RESUMEN IO

Tipo de datos

Consiste en una colección de valores, con una representación asociada, su estructura de datos y un conjunto de operaciones definidas. Sobre ellos, al conjunto de valores se lo llama dominio del tipo.
Ej: tipo de dato int representa valores -2,-1, 0, 1, 2 …, sobre los cuales están definidas las operaciones suma, resta multiplicación, etc

Tipo Abstracto de Datos

Descripción abstracta y formal que define un número de operaciones aplicables, su sintaxis (modo en que puede invocarse cada operación, donde se define el nombre, el dominio y el rango de operaciones) y su semántica (cómo opera cada operación de acuerdo a un conjunto de axiomas). Esta noción matemática define un tipo de datos.

Ejemplo TAD Natural
Nombre: Natural
Dato: secuencia de dígitos precedida por un signo positivo.
Operaciones:
Crear-> Natural
=: (Natural, Natural) -> Booleano(verdadero, falso)
>: (Natural, Natural) -> Booleano(verdadero, falso)
+: (Natural, Natural) -> Natural
….

Sintaxis

Axiomas y Operaciones
0(cero) es un numero natural.
Si n es un numero natural, también lo es S(n), su siguiente.
(S(n)= 0) = falso
(n + 0) = n
si S(n) = S (m)=> n = m

Semántica

Los TADs se definen por su comportamiento exterior (operaciones), no por su estructura.
Todo TAD debe cumplir con 2 ppios:

Ocultación de la info (encapsulación): solo puede accederse a los valores del tipo que define o modificarlos mediante las operaciones abstractas definidas sobre ellos. La encapsulación hace referencia al ocultamiento de la info y a la capacidad para expresar al unidad formada por los valores y las operaciones

Abstracción de Datos: se separan las propiedades lógicas de los datos de su representación o implementación

Tipos abstractos de datos y Orientación a Objetos

Los TADs permiten la creación de instancias con propiedades y comportamiento bien definidos. En POO los TADs son referidos como clases. Por lo tanto una clase define las propiedades de objetos, los que son instancias en un entorno orientado a objetos.

Programación Orientada a Objetos

Según Booch:

La programación Orientada a Objetos es un método de implementación en el cual los programas están organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas vía relaciones de herencia.

Diseño Orientado a Objetos

Diseño orientado a objetos es un método de diseño que guía el proceso de descomposición orientado a objetos y define una notación para expresar tanto los modelos lógico (estructura de clase y objeto) y físico (arquitectura de módulo y proceso) (tanto estáticos como dinámicos).

Programación Orientada a Objetos

Podríamos decir por ahora que desarrollar con objetos es modelar el problema que tenemos que resolver utilizando objetos. La realidad tiene ciertas características interesantes que debemos poder simular si queremos tener un buen modelo. Una de esas características es la interacción entre los elementos de la realidad. Por ejemplo, el gerente de contabilidad le pide al facturador que facture y el facturador le indica a la factura a quién está dirigida. Estos elementos “colaboran” entre sí para llevar adelante el proceso de facturación.

Esta interacción se modela en el paradigma de objetos por medio de “mensajes”. Los objetos se comunican entre sí utilizando “mensajes” (que nada tienen que ver con mails, mensajes de red, etc.) que representan la comunicación o interacción de elementos de la realidad.

Como todo proceso de comunicación hay un emisor y un receptor. Por lo tanto el objeto emisor le indica por medio de un mensaje al objeto receptor qué debe hacer. Definimos entonces que un programa en el paradigma de objetos es “un conjunto de objetos que colaboran entre sí enviándose mensajes”

Otra característica de la realidad consta en definir las responsabilidades más allá de cómo se llevan a cabo. Al gerente de contabilidad no le interesa si Juan (el facturador) factura escribiendo primero a quién va dirigida la factura y luego el detalle de la misma. La manera en que Juan factura no le importa, siempre y cuando realice correctamente su trabajo.

Esta característica permite al gerente de contabilidad despreocuparse sobre cómo se está realizando la facturación y ocuparse de sus propias tareas.
Esta característica en el paradigma de objetos se llama “encapsulamiento” y es la que permite separar qué hacen los objetos (responsabilidad) de cómo lo hacen (implementación).
Por último, otra característica que debemos poder simular en nuestro modelo es delegar las responsabilidades a alguien sin preocuparnos quién sea ese alguien. Al gerente de contabilidad no le importa si el facturador es Juan o Pedro, le importa poder pedirles que facturen y que la facturación se realice. Esa es la responsabilidad del facturador.
Esta característica es muy importante, permite despreocuparse sobre quién hace el trabajo y solamente asegurarse de tener buenos “colaboradores” en quienes delegar tareas. Debemos estar acompañados de colaboradores si queremos llevar adelante ciertas tareas, y lo que nos interesa de esos colaboradores es que cumplan con sus responsabilidades correctamente, más allá de cómo lo hagan.
Esta característica de la realidad no puede escapar de un paradigma que se jacta de modelarla, y por supuesto no se escapó. Esta característica en objetos se denomina “polimorfismo” y justamente permite al objeto emisor del mensaje despreocuparse de quién es exactamente su colaborador, sólo le interesa que sea responsable de llevar adelante la tarea que le encomienda a través del “mensaje”.

Resumiendo
Desarrollar con objetos es modelar la realidad utilizando objetos que deben colaborar entre sí enviándose mensajes. Estos objetos sólo se preocupan de que sus colaboradores sepan qué hacer (responsabilidad), no cómo lo hacen, y no tienen problemas sobre quiénes son sus colaboradores (polimorfismo) siempre y cuando respondan sus mensajes.

En el paradigma orientado a objetos, el marco conceptual es el modelo de objetos. Hay cuatro elementos principales en este modelo:

Abstracción Encapsulación Modularidad Jerarquía

Abstracción

Una abstracción es una descripción simplificada de un sistema que enfatiza algunos de sus detalles o propiedades mientras suprime otros. Una buena abstracción es la que enfatiza detalles que son significantes al lector, y suprime los que no lo son.
La abstracción enfoca las características esenciales de algún objeto, relativa a la perspectiva del observador.
Una abstracción se enfoca sobre una vista externa del objeto, y sirve para separar su comportamiento esencial de su implementación.

Encapsulación

La abstracción de un objeto debería preceder a su implementación. Una vez que la implementación es seleccionada, esta debería tratarse como un secreto de la abstracción y ocultársela a la mayoría de los clientes.

Abstracción y encapsulamiento son conceptos complementarios: la abstracción enfoca la vista externa de un objeto, y el encapsulamiento (ocultamiento de la información) previene que los clientes vean la parte interna, donde el comportamiento de la abstracción es implementada.

Según Booch: "El encapsulamiento es el proceso de ocultar todos los detalles de un objeto que no contribuyen a sus características esenciales.

Modularidad

Si bien el acto de particionar un programa en componentes individuales reduce en algún grado la complejidad, una razón más poderosa para realizarlo es que esto crea un número de contornos bien documentados dentro del programa.

Jerarquía

La abstracción es sin duda algo bueno, pero salvo en aplicaciones triviales, encontramos que hay mas abstracciones diferentes que las que podemos comprender a un dado momento.

El encapsulamiento nos ayuda a manejar esta complejidad ocultando la parte interna de estas abstracciones. La modularidad también ayuda, dándonos una manera de agrupar lógicamente abstracciones relacionadas. Esto no es suficiente. Un conjunto de abstracciones frecuentemente forman una jerarquía, e individualizándolas en nuestro diseño, simplificaremos el entendimiento del problema.

La jerarquía es un ordenamiento de abstracciones.

Las dos jerarquías más importantes son: estructura de clases (jerarquía tipo de) y la estructura de objetos (jerarquía parte de).

¿Qué es un Programa Orientado a Objetos?

Un conjunto de **objetos** que **colaboran** enviándose **mensajes.** Por lo tanto:

- Sólo hay objetos
- Lo único que pueden hacer es enviar y recibir mensajes

Si querés que algo se haga necesitás un objeto que lo haga, y otro objeto que le envíe un mensaje.

Objeto (def. de Wikipedia)

Los objetos son entidades que combinan *estado*, *comportamiento* e identidad:

- El *estado* está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
- El *comportamiento* está definido por los procedimientos o *métodos* con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La *identidad*es una propiedad de un objeto que lo diferencia del resto

Encapsula ***funcionalidad*** e ***información***:

- Retiene cierta ***información***.
- Sabe como realizar ciertas ***operaciones***

A diferencia del diseño estructurado aquí las "***operaciones"*** y la "***información"*** están juntas y sólo se puede acceder a esa información a través de esas operaciones (**encapsulamiento**).

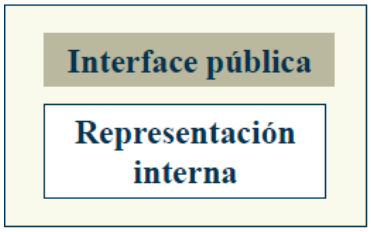
Comportamiento
- El comportamiento indica qué sabe hacer el objeto, es decir sus responsabilidades.
- Se especifica a través del conjunto de mensajes que puede recibir el objeto.

Implementación
- La implementación indica **cómo** hace el objeto para responder a sus mensajes.
- Es especificado mediante:
    - Un conjunto de colaboradores.
    - Un conjunto de métodos.
- Es privado del objeto. Ningún otro objeto debe acceder

Método
- Lo único que puede hacer un objeto cuando recibe un mensaje es enviar mensajes (colaborar) a otros objetos (colaboradores).
- Entonces, un método es simplemente el conjunto de "colaboraciones" que lleva a cabo un objeto para responder un mensaje.
- Al recibir un mensaje, un objeto lleva a cabo la operación mediante la ejecución de un método.
- El método es el algoritmo particular con el que el objeto realiza dicha operación.

Un método está asociado a un mensaje. Generalmente con el mismo nombre.

¿Todo es público?
- La representación interna de un objeto es su lado "privado", sólo tenemos acceso a aquellas partes de su estado que el objeto revela mediante su interface pública



Estado interno de un objeto
- La información almacenada dentro de un objeto conforman su **estado interno**.
- El estado interno de un objeto puede ser cambiado sólo a través de las operaciones provistas por el objeto para dicho fin.

Encapsulamiento

- El encapsulamiento es el proceso de agrupar dentro de un objeto "**colaboradores"** y "**comportamiento"**
- Es una de las principales claves para conseguir software confiable.
- El encapsulamiento permite que los cambios hechos en los programas sean fiables con el menor esfuerzo

*Una de las premisas de la programación orientada a objetos es tratar de **no** violar el Encapsulamiento*

Ocultamiento de la Información

- El estado interno conforma el lado **privado** de un objeto.
- El lado privado maneja la información que es necesaria para su funcionamiento interno, pero innecesaria para los demás objetos
- En él se especifica:
    - ***Cómo*** lleva a cabo los requerimientos que le hacen otros objetos
    - ***Cómo*** representa la información que mantiene
- La manera en que el objeto lleva a cabo estos requerimientos o trata su información "no es asunto" de los demás objetos.
- De esta manera los objetos pueden cambiar el modo de realizar ciertas operaciones o representar cierta información sin afectar el resto del sistema.

Estructuración de responsabilidades

- Un objeto sabe cumplir sólo ***su*** rol y el de ningún otro objeto dentro del sistema.
- No existe información fuera de objetos.
- No existen operaciones fuera de objetos.
- El diseño orientado a objetos, estructura responsabilidades mediante las preguntas: ¿qué puede hacer este objeto? y ¿qué conoce este objeto?

Clases

- Los objetos dentro de un sistema no son completamente distintos uno del otro.
- Distintos objetos pueden comportarse de una manera muy similar.

*Se dice que los objetos que comparten el mismo comportamiento pertenecen a la misma **clase**.*

Es la declaración o abstracción de un objeto cuando se programa según el paradigma de orientación a objetos.

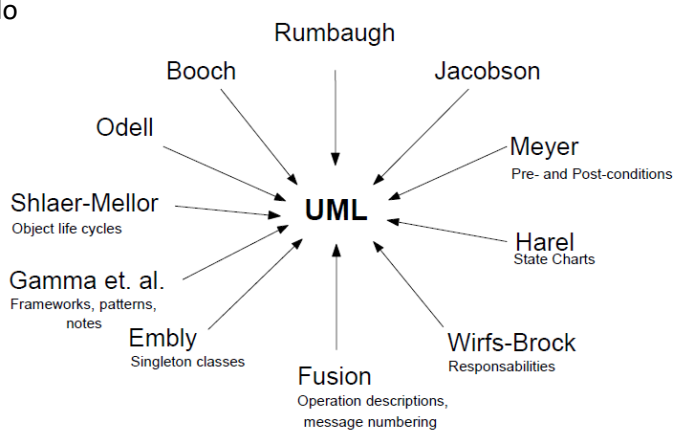
Instancias

- Los objetos que se comportan de la manera descripta en una clase son llamados **instancias** de esa clase.
- Todo objeto es instancia de alguna clase.
- Una instancia de una clase se comporta de la misma manera que las demás instancias de esa clase.
- Almacena su información en variables de instancia.

UML – Lenguaje Unificado de Modelado

EL UML es la creación de G. Booch, J. Rambaugh e I. Jacobson. En la década de los ´80 cada uno diseñó su propia metodología para el análisis y diseño orientado a objetos. Sus metodologías predominaron sobre las de sus competidores. A mediados de los ´90 intercambian sus ideas y deciden trabajar en conjunto UML -Lenguaje Unificado de Modelado

UML aglutina otros enfoques:



Ventajas de la unificación

- Reunir los puntos fuertes de cada método
- Idear nuevas mejoras
- Proporcionar estabilidad al mercado
- Eliminar confusión en los usuarios

UML y el modelado

Lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software

Características de UML

- UML es independiente del lenguaje de implementación.
- UML esta pensado para poder ser implementado en varios lenguajes
- Provee una base formal para el entender el lenguaje de modelado

Compuesto por elementos gráficos, que se combinan para conformar diagramas. Debido a que UML es un lenguaje, cuenta con reglas para combinar dichos elementos.

Diagramas:

- De clases
- De objetos
- De casos de uso
- De estados
- De secuencias
- De colaboraciones

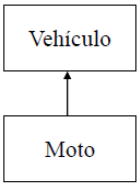
- De componentes
- De distribución

Un modelo UML indica *qué* hará un sistema, y no *cómo* se implementará.

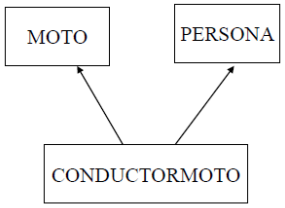
Herencia

En la POO, la herencia permite que unos objetos puedan basarse en otros existentes.
En términos de clases, la herencia es el mecanismo por el cual una clase X puede heredar propiedades de una clase Y (X hereda de Y) de modo que los objetos de la clase X tengan acceso a los atributos y operaciones de la clase Y, sin necesidad de redefinirlos.
Sin embargo, las propiedades de una clase no son necesariamente la suma de las propiedades de todas sus superclases. La herencia crea automáticamente una jerarquía de especialización-generalización.
Se suele denominar clase hija/ padre y subclase/superclase para designar al par de clase que hereda de otra.
También se suele decir que las subclases son especializaciones de la superclase y que la superclase es generalización de las subclases.
La herencia presenta dos cualidades contradictorias entre si: "Una clase hija extiende o amplia el comportamiento de la clase padre, pero también restringe o limita a la clase padre"
Suele identificarse la herencia mediante la regla "es un" o "es un tipo de".

Moto es un Vehículo, luego, la clase Moto es subclase de la clase Vehículo



NO resulta recomendable emplear la herencia si no funciona la regla "X es un Y", es mas, cuando no pueda justificarse que toda instancia de la clase X es también instancia de Y.
Ejemplo:



Clase ConductorMoto. Que hereda de Moto y Persona, es un mal uso de herencia, pues no toda instancia de ConductorMoto es una instancia de Moto.

<u>Consejos Prácticos a la hora de construir Jerarquías</u>

- Construir jerarquías estrechas y profundas. Es decir, con no demasiadas clases en cada nivel de especialización y con muchos niveles.
- Evitar que cualquier clase contenga código para averiguar la clase o el tipo del objeto, la jerarquía bien construida debe ser auténticamente polimorfa

- El diseño de una jerarquía de clases no debe considerarse aislada del tamaño de los métodos de las clases que la componen, porque cuan más extenso sea el código de un método mas difícil resultara reutilizarlo en otras clases de la jerarquía.
- Los métodos no utilizados por la mayoría de las subclases no deben ubicarse en la superclase, sino en las subclases que los usen, evitando así la propagación a lo largo de la jerarquía de métodos poco utilizados.

Clasificación

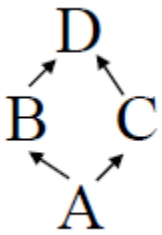
- De especialización: una clase hija es un caso particular de la clase padre. Las subclases, cuando redefinen los métodos heredados de la superclase, especializan o concretan la clase padre. Es la más usada.
- De especificación o comportamiento: una clase padre define el comportamiento de sus subclases, pero no proporciona ninguna implementación por defecto. La superclase es una clase Abstracta. Es la segunda mas usada, útil para definir el comportamiento en común de clases relacionadas.
- De implementación o Estructura: se usa la clase padre solo por su comportamiento, as pesar de que no exista relación entre la clase padre y las subclases. Se utiliza para aprovechar código ya escrito, pero no implementa una generalización. Su uso se considera incorrecto y desaconsejable.

Mas: de generalización, de extensión, de limitación, de variación, de combinación

Tipos de Herencia

- Simple: una clase sólo es subclase de una superclase
- Múltiple: una subclase admite más de una superclase.

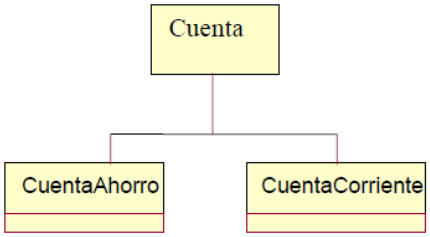
Problemas de Herencia Múltiple



- Herencia Repetida: A hereda de B y C, que a su vez heredan de D, por lo tanto, la clase A hereda 2 veces de D.
- Conflicto de Nombres: Si A hereda simultáneamente de B y C aparece un conflicto si usan el mismo nombre para algún atributo o método

Clase Abstracta

La herencia permite que existan clases que nunca sean instanciadas directamente.
La ausencia de instancias específicas es su única particularidad, para todo lo demás es como cualquier otra clase



Redefinición

- Si en una clase en particular se invoca a un método, y el método no está definido en la misma, es buscado automáticamente en las clases superiores. Sin embargo, si existieran dos métodos con el mismo nombre y distinto código, uno en la clase y otro en una superclase, se ejecutaría el de la clase, no el de la superclase.
- Por lo general, siempre se puede acceder explícitamente al método de la clase superior mediante una sintaxis diferente, la cual dependerá del lenguaje de programación empleado.

Ventajas

- Ayuda a los programadores ahorrar código y tiempo.
- Los objetos pueden ser construidos a partir de otros similares. Para ello es necesario que exista una clase base y una jerarquía de clases.
- La clase derivada puede heredar código y datos de la clase base, añadiendo código o modificando lo heredado.
- Las clases que heredan propiedades de otra clase pueden servir como clase base de otras.

Polimorfismo

Por polimorfismo entendemos aquella cualidad que poseen los objetos para responder de distinto modo ante el mismo mensaje.
En POO el mismo nombre nos sirve para todas las clases creadas si así lo queremos, lo que suele ser habitual.
El mismo nombre suele usarse para realizar acciones similares en clases diferentes.
Si enviamos el mensaje Imprímete a objetos de distintas clases, cada uno se imprimirá como le corresponda, ya que todos saben cómo hacerlo.
El polimorfismo nos facilita el trabajo, ya que gracias a él, el número de nombres de métodos que tenemos que recordar disminuye ostensiblemente.
La mayor ventaja la obtendremos en métodos con igual nombre aplicados a las clases que se encuentran próximas a la raíz del árbol de clases, ya que estos métodos afectarán a todas las clases que de ellas se deriven.

Sobrecarga

La sobrecarga puede ser considerada como un tipo especial de polimorfismo que casi todos los lenguajes de POO incluyen.
Varios métodos (incluidos los "constructores") pueden tener el mismo nombre siempre y cuando el tipo de parámetros que recibe o el número de ellos sea diferente.
Ejemplo: la clase File puede tener tantos método Write() como tipos de datos queramos escribir:
File>>Write(int i); Escribe un integer / File>>Write(long l); Escribe un long

Métodos

Los métodos son la manera de especificar cómo responden a los mensajes los objetos de una determinada clase. Cada método especifica cómo se lleva a cabo la operación para responder a un determinado mensaje.
Un método puede acceder a la estructura interna del objeto así como, también, enviarse mensajes a sí mismo o a otros objetos.
Los métodos describen, igualmente, cuál es la respuesta que recibe el emisor (el objeto que envía el mensaje).

Modificadores de Accesibilidad

Los modificadores de acceso indican la visibilidad que una variable o un método tiene. Tanto los distintos tipos posibles, como la palabra reservada para denotarlos dependen, como es lógico, de cada lenguaje. De todos modos, todos los lenguajes OOP incluyen al menos los tres siguientes:

- Públicos

Son visibles dentro y fuera de la clase sin restricción alguna. La palabra reservada más común para denotarlos es "public". Como ya hemos comentados, los datos no deben ser nunca públicos, ya que romperían el principio de encapsulación que debe seguir todo proyecto OOP.

- Protegidos

Estos miembros de la clase (ya sean datos o métodos) son visibles desde dentro de la clase y desde cualquier otra clase heredada, es decir, clases hijas o subclases. La palabra reservada más común para denotarlos es "protected" o "friend".

- Privados

Lo miembros privados son solo accesibles desde dentro de la clase donde existen. La palabra reservada más común para denotarlos es "private".

Modificadores de Contenido

Los modificadores de contenido afectan a cómo va a ser tratado el contenido de la variable. Así, una variable estática mantiene su contenido para todas las instancias de la clase que se hagan, así como para las subclases que de ella se hereden. A estas, se les llama variables de la clase, como contraposición a las variables de instancia.
Mientras que las variables de instancia se inicializan para cada nueva instancia que se haga de la clase, es decir existe una copia por cada instancia de la clase, de las variables de la clase existe una sola

instancia, independientemente del número de instanciaciones que de la clase se hagan. De este modo, todos los objetos comparten un lugar de almacenamiento común.
El ejemplo más típico de variable de la clase es un contador del número de objetos existentes de la clase.
Para ello, sólo hay que incrementar el contador desde el constructor de la clase.

Variables

Método de Clase:
Se refiere a las porciones de código asociadas exclusivamente con una clase (se los denomina entonces métodos de clase o métodos estáticos)
Un ejemplo típico de un método de clase sería uno que mantuviera la cuenta de la cantidad de objetos creados dentro de esa clase.

Métodos de Instancia:
Están relacionados con un objeto en particular, mientras que los métodos estáticos o de clase están asociados a una clase en particular. En una implementación típica, a los métodos de instancia se les pasa una referencia oculta al objeto al que pertenecen, comúnmente denominada this o self , para que puedan acceder a los datos asociados con el mismo.

Variables de instancia:
Las variables de instancia representan el estado del objeto y perduran durante toda la vida de éste. Dos objetos diferentes, aunque pertenezcan a la misma clase, pueden tener valores diferentes en sus variables de instancia.

Variables de clase:
Las variables de clase son compartidas por las instancias de una clase y sus subclases, manteniendo el mismo valor para todas las instancias. Estas variables se declaran en la definición de la clase.
Ej: Mayoría de edad, cant límite de tarjeta de crédito.

Pseudo-variables

Una pseudo-variable es un identificador que referencia a un objeto. La diferencia con las variables normales es que no se pueden asignar y siempre aluden al mismo objeto.
Esto es, el valor de una pseudo-variable no puede modificarse con una expresión de asignación.

"Pseudo-variables constantes"
- Nil / null. "Referencia a un objeto usado cuando hay que representar el concepto de "nada" o de "vacío". Las variables que no se asignaron nunca, referencian a nil."
- True. "Referencia a un objeto que representa el verdadero lógico."
- False. "Referencia a un objeto que representa el falso lógico."

“Pseudo-variables no-constantes”

- Self/ this . “Referencia al receptor del mensaje.”
- Super. “Referencia al receptor del mensaje, pero indica que no debe usarse la clase del receptor en la búsqueda del método a evaluar. Se usa, sobre todo, cuando se especializa un método en una subclase y se quiere invocar el método de la superclase.” Al contrario que this, super permite hacer referencia a miembros de la clase padre (o a los ancestros anteriores, que no hayan sido ocultados por la clase padre) que se hayan redefinido en la clase hija. Si un método de una clase hija redefine un miembro –ya sea variable o método–de su clase padre, es posible hacer referencia al miembro redefinido anteponiendo super.

Constructores

Para poder utilizar un objeto, previamente tenemos que crearlo, lo hacemos mediante el constructor de la clase y puede haber más de un constructor según la sobrecarga de parámetros.
Para ello, dependiendo del lenguaje existen dos procedimientos:

a) Utilizando un método especial (normalmente con la palabra reservada "constructor"). Este método nos devuelve un objeto nuevo de esa clase. En este caso, a los métodos constructores se les suele llamar New().

b) Utilizando un operador especial que el lenguaje proporciona y que normalmente se llama "new". Es este caso, el constructor o los constructores son notados de una forma especial: en Java, por ejemplo, se notan con el nombre de la clase y no devuelven ningún Tipo.
Así, para crear un objeto de la clase hombre, llamado Juan, escribiremos lo siguiente:

```
1. Hombre hmrJuan = Hombre.New();
2. Hombre hmrJuan = new Hombre();
```

Le estamos diciendo al método constructor que nos devuelva un nuevo objeto.

Constructor argumento-cero

Es aquel que no recibe ningún parámetro. Es también importante, el concepto de "constructor por defecto". Muchos lenguajes de POO, permiten definir una clase sin crear un constructor para la clase. El lenguaje, entonces, utiliza el constructor por defecto (interno al lenguaje) para crear objetos de esa clase. Este método interno, normalmente se limita a reservar el espacio de memoria necesario para almacenar los datos del objeto, pero estos datos no están inicializados o no lo están correctamente, ya que el constructor por defecto no puede saber qué valores son los apropiados para los datos de la clase. En cualquier caso, la misión del constructor es construir adecuadamente el objeto, es decir, cuando el constructor haya terminado su trabajo, el objeto tiene que estar listo para ser usado.

Método plantilla (Template Method)

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

Usando el Template Method, se define una estructura de herencia en la cual la superclase sirve de plantilla de los métodos en las subclases. Una de las ventajas de este método es que evita la repetición de código, por tanto la aparición de errores.
Se vuelve útil cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras.

Método abstracto

Es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código).
Una clase abstracta es una clase que tiene al menos un método abstracto. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

Introd. a Colecciones

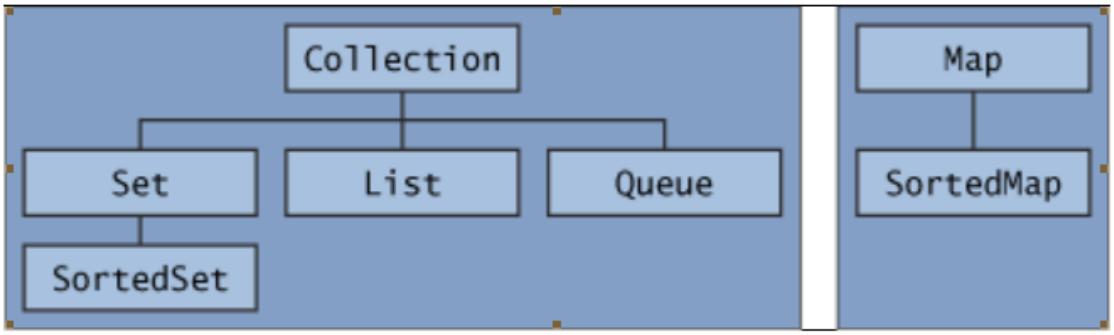
Una Colección es toda estructura que me permite almacenar objetos.
Se puede recorrer (o "iterar") y se puede saber el tamaño.

- Una *colección* es simplemente un objeto que agrupa varios elementos en una sola unidad.
- Las colecciones son utilizadas para almacenar, recuperar y manipular los datos agregados.
- Por lo general, representan elementos de datos que forman un grupo natural, ej: una mano de póker (una colección de cartas).

Un *framework* (armazón) de colecciones es una arquitectura unificada para representar y manipular colecciones. Todos los *frameworks* de las colecciones contienen:

- Interfaces: Estos son los tipos de datos abstractos que representan colecciones. Las interfaces permiten manipular las colecciones de forma independiente de los detalles de su representación. En lenguajes orientados a objetos, las interfaces generalmente forman una jerarquía. Una interface declara un marco abstracto de comportamiento.
- Implementaciones: Estas son las implementaciones concretas de las interfaces de Collection. En esencia, son estructuras de datos reutilizables.
- Algoritmos: Estos son los métodos que realizan cálculos útiles, tales como la búsqueda y la selección, en los objetos que implementan interfaces de la colección. Los algoritmos se dice que son *polimórficos,* es decir, el mismo método se puede utilizar en muchas implementaciones dependiendo de las diferentes de la interfaces que hayamos elegido. En esencia, los algoritmos son funcionalidad reutilizable.

La Figura muestra la jerarquía del núcleo de interfaces de la JCF. Estas interfaces permiten manipular de forma independiente los detalles de su representación.



Set en un tipo especial de Colecction, SortedSet es un tipo especial de Set, y así sucesivamente. Notar que la jerarquía se compone de dos árboles.

- Las colecciones se organizan en:
    - Interfaces: Manipulan los datos independientemente de los detalles de implementación.
    - Clases: Implementan las interfaces.

Las interfaces y las clases están relacionadas en un armazón (framework) de colecciones para facilitar su uso.

Tipos de colecciones

Existen distintos tipos de colecciones; ordenadas, sin orden, indexadas, que permiten elementos repetidos, que no permiten elementos repetidos, que almacena un elemento compuesto (clave, valor)

Protocolo de Iteración

Protocolo para "manipular" esos objetos/elementos. Utilizaremos un iterador para el recorrido de cualquier colección y eliminar un elemento.

Interface Iterator

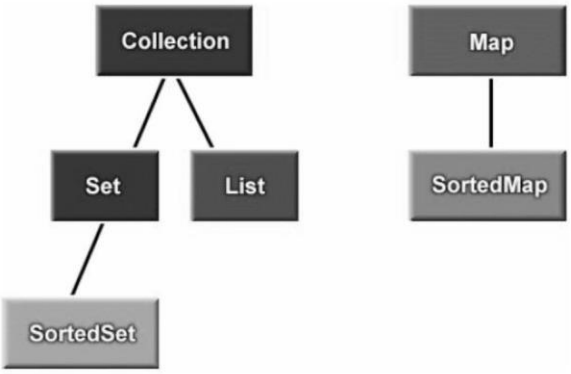
```
Interface Iterator{
        boolean hasNext();
        /* Devuelve true si la iteración tiene mas elementos */

        Object next();
        /* Devuelve el siguiente elemento de la iteración*/

        void remove();
        /* Elimina el último elemento devuelto por la iteración*/
}
```

La interfaz **ListIterator** extiende a Iterator y maneja un objeto List ordenado. Permite iterar hacia delante y hacia atrás.

Ejemplo, en el que el iterador nos permite recorrer los elementos de un *Vector*:

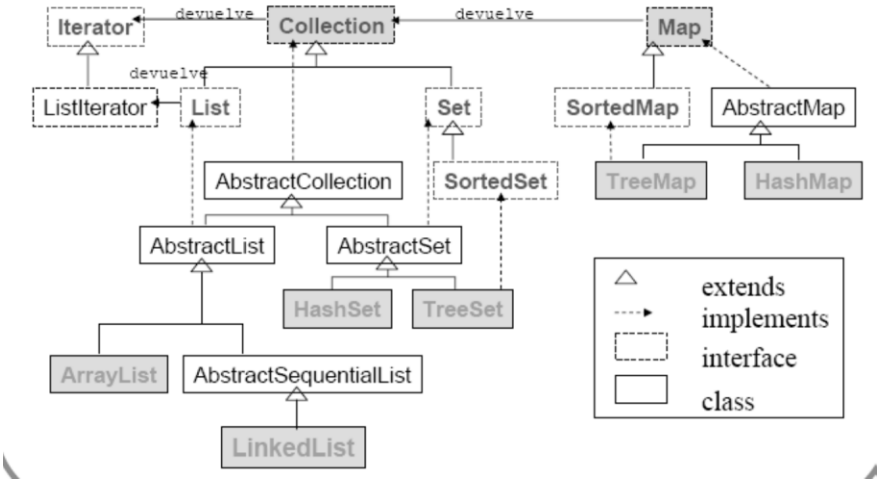
```
Vector vec = new Vector();
vec.add( new String( "hola“ ) );
vec.add( new String( "adios“ ) );
Iterator it = vec.iterator();
while ( it.hasNext() )
        System.out.println( (String) it.next() );
)
```

Jerarquía de interfaces de colecciones



- Collection: Representa un grupo de objetos. Sin implementaciones directas, agrupa la funcionalidad general que todas las colecciones ofrecen.
- Set: Colección que no puede tener objetos duplicados.
- SortedSet: Set que mantiene los elementos ordenados.
- List: Colecciones ordenadas (secuencias) en las que cada elemento ocupa una posición identificada por un índice. El primer índice es el 0. Las listas admiten duplicados.
- Map: Son pares de datos (clave, valor). No puede haber claves duplicadas y cada clave se corresponde con al menos un valor.
- SortedMap: Map que mantiene las claves ordenadas.

Con la aparición de las nuevas versiones: Interface Queue(colas) y LinkedList (Listas Enlazadas).



Ejemplo de clases implementadas

Las interfases List, Set y SortedSet son descendientes de la interfase Collection.
El concepto de Polimorfismo aplica para todas las clases que implementan estas interfases.

- Las clases que implementan la interfase List son: ArrayList y LinkedList
- Las clases que implementan la interfase Set son: HashSet y LinkedHashSet
- La clase que implementa la sub-interfase SortedSet es: TreeSet
- Las clases que implementan la interfase Set son: HashSet y LinkedHashSet

Set

Por sobre lo que es una collection, un set agrega una sola restricción: No puede haber duplicados.
Por lo general en un set el orden no es dado. Si bien es posible que existan sets que nos aseguren un orden determinado cuando los recorremos (por ejemplo obtener strings en orden alfabético), ese orden no es arbitrario y decidido por nosotros, ya que la interfaz Set no tienen ninguna funcionalidad para manipularlo (como si lo admite la interfaz List).
La ventaja de utilizar Sets es que preguntar si un elemento ya está contenido mediante “contains()” suele ser muy eficiente. Entonces es conveniente utilizarlos cada vez que necesitemos una colección en la que no importe el orden, pero que necesitemos preguntar si un elemento está o no.

HashSet

Existen varias implementaciones de Set. La más comúnmente usada es HashSet.
Los Sets (y los Maps) aprovechan una característica de Java: Todos los objetos heredan de Object, por lo tanto todos los métodos de la clase Object están presentes en todos los objetos. Dicho de otra manera, hay ciertas cosas que todo objeto en Java sabe hacer. Dos de estas cosas son:

1. Saber si es igual a otro, con su método equals().
2. Devolver un número entero de modo tal que si dos objetos son iguales ese número también lo será (se conoce esto como un *hash*). Esto todo objeto lo sabe hacer con su método hashCode().

La clase HashSet aprovecha la segunda de las funciones. A cada objeto que se añade a la colección se le pide que calcule su “hash”. Este valor será un número entre -2147483647 y 2147483648. Basado en ese valor se lo guarda en una tabla. Más tarde, cuando se pregunta con contains() si un objeto x ya está, habrá que saber si está en esa tabla.
¿En qué posición de la tabla está? HashSet puede saberlo, ya que para un objeto determinado, el hash siempre va a tener el mismo valor.

La clase que implementa la sub-interfase SortedSet es: TreeSet.

TreeSet

TreeSet usa una técnica completamente diferente a la explicada para HashSet. Construye un árbol con los objetos que se van agregando al conjunto. Un árbol es una forma en computación de tener un conjunto de cosas todo el tiempo en orden, y permitir que se agreguen más cosas y que el orden se mantenga.

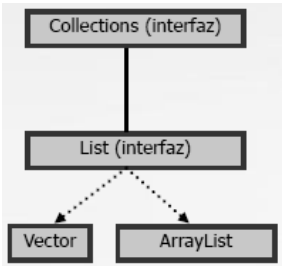
Una ventaja de TreeSet es que el orden en el que aparecen los elementos al recorrerlos es el orden natural de ellos .Una desventaja es que mantener todo ordenado tiene un costo, y esta clase es un poquito menos eficiente que HashSet.

Map

Un Map 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. A los primeros se los llama “claves” o “keys”, ya que nos permiten acceder a los segundos.

Clases

Ahora veremos las clases mas conocidas que implementan las interfaces y se encuentran en el paquete **java.util**



Colecciones basicas

Esto indica que *List* es una *interface* que hereda de la interface *Collection*.
Podemos encontrar dos clases que implementan su comportamiento: *ArrayList* y *Vector*.

Vector

La clase **Vector** es parte del paquete **java.util** de la librería estándar de clases de Java. Ofrece un servicio similar a un arreglo, ya que se pueden almacenar y accesar valores y referencias a través de un índice. Mientras un arreglo tiene un tamaño dado, un objeto de tipo **Vector** puede *dinámicamente crecer y decrecer conforme se vaya necesitando*.
Un elemento puede insertarse y eliminarse de una posición específica a través de la invocación de un sólo método.
A diferencia de un arreglo, un **Vector** no está declarado para ser de un tipo particular.
Un objeto de tipo **Vector** maneja una lista de referencias a la clase **Object.**

Métodos de la clase Vector

- **Vector ( )**Constructor: crea un vector inicialmente vacío
- **void addElement (Objet obj)** Inserta el objeto especificado al final del vector
- **void setElementAt (Object obj, int indíce)** Inserta el objeto específicado en el vector en la posición específicada
- **Object remove (int indíce)** Elimina el objeto que se encuentra en la posición específicada y lo regresa
- **boolean removeElement (Object obj)** Elimina la primera ocurencia del objeto específicado en el vector
- **void removeElementAt (int indíce)** Elimina el objeto específicado en el índice del vector
- **void clear ( )** Elimina todos los objetos del vector
- **boolean contains (Object obj)** Retorna verdadero si el objeto dado pertenece al vector

- **int indexOf (Object obj)** Retorna el índice del objeto específicado. Regresa -1 si no fue encontrado el objeto
- **Object elementAt (int indíce)** Retorna el componente en el índice específicado
- **boolean isEmpty ( )** Retorna verdadero si el vector no contiene elementos
- **int size ( )** Retorna el número de elementos en el vector

Array

Los Arrays son el tipo más sencillo de colección

```
Point2D[] puntos;
puntos = new Point2D [100];
for (int i=0; i <= puntos.length(); i++)
    puntos[i]= new Point2D();
```

El tipo de los objetos del array debe declararse en tiempo de compilación.
Hay dos características que diferencian a los arrays de cualquier otro tipo de colección: **eficiencia y tipo.**
El array es la forma más eficiente que Java proporciona para almacenar y acceder a una secuencia de objetos.
El array es una simple secuencia lineal, que hace que el acceso a los elementos sea muy rápido, pero el precio que hay que pagar por esta velocidad es que cuando se crea un array su tamaño es fijado y no se puede cambiar a lo largo de la vida del objeto.
Se puede sugerir la creación de un array de tamaño determinado y luego, ya en tiempo de ejecución, crear otro más grande, mover todos los objetos al nuevo y borrar el antiguo. Esto es lo que hace que la clase **Vector** sea menos eficiente que un array, en cuestiones de velocidad.

Clase Collections

Ordenar y buscar en Colecciones: Collections
La clase Collections (que no es la interface Collection) nos permite ordenar y buscar elementos en listas.
Se usaran los métodos:

- sort y binarySearch
- equals, hashCode y compareTo.

Patrones

Al comenzar un diseño lo ideal sería que un asesor de nuestro equipo se tome el trabajo de extraer la parte común de muchos otros diseños exitosos y se quede con las mejores ideas a nivel diseño para poder aplicarlas en nuestro proyecto.
Esto nos asegura el éxito de nuestro diseño, siempre y cuando hayamos entendido la idea expresada por el asesor, la apliquemos en el lugar correcto y de manera correcta

La idea de los patrones de diseño es cumplir la función de asesor….PERO es responsabilidad nuestra entender correctamente lo que cada uno de ellos propone y aplicarlos de manera correcta.
Los patrones de diseño son maneras convenientes de *reutilizar código* orientado a objetos entre proyectos y programadores.
La idea detrás de los patrones de diseño es simple: anotar y catalogar aquellas interacciones comunes entre objetos que se encuentran frecuentemente y son útiles.

"Un patrón es una solución a un problema en un determinado contexto"

Los buenos patrones deben tener al menos las siguientes características:
- Solucionar un problema reiterativo
- Ser un concepto probado
- La solución no es obvia
- Describe participantes y relaciones entre ellos

Podemos concluir entonces que para poder asegurar que hemos conseguido un patrón, los siguientes puntos deberán ser demostrados:
- Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón
- Un patrón se adapta para poder usarlo (adaptabilidad)
- Un patrón es útil (utilidad)
- **La repetición** es una característica cuantitativa pura. La podemos probar demostrando la cantidad de veces que fue usada (regla de 3).
- **La adaptabilidad** es una característica cualitativa. (como el patrón es exitoso)
- La **utilidad** también es una característica cualitativa. (porque es exitoso y beneficioso)

<u>Diferentes definiciones</u>
- Soluciones recurrentes a problemas de diseño que se presentan comúnmente.
- Un patrón trata un problema de diseño recurrente que aparecen en situaciones específicas del diseño y presenta una solución al mismo.
- Describe un problema de diseño recurrente y una solución al mismo de manera que para cada ocurrencia del problema, uno pueda instanciar la solución para ese problema
- Discusiones y registros de problemas de diseño
- Recuerdos de soluciones para resolver problemas
- Conjunto de reglas que describen como realizar ciertas tareas en el desarrollo de software

<u>Refinemos entonces la definición de patrón según Christopher Alexander:</u>

"Un patrón de diseño describe un problema que
ocurre una y otra vez en un contexto o medio,
y describe el núcleo de la solución a ese
problema, tal que pueda ser utilizada infinitamente sin
hacer necesariamente lo mismo de la misma manera dos veces"

Que viene “gratis” con los Patrones de Diseño…

- Experiencia.
- Sus nombres forman, colectivamente, un vocabulario que ayuda a los desarrolladores a comunicarse mejor, a hablar en el mismo idioma
- Si la documentación usa patrones, simplifica la lectura y comprensión.
- Si están bien aplicados, hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables

Que no viene “gratis” con los Patrones de Diseño…

La creatividad:

- Un patrón de diseño no garantiza nada por sí solo. Los patrones no buscan suplantar al humano en el proceso creativo
- Necesitamos seguir usando la creatividad para aplicar correctamente los patrones.

¿Cómo ayudan los Patrones a solucionar problemas de diseño?

1-Encontrando objetos apropiados.

- La parte compleja del diseño OO es descomponer el sistema en objetos debido a todos los factores que entran en juego:
    - granularidad
    - encapsulamiento
    - dependencia
    - evolución
    - reusabilidad
- objetos que no se desprenden de la especificación de requerimientos
- decidir entre Herencia o Composición

2-Determinando el número y tamaño de los objetos.

- Los objetos pueden variar enormemente en tamaño y número. ¿Cómo decidimos como debería ser un objeto?

3-Especificando el protocolo de los objetos:

- Los patrones de diseño ayudan a definir el protocolo de las clases.
- Ayudan a detectar que NO poner en la interfaz
- Ayudan a detectar cuando diferentes clases deben implementar las mismas interfaces

4-Diseñar pensando en las interfaces y no en la implementación

- Manipular objetos sólo en términos de sus interfaces.

Ventajas:

- Los clientes se mantienen independientes de las clases de objetos que usan.
- Los clientes sólo conocen las interfaces que estas clases deben implementar.

¿Como describir un patrón de diseño?

- Formato de un patrón:
    - ***Nombre y clasificación***
    - ***Intención: explicar en términos abstractos el objetivo del patrón***
    - ***Nombre alternativo***
    - ***Motivación: contar un problema concreto, la solución y luego la generalización***
    - ***Aplicabilidad: reglas que dicen cuando aplicarlo.***
    - ***Estructura: esquema de objetos.***
    - ***Participantes***
    - ***Colaboraciones***
    - ***Consecuencias: efectos adversos sobre el modelo de OO***
    - ***Implementación: un ejemplo de la solución.***
    - ***Código de ejemplo: un ejemplo en pseudocódigo***
    - ***Usos conocidos: ejemplos de aplicabilidad.***
    - ***Patrones relacionados***

Ventajas de los patrones de Diseño

- Son soluciones concretas:

Un catálogo de patrones es un conjunto de recetas de diseño. Cada patrón es independiente del resto.

- Son soluciones técnicas:

Dada una situación, los patrones indican cómo resolverla.

- Se aplican en situaciones muy comunes:

Proceden de la experiencia y han demostrado su utilidad.

- Son soluciones simples:

Indican cómo resolver un problema utilizando un pequeño número de clases relacionadas de forma determinada. No indican cómo diseñar un sistema completo, sino sólo aspectos puntuales del mismo.

- Facilitan la reutilización del código y del diseño:

Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables. La propia estructura del patrón se reutiliza cada vez que se aplica.

Desventajas de los patrones de Diseño

- Su uso no se refleja claramente en el código:

A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado. No es posible hacer ingeniería inversa.

- Es difícil reutilizar la implementación del patrón:

El patrón describe roles genéricos, pero en la implementación aparecen clases y métodos concretos.

Clasificación de los patrones de Diseño

	De Creación	Estructurales	De Comportamiento
Clase	Factory method	Adapter	Template Method
Objeto	Abstract Factory Builder Prototype Singleton Factory	Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento State Strategy Visitor

Patrón Observer

•**Intención:**

–Define una dependencia uno a muchos entre objetos de modo que, cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

•**Nombre alternativo:**

–Dependants, Publish-Subscribe.

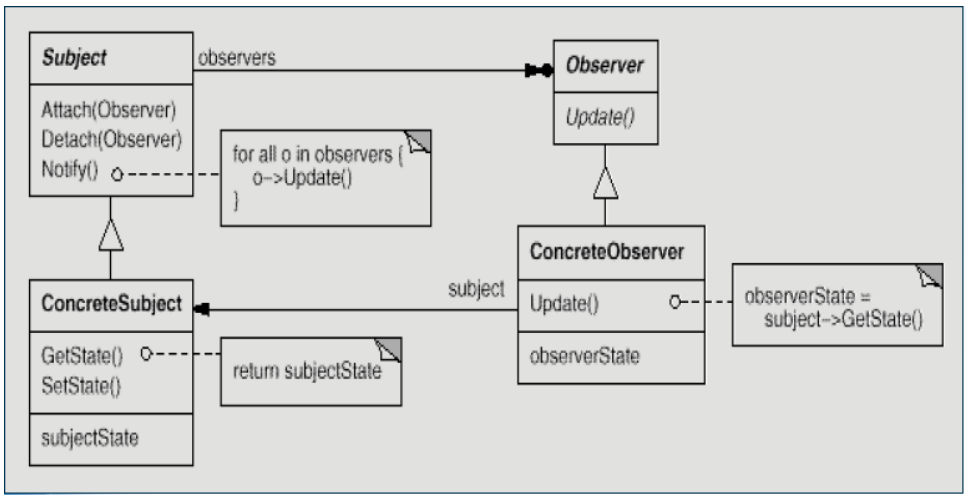
•**Motivación:**

–Un "side-effect" común al particionar el sistema en una colección de clases que cooperan para realizar cierta tarea, es la necesidad de mantener la consistencia entre estos objetos relacionados.

–Obviamente se quiere lograr consistencia, pero no al costo de atentar contra la flexibilidad y el reuso.

–Las aplicaciones con interfaces de usuario son un claro ejemplo de estas situaciones.

• **Estructura:**



•**Participantes:**
•Subject:
–Cualquier cantidad de observadores pueden estar interesados en un subject.
–provee una interface para agregar y borrar dependientes.
•Observer:
–define una interface de actualización para aquellos objetos que deberían ser notificados de cambios.
•ConcreteSubject:
–lleva cuenta del "interés" de los ConcreteObserver.
–manda una notificación de cambio cuando altera su estado.
•ConcreteObserver:
–mantiene una referencia a un objeto ConcreteSubject.
–almacena cierto estado que debería ser consistente con el objeto observado.
–implementa la interface de actualización para mantenerse consistente con su "modelo".
•**Colaboraciones:**
–ConcreteSubject notifica a sus dependientes cada vez que ocurre un cambio.
–Luego de ser informado de la ocurrencia de un cambio, un ConcreteObserver puede llegar a preguntar al subject sobre su estado.
•**Consecuencias:**
–Bajo acoplamiento entre subjects y observers puedo hacerlos interactuar aunque pertenezcan a diferentes layers de un sistema (model - view)
–Se pueden rehusar los subjects sin necesidad de rehusar los observers
–Puede agregar observers en run-time sin necesidad de modificar ni el subject ni los otros observers (+)
–Depende la implementación, puede haber actualizaciones no esperadas. (-)
•**Aplicabilidad:**
–Usar este patrón en cualquier de estas situaciones:
•Cuando una abstracción tiene dos aspectos, uno dependiente del otro.
•Cuando el cambio en un objeto requiere la actualización de otros, sin saber cuantos son.
•Cuando un objeto debería ser capaz de notificar a otros objetos sobre su cambio, sin necesidad de conocer las características de esos objetos
•**Patrones relacionados:**
–Mediator: puede utilizarse para encapsular la lógica de actualización.
–Singleton: puede utilizarse para manejar los sujetos.

Patrón Composite

•**Intención:**
– Construir objetos mediante la composición recursiva de objetos similares de manera similar a un árbol.
•**Motivación:**
– Para crear componentes que pueden agruparse para formar componentes mayores.
•**Aplicabilidad:**
– Respresentar jerarquías de objetos parte-todo. Para que los clientes puedan manejar indistintamente objetos individuales o composiciones de objetos.

•**Participantes:**

– Client, Component, Leaf, Composite

•**Colaboraciones:**

– Los clientes usan la clase Componente para interactuar con los objetos de la estructura Compuesta. Si el recipiente es una Hoja, la petición se maneja directamente. Si se trata de un Compuesto, se pasa a sus componentes hijos, pudiéndose realizar operaciones adicionales antes o después.

•**Consecuencias:**

– El cliente no necesita diferenciar el tratamiento de objetos simples y objetos composites.(+)

– Hace a los clientes más simples. (+)

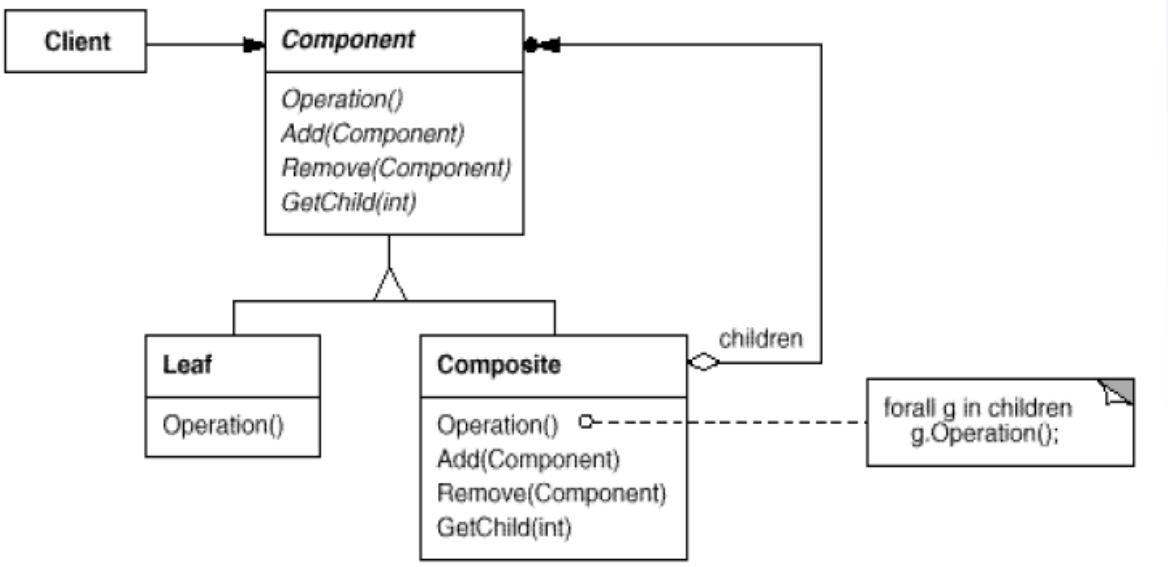
– El cliente no se toca al agregar nuevas componentes.(+)

•**Implementación:**

–La clase Component debe tener TODAS las operaciones que comparten las hojas y los composite.

–Para ganar transparencia (pero perdiendo seguridad) en el cliente, se podrían definir operaciones en component que no son de leaf (en Java se deberían manejar con excepciones)

•**Estructura:**

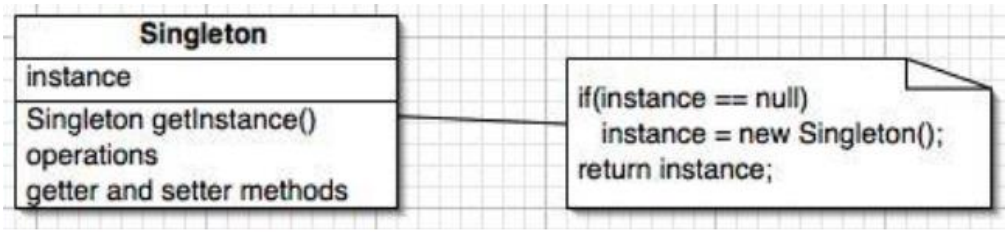


Patrón Singletón

•**Intencion:**

–Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

•**Estructura:**



•**Motivacion:**

– Algunas clases solo necesitan un ejemplar. En lugar de tener una variable global para acceder a ese ejemplar único, la clase se encarga de proporcionar un método de acceso.

•**Colaboraciones:**

– Los clientes acceden al ejemplar de Singleton únicamente a través del método instance de la clase Singleton.

•**Consecuencias**

– Controla el acceso a un ejemplar único.

– Evita la necesidad de usar variables globales.

– Es mas flexible que las operaciones de clase static

Ejemplos:

–contadores para asignar identificadores.

–controladores de colas de impresión.

–clases que representen tipos univaluados.

Patrón State

•**Intención:**

–Encapsula los estados de un objeto como objetos separados

–Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. El objeto parecerá haber cambiado de clase.

Se utiliza cuando se desea tener una clase que puede cambiar su estado. La clase incluye a otra que determina su estado y el mismo puede cambiar.

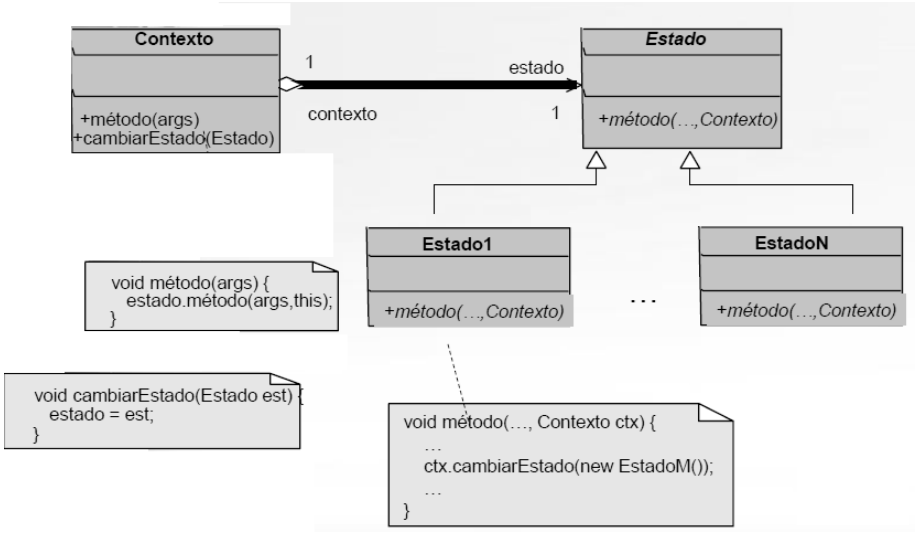
La clase envía los mensajes a su estado y de acuerdo al mismo la funcionalidad varía. Nótese que la responsabilidad sigue siendo de la clase original y no del estado.

State sugiere cambiar entre las clases internas de una manera tal que el objeto simule cambiar su estado.

Este patrón es utilizado para evitar el if-then-else o switch.

Es un patrón de comportamiento para objetos. Se debe usar si el comportamiento de un objeto depende de su estado, y se ve modificado en tiempo de ejecución.

Permite que un objeto cambie su comportamiento cuando cambia su estado interno, tal y como si el objeto cambiase de clase.



Patrón Adapter

•**Intención:**

Convertir el protocolo de una clase en otra, que es la que el objeto cliente espera.

•**Motivación**

– Muchas veces, clases que fueron pensadas para ser reutilizadas no pueden aprovecharse porque su protocolo no es compatible con el protocolo específico del dominio de aplicación con el que se está trabajando.

– Deseamos utilizar una clase existente, cuyo protocolo no encaja con la que necesitamos.

– Deseamos crear una clase que puede llegar a cooperar con otros objetos cuyo protocolo no podemos predecir de antemano.

•**Solución:**

Crear una clase que se encargue de "transformar" los nombres de los mensajes.

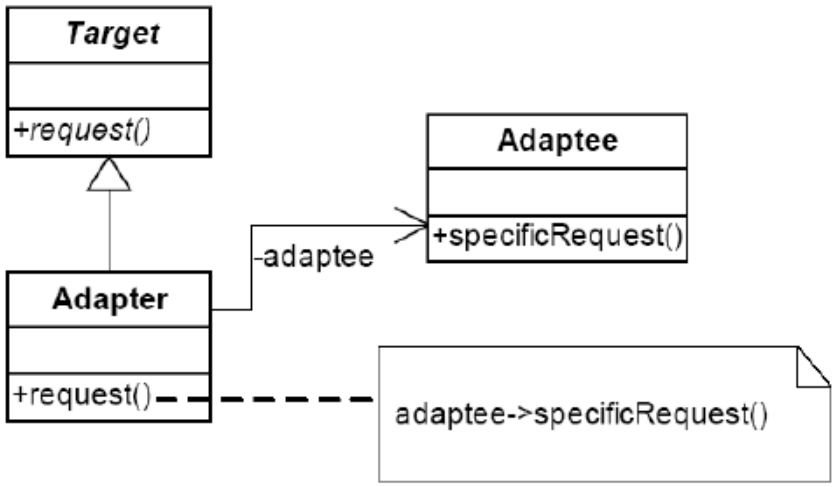
•**Consecuencias:**

– El Adapter puede sobre-escribir algún método del "adaptado".

– Solo introduce un objeto como indirección.

– Adaptar toda una jerarquía requiere trabajo extra para inicializar las referencias.

– Puede ser inconveniente si el "adapter" implementa demasiado comportamiento.



Double Dispatch

"Dispatching" es el proceso que determina el método a ejecutar cuando un objeto recibe un mensaje. El método que se ejecuta cuando un objeto recibe un mensaje depende exclusivamente de la clase del objeto receptor.

Este comportamiento es suficiente en la mayoría de los casos, sin embargo en algunos casos sería deseable que el método elegido dependiera de la clase del receptor y de la clase de alguno de los parámetros del mensaje.

Técnica Double Dispatch

•**Objetivo:** Seleccionar un comportamiento que depende de dos clases de objetos.
•**Problema:** En algunos casos es necesario escribir código cuyo comportamiento dependa de la clase del parámetro de un método.
En la programación procedural este tipo de problemas se resuelve usando un *case*. Hacer esto en objetos (basados en la clase del parámetro) resulta en código que no escala.
En el paradigma de objetos tenemos la ventaja del *polimorfismo*.
•**Solución:**
- Escribir un método secundario por cada clase. Su nombre será similar al principal, con el agregado del nombre de la clase.
- Al llamarse al método principal, éste delegará en el parámetro llamando al método secundario asociado con su clase.
- En el caso de ser necesario, se puede pasar al objeto receptor del mensaje primario como parámetro del mensaje secundario.
- Double dispatch significa que la operación que se ejecuta depende del tipo de 2 receptores.
•**Consecuencias:**
- Double Dispatch elimina las sentencias del estilo *case* basadas en la clase del parámetro.
- Esto hace que el código escale y pueda ser mantenido en forma relativamente sencilla.
- A pesar de esto no es una panacea; al agregar una nueva clase, según en qué jerarquía se agregue, puede ser que sea necesario agregar un método nuevo en todas las clases de la otra jerarquía. De todas maneras, ésta última desventaja es preferible a la sentencia *case*.

Double Dispatch != Overloading (sobrecarga).
Overloading permite que la función invocada dependa del tipo de parámetros y del tipo del receptor.
Overloading se chequea en tiempo de compilación. No considera el tipo de los objetos sino de las variables.
DoubleDispatch asegura (por medio de envío de mensajes) que el objeto apropiado es referenciado.
En C++ y Java es necesario utilizar funciones virtuales.

Lazy Initialization

Pueden existir situaciones en las cuales no deseamos que el compilador cree un objeto por defecto, porque por cada referencia se puede incurrir en una perdida de tiempo innecesaria en muchos casos.
La técnica Lazy initialization (inicialización perezosa) crea la instancia justo antes de que se necesite utilizar el objeto.
Esta puede reducir la sobrecarga en situaciones donde el objeto no necesita ser creado cada vez.