# http://vectorblog.org/wp-content/uploads/2013/06/Telehealth-cartoon-e1370345782337.jpgHoja de trabajo No. 4: *Ordenamiento*

Objetivos de la hoja de trabajo

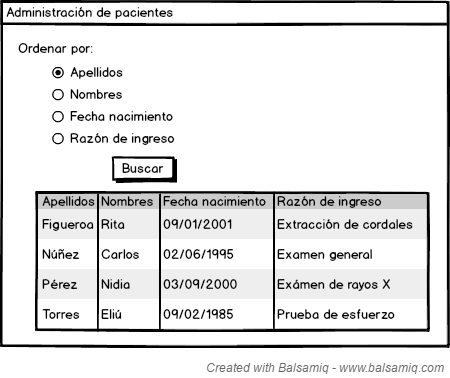
* Identificar estrategias de programación para permitir ordenamiento por varios criterios.
* Identificar ventajas y desventajas de las diferentes estrategias.
* Aplicar conceptos de implementación de interfaces (Programación orientada a objetos).
* Identificar ventajas de mantener información ordenada.

Recomendación: trabajar en grupos de dos integrantes.

Recordemos la aplicación de *Administración de pacientes*.

E:\Documentos\Dropbox\Proyectos\Uni\Curso Estructura de datos\Clases\02 TDA\analisis-diseno-programa-hospital.png

Según comentamos en la clase anterior, el usuario/cliente de este programa apreciaría la funcionalidad de ordenar la información por varios criterios. Se propone la siguiente interfaz de usuario:



Se empezó con el siguiente código para insertar ordenadamente.

|  |
| --- |
| void agregarOrdenado(elemento, int tipoOrdenamiento)  {  switch(tipoOrdenamiento)  {  case 1:  agregarOrdenadoPorApellido(elemento)  case 2:  agregarOrdenadoPorFecha(elemento)  }  }  void agregarOrdenadoPorApellido(elemento)  {  si obtenerCantidadElemento() == 0  Agregar(elemento, 0)  si no  int posicion = 0  mientras ((posicion < contador) y  (String.Compare(ObtenerElemento(posicion).apellido,  elemento.apellido) <= 0))  {  posicion++  }  agregar(elemento, posicion)  fin si  }  void agregarOrdenadoPorFecha(elemento)  {  si obtenerCantidadElemento() == 0  Agregar(elemento, 0)  si no  int posicion = 0  mientras ((posicion < contador) y  (DateTime.Compare(ObtenerElemento(posicion).fechaNacimiento,  elemento.fechaNacimiento) <= 0))  {  posicion++  }  agregar(elemento, posicion)  fin si  } |

1. Escriba un algoritmo para un método de agregarOrdenadoPorRazonIngreso.

void agregarOrdenadoPorRazonIngreso(elemento)

{

si obtenerCantidadElemento() == 0

Agregar(elemento, 0)

si no

int posicion = 0

mientras ((posicion < contador) y

((ObtenerElemento(posicion).razon.CompareTo(elemento.razon)<= 0)

{

posicion++

}

agregar(elemento, posicion)

fin si

Luego se planteó el siguiente código:

|  |
| --- |
| void agregarOrdenado(elemento, int tipoOrdenamiento)  {  si obtenerCantidadElemento() == 0  Agregar(elemento, 0)  si no  int posicion = 0  mientras ((posicion < contador) y  (CompararPaciente(ObtenerElemento(posicion), elemento,  tipoOrdenamiento) <= 0))  {  posicion++  }  agregar(elemento, posicion)  fin si  }  int CompararPaciente(CPaciente paciente1, CPaciente paciente2,  int tipoComparacion)  {  int resultado = 0;  switch(tipoComparacion)  {  case 1:  resultado = String.Compare(paciente1.apellido,  paciente2.apellido)  case 2:  resultado = DateTime.Compare(paciente1.fechaNacimiento,  paciente2.fechaNacimiento)  }  } |

1. ¿Qué ventajas y desventajas identifica de ambos códigos? ¿Cuál recomendaría usted?

Vemos que ahora se han distribuido mejor las responsabilidades entre los métodos, teniendo CompararPaciente para comparar pacientes y AgregarOrdenado para agregar en la posición correcta. Me gusta más el segundo ya que pienso que es más ordenado y limpio, aparte, podríamos necesitar luego el método CompararPaciente para algo más que agregar un nuevo paciente y ya lo tenemos programado. Como desventaja vemos que el método agregarOrdenado sigue estando ligado únicamente a objetos de tipo paciente.

1. Recordemos que estamos trabajando con una lista ordenada (CListaOrdenada), genérica para elementos de tipo T. ¿Cómo afecta que esta lista tenga un método (CompararPaciente) que recibe objetos de tipo específico CPaciente?

Pues ahora la hemos limitado y básicamente daría lo mismo tener una lista no genérica.

¿Y la programación orientada a objetos qué? ¿Qué opina si extraemos el método CompararPaciente y lo colocamos en una clase aparte? Ver siguiente diagrama UML de clases.

E:\Documentos\Dropbox\Proyectos\Uni\Curso Estructura de datos\Clases\06 Ordenamiento\2017-02-21 - Clases ordenamiento.png

1. ¿Cómo modificaría el método agregarOrdenado de CListaOrdenada?

Pues creo que mis pensamientos irian alineados con la propuesta que se muestra. Únicamente pienso que en vez de instanciar un nuevo comparador cada vez que se agrega, podríamos tener uno solo y modificar su propiedad tipoComparación. Tendríamos mas memoria utilizada, pero sería un poquito más eficiente el programa.

A continuación una propuesta:

|  |
| --- |
| void agregarOrdenado(elemento, int tipoOrdenamiento)  {  si obtenerCantidadElemento() == 0  Agregar(elemento, 0)  si no  int posicion = 0  CComparadorPaciente comparador =  new CComparadorPaciente(tipoOrdenamiento)  mientras ((posicion < contador) y  (comparador.CompararPaciente(  ObtenerElemento(posicion), elemento) <= 0))  {  posicion++  }  agregar(elemento, posicion)  fin si  } |

1. ¿Necesitamos todavía un método privado “CompararPaciente” en la clase CListaOrdenada? ¿Cree que hubo una mejora en la clase CListaOrdenada?

Pues de nuevo vemos que se han distribuido mejor las responsabilidades entre las clases, pero seguimos ligados a objetos Cpaciente.

Tome un tiempo para discutir sus respuestas con otro grupo.

¿Cree que se puede mejorar? ¿Qué opina del siguiente código?

|  |
| --- |
| /\* Para llamar al método agregarOrdenado de CListaOrdenada. Por ejemplo este código puede estar en AdminHospital \*/  CComparadorPaciente comparador =  new CComparadorPaciente(tipoOrdenamiento)  agregarOrdenado(nuevoPaciente, comparador)  /\* Método de CListaOrdenada \*/  void agregarOrdenado(elemento, CComparadorPaciente comparador)  {  si obtenerCantidadElemento() == 0  Agregar(elemento, 0)  si no  int posicion = 0  mientras ((posicion < contador) y  (comparador.CompararPaciente(  ObtenerElemento(posicion), elemento) <= 0))  {  posicion++  }  agregar(elemento, posicion)  fin si  } |

1. ¿Cuál es la diferencia? ¿Cree que hay alguna ventaja?

De nuevo vemos que el método AgregarOrdenado necesita un ComparadorPaciente para funcionar.

¿Y dónde están los promotores de la herencia? Observe el siguiente diagrama.

C:\Users\aovalle\Downloads\2017-02-21 - Clases ordenamiento 2.png

1. ¿Cree que todavía es necesario manejar el tipoComparacion? Escriba los métodos “comparar” de cada clase: CComparadorPaciente, CComparadorApellidos, CcomparadorFechaNacimiento.

**Clase CcomparadorPaciente // manera default de comparar paciente, por nombres**

int comparar(CPaciente p1, Cpaciente p2)

{

return p1.nombre.CompareTo(p2.nombre);

}

**Clase CcomparadorApellidos**

int comparar(CPaciente p1, Cpaciente p2)

{

return p1.apellidos.CompareTo(p2.apellidos);

}

**Clase CcomparadorFechaNacimiento**

int comparar(CPaciente p1, Cpaciente p2)

{

return p1.fechaNacimiento.CompareTo(p2.fechaNacimiento);

}

1. ¿Qué cambios haría en el método AgregarOrdenado de CListaOrdenada?

Probablemente utilizaría un switch para instanciar un CcomparadorApellidos o un CcomparadorFechaNacimiento, dependiendo del tipo de ordenamiento deseado.

1. ¿Cree que todavía es desventaja que CListaOrdenada esté relacionada con un tipo de dato específico como CComparadorPaciente?

Claro que si.

Tome un tiempo para discutir sus respuestas con otro grupo.

1. Investigue sobre la interfaz de .NET IComparable. ¿Qué relación existe entre IComparable y CPaciente?

Icomparable permite que dos objetos del mismo tipo se comparen, y la comparación entre pacientes parece una operación natural a la hora de querer ordenarlos.

1. ¿Qué ventajas hay de utilizar la interfaz IComparable?

Me imagino que hay muchas más de las que se, pero por ejemplo, tan solo tenemos que implementar un método, CompareTo(), y ya podríamos comparar paciente y ordenar una lista.

Tome un tiempo para discutir sus respuestas con otro grupo.

1. Investigue sobre la búsqueda secuencial y búsqueda binaria.

* La búsqueda secuencial consiste en iterar por toda la lista, buscando si el elemento que deseamos existe. Complejidad O(n)
* La búsqueda binaria reduce el tiempo de búsqueda, pero la lista debe estar previamente ordenada. Complejidad O(log n)

1. En la aplicación de Administrador de pacientes, ¿qué criterios incluiría para hacer búsquedas?

Hemos incluido búsqueda por nombre, apellidos, dpi, y tipo de sangre.

1. ¿En qué podría ayudar tener la información ordenada?

Podríamos utilizar el algoritmo de búsqueda binaria para reducir considerablemente el tiempo de búsqueda de un paciente. También al ordenar la lista, nos sirve para poder mostrársela al usuario, dependiendo del criterio de ordenamiento que el desee.