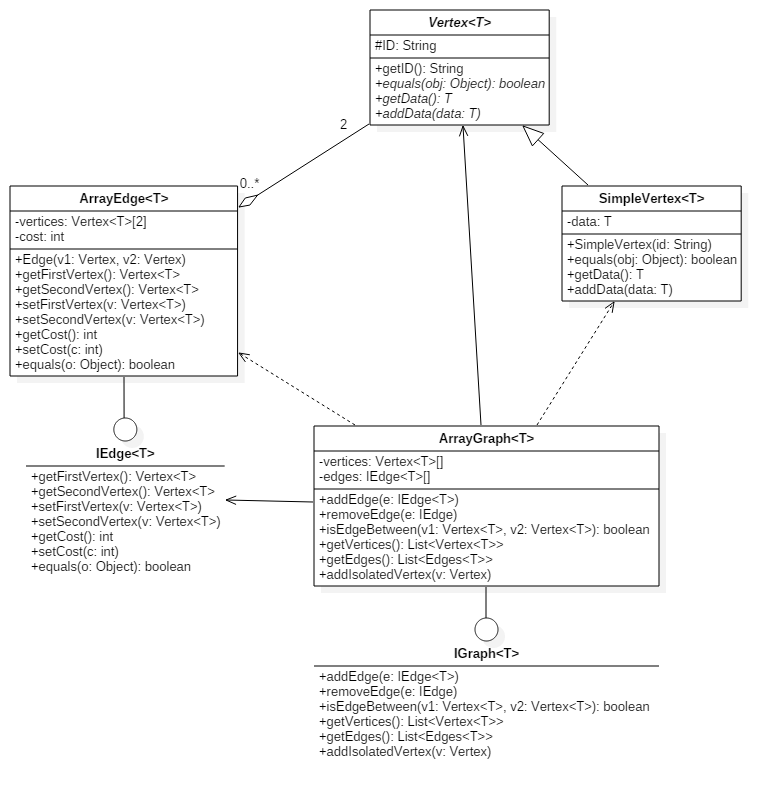
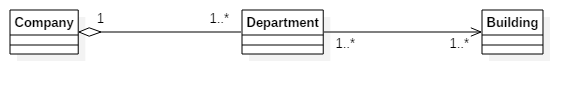
**P1.** Prepare a class model to describe undirected graphs. An undirected graph consists of a set of vertices and a set of edges. Edges connect pairs of vertices. Your model should capture only the structure of graphs (i.e. connectivity).

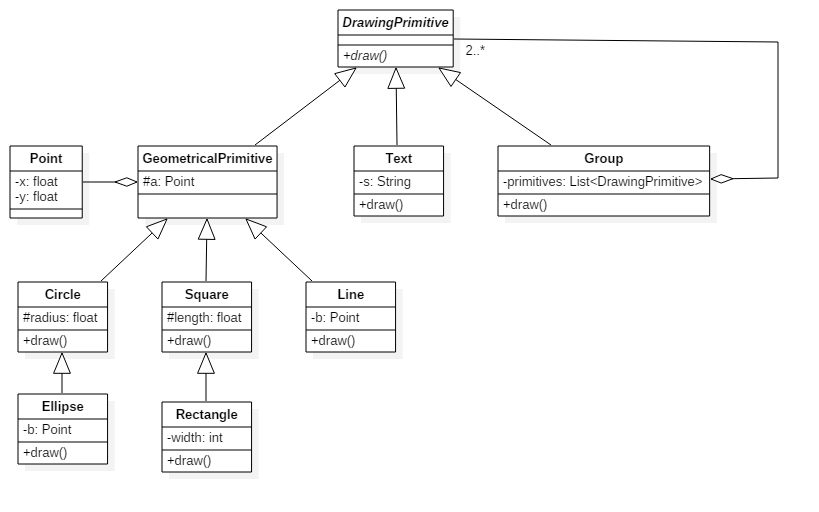


Clasa abstracta Vertex<T> reprezinta un nod un grafului, care este identificata printr-un ID. Un nod al grafului poate stoca o data de tipul generic T. Clasa SimpleVertex<T> mosteneste si implementeaza metodele abstracte ale clasei Vertex<T>. Interfata IEdge<T> descrie comportamentul unei muchii. Clasa ArrayEdge<T> implementeaza interfata IEdge<T> continand un vector de 2 elemente de tipul Vertex<T>. Muchiilor I se poate asocial si un cost daca este cazul (implicit este 0). Interfata IGraph<T> contine metodele principale ale unui graf, si anume adaugarea/stergerea unei muchii, adaugarea unui varf izolat, verificarea existentei unei muchii si intoarcerea muchiilor si a varfurilor care alcatuiesc graful. Clasa ArrayGraph<T> implementeaza interfata IGraph<T> cu ajutorul unor tablouri.

**P2.** Consider the following specification: *“A company consists of several departments. Each department is located in one or more buildings. ”*. Draw a class diagram to model the concepts above (no attributes and operations, just classes and the relationships between them including multiplicities).



**P3.** Prepare a class diagram for a graphical document editor that supports grouping. Assume that a document consists of several sheets. Each sheet contains drawing objects, including text, geometrical objects, and groups. A group is simply a set of drawing objects, possibly including other groups. A group must contain at least two drawing objects. A drawing object can be a direct member of at most one group. Geometrical objects include circles, ellipses, rectangles, lines and squares.

Clasa abstracta DrawingPrimitive contine o singura metoda: draw. Aceasta metoda este folosita pentru afisarea primitive grafice pe ecran. Clasele GeometricalPrimitive, Text si Group mostenesc clasa DrawingPrimitive.

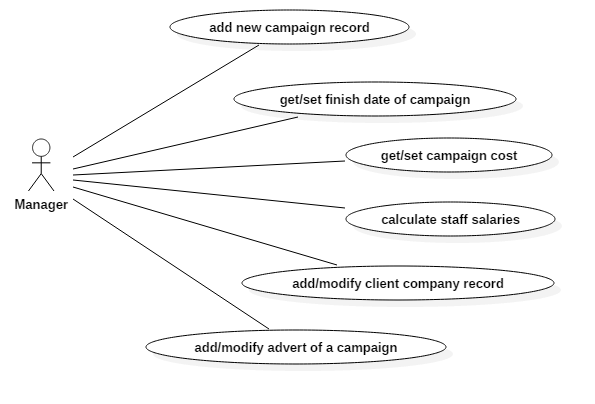
Clasa GeometricalPrimitive este abstracta, unind singurul atribut al primitivelor grafice geometrice: punctul sub forma unei variabile instant de tipul Point. Clasa Circle mosteneste clasa abstracta GeometricalPrimitive si implementeaza metoda draw, este extinsa de clasa Ellipse care la randul ei adauga inca un punct ca atribut. Clasa Square adauga la GeometricalPrimitive lungimea si implementeaza metoda draw, iar clasa Rectangle este si mai specifica adaugand si latimea. Clasa Line adauga la GeometricalPrimitive inca un punct si implementeaza metoda draw.

Clasa Text contine un singur atribut (un sir de caractere). In cazul clasei Group am folosit sablonul de proiectare Composite: Group are o lista de obiecte DrawingPrimitive (superclasa ei).

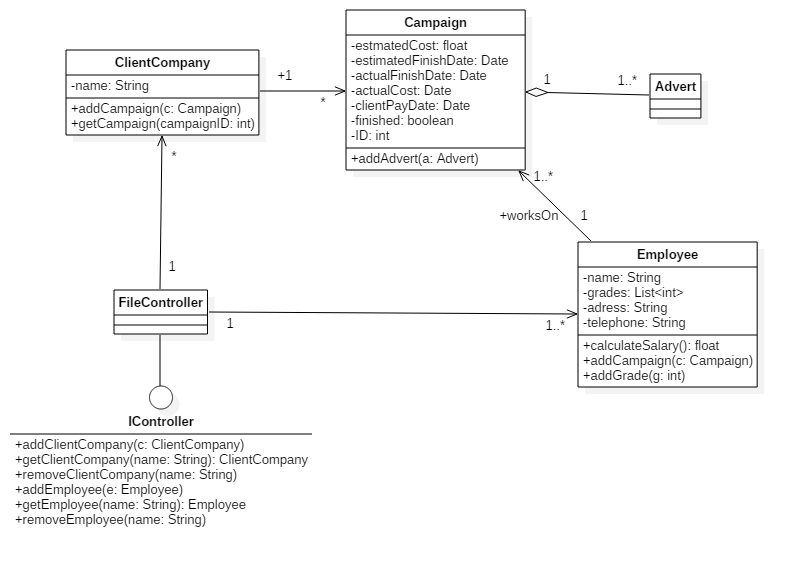
**P4.** A\* is an advertising company in New York. A\* deals with other companies that it calls clients. A record is kept of each client company. Clients have advertising campaigns, and a record is kept of every campaign. Each campaign includes one or more adverts. A\* nominates members of creative team, which work on campaigns. One member of the creative team manages each campaign. Staff may be working on more than one project at a time. When a campaign starts, the manager responsible estimates the likely cost of the client and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. When the campaign is completed, an actual completion date and the actual cost are recorded. When the client pays, the date is recorded. The manager checks the campaign budget periodically. The system should also hold the staff grades, keep a record of the contact information for each staff member, and should calculate staff salaries.

**Requirements**

1. Draw a UML use case diagram for the A\* information system.

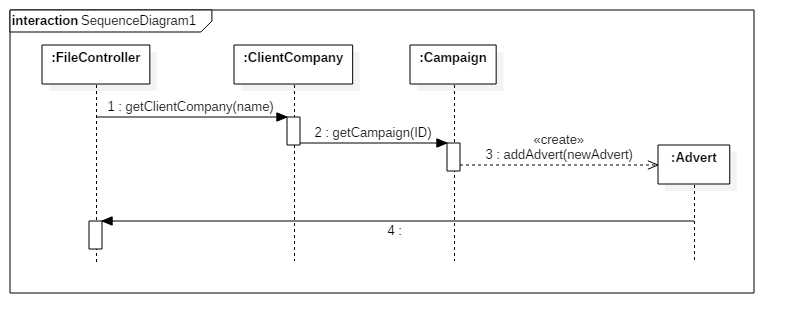


1. Draw a UML class diagram. The class diagram should represent all of the classes, their attributes and operations, relationships between classes, multiplicity specifications.



Interfata IController contine metodele principale prin care se poate realiza management-ul companiei A\*. Implementarea sa pe numele FileController foloseste serializare pentru a stoca informatiile. Se foloseste o interfata pentru a putea extinde cu usurinta metoda de stocare a datelor (de exemplu variant cu baze de date).

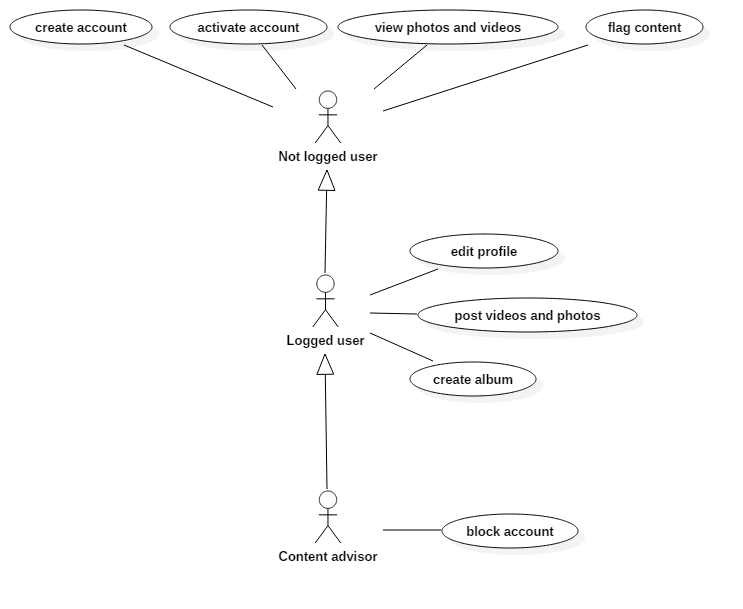
1. Draw a UML sequence diagram for the use case *Add Advertisement*.



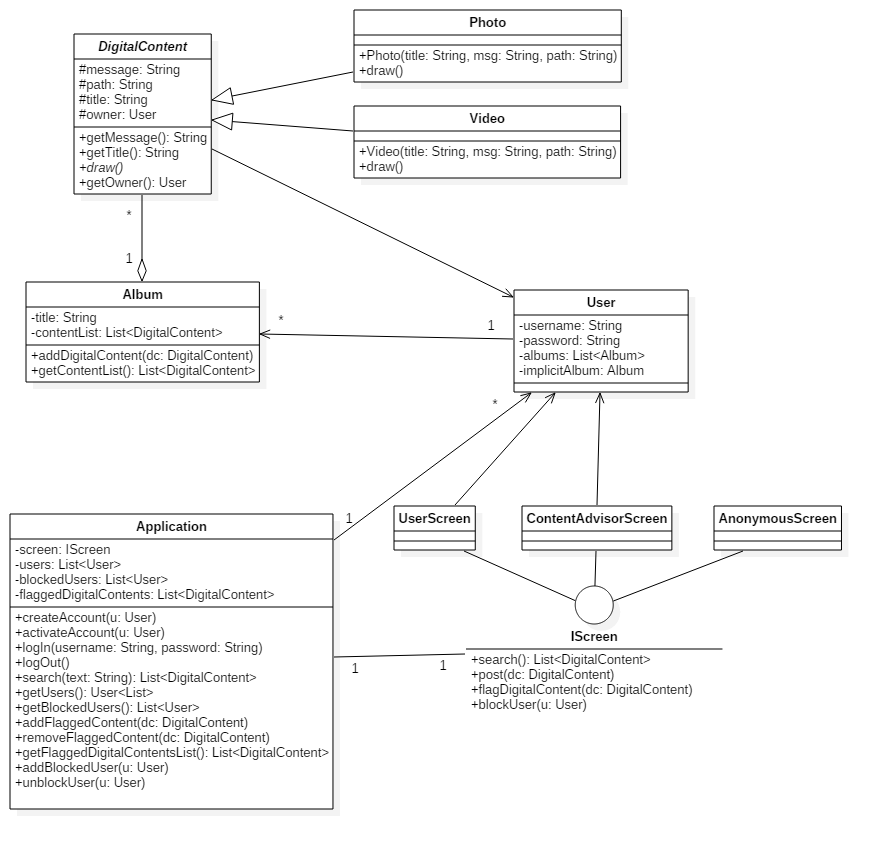
**P5**. Design a photo/video sharing Web application. The application should allow a potential new user to create an account, activate the account via an email link or view photos and videos without being logged in. Login would allow him to edit his profile, post videos and post photos either stand-alone or in an album he creates. Also, upon viewing a photo or a video, anyone, logged in or not, can flag the content for questionable content. A content advisor should review each flagged item once logged in, then decide on whether the content is acceptable for the scope of the Web application. If the content is inappropriate, the content advisor can block the owning user’s account.

**Requirements**

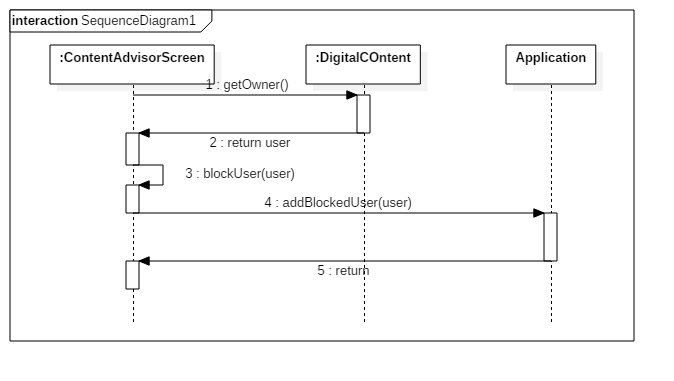
1. Draw a UML use case diagram for the photo/video sharing Web application.



1. Draw a UML class diagram. The class diagram should represent all of the classes, their attributes and operations, relationships between classes, multiplicity specifications.



1. Draw a UML sequence diagram for the use case *Block User*.



**P6.** Draw an UML diagram for the following C++ segment of code:

class Person

{

public:

Person(const char\* aName);

void speak();

void drive();

private:

char\* name; // Persons Name

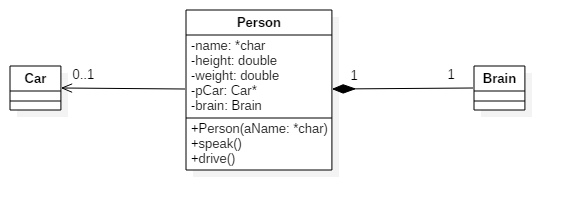
Brain brain; // persons brain

Car\* pCar; // Car owned by the person

double height;

double weight;

};



**P7.** Given the following code:

class Memory {…} // Assume a copy constructor is provided for this class.

class Memory1 extends Memory {…} // Assume a copy constructor is provided.

class Computer

{

private Memory theMemory;

public Computer(Computer another)

{

if (another.theMemory instanceof Memory1)

theMemory = new Memory1(another.theMemory);

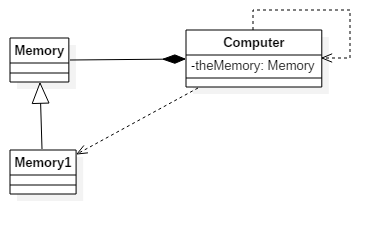
else

theMemory = new Memory(another.theMemory);

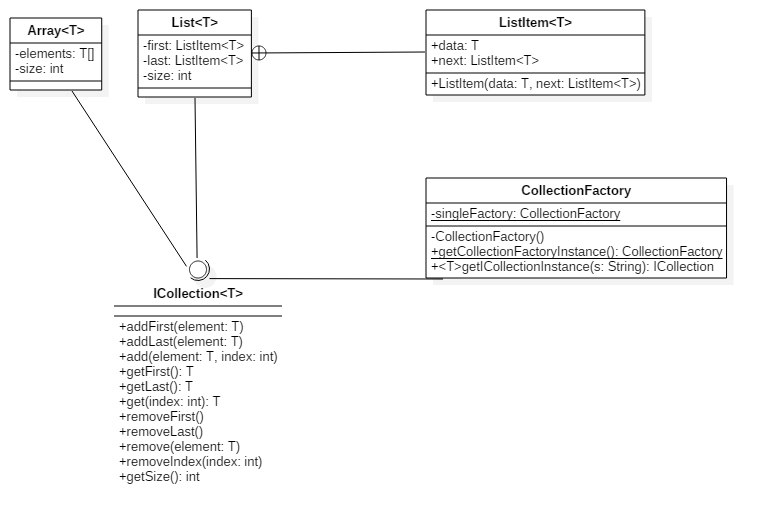
}

}

Draw a UML class diagram showing the relationship between these classes.



**P8.** Choose a design pattern studied in previous courses. Think of a problem in which it could be applied, draw the associated class diagram and write the associated source code.



In cazul acestei probleme am folosit design pattern-urile Factory Method si Singleton.

Interfata ICollection<T> este interfata care contine metodele necesare unui obiect care stocheaza alte obiecte intr-o anumita ordine (avand indecsi). Are doua implementari: Array<T> si List<T>. Clasa List<T> contine o clasa private imbricate folosita pentru reprezentarea unui nod al listei (ListItem<T>). Clasa CollectionFactory este cea care furnizeaza obiecte de tipul ICollection<T> conform unui parametru String. Deasemenea, clasa CollectionFactory este Singleton, exista numai un obiect de acest tip.

**public** **interface** ICollection<T> {

**public** **void** addFirst(T element);

**public** **void** addLast(T element);

**public** **void** add(T element, **int** index);

**public** T getFirst();

**public** T getLast();

**public** T get(**int** index);

**public** **void** removeFirst();

**public** **void** removeLast();

**public** **void** remove(T element);

**public** **void** removeIndex(**int** index);

**public** **int** getSize();

}

**public** **class** Array<T> **implements** ICollection<T> {

**private** T[] elements;

**private** **int** size;

**public** Array() {

elements = **null**;

size = 0;

}

**public** **void** addFirst(T element) {

**if** (size == 0) {

elements = (T[]) **new** Object[1];

elements[size++] = element;

} **else** {

T[] auxElements = elements;

elements = (T[]) **new** Object[++size];

System.*arraycopy*(auxElements, 0, elements, 1, size - 1);

elements[0] = element;

}

}

**public** **void** addLast(T element) {

**if** (size == 0) {

elements = (T[]) **new** Object[1];

elements[0] = element;

size = 1;

} **else** {

T[] auxElements = elements;

elements = (T[]) **new** Object[++size];

System.*arraycopy*(auxElements, 0, elements, 0, size - 1);

elements[size - 1] = element;

}

}

@Override

**public** **void** add(T element, **int** index) {

**if** (size == 0) {

addFirst(element);

} **else** **if** (index >= 0 && index <= size) {

T[] auxElements = elements;

elements = (T[]) **new** Object[++size];

System.*arraycopy*(auxElements, 0, elements, 0, index);

elements[index] = element;

System.*arraycopy*(auxElements, index, elements, index + 1, size - index - 1);

}

}

@Override

**public** T getFirst() {

**if** (size > 0)

**return** elements[0];

**else**

**return** **null**;

}

@Override

**public** T getLast() {

**if** (size > 0)

**return** elements[size - 1];

**else**

**return** **null**;

}

@Override

**public** T get(**int** index) {

**if** (size > 0) {

**if** (index >= 0 && index < size)

**return** elements[index];

**else** **return** **null**;

} **else**

**return** **null**;

}

@Override

**public** **void** removeFirst() {

**if** (size > 1) {

T[] auxElements = elements;

elements = (T[]) **new** Object[--size];

System.*arraycopy*(auxElements, 1, elements, 0, size);

} **else** {

elements = **null**;

size = 0;

}

}

@Override

**public** **void** removeLast() {

**if** (size > 1) {

T[] auxElements = elements;

elements = (T[]) **new** Object[--size];

System.*arraycopy*(auxElements, 0, elements, 0, size);

} **else** {

elements = **null**;

size = 0;

}

}

@Override

**public** **void** remove(T element) {

**for**(**int** i = 0; i < size; i++)

**if** (element.equals(elements[i])) {

**if** (size > 1) {

T[] auxElements = elements;

elements = (T[]) **new** Object[--size];

System.*arraycopy*(auxElements, 0, elements, 0, i);

System.*arraycopy*(auxElements, i + 1, elements, i, size - i);

} **else** {

size = 0;

elements = **null**;

}

}

}

@Override

**public** **void** removeIndex(**int** index) {

**if** (size > 1) {

**if** (index >= 0 && index < size) {

T[] auxElements = elements;

elements = (T[]) **new** Object[--size];

System.*arraycopy*(auxElements, 0, elements, 0, index);

System.*arraycopy*(auxElements, index + 1, elements, index, size - index);

}

} **else** **if** (index == 0) {

elements = **null**;

size = 0;

}

}

@Override

**public** **int** getSize() {

**return** size;

}

**public** String toString() {

String s = "size=" + size;

**for**(**int** i = 0; i < size; i++)

s += " [" + i + "]=" + elements[i].toString();

**return** s;

}

}

**public** **class** List<T> **implements** ICollection<T> {

**private** ListItem<T> first, last;

**private** **int** size;

**private** **class** ListItem<T> {

**public** T data;

**public** ListItem<T> next;

**public** ListItem(T data, ListItem<T> next) {

**this**.data = data;

**this**.next = next;

}

}

**public** List() {

first = last = **null**;

size = 0;

}

@Override

**public** **void** addFirst(T element) {

**if** (size == 0) {

first = last = **new** ListItem<>(element, **null**);

} **else** {

first = **new** ListItem<>(element, first);

}

size++;

}

@Override

**public** **void** addLast(T element) {

**if** (size == 0)

first = last = **new** ListItem<>(element, **null**);

**else** {

last.next = **new** ListItem<>(element, **null**);

last = last.next;

}

size++;

}

@Override

**public** **void** add(T element, **int** index) {

**if** (index == 0)

addFirst(element);

**else** **if** (index == size)

addLast(element);

**else** **if** (index > 0 && index < size) {

ListItem<T> t = first;

**for**(**int** i = 0; i < index - 1; i++, t = t.next);

t.next = **new** ListItem<>(element, t.next);

size++;

}

}

@Override

**public** T getFirst() {

**return** first == **null** ? **null** : first.data;

}

@Override

**public** T getLast() {

**return** last == **null** ? **null** : last.data;

}

@Override

**public** T get(**int** index) {

**if** (index >= 0 && index < size) {

ListItem<T> t = first;

**for**(**int** i = 0; i < index; i++, t = t.next);

**return** t.data;

} **else**

**return** **null**;

}

@Override

**public** **void** removeFirst() {

**if** (size == 1) {

first = last = **null**;

size--;

} **else** **if** (size > 1) {

first = first.next;

size--;

}

}

@Override

**public** **void** removeLast() {

**if** (size == 1) {

first = last = **null**;

size--;

} **else** **if** (size > 1) {

ListItem<T> t;

**for**(t = first; t.next != last; t = t.next);

t.next = **null**;

last = t;

size--;

}

}

@Override

**public** **void** remove(T element) {

**if** (size > 0) {

**if** (first.data.equals(element))

removeFirst();

**else** {

ListItem<T> t, prevt;

**for**(prevt = first, t = first.next; t != **null**; t = t.next, prevt = prevt.next) {

**if** (t.data.equals(element)) {

**if** (t == last)

last = prevt;

prevt.next = t.next;

size--;

}

}

}

}

}

@Override

**public** **void** removeIndex(**int** index) {

**if** (index >= 0 && index < size) {

**if** (index == 0)

removeFirst();

**else** **if** (index == size - 1)

removeLast();

**else** {

ListItem<T> t, prevt;

**int** i;

**for**(i = 1, prevt = first, t = first.next; i < index;

i++, t = t.next, prevt = prevt.next);

prevt.next = t.next;

size--;

}

}

}

@Override

**public** **int** getSize() {

**return** size;

}

**public** String toString() {

String s = "size=" + size;

**int** i = 0;

**for**(ListItem<T> t = first; t != **null**; t = t.next)

s += " [" + i++ + "]=" + t.data.toString();

**return** s;

}

}

**public** **final** **class** CollectionFactory {

**private** **static** CollectionFactory *instance* = **new** CollectionFactory();

**private** CollectionFactory() {}

**public** **static** CollectionFactory getCollectionFactoryInstance() {

**return** *instance*;

}

**public** <T> ICollection<T> getICollectionInstance(String s) {

**if** (s.equals("array"))

**return** **new** Array<T>();

**else** **if** (s.equals("list"))

**return** **new** List<T>();

**else**

**return** **null**;

}

}

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

CollectionFactory cf = CollectionFactory.*getCollectionFactoryInstance*();

ICollection<Integer> c = cf.<Integer>getICollectionInstance("array");

**for**(**int** i = 0; i < 15; i++)

c.addFirst(i);

System.***out***.println(c);

}

}