Patrones de Diseño y Colecciones

Jonatan Goméz Perdomo, Ph.D.
jgomezpe@unal.edu.co

Carlos Andrés Sierra, Ms.C.
casierrav@unal.edu.co

Grupo de investigación en vida artificial – Research Group on Artificial Life – (Alife)
Departamento de Ingeniería de Sistemas e Industrial
Facultad de Ingeniería
Universidad Nacional de Colombia

Agenda

Agenda

Patrones de Diseño

Se derivan de las ideas de Christopher Alexander, quien detectó elementos comunes en el diseño de software. Estos patrones corresponden a la definición de un problema de común ocurrencia, y una solución generalizada y reutilizable para resolver dicho problema.

No son implementaciones de código como tal, sino son diseños basados en objetos transferibles a código de un problema claramente identificado. La idea de estos patrones es usar la experiencia de otros resolviendo problemas similares al que se está resolviendo.

Ventajas y Desventajas

Ventajas

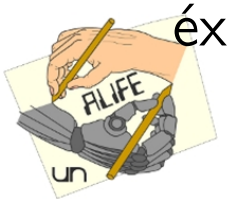
- Se aprovecha el conocimiento y experiencia de otros arquitectos y desarrolladores que han resuelto problemas similares en el pasado.
- Es abstracto, lo que hace a los patrones independientes del lenguaje de programación, librerías, y tecnologías de desarrollo que se estén usando.

Desventajas

- Requiere que la persona que quiere implementarlo tenga la experiencia para relacionar el problema con un patrón ya definido.
- No siempre es fácil ajustar el patrón de diseño a la arquitectura utilizada para resolver el problema bajo estudio.

Recomendaciones de Uso

- No es bueno abordar el uso de patrones de diseño en la vida real sin tener experiencia previa en el diseño de software. Es bueno hacer previamente algunas pruebas de concepto para entender el comportamiento de los patrones y determinar si alguno se ajusta a su necesidad de software.
- En general, los patrones de diseño hacen bastante uso de conceptos de programación orientada a objetos como lo son herencia y polimorfismo. Si el equipo de desarrollo no tiene experiencia o claridad en estos conceptos, se sugiere no hacer uso de patrones de diseño.
- Un patrón de diseño debe ser usado cuando realmente el problema a resolver lo requiere. Usar por usar patrones de diseño, o mezclarnos sin un análisis riguroso puede ser lesivo y contraproducente para el éxito de un proyecto.



Agenda

Patrones de Creación

Definición

Los **Patrones de Creación** son los que facilitan la tarea de creación de nuevos objetos, haciendo énfasis en la encapsulación de la lógica de la instanciación, ocultando los detalles concretos de cada objeto, y permitiendo trabajar con abstracciones.

Estos patrones buscan el incremento en la flexibilidad del diseño y la reutilización de código construido, apuntando a mejorar la calidad del software y disminuir la complejidad para el desarrollador.

Patrones de Creación - Factory Method(I)

Es un patrón que se utiliza para crear objetos en una superclase, mientras que al mismo tiempo las sub-clases pueden alterar el tipo de objetos a crear. Es decir, la idea es crear objetos sin exponer la lógica de creación de dichos objetos.

Para crear los objetos se utiliza un intermediario llamado Fábrica (*Factory*), así que no se usa de la palabra new, por el contrario, se hace uso de métodos de la fábrica para crear objetos según se requiera.

Es útil cuando no se conoce mucho de las dependencias y tipos de objetos con los que se va a trabajar definitivamente, en esencia, porque separa la creación de objetos y las reglas del negocio.

Patrones de Creación - Factory Method (II)

Problema: Generar botones para múltiples GUI

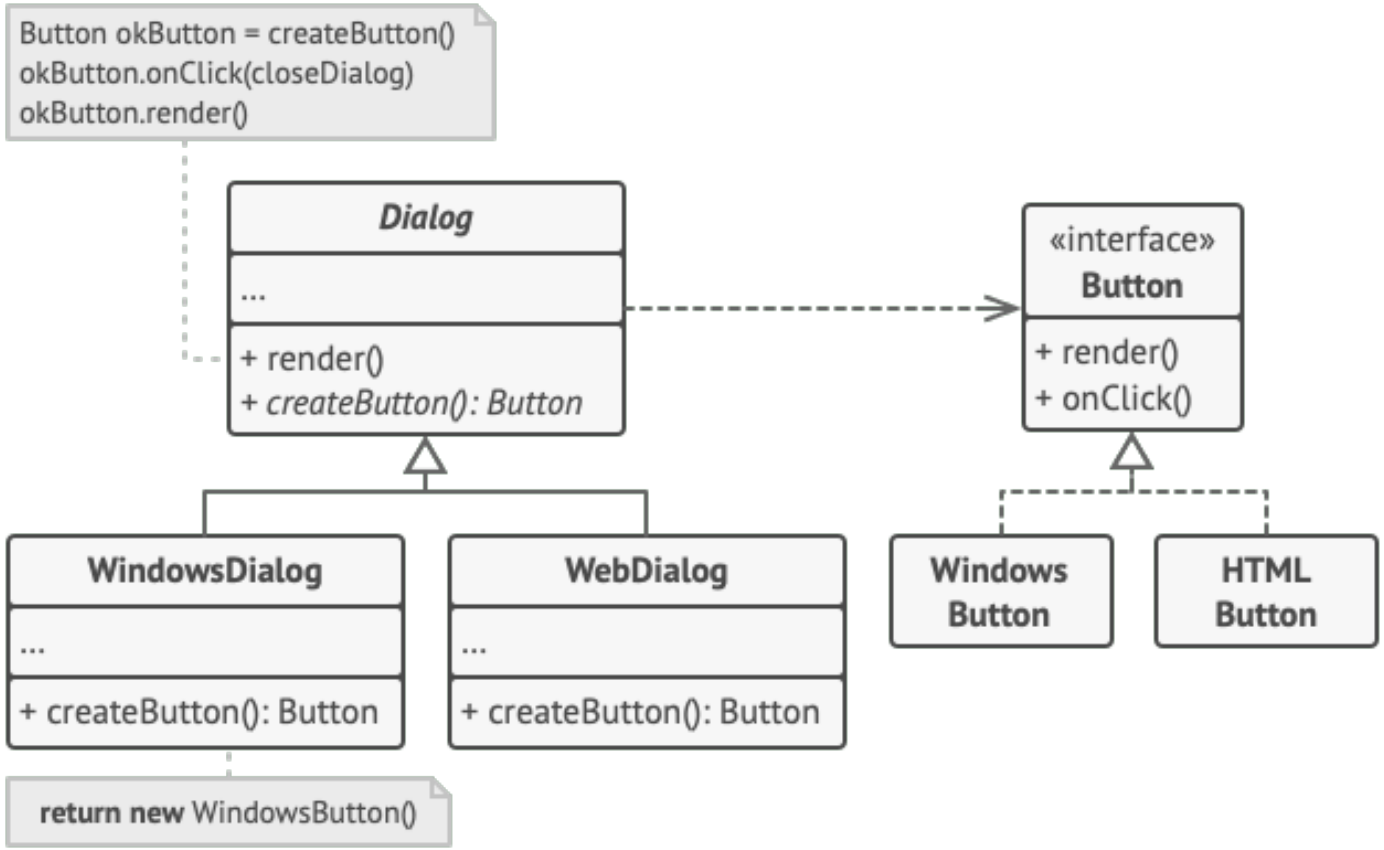


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Creación IV - Singleton

Este es un patrón de diseño creacional que permite asegurar que una clase tenga una única instancia, al mismo tiempo que proporciona un punto de acceso global a dicha instancia.

La intención más habitual de tener un solo objeto instancia de una clase es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo. Este objeto puede ser relacionado desde cualquier otra parte de código.

A nivel de programación, la particularidad de este patrón es que los constructores del objeto son privados. Para la creación y acceso a información se usan componentes de tipo estático (similar a System.out.print(), donde System es una clase).

Patrones de Creación V - Singleton

Problema: Gerencia de una Empresa

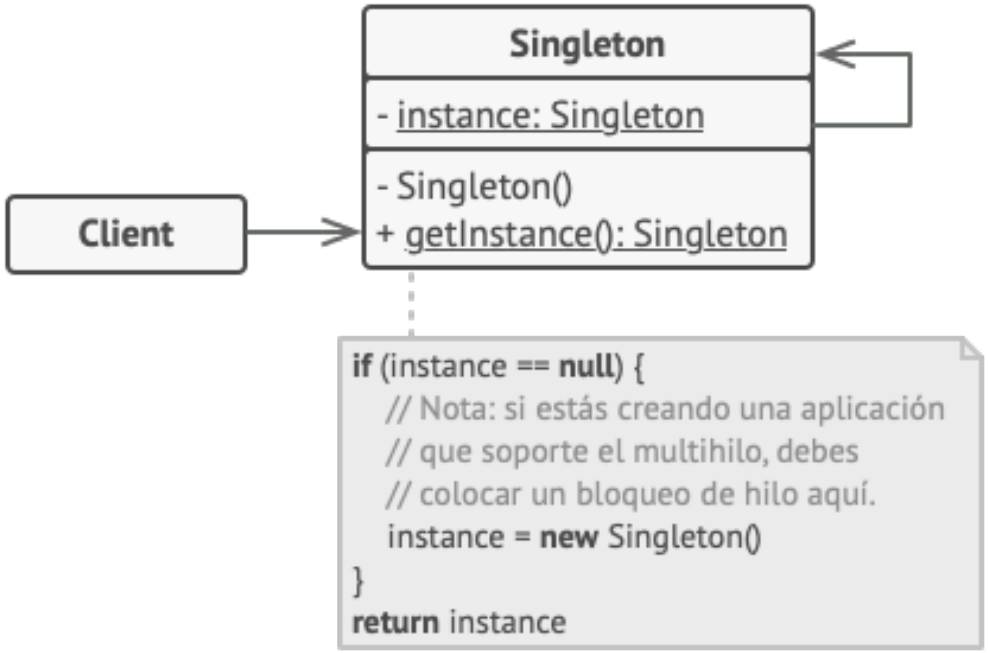


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Creación VI - Otros Patrones

- **Abstract Factory:** Permite la creación de un conjunto o familia de objetos relacionados sin necesidad de especificar en ningún momento cuáles son las implementaciones concretas.
- **Builder:** Separa la creación de un objeto complejo de su estructura, de tal forma que si se debe trabajar con objetos cuya definición es compleja, se hace la creación de distintas reprentaciones de un objeto usando el mismo código de construcción.
- **Prototype:** Permite la creación de objetos basados en *plantillas*, es decir, un nuevo objeto se crea a partir de la clonación de otro objeto sin generar dependencia de las clases.

Patrones de Estructura

Los **Patrones de Estructura** explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras. Son patrones que facilitan la modelización de software, especificando la forma en la que unas clases se relacionan con otras.

De manera más específica, el concepto de herencia es usado para construir interfaces y definir maneras de componer objetos para obtener cada vez más funcionalidades, pero manteniendo de manera simple el diseño de las relaciones.

Patrones de Estructura - Decorator(I)

Es un patrón de diseño estructural que permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores (*wrappers*) especiales que contienen estas funcionalidades.

Es útil cuando se requiere asignar funcionalidades adicionales a objetos durante el tiempo de ejecución, pero sin querer afectar la base de dichos objetos. Esto brinda flexibilidad y adaptabilidad en tiempo real.

En algunos casos puede ser usado como un reemplazo a la herencia, especialmente cuando no sea sencillo el diseño bajo herencia, o cuando se deban agregar funcionalidades a un mismo objeto en distintos instantes de la ejecución.

Patrones de Estructura - Decorator(II)

Problema: Manejo de Múltiples Notificaciones

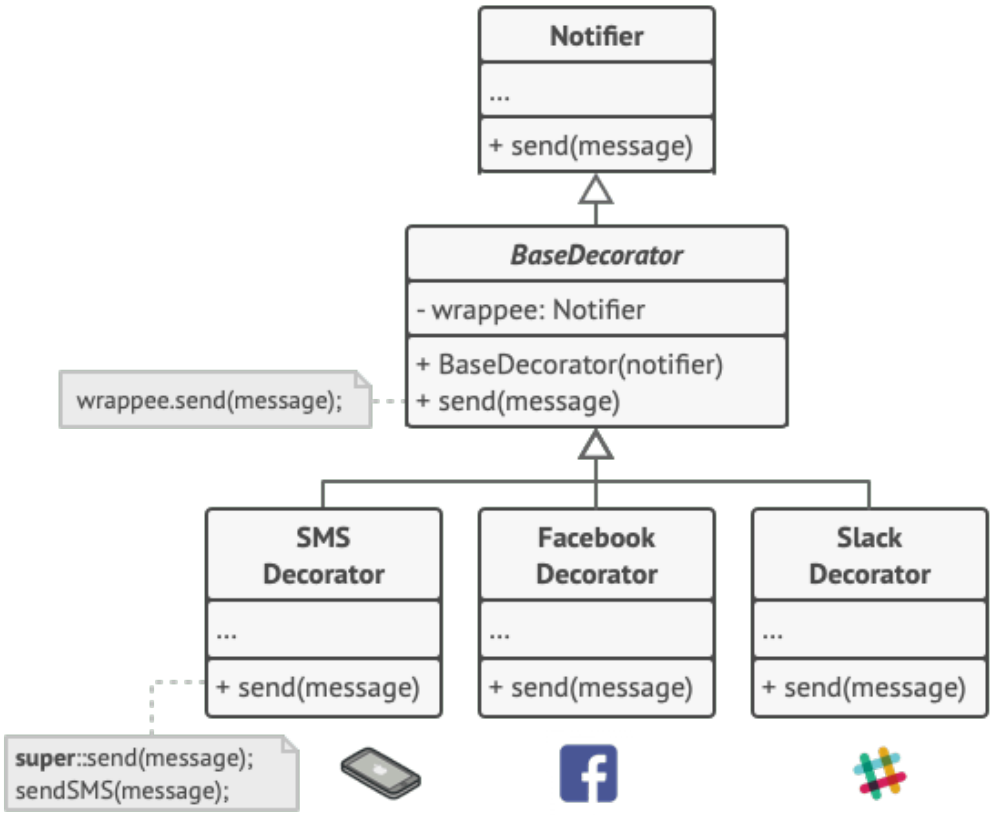


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Estructura - Facade(I)

Es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases, facilitando el acceso y interconexión con las mismas.

En este sentido, la fachada (*Facade*) permite el acceso sencillo a las funciones que requiere el usuario, no necesariamente a todas las funciones del sub-sistema cubierto. Esto determina un conjunto limitado de funcionalidades a la medida del usuario.

También puede ser utilizado para dividir grupos de funcionalidades como componentes abstractos, disminuyendo el acoplamiento, y generando un diseño de capas sencillo de comprender.

Patrones de Estructura - Facade(II)

Problema: Centro Automatizado de Atención al Cliente

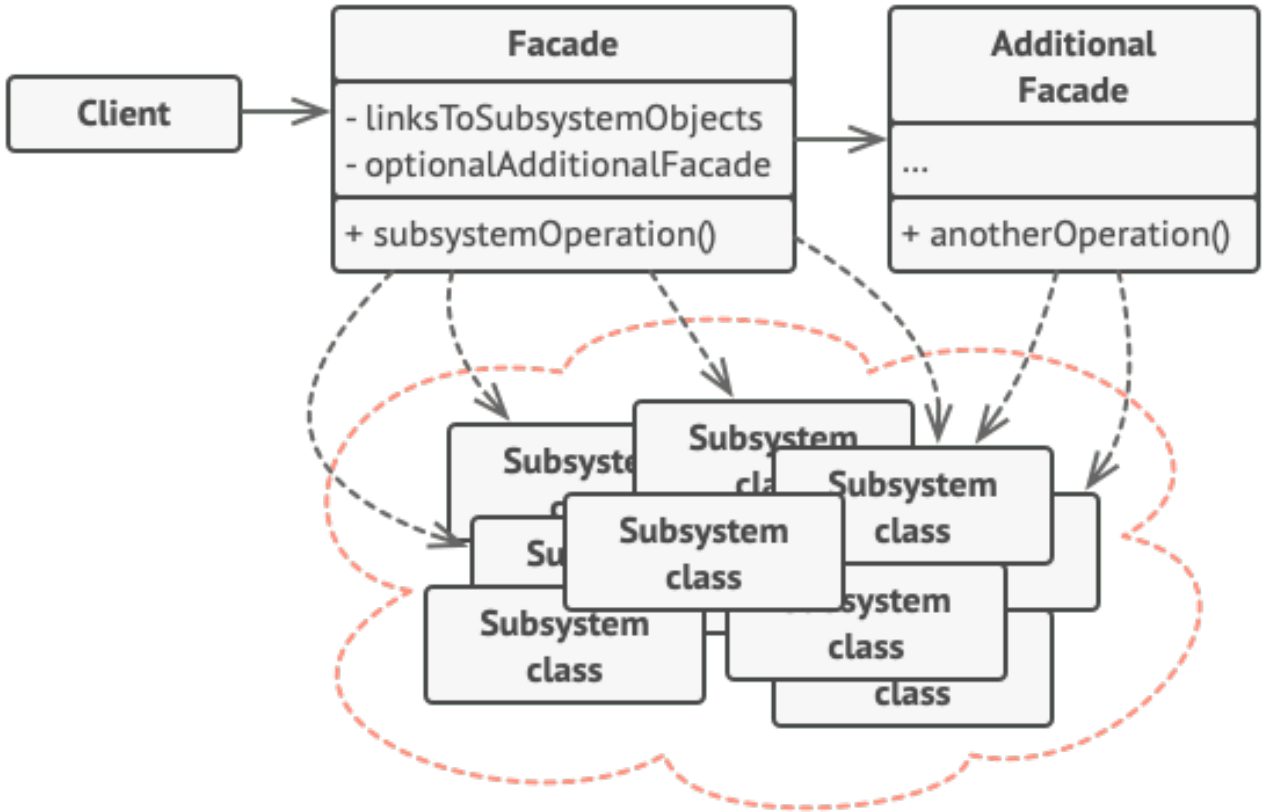
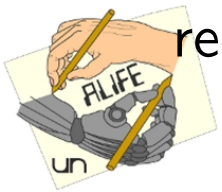


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Estructura - Otros Patrones

- **Adapter:** Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el que se comunican e interactúan.
- **Bridge:** Desacopla relaciones a jerarquías abstracción - implementación, para que las dos puedan evolucionar de forma independiente.
- **Composite:** Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una misma interfaz.
- **Flyweight:** Una gran cantidad de objetos comparte un mismo objeto con propiedades comunes con el fin de ahorrar memoria.
- **Proxy:** Es una clase que funciona como interfaz hacia cualquier otra cosa: una conexión a internet, un archivo en disco, o cualquier otro recurso que sea costoso o imposible de duplicar.



Patrones de Comportamiento

En este grupo se encuentran la mayoría de los patrones. Los **Patrones de Comportamiento** se usan para gestionar algoritmos, relaciones y responsabilidades entre objetos.

En estos patrones de diseño la interacción entre los objetos debe ser de tal manera que puedan comunicarse fácilmente entre sí, pero sin perder la independencia de cada objeto. Su objetivo principal es aumetar la flexibilidad en la comunicación entre objetos.

Patrones de Comportamiento - Observer(I)

Observer es un patrón de diseño de comportamiento que permite definir un mecanismo de suscripción para notificar a varios objetos (observadores) sobre cualquier evento que le suceda al objeto que están observando.

Se recomienda utilizarlo cuando los cambios en el estado de un objeto pueden tener efecto en el estado de otros objetos, en donde estos objetos perteneces a un mismo grupo que puede cambiar de manera dinámica.

Los observadores son independientes, así que cada uno puede tener un mecanismo o parámetros de actualización particulares. Sin embargo, no existe un orden pre-establecido para generar las notificaciones, debido a que la lista de observadores puede cambiar en tiempo de ejecución.

Patrones de Comportamiento - Observer(II)

Problema: Promociones de la Tienda Virtual Unaleña

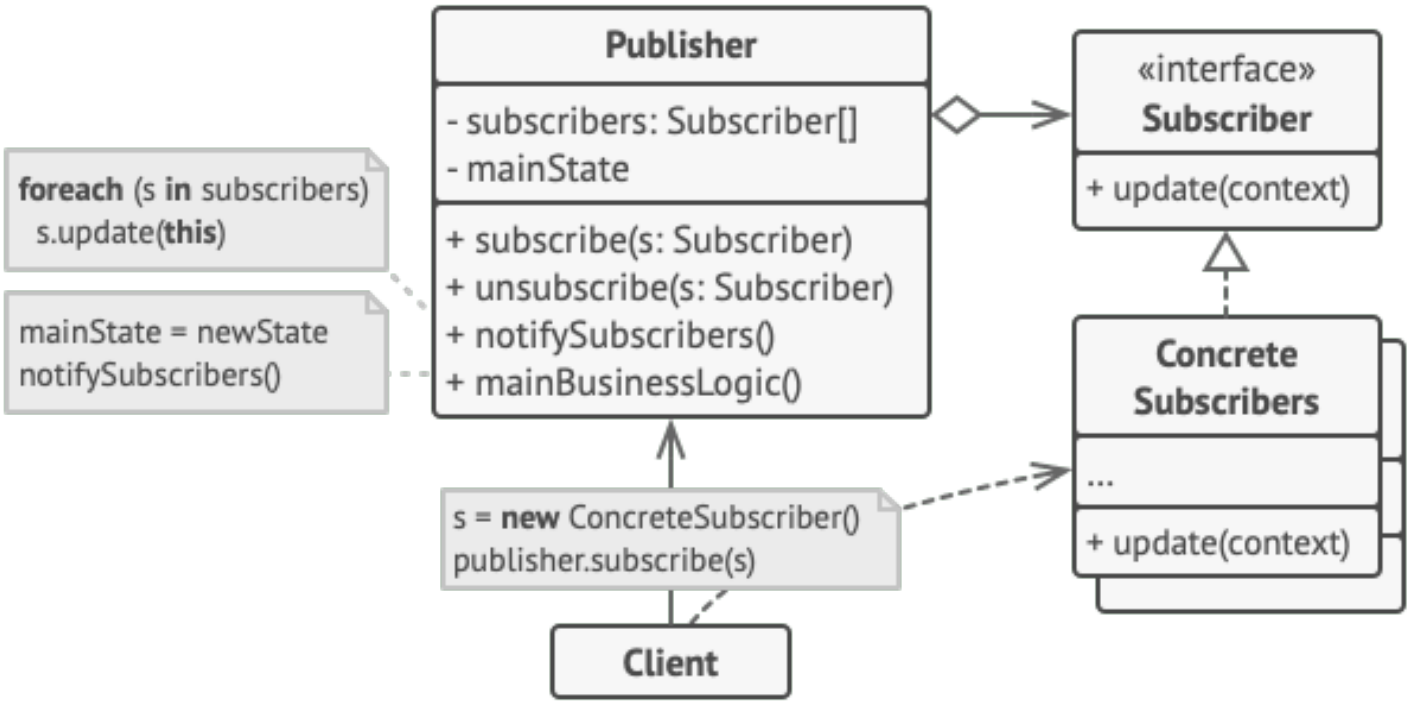


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Comportamiento - Strategy(I)

Es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada (estrategias) y hacer sus objetos intercambiables.

Desde una clase base (contexto) se permite la selección del algoritmo que ejecute cierta acción en tiempo de ejecución dependiendo de la parametrización realizada. Esto quiere decir que el contexto delega la tarea a una estrategia de acuerdo a la solicitud del cliente.

Una de sus grandes ventajas es que aisla la lógica del negocio de los detalles de implementación de los algoritmos, que a la final no son importantes para el contexto. Al contexto le interesa solamente el resultado de la ejecución de los algoritmos.

Patrones de Comportamiento - Strategy(II)

Problema: Obtener la Mejor Ruta con Diferentes Medios de Transporte

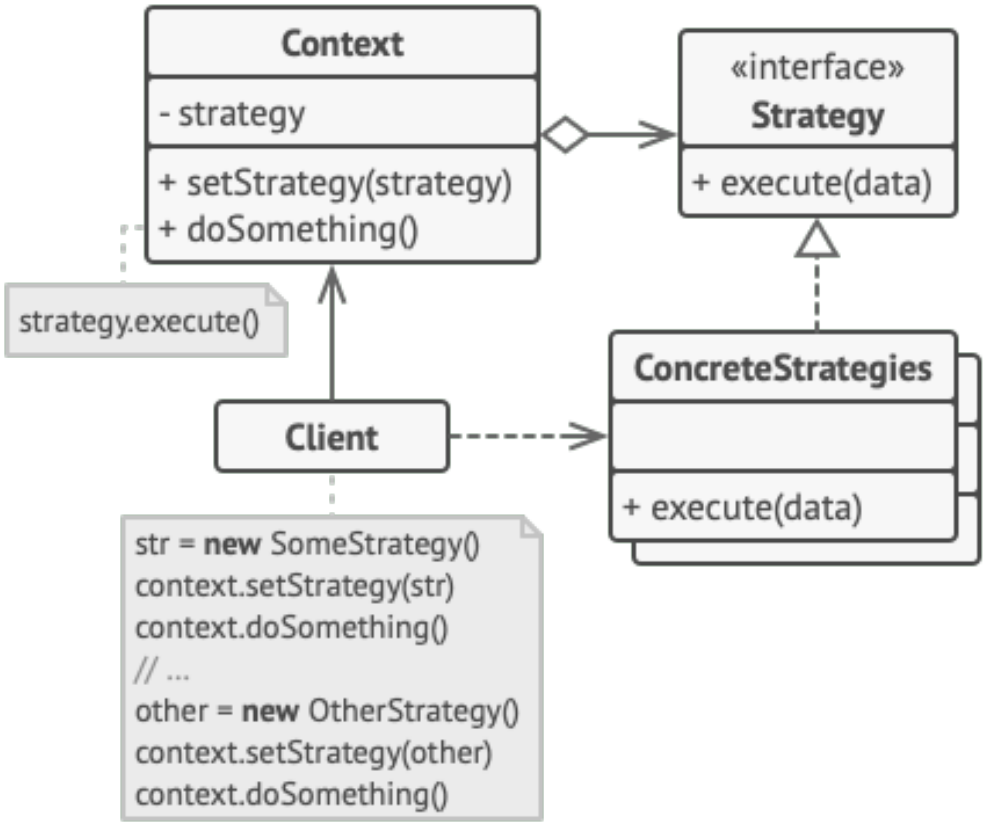


Figure: Tomado de RefactoringGuru. Detalles del ejemplo aquí.

Patrones de Comportamiento - Otros Patrones

Algunos de los más conocidos (por citar unos pocos) son:

- **Command:** Son objetos que encapsulan una solicitud y los parámetros que necesitan para ejecutarse.
- **Interpreter:** Define una representación para una gramática así como el mecanismo para evaluarla. El árbol de sintaxis del lenguaje se suele modelar mediante el patrón Composite.
- **Iterator:** Se utiliza para poder movernos por los elementos de una colección sin necesidad de exponer su implementación específica.
- **Template Method:** Especifica el esqueleto de un algoritmo, permitiendo a las subclases sobreescribir algunos pasos del algoritmo.

Agenda

Agenda

Colecciones en Java

Las colecciones pueden ser básicamente interpretadas como un **conjuntos de objetos** (la definición del objeto es libre, pero toda la colección debe ser del mismo tipo de objeto).

A diferencia de los arreglos, las colecciones permiten redimensionamiento y modificación en tiempo de ejecución. Las colecciones son usadas para **almacenar**, **recuperar**, **manipular** y comunicar conjuntos de datos.

Generalmente, representan estructuras de datos que forman un grupo natural, como una mano de póker, una carpeta de correo ó un directorio telefónico.

Sugerencias de Implementación

- Cuando se deba trabajar con fuentes de datos de tamaños variables, con estructuras de datos claramente definidas, y necesidad de modificar el contenido de la colección constantemente durante el tiempo de ejecución.
- Cuando se requiere la implementación de algoritmos avanzados para búsqueda y procesamiento de bloques de información, facilitando la manipulación de información y disminuyendo el consumo de recursos computacionales.
- Cuando se deben trabajar estructuras de datos no lineales, como conceptos basados en árboles, hashing, y grafos, abriendo el espectro para resolver problemas de manera eficiente y exitosa.

Agenda

Interface List

Un **List**, o simplemente lista, es una colección que ordena sus elementos en una secuencia lineal indexada, permite duplicados, completamente mutable.

Generalmente, quien usa la lista requiere un control sobre el lugar donde se insertará cada componente de la lista, y poder acceder al componente por medio de un **índice** (entero) que acompaña al dato.

De esta interface derivan dos implementaciones como ArrayList, LinkedList y Vector. Aunque estas implementaciones son similares, varian en algunos detalles internos que generan diferencias que es importante tener en cuenta para seleccionar con cual trabajar.

List - ArrayList

Un **ArrayList** es muy similar a los arreglos comunes, la ventaja de ArrayList sobre un array es que es expandible, es decir que crece a medida que se le añaden elementos (el tamaño de un arreglo es fijo desde su creación).

Uno de sus puntos fuertes es que el tiempo de almacenamiento y acceso a un elemento es bastante rápido (más que Vector y LinkedList). Sin embargo, en términos de manipulación de los elementos que contiene, no tiene tan buen desempeño como otras implementaciones de listas.

Para más información vaya a esta documentación.

List - LinkedList

En ésta implementación los elementos son mantenidos en una serie de nodos enlazados entre sí como eslabones de una cadena. Cada uno de estos nodos apunta tanto a su antecesor y al elemento que le sigue.

No existe información en los nodos que tenga algo que ver con la posición en la lista. De modo que para obtener un elemento, es necesario recorrer la lista desde el primer nodo e ir avanzando hasta el nodo deseado.

Esta implementación es más lenta que otras en términos de velocidad de almacenamiento y acceso a datos. En cuanto a la manipulación es los mismos, es considerablemente más rápida.

Para más información vaya a esta documentación.

List - ArrayList Vs. LinkedList

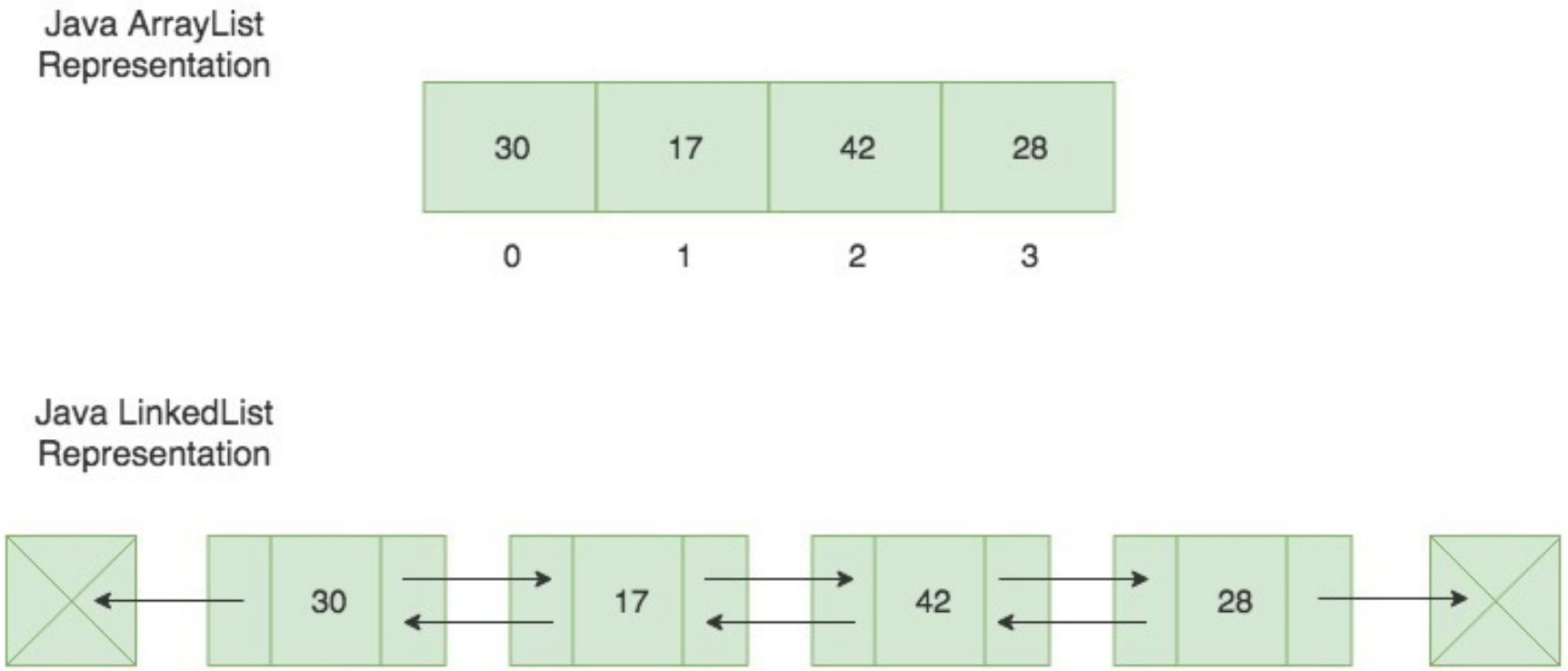


Figure: Tomado de CalliCoder

List - Vector

Similar a ArrayList, está basado en un arreglo dinámico, lo que le permite expandirse en tiempo de ejecución. Esta implementación de lista varia un poco en el nombre de algunas operaciones, pero su comportamiento es similar al de otras inmplementaciones.

Vector es una implementación sincronizada, esto quiere decir que no admite múltiples procesos accediéndolo al mismo tiempo. Tiene buen desempeño en términos de la expandibilidad, pero suele ser más lento que otras implementaciones de listas.

Para más información vaya a esta documentación.

Map

Un **Map** es una interface que representa lo que en otros lenguajes se conoce como *diccionario*, y que se suele asociar a la idea de tabla hash (aunque no se implemente necesariamente con esa técnica).

Un Map es un **conjunto de valores**, con el detalle de que cada uno de estos valores tiene un objeto extra asociado conocido como clave o key, la cual permite acceder a dichos valores.

Para más información vaya a esta documentación.

Map - HashMap

Esta implementación almacena las claves en una **tabla hash**. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones.

Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la *función hash* disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

Para más información vaya a esta documentación.

Map - TreeMap

Esta implementación almacena las claves ordenándolas dentro de una estructura de datos de árbol. Es **más lento** que HashMap en el almacenamiento, pero es más rápido en recuperación de la información.

Las claves almacenadas deben implementar una interfaz comparable. Esta implementación garantiza un rendimiento de $log(n)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

Para más información vaya a esta documentación.

Problemas varios I

Problemas

1. Dada una lista de números, eliminar los duplicados e imprimir la lista procesada.
2. Dadas dos listas de números, las cuales pueden ser de distinto tamaño, e imprimir los números se encuentran en ambas listas.
3. Dada una lista de objetos, imprimir la inversa de dicha lista.