, antes de resolver un problema nos fijamos si ya lo tenemos resuelto en la variable que hayamos dedicado para esto. Si está resuelto, listo usamos ese dato; de no estar lo tenemos que resolver, una vez resuelto lo ponemos dentro de la variable y finalmente hacemos uso de él.

La clave está en recordar los resultados que obtuvimos antes, para ahorrar tiempo de procesamiento.

En la programación dinámica se resuelven primero los subejemplares más pequeños y por tanto más simples. Combinando las soluciones se obtienen soluciones de ejemplares sucesivamente más grandes hasta llegar al objetivo.

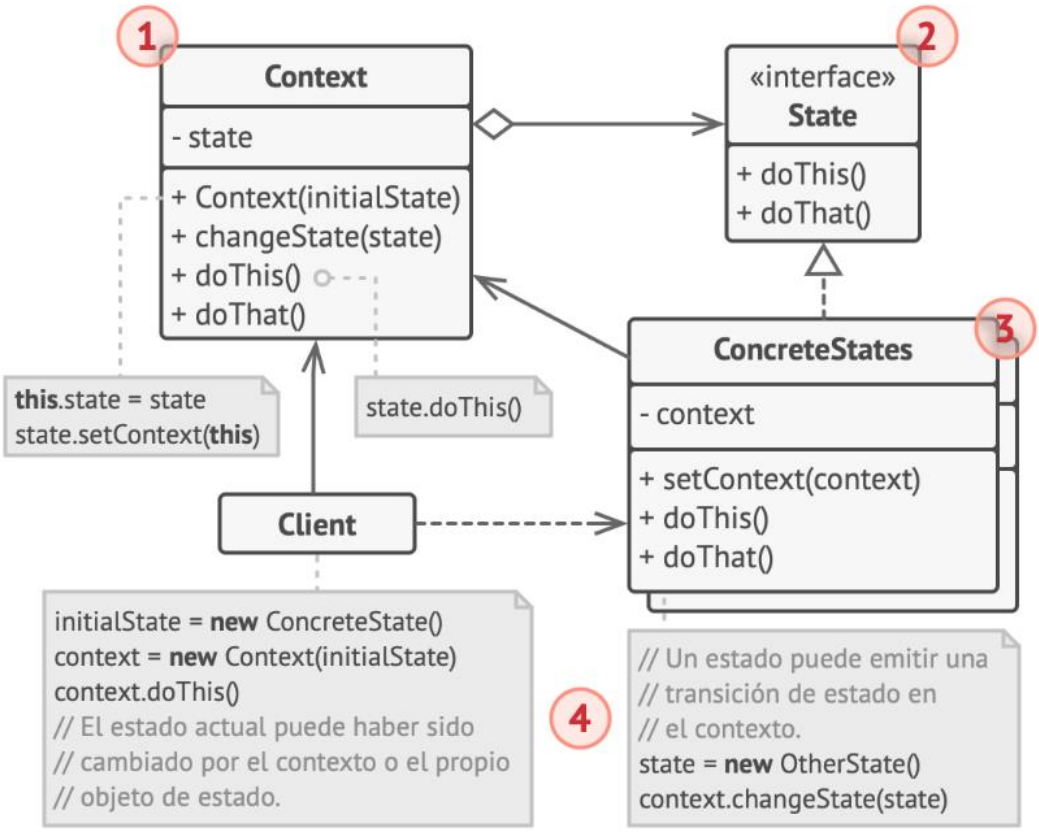
Patrones de Diseño

State

Este patrón de diseño es aplicable para cuando nuestro objeto debe cambiar repentinamente sus funcionalidades, respecto de algunas condiciones.

El ejemplo más claro es el del celular. Cuando este tiene la pantalla bloqueada, si nosotros presionamos un botón lo que hará el celular es prender la pantalla. Pero ahora, si nosotros presionamos ese mismo botón con la pantalla desbloqueada, ese botón tendrá otra funcionalidad. Y eso, debido a que el "celular" cambio de estado.

Estado bloqueado a estado desbloqueado.


1
Context
- state
+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()
2
«interface»
State
+ doThis()
+ doThat()
3
ConcreteStates
- context
+ setContext(context)
+ doThis()
+ doThat()
Client
this.state = state
state.setContext(this)
state.doThis()
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// El estado actual puede haber sido
// cambiado por el contexto o el propio
// objeto de estado.
4
// Un estado puede emitir una
// transición de estado en
// el contexto.
state = new OtherState()
context.changeState(state)

1. La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2. La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.

3. Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

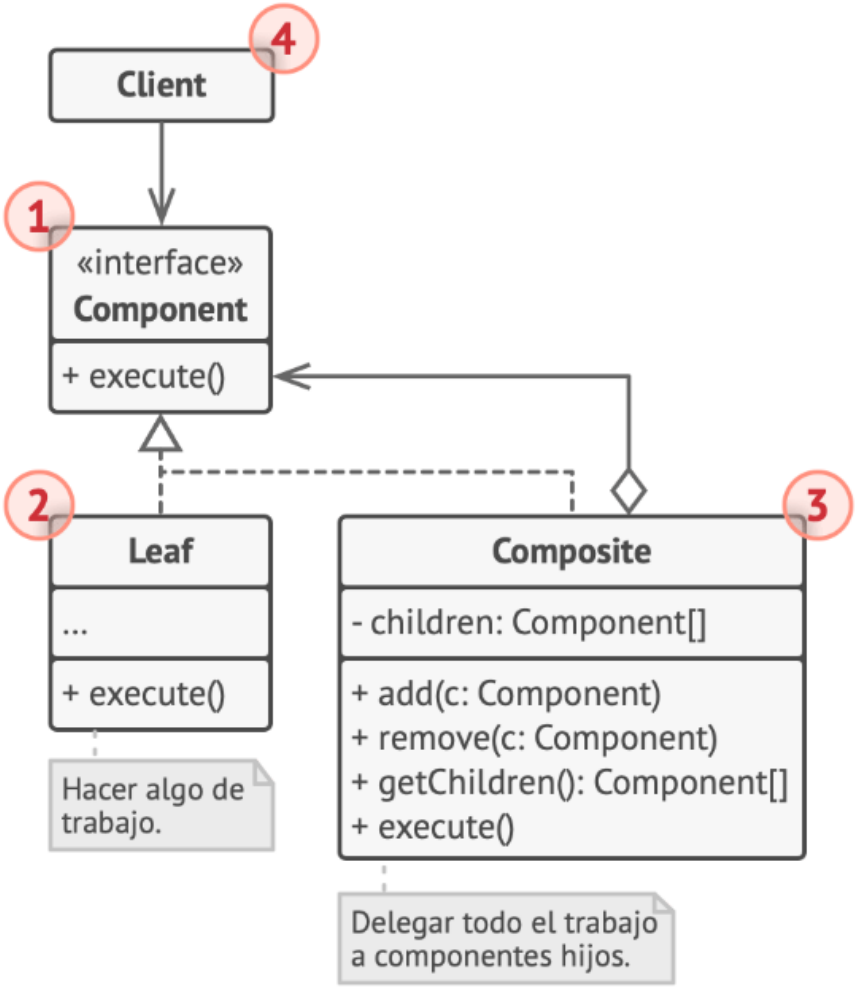
   Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

Composite

El uso de este patrón de diseño solo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.

El clásico ejemplo es el de las Cajas y Productos. Donde una caja puede contener varios productos, como también puede contener otras cajas; y esas cajas lo mismo, pueden contener productos o también cajas. Y así sucesivamente.


4
Client
1
«interface»
Component
+ execute()
2
Leaf
...
+ execute()
Hacer algo de trabajo.
3
Composite
- children: Component[]
+ add(c: Component)
+ remove(c: Component)
+ getChildren(): Component[]
+ execute()
Delegar todo el trabajo a componentes hijos.

1. La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

2. La **Hoja** es un elemento básico de un árbol que no tiene subelementos.

   Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.

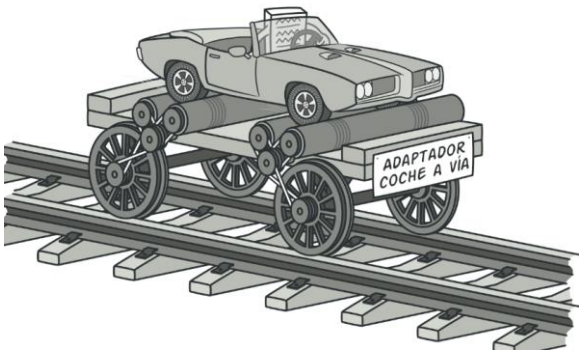
3. El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.

   Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

4. El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

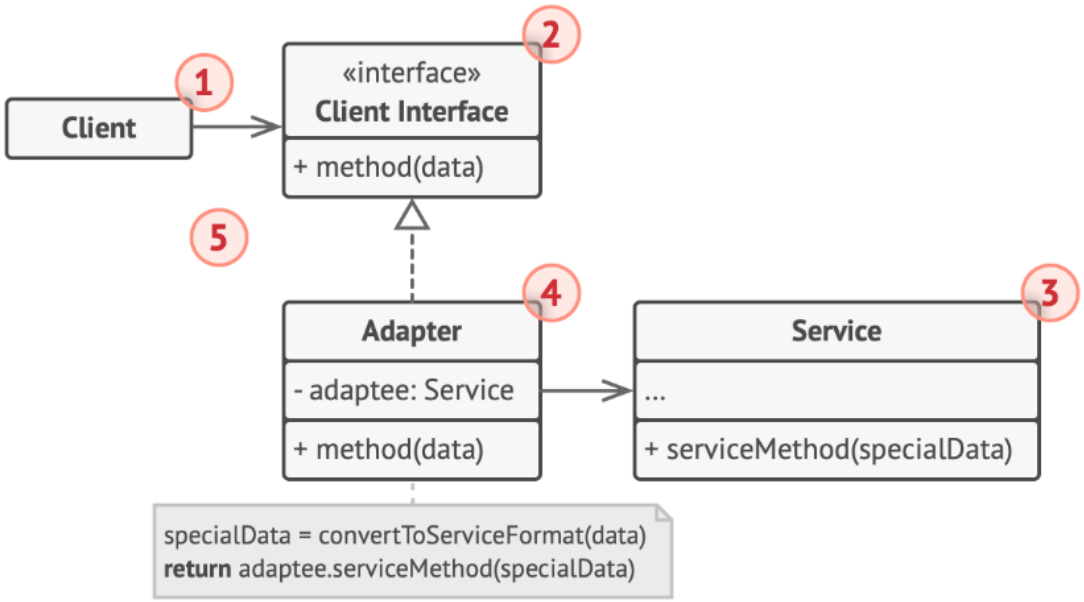
Adapter

Es un patrón de diseño que permite la colaboración entre objetos con interfaces diferentes.



En este ejemplo visual vemos como adaptamos un coche, a que ande en una vía de tren. Ese es el concepto clave del patrón de diseño Adapter.

El Adapter se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.



1. La clase **Cliente** contiene la lógica de negocio existente del programa.

2. La **Interfaz con el Cliente** describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.

3. **Servicio** es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.

4. La clase **Adaptadora** es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz adaptadora y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.

5. El código cliente no se acopla a la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, puedes introducir nuevos tipos de adaptadores en el programa sin descomponer el código cliente existente. Esto puede resultar útil cuando la interfaz de la clase de servicio se cambia o sustituye, ya que puedes crear una nueva clase adaptadora sin cambiar el código cliente.