

ESTRUCTURAS DE DATOS

MATERIAL DE LECTURA Y SOPORTE EN JAVA

CONTENIDO

Introducción.....	7
Organización de la lectura	7
Complejidad y Recursividad.....	9
Informática Teórica.....	9
Concepto.....	9
Necesidad de formalismo	10
Notación asintótica.....	10
Complejidad de un algoritmo	11
Tiempo de ejecución.....	11
Verificación de complejidad n^2	12
Complejidad en otros algoritmos	12
Concepto de Recurrencia.....	13
Ejemplos básicos de recurrencia.....	13
Tips para un método recurrente.....	14
Patron de Diseño: Observer.....	14
Haciendo varias cosas al mismo tiempo	15
Torres de Hanoi.....	16
Consideraciones de la propuesta de implementación.....	18
Detalles de la implementación	19
Complejidad del algoritmo de Hanoi	22
Quicksort.....	23
Complejidad de Quicksort.....	24
Aplicación grafica de Quicksort	24
Recurrencia Gráfica.....	29
Curva de Von Koch.....	29
Triangulo y Cuadrado de Sierpinsky.....	30

Un poco de Logo	31
Curva de Hilbert	32
Preguntas sobre complejidad y recurrencia	32
Estructura de Datos	34
Estructuras de Datos Estaticas	34
Estructuras estaticas lineales: Arreglos	34
Estructuras estaticas no lineales: Matrices	34
Matrices: Juego de la vida de Conway	36
Matrices: Imágenes	39
Patron de Diseño: Modelo – Vista – Controlador MVC	42
Matrices: Fractal de Mandelbrot	43
Galeria de imágenes de Mandelbrot	44
Implementacion en Java	44
Estructuras de Datos con Comportamiento	48
Pila	48
Cola	49
Generics	50
Estructuras de Datos Dinámicas	51
Estructuras de Datos Dinámicas Lineales	51
Estructuras de Datos Dinámicas NO Lineales	51
EDD Dinamica Lineal: Cadenas	51
Implementación de Cadena en Java	53
Patrón de diseño Iterator	55
EDD Dinámica Lineal: Cadena Doble	56
EDD Dinámica Lineal: Anillo	57
EDD Dinámica Lineal: Anillo Doble	57
EDD Dinámica Lineal: Tablas de Hash	57

En Java	58
Patrón de Diseño: Comparator	58
Solucion en Java	58
EDD Dinámica Lineal: Cadena Ordenada	59
EDD Dinámica No Lineal: Árbol	60
Terminología	60
Uso de la estructura Árbol	60
Implementación de arboles	60
Jerarquia de Clases Java	61
Algoritmos de recorrido de árbol	63
EDD Dinámica No Lineal: Árbol Binario	64
Árbol equilibrado	65
EDD Dinámica No Lineal: Árbol B	65
EDD Dinámica No Lineal: Grafos	67
Ejemplos de grafos	67
Implementación de grafos	68
Grafo con Programación Orientada Objetos	69
Problemas que se solucionan con Grafos	70
Recorrido en Profundidad: DFS (Depth First Search)	70
Recorrido en Anchura: BFS (Breadth First Search)	71
Implementación búsqueda con DFS/BFS	72
EDD Dinámica No Lineal: Redes	74
Implementación de Redes	74
Algoritmo de Dijkstra	75
Preguntas	84
Acceso a Datos	86
Persistencia de tipos de datos abstractos	86

Archivos / Ficheros.....	86
Lectura y escritura en ficheros.....	87
Bases de Datos.....	87
Conceptos importantes	88
El Lenguaje SQL.....	88
Motores de Bases de Datos	89
Lectura y escritura en base de datos	90
Patrón de Diseño: Decorator	92
Programación en Capas	93
Patrón de diseño DAO	94
Patrón de Diseño: Singleton	96
Patrón de diseño: Abstract Factory	97
Implementacion DAO con Singleton y Abstract Factory.....	98
Ejemplo de Uso de DAO.....	100
Ventajas del patrón de diseño DAO.....	103
Persistencia moderna	103
Preguntas.....	103
Útiles de Programación.....	105
Registro de Eventos	105
LOG4J	105
Documentacion y JAVADOC.....	108
Buenas prácticas para nuestro código.....	110
Legible.....	110
Programación Defensiva	110
Logs	111
Complejidad Ciclomática	111

INTRODUCCIÓN

El siguiente texto es un pequeño aporte a la ya cargada bibliografía de textos para estructuras de datos informáticas. Ha sido realizado con el aporte de las clases dadas en la Universidad NUR a estudiantes de pregrado durante 18 semestres desde enero 2005 a diciembre 2013.

En este tiempo se ha tratado de identificar las razones por las cuales un estudiante se motiva para comprender y estudiar la materia. Me he permitido entonces elaborar un pequeño perfil de estudiante para la materia:

Soy una persona de 19 años o más que se interesa a la ingeniería de sistemas. Me gusta la programación y me divierten los algoritmos recurrentes como el quicksort. Estoy comprometido con aprender al menos un nuevo lenguaje cada 6 meses. También me interesan los nuevos algoritmos; luego de aprender las bases de algoritmos me interesa poder entender algoritmos como el PageRank de búsqueda de Google, para esto creo que el inglés(leído) es una herramienta fundamental que no puedo flanquear bajo ningún motivo. Creo fervientemente que el potencial de los profesionales de sistemas esta apenas comenzando y debería convertirse en 10 años en una industria que represente al menos el 5% del PIB del país.

Con esta perspectiva en mente lo que se busca es motivar al estudiante para que complemente su estudio con las innumerables referencias disponibles en el mundo en línea de hoy. A partir de este momento es cuando el estudiante deberá empezar a investigar para complementar la enseñanza. Existen muchas métricas en el mundo de la informática:

- Los procesadores son el doble de rápidos y el doble de pequeños cada 18 meses.
- Cualquier tecnología (hardware) de más de 18 meses es considerada obsoleta.
- Un profesional de sistemas, luego de 18 meses sin ningún tipo de INPUT nuevo, es considerado obsoleto.

La última atañe al estudiante de sistemas. Y es justamente a partir de esta materia, luego de haber pasado al menos un año en las aulas de la universidad que el estudiante debe comenzar a cultivar las buenas costumbres que lo lleven a realizarse como profesional.

ORGANIZACIÓN DE LA LECTURA

El siguiente texto ha sido elaborado siguiendo el ejemplo de varios cursos de estructura de datos de varias universidades del mundo. Se puede decir que el contenido es bastante estándar para la materia en cuestión. El aporte que se ha querido hacer esta en el énfasis en el aprendizaje de los métodos y las técnicas y no en la dependencia al lenguaje de programación. En este sentido se libra una batalla entre aquellos que no piensan en Java como un lenguaje apto para el aprendizaje ya que esconde varias tareas al programador debutante.

Los ejemplos están acompañados de didácticas pruebas de escritorio que permitan al estudiante entender a cabalidad el algoritmo que se desea realizar. También se incluyen ejemplos elaborados donde se ven los patrones de diseño en programación, se han incluido 7 patrones de diseño del GoF. En cada uno de los algoritmos y ejercicios explicados se irá en detalle en la implementación en la medida que ayude al programador a entender el objetivo del ejercicio. Los detalles de la implementación que sean evidentes serán dejados como ejercicio.

Al final de cada capítulo se pueden observar algunas preguntas referentes al capítulo. Todos los capítulos están acompañados de ejemplos prácticos con el código fuente incluido suficiente como para que el estudiante pueda replicar los mismos. Al final de la lectura se pueden encontrar las referencias bibliográficas.

El primer capítulo trata de la recurrencia y la importancia de la complejidad de un algoritmo en informática. Se aconseja que para este capítulo se dediquen al menos 8 horas para la lectura y la practica correspondiente.

El segundo capítulo es el más importante y trata de las estructuras de datos básicas en programación. Se recorren los conceptos, ejemplos y explicaciones de estructuras de datos estáticas, dinámicas, lineales y no lineales. Se repasan estas estructuras con sus nombres característicos: arboles, grafos, cadenas, etc.

El tercer capítulo es la implementación de las estructuras en un ejemplo real como es el acceso a datos. Este capítulo integra todo lo aprendido durante la materia y asegura que el estudiante ha comprendido todos los detalles y pormenores de la programación orientada a objetos.

Como anexo a este texto se han adjuntado una serie de útiles de programación destinados a mejorar la calidad de la programación del programador promedio.

COMPLEJIDAD Y RECURSIVIDAD

“Un hombre provisto de papel, lápiz y goma, y con sujeción a una disciplina estricta, es en efecto una máquina de Turing universal.”. Alan Turing

En esta unidad se ven los conceptos de complejidad en informática teórica. Los conceptos de recursividad se ven en detalle y se ven todos los ejemplos de recursividad desde lo básico a lo complejo.

INFORMÁTICA TEÓRICA

CONCEPTO

La informática teórica es una rama de la informática que estudia la formalización de algoritmos y la validez formal del cálculo científico realizado por una computadora. Entre otras cosas nos indica también la factibilidad de realizar algoritmos.

Es muy importante conocer algunos de los conceptos de esta rama ya que cualquier profesional de sistemas se ve envuelto en la resolución de un problema que puede ser NO resoluble.

Se estudian los algoritmos por medio de autómatas, máquinas de Turing y se averigua, por medio de estos modelos, la complejidad de los algoritmos. La complejidad de los algoritmos ha repartido los problemas en: polinomiales, P y NP. La gran pregunta es saber si los problemas NP (muy complejos) se pueden reducir a que sean solamente P (resueltos en tiempo polinomiales).¹

Para los matemáticos surgió una pregunta al comienzo de la era de la computación:

- ¿Qué prueba se tiene de que el cálculo realizado por la computadora está bien hecho?
- ¿Existe alguna manera formal de probar que una máquina va a hacer bien los cálculos en todos los casos?

Para poder responder a esta pregunta los matemáticos e informáticos pusieron las bases de la informática teórica creando el concepto de Máquina de Turing². Este concepto permite modelar problemas complejos y saber si el modelo soporta la resolución de un problema en un tiempo aceptable. Se pueden responder a preguntas como:

- Mi algoritmo se va a terminar? En cuanto tiempo?
- ¿Me da la respuesta correcta?
- ¿Por qué?

La resolución que tiene cada problema se subdivide básicamente en dos: problemas solubles en tiempo polinomiales y problemas NP. Los problemas solubles en tiempo polinomiales tienen la característica que a medida que el problema se hace más grande, los recursos necesarios para solucionarlo crecen proporcionalmente. Los problemas NP sin embargo, a medida que crecen puede que no tengan solución en un tiempo aceptable o simplemente no tienen ninguna solución conocida. Algunos ejemplos de estos problemas:

- Factorización de números grandes (números de 200 o más dígitos)
- Saber si un número grande (más de 200 dígitos) es primo

¹ Este es uno de los llamados problemas del milenio: ver <http://www.claymath.org/millennium/>

² Alan Mathison Turing fue un científico de la informática inglés. Es considerado uno de los padres de la Ingeniería informática siendo el precursor de la Ciencia de la computación moderna. Ver http://es.wikipedia.org/wiki/Alan_Turing

- Hacer un programa que tome como entrada otro y nos diga si funciona o no.

NECESIDAD DE FORMALISMO

Al hacer uso de la computadora para resolver nuestros problemas partimos del hecho que es una maquina exacta que nos resuelve el problema de la manera más precisa posible. Sin embargo, a veces nuestros programas se cuelgan y dejan a la maquina en un estado de latencia extrema. Cómo podemos estar seguros entonces que un algoritmo programado da efectivamente la respuesta correcta? La única manera es haciendo una prueba formal, por ejemplo:

La suma de los primeros n números naturales está dada por la fórmula: $\sum_{k=1}^n k = \frac{n(n+1)}{2}$

Esto se puede probar fácilmente de la siguiente manera:

Probemos que es válido para $n = 2$

$$1 + 2 = \frac{2(2 + 1)}{2} = 3$$

Ahora suponemos que es válido para n y probamos que es válido para $n+1$:

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^n k + (n + 1) = \frac{n(n + 1)}{2} + (n + 1) = \frac{n^2 + n + 2n + 2}{2} = \frac{(n + 1)((n + 1) + 1)}{2}$$

Lo cual nos prueba que la formula funcionara para cualquier n positivo mayor o igual a 1.

Todos los algoritmos son similares a las funciones matemáticas, sin embargo, no sabemos por ejemplo si el sistema operativo Windows funcionara siempre y en todos los casos, no sabemos si los programas que controlan las impresoras funcionarían siempre y en todos los casos. Necesitamos entonces algunas herramientas que nos permitan por lo menos visualizar la validez de nuestros algoritmos.

NOTACIÓN ASINTÓTICA

La notación asintótica es un recurso que nos permite estimar la cantidad máxima de operaciones que nuestro algoritmo va a realizar. Se trata de encontrar la función o la formula que nos da el número máximo de operaciones que realiza el algoritmo. El formalismo esta dado por la siguiente expresión (O grande de f , donde f es nuestro algoritmo):

$$\vartheta(f) = \{g: N \rightarrow R^+ \mid \exists c \in R^+, \exists n_0 \in N: \forall n > n_0, g(n) > cf(n)\}$$

Básicamente, esto significa que podemos encontrar una función matemática que nos dice cuánto tiempo va a tardar, como máximo, nuestro algoritmo. Algunas cosas que sabemos sobre la forma de estas funciones asintóticas:

- Si son polinomiales, entonces se piensa en una solución posible, escalable y perfectamente programable.
- Si son exponenciales, entonces solamente se pueden resolver para problemas muy pequeños
- Si son logarítmicas también son posibles, escalables y programables.

Veamos esto en un algoritmo sencillo para entender el concepto.

```
public static int[] ordenarSimple(int[] lista) {
    for(int i=0; i<lista.length; i++) {
```

```

    for(int j=i+1; j<lista.length; j++) {
        if (lista[i] > lista[j]) {
            int aux = lista[i];
            lista[i] = lista[j];
            lista[j] = aux;
        }
    }
    return lista;
}

```

Contemos el número máximo de operaciones que este algoritmo va a realizar cuando `lista.length = n`. Una operación dentro de una computadora puede ser: una asignación, una operación aritmética o una comparación.

- Siendo estrictos tenemos, para el primer for, n veces 3 operaciones (la comparación, la operación ++ y la asignación que el ++ implica)
- El segundo for se repite n veces.
- Aquí aplicamos notación asintótica y decimos que a lo sumo (máximo) el segundo for ejecuta su contenido n veces (en realidad es menos ya que el índice j comienza desde $i+1$).
- Dentro del segundo for se realizan a lo sumo (no entra siempre en la condición pero pensemos que sí) 4 operaciones: 3 asignaciones y una comparación.

Todo nos da:

$$\vartheta(n) = 3n + n(4n) = 3n + 4n^2$$

Cuando n tiende a ser muy grande, el término $3n$ queda muy chico a comparación a de $4n^2$. De la misma manera, el término 4 se puede obviar para n grande. Lo cual nos deja la siguiente fórmula:

$$\text{Si } n \rightarrow \infty \text{ entonces } \vartheta(n) = n^2$$

Esta es la función asintótica del algoritmo descrito anteriormente.

COMPLEJIDAD DE UN ALGORITMO

La complejidad nos indica cuán difícil es para la máquina realizar un algoritmo específico. En los anteriores párrafos se ha mostrado un ejemplo básico para encontrar esta función. En los siguientes se explorarán las implicaciones de la complejidad en el tiempo de ejecución.

TIEMPO DE EJECUCIÓN

Tomando el ejemplo anterior, si tenemos un arreglo con 1000 números, tendremos entonces (a lo máximo):

$$\text{Si } n = 1000 \rightarrow \vartheta(n) = n^2 = 1000 * 1000 = 1000000$$

Si el procesador que tenemos a disposición es un antiguo Intel 286 de 4Mhz³ entonces el problema será resuelto en t segundos, donde t depende de op (número de operaciones) y v (velocidad del procesador):

³ Mhz es la medida de operaciones por segundo que puede realizar el procesador

$$t = \frac{op}{v} = \frac{\vartheta(n)}{v} = \frac{1000000}{4000000} = 0.25s$$

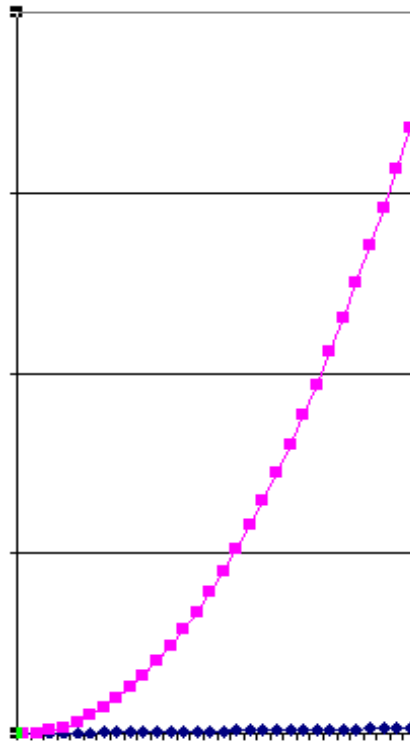
Qué pasa con un 'pequeño' arreglo de 10 millones sobre una máquina nueva de 5Ghz:

$$t = \frac{op}{v} = \frac{\vartheta(n)}{v} = \frac{(1 * 10^7)^2}{5 * 10^9} = 0.2 * 10^5 = 20000s$$

Lo cual equivale a más de 5 horas! Esto nos hace pensar en cómo hace Google y otros para indexar (ordenar por un criterio) todas las páginas de Internet, como lo hacen tan rápido?

VERIFICACIÓN DE COMPLEJIDAD n^2

Lo que debemos hacer acá es ejecutar nuestro algoritmo con arreglos de diferente tamaño tomando cada vez el tiempo necesario para su resolución. En la grafica se puede ver este procedimiento:



Los puntos rosados nos dan la forma que tiene nuestro algoritmo, la cual corresponde exactamente a la ecuación dada por nuestra función de complejidad: n^2

COMPLEJIDAD EN OTROS ALGORITMOS

Estudiar la complejidad del siguiente algoritmo:

```
public void ordenar(int[] arreglo) {  
    for(int i=1; i<arreglo.length; i++) {  
        int temp = arreglo[i];  
        int j= i-1;
```

```
while ((j >= 0) && (arreglo[j] > temp)) {
    arreglo[j+1] = arreglo[j];
    j--;
}
arreglo[j+1] = temp;
}
```

En el caso de este algoritmo, dejamos como ejercicio para el lector. Visiblemente se llegara a una fórmula similar que el anterior algoritmo y, la función asintótica del mismo será: n^2 . Lo cual nos indica que, en esencia, no existe diferencia de complejidad entre este y el anterior algoritmo.

CONCEPTO DE RECURRENCIA

De manera muy simple, es cuando se especifica un proceso basado en su propia definición. En español se utiliza el término *recurrencia*, sin embargo, es mucho más conocido el término *recursividad*. Se pueden utilizar ambos sin perjuicio de malentendidos u otros.

Algunos ejemplos famosos donde se utiliza este concepto:

- **Fibonacci:** Es una serie de números naturales donde el termino n es resultado de sumar el termino $n-1$ y el termino $n-2$.
- **Hanoi:** Es un juego que consta de 3 columnas que pueden contener anillos. Cada anillo es de tamaño diferente. El jugador debe mover los anillos de una columna a otra siguiendo 2 reglas 1) no puede mover más de un anillo a la vez y 2) sobre un anillo en una columna no se puede colocar otro más grande, siempre debe ser más pequeño.
- **Von Koch:** A partir de una línea, se van obteniendo diferentes trazos que dan lugar a lo que se llama el copo de nieve de Von Koch.
- **Mandelbrot:** Fractal famosos que se construye a partir de recurrencia sobre números complejos.
- **PHP:** Siglas que corresponden al lenguaje de programación **PHP** Hypertext Preprocessor. Nótese que la primera P corresponde justamente a la definición.
- **GNU:** Siglas que corresponden a la librería de los sistemas operativos basados en UNIX. **GNU's Not Unix**.

Analizando el concepto, en el caso de programación, un proceso es nada más un algoritmo. Un algoritmo es una serie de instrucciones. Para el caso de la POO, los algoritmos están escritos en los métodos de la clase.

Entonces, para comenzar a utilizar recurrencia, es necesario escribir un método que tenga entre sus instrucciones un llamado al mismo método.

Por otro lado, si hacemos esto de manera inocente no termina nunca. Por lo tanto, debemos programar en algún lugar del método una condición que le permita al algoritmo dar una respuesta.

EJEMPLOS BÁSICOS DE RECURRENCIA

Sumatoria

Escribir el código de un método que devuelve un entero y calcula la suma de los primeros n (parámetro entero) números naturales.

```
public int suma(int hasta) {
```

```
    if (hasta == 1)
        return 1;
    return hasta + suma(hasta-1);
}
```

Fibonacci

Esta es quizá la serie de números más conocida. Tiene una regla muy sencilla, comienza con 0 y 1. El tercer número se forma de sumar los últimos dos y así sucesivamente. Los primeros 10 números de la serie son:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Escribir un método que calcule el termino n (parámetro entero) de la serie de Fibonacci.

```
public int fibo(int numero) {
    if (numero == 1)
        return 0;
    if (numero == 2)
        return 1;
    return fibo(numero - 2) + fibo(numero - 1);
}
```

TIPS PARA UN MÉTODO RECURRENTE

Un método recurrente siempre se puede construir de la siguiente manera:

1. Identificar los casos excepción que sirven para detener el algoritmo
2. Identificar el algoritmo para el caso n.

PATRON DE DISEÑO: OBSERVER

Antes de entrar en materia de recurrencia, primero veremos la implementación del patrón de diseño Observer⁴.

Problema: Cómo hacer que un objeto, cuando cambie, avise a todos los demás objetos que lo están observando.

Solución Observer: Define una asociación uno a muchos, de manera a que cuando un objeto cambia, todos los objetos que lo observan, cambian en consecuencia.

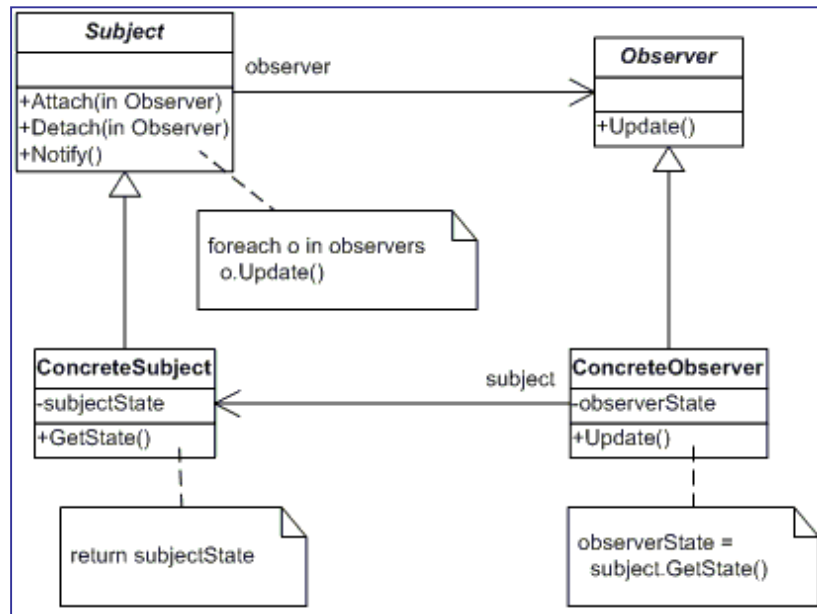
Para entender la implementación de la solución Observer se deben revisar 3 conceptos:

- **Evento:** Lo más importante es identificar el evento que vamos a manejar. Un evento puede ser:
 - Un clic del ratón
 - Que el ratón se mueva
 - Que la ventana se cierre
 - Que la ventana cambie de tamaño
 - Que el valor de una variable llegue a un valor específico
 - Que se destruya o crea un objeto
- **Observador:** El observador será un OBJETO que DEBE hacer algo cuando ocurre el EVENTO

⁴ Puede ver una explicación en línea del patrón de diseño en <http://www.dofactory.com/Patterns/Patterns.aspx>

- **Observado:** El observado es el OBJETO que mantiene una lista de observadores y les AVISA cuando ocurre el EVENTO

El diagrama que se utiliza para implementar este patrón es el siguiente:



LA forma de funcionamiento es la siguiente:

1. Se inicializa el patrón de diseño asignando el observador a la lista de observadores del observado
2. Cuando el observado cambia, llama a notificar a los observadores.
3. Gracias al diseño, el observador ejecuta el método `update()`

En Java ya está programado el patrón de diseño en una clase y una interface:

1. La clase `Observable` del paquete `java.util` tiene toda la implementación que se ve en el **Subject** del diagrama.
2. La interface `Observer` define el método `update()` tal como se ve en el diagrama

Luego, el objeto observado de nuestro programa solamente debe extender la superclase `Observable`, convirtiéndose en el **ConcreteSubject** del diagrama; y el objeto observador implementar la interface `Observer` definiendo el método `update()`, convirtiéndose en el **ConcreteObserver** del diagrama.

HACIENDO VARIAS COSAS AL MISMO TIEMPO

Un concepto importante en programación es el de tener múltiples procesos ejecutando métodos al mismo tiempo. Este es un concepto que es tradicionalmente llevado en los últimos cursos de programación universitaria. Sin embargo, viendo el avance de la tecnología y sobre todo el avance de Internet con sus redes sociales, juegos y otros es inevitable hablar de esto como programadores inclusive en los primeros semestres de estudio.

Lo que normalmente ocurre con cualquier de nuestros programas es que existe el único y solo proceso de la JVM que se encarga de ejecutar las instrucciones una después de la otra.

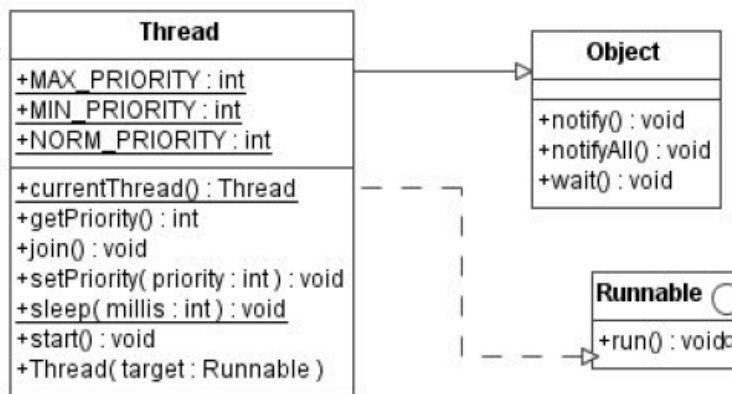
- Sin embargo, existen muchos casos donde necesitamos realizar cosas en paralelo:

- Word necesita revisar la ortografía al mismo tiempo que el usuario va escribiendo o haciendo otra acción.
- El navegador trae la información de Internet al mismo tiempo que la muestra.
- El navegador trae mucha información al mismo tiempo cuando se tienen abiertas varias pestañas
- Nuestros programas necesitan realizar acciones que nuestras interfaces deben mostrar al mismo tiempo

Como todo en Java es un objeto, un proceso (o tarea) es también un objeto. Lo único que necesita este tipo de objeto es:

1. Saber que código debe ejecutar
2. Poder comenzar la ejecución de ese código.

La clase `Thread` representa un proceso dentro de la JVM. La clase `Thread` implementa la interface `Runnable`. Los métodos importantes son `start()` y `run()`. Los demás métodos son para comunicación de threads. Se puede ver un diagrama inicial aquí:



Para poder implementar este modelo en nuestro código debemos hacer uso de la interface `Runnable`. Por ejemplo, imaginemos que tenemos un método `m()` que queremos ejecutar en paralelo con el resto de nuestro programa. Este método debe estar en una clase que implemente la interface `Runnable`.

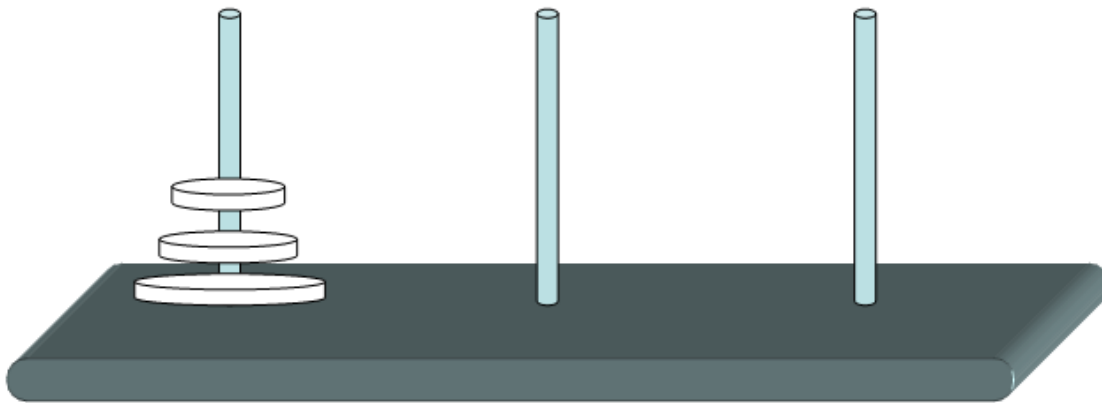
Luego debemos crear un objeto `Thread` con nuestro objeto `Runnable`. Y finalmente llamar a `start()` para que comience la ejecución del thread. Al hacer `start()` lo que hacemos es decirle a la máquina virtual de Java que cree los recursos para iniciar al proceso y se llama automáticamente al método `run()` del objeto `Runnable`, es decir al código que queríamos ejecutar.

TORRES DE HANOI

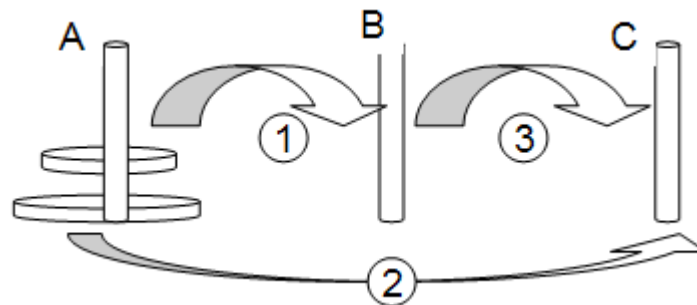
El problema clásico más conocido de recurrencia es el de las Torres de Hanoi. El juego, en su forma más tradicional, consiste en tres varillas verticales. En una de las varillas se apila un número indeterminado de discos (elaborados de madera) que determinará la complejidad de la solución, por regla general se consideran ocho discos. Los discos se apilan sobre una varilla en tamaño decreciente. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio en una de las varillas, quedando las otras dos varillas vacantes. El juego consiste en pasar todos los discos de la varilla ocupada (es decir la que posee la torre) a una de las otras varillas vacantes. Para realizar este objetivo, es necesario seguir tres simples reglas:

1. Sólo se puede mover un disco cada vez.
2. Un disco de mayor tamaño no puede descansar sobre uno más pequeño que él mismo.
3. Sólo puedes desplazar el disco que se encuentre arriba en cada varilla.

Existen diversas formas de realizar la solución final, todas ellas siguiendo estrategias diversas. A continuación se puede ver un diagrama de la forma básica del juego.



La solución básica para el problema con dos discos es la siguiente, en 3 pasos:



Se puede resolver este problema de manera muy sencilla con recurrencia. La resolución en Java se puede ver a continuación:

```
import java.util.*;

public class Hanoi {

    public static void main(String args[]){
        int n;
        Scanner sc=new Scanner(System.in);

        System.out.print("Escriba el número de discos en la torre de hanoi: ");
        n=sc.nextInt();
        while(n<0){
            System.out.print("Error, reescriba el número de discos en la torre: ");
            n=sc.nextInt();
        }

        algoritmoHanoi(n,"Primera Torre","Segunda Torre","Tercera Torre");
    }

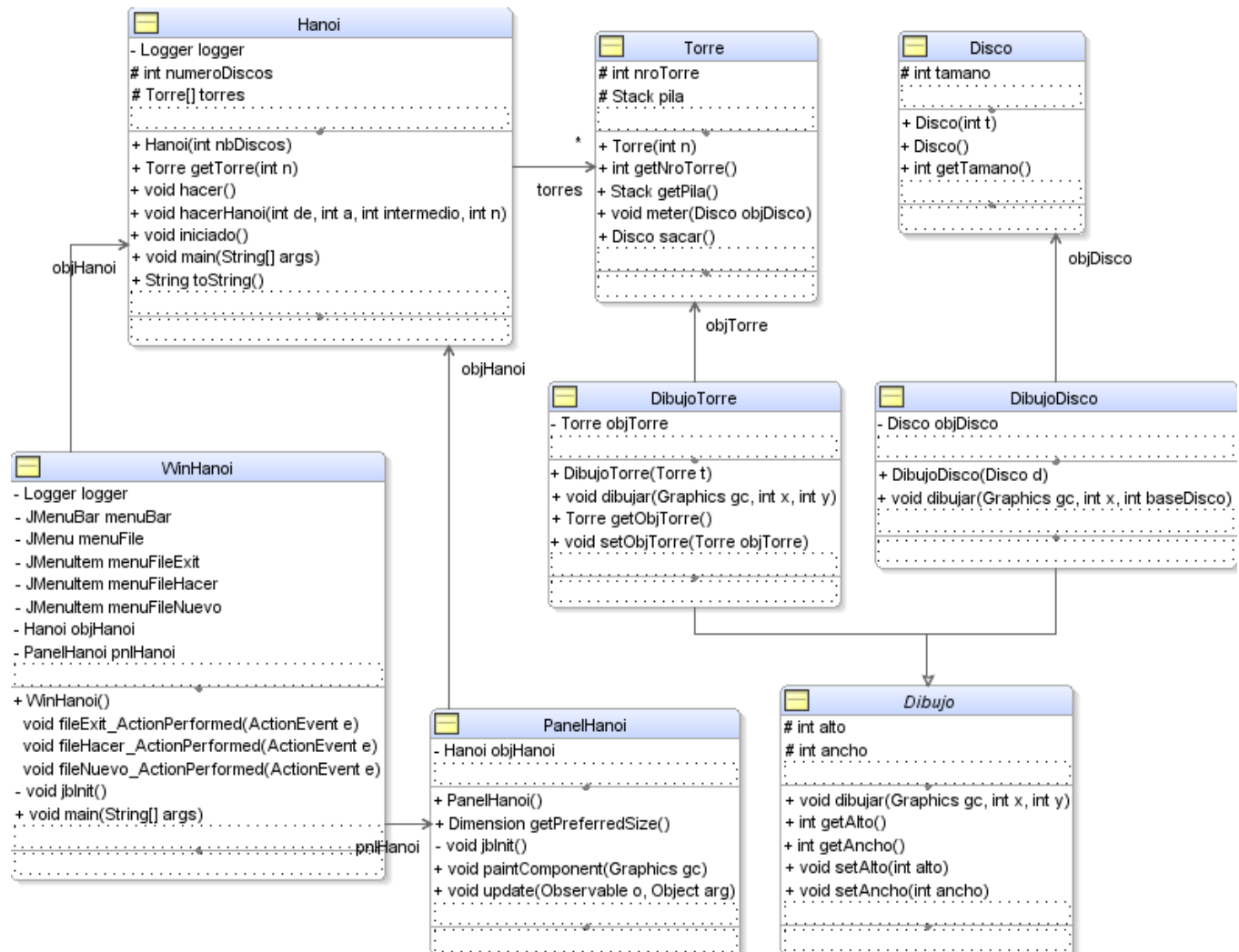
    public static void algoritmoHanoi(int n, String from, String temp, String to) {
        if (n == 0) return;
```

```

    algoritmoHanoi(n-1, from, to, temp);
    System.out.println("Mover disco " + n + " de " + from + " a " + to);
    algoritmoHanoi(n-1, temp, from, to);
}
}

```

Sin embargo, vamos a solucionar el problema con programación orientada objetos. Para ello primero modelamos nuestro diagrama de clases:



CONSIDERACIONES DE LA PROPUESTA DE IMPLEMENTACIÓN

PROGRAMACIÓN ORIENTADA A OBJETOS

Todos los objetos utilizados en el problema real se encuentran representados en el programa de manera directa. Se tiene Hanoi que es el juego; Torre que es la clase de los 3 objetos torre que hay en el juego; y Disco que es la clase de los n objetos disco que hay en el juego.

USO DE PILAS STACK

Una pila es un contenedor (un arreglo dinámico) de objetos de un mismo tipo que tiene una cualidad importante: el objeto que se coloca primero es el último que se recupera. Este objeto simula las reglas del juego sobre cada una de las torres. Es decir, cada torre funciona como una pila de discos.

SEPARACIÓN EN CAPAS

A cualquier nivel de programación el término de programación en capas ya ha sido escuchado y es el típico caso de una expresión que es muy difícil de poner en práctica. Por ello, en el ejemplo se puede observar que la programación del juego (la lógica de negocios) es de exclusiva responsabilidad de 3 clases: Hanoi, Torre y Disco. Por otro lado, toda la programación que tiene que ver con la interface del usuario se encuentra en otras clases WinHanoi, PanelHanoi, Dibujo, DibujoTorre, DibujoDisco.

LOGGING

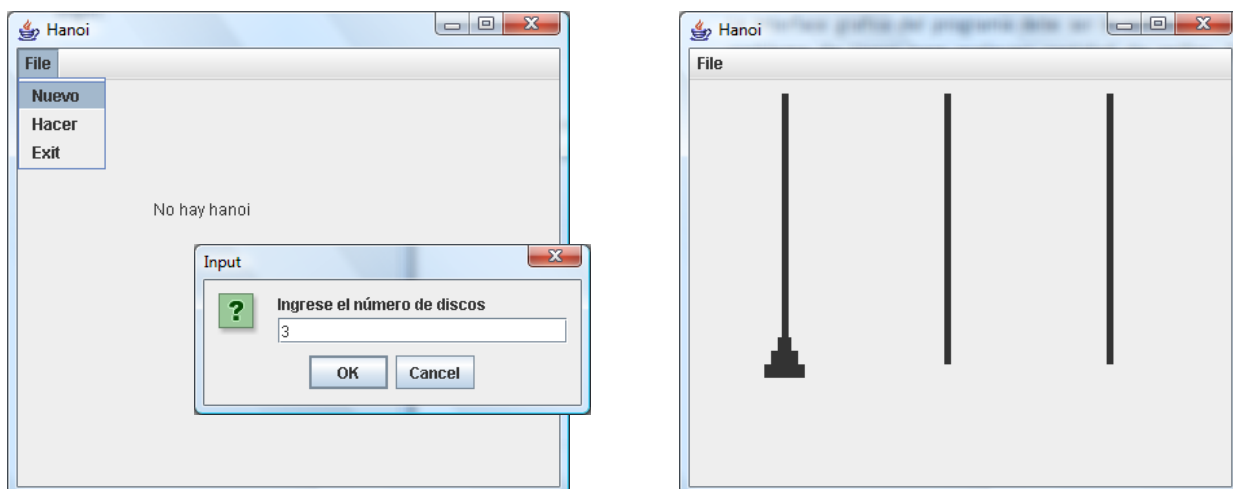
La estructura presenta objetos de tipo Logger que permiten a todas las clases registrar los eventos que sucedan en los métodos de las mismas. Ver el anexo al final del texto para poder implementar el ejercicio con registro de eventos con Log4J.

DETALLES DE LA IMPLEMENTACIÓN

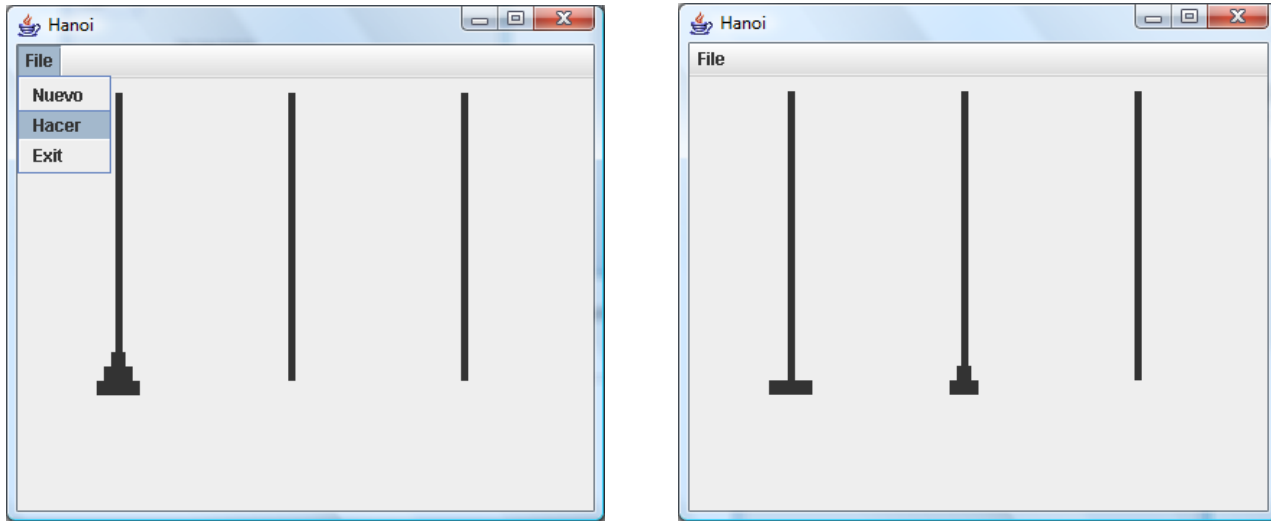
Para este ejemplo se deben considerar primero los patrones de diseño Observer e Iterator. Estos se pueden observar en los anexos

INTERFACE GRAFICA

La interface grafica del programa debe ser lo suficientemente potable como para que se pueda ver la resolución del problema de Hanoi para cualquier cantidad de discos. En los gráficos siguientes podemos ver la interface y el funcionamiento de los dos comandos en el menú:



Luego de iniciar el juego, podemos llamar al método para encontrar la solución, que nos muestra en una animación la solución del mismo:



En la grafica solamente vemos una de las etapas del movimiento. Cada medio segundo se hace el movimiento siguiente.

Dejamos para el lector el armado de la pantalla. Sin embargo, es importante ver la forma en que se dibuja el juego Hanoi en el PanelHanoi ya definido anteriormente:

```
public void paintComponent(Graphics gc) {
    super.paintComponent(gc);

    if (objHanoi == null) {
        gc.drawString("No hay hanoi", 100, 100);
        return;
    }
    Torre t = objHanoi.getTorre(0);
    DibujoTorre dt = new DibujoTorre(t);
    dt.dibujar(gc, 70, 10);

    t = objHanoi.getTorre(1);
    dt = new DibujoTorre(t);
    dt.dibujar(gc, 190, 10);

    t = objHanoi.getTorre(2);
    dt = new DibujoTorre(t);
    dt.dibujar(gc, 310, 10);
}
```

Lo que se hace es obtener de cada una de las torres un objeto de la clase DibujoTorre. Este objeto es el que sabe dibujar una torre, solamente debemos indicarle donde queremos que la dibuje.

Luego, en cada DibujoTorre, se debe proceder a dibujar la torre que no es más que dibujar los objetos Disco que se encuentran en la pila del objeto Torre:

```
public void dibujar(Graphics gc, int x, int y) {
    gc.fillRect(x- this.getAncho() / 2, y, ancho, alto);
}
```

```

        int baseDisco = y + this.getAlto();
        Iterator objIteracion = objTorre.getPila().iterator();
        while(objIteracion.hasNext()) {

            Disco obj = (Disco)objIteracion.next();
            DibujoDisco dibujo = new DibujoDisco(obj);

            dibujo.dibujar(gc, x, baseDisco);
            baseDisco -= dibujo.getAlto();

        }
    }
}

```

Nuevamente, lo único que se hace es dibujar uno a uno los discos. Vea el uso del patrón de diseño Iterator en el código para iterar sobre la pila.

Finalmente, el dibujar de Disco solamente coloca un rectángulo en el lugar correspondiente:

```

public void dibujar(Graphics gc, int x, int baseDisco) {
    gc.fillRect(x - this.getAncho() / 2, baseDisco, getAncho(), getAlto());
}

```

Esta delegación de funciones es muy conveniente ya que hemos eliminado toda la complejidad de saber en qué etapa esta? si se acaba de mover o no?

ALGORITMO HANOI Y OBSERVER

Una vez que esta lista nuestra interface, debemos poder indicarle cuando es que debe dibujar. En el algoritmo de Hanoi, cada vez que realizamos un movimiento de un disco, se debe notificar a la interface para que esta dibuje la nueva posición.

Aquí se puede ver el núcleo del programa en la lógica de negocios del método hacerHanoi() de la clase Hanoi:

```

public void hacerHanoi(int de, int a, int intermedio, int n) {
    if (n == 1) {
        torres[a].meter(torres[de].sacar());
        logger.debug("Mueve anillo de: " + (de+1) + " a: " + (a+1));
        this.setChanged();
        this.notifyObservers();
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) { ; }
        return;
    }

    hacerHanoi(de, intermedio, a, n - 1);
    hacerHanoi(de, a, intermedio, 1);
    hacerHanoi(intermedio, a, de, n - 1);
}

```

Claramente, cuando n es 1 (solamente 1 anillo) se cambia el estado del observado (el objeto Hanoi) y se notifica a todos los observadores (siguiendo el patrón de diseño). Luego se llama a la función para que detenga la ejecución un momento ya que si no la interface dibujaría tan rápido que no se verían los cambios de manera correcta.

INICIALIZACION DE HANOI

En WinHanoi programamos todo el control del programa que consiste en crear uno nuevo, llamar a hacer el algoritmo y salir. Aquí es donde se le asigna al Hanoi creado la interface como observador del mismo:

```
void fileNuevo_ActionPerformed(ActionEvent e) {
    String nb = JOptionPane.showInputDialog("Ingrese el número de discos");
    int nDiscos = Integer.parseInt(nb);

    logger.debug("Hace nuevo hanoi con " + nDiscos + " discos");
    objHanoi = new Hanoi(nDiscos);

    logger.debug("Le añade el observadore panelHanoi: " + pnlHanoi.toString());
    objHanoi.addObserver(pnlHanoi);

    logger.debug("lo marca como iniciado para que sea pintado");
    objHanoi.iniciado();
}
```

Luego, cuando queremos comenzar el programa, lo que hacemos es crear un thread que inicie un proceso de resolución de Hanoi y que llame a los observadores cuando nuestro modelo Hanoi cambie:

```
void fileHacer_ActionPerformed(ActionEvent e) {
    logger.debug("Comienza el thread con " + objHanoi.toString());
    Runnable worker = new Runnable() {
        public void run() {
            objHanoi.hacer();
        }
    };
    Thread t = new Thread(worker);
    t.start();
}
```

COMPLEJIDAD DEL ALGORITMO DE HANOI

Veamos cual sería la complejidad del algoritmo de Hanoi. Para ello necesitamos saber cuántas operaciones va a realizar el algoritmo para un problema de tamaño n . Hagamos una pequeña tabla:

Tamaño	Nro Operaciones
1	1
2	3
3	7
4	15
5	31
6	63

Vemos claramente que la complejidad nos lleva a la ecuación: $2^n - 1$

La complejidad del algoritmo de Hanoi es exponencial, lo cual nos indica que el algoritmo crecerá muchísimo para un número relativamente pequeño.

QUICKSORT

El **ordenamiento rápido** (**quicksort** en inglés) es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a $n \log n$.

Fue creado por C.A.R Hoare en 1960. El funcionamiento del mismo toma en cuenta una lista de objetos ordenables y es como sigue:

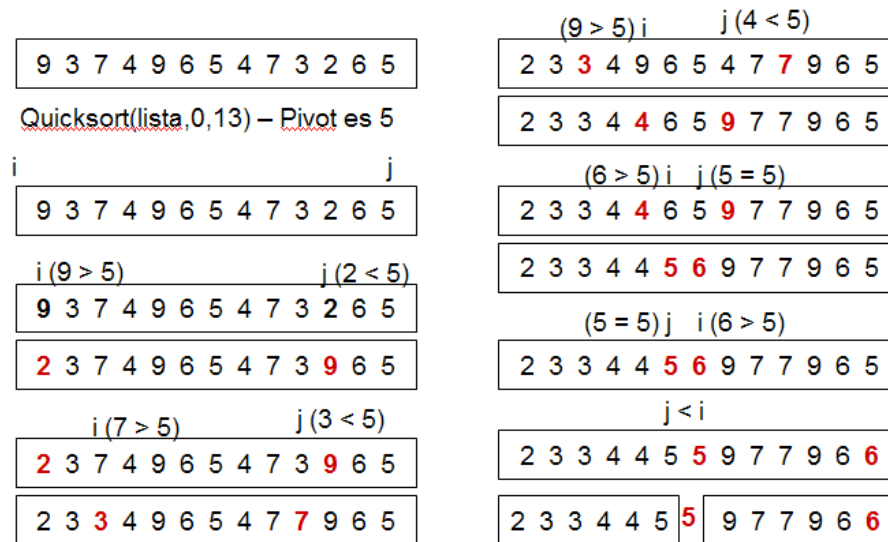
- Elegir un elemento de la lista
- Se busca su posición en la lista
- Se hacen dos listas, una con los menores y otra con los mayores a este elemento
- Mientras las listas tengan un largo mayor a 1, se repite el proceso con las sublistas

El algoritmo propuesto tiene la siguiente forma:

```
public static void quicksort(int a[], int izq, int der) {
    int i = izq;
    int j = der;
    int pivote = a[(izq + der) / 2];
    do {
        while(a[i] < pivote) i++;
        while(a[j] > pivote) j--;
        if (i <= j) {
            int aux = a[i];
            a[i] = a[j];
            a[j] = aux;
            i++;
            j--;
        }
    } while(i <= j);
    if (izq < j) quicksort(a, izq, j);
    if (der > i) quicksort(a, i, der);
}
```

Para ver el procedimiento de manera más grafica aquí un ejemplo donde se ve la primera iteración que consiste en escoger un pivote y obtener las dos sublistas de menores y mayores. Luego se aplica el mismo procedimiento a las sublistas hasta que cada una tenga 1 elemento⁵.

⁵ Ver una excelente animación aquí: <http://dvegaf.iespana.es/quicksort.html>



COMPLEJIDAD DE QUICKSORT

En cada iteración del algoritmo básicamente se recorren todos los elementos del arreglo para ver cual elemento va a la sublista de los menores y cuál va a la de los mayores.

Lo que nos resta saber es cuantas veces debemos iterar para llegar a ordenar el arreglo. Si cada vez obtenemos dos sublistas entonces la operación que hacemos cada vez es dividir entre 2 el número de elementos en el arreglo. Hagamos un ejemplo con un arreglo de 20 elementos:

Número de listas	Número de elementos en cada lista	Total operaciones en iteración
1	20	20
2	10	$2 \times 10 = 20$
4	5	$4 \times 5 = 20$
8	3 y 2	$4 \times 3 + 4 \times 2 = 20$
16	1 y 2	$12 \times 1 + 4 \times 2 = 20$

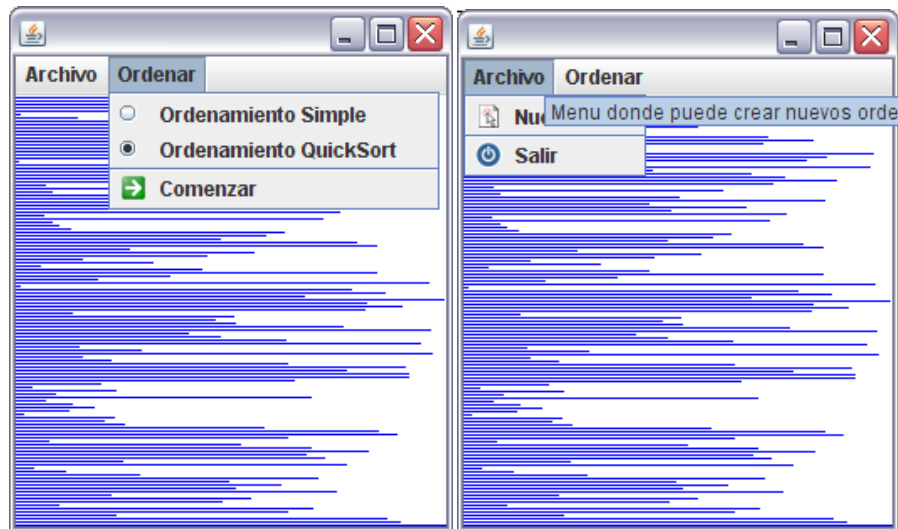
Se puede ver que a lo máximo se realizaran 6 iteraciones. ¿Qué relación podemos encontrar entre 20 y 6? Pensemos, si se hacen 5 iteraciones se pueden ordenar hasta 16 elementos, si se hacen 6 iteraciones se pueden ordenar 32 elementos, lo cual está dentro del caso de los 20 elementos: es la inversa de la exponenciación.

Entonces la complejidad de Quicksort es:

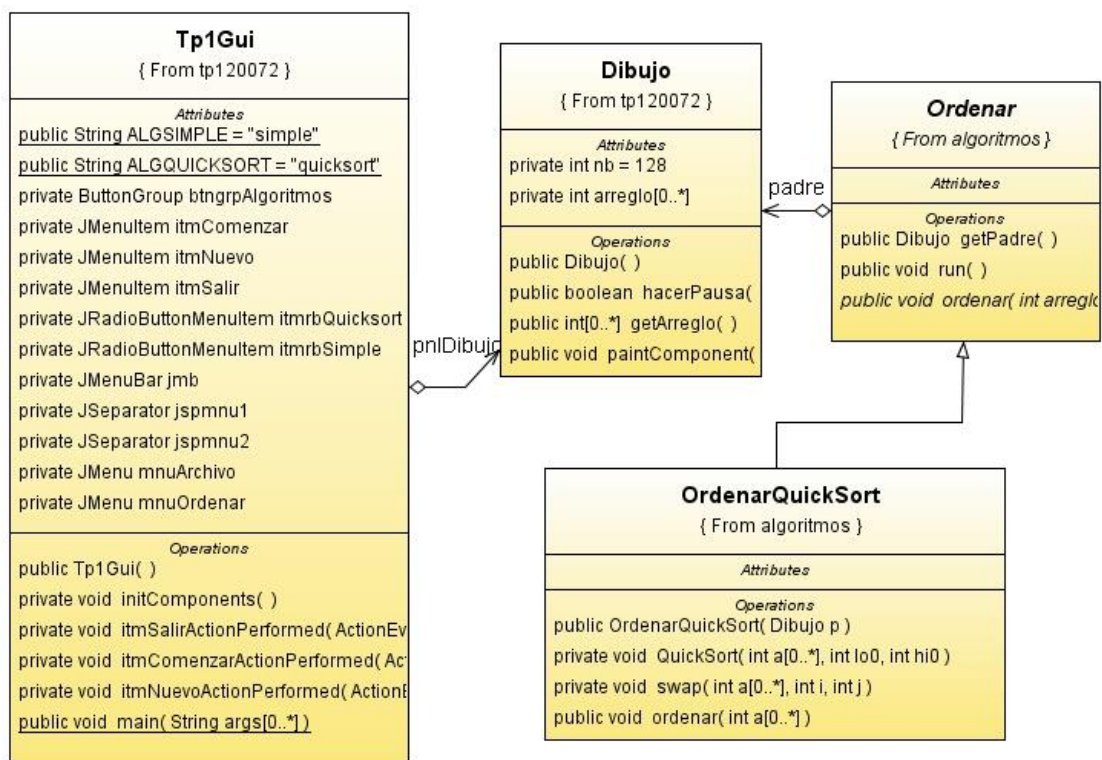
$$\vartheta(n) = n \log_2 n$$

APLICACIÓN GRAFICA DE QUICKSORT

Realizar la implementación del algoritmo de Quicksort para demostrar gráficamente su funcionamiento. La interface grafica debe ser la siguiente:



El diagrama de clases propuesto para la aplicación utiliza tanto el patrón Observer como el concepto de Threads:



Detalles de la implementación

El programa, al recibir la confirmación para comenzar el algoritmo realiza lo siguiente:

```

private void itmComenzarActionPerformed(ActionEvent evt) {
    if (this.pnlDibujo == null)
        JOptionPane.showMessageDialog(this,
            "Debe crear un nuevo panel primero");
}
  
```

```

        if (this.btngroupAlgoritmos.getSelection() != null) {
            ButtonModel btn = this.btngroupAlgoritmos.getSelection();
            Ordenar algoritmo = null;
            if (btn.getActionCommand().equals(ALGSIMPLE)) {
                ;
            } else if (btn.getActionCommand().equals(ALGQUICKSORT)) {
                algoritmo = new OrdenarQuickSort(this.pnlDibujo);
            }
            Thread t = new Thread(algoritmo);
            t.start();
        } else {
            JOptionPane.showMessageDialog(this,
                "Debe elegir un algoritmo");
        }
    }
}

```

La clase dibujo contiene lo necesario para dibujar el arreglo en el Panel (se ha omitido el código evidente)

```

public class Dibujo extends JPanel {
    ...
    public Dibujo() {
        arreglo = new int[nb];
        for(int i=0; i<nb; i++) {
            arreglo[i] = (int) (256 * Math.random()) + 1;
        }
        this.setPreferredSize(new Dimension(256,256));
        this.setBackground(Color.WHITE);
    }

    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);
        this.setForeground(Color.BLUE);
        for (int i=0; i<nb; i++) {
            gc.drawLine(0, i*2 + 1, arreglo[i], i*2 + 1);
        }
    }
}

```

La clase Ordenar nos permite agrupar cualquier algoritmo de ordenamiento para analizar su funcionamiento:

```

public abstract class Ordenar implements Runnable {

    public void run() {
        this.ordenar(padre.getArreglo());
    }

    public abstract void ordenar(int[] arreglo);
}

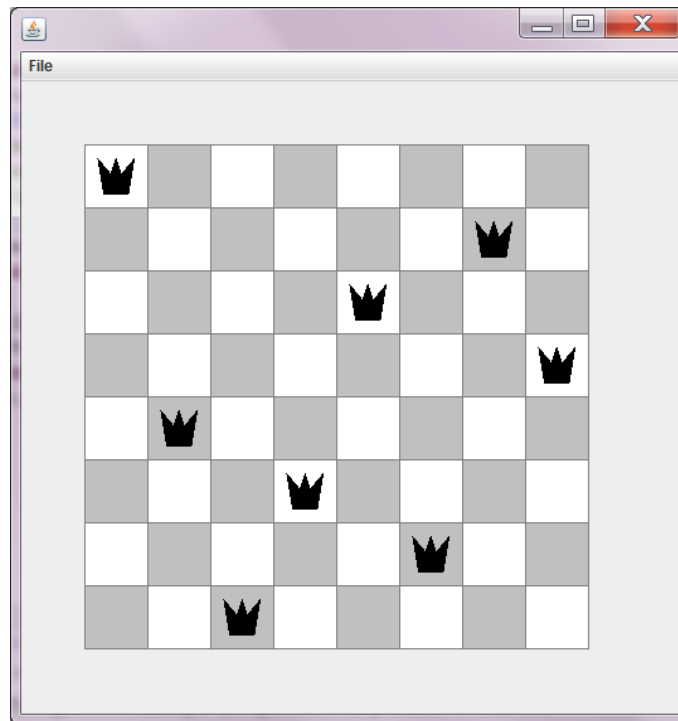
```

Se ha dejado como ejercicio la implementación de los algoritmos.

PROBLEMA N-REINAS

Un problema muy conocido y muy utilizado es el de las N-Reinas. El objetivo es muy sencillo:

En un tablero de ajedrez, colocar 8 reinas sin que ninguna de ellas pueda comerse.



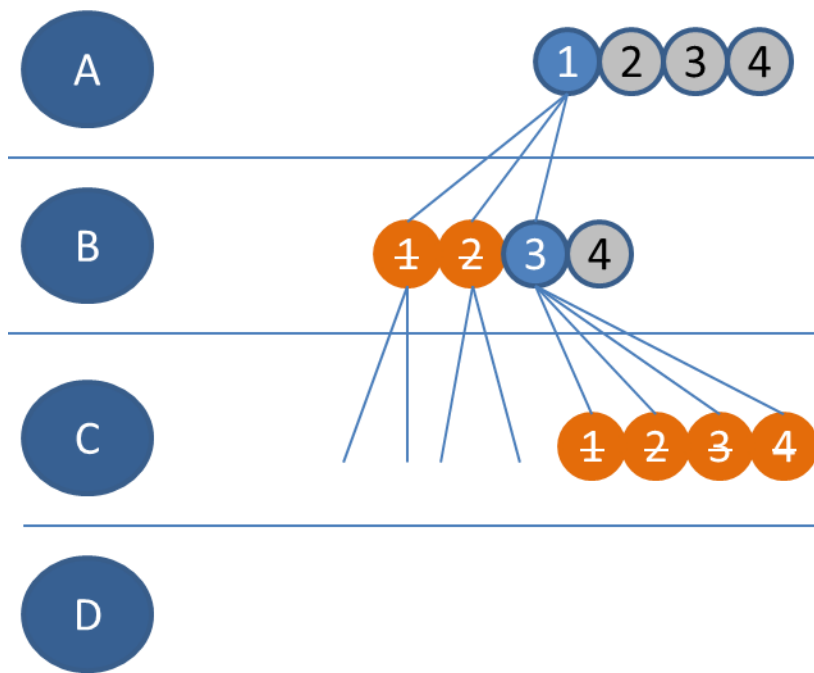
Ahora, el problema es identificar cómo vamos a resolver ese problema con la recurrencia que se ve en este capítulo. Lo primero que debemos hacer es simplificar el problema al mínimo. En nuestro caso, hay que simplificar este problema de la siguiente manera: 4 reinas en un tablero 4x4.

PROBLEMA SIMPLIFICADO

En nuestro problema simplificado colocamos la primera reina en el primer lugar posible de la primera columna y tenemos la siguiente notación:

	A	B	C	D
0	♛			
1				
2				
3				

El problema es encontrar la próxima posición en la columna B. Pongamos nuestras decisiones de la siguiente manera:



Aquí vemos que nuestro algoritmo debería hacer lo siguiente:

1. Próxima columna A
2. Próxima posición 1
3. A1 es una posición correcta? Si, entonces ver próxima columna
4. A1B1 es una configuración correcta? No, entonces avanzar en fila
5. A1B2 es una configuración correcta? No, entonces avanzar en fila
6. A1B3 es configuración correcta? Si, escoger próxima columna
7. A1B3C1? No, ... y así hasta A1B3C4 que NO es válido
8. No se ha encontrado ninguna solución posible, entonces se cambia la primera hipótesis
9. A2, y así se encuentra A2B4C2D3

Podemos ver una recurrencia posible en este código:

```
public boolean colocarReinaEnColumna(int col) {

    // Condición de fin de recurrencia
    if (col >= tablero.length)
        return true;

    for (int fila = 0; fila < tablero.length; fila++) {

        if(esConfiguracionCorrecta (fila, col)) {
            this.ponerReina(fila, col);

            // Aquí es la importancia de la recurrencia
            if (colocarReinaEnColumna(col+1))
                return true;
        }
    }
}
```

```
        this.quitarReina(fila, col);  
    }  
}  
  
return false;  
}
```

Para cada columna se recorre una a una las filas. Cuando tenemos una configuración que parece correcta, se coloca la reina en el lugar escogido y se llama al mismo método para resolver un problema idéntico, pero más pequeño.

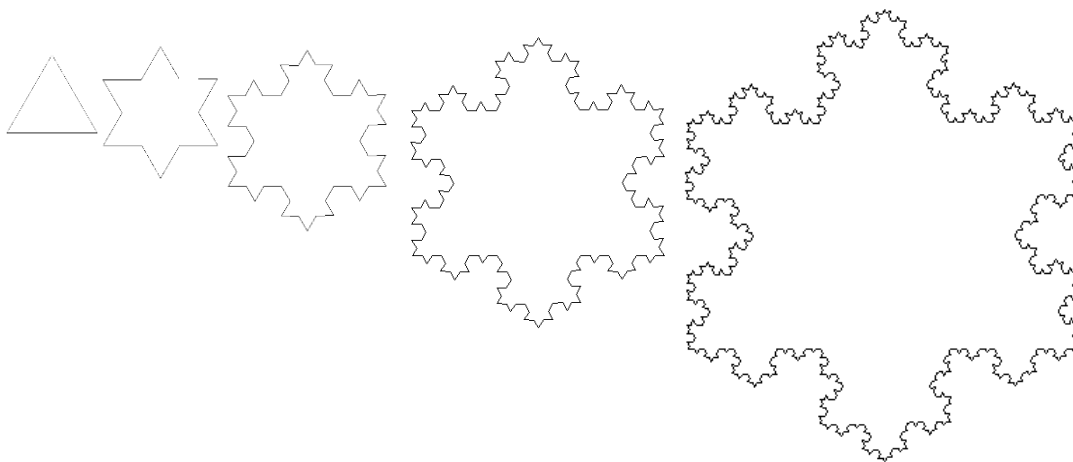
RECURRENCIA GRÁFICA

En esta sección se verá la recurrencia en su forma gráfica. De esta manera se verán los problemas más interesantes. Es importante primero conocer un modelo que nos permita manejar acciones separando tanto la capa de presentación como la lógica de negocios.

CURVA DE VON KOCH

El creador en 1904 de este monstruo fue Niels Fabian Helge von Koch, matemático sueco. Partamos de un triángulo equilátero de lado unidad. Dividimos en tres partes iguales de longitud $1/3$ cada lado. Sustituimos el segmento central por dos segmentos de tamaño idéntico formando un diente como muestra la animación en la iteración $n=1$. Tenemos una curva poligonal P_1 de longitud $3 \cdot 4 \cdot 1/3 = 4$. Repetimos la operación ($n=2$) con cada uno de los cuatro nuevos segmentos de cada uno de los "lados". Obtendremos así la curva P_2 de longitud $3 \cdot 4^2 \cdot 1/3^2 = 16/3$. La iteración indefinida nos proporciona la isla de Koch o copo de nieve de Koch.⁶

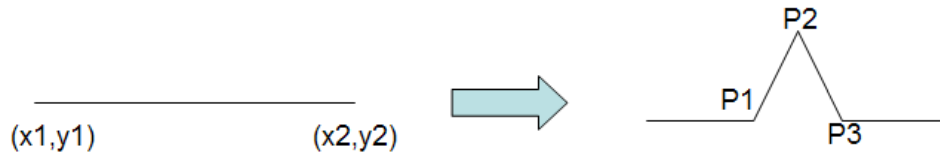
Veamos la progresión de la curva para cada repetición:



RECURSOS MATEMÁTICOS PARA LA RESOLUCIÓN DE VON KOCH

Para ir de la línea recta a las 4 líneas que forman la iteración de Koch necesitamos lo siguiente:

⁶ Tomado de <http://www.dmae.upm.es/cursofractales/capitulo1/2.html>



$$\text{angulo} = \tan^{-1} \frac{y2 - y1}{x2 - x1}$$

$$\text{unTercioX} = \frac{x2 - x1}{3}, \quad \text{unTercioY} = \frac{y2 - y1}{3}$$

$$\text{unTercioDistancia} = \frac{\sqrt{(y2 - y1)^2 + (x2 - x1)^2}}{3}$$

Con estas formulas llegamos a los puntos P1, P2 y P3:

$$p1x = x1 + \text{unTercioX}, \quad p1y = y1 + \text{unTercioY}$$

$$p2x = p1x + \text{unTercioDistancia} * \cos(\text{angulo} - \frac{\pi}{3}), \quad p2y = p1y + \text{unTercioDistancia} * \sin(\text{angulo} - \frac{\pi}{3})$$

$$p3x = x1 + 2 * \text{unTercioX}, \quad p3y = y1 + 2 * \text{unTercioY}$$

Una vez tenemos las ecuaciones, el algoritmo es bastante sencillo:

```
Vonkoch(punto P1, punto P5, int n) {
  Si (n=1)
    Hacer línea entre P1 y P5
  Sino
    Encontrar los 3 puntos intermedios P2, P3 y P4
    Vonkoch(P1, P2, n-1)
    Vonkoch(P2, P3, n-1)
    Vonkoch(P3, P4, n-1)
    Vonkoch(P4, P5, n-1)
}
```

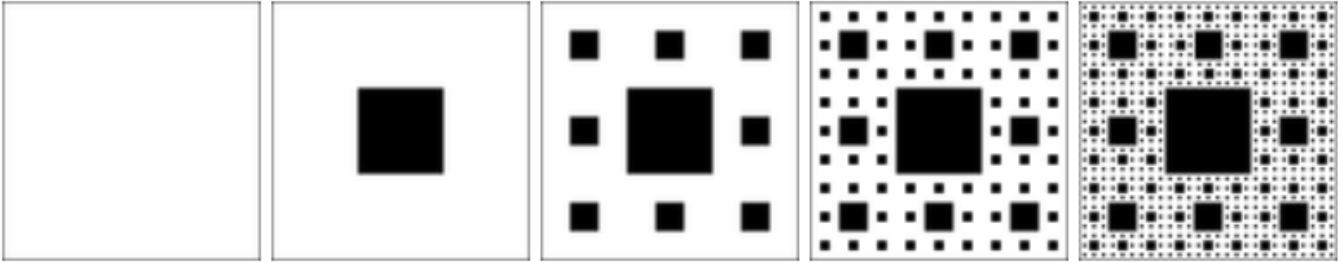
TRIANGULO Y CUADRADO DE SIERPINSKY

Estos dos son problemas muy similares al de Von Koch solamente que nos dan figuras diferentes. El triángulo de Sierpinsky es una figura que parte de un triángulo y en cada iteración construye sub triángulos dentro de los mismos. Aquí se ve un ejemplo⁷:



⁷ Ejemplo sacado de http://commons.wikimedia.org/wiki/File:Sierpinsky_triangle_%28evolution%29.png

De la misma manera, se puede realizar el mismo procedimiento a un cuadrado:



Esto nos da la alfombra de Sierpinsky. Cualquiera de estos dos fractales se resuelve de manera muy sencilla y similar a la que se utilizó en von Koch.

Para ver un ejemplo de la resolución de la alfombra de Sierpinsky, aquí el código que se encarga de la recurrencia:

```
public void hacerSierpinsky(
    int x1, int y1,
    int ancho, int alto, int n,
    Graphics gc) {

    int pAncho = ancho / 3;
    int pAlto = alto / 3;
    if (n == 1) {
        gc.drawRect(x1, y1, ancho, alto);
        gc.drawRect(x1 + pAncho, y1 + pAlto, pAncho, pAlto);
        return;
    } // end of if (n == 1)

    /**
     * ***** 1 * 2 * 3 *
     * ***** 4 *   * 5 *
     * ***** 6 * 7 * 8 * *****
     */
    hacerSierpinsky(x1, y1, pAncho, pAlto, n - 1, gc);
    hacerSierpinsky(x1 + pAncho, y1, pAncho, pAlto, n - 1, gc);
    hacerSierpinsky(x1 + 2 * pAncho, y1, pAncho, pAlto, n - 1, gc);
    hacerSierpinsky(x1, y1 + pAlto, pAncho, pAlto, n - 1, gc); // 4
    hacerSierpinsky(x1 + 2 * pAncho, y1 + pAlto, pAncho, pAlto, n - 1, gc); // 5
    hacerSierpinsky(x1, y1 + 2 * pAlto, pAncho, pAlto, n - 1, gc); // 6
    hacerSierpinsky(x1 + pAncho, y1 + 2 * pAlto, pAncho, pAlto, n - 1, gc); // 7
    hacerSierpinsky(x1 + 2 * pAncho, y1 + 2 * pAlto, pAncho, pAlto, n - 1, gc); // 8
}
```

UN POCO DE LOGO

Logo es un lenguaje de alto nivel en parte funcional en parte estructurado, de muy fácil aprendizaje, razón por la cual suele ser el lenguaje de programación preferido para trabajar con niños y jóvenes. Fue diseñado con fines didácticos por Danny Bobrow, Wally Feurzeig y Seymour Papert, los cuales se basaron en las características del lenguaje Lisp.

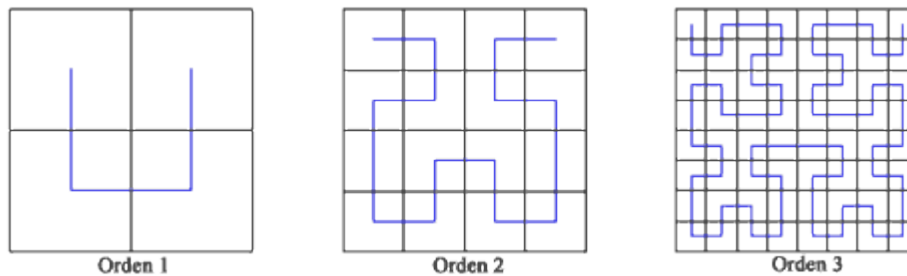
Una característica más explotada de Logo es poder producir gráficos de tortuga, es decir, poder dar instrucciones a una tortuga virtual, que en algunas versiones es un triángulo, mediante palabras escritas, por ejemplo:

- forward 100 (la tortuga camina hacia delante 100 pasos)
- turnright 90 (la tortuga se gira hacia la derecha 90°)
- turnleft 30 (la tortuga se gira hacia la izquierda 30°)

Se puede crear una pequeña librería que permita dibujar como si uno tuviera una tortuga.

CURVA DE HILBERT

Un último ejemplo de recurrencia gráfica es la llamada curva de Hilbert. Estas curvas son interesantes porque permiten llenar una superficie con una línea. Es decir, al infinito, la línea comienza a tener una superficie (lo cual trajo muchos problemas de interpretación a los matemáticos en su tiempo). Veamos la progresión de esta curva:



Para solucionar este problema, utilizando un funcionamiento como el de logo, analice el siguiente algoritmo:

```
private void generaHilbert(Graphics g, int nivel, int direccion,
                           double distancia) {
    if (nivel <= 0)
        return;

    obj.Girar(-90 * direccion);
    generaHilbert(g, nivel - 1, -direccion, distancia);
    obj.Trazar(g, distancia);
    obj.Girar(90 * direccion);
    generaHilbert(g, nivel - 1, direccion, distancia);
    obj.Trazar(g, distancia);
    generaHilbert(g, nivel - 1, direccion, distancia);
    obj.Girar(90 * direccion);
    obj.Trazar(g, distancia);
    generaHilbert(g, nivel - 1, -direccion, distancia);
    obj.Girar(-90 * direccion);
}
```

Fíjese que los trazos son siempre del mismo tamaño y por lo tanto la distancia no cambia. Lo único que cambia es el nivel.

PREGUNTAS SOBRE COMPLEJIDAD Y RECURRENCIA

Pregunta 1. Recurrencia simple

Explique el concepto de recurrencia en programación. A modo de práctica resuelva los siguientes ejercicios con una función recurrente (utilice programación defensiva para los parámetros):

- La suma de los primeros n números impares
- La función debe imprimir los números naturales hasta n en orden inverso. Por ejemplo $f(10) = 10\ 9\ 8\ 7\ 6\ 5\ 4\ 3\ 2\ 1\ 0$
- Si $n=1$, imprimir 'ab', si $n=2$, imprimir 'aabb' y así sucesivamente. Escribir la función que toma como argumento un entero n .

| Pregunta 2. Complejidad Burbuja

Explique cuál es la complejidad de un algoritmo de ordenamiento como el algoritmo de burbuja. Fundamente su respuesta para llegar a la expresión matemática de la complejidad de este tipo de algoritmos

| Pregunta 3. Programa carrera de caballos

Escriba un pequeño programa donde se simule una carrera de caballos. El programa debe contar con 2 clases, Programa y Caballo. Además, Caballo debe implementar Runnable para que pueda ser utilizado como Thread. Se deja a criterio del usuario el código de `run()` para todos los caballos simulando una carrera.

| Pregunta 4. Hanoi 64

Se dice que cuando el universo comenzó se inició un Hanoi con 64 discos. Y que, de la misma manera, cuando el Hanoi sea resuelto, el universo terminará. Si el universo comenzó hace 13.700.000.000 años⁸ y los monjes en Hanoi logran mover correctamente un anillo en 1 segundo. ¿Cuánto tiempo le queda de vida al universo?

| Pregunta 5. Quicksort mejorado

Se puede ver fácilmente que Quicksort se comporta muy mal para arreglos pequeños (de 50 elementos o menos), adicionalmente, dependiendo cómo elegimos el pivote, el algoritmo mejora o no. ¿Qué alternativas se tienen para mejorar el algoritmo? Proponga una implementación con mejora.

| Pregunta 6. Complejidad Vonkoch

¿Cuál es la complejidad del algoritmo de copo de nieve de von Koch?

| Pregunta 7. Diseño de aplicaciones Von Koch y Sierpinsky

¿Describa y dibuje el diagrama de clases que propondría para la realización de un programa que permita elegir el fractal a mostrar: von Koch, triángulo de Sierpinsky o cuadrado de Sierpinsky y que permita elegir la profundidad? Indique los patrones de diseño que se utilizan y también si utiliza herencia o no.

⁸ Ver http://es.wikipedia.org/wiki/Edad_del_Universo

ESTRUCTURA DE DATOS

“Cuidado con los bugs en el anterior código. Solamente he probado que es correcto, no lo he usado” Donald Knuth

En esta unidad se ven en detalle la mayor parte de las estructuras de datos básicas. A partir de estas estructuras se puede formar e idear otras. Además, el estudio de estas estructuras permite comenzar el estudio de conceptos de programación avanzados como la programación dinámica y otras.

Tomando en cuenta la cita del capítulo; es demás indicar al lector que, a partir de este momento, todo código debe ser testeado de manera exhaustiva para que el mismo tenga alguna validez profesional. Si no, toda la construcción de su programa solamente servirá a efectos de prototipo.

ESTRUCTURAS DE DATOS ESTATICAS

Son estructuras de datos que, una vez definidas, no pueden ser cambiadas por otras de otra naturaleza, o de diferente tamaño. Este tipo de estructuras no se pueden utilizar para problemas donde la información cambie al momento de la ejecución.

ESTRUCTURAS ESTATICAS LINEALES: ARREGLOS

Es una lista contigua (físicamente inclusive) de datos del mismo tipo que son accesibles por un índice. Es muy probable que el lector ya haya utilizado este tipo de estructuras de manera continua.

Para definir un arreglo de 10 enteros:

```
int[] a = new int[10]
```

a[5] significa que, del lugar donde ha sido declarado el arreglo en memoria, se corren 5 lugares y se obtiene el entero que ha sido colocado ahí.

No olvide que los elementos son siempre del mismo tipo y que el índice siempre comienza en 0.

Ejemplo: Crear un arreglo de 100 enteros. Cada valor debe llevar un número aleatorio entre 1 y 20.

```
int[] arregloEnteros = new int[100];
for(int i=0; i<arregloEnteros.length; i++) {
    arregloEnteros[i] = (int) (Math.random() * 20.0) + 1;
}
```

USO DE ARREGLOS

Los arreglos son utilizados en casi todos los programas que se realizan (por no decir en todos). El concepto de tener que manejar varios elementos de un mismo tipo es necesario todo el tiempo. En el anterior capítulo se han utilizado los arreglos para referenciar a las 3 torres. Sabemos que siempre van a ser 3, por ello utilizamos un arreglo.

ESTRUCTURAS ESTATICAS NO LINEALES: MATRICES

Son estructuras de tipo arreglo en las que cada objeto es a su vez un arreglo. Se puede decir que es un arreglo de arreglos. En otros casos se entiende la idea como un arreglo en dos dimensiones. En este caso el índice es reemplazado por dos índices.

En JAVA: matriz de 20 filas por 30 columnas

```
int[][] a = new int[20][30];
```

Las matrices, al igual que los arreglos, pueden contener tipos primitivos o tipos objeto. Sin embargo, siempre del mismo tipo.

Ejercicio: Llenar una matriz de 8 x 8 de la forma

1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	15

El código para resolver el ejercicio puede ser el siguiente:

```
int filas = 8;
int columnas = 8;
int[][] m = new int[filas][columnas];
for(int i=0; i<filas; i++) {
    int valorInicial = i+1;
    for(int j=0; j<columnas; j++) {
        m[i][j] = valorInicial;
        valorInicial++;
    }
}
```

USO DE MATRICES

Matemáticas

Todos nos acordamos aquellas clases de algebra lineal donde se utilizaban matrices. Las estructuras de datos matrices representan exactamente eso que estudiábamos.

Relación de Datos

Relaciones entre dos listas. Este es un caso particular cuando queremos marcar que entre dos elementos hay una relación: por ejemplo, si se tiene una lista de alumnos y una lista de clases.

Imágenes

Las imágenes no son más que matrices (bastante grandes) cuyo valor de color se encuentra en cada celda.

MATRICES: JUEGO DE LA VIDA DE CONWAY

Juego de la vida es un ejercicio donde se puede ver la representación de la reacción de un conjunto de sujetos cuando interaccionan entre sí en un entorno controlado por ciertas reglas. Este tipo de algoritmos se utiliza sobre todo para simular colonias de hormigas, tráfico vehicular, etc.

En el ejercicio actual se trata de realizar un programa, donde, a partir de una configuración inicial, se desarrolle automáticamente la animación de las etapas del juego de la vida de Conway.

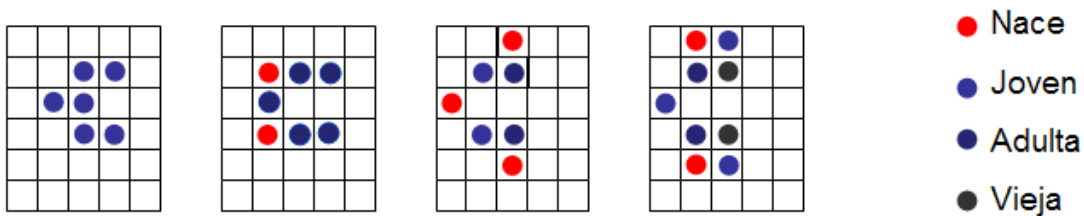
Una configuración inicial significa un ordenamiento de los valores dentro de una matriz de una manera particular. Las configuraciones subsecuentes son resultado de operaciones en cada una de las celdas de la configuración inicial.

Reglas

Estas son las operaciones que se realizan en cada una de las celdas para ir modificando una configuración y llegar a la siguiente. Para el juego de la vida de Conway solamente se requieren dos reglas:

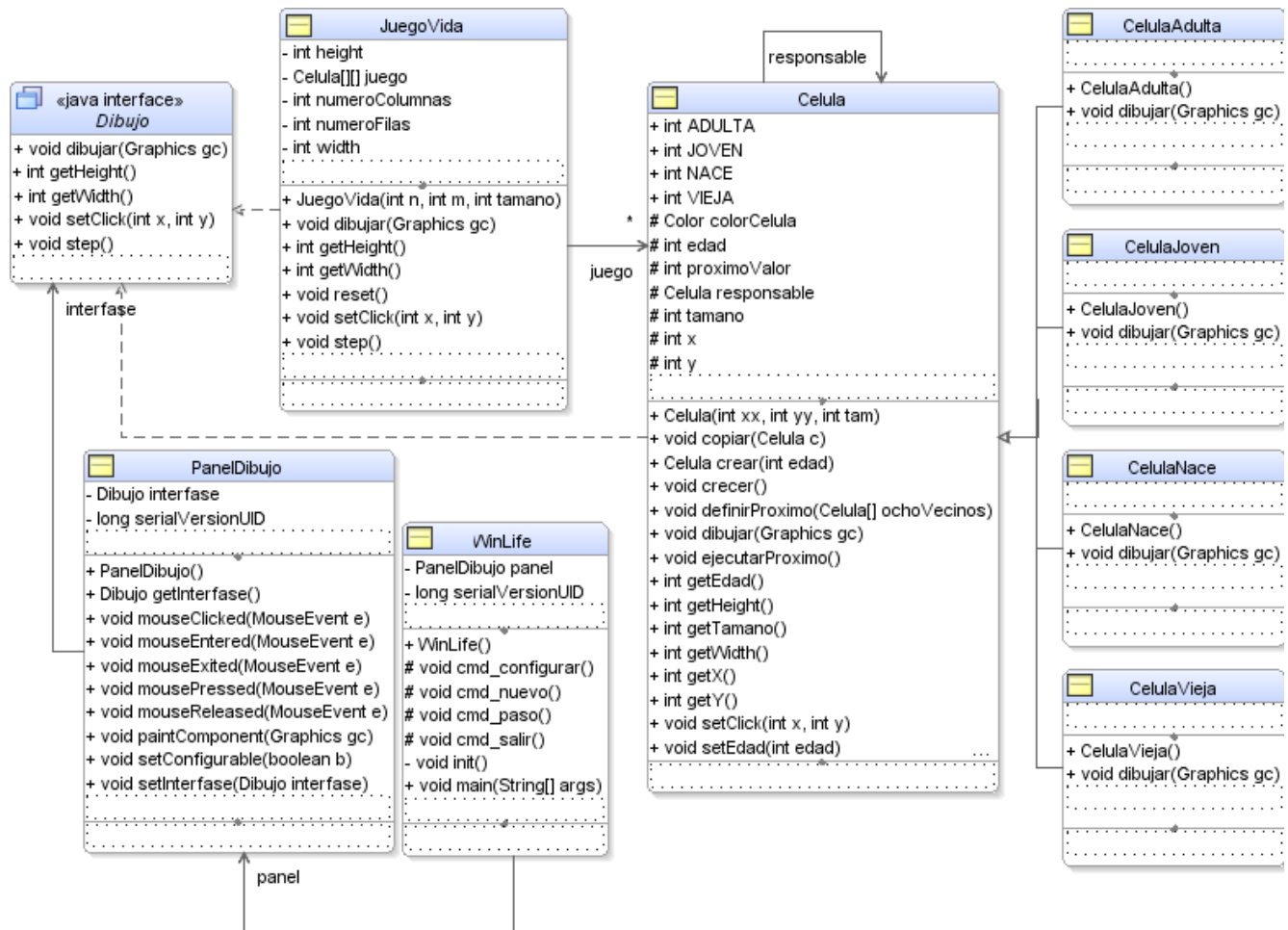
1. Una célula muerta con exactamente 3 células vecinas vivas nace.
2. Una célula viva con 2 o 3 células vecinas vivas, sigue viva, en otro caso muere o permanece muerta.

Aquí se pueden ver las 4 primeras etapas a partir de la configuración inicial siguiendo las 2 reglas:



IMPLEMENTACIÓN DEL JUEGO DE LA VIDA

La implementación consta del siguiente diseño (similar a los que vimos anteriormente):



Básicamente lo que hay que hacer es programar la próxima iteración y dejar que el Observer llame a `paintComponent()` para que veamos el resultado. Aquí se puede ver el código para un paso:

```

public void step() {
    for (int i = 0; i < this.numeroFilas; i++) {
        for (int j = 0; j < this.numeroColumnas; j++) {
            Celula[] vecinos = new Celula[8];

            try {
                vecinos[0] = juego[i - 1][j - 1];
            } catch (ArrayIndexOutOfBoundsException err) {
                vecinos[0] = null;
            }
            try {
                vecinos[1] = juego[i - 1][j];
            } catch (ArrayIndexOutOfBoundsException err) {
                vecinos[1] = null;
            }
            try {
                vecinos[2] = juego[i - 1][j + 1];
            } catch (ArrayIndexOutOfBoundsException err) {
                vecinos[2] = null;
            }
        }
    }
}

```

```

        try {
            vecinos[3] = juego[i][j - 1];
        } catch (ArrayIndexOutOfBoundsException err) {
            vecinos[3] = null;
        }
        try {
            vecinos[4] = juego[i][j + 1];
        } catch (ArrayIndexOutOfBoundsException err) {
            vecinos[4] = null;
        }
        try {
            vecinos[5] = juego[i + 1][j - 1];
        } catch (ArrayIndexOutOfBoundsException err) {
            vecinos[5] = null;
        }
        try {
            vecinos[6] = juego[i + 1][j];
        } catch (ArrayIndexOutOfBoundsException err) {
            vecinos[6] = null;
        }
        try {
            vecinos[7] = juego[i + 1][j + 1];
        } catch (ArrayIndexOutOfBoundsException err) {
            vecinos[7] = null;
        }
        juego[i][j].definirProximo(vecinos);
    }
}

for (int i = 0; i < this.numeroFilas; i++) {
    for (int j = 0; j < this.numeroColumnas; j++) {
        juego[i][j].ejecutarProximo();
    }
}
}

```

Y la programación de las reglas se hace en cada célula con el método `ejecutarProximo()`

```

/**
 * Los ocho vecinos necesarios son
 * --- 0 --- 1 --- 2 <br>
 * --- 3 --- X --- 4 <br>
 * --- 5 --- 6 --- 7 <br>
 * Si los vecinos no existen (los bordes del sistema)
 * entonces se coloca el vecino a null
 * @param ochoVecinos
 */
public void definirProximo(Celula[] ochoVecinos) {
    int numeroCelulasVecinasVivas = 0;
    for (int i=0; i<8; i++) {
        // si está viva
        if (ochoVecinos[i] != null &&
            ochoVecinos[i].getEdad() > 0)
            numeroCelulasVecinasVivas++;
    }
}

```

```

        this.proximoValor = -1;
        if (this.edad == 0 && numeroCelulasVecinasVivas == 3)
            this.proximoValor = Celula.NACE;
        if (this.edad > 0 &&
            (numeroCelulasVecinasVivas == 2 || numeroCelulasVecinasVivas == 3))
            this.proximoValor = this.edad + 1;
        if (this.proximoValor < 0)
            this.proximoValor = 0;
    }

```

En este código se puede ver el uso de comentarios encima de los métodos. El comentario esta en lenguaje html. Ver el anexo para el uso de Javadoc para la documentación de sus programas.

MATRICES: IMÁGENES

Una imagen no es más que una lista de pixeles ordenados en una matriz del tamaño de la imagen. Por ejemplo, una imagen de 300x200 es una matriz de 300x200 pixeles.

Un pixel no es más que un entero que guarda el valor del color de ese punto. Los colores posibles se forman a partir de tonalidades de Rojo, Verde y Azul (256 de cada uno). Entonces: $\text{color} = \text{tR} + \text{tV} * 256 + \text{tA} * 256 * 256$.

Tenemos 3 opciones para representar una imagen de 100 x 100:

1. Se utilizan matrices de 100x100, una para los tonos de Rojo, otro para los de Verde y otros para los de Azul

```

int[][] rojos = new int[100][100];
int[][] verdes = new int[100][100];
int[][] azules = new int[100][100];

```

Luego, en el `paintComponent()` del panel que va a pintar la imagen se puede pintar el punto de la manera siguiente:

```

gc.setColor(new Color(rojos[i][j], verdes[i][j], azules[i][j]));
gc.drawLine(i, j, i, j);

```

2. Se utiliza una sola matriz, pero el color está formado por los 3 tonos.

```

int[][] imagen = new int[100][100];
int[i][j] = r + g * 256 + b * 256*256; // para cada i, j

```

Luego, en el `paintComponent()` del panel que va a pintar la imagen se puede pintar el punto de la manera siguiente:

```

gc.setColor(new Color(imagen[i][j]));
gc.drawLine(i, j, i, j);

```

3. Se utiliza un arreglo para representar todos los pixeles de la imagen. Esto es útil para algunos métodos que necesitan exponer a la imagen como un arreglo. En este caso el color y la posición *i, j* se calculan como sigue:

```

int[] imagen = new int[100*100];
imagen[i * 100 + j] = r + g * 256 + b * 256*256; // para cada i, j

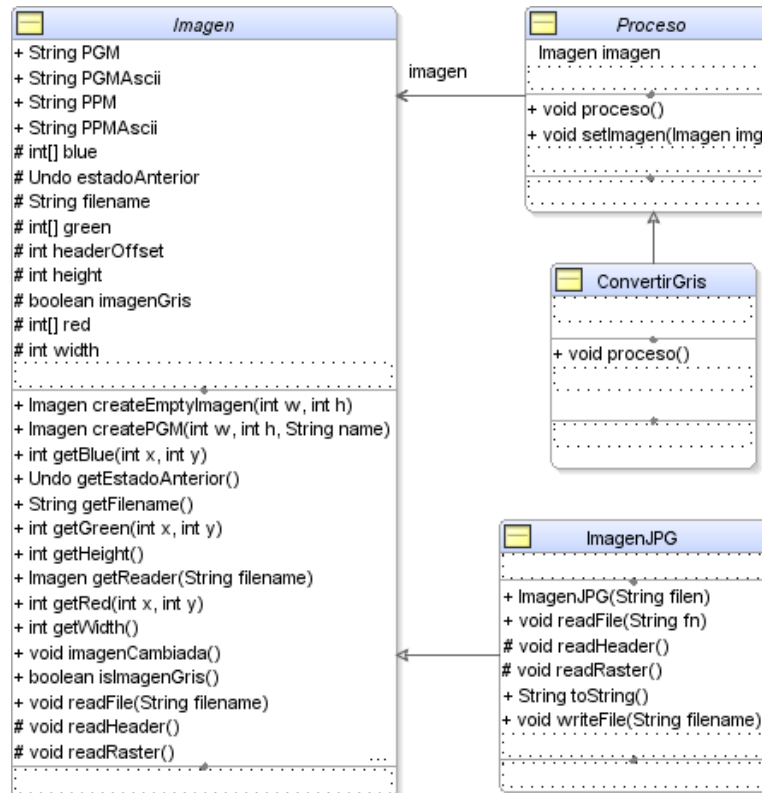
```

Cuando queremos solamente utilizar tonos de gris, los 3 valores (R, G y B) de los tonos tienen el mismo valor.

IMPLEMENTACIÓN DE UN PEQUEÑO MANIPULADOR DE IMÁGENES

Se trata de realizar un pequeño programa donde se carguen imágenes y se les pueda aplicar procedimientos como cambiar a grises, flip horizontal, cambiar a tonos blanco y negro, agrandar, achicar, etc.

Podemos hacer una interface grafica tan complicada como queramos, sin embargo, vamos a concentrarnos en el modelo de la imagen y los procesos que le podemos aplicar. Aquí la propuesta de diseño:



Entonces para poder leer una imagen JPG tenemos:

```

public void readFile(String fn) {
    if (fn != null)
        this.filename = fn;
    try {
        File thefile = new File(filename);
        BufferedImage bufimg = ImageIO.read(thefile);
        this.width = bufimg.getWidth();
        this.height = bufimg.getHeight();
        this.red = new int[this.width * this.height];
        this.green = new int[this.width * this.height];
        this.blue = new int[this.width * this.height];

        for (int y = 0; y < bufimg.getHeight(); y++) {
            for (int x = 0; x < bufimg.getWidth(); x++) {
                int pixel = bufimg.getRGB(x, y);

                int r = pixel & 0x00ff0000;
            }
        }
    }
}

```



```
        int g = pixel & 0x0000ff00;
        int b = pixel & 0x000000ff;
        g = g >> 8;
        r = r >> 16;
        this.setPoint(x, y, r, g, b);
    }
}
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Se puede ver el manejo de bits al asignar el valor de los tonos en la matriz.

Luego, para que el programa pueda realizar operaciones, se sigue el modelo propuesto. Aquí se tiene la implementación, como ejemplo, de la transformación de la imagen a tonos de gris:

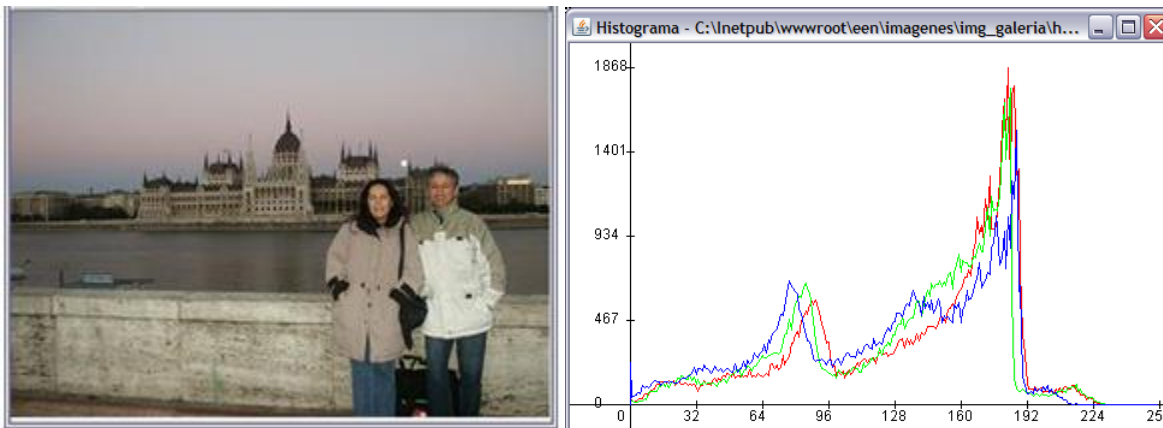
```
public void proceso() {
    super.proceso();
    int w = imagen.getWidth();
    int h = imagen.getHeight();

    for (int i=0; i<w; i++) {
        for (int j=0; j<h; j++) {
            int color =
                (imagen.getRed(i,j) +
                 imagen.getGreen(i,j) +
                 imagen.getBlue(i,j)) / 3;
            imagen.setRed(i,j,color);
            imagen.setGreen(i,j,color);
            imagen.setBlue(i,j,color);
        } // end of for (int j=0; j<h; j++)

    } // end of for (int i=0; i<w*h; i++)
    imagen.setImagenGris(true);
}
```

HISTOGRAMA

El histograma de una imagen no es más que el conteo de cada uno de los valores de color en cada una de las tonalidades. Por ejemplo, aquí se puede observar la imagen con su histograma respectivo:



Como se puede observar, hay más de 1800 píxeles que llevan tonalidades de rojo, verde y azul con valor de 180 (vea en la imagen las coordenadas).

PATRÓN DE DISEÑO: MODELO – VISTA – CONTROLADOR MVC

Model–View–Controller (MVC) es una arquitectura de software, actualmente considerada como un patrón de arquitectura esta técnica es usada en ingeniería de software. El patrón aísla la lógica de negocios de la entrada y presentación (GUI) permitiendo que el desarrollo, testeo y mantenimiento sea independiente para cada uno.

El **modelo** es la representación específica del dominio de los datos sobre el cual la aplicación opera. Esta lógica agrega significado a los datos de la base de datos (raw data – datos crudos). Cuando un modelo cambia su estado, avisa a las vistas de ese modelo (Observer) de manera que pueden refrescar la vista.

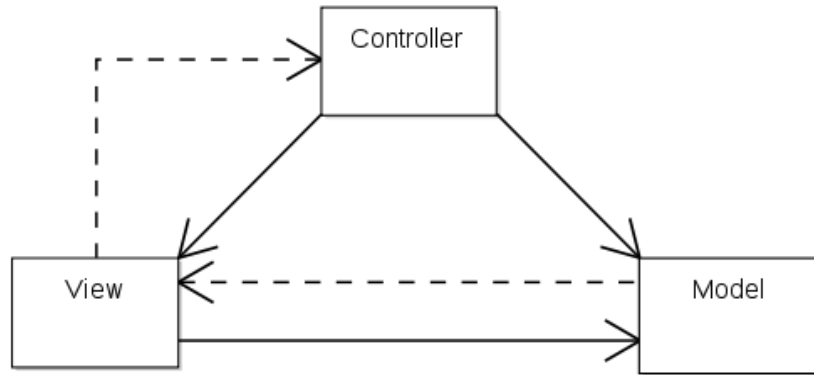
Muchas aplicaciones utilizan un mecanismo persistente como bases de datos para guardar la información. MVC no menciona nada específico respecto a la capa de datos porque se sobreentiende que está dentro del modelo. El modelo no son objetos de acceso a datos; sin embargo, en aplicaciones sencillas que tienen un pequeño dominio lógico no hay distinción entre ambos.

La **vista** muestra el modelo en una forma que sea fácil de representar para interactuar con ella. Típicamente para que sea fácil de representar por una interface de usuario. Pueden existir varias vistas para un mismo modelo.

El **controlador** recibe la entrada e inicia una respuesta haciendo llamados a modelos de objetos.

Una aplicación MVC puede ser una colección de tripletas modelo/vista/controlador cada uno responsable de un elemento de interface de usuario.

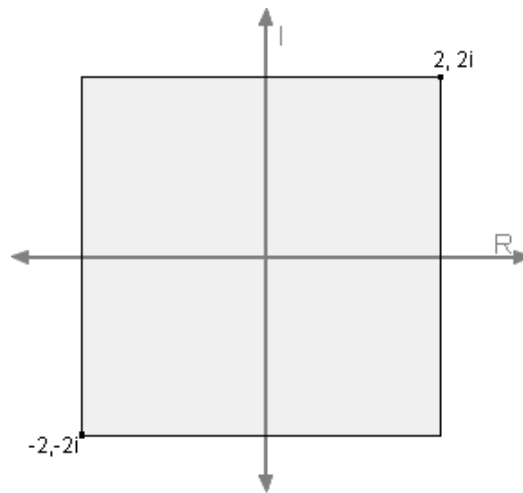
MVC es a menudo visto en aplicaciones web donde la vista es el código HTML generado por la aplicación. El controlador recibe el GET o POST y decide qué hacer con él, pasándole la mano a los objetos del dominio (el modelo) que contienen las reglas de negocio y saben cómo llevar a cabo las tareas de procesamiento como por ejemplo: nueva suscripción, eliminar usuario, etc.



MATRICES: FRACTAL DE MANDELBROT

Este es quizá el fractal de mayor impacto visual que se conoce. Es un fractal muy sencillo, sin embargo, da resultados muy impresionantes.

El objetivo del ejercicio es mostrar la convergencia o no de una serie para cada uno de los puntos que se encuentran en un plano real-imaginario. Por ejemplo, sea el plano $(-2, -2i) \rightarrow (2, 2i)$ que se puede ver aquí:



El área sombreada es el plano que resulta entre los dos puntos escogidos. La idea es saber para cada uno de los puntos de este plano, si hacen converger o no la serie de Mandelbrot. Vamos a realizar la operación para un punto $Z_0 = (x_0, y_0 i)$ que está en el plano. La serie comienza con Z_0 y debemos encontrar los Z sucesivos (1, 2, 3, ...) con la siguiente función recurrente:

$$Z_n = (x_n, y_n) = Z_{n-1}^2 + Z_0$$

$$x_n = x_{n-1}^2 - y_{n-1}^2 + x_0$$

$$y_n = 2x_{n-1}y_{n-1} + y_0$$

En cada iteración, o sea, cada vez que encontramos un nuevo termino en la serie, debemos testear para ver si la serie converge o no. Se ha establecido esta fórmula de la manera siguiente:

Si $|Z_n| > 2 \rightarrow \text{serie divergente}$

O lo que es lo mismo

Si $x_n^2 + y_n^2 > 4 \rightarrow \text{serie divergente}$

Ahora, sabemos que existen infinitos puntos en el plano, lo único que hacemos es decidir cuántos puntos vamos a tener. Esto determina el tamaño de la imagen que vamos a crear. Por ejemplo, podemos decir que queremos 400x400 puntos en nuestra imagen.

Finalmente, solamente nos queda conocer el color que vamos a colocar en el punto que calculemos. Tenemos normalmente 256 colores posibles en un tono (por ejemplo en gris). Así que hacemos la regla siguiente:

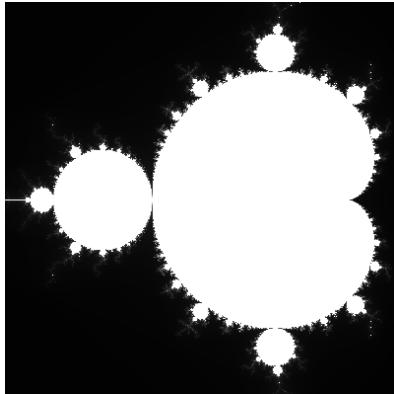
Si la serie sigue convergiendo luego de 255 iteraciones, consideramos que va a ser convergente siempre.

Luego, el color que le damos al punto es el número n de elementos de la serie que tuvimos que calcular hasta darnos cuenta si la serie convergía o no.

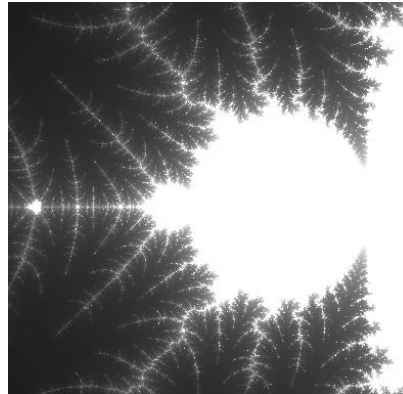
GALERIA DE IMÁGENES DE MANDELBROT

Aquí algunas imágenes obtenidas con un pequeño graficador de Mandelbrot con zoom.

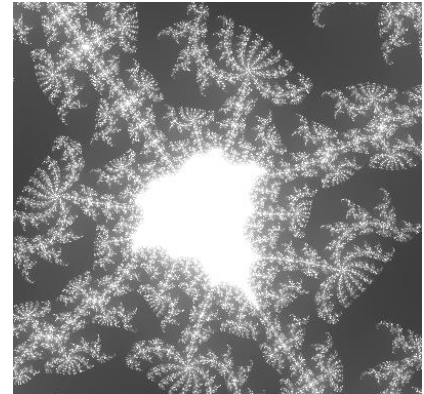
Clasico Mandelbrot entre $(-1.5, -1.5i) - (1, 1.5i)$



Similitud en zoom $(-1.4, 0.0065) - (-1.39, -0.0075)$



Figuras $(0.349, 0.067) - (0.35, 0.066)$



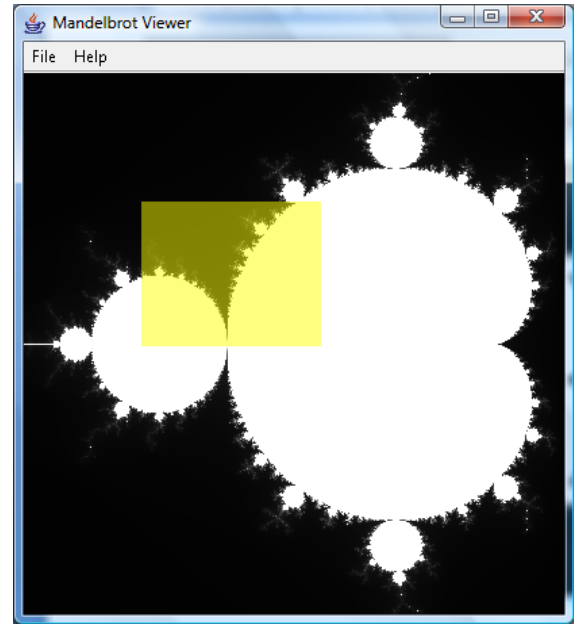
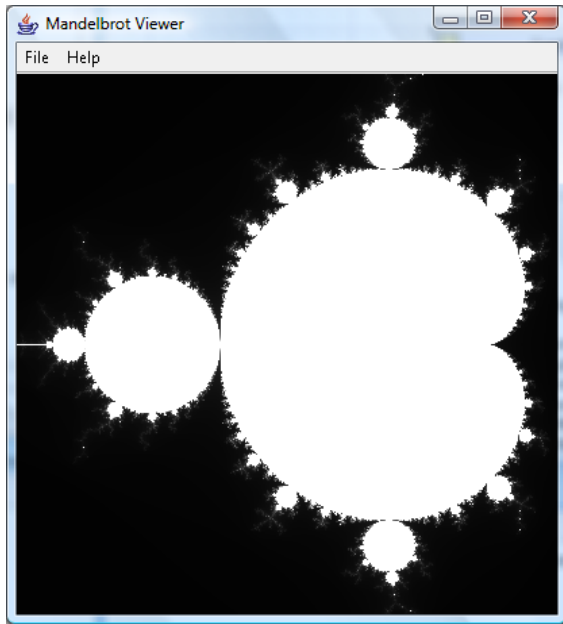
IMPLEMENTACION EN JAVA

El objetivo del ejercicio es poder lograr una aplicación que nos permita explorar el conjunto de Mandelbrot y que nos permita realizar zoom sobre un área que elijamos con el mouse.

La interface gráfica del programa debe permitir al usuario elegir un área con el mouse que define el zoom que se debe realizar. En el canvas de la aplicación se debe actualizar la imagen de Mandelbrot aplicando las nuevas coordenadas.

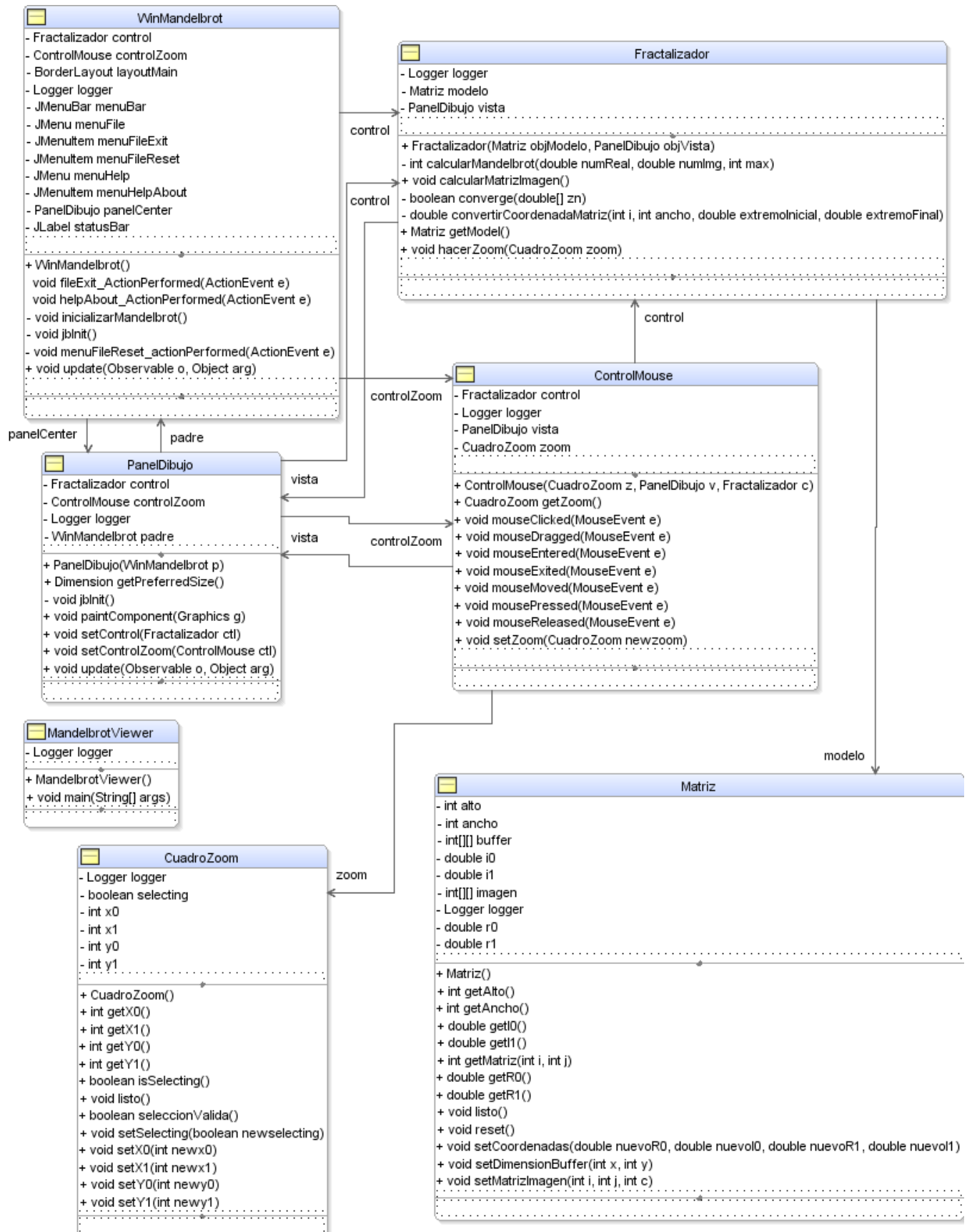
Menu superior con el canvas de Mandelbrot desplegado

Posibilidad de zoom con el mouse



Para ejercicio del lector, dejamos una propuesta del diagrama de clases que se podría utilizar para este efecto. Cabe recalcar que en este caso se ha tratado de seguir el patrón de diseño MVC visto anteriormente.

- **Modelo:** Las clases Matriz y CuadroZoom que tienen la información del modelo de datos.
- **Vista:** Las clases PanelDibujo y WinMandelbrot quienes tienen relación directa con los controladores y son observadores del modelo.
- **Controlador:** Las clases Fractalizador y ControlMouse que se encargan de ejecutar las acciones mandadas por los eventos que surgieron en la interfase.



Como siempre, queremos colaborar con el programador, por ello se puede observar en el siguiente código el manejo de la creación de la imagen. Este código se encuentra en la clase Fractalizador:

```
/**
 * Todo el calculo de la matriz de imagen mandelbrot que se visualizara.
 * Cuando se termina de calcular se marca el modelo como listo para que el
 * observador del modelo pueda dibujarlo.
 */
public void calcularMatrizImagen() {
    double diferenciaR = modelo.getR1() - modelo.getR0();
    int ancho = modelo.getAncho();
    int alto =
        (int) Math.abs((modelo.getI1() - modelo.getI0()) *
            (double) ancho /
            diferenciaR);
    modelo.setDimensionBuffer(modelo.getAncho(), alto);

    for(int i=0; i<ancho; i++) {
        for(int j=0; j<alto; j++) {
            double numReal =
                convertirCoordenadaMatriz(i, ancho,
                    modelo.getR0(), modelo.getR1());
            double numImg =
                convertirCoordenadaMatriz(j, alto,
                    modelo.getI0(), modelo.getI1());
            int c = calcularMandelbrot(numReal, numImg, 255);

            modelo.setMatrizImagen(i, j, c);
        }
    }

    logger.info("Ha creado la matriz y llama update de los observers");
    modelo.listo();
}
```

En este caso hemos querido añadir inclusive el comentario y las llamadas a los registros de logs. Esto para que usted vea como ejemplo el tipo de código fuente que debería estar produciendo.

Asimismo, mostramos el código donde se hace uso de la formula para calcular el valor de un punto:

```
/**
 * Calcula una iteracion de mandelbrot.
 * @param numReal
 * @param numImg
 * @param max El maximo de iteraciones que se van a realizar
 * @return
 */
private int calcularMandelbrot(double numReal, double numImg, int max) {
    int numIter = 0;

    double[] z0 = new double[2];
    z0[0] = numReal;
    z0[1] = numImg;
    double[] zn = new double[2];
```

```
zn[0] = numReal;
zn[1] = numImg;

double[] zn1 = new double[2];
while(numIter < max && converge(zn)) {
    zn1[0] = zn[0]*zn[0] - zn[1]*zn[1] + z0[0];
    zn1[1] = 2*zn[0]*zn[1] + z0[1];

    zn[0] = zn1[0];
    zn[1] = zn1[1];
    numIter++;
}

return numIter;
}
```

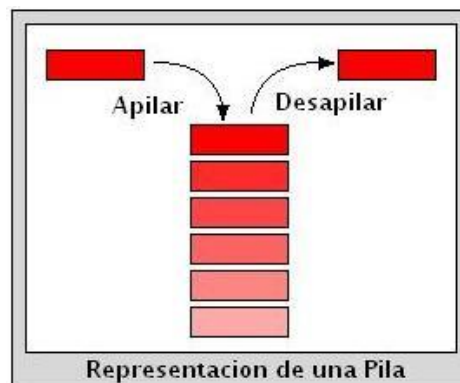
Dejamos como ejercicio al lector la implementación de este programa.

ESTRUCTURAS DE DATOS CON COMPORTAMIENTO

Existen algunas estructuras de datos que pueden ser tanto estáticas como dinámicas, lineales o no lineales; sin embargo, incorporan otros elementos que tienen que ver con el comportamiento de la estructura. En este apartado veremos dos tipos de esas estructuras: pila y cola.

PILA

Una pila es un contenedor de objetos que solo permite sacar el último objeto que ha sido puesto. Este tipo de estructuras son llamadas estructuras LIFO (Last-In First-Out), y por supuesto en alguna literatura también se utiliza la palabra FILO. Esta imagen tomada de Wikipedia nos muestra claramente lo que se puede hacer con esta estructura y como se obtienen y coleccionan objetos en la misma.



Las operaciones dentro de una pila tienen nombres que vienen desde hace varias décadas:

- **Push:** empilar un objeto dentro de la pila. Es decir, colocarlo como primer objeto a recuperar
- **Pop:** desapilar un objeto desde la pila. Es decir, quitarlo de la pila.

IMPLEMENTACIÓN DE PILA

Para implementar una pila se puede utilizar un arreglo. Sin embargo, debemos saber que nuestra pila nunca podrá superar la cantidad de elementos que definamos en el arreglo. Veamos cómo podemos implementar una pila de palabras:

```
public class PilaEstatica {
    private static final int MAX100 = 100;
    private String[] elementos;
    private int tope = 0;

    public PilaEstatica() {
        Elementos = new String[MAX100];
    }

    public void push(String obj) {
        elementos[tope] = obj;
        tope++;
    }

    public String pop() {
        String obj = elementos[tope];
        tope--;
        return obj;
    }
}
```

En Java podríamos utilizar la clase Stack que es una implementación dinámica de una pila. Es decir, esta pila puede crecer al infinito.

USO DE PILAS

Las pilas son estructuras de datos muy utilizadas en algoritmos de búsqueda, en grafos y en análisis de compilación de código y ejecución de programas en el sistema operativo.

COLA

Una cola es un contenedor de objetos que solo permite sacar el objeto que mas antes haya entrado al contenedor. Este tipo de estructuras son llamadas estructuras FIFO (First-In First-Out). Como se puede apreciar, esta estructura es muy similar a la anterior y se implementa de manera similar.

Las operaciones dentro de una cola tienen nombres que vienen desde hace varias décadas:

- **Push:** encolar un objeto dentro de la cola. Es decir, colocarlo como ultimo objeto a recuperar
- **Pop:** desencolar un objeto desde la cola. Es decir, quitarlo de la cola.

En Java se puede utilizar el objeto Queue de la librería java.util para tener una representación de una cola.⁹

USO DE COLAS

⁹ Ver la documentación de la implementación ya que no utiliza forzosamente la notación estándar de las operaciones.

Las colas son estructuras de datos muy utilizadas en algoritmos de búsqueda, en grafos, encolar las tareas de ciertos dispositivos como impresoras, dispositivos de red, etc.

GENERICS

Considerando la estructura de datos pila, que pasaría si en algún momento necesitaríamos una pila de enteros y no de palabras? Es probable que tuviésemos que crear las clases `PilaEnteros`, `PilaPalabras`, `PilaX`, etc.

Para evitar esto, podríamos crear el objeto `PilaObjeto`. En Java todos heredan de objeto, por lo tanto nos serviría para cualquier tipo de objeto. Aunque no nos serviría para los tipos primitivos, para estos habría que utilizar las clases wrapper¹⁰ de tipos primitivos. De todas maneras, tendríamos algunos inconvenientes.

Sea el siguiente problema:

```
Object o1 = new Algo();
Object o2 = new OtraCosa();
PilaObjeto s = new PilaObjeto();
s.push(o1);
s.push(o2);
OtraCosa a = (OtraCosa)s.pop();
Algo a = (Algo)s.pop();
```

Los problemas más importantes son:

- Se pierde la noción que una colección mantenga un solo tipo de objetos para todos los elementos.
- Se debe educar muchísimo al programador para que no se equivoque en el orden en que coloca y saca las cosas ya que puede que al hacer casting haya un error de tipo de objeto.

Para solucionar esto Java (y casi todos los lenguajes orientado objetos) incorpora el concepto de Generics. Estas son clases definidas con un parámetro que se reemplaza por una clase al momento de la ejecución.

Por ejemplo, si queremos tener una pila de Anillo, una de String y una de Persona. NO se hacen tres clases, sino, solamente se utiliza la clase `Stack`:

```
Stack<Anillo> pilaDeAnillo = new Stack<Anillo>();
Stack<String> pilaDeString = new Stack<String>();
Stack<Persona> pilaDePersona = new Stack<Persona>();
```

Este tipo de clases evita el tener que hacer casting y nos cubre de los posibles errores.

Como ejemplo del uso de Generics, aquí se puede ver un ejemplo de la implementación de la misma pila definida anteriormente, pero para cualquier tipo de objetos.

```
public class PilaEstatica<E> {
    private static final int MAX100 = 100;
    private E[] elementos;
    private int tope = 0;
```

¹⁰ Las clases wrapper son las clases `Integer` para `int`, `Char` para `char`, `Double` para `double`, y así sucesivamente. Encapsulan un tipo primitivo bajo un objeto Java.

```
public PilaEstatica() {
    Elementos = new E[MAX100];
}

public void push(E obj) {
    elementos[tope] = obj;
    tope++;
}

public E pop() {
    E obj = elementos[tope];
    tope--;
    return obj;
}
}
```

ESTRUCTURAS DE DATOS DINÁMICAS

Antes hemos tocado las estructuras de datos estrictamente estáticas. Estas son muy limitadas ya que no permiten que el programa se adapte al crecimiento dinámico que tienen los problemas del día a día. Hay diferentes cantidades de alumnos en una clase, cantidad de empleados en una empresa, en un departamento, etc.

Gracias a que podemos referenciar un objeto a través de una variable, o lo que es lo mismo, una posición en memoria, veremos diferentes enfoques para formar estructuras de datos que no solamente son dinámicas sino que tienen diferentes topologías.

ESTRUCTURAS DE DATOS DINÁMICAS LINEALES

Son las estructuras de datos que solamente se extienden en una dimensión, tienen el concepto de contigüidad entre sus elementos (uno al lado de otro), a manera de una lista. El hecho de que sean dinámicas significa que podrán crecer en el tiempo de manera indefinida.

Ventajas de las estructuras dinámicas lineales sobre las estáticas lineales:

- Tamaño estático de la estructura
- Inserción de un nuevo elemento es una operación complicada
- Eliminación de un elemento es una operación complicada

ESTRUCTURAS DE DATOS DINÁMICAS NO LINEALES

Son las estructuras de datos que se pueden extender en dos o más dimensiones. Los elementos de la misma están generalmente conectados pero las conexiones pueden ser múltiples y pueden o no respetar una jerarquía. Estas estructuras pueden llegar a ser muy complejas, por ejemplo, considere que uno quisiera representar un árbol genealógico de una familia muy grande considerando divorcios, posibles incestos, matrimonios múltiples, adopciones, etc.

Este tipo de estructuras serán estudiadas al final de este capítulo.

EDD DINAMICA LINEAL: CADENAS

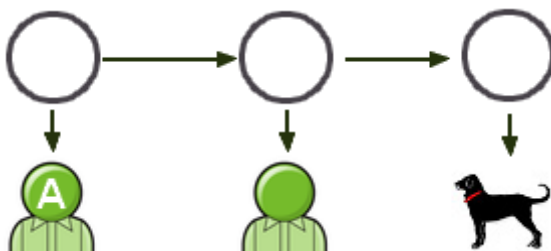
Es una estructura de datos dinámica lineal en la que cada elemento tiene una conexión hacia el siguiente elemento. El último de los elementos tiene esta variable igual a null.

Imaginemos que queremos tener una cadena con un objeto de tipo Alumno, otro de tipo Persona y otro de tipo Perro. Tenemos 3 objetos que tienen que acomodarse uno detrás de otro.



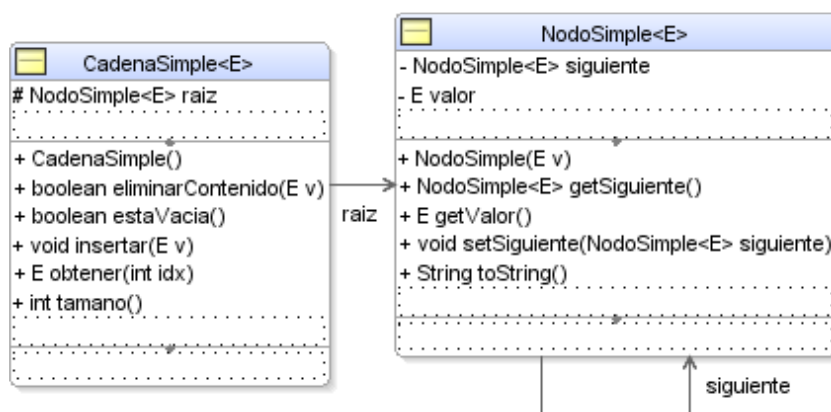
Sin embargo, tenemos que pensar en separar la lógica de la cadena de la lógica que representa a cada uno de los objetos. Es decir, una cosa es la cadena y otra son el alumno, persona y el perro. Por ello vamos a utilizar una noción de elemento dentro de una cadena: nodo.

Un nodo va a tener una referencia a un objeto del tipo que uno desee y una referencia al siguiente nodo de la cadena. Nuestra representación cambia y se transforma en lo siguiente:



Observe que los objetos Nodo van formando una cadena. En cada uno de los nodos podemos colocar el objeto que necesitamos.

El diseño de una estructura orientada a objetos, una vez que se tiene la forma en que interactúan estos objetos no es muy complicado y se puede ver como se llega a obtener esta misma estructura con este diagrama de clases:



Solamente con esta representación sencilla tenemos varias ventajas ya sobre la estructura estática:

- Es una estructura que crece a necesidad
- Insertar es sencillo
- Eliminar es muy sencillo

- Unir dos listas es sencillo
- Es genérico y se puede utilizar para cualquier tipo de objetos

IMPLEMENTACIÓN DE CADENA EN JAVA

El momento en que debemos comenzar a construir estructuras de este tipo, nos damos cuenta que las diferentes estructuras van a necesitar `Nodo(s)` y que la estructura de la clase `Nodo` puede variar mucho.

Por ello, para la implementación de una Cadena en Java, se utilizan clases internas que nos permiten escribir el código de `Nodo` dentro de la clase `CadenaSimple`. Así, el código para la implementación de una cadena, utilizando Generics y clases internas sería:

```
public class CadenaSimple<E> {  
  
    protected NodoSimple<E> raiz;  
  
    public CadenaSimple() {  
        raiz = null;  
    }  
  
    public boolean eliminarContenido(E v) {  
        if (raiz == null) {  
            return false;  
        }  
  
        NodoSimple<E> actual = raiz;  
        // Mientras no llegue al final y no sea igual a v  
        while (actual != null && actual.getValor() != v) {  
            actual = actual.getSiguiente();  
        }  
  
        // Llego al final y no encuentro al contenido  
        if (actual == null)  
            return false;  
  
        return true;  
    }  
  
    public void insertar(E v) {  
        NodoSimple<E> n = new NodoSimple<E>(v);  
        if (raiz == null) {  
            raiz = n;  
            return;  
        }  
        n.setSiguiente(raiz);  
        raiz = n;  
    }  
  
    public E obtener(int idx) throws IndexOutOfBoundsException {  
        if (idx < 0)  
            throw new IndexOutOfBoundsException("Indice no puede ser negativo: " +  
                idx);  
        if (idx >= this.tamano())
```

```
        throw new IndexOutOfBoundsException("Tamano CadenaSimple: " +
            this.tamano() + " <= " + idx);

        int idxActual = 0;
        NodoSimple<E> actual = this.raiz;
        while(idxActual < idx) {
            actual = actual.getSiguiente();
            idxActual++;
        }

        return actual.getValor();
    }

    public int tamano() {
        if (raiz == null)
            return 0;
        int tam = 0;
        NodoSimple<E> actual = raiz;
        while (actual != null) {
            actual = actual.getSiguiente();
            tam++;
        }
        return tam;
    }

    public boolean estaVacia() {
        return raiz == null;
    }

    static class NodoSimple<E> {

        private E valor;

        private NodoSimple<E> siguiente;

        public NodoSimple(E v) {
            siguiente = null;
            valor = v;
        }

        public NodoSimple<E> getSiguiente() {
            return siguiente;
        }

        public void setSiguiente(NodoSimple<E> siguiente) {
            this.siguiente = siguiente;
        }

        public E getValor() {
            return valor;
        }

        public String toString() {
            return this.hashCode() + "-> " + valor.toString();
        }
    }
}
```

```

    }
}

```

La observación más importante de esta implementación es el uso de clases internas. Las clases internas nos permiten elaborar una lógica de negocio dentro de una clase de manera que sea específica para esa clase. En nuestro caso y durante todo el resto del capítulo, utilizaremos la clase `Nodo` para indicar un elemento de una colección de objetos.

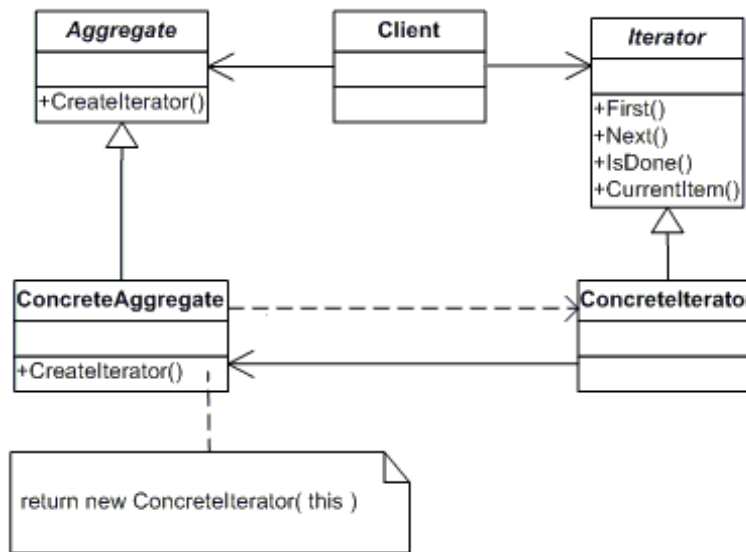
Estudie cada una de las implementaciones de los métodos ya que se puede ver el uso y manejo de la programación orientada objetos en todo su esplendor.

PATRÓN DE DISEÑO ITERATOR

Otro patrón de diseño importante que tiene que ver con estructuras de colecciones de objetos es el patrón de diseño `Iterator`. El problema es como iterar sobre cada uno de los elementos de cualquier colección de manera eficiente y estándar.

Para lograr esto, se debe separar la tarea de iterar en clases diferentes, la colección y el tipo de objetos dentro de la colección no importan. La implementación permite utilizar el patrón para cualquier tipo de listas.

El diagrama de clases tipo utilizado para la implementación de este patrón de diseño es el siguiente:



El `Aggregate` en este caso es una colección de objetos, el `ConcreteAggregate` es cualquier colección de objetos, en este caso podría ser una Cadena o cualquier otra colección.

Específicamente, en el lenguaje de programación Java, ya existe la interface `Iterator` que define algunos métodos considerados estándar. Lo que debemos hacer es escribir nuestro `ConcreteIterator` propio a la colección que tenemos.

Y por supuesto, el `ConcreteIterator` es también una clase interna de la clase `CadenaSimple` para que la misma tenga la característica de poder navegar por sus objetos con el patrón de diseño `Iterator`. Aquí el código de esta clase interna:

```

static class IteradorCadena<E> implements Iterator<E> {

    private CadenaSimple.NodoSimple<E> actual;

```

```
public IteradorCadena(CadenaSimple.NodoSimple<E> r) {
    actual = r;
}

public boolean hasNext() {
    return actual != null;
}

public E next() {
    CadenaSimple.NodoSimple<E> anterior = actual;
    actual = actual.getSiguiente();

    return anterior.getValor();
}

public void remove() {
    // se deja como ejercicio
}
}
```

Para cumplir con el patrón de diseño nos falta realizar en nuestra colección de objetos un método que nos permita recuperar un `Iterator` de manera que pueda ser utilizado por el cliente. Aquí el código del método `iterator()` de `CadenaSimple`:

```
public Iterator<E> iterator() {
    CadenaSimple.IteradorCadena<E> iterador;
    iterador = new CadenaSimple.IteradorCadena<E>(this.raiz);
    return iterador;
}
```

Luego, la forma de utilizar el patrón de diseño es la siguiente:

```
CadenaSimple<String> palabras = new CadenaSimple<String>();
// Se insertan varios elementos
Iterator<String> i = palabras.iterator();
while(i.hasNext()) {
    String s = i.next();
    System.out.println("Una palabra: " + s);
}
```

Se puede ver el uso de Generics en este pequeño código para evitar el casting. Algo que es muy importante es que esta manera de iterar sobre los elementos de una colección es estándar, eso significa que sea la colección que sea, de cualquier tipo, normalmente deberíamos poder iterar de esta manera.

EDD DINÁMICA LINEAL: CADENA DOBLE

Si tenemos nuestra cadena, veremos que hay algunas operaciones son muy sencillas como insertar al inicio de la cadena. Sin embargo hay otras operaciones que no son tan sencillas como aumentar un elemento al final.

O en el case de que quisiéramos insertar un nodo antes del nodo que tenemos como referencia; con una cadena es simplemente imposible ya que es el nodo anterior el que debe hacer referencia a este nuevo nodo que queremos insertar.

En el caso de la cadena doble, la estructura incorpora en sus nodos una referencia al nodo anterior también. Y por supuesto, en la clase CadenaDoble, se aumenta el elemento cola al ya existente raíz. Este elemento apunta al ultimo elemento de la cadena.

EDD DINÁMICA LINEAL: ANILLO

Anillo es una estructura de datos que transforma a nuestra cadena en una cadena infinita, ya que el nodo siguiente del último es la raíz nuevamente. En forma de código si Nodo n es el último nodo de la cadena, entonces:

```
n.setSiguiente(raíz);
```

El uso de anillos es útil cuando necesitamos iterar indefinidamente sobre la estructura. En el caso del sistema operativo, el procesador solamente puede ejecutar un proceso a la vez. Lo que hace el sistema operativo es colocar a todos los procesos en un anillo y cada proceso es ejecutado un tiempo pequeño (<1ms) y luego se pasa la mano al otro proceso. De esta manera todos los procesos son ejecutados con la ilusión de que varios se ejecutan al mismo tiempo.

EDD DINÁMICA LINEAL: ANILLO DOBLE

Así como en la cadena, en el anillo también podemos utilizar una referencia al elemento anterior. En este caso el elemento anterior de la raíz nos dará el último elemento de la cadena.

EDD DINÁMICA LINEAL: TABLAS DE HASH

Existe un problema recurrente en programación que consiste en encontrar un dato en una colección de datos. EN el caso de las colecciones que tenemos estamos obligados a recorrer potencialmente toda la cadena en búsqueda de ese elemento.

Cuando debemos realizar este tipo de operación muchas veces y tenemos una cadena con un importante número de nodos, entonces estamos ante un problema serio ya que nuestra aplicación comienza a ser muy lenta. La solución que veremos es el uso de tablas de hash.

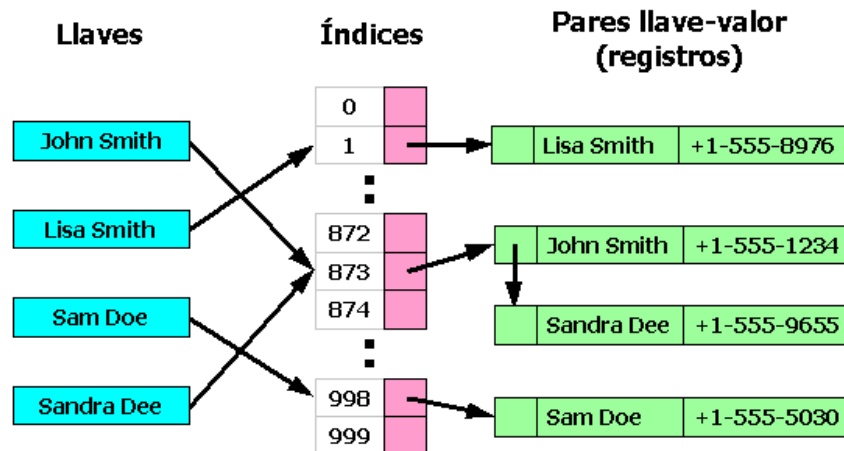
Las tablas de hash son estructuras de datos que permiten encontrar UN elemento sobre un arreglo en UN solo salto (o en muy pocos saltos). Esto lo realizamos haciendo algunos pasos previos:

1. Crear/Tener una función que permita encontrar un valor (hash) a partir de una llave (un String, un objeto, etc.)
2. La función que se designe no debe aceptar colisiones. Es decir, si dos llaves son diferentes el hash DEBE ser diferente.

Una vez tenemos esto, luego creamos un arreglo donde colocaremos cada elemento en el lugar que corresponda. Por ejemplo:

John Smith	$f('John\ Smith') = 873$	Arreglo[873] = John Smith
Lisa Smith	$f('Lisa\ Smith') = 1$	Arreglo[1] = Lisa Smith
Sam Doe	$f('SamDoe') = 998$	Arreglo[998] = Sam Doe

Es imposible realizar una función que NO tenga colisiones para todos los casos, por ello, el manejo de colisiones en una tabla de hash hace que en el lugar de la colisión se utilice una cadena para albergar todos los objetos que colisionan. En el ejemplo vemos como 'Sandra Dee' colisiona con John Smith.



Y para buscar UN elemento, digamos el objeto con llave 'John Smith', la función nos da el valor 873. Así que si buscamos en la posición 873 del arreglo encontraremos TODA la información de John Smith.

EN JAVA

En el lenguaje de programación Java ya existe una implementación de Hashtable que se aconseja utilizar. Esta implementación se encuentra en el paquete `java.util`. La implementación es genérica tanto para las llaves como para los objetos que apuntan estas llaves.

Java se encarga de manejar internamente el tamaño del arreglo que contiene los elementos de la tabla de hash.

PATRÓN DE DISEÑO: COMPARATOR

Este patrón soluciona un problema general que es el de poder comparar dos objetos del mismo tipo siguiendo un criterio. Esto a su vez permite ordenar una lista de elementos a partir de ese criterio.

El criterio de comparación debe poder ser programado en los objetos a comparar. No se compara de la misma manera un String que una Persona, o un Empleado.

SOLUCION EN JAVA

Para solucionar este problema, debemos definir una interface que defina la comparación. En Java esta interfase es Comparable.

Los elementos deben ser instancias de una clase que implemente la interface `Comparable<E>`. La forma de la interface es la siguiente:

```
public interface Comparable<E>{
    public int compareTo(E e);
}
```

La regla es:

- Si la función devuelve 0, los objetos son iguales
- Si la función devuelve 1 (número positivo), entonces el objeto (this) es mayor que el objeto en parámetro
- Si la función devuelve -1 (número negativo) entonces el objeto en parámetro es mayor que el objeto this.

Cuando se tiene una estructura de datos con muchos elementos, el comparador permite ordenar la estructura aparte de las herramientas que nos da para comparar los elementos.

Ejemplo de ordenamiento:

```
Vector<Alumno> v = new Vector<Alumno>();  
  
v.add(new Alumno("hugo"));  
v.add(new Alumno("paco"));  
v.add(new Alumno("luis"));  
  
Collections.sort(v);
```

Este pequeño código nos muestra cómo se realiza el ordenamiento de un arreglo o una lista dinámica. La forma de comparar los objetos viene del hecho que Alumno implementa la interface Comparable.

Para ordenar la colección se ha utilizado la clase Collections que tiene un método que aplica Quicksort de acuerdo al ordenamiento dictado en el método implementado en Alumno para la interface Comparable.

EDD DINÁMICA LINEAL: CADENA ORDENADA

Una cadena ordenada es una cadena que además tiene la propiedad de que sus elementos siempre están en orden. Esto es muy útil en algunos casos ya que evita tener que ordenar la colección cada vez.

Para lograr esto lo que tenemos que hacer es modificar nuestro método insertar para que:

- Primero se haga el test si el objeto a ingresar es un Comparable
- Si lo es, entonces comenzar a buscarle el lugar correcto en la cadena.
- Una vez encontrado su lugar, se lo coloca y se mantiene en todo momento la cadena ordenada.

Se deja como ejercicio el realizar la cadena ordenada, y aquí se mostrara un ejemplo de su uso en un caso simple.

Lo primero es contar con la clase que implementa la interface Comparable para tener los objetos que comparar. Utilizaremos la clase Empleado que tiene nombre y salario. La comparación se hará por salario:

```
public class Empleado implements Comparable<Empleado> {  
    private String nombre;  
    private double salario;  
  
    public double getSalario() {  
        return salario;  
    }  
  
    public int compareTo(Empleado e) {  
        if (this.salario == e.getSalario())  
            return 0;  
        return (this.salario > e.getSalario() ? -1 : 1);  
    }  
}
```

```
}  
}
```

Luego, para utilizar la Cadena ordenada es muy sencillo:

```
Cadena<Empleado> c = new Cadena<Empleado>();  
c.insertar(new Empleado("juan", 2400.0);  
c.insertar(new Empleado("hugo", 6000.0);  
c.insertar(new Empleado("paco", 1800.0);  
c.insertar(new Empleado("luis", 4500.0);
```

Si la cadena ha sido implementada correctamente, al recorrer la misma deberíamos tener la serie: hugo, luis, juan, paco.

EDD DINÁMICA NO LINEAL: ÁRBOL

Un árbol es una estructura de datos ampliamente usada que imita la forma de un árbol (un conjunto de nodos conectados). Un nodo es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él.

TERMINOLOGÍA

Se dice que un nodo **A** es padre de un nodo **B** si existe un enlace desde A hasta B (en ese caso, también decimos que B es hijo de A).

Sólo puede haber un único nodo sin padres, que llamaremos raíz. Un nodo que no tiene hijos se conoce como hoja. Los demás nodos (tienen padre y uno o varios hijos) se les conoce como rama

- Nodo de un árbol: hoja, raíz
- Longitud de un camino
- Antecesor, descendiente
- Grado de un árbol
- Altura: longitud del camino más largo de un nodo a una hoja
- Nivel o profundidad de un nodo: longitud de la raíz a ese nodo

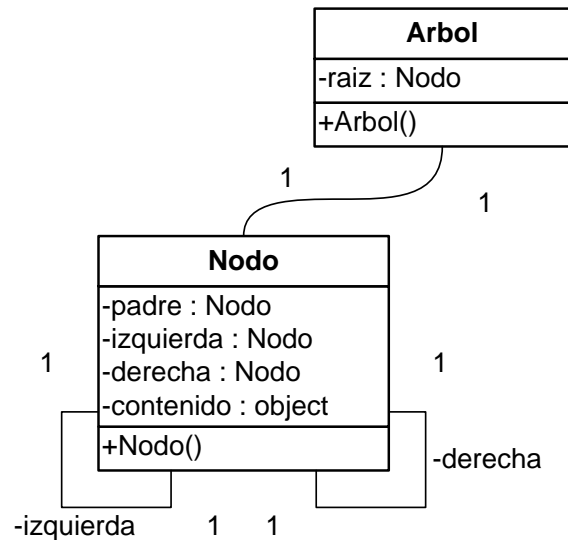
USO DE LA ESTRUCTURA ÁRBOL

Las estructuras de tipo árbol son ampliamente utilizadas en la programación. En especial para cualquier representación que necesite la noción de jerarquía. A continuación veremos algunos ejemplos:

1. **Sistemas de archivos:** Cada vez que abrimos nuestro explorador de archivos podemos ver a la izquierda la representación en árbol de los archivos en el sistema.
2. **Organización de recursos humanos:** En cualquier aplicación de recursos humanos en una empresa se tiene el concepto de que el empleado siempre tiene un jefe, salvo el gerente general o CEO.
3. **Navegación web:** El historial de navegación que tiene un usuario, a partir de una página dada se puede representar con un árbol.

IMPLEMENTACIÓN DE ARBOLES

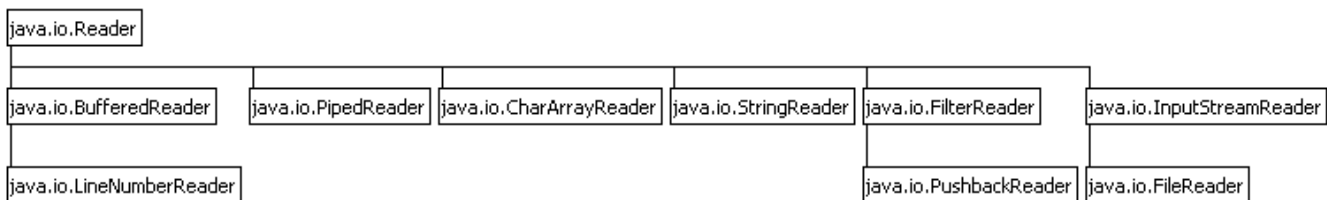
La implementación de árboles es muy similar a la de cadenas solamente se debe definir lo que puede y no puede tener un nodo. En la implementación siguiente se puede ver un árbol binario:



La implementación de un árbol depende de la cantidad de hijos que pueda tener cada nodo. Si por ejemplo se utilizan nodos que tienen muchos hijos, considerar el uso de arreglos. Si se utilizan nodos que tienen un número de hijos arbitrarios (caso raro), entonces se debería utilizar una cadena o lista de dinámica de nodos hijos.

JERARQUIA DE CLASES JAVA

Un ejercicio muy interesante de realizar es poder mostrar la jerarquía de las clases Java, mejor aun de manera grafica. Por ejemplo, la clase `java.util.Reader` tiene los siguientes 'hijos':



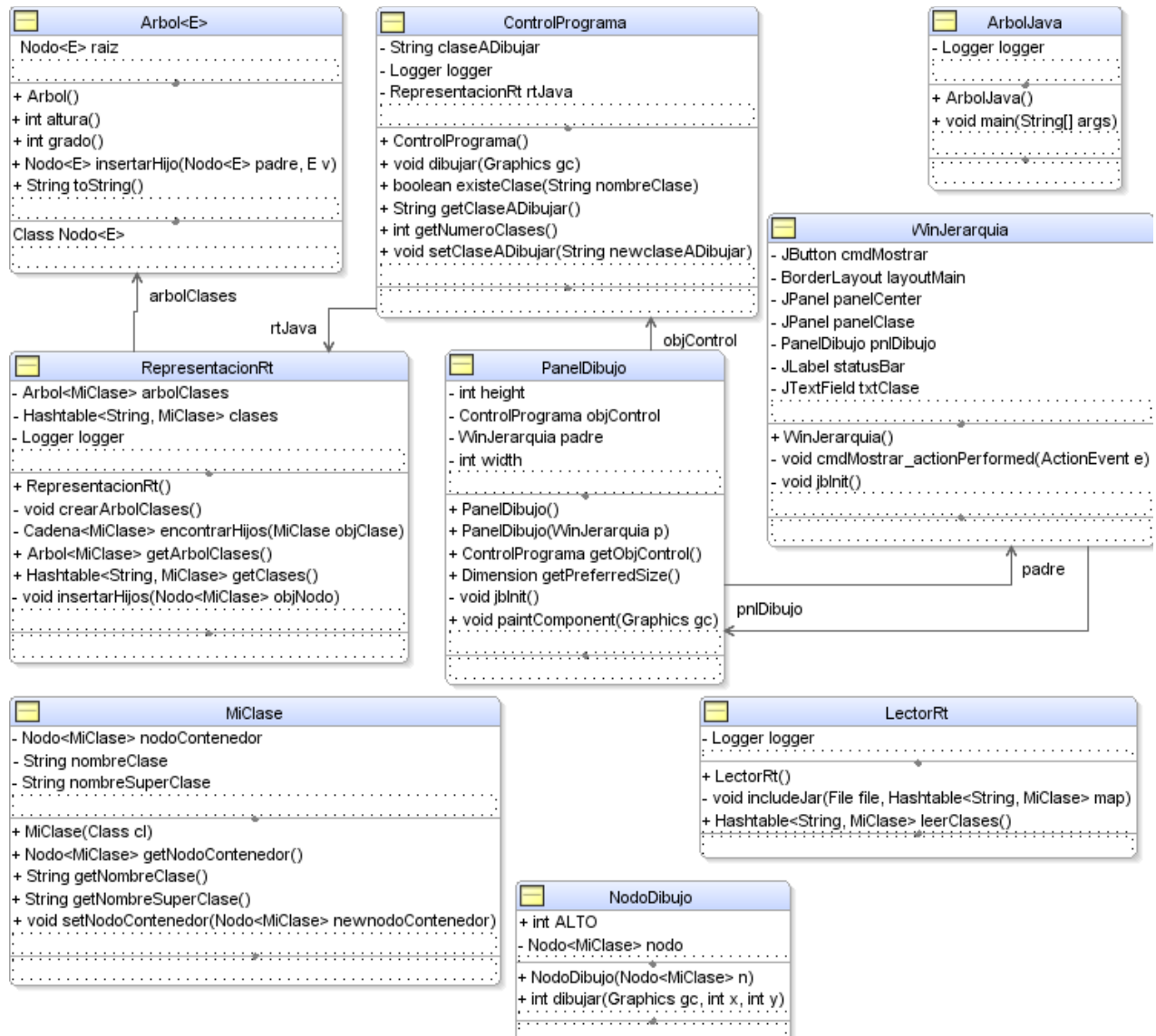
Para poder realizar este programa se ha tenido que trabajar con Reflection.

UN POCO DE REFLECTION

En la terminología de la programación orientada objetos se manejan todo el tiempo los conceptos de clase, objeto, método, atributo, interface. Para la maquina virtual Java, cada uno de esos conceptos es un tipo de objeto. Por ejemplo, la clase `java.lang.String`, el código compilado Java que representa a esta clase, no es nada más que un objeto de la clase `java.lang.Class`.

Así podemos tratar una clase como un objeto y utilizar los métodos que tenemos disponibles para obtener la superclase, las interfaces implementadas, los atributos, los métodos declarados, etc.

Para efectos de nuestro programa la implementación sugerida es la siguiente:



Este programa también trata de utilizar el patrón de diseño MVC. Pero además, tiene una complejidad adicional que es la de precargar todo el árbol de clases Java en una estructura de árbol. Aquí se puede ver el código que carga la lista de clases en la estructura:

```

public RepresentacionRt() {
    LectorRt objLector;
    objLector = new LectorRt();
    clases = objLector.leerClases();
    crearArbolClases();
}

private void crearArbolClases() {
    MiClase inicio = clases.get("java.lang.Object");
    arbolClases = new Arbol<MiClase>();
    Arbol.Nodo<MiClase> objNodo = arbolClases.insertarHijo(null, inicio);
    inicio.setNodoContenedor(objNodo);
    insertarHijos(objNodo);
}

```

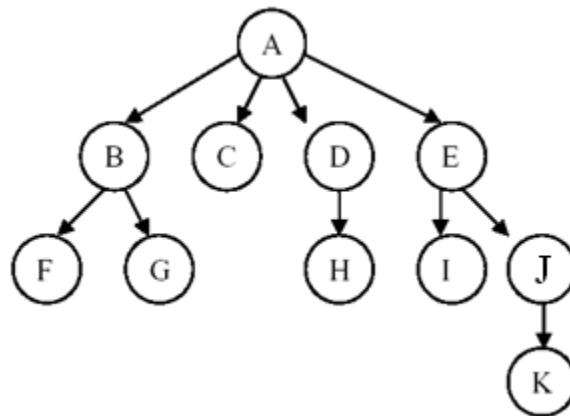
}

Por supuesto, las clases se deben cargar del archivo JAR que contiene las clases de Java que es el archivo `rt.jar`.¹¹

La interface pedirá el nombre de una clase en Java (se debe dar el nombre completo incluyendo el paquete) y luego pasará a dibujar la estructura de árbol de todas las subclases a partir de esa clase.

ALGORITMOS DE RECORRIDO DE ÁRBOL

Recorrer un árbol significa visitar cada uno de los nodos del árbol de una manera ordenada y sistemática. Dependiendo de lo que queremos conseguir, se puede visitar un árbol de diferentes maneras. Tomemos como ejemplo el siguiente árbol:



Existen dos recorridos típicos para listar los nodos de un árbol:

1. **Recorrido en profundidad**, en el ejemplo, el orden de visita de los nodos sería: ABFGCDHEIJK

La secuencia nos muestra que el recorrido avanza en profundidad desde la raíz. Generalmente se utiliza para investigar todos los caminos posibles de un nodo a otro. “Todos los caminos que conducen a Roma”

2. **Recorrido en anchura**, en el ejemplo, el orden de visita sería: ABCDEFGHIJK

La secuencia nos muestra un recorrido por niveles. Generalmente se utiliza para encontrar el camino más corto entre dos nodos. “El camino más corto que lleva a Roma”

Estos dos recorridos con sus respectivos algoritmos serán vistos en detalle en la sección de Grafos. Otros recorridos típicos del árbol son variaciones del recorrido en profundidad.

PREORDEN

El recorrido en preorden, también llamado orden previo consiste en recorrer en primer lugar la raíz y luego cada uno de los hijos en orden previo. En el ejemplo el recorrido sería: ABFGCDHEIJK

¹¹ Para encontrar la ubicación del archivo `rt.jar`, visite la carpeta de su instalación de Java y busque en la carpeta `lib`. La lectura de un archivo JAR es similar a la de un ZIP.

INORDEN

El recorrido en inorden, también llamado orden simétrico (aunque este nombre sólo cobra significado en los árboles binarios) consiste en recorrer en primer lugar el hijo de la izquierda, luego la raíz y luego cada uno de los hijos de la derecha. En el ejemplo tendríamos:

FBGACHDIEKJ

POSTORDEN

El recorrido en postorden, también llamado orden posterior consiste en recorrer en primer lugar cada uno de los hijos y por último la raíz. En el ejemplo tendríamos:

FGBCHDIKJEA

EDD DINÁMICA NO LINEAL: ÁRBOL BINARIO

En ciencias de la computación, un árbol binario es una estructura de datos dinámica no lineal en la cual cada nodo solamente tiene las siguientes posibilidades:

- No tiene hijos (hoja).
- Tiene un hijo izquierdo y un hijo derecho.
- Tiene un hijo izquierdo.
- Tiene un hijo derecho

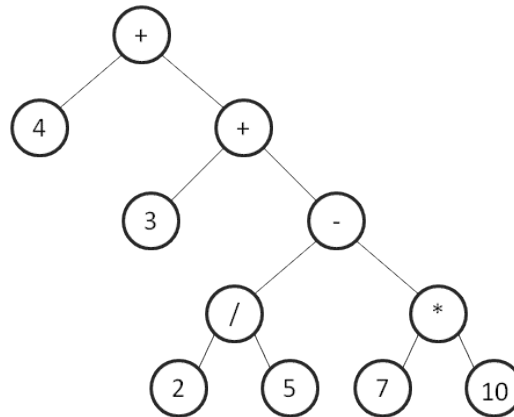
Los árboles binarios se usan mucho ya que son sencillos de representar y simplifican un problema. En el caso de una expresión aritmética por ejemplo:

$$4 + 3 + \frac{2}{5} - 7 * 10$$

Se puede utilizar un árbol binario para evaluar esta expresión, si reordenamos a:

$$4 + \left(3 + \left(\left(\frac{2}{5} \right) - (7 * 10) \right) \right)$$

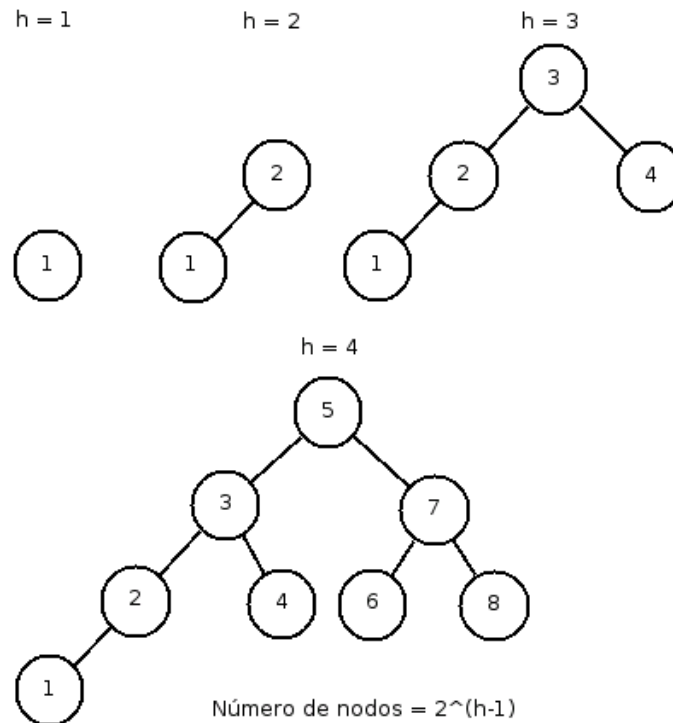
Esto nos permite colocar la expresión en un árbol de la siguiente manera:



Lo cual, en postorden nos permite evaluar la expresión aritmética.

ÁRBOL EQUILIBRADO

Es un árbol cuyo número de hojas en cada una de las ramas se encuentra en cantidades iguales o al menos similares.



En este caso se puede ver cómo un árbol de máximo dos nodos hijos (árbol binario) se va armando de manera que esté equilibrado. Lo importante es notar que NO importa el orden en que hayan llegado los nodos a la estructura. El más famoso es justamente el árbol AVL (Adelson-Velskii, Landis). Autobalanceable.

EDD DINÁMICA NO LINEAL: ÁRBOL B

En las ciencias de la computación, los árboles-B ó B-árboles son estructuras de datos de árbol que se encuentran comúnmente en las implementaciones de bases de datos y sistemas de archivos. Los árboles B mantienen los datos ordenados y las inserciones y eliminaciones se realizan en tiempo logarítmico amortizado.


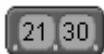

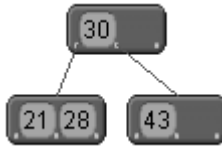
Los creadores del árbol B, Rudolf Bayer y Ed McCreight, no han explicado el significado de la letra B de su nombre. Se cree que la B es de balanceado, dado que todos los nodos hoja se mantienen al mismo nivel en el árbol. La B también puede referirse a Bayer, o a Boeing, porque sus creadores trabajaban en el Boeing Scientific Research Labs en ese entonces.

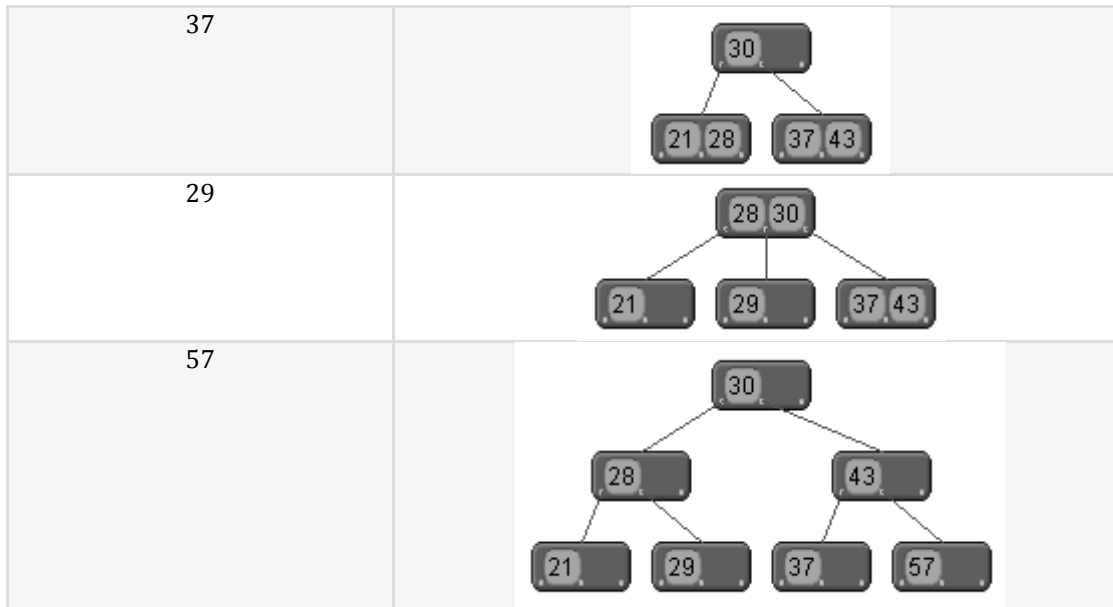
La diferencia principal con los arboles tradicionales es que en su estructura, en cada nodo, se colocan varios objetos. Un árbol-B de orden M (el máximo número de hijos que puede tener cada nodo) es un árbol que satisface las siguientes propiedades:

- Cada nodo tiene como máximo M hijos.
- Cada nodo (excepto raíz y hojas) tiene como mínimo $M/2$ hijos.
- La raíz tiene al menos 2 hijos si no es un nodo hoja.
- Todos los nodos hoja aparecen al mismo nivel, y no tienen hijos.
- Un nodo no hoja con k hijos contiene k-1 elementos almacenados.
- Los hijos que cuelgan de la raíz (r_1, \dots, r_m) tienen que cumplir ciertas condiciones :
 - El primero tiene valor menor que r_1 .
 - El segundo tiene valor mayor que r_1 y menor que r_2 etc.
 - El último hijo tiene mayor que r_m .

Para ver un buen ejemplo de este tipo de arboles se aconseja visitar el enlace <http://slady.net/java/bt/view.php>

Veamos un ejemplo sacado de la página donde cada nodo puede tener hasta dos elementos y un máximo de 3 hijos. Veremos el ejemplo en forma de pasos o etapas donde se verá el cambio del árbol en cada una de ellas.

Numero ingresado	Estado del árbol luego de la inserción
30	
21	
43	
28	



Observe como el árbol se mantiene siempre lo más equilibrado posible. Del ejemplo se puede fácilmente obtener la regla para la inserción:

- La estructura necesita que haya el concepto de ordenamiento entre los elementos
- En cada nodo los elementos se colocan en orden
- Si ya no hay campo para colocar un elemento en un nodo, entonces se obtiene la mediana entre los elementos del nodo y el nuevo elemento. Ese es el elemento que se coloca como raíz y se divide el nodo actual en dos nodos.

EDD DINÁMICA NO LINEAL: GRAFOS

Son aquellos que tienen un número arbitrario de conexiones en cada nodo. Esto hace que se pierda la noción de jerarquía que había en los árboles para tener una estructura más libre y general.


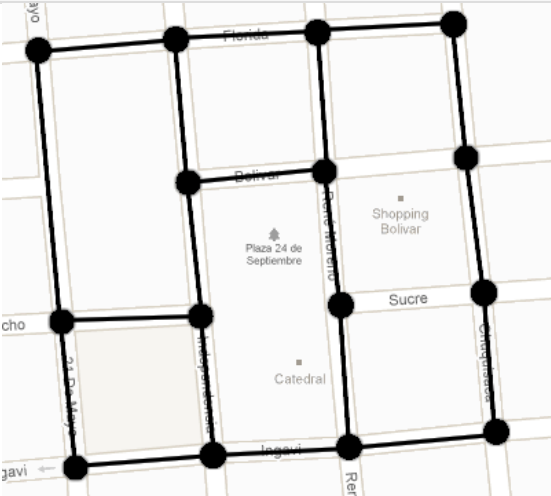
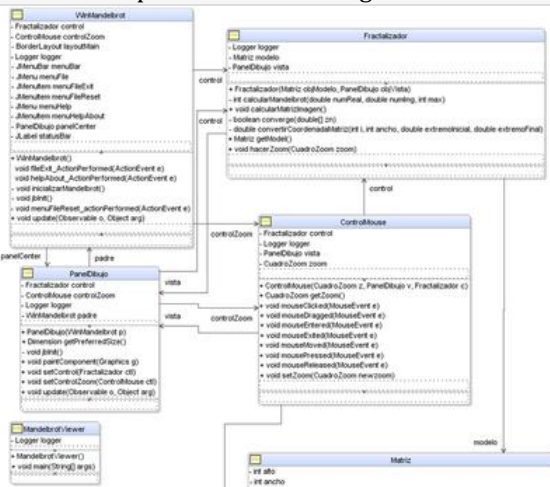
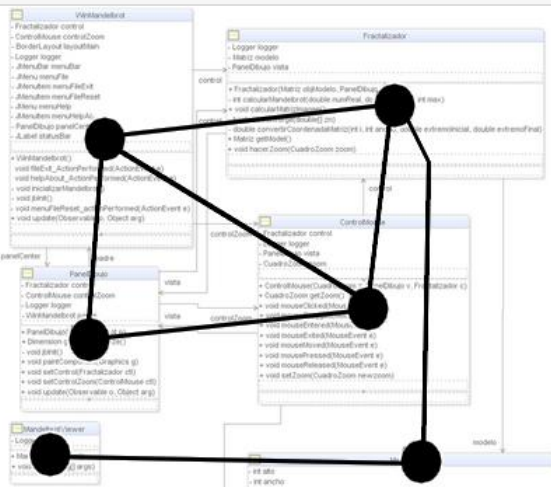
La siguiente es la terminología generalmente utilizada para grafos y la que se usará en el resto del capítulo:

- **Nodo:** Un nodo, al igual que en los árboles, representa a una unidad de la estructura de grafos que puede contener un objeto y que puede también estar conectado a otros nodos.
- **Arista:** La arista es el artificio que permite crear una relación entre dos nodos. Las aristas pueden también tener dirección y peso. Si no tienen dirección se sobreentiende que la relación es de ambos lados.
- **Camino:** Es una lista de nodos. Entre los nodos, deben existir relaciones (aristas) para que tengamos un camino conexo.
- **Grafo dirigido:** Cuando las aristas del grafo tienen dirección.

Los grafos se van complicando a medida de que se coloquen o no diferentes pesos en los arcos y si tienen o no dirección. De la misma manera, los nodos de un grafo no tienen por qué estar conectados forzosamente entre si y puede un grafo tener varias componentes conexas.

EJEMPLOS DE GRAFOS

Para poder apreciar el potencial de esta estructura veamos las posibles situaciones reales que esta estructura de datos puede representar:

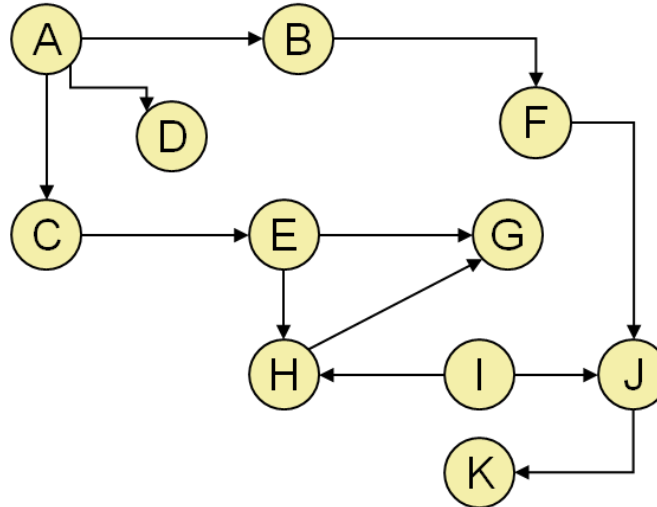
Situación Real	Representación Grafo
<p>Mapa de una ciudad, encontrar caminos</p> 	<p>Cada nodo es una esquina</p> 
<p>Representación de diagramas</p> 	<p>Cada clase es un nodo</p> 

Por la libertad que tienen, los grafos nos permiten representar una infinidad de situaciones y estructuras, sin embargo, el uso más efectivo de un grafo esta cuando debemos explorar los diferentes caminos en un grafo.

IMPLEMENTACIÓN DE GRAFOS

Existen al menos dos formas fundamentalmente diferentes para realizar la implementación de un grafo. En este capítulo veremos las dos principales y se dejara al lector la oportunidad de explorar otras maneras de representar un grafo.

Para demostrar ambas implementaciones se utilizara un mismo ejemplo que es el que se ve a continuación:



MATRIZ DE ADYACENCIA

Para representar la estructura anterior se puede utilizar una matriz. En la matriz. La matriz tendrá n columnas y n filas, donde n es el número de nodos que tiene el grafo. Para el caso de la figura hablamos de una matriz:

$$n = 11 \rightarrow m(n * n) \rightarrow m(11 * 11)$$

El nodo de cada fila nos indica el origen de una arista y el nodo de una columna nos indica el destino de la arista. De esta manera se pueden realizar todas las asociaciones entre nodos.

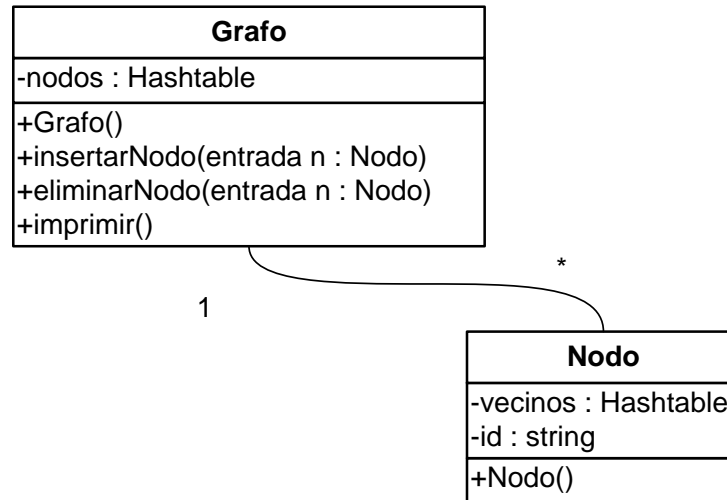
Inclusive, si la arista tiene un valor (peso), se lo puede colocar numéricamente en la matriz (el 0 significa que NO existe arista entre los nodos).

	A	B	C	D	E	F	G	H	I	J	K
A	0	1	1	1	0	0	0	0	0	0	0
B	0	0	0	0	0	1	0	0	0	0	0
C	0	0	0	0	1	0	0	1	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1	1	0	0	0
F	0	0	0	0	0	0	0	0	0	1	0
G	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	1	0
J	0	0	0	0	0	0	0	0	0	0	1
K	0	0	0	0	0	0	0	0	0	0	0

La principal ventaja de esta representación es que es muy sencilla de realizar. Sin embargo, la desventaja mayor es que se trabaja con estructuras estáticas. Se podría reemplazar la matriz por cadenas de cadenas, sin embargo, el manejo de índices en ese momento se torna un poco más complejo y de mayor cuidado.

GRAFO CON PROGRAMACIÓN ORIENTADA OBJETOS

La representación de grafos con elementos de la programación orientada a objetos pasa por representar las clases de Nodo, Grafo, etc. Esta representación permite simplificar o hacer más compleja la estructura que se quiera representar.



La implementación lleva como atributos objetos de tipo Hashtable (ver EDD Dinámica Lineal: Tablas de Hash). Aquí se ve la ventaja de estos objetos ya que se puede encontrar al objeto nodo directamente por su nombre (o id).

PROBLEMAS QUE SE SOLUCIONAN CON GRAFOS

Aquí algunos de los problemas típicos de grafos, generalmente, al enfrentar un problema de grafos, se puede reducir el problema a alguno de estos:

- **Camino más corto:** Para saber si existe un camino entre dos nodos y estar seguros que es el mas corto.
- **Conectividad simple:** Existen algoritmos para saber si se puede ir desde cualquier nodo de un grafo a cualquier otro.
- **Detección de ciclos:** Se puede saber también si el grafo presenta caminos que vuelven otra vez al principio cada vez
- **Camino de Euler:** Se pueden colocar condiciones interesantes en el paseo por un grafo. El camino de Euler es pasar por todos los nodos cruzando solamente una vez por cada arista
- **Camino de Hamilton:** El camino de Hamilton es una posible solución al problema del viajero. ¿Como se puede pasar por todos los nodos una sola vez?

RECORRIDO EN PROFUNDIDAD: DFS (DEPTH FIRST SEARCH)

Se pueden tener muchos objetivos para el hecho de recorrer un grafo. La forma de hacerlo es lo que importa para poder resolver uno u otro problema.

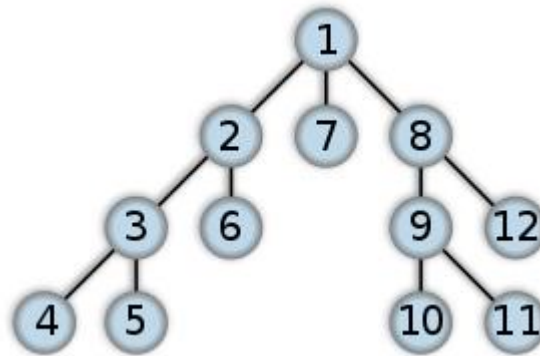
Vamos a suponer que todos los grafos con los que trabajemos son grafos conexos, es decir siempre se puede llegar desde el nodo que elijamos como inicio a otro nodo dentro del grafo.

En el caso de DFS, la forma del recorrido es la siguiente:

- A partir del nodo inicial (puede ser cualquiera del grafo) se escoge un nodo que será el próximo a visitar.

- Luego de escogido, se lo visita y se lo toma como nodo inicial.
- Se hace así hasta que no queden otros nodos posibles de visitar sin repetir

El recorrido se llama en profundidad ya que se avanza hasta donde no se puede más y luego se continúa con la rama que toque. En el grafo se puede ver el orden en que se visitaría con el algoritmo de DFS en el caso de este grafo:



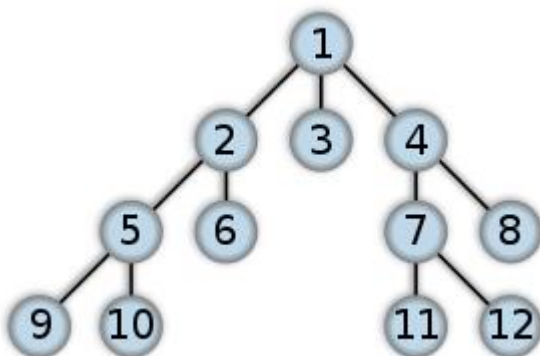
El funcionamiento del algoritmo es el siguiente:

1. Se crea una pila de visitas de nodos
2. Se coloca el nodo inicial N en la pila
3. Mientras la pila no esté vacía:
 - Se toma el nodo que esté en el tope de la pila (pop)
 - Se visita el nodo
 - Se recuperan TODOS los vecinos que no estén visitados y que NO se encuentren en la pila.
 - Se colocan todos los vecinos así encontrados en la pila
4. La componente conexas del grafo ha sido completamente visitada con DFS

RECORRIDO EN ANCHURA: BFS (BREADTH FIRST SEARCH)

En el caso de BFS la visita de los nodos tiene un cambio importante. La idea es atacar la visita de los nodos hacia todas las posibilidades al mismo tiempo. Esto permite encontrar soluciones un poco antes que en el caso de DFS.

El orden de la visita de los nodos de un grafo se puede ver en la siguiente figura:



El funcionamiento del algoritmo es el siguiente:

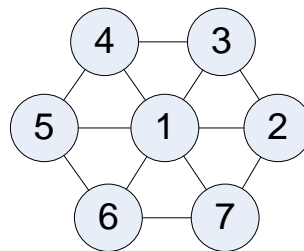
1. Se crea una cola de visitas de nodos
2. Se coloca el nodo inicial N en la cola
3. Mientras la cola no esté vacía:
 - a. Se toma el nodo que esté al final de la cola (pull)
 - b. Se visita el nodo
 - c. Se recuperan TODOS los vecinos que no estén visitados y que NO se encuentren en la cola.
 - d. Se colocan todos los vecinos así encontrados en la cola.
4. La componente conexas del grafo ha sido completamente visitada con BFS

Se puede ver que el recorrido de un grafo, ya sea con DFS o con BFS es muy similar. Uno de los problemas típicos que se puede resolver con BFS es el de poder encontrar el camino más corto entre dos nodos.

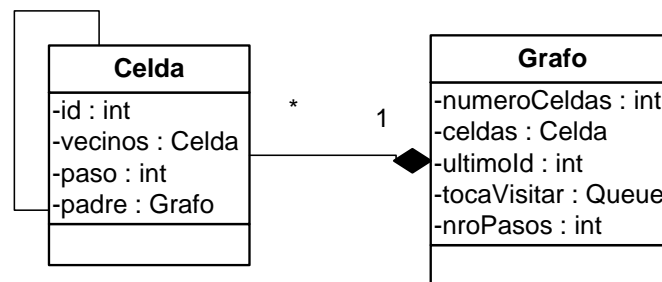
IMPLEMENTACIÓN BÚSQUEDA CON DFS/BFS

A continuación se verá en detalle la implementación del algoritmo de BFS. Como el algoritmo es sumamente similar al de DFS se deja su implementación como ejercicio.

Supongamos que tiene un grafo como el siguiente con cientos de nodos (aquí solamente mostramos una parte):



Entonces, el diagrama de clases que se tiene para la estructura es el siguiente:



Ahora, el problema es saber cuántos saltos tenemos que dar para ir del nodo 5233 al 3289. Se resuelve con el método de BFS con el siguiente código:

```
public int caminoMasCorto(int de, int a) {
    this.tocaVisitar = new LinkedList<Celda>();
    for(int i=0; i<this.numeroCeldas; i++) {
        celdas[i].setPaso(0);
    }

    boolean encontro = false;
```



```
celdas[de - 1].setPaso(1);
this.tocaVisitar.add(celdas[de-1]);
// sino, seguimos buscando
while(!this.tocaVisitar.isEmpty() && !encontro) {
    Celda c = this.tocaVisitar.poll();
    encontro = this.visitarCelda(c.getId(), a);
}
return celdas[a - 1].getPaso() - 1;
}
```

Se puede ver que primero se coloca a 0 el paso de todas las celdas. Luego se coloca en la lista de las que toca visitar la primera celda (de).

Comienza el algoritmo tal como está enunciado: mientras la cola no esté vacía, se visita la celda. Hay un paso que NO se encuentra en el algoritmo que justamente es para testear si es que no se ha encontrado ya el camino más corto. Cada vez que se visita una celda se devuelve verdadero o falso.

En el método visitar Celda se colocan los demás pasos que siguen en el algoritmo, como se puede ver aquí:

```
public boolean visitarCelda(int id, int a) {
    Celda c = this.celdas[id - 1];

    for(int i=0; i<6; i++) {

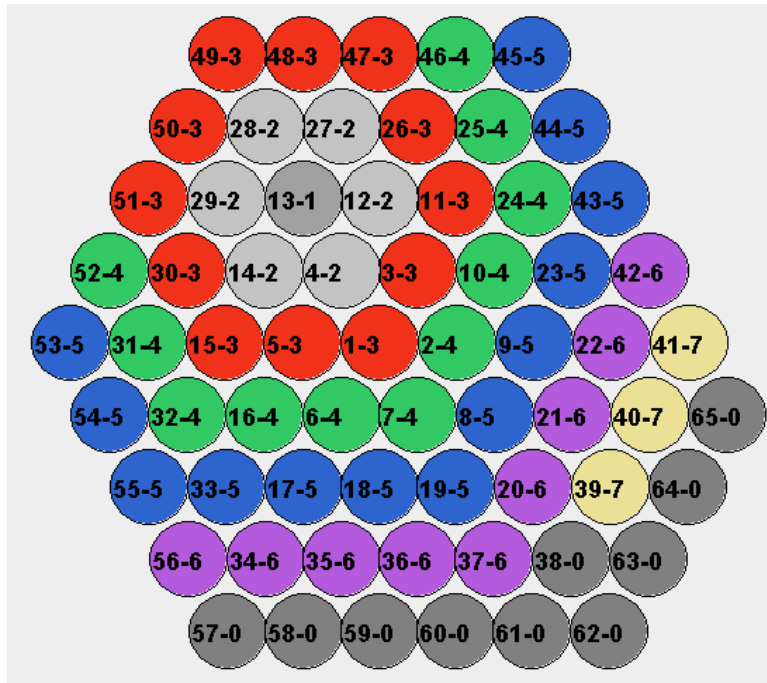
        // Si no es nulo y no ha sido visitado
        Celda vecinoDeCenI = c.getVecino(i);
        if (vecinoDeCenI != null) {
            // colocarlo en espera de la pila
            if (vecinoDeCenI.getId() == a) {
                vecinoDeCenI.setPaso(c.getPaso() + 1);
                return true;
            }
            if (vecinoDeCenI.getPaso() == 0)
                this.tocaVisitar.add(vecinoDeCenI);
            vecinoDeCenI.setMinPaso(c.getPaso() + 1);
        }
    }
    return false;
}
```

En este método se lleva a cabo realmente el algoritmo de búsqueda, se testean todos los vecinos y se los coloca en la cola si es que no están ya allí.

Se puede ver que se tiene una condición para saber si corresponde a la celda de destino.

Solamente si NO ha sido visitado se lo coloca en la cola (es justamente lo que dice el algoritmo). En todos los casos se marca el paso que correspondería, esto para saber en cuantos saltos alcanzamos a esta celda.

Haciendo una representación gráfica del algoritmo en ejecución tenemos lo siguiente:



Aquí se puede ver claramente el ID de cada celda y el paso escrito justo a su lado. En este caso el ejercicio pedía ir del 13 al 39. Algo que es interesante (y los colores nos ayudan a eso) es ver la forma en que ataca BFS este problema y de donde lleva claramente su nombre: búsqueda en anchura.

EDD DINÁMICA NO LINEAL: REDES

Ya hemos visto que los grafos se pueden complicar más cuando se utilizan pesos en los arcos y cuando estos pueden ser unidireccionales o bidireccionales.

Las redes son grafos que además tienen la complejidad de añadir el peso en los arcos. Con las redes se pueden representar fácilmente:

- Redes de Carreteras
- Redes ferroviarias
- Redes aéreas
- Redes de computadoras

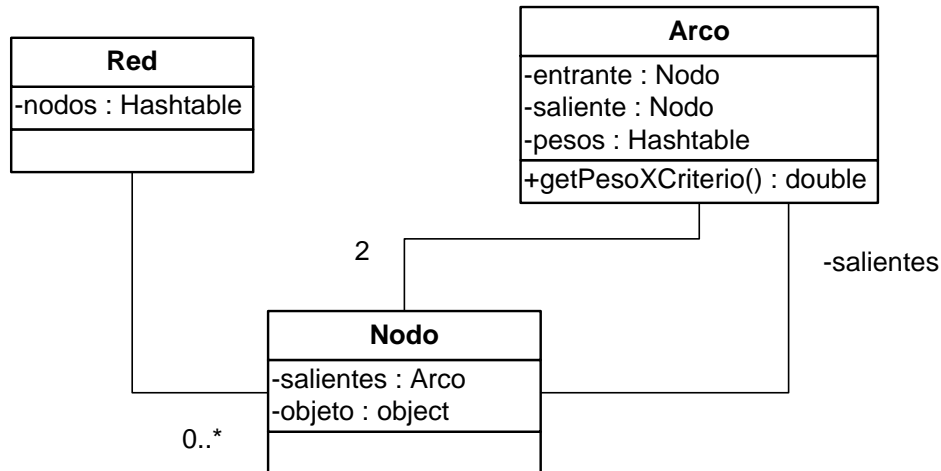
Este tipo de estructuras son utilizadas frecuentemente para realizar simulaciones.

IMPLEMENTACIÓN DE REDES

Las redes se pueden representar, al igual que los grafos, de dos maneras:

- Matriz de adyacencia
- Programación orientada objetos

Solamente veremos aquí la representación orientada objetos, que difiere un poco de lo que se vio anteriormente en el caso de los grafos. La implementación con matriz de adyacencia es igual a la de los grafos.



Esta estructura permite tener un grafo con cualquier cantidad de nodos, que a su vez pueden tener cualquier cantidad de arcos enlazados. Por otro lado, cada arco puede tener una lista de pesos asociada, con lo cual se pueden definir en el mismo grafo varios criterios, por ejemplo: quiero saber cuál es el camino más corto entre Santa Cruz y La Paz y también cuál es el más caro.

ALGORITMO DE DIJKSTRA

Encontrar el camino más corto en una red es muy diferente de aplicar el algoritmo de BFS ya que los pesos de los arcos hacen que la elección del mejor camino sea mucho más difícil.

Dijkstra es una de las grandes mentes del siglo XX en cuanto a programación se refiere. De las contribuciones más importantes que tiene son:

- El discurso el Humilde Programador que dictó en 1972 cuando le dieron el premio Turing¹²
- El artículo 'Un caso contra la palabra GOTO', que fue renombrado por el editor como 'GOTO considerado dañino' y que dio lugar a TODA una serie de artículos de otros pensadores con el título 'X considerado dañino'¹³
- Y Finalmente, el gran algoritmo de Dijkstra que veremos en este documento.

Para implementar este algoritmo se necesita una estructura de grafo. La estructura debe implementar un método de búsqueda del camino más corto, el pseudo código del método es el siguiente:

```

1  metodo Dijkstra(objGrafo, objNodoInicial):
2      for each Nodo v in objGrafo:
3          dist[v] := infinito
4          anterior[v] := vacio
5
6      dist[objNodoInicial] := 0
7      Q := the set of all nodes in Graph
8
9      while Q is not empty:                // The main loop
10         u := Nodo en Q con la más corta distancia
  
```

¹² <http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html>

¹³ <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html>

```

9      if dist[u] == infinito:
10         break

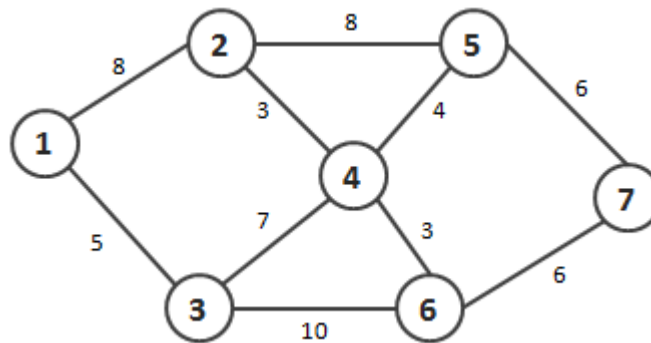
11     quitar u de Q

12     for each vecino v de u:
13         alt := dist[u] + dist_entre(u, v)
14         if alt < dist[v]:
15             dist[v] := alt
16             anterior[v] := u
17     return dist[]

```

Para ver el camino más corto solamente se debe navegar en sentido inverso el arreglo dist.

Para ver la ejecución del algoritmo, hagamos el ejemplo sobre la red definida aquí:



Para ver el estado de la ejecución del algoritmo en todo momento debemos ver los valores que se encuentran en:

- `dist[]`: arreglo de enteros cuyo índice es un nodo. El entero contenido significa la distancia desde el nodo origen.
- `anterior[]`: arreglo de enteros cuyo índice es un nodo, que ayuda a decidir por el menor número en la visita de un nodo.
- `Q`: conjunto de nodos
- `u(Q)`: nodo en `Q` con la distancia más corta.
- `v(u)`: conjunto de vecinos de `u`.

Numero de línea en algoritmo	Notas	Estado de la ejecución	
2 - 4	Se inicializan los arreglos dist y anterior	dist[] =	{∞, ∞, ∞, ∞, ∞, ∞, ∞}
		anterior[] =	{0, 0, 0, 0, 0, 0, 0}
5 - 6	Se inicializa el arreglo Q y se marca la distancia al nodo de inicio que sería 0	dist[] =	{0, ∞, ∞, ∞, ∞, ∞, ∞}
		anterior[] =	{0, 0, 0, 0, 0, 0, 0}
		Q[] =	{1, 2, 3, 4, 5, 6, 7}
7	Q no está vacío		
8 - 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 1 tiene distancia 0	dist[] =	{0, ∞, ∞, ∞, ∞, ∞, ∞}
		anterior[] =	{0, 0, 0, 0, 0, 0, 0}
		Q[] =	{1, 2, 3, 4, 5, 6, 7}
		u(Q) =	1
11	Se quita el nodo 1 de Q	dist[] =	{0, ∞, ∞, ∞, ∞, ∞, ∞}

		$\text{anterior[]} = \{0, 0, 0, 0, 0, 0, 0\}$ $Q[] = \{2, 3, 4, 5, 6, 7\}$ $u(Q) = 1$	
12	Para cada vecino de u. En este caso el primer vecino de u es 2 y luego el nodo 3	$\text{dist[]} = \{0, \infty, \infty, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 0, 0, 0, 0, 0, 0\}$ $Q[] = \{2, 3, 4, 5, 6, 7\}$ $u(Q) = 1$ $v(u)[] = \{2, 3\}$	
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $\text{dist}[1] + \text{dist}(1,2) = 0 + 8$	$\text{dist[]} = \{0, \infty, \infty, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 0, 0, 0, 0, 0, 0\}$ $Q[] = \{2, 3, 4, 5, 6, 7\}$ $u(Q) = 1$ $v(u)[] = \{2, 3\}$ $\text{alt} = 8$	
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es infinito)	$\text{dist[]} = \{0, 8, \infty, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 0, 0, 0, 0, 0\}$ $Q[] = \{2, 3, 4, 5, 6, 7\}$ $u(Q) = 1$ $v(u)[] = \{2, 3\}$ $\text{alt} = 8$	
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $\text{dist}[1] + \text{dist}(1,3) = 0 + 5$	$\text{dist[]} = \{0, 8, \infty, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 0, 0, 0, 0, 0\}$ $Q[] = \{2, 3, 4, 5, 6, 7\}$ $u(Q) = 1$ $v(u)[] = \{2, 3\}$ $\text{alt} = 5$	
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es infinito)	$\text{dist[]} = \{0, 8, 5, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 1, 0, 0, 0, 0\}$ $Q[] = \{3, 2, 4, 5, 6, 7\}$ $u(Q) = 1$ $v(u)[] = \{2, 3\}$ $\text{alt} = 5$	
7	Q no está vacío		
8 – 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 3 tiene distancia 5	$\text{dist[]} = \{0, 8, 5, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 1, 0, 0, 0, 0\}$ $Q[] = \{3, 2, 4, 5, 6, 7\}$ $u(Q) = 3$	
11	Se quita el nodo 3 de Q	$\text{dist[]} = \{0, 8, 5, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 1, 0, 0, 0, 0\}$ $Q[] = \{2, 4, 5, 6, 7\}$ $u(Q) = 3$	
12	Para cada vecino de u. En este caso el primer vecino de u es 4 y luego el nodo 5	$\text{dist[]} = \{0, 8, 5, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 1, 0, 0, 0, 0\}$ $Q[] = \{2, 4, 5, 6, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$	
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $\text{dist}[3] + \text{dist}(3,4) = 5 + 7$	$\text{dist[]} = \{0, 8, 5, \infty, \infty, \infty, \infty\}$ $\text{anterior[]} = \{0, 1, 1, 0, 0, 0, 0\}$ $Q[] = \{2, 4, 5, 6, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$ $\text{alt} = 12$	

14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es infinito)	$dist[] = \{0, 8, 5, \mathbf{12}, \infty, \infty, \infty\}$ $anterior[] = \{0, 1, 1, \mathbf{3}, 0, 0, 0\}$ $Q[] = \{2, 4, 5, 6, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$ $alt = 12$
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $dist[3] + dist(3,6) = 5 + 10$	$dist[] = \{0, 8, 5, 12, \infty, \infty, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, 0, 0\}$ $Q[] = \{2, 4, 5, 6, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$ $alt = \mathbf{15}$
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es infinito)	$dist[] = \{0, 8, 5, 12, \infty, \mathbf{15}, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, \mathbf{3}, 0\}$ $Q[] = \{2, 4, \mathbf{6}, \mathbf{5}, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$ $alt = 15$
7	Q no está vacío	
8 - 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 2 tiene distancia 8	$dist[] = \{0, 8, 5, 12, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, 3, 0\}$ $Q[] = \{2, 4, 6, 5, 7\}$ $u(Q) = \mathbf{2}$
11	Se quita el nodo 2 de Q	$dist[] = \{0, 8, 5, 12, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, 3, 0\}$ $Q[] = \{\mathbf{4}, \mathbf{6}, \mathbf{5}, \mathbf{7}\}$ $u(Q) = 2$
12	Para cada vecino de u. En este caso el primer vecino de u es 4 y luego el nodo 5	$dist[] = \{0, 8, 5, 12, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, 3, 0\}$ $Q[] = \{4, 6, 5, 7\}$ $u(Q) = 2$ $v(u)[] = \{\mathbf{4}, \mathbf{5}\}$
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $dist[2] + dist(2,4) = 8 + 3$	$dist[] = \{0, 8, 5, 12, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, 3, 0, 3, 0\}$ $Q[] = \{4, 6, 5, 7\}$ $u(Q) = 2$ $v(u)[] = \{4, 5\}$ $alt = \mathbf{11}$
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es 12)	$dist[] = \{0, 8, 5, \mathbf{11}, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, \mathbf{2}, 0, 3, 0\}$ $Q[] = \{4, 6, 5, 7\}$ $u(Q) = 2$ $v(u)[] = \{4, 5\}$ $alt = 11$
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $dist[2] + dist(2,5) = 8 + 8$	$dist[] = \{0, 8, 5, 11, \infty, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 0, 3, 0\}$ $Q[] = \{4, 6, 5, 7\}$ $u(Q) = 2$ $v(u)[] = \{4, 5\}$ $alt = \mathbf{16}$
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es	$dist[] = \{0, 8, 5, 11, \mathbf{16}, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, \mathbf{2}, 3, 0\}$

	infinito)	$Q[] = \{4, 6, 5, 7\}$ $u(Q) = 3$ $v(u)[] = \{4, 6\}$ $alt = 15$	
7	Q no está vacío		
8 – 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 4 tiene distancia 11	$dist[] = \{0, 8, 5, 11, 16, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 2, 3, 0\}$ $Q[] = \{4, 6, 5, 7\}$ $u(Q) = 4$	
11	Se quita el nodo 4 de Q	$dist[] = \{0, 8, 5, 11, 16, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 2, 3, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$	
12	Para cada vecino de u. En este caso el primer vecino de u es 5 y luego el nodo 6	$dist[] = \{0, 8, 5, 11, 16, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 2, 3, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$ $v(u)[] = \{5, 6\}$	
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $dist[4] + dist(4,5) = 11 + 4$	$dist[] = \{0, 8, 5, 11, 16, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 2, 3, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$ $v(u)[] = \{5, 6\}$ $alt = 15$	
14-16	Condición para ir armando el camino más óptimo. En este caso la distancia calculada es menor que la actual (que es 16)	$dist[] = \{0, 8, 5, 11, 15, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 4, 3, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$ $v(u)[] = \{5, 6\}$ $alt = 15$	
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $dist[4] + dist(4, 6) = 11 + 3$	$dist[] = \{0, 8, 5, 11, 15, 15, \infty\}$ $anterior[] = \{0, 1, 1, 2, 4, 3, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$ $v(u)[] = \{5, 6\}$ $alt = 14$	
14-16	Condición para ir armando el camino más óptimo. En este caso la distancia calculada es mayor que la actual (que es 15), no se cambia nada	$dist[] = \{0, 8, 5, 11, 15, 14, \infty\}$ $anterior[] = \{0, 1, 1, 2, 4, 4, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 4$ $v(u)[] = \{5, 6\}$ $alt = 14$	
7	Q no está vacío		
8 – 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 6 tiene distancia 15	$dist[] = \{0, 8, 5, 11, 15, 14, \infty\}$ $anterior[] = \{0, 1, 1, 2, 4, 4, 0\}$ $Q[] = \{6, 5, 7\}$ $u(Q) = 6$	
11	Se quita el nodo 6 de Q	$dist[] = \{0, 8, 5, 11, 15, 14, \infty\}$ $anterior[] = \{0, 1, 1, 2, 4, 4, 0\}$ $Q[] = \{5, 7\}$ $u(Q) = 6$	

12	Para cada vecino de u. En este caso el vecino de u es el nodo 7	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, \infty\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 0\}$ $Q[] = \{5, 7\}$ $u(Q) = 6$ $v(u)[] = \{7\}$
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $\text{dist}[6] + \text{dist}(6, 7) = 14 + 6$	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, \infty\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 0\}$ $Q[] = \{5, 7\}$ $u(Q) = 6$ $v(u)[] = \{7\}$ $\text{alt} = 20$
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es menor que la actual (que es infinito)	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{5, 7\}$ $u(Q) = 6$ $v(u)[] = \{7\}$ $\text{alt} = 20$
7	Q no está vacío	
8 - 10	Se busca el nodo en Q que tenga la menor distancia, para esto conviene que dist sea una lista ordenada por distancia. En este caso el nodo 5 tiene distancia 15	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{5, 7\}$ $u(Q) = 5$
11	Se quita el nodo 5 de Q	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{7\}$ $u(Q) = 5$
12	Para cada vecino de u. En este caso el vecino de u es el nodo 7	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{7\}$ $u(Q) = 5$ $v(u)[] = \{7\}$
13	Se calcula alt con la distancia de u al vecino analizado. En este caso, la $\text{dist}[5] + \text{dist}(5, 7) = 15 + 6$	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{7\}$ $u(Q) = 6$ $v(u)[] = \{7\}$ $\text{alt} = 21$
14-16	Condición para ir armando el camino más optimo. En este caso la distancia calculada es mayor que la actual (que es 20), no se cambia nada	$\text{dist}[] = \{0, 8, 5, 11, 15, 14, 20\}$ $\text{anterior}[] = \{0, 1, 1, 2, 4, 4, 6\}$ $Q[] = \{7\}$ $u(Q) = 6$ $v(u)[] = \{7\}$ $\text{alt} = 21$

Para la ultima iteración el nodo 7 ya no tiene vecinos que aporten mejoría entonces se lo quita de Q y termina la iteración. Es interesante ver lo que ocurre con el arreglo anterior, nos indica exactamente el camino que hay que recorrer, en sentido inverso:

- En la posición 7, marca el nodo 6.
- En la posición 6, marca el nodo 4.
- En la posición 4, marca el nodo 2

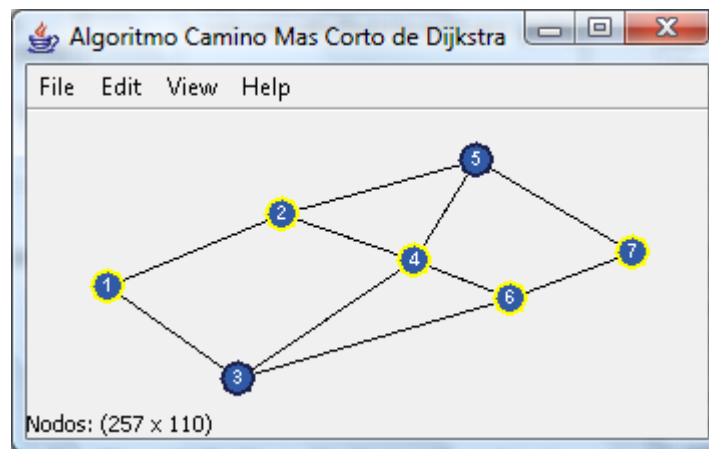
- En la posición 2, marca el nodo 1

Y nuestro camino es entonces: $1 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7$

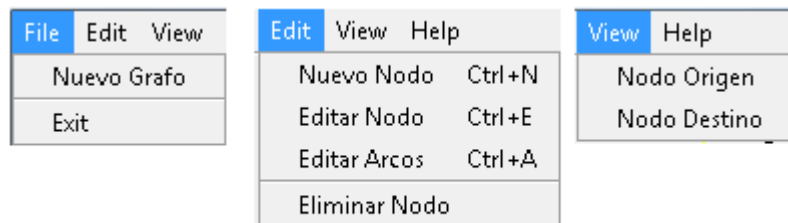
EJEMPLO DE IMPLEMENTACIÓN

Vamos a realizar la implementación del algoritmo de Dijkstra por su importancia y porque integra varias estructuras de datos y patrones que hemos visto a lo largo del capítulo. En este caso la implementación que haremos será de un pequeño programa de creación de grafos donde el peso de los arcos esta dado por la distancia que hay entre los nodos que unen ese arco.

Aquí vemos el ejemplo de la interface:

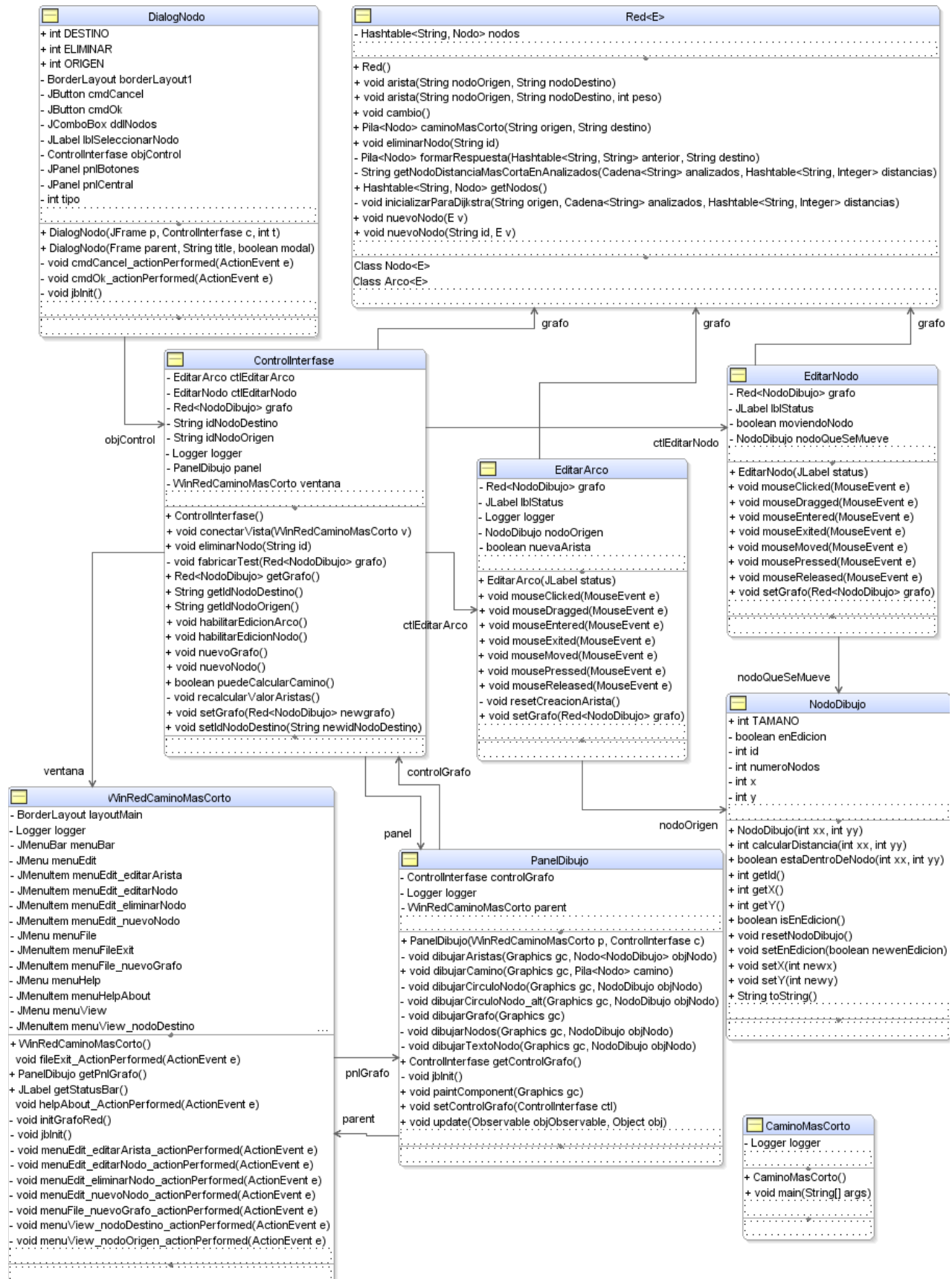


Los arcos son bidireccionales automáticamente y el calculo del peso de los arcos también es automatico. Se puede ver que se ejecuta el algoritmo de Dijkstra todo el tiempo para saber el camino mas corto entre dos nodos. Se han agregado varias funcionalidades:



- **Nuevo Grafo:** crea una red con dos nodos (1 y 2) unidos por un arco y coloca el origen en 1 y el destino en 2.
- **Nuevo Nodo:** Permite crear un nuevo nodo en una posición estatica
- **Editar Nodo:** Permite ir con el mouse y mover un nodo al seleccionarlo directamente en la interface
- **Editar Arco:** Permite hacer clic en un nodo y luego en otro para crear un arco entre los dos.
- **Nodo Origen y Destino:** Permiten seleccionar de una lista de nodos el nodo que será tomado como origen y destino.

El diagrama de clases propuesto sigue a continuación:



El método interesante en esta implementación es por supuesto el método de búsqueda de camino más corto de Dijkstra. Este método es accionado cada vez que cambia la configuración de la red: si se aumenta o elimina un nodo, si se lo mueve, si se crea un nuevo arco.

```
/**
 * Calcula el camino mas corto entre dos nodos. Los nodos deben existir
 * en el grafo
 * @param origen El identificador del nodo de origen
 * @param destino El identificador del nodo destino
 * @return El numero de pasos o saltos que se necesita para ir de de a a.
 */
public Pila<Nodo<E>> caminoMasCorto(String origen, String destino) {
    if (nodos.get(origen) == null || nodos.get(destino) == null)
        return null;
    // Coloca las distancias de todos los nodos a infinito
    // Se agrma una cadena analizados con todos los nodos
    Cadena<String> analizados = new Cadena<String>();
    Hashtable<String, Integer> distancias =
        new Hashtable<String, Integer>();
    Hashtable<String, String> anterior = new Hashtable<String, String>();

    inicializarParaDijkstra(origen, analizados, distancias);

    // comienza la iteracion
    while (analizados.tamano() > 0) {
        String idNodo =
            getNodoDistanciaMasCortaEnAnalizados(analizados, distancias);
        // Lo quita de la lista de analizados
        analizados.eliminarContenido(idNodo);

        if (idNodo.equals(destino)) {
            Pila<Nodo<E>> resultado = formarRespuesta(anterior, destino);
            return resultado;
        }

        Iterator<Arco<E>> i = nodos.get(idNodo).getSalientes().iterator();
        while (i.hasNext()) {
            Arco<E> objArco = i.next();
            // Solamente los que estan en analizados
            if (!analizados.existe(objArco.getDestino().getId())) {
                continue;
            }

            int calculoDistancia =
                distancias.get(idNodo).intValue() + objArco.getPeso();
            int distanciaParaNodoVecino =
                distancias.get(objArco.getDestino().getId()).intValue();
            if (calculoDistancia < distanciaParaNodoVecino) {
                distancias.put(objArco.getDestino().getId(),
                    new Integer(calculoDistancia));
                anterior.put(objArco.getDestino().getId(), idNodo);
            }
        }
    }
}
```

```
    return null;
}
```

Se puede ver que se sigue paso a paso lo indicado en el pseudo código. La pila devuelve naturalmente la lista de nodos por los que hay que pasar para ir de origen a destino. Son estos nodos que pintaremos de otro color.

PREGUNTAS

Pregunta 8. Optimización Juego de la Vida

¿Cómo se puede optimizar el código de un paso del juego de la vida para que no tengamos una familia de ifs uno detrás del otro? Analice si el código sigue siendo legible.

Pregunta 9. Flip Horizontal y Vertical de una imagen

Escribir el código de la clase FlipHorizontal y la clase FlipVertical para implementar el proceso de volcar horizontal o verticalmente la imagen. El volcado horizontal es como cuando uno observa la imagen en un espejo lateral.

Pregunta 10. Histograma de una imagen

Proponga el modelo de diagrama de clases para dibujar el histograma de una imagen. Indique el código central donde se calcula el histograma de una imagen.

Pregunta 11. Implementación de cola con Generics

Implementar la clase ColaEstatica con Generics para un máximo de 100 elementos. Inspírese del código de implementación de una pila.

Pregunta 12. Unión de dos cadenas

En la clase CadenaSimple del texto, escribir el código del método

```
public void unirCon(CadenaSimple<E> otraCadena) {
    ...
}
```

Para que se puedan unir dos cadenas.

Pregunta 13. Iterador para utilizar foreach

Modifique la clase CadenaSimple para que implemente la interface Iterable (con Generics). Comprobar que la implementación está correcta ejecutando este código

```
CadenaSimple<String> palabras = new CadenaSimple<String>();
palabras.insertar("hugo");
palabras.insertar("paco");
palabras.insertar("luis");

for(String s : palabras) {
    System.out.println(s);
}
```

Pregunta 14. Implementación métodos para cadena doble

Implemente el método de insertar(int posición, E obj) a la estructura de datos cadena doble donde el elemento pasado como parámetro es colocado en la posición pasada como parámetro. Considere que utiliza una cadena doble.

| **Pregunta 15. Eliminar para anillo doble**

Implemente el método eliminar(int posición) a la estructura de datos anillo doble donde se elimina el elemento en la posición pasada como parámetro.

| **Pregunta 16. Implementación de inserción en cadena ordenada**

Escriba el código del método insertar(E obj) en una Cadena que sea ordenada. Es decir, al insertar el objeto, la Cadena lo coloca en el lugar que corresponde y entonces la Cadena siempre esta ordenada. Considere que se utiliza una cadena doble.

| **Pregunta 17. Evaluación de una expresión aritmética**

Utilice una estructura de datos árbol para representar una operación aritmética. EL árbol será binario. En cada nodo que no sea hoja se coloca un operador aritmético (+, -, *, /) y en las hojas se colocara el numero. En especial describa el código que debe realizar para evaluar el valor de la expresión siguiendo el diagrama de clases que proponga.

| **Pregunta 18. Optimizaciones representación de grafos**

¿Qué optimizaciones puede proponer a la representación de grafos por matriz de adyacencia? De igual manera, explique al menos 2 mejoras que se puede realizar a la representación orientada objetos de un grafo.

| **Pregunta 19. Código DFS**

Realice el diagrama de clases para representar un grafo donde cada nodo puede tener hasta 3 aristas. ¿Cuál sería el código para recorrer el algoritmo en DFS y como nos permitiría obtener un camino desde un nodo a cualquier otro?

| **Pregunta 20. Redes en un archivo**

Discuta como podría hacer un programa para cargar redes descritas en un archivo. Describa el formato del archivo. Indique el código para leer el archivo.

ACCESO A DATOS

“Los mejores diseñadores de software utilizaran muchos patrones de diseño unidos armoniosamente y funcionales entre sí para producir un elemento mayor” Erich Gamma

En esta unidad se verán los conceptos necesarios para poder acceder a una fuente persistente de datos. También se podrá ver como favorece el uso extensivo de patrones para lograr una estrategia específica.

El capítulo comienza describiendo varios tipos de persistencia. Luego, se ven los problemas para poder acceder a una base de datos. Finalmente, el capítulo brinda una solución conceptualmente robusta para el manejo de una capa de acceso a datos.

PERSISTENCIA DE TIPOS DE DATOS ABSTRACTOS

Hasta ahora, cada vez que ejecutábamos un programa, hacíamos operaciones y luego cerrábamos el programa. Sin embargo, todas las operaciones que hicimos, los resultados que obtuvimos, las opciones que elegimos, etc.; desaparecen completamente.

Persistencia es cuando logramos que esos datos, esa información resultante de la ejecución, se queden en algún lugar de forma permanente y se pueda recuperar de alguna manera por nuestros programas.

Nuestros datos abstractos son nuestros objetos. El reto es lograr que lo que guardemos de una manera lo recuperemos como objetos. Ejemplo: ¿cómo podemos lograr que una instancia de la clase Persona se pueda quedar grabada en el disco duro?

ARCHIVOS / FICHEROS

Un archivo es una unidad persistente en el disco duro. El concepto de archivos no es posible entenderlo sin el concepto de sistema de ficheros. Un sistema de ficheros es la forma en que se guardan los archivos en un dispositivo físico. Por ejemplo:

- Cómo sabe la computadora buscar los archivos y carpetas dentro de un USB?
 - Lo sabe porque al inicio del dispositivo se encuentra una región del mismo con un formato específico que indica registro por registro el nombre y la dirección en el USB de cada uno de los archivos en una estructura de jerarquía como es una carpeta.

A este formato se le llama sistema de ficheros. Existen varios sistemas de ficheros en los sistemas de computación, los más conocidos son los siguientes:

Sistema de archivos	Fabricante y uso	Descripción
FAT32	Microsoft. Ampliamente utilizado en disquetes (formato FAT8) y en los Flash actuales.	Es un sistema de ficheros sencillo donde la primera parte del dispositivo contiene una tabla de encabezados de los archivos y las direcciones en el disco donde se encuentra el resto del archivo.
NTFS	Microsoft. Este es el sistema de ficheros que utiliza el sistema operativo Windows.	Es una versión mejorada del sistema de archivos FAT32. Por la estructura del mismo puede ver que mover o borrar archivos puede tomar cierto tiempo.

EXT3

Es el sistema de archivos utilizado en el sistema operativo Linux.

La estructura de archivos se asemeja a la de un árbol y como tal en el disco está guardado el árbol con todos los nodos archivos.

Un archivo no es más que el nodo de un sistema de archivos. Todos los lenguajes de programación permiten realizar operaciones sobre los archivos de manera relativamente sencilla y estándar.

LECTURA Y ESCRITURA EN FICHEROS

Escribir y leer de un archivo en Java es sumamente sencillo. A notar que se utiliza todo el tiempo el tema del patrón Decorator para la librería java.io:

Este es el código para leer línea por línea un archivo de texto:

```
File f = new File("datos.txt");
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);
String línea = "";
while(br.ready()) {
    línea = br.readLine();
    System.out.println(línea);
}
br.close();
fr.close();
```

Este es el código para escribir una línea en un archivo:

```
FileOutputStream fos = new FileOutputStream("datos.txt");
PrintWriter out = new PrintWriter(fos);
out.println("hola a todos");
out.close();
fos.close();
```

En algunos casos, para que efectivamente la línea esté escrita en el archivo justo después de que la escribimos se puede utilizar el método `flush()` que limpia el búfer de escritura y entonces hace que todos los datos del búfer pasen al archivo.

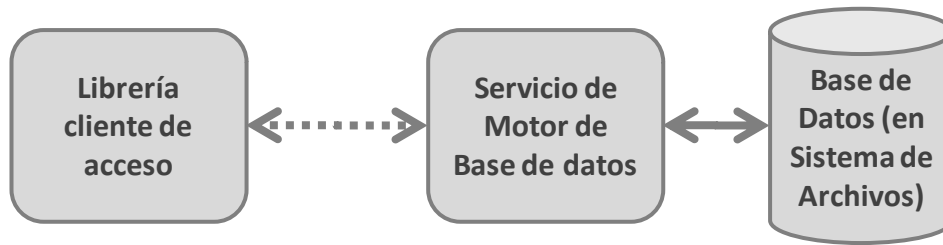
Ahora, una vez que ya podemos escribir en archivos, debemos saber cómo escribir en archivos estructuras complejas, por ejemplo, la clase `Persona`, cómo podemos escribir en un archivo un objeto `Persona`, o una lista de objetos `Persona`?

Existen variadas formas de grabar la información de un objeto en un archivo. En TODOS los casos se debe definir un **FORMATO** del archivo que permita hacer la conversión entre el archivo y el objeto. Esto es lo que hace la gente de Microsoft por ejemplo con el fichero Word, que al final, dentro del programa, se convierte en un objeto de tipo `Documento`.

El mejor formato para guardar registros es una base de datos. Estos son ficheros especiales tipo Excel que pueden contener cientos de miles de registros de información y tienen maneras estándar de lidiar con el alta, baja y/o modificación de estos registros.

BASES DE DATOS

Son poderosos contendores de información que funcionan casi siempre con el esquema cliente servidor. Es decir, el acceso al archivo que contiene la o las bases de datos se hace a través de un programa. El diagrama siguiente indica esta situación:



Lo más importante a retener es que el archivo que contiene la base de datos es solamente accedido por el motor de la base de datos. Y la única manera que desde nuestros programas o por nuestra cuenta accedamos al motor es a través de un programa que haga de cliente de acceso al motor de la base de datos.

Esta infraestructura permite que el cliente pueda conectarse remotamente al motor de la base de datos y ampliar entonces la disponibilidad de la información contenida en la base de datos.

CONCEPTOS IMPORTANTES

Los conceptos más importantes incluyen las estructuras en las que se guarda la información en el motor: tabla, atributo y tupla o registro.

- **Tabla:** Una tabla es casi exactamente lo mismo que una hoja Excel: tiene columnas y se pueden definir cuantas líneas se quiera.
- **Atributo:** Los atributos de una tabla indican las columnas que tiene disponibles esa tabla.
- **Tupla:** Una tupla dentro de una tabla corresponde a un registro que contiene la información que describe el atributo.

EL LENGUAJE SQL

Para poder operar sobre la base de datos, los creadores de los motores de bases de datos han ideado todo un lenguaje para poder acceder a las mismas. Es decir, las conversaciones que el cliente tiene con el servidor de base de datos deben estar en el lenguaje SQL soportado por el motor de base de datos al que se accede.

Aquí veremos algunos ejemplos del uso de este lenguaje. Este texto NO ha sido elaborado con el objetivo de enseñar SQL pero si es importante que el usuario tenga las herramientas iniciales para trabajar con estos sistemas.

A continuación se puede observar el script para crear una tabla en MySQL:

```
CREATE TABLE IF NOT EXISTS `usuario` (  
  `ID` BIGINT(20) NOT NULL ,  
  `LOGIN` VARCHAR(255) NULL DEFAULT NULL ,  
  `CLAVE` VARCHAR(255) NULL DEFAULT NULL ,  
  PRIMARY KEY (`ID`) )
```

La tabla usuario contendrá tuplas con las siguientes columnas: id (numero entero), login (cadena de hasta 255 caracteres que por defecto es nulo), clave (cadena de hasta 255 caracteres que por defecto es nulo).

A parte de todo esto, se podrán encontrar los registros dando como referencia nada mas el id, que debe ser único (llave primaria id).

Podemos hacer una consulta para conocer todas las tuplas que se encuentran en la tabla actualmente:

```
Select id, login, clave from usuario
```

Aquí la consulta para insertar una nueva tupla:

```
Insert into usuario  
(id, login, clave)  
Values  
(1, 'vladimir', 'passw0rd')
```

Aquí la consulta para actualizar informacion de la tupla anterior:

```
update usuario  
set login='vladimir calderon'  
where id = 1
```

Finalmente aquí la consulta para eliminar una tupla:

```
Delete from usuario where id=1
```

MOTORES DE BASES DE DATOS

Existen varios motores de base de datos con sus diferencias. Entre las características más importantes que estos programas han adquirido con los años:

- Indexación con arboles B
- Manejo de redundancia
- Manejo de usuarios, autenticación y seguridad
- Encriptación de los datos de algunas columnas y/o tablas
- Vistas de tablas
- Procedimientos almacenados
- Condiciones de validación para los datos que se graban y/o cambian
- Registro de todos los cambios en la base de datos para hacer seguimiento
- Optimización de las consultas mediante índices.

Los más conocidos son los siguientes:

Motor de Base de Datos	Descripción
Sql Server	de Microsoft, tiene una versión gratuita (Express) que, aunque limitada, permite realizar las operaciones básicas de base de datos
Oracle	de Oracle / Sun tiene una versión gratuita (Oracle XE) que, aunque limitada, permite realizar las operaciones básicas de base de datos
MySQL	Base de datos gratuita que fue adquirida por Sun, es la que utiliza Google
PostgreSQL	base de datos gratuita muy poderosa y preferida por los usuarios de software gratuitos

LECTURA Y ESCRITURA EN BASE DE DATOS

Para poder escribir y leer de una base de datos se utilizan consultas SQL que pueden ser ejecutadas directamente por un cliente SQL o por un programa en Java u otro lenguaje.

Java utiliza el JDBC para la conexión con cualquier base de datos. Java Database Connectivity es una interface para poder conectarse con cualquier base de datos.

En el caso de Java, se debe escoger una base de datos (por ejemplo MySQL) y luego escoger su conector JDBC. En el caso de MySQL este conector se puede encontrar aquí:

<http://www.mysql.com/downloads/connector/j/5.1.html>

Este conector se debe utilizar como librería en nuestro proyecto de Java. Una vez realizado esto se pueden realizar los pasos para una conexión con Java en el programa TestMySQL:

```
package capadatos.test;

import java.net.URL;
import java.sql.*;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

/**
 * Esta clase es nada más para realizar un test contra una base de datos
 * Mysql. El script de la tabla a la cual se hace consulta sigue a
 * continuación:
 *
 * create table persona (
 *   id int(10) unsigned NOT NULL auto increment,
 *   nombre varchar(100) default NULL,
 *   fechaNacimiento datetime NOT NULL,
 *   salario float NOT NULL,
 *   PRIMARY KEY (id)
 * );
 */
public class TestMySQL {

    private static Logger log = Logger.getRootLogger();

    public static void main(String[] args) {
        String resource = "/auditoria.properties";
        URL configFileResource = TestMySQL.class.getResource(resource);
        PropertyConfigurator.configure(configFileResource);

        String host = "localhost";
        String database = "estructura";
        String instance = "";
        int port = 3306;
        String username = "root";
```

```
String password = "....."; // colocar la contraseña

String url = "jdbc:mysql://" + host + "/" + database;
Connection con = null;

boolean driverOk = false;
try {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    driverOk = true;
} catch (InstantiationException e) {
    log.error("No pudo instanciar, seguramente no hay la librería", e);
} catch (IllegalAccessException e) {
    log.error("No tiene permisos para instanciar", e);
} catch (ClassNotFoundException e) {
    log.error("No encuentra esa clase, revise librería y versión", e);
}
if (!driverOk)      System.exit(0);

try {
    con = DriverManager.getConnection(url, username, password);
} catch (SQLException e) {
    log.error("No pudo conectarse a la base de datos", e);
    System.exit(0);
}

String query = "SELECT id, nombre, fechaNacimiento, salario FROM Persona";
ResultSet res = null;
try {
    Statement stmt = con.createStatement();
    res = stmt.executeQuery(query);
} catch (SQLException e) {
    log.error("No puede ejecutar la consulta SQL", e);
    System.exit(0);
}

try {
    while (res.next()) {
        int _id = res.getInt("id");
        String _nombre = res.getString("nombre");
        Date _fechaNacimiento = res.getDate("fechaNacimiento");
        double _salario = res.getDouble("salario");

        System.out.println("(" + _id + ") '" + _nombre + "' " +
                           _fechaNacimiento.toString() + " : " + _salario);
    }

    con.close();
} catch (SQLException e) {
    log.error("Error en el motor SQL", e);
}
}
```

Para un estudio exhaustivo de JDBC se puede ir a este enlace:

<http://java.sun.com/docs/books/tutorial/jdbc/index.html>

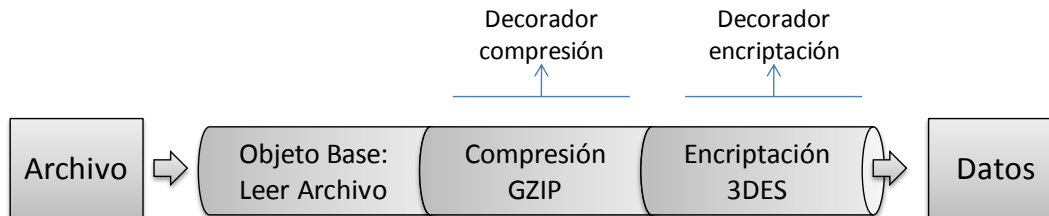
Donde se ve, de manera cuasi exhaustiva las posibilidades que ofrece la librería JDBC y lo que debe implementar cualquier driver JDBC.

PATRÓN DE DISEÑO: DECORATOR

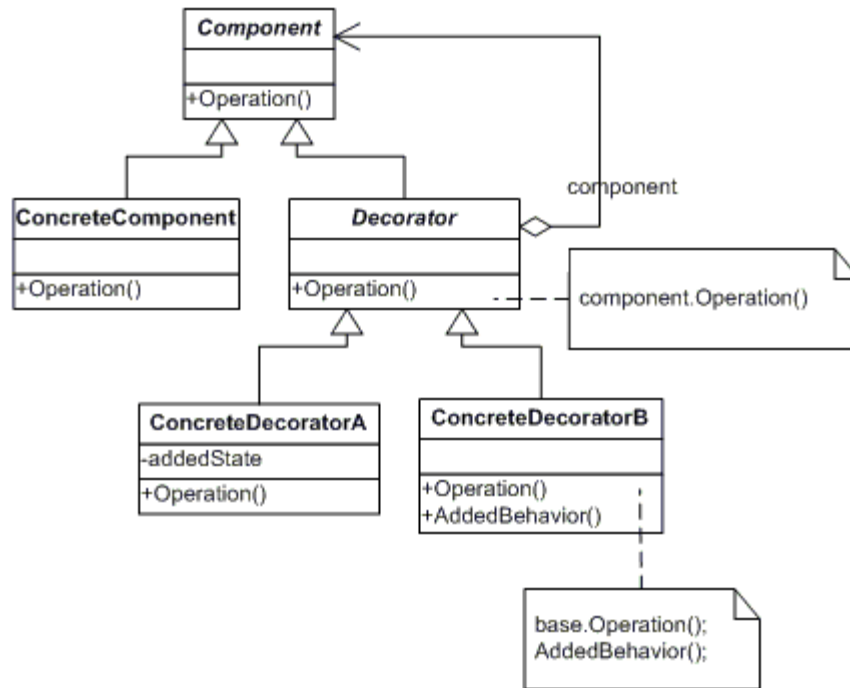
Mientras se hacía el uso de java.io para acceder a la información de un archivo se pudo ver que se instanciaban varios objetos para poder leer de una manera específica. Además, al instanciar estos objetos se utilizaban el objeto anterior. ¿De dónde viene todo esto? ¿Por qué se utiliza así?

El problema es el siguiente: al abrir un archivo y tener un canal de entrada de los datos, nosotros podemos por ejemplo querer 1) que los datos se descompriman a medida que los leemos, 2) que se lea byte por byte en vez de bit a bit, 3) que los datos se descripten a medida que los leemos.

Y ahora, imagine que quisiéramos mezclar estas posibilidades. El manejo de tales posibilidades se torna imposible si no tenemos una estrategia. El principio es que tenemos un objeto base y podemos colocarle cualquier tipo de decoraciones:



Como se ve en la figura, al objeto le hemos pasado el decorador de compresión y encriptación. En resumen, hemos cambiado el comportamiento del objeto base de manera dinámica, es decir, cuando se está ejecutando el programa. El diagrama de clases que resuelve este problema se puede ver a continuación:



Veamos los roles de cada una de estas clases:

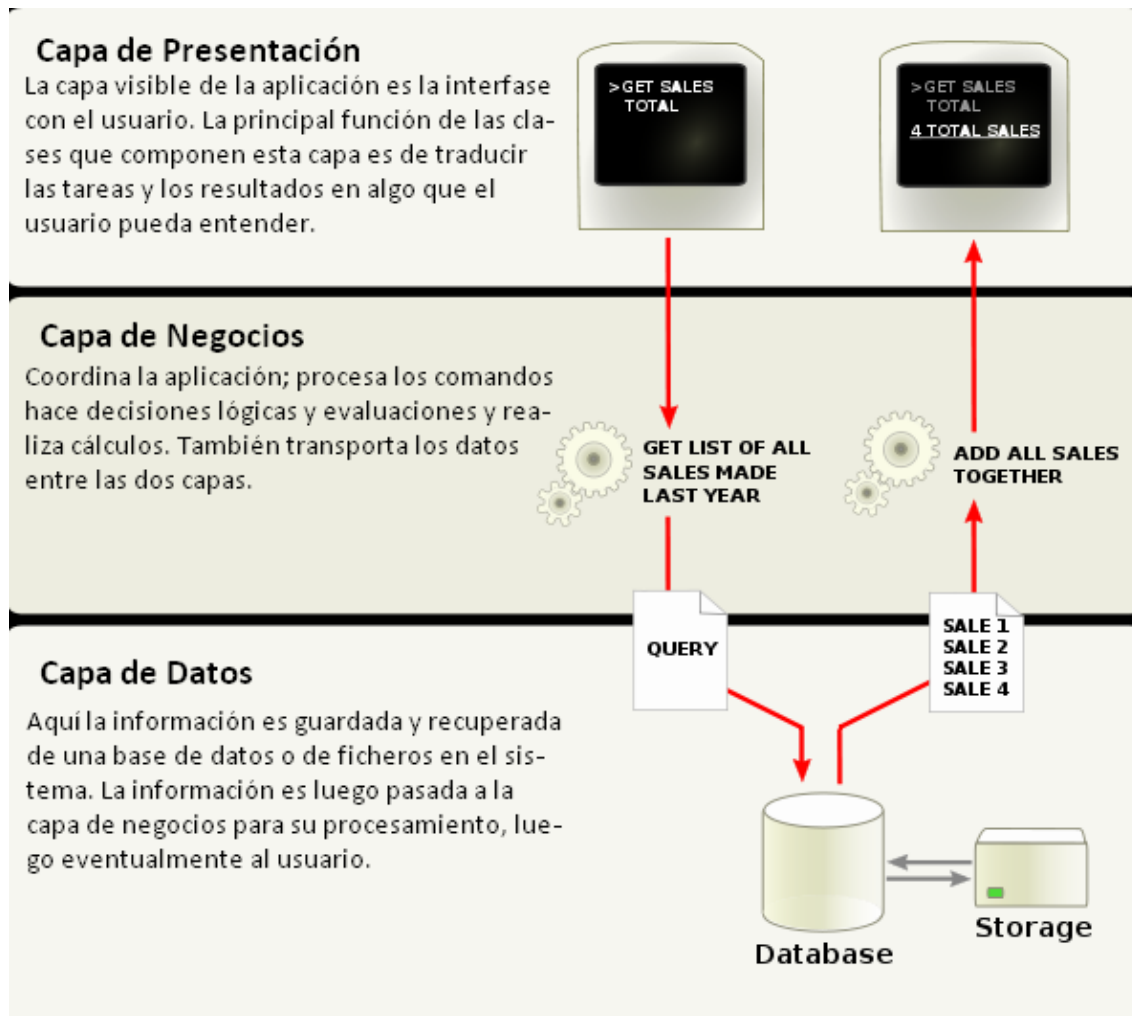
- **Component:** Define la interface para los objetos los que se les puede añadir responsabilidades de manera dinámica.
- **ConcreteComponent:** Define un objeto al que se le pueden colocar responsabilidades (decoraciones).
- **Decorator:** Mantiene una referencia a un objeto de tipo Component y define una interface conforme a la interface de Component.
- **ConcreteDecorator:** Aquí es donde se coloca una responsabilidad que se colocará al objeto base.

PROGRAMACIÓN EN CAPAS

La programación en capas es nada más tener el cuidado de programar de manera que las clases dentro de nuestro programa tengan tareas bien definidas que NO se crucen.

Por ejemplo, en Hanoi, la torre se encarga exclusivamente de la lógica de negocios de la torre. El hecho de que la torre se pueda dibujar o no y como se dibuje depende de otra clase: DibujoTorre.

Típicamente se separan a las clases dependiendo de 3 funcionalidades indicadas en la figura:



En toda aplicación de negocios DEBEN existir estas 3 capas de alguna u otra manera. Las tecnologías cambian sin embargo el ordenar las clases de esta manera es un patrón de diseño que debe ser respetado en todo desarrollo.

Es muy fácil recitar lo que significa la capa de datos, negocios y presentación; sin embargo, poner estos conceptos en práctica es lo que realmente nos pone un paso por delante del resto de los programadores.

Aquí vemos un primer ejemplo muy pequeño para una aplicación que muestra

PATRÓN DE DISEÑO DAO

Muchas aplicaciones de la plataforma J2EE en el mundo real necesitan utilizar datos persistentes en algún momento. Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos, y hay marcadas diferencias en los APIS utilizados para acceder a esos mecanismos de almacenamiento diferentes. Otras aplicaciones podrían necesitar acceder a datos que residen en sistemas diferentes. Por ejemplo, los datos podrían residir en sistemas mainframe, repositorios LDAP, etc. Otro ejemplo es donde los datos los proporcionan servicios a través de sistemas externos como los sistemas de integración negocio-a-negocio (B2B), servicios de tarjetas de crédito, etc.

Las aplicaciones pueden utilizar el API JDBC para acceder a los datos en un sistema de control de bases de datos relacionales (RDBMS). Este API permite una forma estándar de acceder y manipular datos en un almacenamiento

persistente, como una base de datos relacional. El API JDBC permite a las aplicaciones J2EE utilizar sentencias SQL, que son el método estándar para acceder a tablas RDBMS. Sin embargo, incluso dentro de un entorno RDBMS, la sintaxis y formatos actuales de las sentencias SQL podrían variar dependiendo de la propia base de datos en particular.

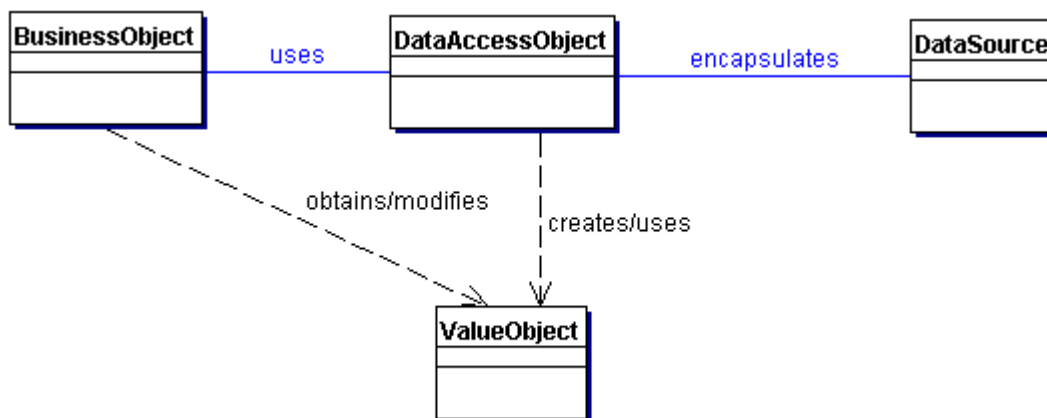
Incluso hay una mayor variación con diferentes tipos de almacenamientos persistentes. Los mecanismos de acceso, los APIs soportados, y sus características varían entre los diferentes tipos de almacenamientos persistentes, como bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos, etc. Las aplicaciones que necesitan acceder a datos de un sistema legal o un sistema dispar (como un mainframe o un servicio B2B) se ven obligados a utilizar APIs que podrían ser propietarios. Dichas fuentes de datos dispares ofrecen retos a la aplicación y potencialmente pueden crear una dependencia directa entre el código de la aplicación y el código de acceso a los datos.

Solución:

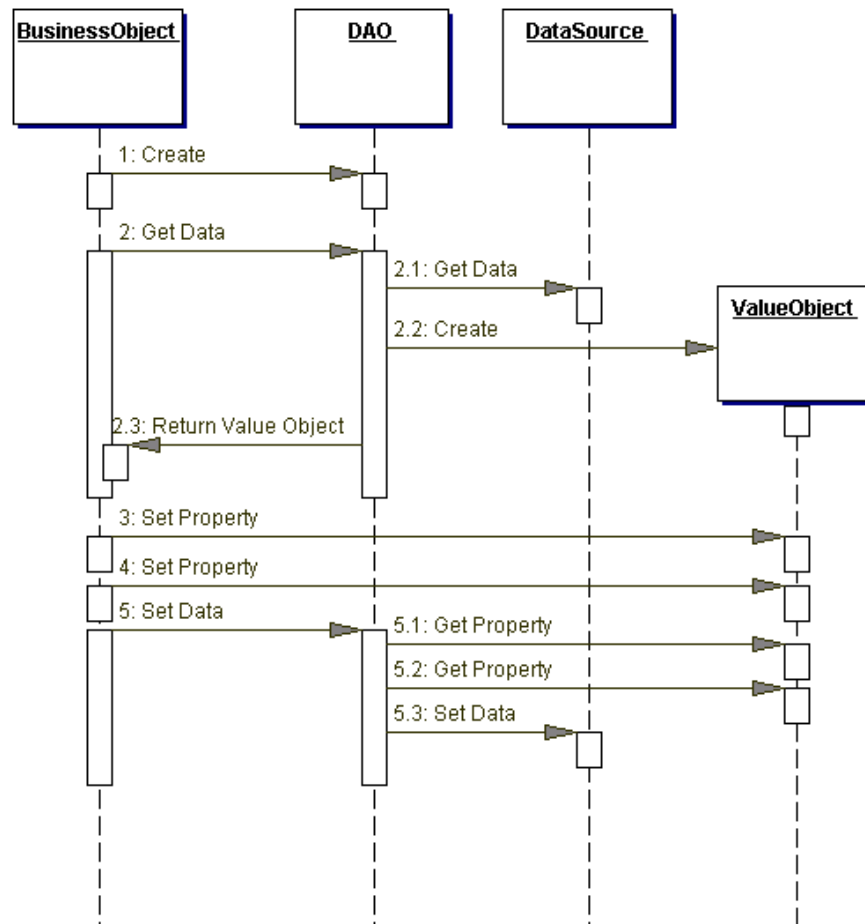
Utilizar un Data Access Object (DAO) para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos.

El DAO implementa el mecanismo de acceso requerido para trabajar con la fuente de datos. Esta fuente de datos puede ser un almacenamiento persistente como una RDBMS, un servicio externo como un intercambio B2B, un repositorio LDAP, o un servicio de negocios al que se accede mediante CORBA Internet Inter-ORB Protocol (IIOP) o sockets de bajo nivel. Los componentes de negocio que tratan con el DAO utilizan un interface simple expuesto por el DAO para sus clientes. El DAO oculta completamente los detalles de implementación de la fuente de datos a sus clientes. Como el interface expuesto por el DAO no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio. Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos.

La siguiente figura muestra las clases base y sus relaciones:



En esta figura se puede ver las responsabilidades de la clase base:



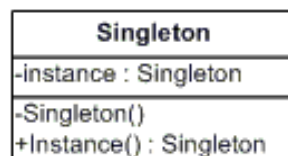
Este diagrama nos muestra que el código para usar este patrón se puede escribir así:

```

DAO d = new DAO();
ValueObject v = d.GetData();
v.setProperty(valor);
d.setData();
  
```

PATRÓN DE DISEÑO: SINGLETON

El patrón de diseño Singleton asegura que una clase tiene una y solamente una instancia y provee un punto de acceso único y global a esa instancia. El diagrama de clase representativo es:



Se puede observar que el constructor es privado, asegurando por lo tanto la NO creación de instancias suplementarias con el constructor. El punto de acceso único a la instancia es el método estático que provee la clase (en el dibujo Instance())

El código en Java que se suele utilizar para el singleton es el siguiente:

```
public class MyClassSingleton {

    private static MyClassSingleton instance;

    //Constructor must be protected or private
    protected MyClassSingleton() { ; }

    //could be synchronized
    public static MyClassSingleton getInstance() {
        if (instance==null)
            instance = new MyClassSingleton()
        return instance;
    }

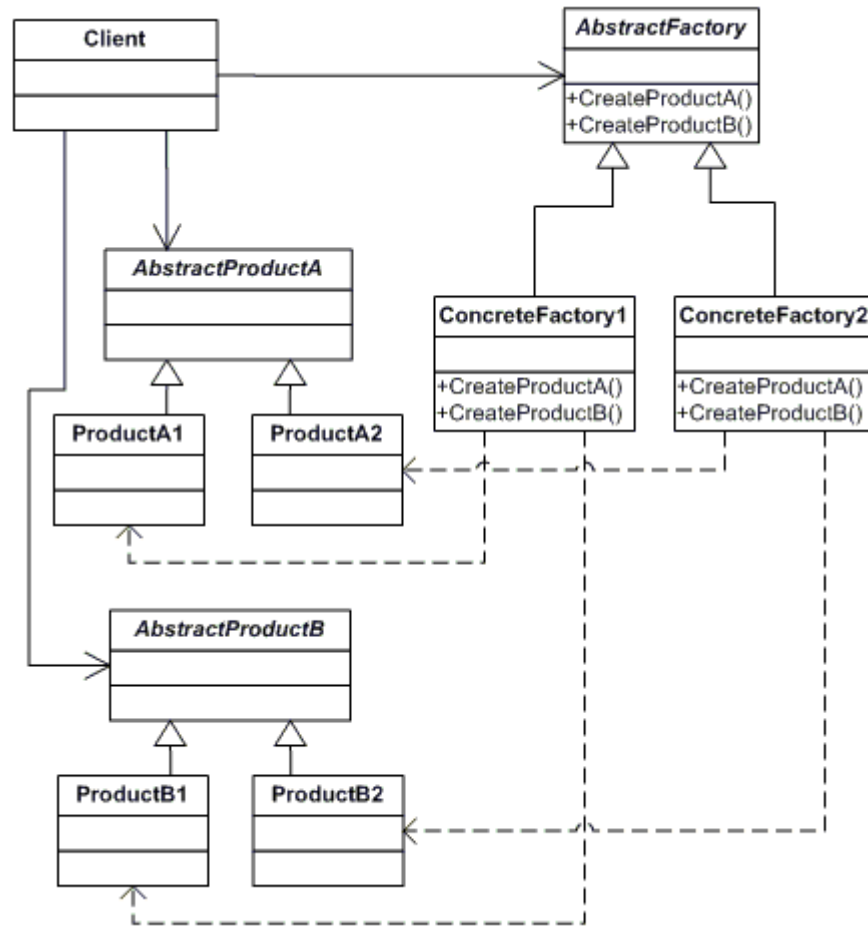
    public void method1(){
        System.out.println("hello singleton");
    }
} //end of class
```

Para evitar problemas con threads (si es que su programa trabaja con varios threads a la vez):

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() { ; }
    public static Singleton getInstance() { return INSTANCE; }
}
```

PATRÓN DE DISEÑO: ABSTRACT FACTORY

Provee una interface para crear familias de objetos relativos entre sí SIN especificar la clase concreta que se va a devolver. El diagrama de clases es el siguiente:



Se deben primero identificar las familias que uno está creando. Luego se deben preparar los constructores para esas familias (Factory). Finalmente, se aconseja para el Factory superclase, utilizar un singleton que permita obtener rápidamente el Factory que se debe obtener.

IMPLEMENTACION DAO CON SINGLETON Y ABSTRACT FACTORY

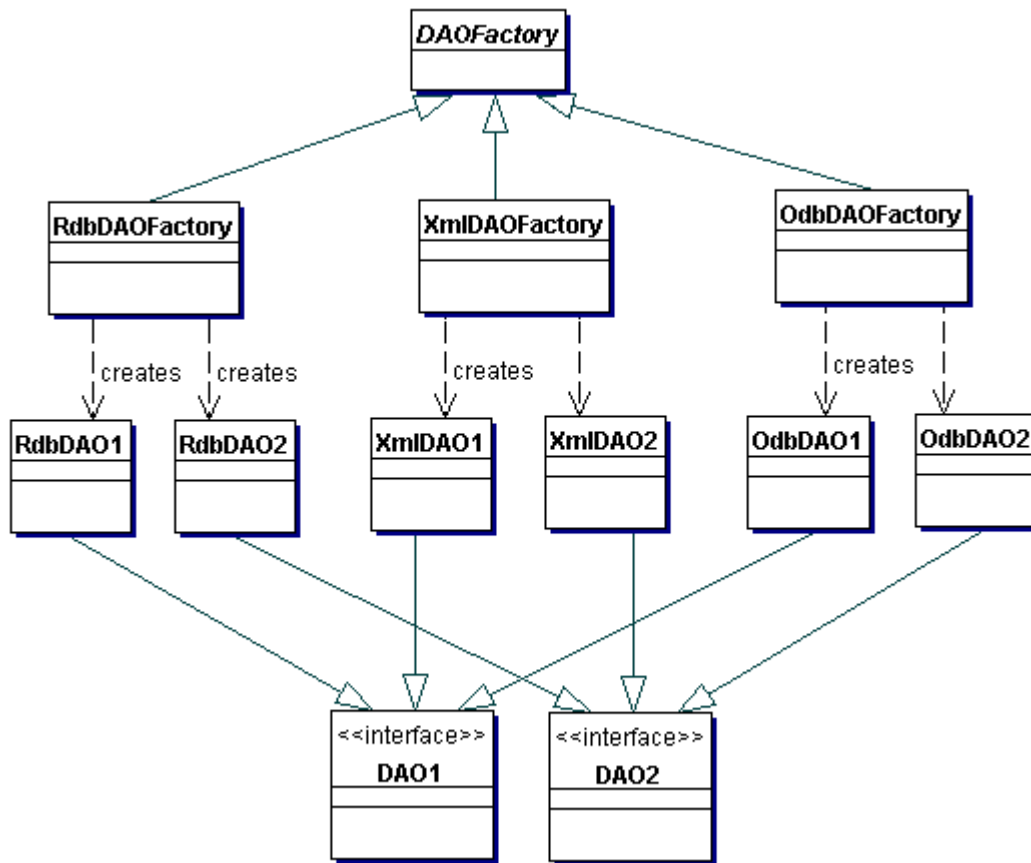
El patrón DAO se puede flexibilizar adoptando los patrones Abstract Factory y Factory Method.

Cuando el almacenamiento subyacente no está sujeto a cambios de una implementación a otra, esta estrategia se puede implementar utilizando el patrón Factory Method para producir el número de DAOs que necesita la aplicación. En la siguiente figura podemos ver el diagrama de clases para este caso:

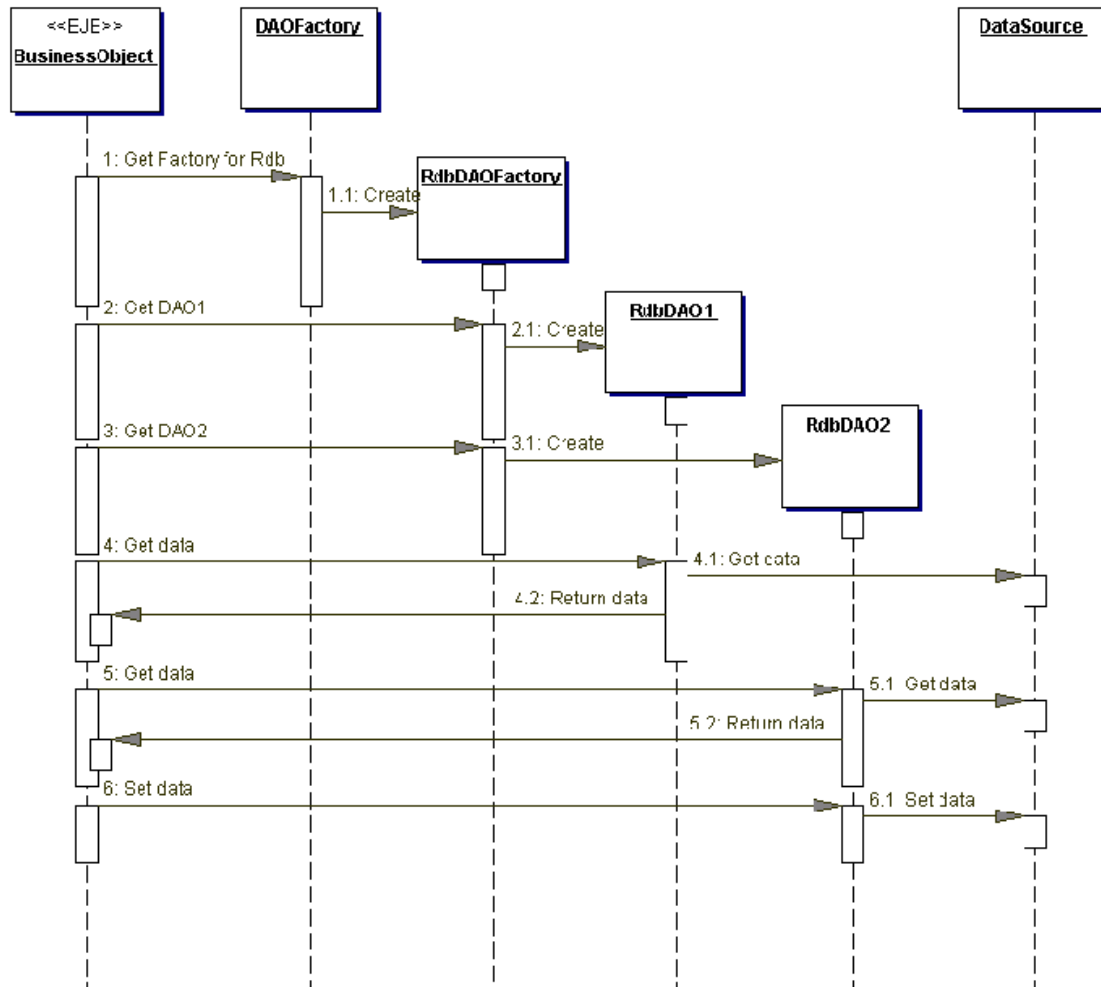
Cuando el almacenamiento subyacente si está sujeto a cambios de una implementación a otra, esta estrategia se podría implementar usando el patrón Abstract Factory. Este patrón a su vez puede construir y utilizar la implementación Factory Method. En este caso, esta estrategia proporciona un objeto factoría abstracta de DAOs (Abstract Factory) que puede construir varios tipos de factorías concretas de DAOs, cada factoría soporta un tipo diferente de implementación del almacenamiento persistente. Una vez que obtenemos la factoría concreta de DAOs para una implementación específica, la utilizamos para producir los DAOs soportados e implementados en esa implementación.

En la siguiente figura podemos ver el diagrama de clases para esta estrategia. En él vemos una factoría base de DAOs, que es una clase abstracta que extienden e implementan las diferentes factorías concretas de DAOs para soportar el acceso

específico a la implementación del almacenamiento. El cliente puede obtener una implementación de la factoría concreta del DAO como una *RdbDAOFactory* y utilizarla para obtener los DAOs concretos que funcionan en la implementación del almacenamiento. Por ejemplo, el cliente puede obtener una *RdbDAOFactory* y utilizarlas para obtener DAOs específicos como *RdbCustomerDAO*, *RdbAccountDAO*, etc. Los DAOs pueden extender e implementar una clase base genérica (mostradas como *DAO1* y *DAO2*) que describa específicamente los requerimientos del DAO para el objeto de negocio que soporta. Cada DAO concreto es responsable de conectar con la fuente de datos y de obtener y manipular los datos para el objeto de negocio que soporta.



En la siguiente figura se puede ver el diagrama de secuencia de la ejecución para el diagrama de clases mostrado:



EJEMPLO DE USO DE DAO

El DAO permite que se pueda generar fácilmente las clases necesarias para tener una capa de acceso a datos robusta y una estrategia generalizada para toda la aplicación.

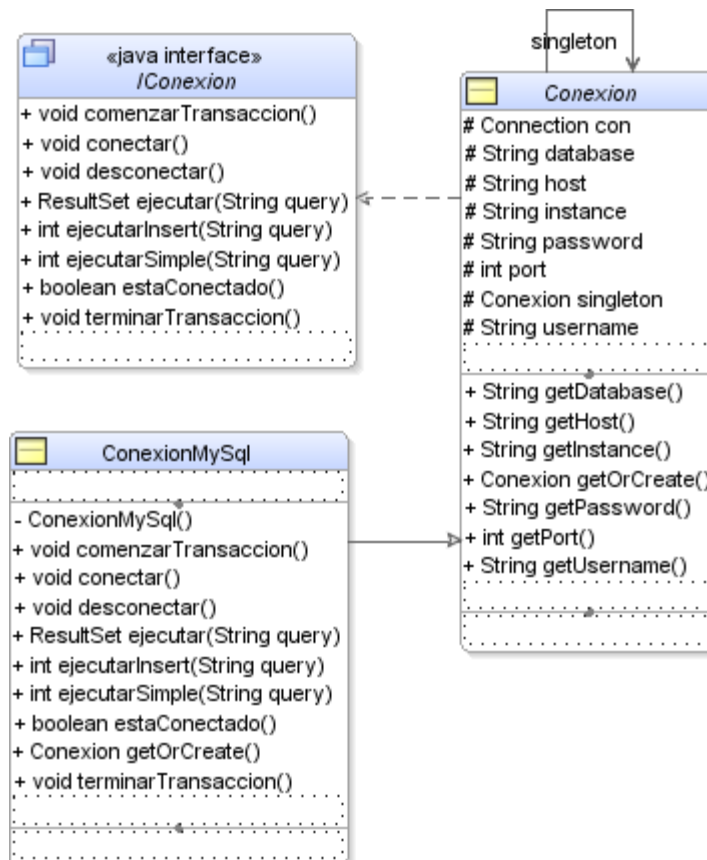
A continuación un ejemplo completo del uso del patrón DAO en Java. El ejemplo está hecho con una tabla y una base de datos MySQL. La tabla contiene atributos de solamente dos tipos: entero y string. Se puede realizar el patrón para atributos de cualquier tipo, el único cuidado hay que tenerlo con atributos de tipo Date que se manejan siempre de manera particular dependiendo de cada lenguaje.

TABLA EN BASE DE DATOS

La tabla Alumno en MySQL tiene las siguientes columnas: id (int), nombre (varchar), edad (int)

CONEXIÓN CON LA BASE DE DATOS

La conexión con la base de datos debe ser lo más genérica posible para que se pueda utilizar rápidamente en otra base de datos o en otro tipo de dispositivo persistente.

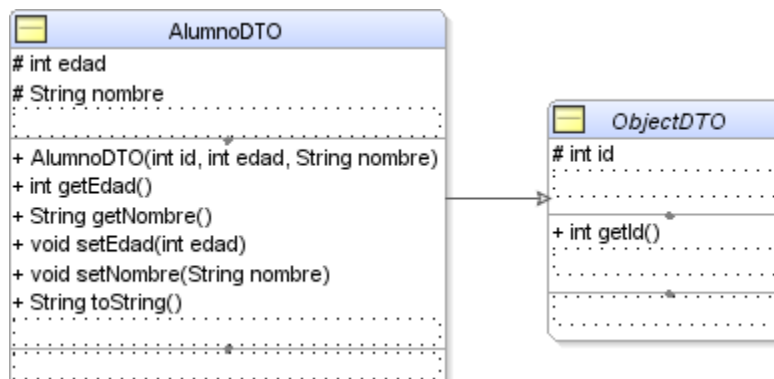


Utilizamos la interface `IConexion` para que se definan todos los métodos que se deben implementar. La clase abstracta `Conexion` es un Singleton que permite escoger mediante configuración la instancia única de conexión para la aplicación.

Finalmente la conexión a MySQL tiene la implementación de todo y podemos utilizarla para conectarnos a la base de datos y enviarle comandos de ejecución de SQLs.

OBJETO DE TRANSFERENCIA DE DATOS

Lo primero que hay que entender es la propuesta de diagrama de clases que debe tener la aplicación. Llamaremos DTO a la estructura que se tiene para albergar los objetos que representan un registro en una tabla. Se puede ver que utilizamos una superclase `ObjectDTO` que representa a cualquier registro de cualquier tabla. Todos tienen en común un ID.

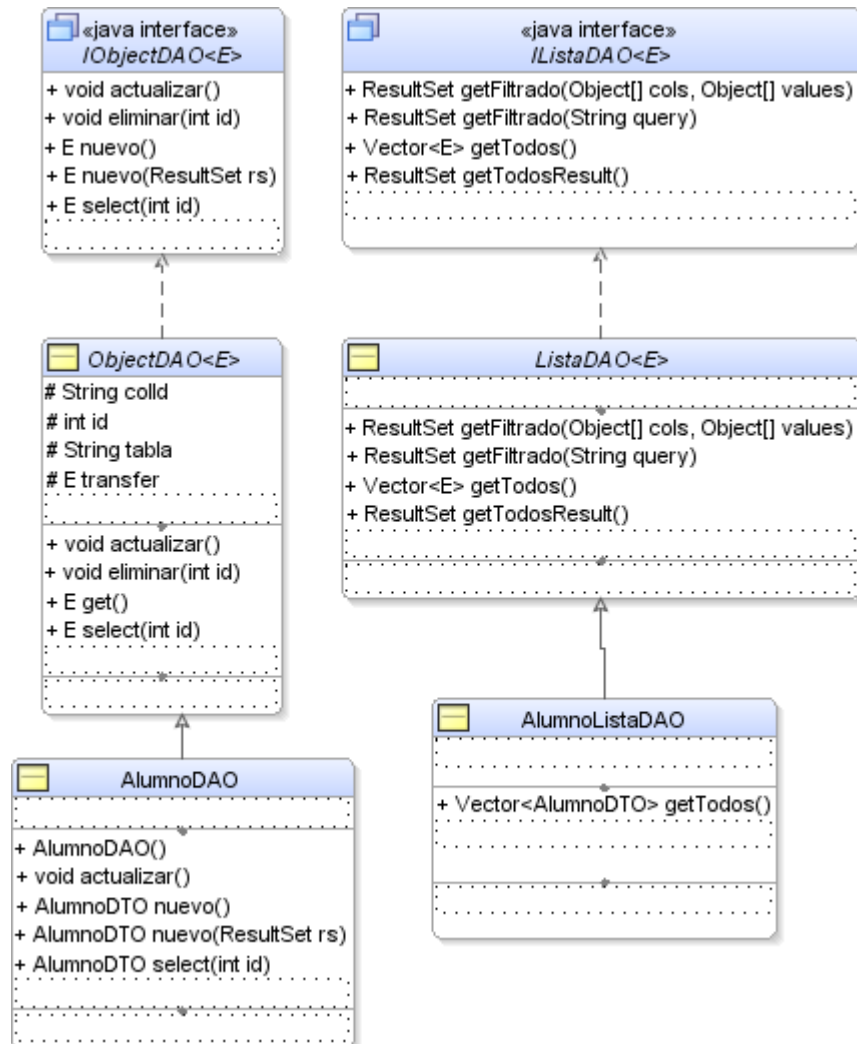


No se debe colocar ninguna otra lógica de negocios en esta capa.

OBJETO DE ACCESO A DATOS

Este objeto lo que hace es encapsular todas las operaciones de alta baja y modificación para los registros de la tabla Alumno. Este objeto tiene una relación uno a uno con un objeto de tipo DTO.

Para diferenciar esta relación uno a uno con una relación de uno a muchos (en el caso que quisiéramos obtener una lista de registros) es que se tiene la clase con el nombre Lista que hace una tarea similar.



COMO SE UTILIZA?

Vamos a ver diferentes ejemplos con su respectivo código fuente.

Traer el alumno con id = 4

```
AlumnoDAO dao = new AlumnoDAO();
AlumnoDTO dto = dao.select(4);
```

Crear un nuevo alumno "Paco" con edad 24

```
AlumnoDTO dao = new AlumnoDAO();
AlumnoDTO dto = dao.nuevo();
dto.setEdad(24);
dto.setNombre("Paco");
dao.actualizar();
```

Actualizar el nombre a 'Hugo' del alumno con id 2

```
AlumnoDAO dao = new AlumnoDAO();
AlumnoDTO dto = dao.select(2);
dto.setNombre("Hugo");
dao.actualizar();
```

Obtener la lista de alumnos

```
AlumnoListaDAO dao = new AlumnoListaDAO();
Vector<AlumnoDTO> lista = dao.getTodos();
```

VENTAJAS DEL PATRÓN DE DISEÑO DAO

Las ventajas de este modelo son las siguientes:

- Se tiene una forma estándar de acceder a las tablas de la base de datos de cualquier sistema.
- La base de datos puede cambiar y en el sistema solamente se hace la librería para esa base de datos y se cambia la configuración.
- Los programadores tienen menos problema para adaptarse al modelo ya que todos hacen lo mismo.
- Generar la capa de acceso a datos es algo tan operativo que puede ser automatizado fácilmente.
- El modelo sigue el patrón de diseño DAO en conjunto con otros patrones de diseño como Abstract Factory y Singleton.

PERSISTENCIA MODERNA

El patrón de diseño DAO se ha utilizado satisfactoriamente desde hace al menos 15 años y la idea desde hace mucho más tiempo. Sin embargo, siempre hay el problema del mapeo a clases que no queda muy elegante.

Hace ya casi 7 años se viene desarrollando una tecnología que permite una interacción un poco más natural con la base de datos. Esta tecnología sin embargo ha exigido que se cambien alguna terminología del lenguaje e inclusive la sintaxis y gramática del mismo.

En los siguientes párrafos veremos los principios básicos de esta tecnología aplicados en Java y las implicaciones que representan estos cambios. Al final debemos hacer un balance de esta tecnología contra la del patrón de diseño DAO.

PREGUNTAS

Pregunta 21. Formato para imagen

Proponga un formato para guardar una imagen. Escriba el código para guardar y leer una imagen en este formato.

Pregunta 22. Patrón Decorator

Escribir un programa que permita crear los siguientes productos: café, café con leche, chocolate, y las decoraciones, con crema, doble.

ÚTILES DE PROGRAMACIÓN

En esta sección, que es un anexo a este texto, se verá algunos útiles de programación que deberían permitir elevar sustancialmente la calidad del programador. Ojo, esto no hace programadores que hacen algoritmos más rebuscados y potencialmente más interesantes. Estos útiles de programación lo que hacen es elevar la calidad de la programación profesionalizando lo que el programador ya sabe realizar.

REGISTRO DE EVENTOS

Quizá recuerdes la última vez que hiciste un código similar al siguiente:

```
public void metodo() {  
    LeerParametros();  
    System.out.println("Lee los parámetros desde consola");  
    int n = CantidadParametros();  
    System.out.println("Hay " + n + " parámetros");  
    ...  
}
```

Este código trata de comunicar al programador lo que está ocurriendo en el método en cada instrucción que pasa. Esto suele ocurrir cuando surgen problemas y comenzamos a ver donde podría estar el problema. Lo más normal es que, cuando ya estamos terminando el programa, eliminemos o comentemos algunas o todas estas líneas de código. Esta es una pésima práctica a la que nunca más debe recurrir ningún programador.

¿Qué opciones tenemos?

1. Cuando hay un bug pues entonces se utiliza el debugger y se analiza línea por línea lo que esta pasando.
2. Es normal que uno quiera registrar los eventos a medida que van pasando, pero esto debe hacerse de otra manera, con una librería de log.

LOG4J

Insertar código de registro de eventos en tus programas es un método arcaico que se utiliza para deboguear. Lastimosamente, puede ser que sea el único método disponible dependiendo del ambiente en que esta el ejecutable o programa. Este es el caso para aplicaciones ya desplegadas en el cliente.

Por otro lado, alguna gente argumenta que las sentencias de registro de eventos estorban la legibilidad del mismo (nosotros pensamos que lo contrario es lo correcto).

Con Log4j es posible habilitar el registro de eventos en una aplicación sin modificar el ejecutable. EL paquete está diseñado para que estas sentencias estén en el código sin gran costo de la performance. El comportamiento de la librería puede ser controlado por un archivo de configuración, sin tocar el código ejecutable. Se puede acceder al último compilado aquí:

<http://logging.apache.org/log4j/1.2/download.html>

INSTALACION

Típicamente se puede colocar todo el contenido del ZIP en una carpeta de fácil acceso, por ejemplo c:\log4j. Puede utilizar cualquier otra carpeta.

Ubique el archivo **log4j-1.2.15.jar** ya que es el único que es importante. Para el caso haremos referencia al mismo entonces con **c:\log4j\log4j-1.2.15.jar**.

USO EN PROGRAMAS JAVA

En su IDE de preferencia, en el proyecto actual, indicar en las preferencias que desea utilizar la librería log4j (apuntando al JAR que acaba de descomprimir)

Crear el archivo **auditoria.properties** para la configuración de log4j para este proyecto. El contenido del archivo debe ser:

```
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender which outputs to System.out.
log4j.appender.A1=org.apache.log4j.RollingFileAppender
log4j.appender.A1.File=Logs/RollingLog.log
log4j.appender.A1.MaximumFileSize=10000
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout

# The conversion pattern uses format specifiers. You might want to
# change the pattern and watch the output format change.
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5l - %m%n
# %-4r %-5p [%t] %37c %3x - %m%n
```

En el archivo que contiene el método main() de la aplicación incluir las siguientes líneas (de preferencia al principio de la ejecución del programa):

```
public static void main(String[] args) {
    String resource =
        "/auditoria.properties";
    URL configFileResource;
    configFileResource = MiPrograma.class.getResource(resource);
    PropertyConfigurator.configure(configFileResource);
}
```

En todos los lugares donde se desee registrar un evento se debe incluir, al principio de la clase, el objeto log que se utilizara para hacer registro de eventos y por supuesto las líneas de llamado a este objeto. Aquí se tiene una clase de ejemplo:

```
public class MiObjeto {

    private static Logger logger = Logger.getLogger(MiObjeto.class);

    public void metodo() {
        logger.info("Informacion, va a dividir / 0 para que haya error");
        try {
            int a = 1 / 0;
        } catch (Exception err) {
            logger.error("Salta el error y se lo ve aqui en el log:", err);
        }
    }
}
```

```
}
```

RESULTADOS

Si se ejecuta el programa no pasa absolutamente nada a nivel de interface pero un archivo de logs va a ser creado en la carpeta logs de su proyecto con información similar a la siguiente:

```
2010-02-08 03:16:43,951 [main] ejemplolog4j.MiObjeto.metodo(MiObjeto.java:13) - Este
es un mensaje de informacion, va a dividir / 0 para que haya error
2010-02-08 03:16:43,956 [main] ejemplolog4j.MiObjeto.metodo(MiObjeto.java:17) - Salta
el error y se lo ve aqui en el log:
java.lang.ArithmeticException: / by zero
    at ejemplolog4j.MiObjeto.metodo(MiObjeto.java:15)
    at ejemplolog4j.MiPrograma.main(MiPrograma.java:17)
```

DOCUMENTACION Y JAVADOC

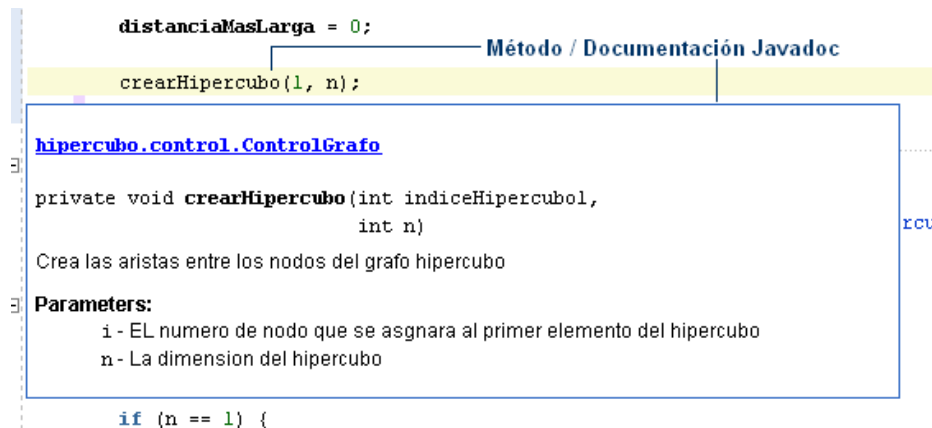
Durante el desarrollo, es imprescindible que el desarrollador se esfuerce en que su código sea legible para el mismo y para cualquier otra persona que lo analice posteriormente.

Esto implica llevar a cabo buenas prácticas de escritura de código. Pero además, obliga al desarrollador a una disciplina en la escritura de los comentarios de las clases y métodos que escribe. Específicamente, al escribir un nuevo método o clase para un programa que va a ser utilizado o presentado se DEBE realizar lo siguiente:

Una vez escrita la firma del campo, método o la clase, escribir el comentario del mismo.

- Si se escribe el comentario de una clase, describir de manera general el objeto que se va a formar al instanciar esa clase o el propósito de la misma si es una clase abstracta o interface.
 - Luego escribir el código de la clase
- Si se escribe el comentario de un método, escribir de manera general lo que hace (si es que el título del método no es suficiente) y describir el algoritmo de lo que se pretende realizar.
 - Luego escribir el código del método
- Si se escribe el comentario de un campo de la clase, escribir la descripción de lo que representa el campo y, de ser necesario, los valores permitidos para el mismo

Cualquier IDE hoy en día reconoce estos comentarios y los incorpora a la ayuda que sale en el texto cuando se escribe código. Aquí se puede ver un ejemplo:



Por ejemplo, para poder tener la documentación de ese método en ese formato, lo único que se debe hacer es colocar el comentario correcto en el método. Aquí el ejemplo:

```

/**
 * Crea las aristas entre los nodos del grafo hipercubo
 * @param i EL numero de nodo que se asgnara al primer elemento del hipercubo
 * @param n La dimension del hipercubo
 */
private void crearHiperCubo(int indiceHiperCubo1, int n) {
    ...
}

```

En realidad, dentro del comentario se puede colocar código HTML que será interpretado por el visualizador de Javadoc que se utilice (dependiendo de su IDE).

Los comandos más utilizados que puede usar dentro de los comentarios son:

```
* @author      (classes and interfaces only, required)
```

Donde se indica el nombre del autor, solamente para las cabeceras de las clases y las interfaces.

```
* @version      (classes and interfaces only, required)
```

Donde se coloca el número de versión de la clase o interface.

```
* @param      (methods and constructors only)
```

Solamente para los métodos y constructores, es el prefijo para indicar el comentario para un parámetro de la función.

```
* @return      (methods only)
```

Es el prefijo para indicar el comentario para el valor de retorno de la función.

```
* @exception    (@throws is a synonym added in Javadoc 1.2)
```

Indica las excepciones que pueden ocurrir al llamar a este método o constructor.

Finalmente, también se puede generar la documentación del código con cualquier IDE que utilice. Solamente debe indicar al IDE que genere la documentación y el IDE generará los archivos HTML que contienen toda la documentación de su proyecto o librería.

En el caso de las clases de Java, su documentación ha sido generada también con Javadoc y se puede ver la misma en línea aquí http://download.oracle.com/docs/cd/E17409_01/javase/7/docs/api/

BUENAS PRÁCTICAS PARA NUESTRO CÓDIGO

En lo posible, todo el código producido por un desarrollador debe ser revisado por un segundo desarrollador o por el inmediato superior. La revisión supone que la persona que revise realice lo siguiente:

- Es legible: correcta indentación del código, el nombre de variables es adecuado y sigue los estándares de la aplicación
- Una correcta programación defensiva
- El uso de registros de logs para la trazabilidad de las operaciones, especialmente aquellas consideradas críticas
- La complejidad ciclomática no sobrepasa 3 en ningún caso salvo alguna documentación que así lo justifique.

LEGIBLE

A continuación algunos consejos que mejoran la lectura de código fuente (acuérdate que tratar de ver un código escrito por alguien más suele ser una pesadilla).

- Utilizar espacios para las asignaciones y otros:

```
if (n == 1) { // correcto
for(int i=1; i<10; i++) { // incorrecto
for (int i = 1; i < 10; i++) { // correcto
```

- Utilizar nombres de variables que signifiquen algo:

Por ejemplo, un código muy usado sería

```
public void suma(int n) {
    int r = 0;
    for(int i=0; i<n; i++)
        r += i;
    return r;
}
```

Utilizar resultado en vez de r. Espacios en el for.

- Nombres de métodos y variables, aunque sean largos, que sean legibles e indiquen lo que hacen
- Los nombres de botones pueden ser: btnGuardar (observe el prefijo). Los nombres de un ítem menú: mnuFile_Nuevo (observe el prefijo).
- Las funciones que se encargan de reaccionar ante un clic del botón deben ser claras. Por ejemplo, para el botón btnGuardar la función se puede llamar: btnGuardar_Clicked.

PROGRAMACIÓN DEFENSIVA

El desarrollador debe utilizar extensivamente el modelo de manejo de errores de la tecnología que desarrolle para tomar en cuenta todas las posibilidades de error en los datos. Especialmente:

- Revisará que los datos de entrada de cualquier función sean correctos (en caso de que pudiesen llegar incompletos).
- Revisará que los datos de salida de las funciones sean correctos.

Veamos un ejemplo en algo sencillo. ¿Cómo se puede verificar que un número es impar? Generalmente hacemos algo así:

```
if (n % 2 == 1)
```

Programación defensiva significa tomar en cuenta todas las posibilidades. En este caso, que pasa si el número es negativo, el resultado de $n\%2$ nos saldrá -1 en el caso de impares y habremos fallado en la programación.

LOGS

Estos ya fueron vistos en el anterior anexo.

COMPLEJIDAD CICLOMÁTICA

En el código fuente se debe tratar de evitar imbricar demasiados niveles de complejidad (bloques de código). El ser humano puede manejar hasta 3 niveles de complejidad de manera natural. Cuando se tienen más niveles la comprensión del código fuente se torna muy complicada y por lo tanto difícil de mantener. Para dar un ejemplo se pueden ver dos versiones del mismo código con complejidades diferentes.

Este código tiene complejidad ciclomatica 3:

```
public void hacer() {
    For(i=0; i<100; i++) {
        If (algo es verdad) {
            hacerUno();
            hacerDos();
            ...
            hacerN();
        }
    }
}
```

Y este, que es el mismo, tiene complejidad ciclomatica 2:

```
public void hacer() {
    For(i=0; i<100; i++) {
        If (algo NO es verdad) {
            Continue;
        }
        hacerUno();
        hacerDos();
        ...
        hacerN();
    }
}
```

Bloques Try / Catch

Luego de muchas experiencias en desarrollo de sistemas y la realización de programas uno se da cuenta de lo poderosa que es la herramienta de try / catch. Pero el objetivo real de estas herramientas solamente se puede ver si tomamos una perspectiva más de administración del desarrollo de un programa o sistema.

Capas de