

RESOLUÇÃO COMENTADA

Prova T14 — Mitológico

Implementação do diagrama UML, classe por classe

1

Mito (abstrata)

2

Godzilla

3

Alado

4

KingKong

5

Ilha + main

Como as classes se relacionam

Setas brancas (▷)

→ herança (Godzilla, KingKong herdam de Mito; Alado herda de Godzilla)

Linhas verdes

→ composição (Ilha contém Godzilla, Alado e ponteiros para KingKongs)

Mito «abstrata»

Godzilla

KingKong

Alado

Ilha «principal»

PACOTE

Mitologico →
namespace

Mito — atributos e o contador estático

Mito é abstrata (lutar é virtual puro). Tem id único alimentado por um contador static, e vivo (default true).

```
class Mito {  
    private:  
        static int cont;           // declaração do static  
        short int id;  
    protected:  
        bool vivo;  
        void informar() const {  
            cout << "Sou o Mito de id " << id << endl;  
        }  
};  
int Mito::cont = 0; // definição (sempre FORA da classe)
```

static

cont está DECLARADO dentro e DEFINIDO fora. Sem essa linha extra, o linker reclama de "undefined reference".

protected

vivo e informar() são protected — visíveis em Godzilla e KingKong, escondidos do mundo exterior.

const

informar() é const → não modifica o objeto e pode ser chamado mesmo via referência const.

Mito — construtora, destrutora e lutar()

```
class Mito {  
    public:  
        // Construtora – id alimentado por cont++ (pós-incremento)  
        Mito(const bool v = true) : id(cont++), vivo(v) { }  
        // Destrutora VIRTUAL – obrigatória em hierarquias polimórficas  
        virtual ~Mito() { vivo = false; }  
        const int getId() const { return id; }  
        void setVivo(const bool v) { vivo = v; }  
        const bool getVivo() const { return vivo; }  
        // VIRTUAL PURO ⇒ Mito é abstrata, não instanciável  
        virtual void lutar(Mito* pm) = 0;  
};
```

POR QUE A DESTRUTORA TEM DE SER virtual?

Em Ilha::~Ilha, fazemos delete pkings[i] onde pkings[i] é KingKong*. Sem virtual em ~Mito, só ~KingKong rodaria — bugs e vazamentos. Com virtual, a chamada é resolvida em runtime.

Godzilla — atributos com dependência na lista

idade entre 0 e 1000 (sorteada na construtora). irritado depende de idade — por isso a ORDEM na lista de inicialização importa.

```
class Godzilla : public Mito {  
    private:  
        short int idade;  
        short int irritado;  
    public:  
        Godzilla(const bool v = true) :  
            Mito(v),           // 1) construtora da BASE  
            idade(rand() % 1001), // 2) sorteia 0..1000  
            irritado(1000 - idade) // 3) usa idade já inicializada  
        { }  
        ~Godzilla() { idade = -1; irritado = -1; }  
        const int getIdade() const { return idade; }  
        const int getIrritado() const { return irritado; }  
};
```

ATENÇÃO À ORDEM

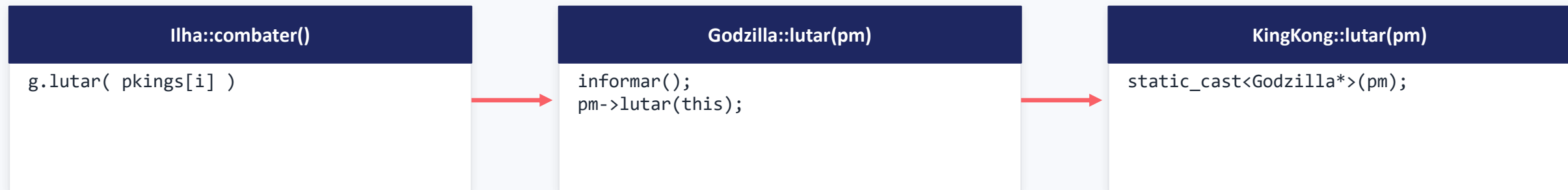
Na lista de inicialização, irritado(1000 - idade) só funciona porque idade aparece ANTES dele. C++ inicializa atributos na ordem em que estão DECLARADOS na classe — ainda assim, escreva-os na lista nessa mesma ordem para evitar warnings.

Godzilla::lutar — a ponte do polimorfismo

Esse método é a "ponte" do polimorfismo: o Godzilla chama o lutar do KingKong passando a si mesmo como Mito*. O KingKong então faz upcast para Godzilla*.

```
void Godzilla::lutar(Mito* pm)
{
    informar();          // método protected herdado de Mito
    if (pm) {
        pm->lutar(this); // chama KingKong::lutar(Mito*)
    } else {
        cout << "Godzilla::lutar - ponteiro nulo!" << endl;
    }
}
```

Fluxo da chamada



Alado — herda de Godzilla, idoso se idade > 700

Alado é Godzilla. idoso é true quando a idade ultrapassa 700. Se for idoso, NÃO luta. operator> compara com outro Godzilla.

```
class Alado : public Godzilla {
private:
    bool idoso;
public:
    Alado(const bool v = true) :
        Godzilla(v),           // idade já está sorteada após esta linha
        idoso(getIdade() > 700) // agora podemos consultar idade
    { }
    ~Alado() { }
    void lutar(Mito* pm) {
        if (!idoso) Godzilla::lutar(pm); // reaproveita o lutar da BASE
    }
    const bool operator>(Godzilla& gz) const {
        return getIdade() > gz.getIdade();
    }
};
```

operator> ACEITA Godzilla& — POR QUÊ?

Porque Alado É-UM Godzilla (herança pública). No main: a > g compara um Alado com um Godzilla — o parâmetro precisa ser do tipo Godzilla& para aceitar tanto um Godzilla quanto qualquer derivada (incluindo outro Alado).

KingKong — atributo static e líder do bando

kings é static: vetor compartilhado por TODOS os KingKongs, contendo apenas os vivos. lider é true para o primeiro criado (quando kings ainda está vazio).

```
class KingKong : public Mito {
private:
    static vector<KingKong*> kings;    // só os vivos
    bool lider;
public:
    KingKong(const bool v = true) :
        Mito(v),
        lider(kings.empty())    // é líder se nenhum existia antes
    {
        if (vivo) considerar(this);    // se vivo, entra no static
    }
    ~KingKong() { }
    static void considerar(KingKong* pk) {
        if (pk) kings.push_back(pk);
    }
};
vector<KingKong*> KingKong::kings;    // definição (FORA da classe)
```

MÉTODO static

considerar() não usa this — só manipula o atributo static kings. Pode ser chamado como KingKong::considerar(p) sem precisar de objeto. Aqui, a construtora chama via this para se auto-incluir no vetor.

KingKong::lutar — a regra de combate

Recebe um Mito* (que é Godzilla na prática) e aplica as regras do enunciado.

1

Pré-condições

pm não nulo, este KingKong vivo, e o Godzilla apontado também vivo. Senão, sai.

2

Há combate?

Só luta se idade do Godzilla < 500 E irritado > 100. Caso contrário, nada acontece.

3

Mais que 2 KKs

Bando grande domina: pG->setVivo(false) — Godzilla é neutralizado.

4

Menos que 3 KKs

Bando pequeno apanha: cada não-líder morre; líder tem 50% de chance. Vetor é limpo.

Trecho-chave: "limpar os mortos" do vetor static (padrão while + erase)

```
vector<KingKong*>::iterator it = kings.begin();
while (it != kings.end()) {
    if (!(*it)->getVivo())
        it = kings.erase(it);    // erase devolve o próximo iterador válido
    else
        it++;
}
```

KingKong::lutar — o código todo

```
void KingKong::lutar(Mito* pm) {
    informar();
    if (!pm) { cout << "ponteiro nulo!"; return; }
    if (!vivo) return;
    Godzilla* pG = static_cast<Godzilla*>(pm); // upcast nem precisa, pra deixar explícito
    if (!pG->getVivo()) return;
    if (pG->getIdade() < 500 && pG->getIrritado() > 100) {
        int tam = (int) kings.size();
        if (tam > 2) {
            pG->setVivo(false); // Godzilla apanha
        } else {
            for (int i = 0; i < tam; i++) {
                if (kings[i]->lider) {
                    if ((rand() % 2) == 0) kings[i]->setVivo(false);
                } else {
                    kings[i]->setVivo(false);
                }
            }
            // (segue a limpeza com while+erase do slide anterior)
        }
    }
}
```

Ilha — a classe principal

Ilha tem composição: um Godzilla, um Alado e um vetor de 5 PONTEIROS para KingKong (criados dinamicamente).

```
#define MAX 5
class Ilha {
private:
    Godzilla    g;
    Alado       a;
    KingKong*   pkings[MAX];
    void criarKings(const short int x) {
        for (int i = 0; i < MAX; i++)
            if ((rand() % 100) < x)
                pkings[i] = new KingKong();
    }
public:
    Ilha(const short int x = 50) : g(), a(), pkings() {
        for (int i = 0; i < MAX; i++) pkings[i] = NULL;
        criarKings(x);
    }
};
```

DICA

pkings() na lista zera o array; o for de NULL é defesa redundante mas pedida em estilo.

Ilha::~~Ilha() e Ilha::combater()

Destrutora — padrão dos 3 passos

```
~Ilha() {
    for (int i = 0; i < MAX; i++) {
        if (pkings[i]) {          // 1) verifica
            delete pkings[i];     // 2) libera
            pkings[i] = NULL;     // 3) anula
        }
    }
}
```

POR QUE delete?

pkings[i] foi alocado com new em criarKings — TEM de ter delete correspondente, senão vazamento.

combater() — laço + comparação final

```
void combater() {
    for (int i = 0; i < MAX; i++) {
        if (pkings[i]) {
            g.lutar(static_cast<Mito*>(pkings[i]));
            a.lutar(static_cast<Mito*>(pkings[i]));
        }
    }

    if (a > g) cout << "a > g" << endl;
    else      cout << "a <= g" << endl;
}
```

POR QUE static_cast<Mito*>?

lutar() recebe Mito*. KingKong* é Mito* por herança — o cast deixa explícito o upcast pedido na prova.

a > g — QUEM CHAMA O operator>?

Como > foi sobrecarregado em Alado (a é Alado, g é Godzilla), C++ resolve a chamada como a.operator>(g). Funciona porque o parâmetro é Godzilla& e a herança permite passar referências de derivadas.

main() e o namespace Mitologico

O enunciado fixa: int main() { Ilha perdida(70); perdida.combater(); return 0; }

```
namespace Mitologico {  
    class Mito    { ... };  
    class Godzilla { ... };  
    class Alado   { ... };  
    class KingKong { ... };  
    class Ilha    { ... };  
}  
using namespace Mitologico;  
int main() {  
    sementear();  
    Ilha perdida(70);  
    perdida.combater();  
    return 0;  
}
```

ATENÇÃO

O enunciado da T14 explicita: "O pacote precisa se tornar namespace". A T16, ao contrário, marca "Pacote que NÃO se torna namespace".

CHECKLIST

Antes de entregar a sua resolução

- | | |
|---|--|
| ✓ Mito tem destrutora virtual e método lutar virtual puro (= 0) | ✓ <code>static int Mito::cont</code> definido FORA da classe |
| ✓ id alimentado por <code>cont++</code> na lista de inicialização | ✓ Godzilla tem idade entre 0 e 1000, irritado = 1000 - idade |
| ✓ Lista de Godzilla na ordem: Mito(v) → idade → irritado | ✓ Alado herda de Godzilla; idoso só inicializado APÓS Godzilla(v) |
| ✓ <code>Alado::lutar</code> reaproveita <code>Godzilla::lutar</code> quando lidoso | ✓ <code>operator></code> de Alado tem assinatura <code>Godzilla&</code> e é const |
| ✓ <code>static vector<KingKong*> kings</code> definido FORA da classe | ✓ <code>lider = kings.empty()</code> — primeiro KingKong é o líder |
| ✓ Construtora de KingKong se auto-inclui em kings via considerar | ✓ <code>KingKong::lutar</code> trata pm nulo, !vivo e adversário não-vivo |
| ✓ Limpeza dos mortos: <code>while + erase</code> (it++ apenas no else) | ✓ <code>Ilha::~Ilha</code> desaloca <code>pkings[i]</code> e zera o ponteiro |
| ✓ <code>Ilha::combater</code> usa <code>static_cast<Mito*></code> nos upcasts | ✓ Pacote Mitologico virou namespace; using namespace antes do main |