# Tema 2. Problema 1: La llista doblement enllaçada i polimòrfica

Esteve Brugulat, Josep M. Ribó

15 de novembre de 2010

## 1 Objectius

- Revisar el polimorfisme
- Representar les llistes amb un doble enllaç

Per fer aquest problema, cal que us llegiu:
- Els apunts (seccions 2.1-2.4)

## 2 Enunciat

Simplifiquem la plantilla de classes `List<T>` de manera que només tingui les operacions següents:

```
template <class T>
class List :public Container {

public:

  List();

  virtual ~List(){}

  virtual void putFirst(const T&)=0;
  virtual void put(const T&, ListIterator<T>&)=0;
  virtual T* get(ListIterator<T>&) const
                throw (IteratorException) =0;
  virtual void remove(ListIterator<T>&) =0;
                throw (IteratorException) =0;
  MyString toString() const;
 };
```

Simplificarem també la classe `Iterator`, la qual ara oferirà les operacions següents:

```
template<class T>
class Iterator :public Object{
protected:
  Container* conta;

public:
```

```
 Iterator();
 Iterator(const Container& c);
 virtual ~Iterator(){}

 virtual void setFirst()=0;
 virtual void operator++() throw (IteratorException)=0;
 virtual void operator--() throw (IteratorException)=0;
 virtual bool isEnd() const =0;
 MyString toString();
};
```

## 2.1 Objectiu del problema

L'objectiu del problema és implementar dues classes:

- `DLinkedList<T>` (*Double linked list*)

  Llista enllaçada en què els nodes contenen dos apuntadors: un a l'anterior node i l'altre al següent.

  Els elements que podrà emmagatzemar aquesta llista seran objectes de la classe `T` **i de qualsevol subclasse de** `T`.

  Voldrem que les operacions `put` i `putFirst` insereixin a la llista **una còpia** de l'objecte que rebin com a paràmetre. Voldrem també que l'operació `get` retorni un apuntador a **una còpia** de l'objecte al que es refereix l'iterador.

- `DLinkedListIterator<T>`

  Iterador sobre objectes de la classe `DLinkedList`.

Per tal d'orientar-vos sobre el desenvolupament d'aquestes classes us proposem les següents consideracions.

## 2.2 Consideracions sobre aquestes classes

- Per implementar la classe `DLinkedList`, quants apuntadors haurà de tenir la classe `Node`?
- L'atribut `ed` de la classe `DLinkedListIterator`, serà necessari que apunti a l'element anterior d'aquell al que es refereix (com passava en el cas de `LinkedList`)?
- En una representació d'una llista amb enllaços al següent i a l'anterior, quants fantasmes seran necessaris per tal d'evitar haver de considerar casos particulars a l'hora de programar les insercions i les eliminacions? Com es representarà la llista buida?
- Volem que la classe `DLinkedList<T>` sigui capaç de contenir objectes de qualsevol subclasse de `T`. O sigui:

```
DLinkedList<RegularPolygon> dl;

Square sq(1.0, 2.0);

Circumference cr(3.0, 4.0);

dl.putFirst(sq);
dl.putFirst(cr);
```

- Com a conseqüència, l'atribut `val` de la classe `Node<T>` no podrà ser de classe `T` sinó ... a `T`. Per què?

```
template <class T>
class Node{
public:
 T val;  //Mmmmmmm..... Segur???? ...
 .....
};
```

- Volem que la classe `DLinkedList` guardi còpies dels objectes que s'hi insereixen. Exemple:

```
template <class T>
 void DLinkedList<T>::put(const T& x, ListIterator<T>& it)
{
  //(1) Transformar it en DLinkedListIterator<T>
  //                                ===>dynamic_cast
  //(2) aux=copia de x
  //(3) Inserir aux a la llista al lloc anterior a it
}
```

  Però com fem (2) si no sabem el tipus exacte de `x`? (Pot ser de qualsevol subclasse de `T`). No podem fer: `aux= new T;` perquè `x` podria ser d'una subclasse de `T`.

  Per resoldre això haureu d'usar l'operació `clone`.

## 2.3 Una altra volta de cargol amb el polimorfisme (Optatiu)

Els iteradors, tal com els hem presentat, plantegen un problema a l'hora de treballar polimòrficament. Imaginem que volem dissenyar una acció `showContainer(const Container& c)` que mostri per la sortida estàndar (típicament, el teclat) tots els objectes que conté un determinat contenidor.

El que sabem del polimorfisme ens invita a dissenyar `showContainer` de la manera següent:

```
template<class T>
void showContainer(const Container& c)
{
  Iterator<T> it(c);
  T* aux;

  it.setFirst();

  while(!it.isEnd()){
     cout<<(*c).toString()<<endl;
     it++;
  }
}

int main()
{
  DLinkedList<RegularPolygon> dl;
  ArrayStack<RegularPolygon> s;

  Square sq(1.0, 2.0);
  Circumference cr(3.0, 4.0);
```

```
  dl.putFirst(sq);
  dl.putFirst(cr);

  s.push(sq);
  s.push(cr);

  cout<<"Contingut de la llista:"<<endl;
  showContainer(dl);

  cout<<"Contingut de la pila:"<<endl;
  showContainer(c);
}
```

Amb l'objectiu que aquest programa mostri com a resultat:

```
Contingut de la llista:

CIRCUMFERENCE

Edgelength=0
Color=4.0
Area=29.57
Perimeter=9.42
Radius=3.0
-----------------
SQUARE

Edgelength=1.0
Color=2.0
Area=1.0
Perimeter=4.0

Contingut de la pila:

CIRCUMFERENCE

Edgelength=0
Color=4.0
Area=29.57
Perimeter=9.42
Radius=3.0
-----------------
SQUARE

Edgelength=1.0
Color=2.0
Area=1.0
Perimeter=4.0
```

**Però aquest programa NO FUNCIONA.** Per què? A més a més, no és fàcil de fer-lo funcionar amb les operacions que té la plantilla de classes `List<T>`.

**Quina operació caldria afegir a la plantilla de classes `List<T>` per tal que es pugui esmenar el programa anterior i assolir el resultat desitjat d'aquest programa?** .