Universidade Tecnológica Federal do Paraná
UTFPR – Campus Curitiba

# Orientação a Objetos
-
# Programação em C++

**Grupo de Slides 10 – Parte A:**

10.1 Noção de Evento.

**10.2 Gabaritos ou *Templates*.**

**Prof. Dr. Jean Marcelo SIMÃO – DAINF / UTFPR**

# Eventos

- Noções de:
  - Programação orientada a eventos
  - Conceitos de estados e eventos.
  - Comunicação entre objetos via eventos.

```
void Principal::MenuCad ( )
{
   int op = -1;
   while (op != 5)
   {
      system("cls");
      cout   << "  Informe sua opção:              "          << endl;
      cout   << "  1 - Cadastrar Disciplina.       "          << endl;
      cout   << "  2 - Cadastrar Departamentos.    "          << endl;
      cout   << "  3 - Cadastrar Universidade.     "          << endl;
      cout   << "  4 - Cadastrar Aluno.            "          << endl;
      cout   << "  5 – Sair.                       "          << endl;
      cin    >> op;

      switch ( op )
      {
            case 1 :    { CadDisciplina ();                   }
                        break;
            case 2:     { CadDepartamento ();                 }
                        break;
            case 3:     { CadUniversidade ();                 }
                        break;
            case 4:     { CadAluno ();                        }
                        break;
            case 5:     { cout << " FIM " << endl;            }
                        break;
            default:    {
                          cout << "Opção Inválida - Pressione uma tecla." << endl;
                          getchar();
                        }
      }
   }
}
```

Evento

**Um evento dispara a execução deste método...**

```
void Principal::CadDepartamento ( )
{
    char nomeUniversidade [150];
    char nomeDepartamento [150];
    Universidade* univ;
    Departamento* depart;

     cout << "Qual o nome da universidade do departamento" << endl;
     cin  >> nomeUniversidade;
     univ = LUniversidades.localizar (nomeUniversidade);

     if ( univ != NULL )
     {
              cout << "Qual o nome do departamento" << endl;
              cin >> nomeDepartamento;
              depart = new Departamento ( );
              depart->setNome ( nomeDepartamento );
              depart->setUniversidade ( univ );
              LDepartamentos.incluaDepartamento ( depart );      }
      else
     {
              cout << "Universidade inexistente." << endl;
     }
}
```

**...que promove a comunicação entre alguns objetos...**

**...e que promove a alteração de estados de outros objetos.**

# Segunda Parte

## Gabaritos ou *Templates*

# Retomando conteúdo anterior

- Cada Disciplina deve ser capaz de armazenar uma lista de Alunos.

- A classe *Aluno*, entretanto, não deverá possuir um ponteiro para o Próximo. Isto deverá estar em uma classe associada chamada *ElAluno relacionada a classe ListaAlunos*...

**Reflexão: Esta solução de criar uma classe El'*Algo*' cada vez que se faz necessário uma lista em um classe dada, parece-lhes uma solução inteligente ou mesmo razoável?**

**Não seria melhor ter um classe *Elemento* genérica para tratar disto?**

# Gabaritos ou Templates

**Gabaritos permitem criar classes genéricas... Vejamos um exemplo:**

```
#ifndef _ELEMENTO_H_
#define _ELEMENTO_H_

// Um gabarito se define inteiramente no .h
template<class TIPO>
class Elemento
{
private:
   Elemento<TIPO>*   pProximo;
   Elemento<TIPO>*   pAnterior;
   TIPO*             pInfo;
   char              nome[150];

public:
   Elemento ( );
   ~Elemento ( );

   void setProximo ( Elemento<TIPO>* pp );
   Elemento<TIPO>* getProximo ( );

   void setAnterior ( Elemento<TIPO>* pa );
   Elemento<TIPO>* getAnterior ( );

   void setInfo ( TIPO* pi );
   TIPO* getInfo ( );

   void setNome ( char* n );
   char* getNome ( );
};
```

```
#ifndef _ELEMENTO_H_
#define _ELEMENTO_H_

// Um gabarito se define inteiramente no .h
template<class TIPO>
class Elemento
{
private:
   Elemento<TIPO>*          pProximo;
   Elemento<TIPO>*          pAnterior;
   TIPO*                    pInfo;
   char                     nome [ 150 ];

public:
   Elemento ( );
   ~Elemento ( );

   void setProximo ( Elemento<TIPO>* pp );
   Elemento<TIPO>* getProximo ( );

   void setAnterior ( Elemento<TIPO>* pa );
   Elemento<TIPO>* getAnterior ( );

   void setInfo ( TIPO* pi );
   TIPO* getInfo ( );

   void setNome ( char* n );
   char* getNome ( );
};
```

```
template<class TIPO>
Elemento<TIPO>::Elemento ( )
{
   pProximo = NULL;  pAnterior = NULL; pInfo = NULL;
}

template<class TIPO>
void Elemento<TIPO>::setProximo ( Elemento<TIPO>* pp )
{
   pProximo = pp;
}
```

```
template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getProximo ( )
{
   return pProximo;
}

template<class TIPO>
void Elemento<TIPO>::setAnterior ( Elemento<TIPO>* pa )
{
   pAnterior = pa;
}

template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getAnterior ( )
{
   return pProximo;
}

template<class TIPO>
void Elemento<TIPO>::setInfo ( TIPO* pi )
{
   pInfo = pi;
}

template<class TIPO>
TIPO* Elemento<TIPO>::getInfo ( )
{
   return pInfo;
}

template<class TIPO>
void Elemento<TIPO>::setNome ( char*  n )
{
   strcpy ( nome, n );
}

template<class TIPO>
char* Elemento<TIPO>::getNome ()
{
   return nome;
}
#endif
```

**Atenção: *Templates* são implementados somente no .h, incluindo a implementação dos métodos!**

```
#ifndef _ELEMENTO_H_
#define _ELEMENTO_H_
// Um gabarito se define inteiramente no .h
template<class TIPO>
class Elemento
{
private:
   Elemento<TIPO>*          pProximo;
   Elemento<TIPO>*          pAnterior;
   TIPO*                    pInfo;
   char                     nome [ 150 ];

public:
   Elemento ( );
   ~Elemento ( );

   void setProximo ( Elemento<TIPO>* pp );
   Elemento<TIPO>* getProximo ( );

   void setAnterior ( Elemento<TIPO>* pa );
   Elemento<TIPO>* getAnterior ( );

   void setInfo ( TIPO* pi );
   TIPO* getInfo ( );

   void setNome ( char* n );
   char* getNome ( );
};
```

```
template<class TIPO>
Elemento<TIPO>::Elemento ( ) {
   pProximo = NULL;  pAnterior = NULL; pInfo = NULL;
}

template<class TIPO>
void Elemento<TIPO>::setProximo ( Elemento<TIPO>* pp )
{
   pProximo = pp;
}
```

```
template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getProximo ( )
{
   return pProximo;
}

template<class TIPO>
void Elemento<TIPO>::setAnterior ( Elemento<TIPO>* pa )
{
   pAnterior = pa;
}

template<class TIPO>
Elemento<TIPO>* Elemento<TIPO>::getAnterior ( )
{
   return pProximo;
}

template<class TIPO>
void Elemento<TIPO>::setInfo ( TIPO* pi )
{
   pInfo = pi;
}

template<class TIPO>
TIPO* Elemento<TIPO>::getInfo ( ) {
   return pInfo;
}

template<class TIPO>
void Elemento<TIPO>::setNome ( char*  n ) {
   strcpy ( nome, n );
}

template<class TIPO>
char* Elemento<TIPO>::getNome () {
   return nome;
}
#endif
```

```
#ifndef _LISTAALUNOS_H_
#define _LISTAALUNOS_H_

#include "Elemento.h"
#include "Aluno.h"

class ListaAlunos
{
private:
      int         cont_alunos;
      int         numero_alunos;
      char        nome [ 150 ];

      Elemento<Aluno> *pElAlunoPrim;
      Elemento<Aluno> *pElAlunoAtual;

public:

     ListaAlunos ( int na = -1, char* n = "" );
     ~ListaAlunos ( );

     void limpaLista ( );
     void incluaAluno ( Aluno* pa );
     void listeAlunos ( );
     void listeAlunos2 ( );
     void graveAlunos ( );
     void recupereAlunos ( );
};

#endif
```

```
#include "stdafx.h"
#include "ListaAlunos.h"

ListaAlunos::ListaAlunos (int na, char* n)
{
    numero_alunos = na;
    cont_alunos       = 0;
    ElAlunoPrim      = NULL;
    ElAlunoAtual     = NULL;
    strcpy (nome, n );
}

ListaAlunos::~ListaAlunos ()
{
  limpaLista ();
}

void ListaAlunos::limpaLista ()
{
  Elemento<Aluno> *paux1;
  Elemento<Aluno> *paux2;

  paux1 = pElAlunoPrim;
  paux2 = paux1;

  while ( paux1 != NULL )
  {
      paux2 = paux1->getProximo();
      delete ( paux1 );
      paux1 = paux2;
  }

  pElAlunoPrim  = NULL;
  pElAlunoAtual = NULL;
}
```

```
void ListaAlunos::incluaAluno ( Aluno* pa )
{
   if (
        ( ( cont_alunos < numero_alunos ) && ( pa != NULL ) ) ||
        ( ( -1 == numero_alunos) && ( pa != NULL ) )
      )
   {
      // Aqui é criado um ponteiro para LAluno
      Elemento <Aluno>* paux;

      // Aqui é criado um objeto LAluno,
      // sendo seu endereço armazenado em aux
      paux = new Elemento<Aluno>( );
      paux->setNome ( pa->getNome() );

      // Aqui recebe uma ref do objeto interm.
      paux->setInfo ( pa );

      if ( pElAlunoPrim == NULL )
      {
        pElAlunoPrim   = paux;
        pElAlunoAtual  = paux;
      }
      else
      {
        pElAlunoAtual->setProximo ( paux );
        paux->setAnterior ( pElAlunoAtual );
        pElAlunoAtual = paux;
      }
      cont_alunos++;
   }
   else
   {
       cout  << "Aluno não incluído. Turma já lotada em "
             << numero_alunos << " alunos." << endl;
   }
}
```

```
void ListaAlunos::listeAlunos ( )
{
   Elemento < Aluno >* paux;
   paux = pElAlunoPrim;

   while ( paux != NULL )
   {
        cout   << " Aluno " << paux->getNome()
               << " matriculado na Disciplina "
               << nome << "." << endl;
       paux = paux->getProximo();
   }
}
```

```
void ListaAlunos::incluaAluno ( Aluno* pa )
{
   if (
        ( ( cont_alunos < numero_alunos ) && ( pa != NULL ) ) ||
        ( ( -1 == numero_alunos) && ( pa != NULL ) )
      )
   {
     // Aqui é criado um ponteiro para LAluno
     Elemento <Aluno>* paux;

     // Aqui é criado um objeto LAluno,
     // sendo seu endereço armazenado em aux
     paux = new Elemento<Aluno>( );
     // paux->setNome ( pa->getNome() );

     // Aqui recebe uma ref do objeto interm.
    paux->setInfo ( pa );

     if ( pElAlunoPrim == NULL )
     {
       pElAlunoPrim   = paux;
       pElAlunoAtual  = paux;
     }
     else
     {
       pElAlunoAtual->setProximo ( paux );
       paux->setAnterior ( pElAlunoAtual );
       pElAlunoAtual = paux;
     }
     cont_alunos++;
   }
   else
   {
      cout  << "Aluno não incluído. Turma já lotada em "
            << numero_alunos << " alunos." << endl;
   }
}
```

```
void ListaAlunos::listeAlunos ( )
{
   Elemento < Aluno >* paux;
   paux = pElAlunoPrim;

   while ( paux != NULL )
   {
        cout   << " Aluno " << paux->getNome()
               << " matriculado na Disciplina "
               << nome << "." << endl;
        paux = paux->getProximo();
   }
}
```

```
void ListaAlunos::listeAlunos ( )
{
   Aluno * pauxAl = NULL

   Elemento < Aluno >* pauxEL = NULL;
   pauxEL = pElAlunoPrim;

   while ( pauxEL != NULL )
   {
        pauxAl = pauxEL->getInfo();

        cout   << " Aluno " << pauxAl->getNome()
               << " matriculado na Disciplina "
               << nome << "." << endl;
       pauxEL = pauxEL->getProximo();
   }
}
```

# Retomando o exercício anterior

- Cada Departamento deve ser capaz de armazenar uma lista de disciplinas.

- A classe *Disciplina*, entretanto, não deverá possuir um ponteiro para o Próximo. Isto deverá estar em uma classe associada chamada "*ElDisciplina*" *relacionada a ListaDisciplina*...

**Utilizar o gabarito** ***template<class*** *TIPO>* ***class*** *Elemento* para resolver o exercício em questão

# Reflexão

**Continuando Reflexão: A solução utilizando o gabarito *template<class* *TIPO*> *class* *Elemento* facilita a composição de listas em cada classe pertinente.**

**Mas o tratamento de inclusão e consulta da lista em cada classe que necessita de uma lista parece-lhes uma solução inteligente ou mesmo razoável?**

**Não seria melhor ter um classe Lista genérica para tratar disto ou parte disso?**

```
#ifndef _LISTA_H_
#define _LISTA_H_

#include "Elemento.h"

template<class TIPO>
class Lista
{

private:
   Elemento<TIPO>*  pPrimeiro;
   Elemento<TIPO>*  pAtual;

public:

   Lista ( );
   ~Lista ( );
   void inicializa ( );
   bool incluaElemento ( Elemento<TIPO>* pElemento );
   bool incluaInfo ( TIPO* pInfo, char* nome = "" );
   void listeInfos ( );

};

template<class TIPO>
Lista<TIPO>::Lista ( )
{
   inicializa ( );
}

template<class TIPO>
Lista<TIPO>::~Lista ( )
{
  // Fazer código de desalocação...
}
```

```
template<class TIPO>
void Lista<TIPO>::inicializa ( )
{
    pPrimeiro = NULL;
    pAtual    = NULL;
}

template<class TIPO>
bool Lista<TIPO>::incluaElemento ( Elemento<TIPO>* pElemento )
{

   if (NULL != pElemento)
   {

     if (NULL == pPrimeiro)
     {
        pPrimeiro = pElemento;
        pAtual = pPrimeiro;
     }
     else
     {
        pElemento->setAnterior ( pAtual );
        //pElemento->setProximo ( NULL );
        pAtual->setProximo ( pElemento );
        pAtual = pAtual->getProximo ( );
     }
     return true;
   }
   else
   {
      cout << " Erro, elemento nulo na lista. " << endl;
      return false;
   }

}
```

```
template<class TIPO>
bool Lista<TIPO>::incluaInfo ( TIPO * pInfo, char* nome )
{
    if (NULL != pInfo)
    {
        Elemento<TIPO>* pElemento = NULL;
        pElemento = new Elemento<TIPO>();
        pElemento->setNome ( nome );
        pElemento->setInfo ( pInfo );
        incluaElemento ( pElemento );
        return true;
    }
    else
    {
        printf ( "Erro, elemento (informação) nulo(a) na lista. \n" );
        return false;
    }
}

template<class TIPO>
void Lista<TIPO>::listeInfos ( )
{
   Elemento<TIPO>* pAux;
   pAux = pPrimeiro;

   if (NULL != pPrimeiro )
   {
       while ( NULL != pAux )
       {
           printf ( " Elemento na lista %s \n", pAux->getNome() );
           pAux = pAux->getProximo();
       }
   }
}

#endif
```

```
class Disciplina
{
private:
   int id;
   char nome [150];
   char area_conhecimento [150];

   Departamento* pDeptoAssociado;
   Lista < Aluno > ObjLAlunos;

public:
   Disciplina ( int i, int na = 45, char* ac = "" );
   ~Disciplina ( );

   void setId ( int i );
   int getId ();

   void setNome (char* n);
   char* getNome ();

   void setDepartamento (Departamento* pd);
   Departamento* getDepartamento ( );

   void incluaAluno ( Aluno* pa );
   void listeAlunos ( );
   void listeAlunos2 ( );
};

#endif
```

**Disciplina.h**

```
#include "stdafx.h"
#include "Disciplina.h"

Disciplina::Disciplina (int i, int na, char* ac):
ObjLAlunos ()
{
   id = i;
   pDeptoAssociado = NULL;

   strcpy (area_conhecimento, ac );
}


Disciplina::~Disciplina ()
{
  DeptoAssociado  = NULL;
}


void  Disciplina::setId ( int i )
{
  id = i;
}

int   Disciplina::getId ( )
{
   return id;
}
```

**Disciplina.cpp**

```
void  Disciplina::setNome ( char* n )
{
    strcpy ( nome, n );
}

char* Disciplina::getNome ( )
{
    return nome;
}

void Disciplina::setDepartamento (Departamento* pd)
{
  // Cada vez que um Departamento
  // é associado a uma Disciplina,
  // esta Disciplina passa a fazer
  // parte da lista de disciplina
  // do Departamento, por meio do comando abaixo.
  pDeptoAssociado = pd;
  pDeptoAssociado->incluaDisciplina ( this );
}

Departamento* Disciplina::getDepartamento ( )
{
   return  DeptoAssociado;
}

void Disciplina::incluaAluno ( Aluno* pa )
{
   ObjLAlunos.incluaInfo ( pa, pa->getNome() );
}
```

```
void Disciplina::listeAlunos ()
{
   ObjLAlunos.listeInfos ();

}

void Disciplina::listeAlunos2 ( )
{
    // ObjLAlunos.listeAlunos2 ();
}
```

```
void  Disciplina::setNome ( char* n )
{
    strcpy ( nome, n );
}

char* Disciplina::getNome ( )
{
    return nome;
}

void Disciplina::setDepartamento (Departamento* pd)
{
  // Cada vez que um Departamento
  // é associado a uma Disciplina,
  // esta Disciplina passa a fazer
  // parte da lista de disciplina
  // do Departamento, por meio do comando abaixo.
  pDeptoAssociado = pd;
  pDeptoAssociado->incluaDisciplina ( this );
}

Departamento* Disciplina::getDepartamento ( )
{
   return  DeptoAssociado;
}

void Disciplina::incluaAluno ( Aluno* pa )
{
   ObjLAlunos.incluaInfo ( pa );
}
```

```
void Disciplina::listeAlunos ()
{
   // ObjLAlunos.listeInfos ();
   Elemento<Aluno>* pAux;
    pAux = ObjLAluno.getPrimeiro();

    if (NULL != pPrimeiro )
    {
        while ( NULL != pAux )
        {
            Aluno* pAluno = NULL;
            pAluno = pAux->getInfo();

            printf ( " Elemento na lista %s %d \n",
                     pAluno->getNome(),
                     pAluno->getRA() );

            pAux = pAux->getProximo();
        }
    }

}

void Disciplina::listeAlunos2 ( )
{
    // ObjLAlunos.listeAlunos2 ();
}
```

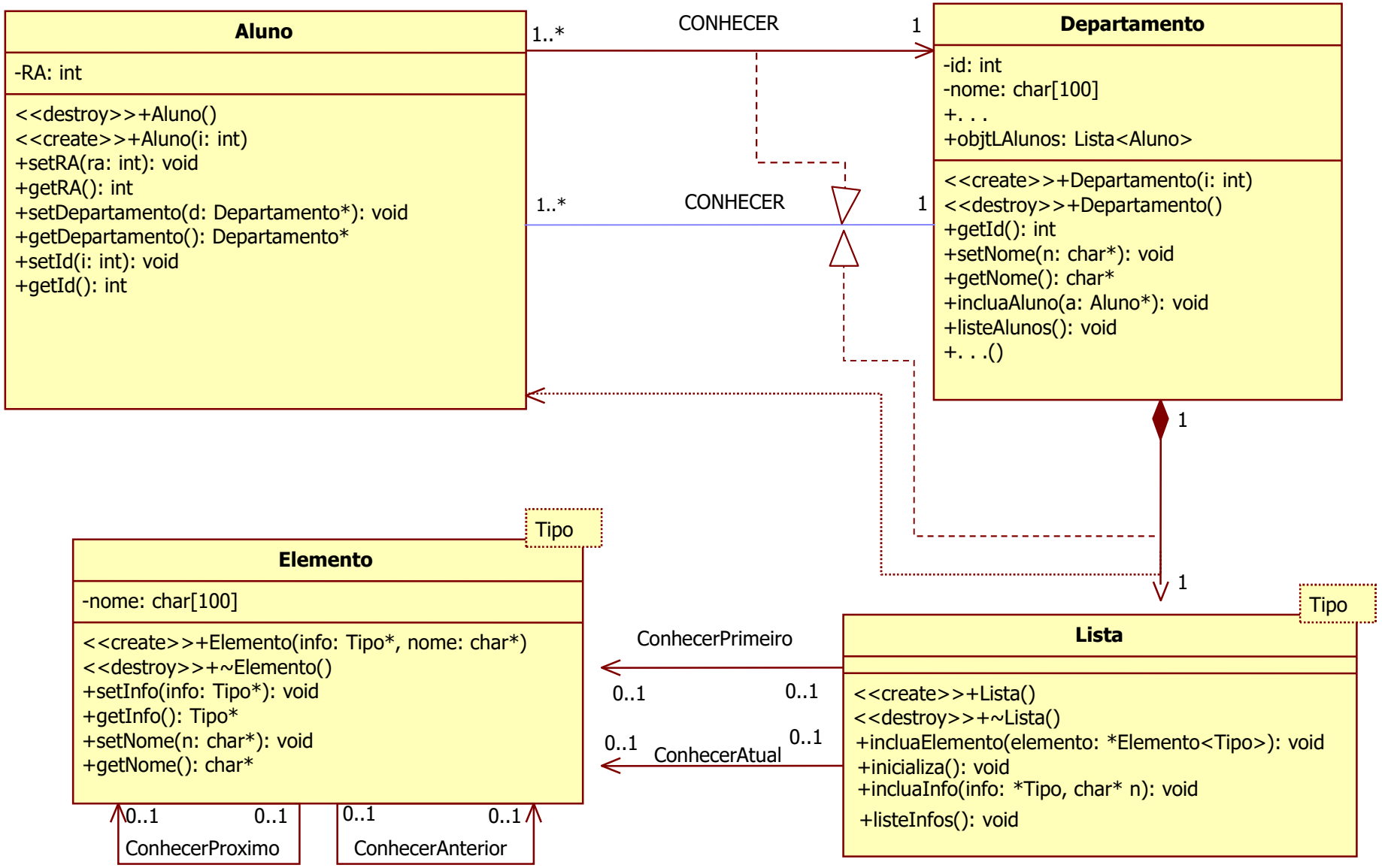
# Exercício

**a)** Melhorar o *template* de Lista desenvolvido e a solução como um todo para que suas instâncias que tratam de lista tenham no âmbito de suas classes um bom equilíbrio entre generalidade e especificidade, sempre a luz dos princípios de coesão e desacoplamento.

**b)** Substituir todas as outras listas 'específicas' elaboradas nas versões anteriores do 'sistema exemplo', por 'instâncias' conforme a solução supostamente desenvolvida no tópico anterior, nomeadamente tópico (a).

**c)** Retomando um exercício anterior a luz dos avanços alcançado até então nos tópicos (a) e (b):
- Elaborar uma solução para o armazenar as notas (1ª parcial, 2ª parcial e final) e número de faltas de cada aluno em cada disciplina.

# *Templates* em UML (conceito de dependência)



# *Templates* em UML (conceito de dependência)