



Residência
em Software

Composição e Agregação

Professores:
Álvaro Coelho, Edgar Alexander, Esbel
Valero e Hélder Almeida

INSTITUIÇÃO EXECUTORA



COORDENADORA



APOIO



Um exemplo

- Um sistema de pacotes turísticos.
 - Um pacote é contratado por clientes.
 - Clientes possuem um CPF e um nome
 - Clientes podem possuir dependentes
 - Dependentes possuem um nome
 - Um pacote tem um nome e vários eventos
 - Um evento tem uma duração prevista
 - Um evento de deslocamento permite que um roteiro seja complementado com outro. Importante: deslocamentos também tem uma duração prevista
 - Um pacote pode possuir também um ou mais eventos de pernoite
 - Então, Pernoites, deslocamentos e roteiros são eventos. E estão sempre ordenados dentro do pacote.
 - Um pernoite tem uma duração prevista

Um exemplo do exemplo

- Um sistema de pacotes turísticos.
 - Um pacote: São João em Campina Grande é contratado por Joana da Silva
 - Joana tem o cpf 111.111.111-11
 - Joana tem dois dependentes: Pedro da Silva e João da Silva
 - O pacote São João em Campina Grande possui os seguintes roteiros, com suas respectivas durações.
 - (R1) Noite no Parque do Povo(1): 4h
 - (R2) Trem do Forró(2): 8h
 - (R3) Visita ao Armorial Ariano Suassuna(3): 2h
 - (R4) Vila Forró(4): 6h
 - (R5) Noite no Parque do Povo(5): 4h

Um exemplo do exemplo

- O pacote possui os deslocamentos
 - (D1) Deslocamento Hotel-Noite no Parque do Povo:1h,
 - (D2) Noite no Parque do Povo-Hotel:1h
 - (D3) Hotel-Trem do Forró: 2h,
 - (D4) Trem do Forró-Armorial: 1h
 - (D5) Armorial-Vila Forró:2h
 - (D6) Vila Forró-Noite no Parque do Povo:1h,
 - (D7) Noite no Parque do Povo-Hotel:1h
- E os pernoites
 - (P1) Pernoite:10h
 - (P2) Pernoite:10h.
- Assim, o roteiro São João em Campina Grande tem a seguinte seqüencia de eventos
- D1-> R1-> D2 -> P1 -> D3 ->R2 -> D4 -> R3 -> D5 ->R4-> D6 -> R5 -> D7 -> P2

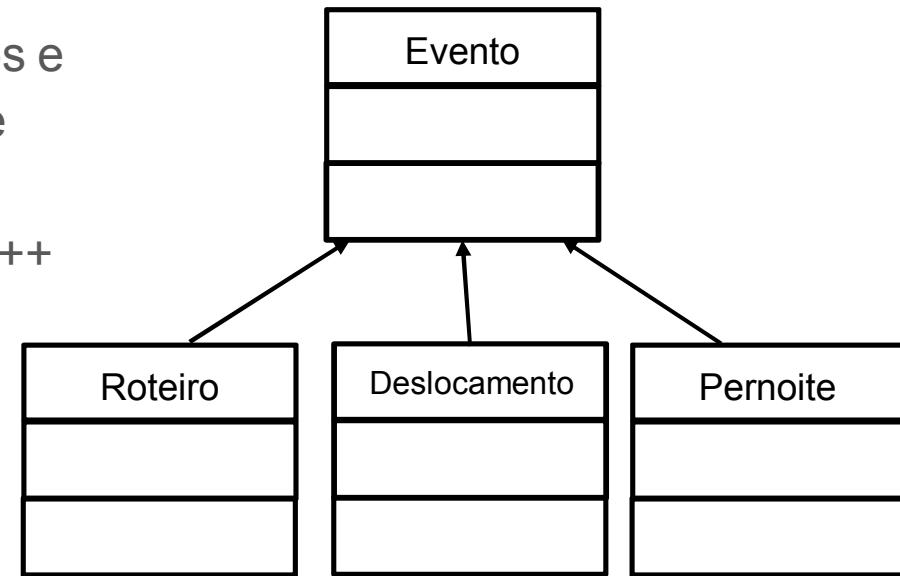


Residência
em Software

**Não precisaremos implementar tudo isso
(ainda...)**

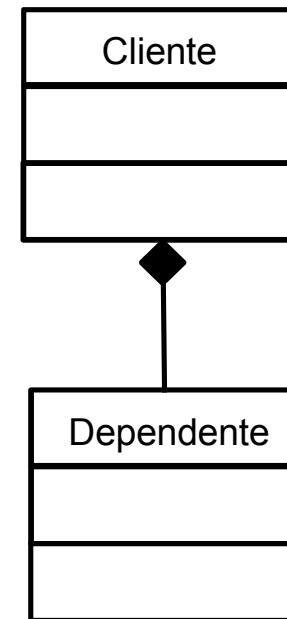
Olhando para algumas classes

- As classes Roteiros, Deslocamentos e Pernoites são subclasses da classe Eventos
- Já sabemos implementar isto em C++



Olhando para algumas classes

- As classes Clientes e Dependentes possuem um tipo diferente de relação
- No contexto do domínio do problema
 - Um dependente *precisa* de um cliente pra existir
 - (no mundo real, mais amplo, não há esta restrição)
- Composição
 - Relação entre classes de forma que uma é parte de outra, e não tem sentido existir sem ela



Implementação

- A classe principal deve ser implementada antes
 - Ela vai ser um atributo da classe dependente

```
class cliente {  
    private:  
        string nome, cpf;  
    public:  
        cliente(string n, string c) {  
            nome = n;  
            cpf = c;  
        }  
};
```



Implementação

- A classe principal deve ser implementada antes
 - Ela vai ser um atributo da classe dependente
- A classe dependente possui um *ponteiro* para um objeto da classe principal

```
class cliente {
    private:
        string nome, cpf;
    public:
        cliente(string n, string c) {
            nome = n;
            cpf = c;
        }
};

class dependente {
    private:
        string nome;
        cliente *dependente_de;
    public:
        dependente(cliente t, string n) {
            dependente_de = &t;
            nome = n;
        }
};
```

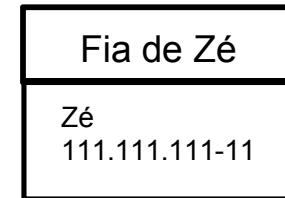
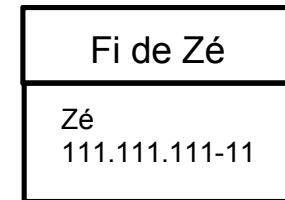
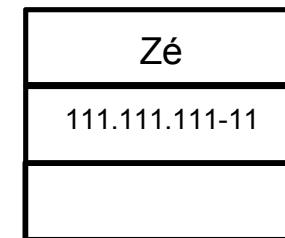


Residência
em Software

Porque um ponteiro?

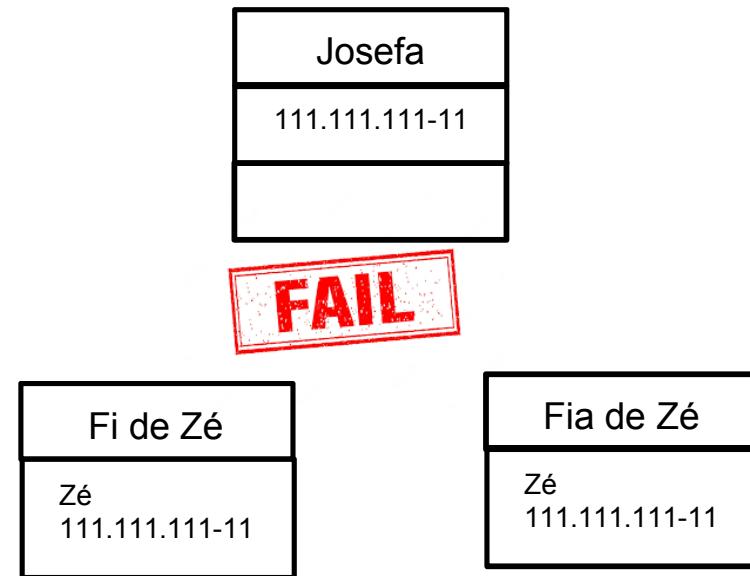
Porque um ponteiro

- Imagine que os quadrados ao lado são objetos na memória
 - Se quiser pode imaginar que o primeiro pixel onde ele está desenhado é seu endereço
- Note que há uma “cópia” dos dados de um cliente dentro do dependente
- Quando eu crio o dependente “Fia de Ze” eu posso colocar lá dentro uma “cópia” do objeto
 - Mas e se eu mudar o nome de Zé na origem?



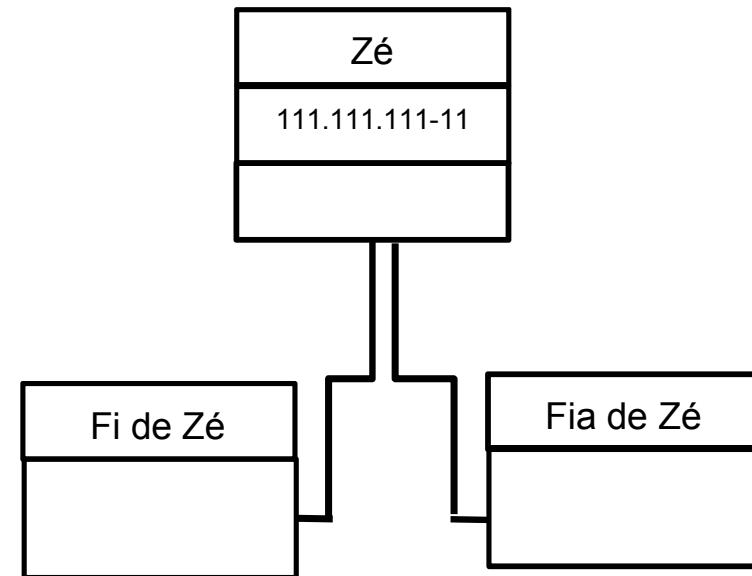
Porque um ponteiro

- Imagine que os quadrados ao lado são objetos na memória
 - Se quiser pode imaginar que o primeiro pixel onde ele está desenhado é seu endereço
- Note que há uma “cópia” dos dados de um cliente dentro do dependente
- Quando eu crio o dependente “Fia de Ze” eu posso colocar lá dentro uma “cópia” do objeto
 - Mas e se eu mudar o nome de Zé na origem?



Porque um ponteiro

- Ao invés da cópia, uma referência (ponteiro) para o dado original
- Qualquer alteração no objeto Zé é imediatamente refletida nos seus dependentes



Considerações adicionais

- Pelo fato de ser um ponteiro, a forma de se acessar os atributos e/ou métodos é ligeiramente diferente
- Suponha que um dependente tenha um método pra listar os dados de seu “tutor”

```
class dependente {  
    private:  
        string nome;  
        cliente *dependente_de;  
    public:  
        dependente(cliente t, string n) {  
            dependente_de = &t;  
            nome = n;  
        }  
        void listaDependente() {  
            cout << nome << " :dependente de "  
            << dependente_de->getNome();  
        }  
};
```

Considerações adicionais

```
class cliente {
    private:
        string nome, cpf;
    public:
        cliente(string n, string c) {
            nome = n;
            cpf = c;
        }
        string getNome(){
            return nome;
        }
        string getCpf(){
            return cpf;
        }
};
```

```
class dependente {
    private:
        string nome;
        cliente *dependente_de;
    public:
        dependente(cliente t, string n) {
            dependente_de = &t;
            nome = n;
        }
        void listaDependente() {
            cout << nome << " :dependente de "
                << dependente_de->getNome();
        }
};
```



Residência
em Software

Outras formas de se implementar a composição?

- Sim!
 - Uma coleção (vector?) de objetos do tipo dependente na classe principal
- DICA: esta estrutura normalmente é mais usada em agregações, como veremos adiante

Usando uma coleção

- Cada cliente terá uma coleção (vector) de dependentes

```
class Dependente {
    private:
        string nome;
    public:
        Dependente(string n) {
            nome = n;
        }
};

class Cliente {
    private:
        string nome, cpf;
        vector<Dependente> dependentes;
    public:
        Cliente(string n, string c) {
            nome = n;
            cpf = c;
        }
        void novoDependente(Dependente d) {
            dependentes.push_back(d);
        }
};
```

Exercício

- Crie uma classe Estado com os atributos nome e sigla (UF).
- Crie uma classe Cidade que tenha os atributos nome e estado (ponteiro para objeto do tipo Estado)
- Crie um programa em C para criar uma coleção (vector) de Estados (lido do usuário).
- Crie um programa em C para criar uma coleção (vector) de Cidades (lido do usuário)
 - Lembre que cada cidade deve pertencer a um Estado

Exercício

- Crie uma classe Estado com os atributos nome, sigla (UF) e uma coleção de objetos do tipo Cidade.
- Crie uma classe Cidade que tenha os atributos
- Crie um programa em C para criar uma coleção (vector) de Estados (lido do usuário).
- Crie um programa em C para, percorrendo cada um dos estados, criar uma coleção (vector) de Cidades (lido do usuário)
 - Lembre que cada cidade deve pertencer a um Estado

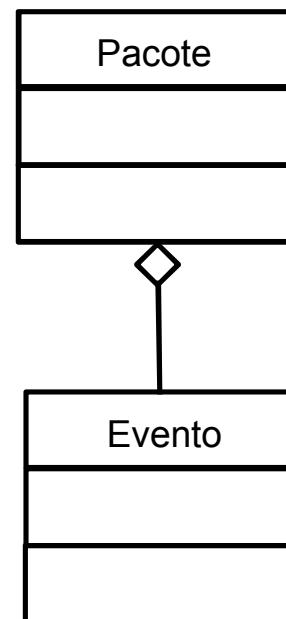
Agregação

- Voltando ao exemplo dos pacotes de turismo
- Lembrando que Evento é uma superclasse, que pode ser do tipo Pernoite, Deslocamento ou Roteiro
- Entre as classes Pacote e Evento existe uma relação parecida com a composição
 - Mas na composição uma classe precisa da outra para existir (dependente e Cliente)
- Numa agregação temos duas classes que se completam. Mas cada uma delas pode existir independente da outra.
- Por exemplo, o Evento Deslocamento Hotel-Parque do Povo não precisa de nenhum Pacote específico para existir.
 - E, na verdade, pode existir em muitos pacotes diferentes!



Residência
em Software

Agregação



Como implementar

- Não há grandes diferenças na construção dos atributos das classes no caso da agregação
 - Basicamente são as mesmas estratégias que usamos na Composição
- A diferença é conceitual – Algoritmica, portanto
 - Seu sistema pode (ou não) permitir a existência de uma classe sem a outra?
- No caso dos pacotes e eventos:
 - Uma lista de eventos dentro de cada pacote
 - Sem restrição de que um mesmo evento pertença a mais de um pacote
 - Sem restrição de que algum evento (por exemplo, Visita ao Museu Luiz Gonzaga) eventualmente não esteja em nenhum pacote (pelo menos por enquanto)

Implementação

- Uma implementação possível é esta ao lado.
- CUIDADO!
- Estamos incluindo cópias dos objetos Evento Originais
- Se mudamos o objeto “original” a mudança não se refletirá a cópia

```
class Evento {  
private:  
    int duracao;  
public:  
    Evento(int d) {  
        duracao = d;  
    }  
  
};  
  
class Pacote {  
private:  
    string nome;  
    vector<Evento> eventos;  
public:  
    Pacote(string n) {  
        nome = n;  
    }  
    void adicionaEvento(Evento evento) {  
        eventos.push_back(evento);  
    }  
};
```



Residência
em Software

- Execute o programa ao lado
- O que será exibido?
- Porque?

```
int main () {
    Pacote p1("Pacote 1");
    Pacote p2("Pacote 2");
    Evento e1(1);
    Evento e2(2);
    Evento e3(3);
    Evento e4(4);
    p1.adicionaEvento(e1);
    p1.adicionaEvento(e2);
    p2.adicionaEvento(e2);
    p2.adicionaEvento(e3);
    p1.listaEventos();
    p2.listaEventos();
    e2.setDuracao(10);
    p1.listaEventos();
    p2.listaEventos();
}
```

```
class Evento {
private:
    int duracao;
public:
    Evento(int d) {
        duracao = d;
    }
    int getDuracao() {
        return duracao;
    }
    void setDuracao(int d) {
        duracao = d;
    }
};

class Pacote {
private:
    string nome;
    vector<Evento> eventos;
public:
    Pacote(string n) {
        nome = n;
    }
    void adicionaEvento(Evento evento) {
        eventos.push_back(evento);
    }
    void listaEventos() {
        cout << "Eventos de " << nome << endl;
        for (Evento e: eventos) {
            cout << e.getDuracao() << endl;
        }
    }
};
```

Usando ponteiros

- Precisaremos manter uma lista de ponteiros de eventos
 - `vector<Evento*> eventos;`
- E, portanto, adicionaremos um ponteiro para um evento quando necessário
 - `void adicionaEvento(Evento *evento) { ... }`
- Assim, quando quisermos percorrer todos os eventos, criamos um iterator adequado: iterando para um ponteiro
 - `for (Evento* e: eventos) { ... }`
- Finalmente, mandamos um endereço de um Evento (e não o evento em si) para ser acrescentado.
 - `Evento e1(1);`
 - `p1.adicionaEvento(&e1);`



Residência
em Software

```
int main () {
    Pacote p1("Pacote 1");
    Pacote p2("Pacote 2");
    Evento e1(1);
    Evento e2(2);
    Evento e3(3);
    Evento e4(4);
    p1.adicionaEvento(&e1);
    p1.adicionaEvento(&e2);
    p2.adicionaEvento(&e2);
    p2.adicionaEvento(&e3);
    p1.listaEventos();
    p2.listaEventos();
    e2.setDuracao(10);
    p1.listaEventos();
    p2.listaEventos();
}
```

```
class Pacote {
private:
    string nome;
    vector<Evento*> eventos;
public:
    Pacote(string n) {
        nome = n;
    }
    void adicionaEvento(Evento *evento) {
        eventos.push_back(evento);
    }
    void listaEventos() {
        cout << "Eventos de "
            << nome << endl;
        for (Evento* e: eventos) {
            cout << e->getDuracao() << endl;
        }
    }
};
```

Como fica (teste!)

Exercício (este aqui foi by ChatGPT (adaptado) !!!)

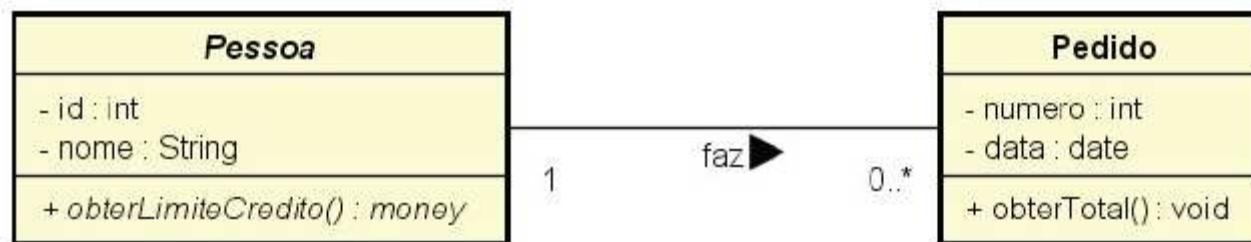
- Crie duas classes em C++: Aluno e Turma.
 - A classe Aluno deve ter os seguintes atributos:
 - Nome do aluno (uma string).
 - Número de identificação do aluno (um inteiro).
 - Crie um construtor para a classe Aluno que permita inicializar os atributos.
 - Crie um método exibirDetalhes() na classe Aluno que exiba os detalhes do aluno (nome e número de identificação).
- Classe Turma:
 - A classe Turma deve ter um vector de objetos da classe Aluno.
- Crie um método adicionarAluno() na classe Turma que permita adicionar um aluno à coleção.
- Crie um método listarAlunos() na classe Turma que liste todos os alunos da turma, exibindo seus detalhes usando o método exibirDetalhes() da classe Aluno.
- Crie uma função main() pra testar tudo isso

Finalmente... associação

- Dominando os conceitos de Composição e Agregação é fácil entender uma associação
 - É quando um objeto de uma classe se associa (ou se relaciona) com objetos de outra classe.
- Cardinalidade
 - 1 para 1: um objeto de uma classe A se associa a um e exatamente um objeto de uma classe B
 - Ex. Pagamento (com cartão) e Autorização (do Banco)
 - 1 para muitos: é quando um objeto de uma classe A se associa a vários objetos de uma classe B, mas cada objeto de B se associa a apenas um da classe A
 - Ex. Professor e Turmas
 - Muitos para muitos: é quando um objeto de uma classe A se associa a vários objetos de uma classe B, e também um objeto de uma classe B se associa a vários objetos da classe A,
 - Ex. Alunos e Turmas

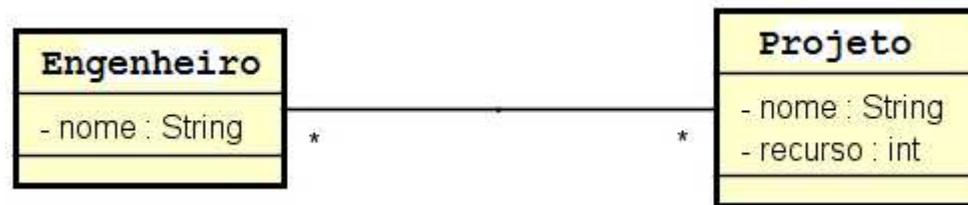
Opcionalidade

- Além da cardinalidade (quantos objetos estão associados) também devemos considerar a optionalidade: a associação é uma necessidade para a existência do objeto (como na composição) ou não.
- Exemplo (pessoa pode ter de 0 a muitos pedidos. Pedido tem que ter uma pessoa que o solicitou)



Outro exemplo

- Um engenheiro pode estar em vários (possivelmente nenhum) projeto e um projeto pode ter vários (possivelmente nenhum) engenheiro



Implementação

- Tipicamente as relações de 1 para muitos são implementadas em objetos de coleção
 - Vector, por exemplo
- Relações de 1 para 1 são implementadas em atributos singulares
- Relações de 1 para muitos são mais complexas.
 - Uma coleção em cada classe
 - Um objeto intermediário implementando a associação



Exemplos

- Note que foi necessário uma declaração (assinatura) da classe Projeto antes de sua implementação

```
class Projeto;

class Engenheiro {
    private:
        vector<Projeto*> projetos;
    public:
        void adicionaProjeto(Projeto *projeto) {
            ...
            projetos.push_back(projeto);
        }
};

class Projeto {
    private:
        vector<Engenheiro*> engenheiros;
    public:
        void adicionaEngenheiro(Engenheiro *engenheiro) {
            ...
            engenheiros.push_back(engenheiro);
        }
};
```



Residência
em Software

Exemplos

- Note que foi necessário uma criar uma classe em que cada objeto dela representa uma associação entre um engenheiro e um projeto

```
class EngenheiroProjeto {  
    private:  
        Projeto* projeto;  
        Engenheiro* engenheiro;  
    public:  
        void associa(Engenheiro *eng,  
                     Projeto *proj) {  
            projeto = proj;  
            engenheiro = eng;  
        }  
        int main () {  
            vector<EngenheiroProjeto> engs_projs;  
            EngenheiroProjeto ass1, ass2, ass3;  
            Engenheiro e1;  
            Engenheiro e2;  
            Projeto p1;  
            Projeto p2;  
            ass1.associa(&e1, &p1);  
            ass2.associa(&e1, &p2);  
            ass3.associa(&e2, &p1);  
            engs_projs.push_back(ass1);  
            engs_projs.push_back(ass2);  
            engs_projs.push_back(ass3);  
        }  
}
```