Um computador faz exatamente o que você manda ele fazer, mas isso pode ser muito diferente do que você gostaria que ele fizesse.

Classes e funções amigas

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Atualizando os objetos associados

Na aula passada foi solicitado que:

- Ao modificar a sala de aula de uma Disciplina, a SalaAula fosse informada disso automaticamente.
- Opcional: Ao adicionar uma Disciplina em uma SalaAula, a disciplina fosse informada disso automaticamente.

Opções

No momento temos duas opções:

1. Os sets e afins não comunicam as alterações para os objetos associados (ou agregados).
   Nesse caso, quem usar as classes terá a responsabilidade de manter tudo consistente.

2. Os sets e afins comunicam os objetos associados (ou agregados) as suas alterações.

Vantagens? Desvantagens?

Opções

1. Os sets e afins não comunicam as alterações para os objetos associados (ou agregados).
   + Baixo overhead;
   - O programador agora deve conhecer mais detalhes, e cuidar deles.
     Se não tomar cuidado, pode facilmente deixar o sistema inconsistente.
2. Os sets e afins comunicam os objetos associados (ou agregados) as suas alterações.
   + Transparência: As classes gerenciam suas relações, diminuindo a responsabilidade do programador;
   + Menor chance de gerar inconsistências.
   - Overhead: As lógicas envolvidas podem ser complexas e custar caro para a máquina.

Terceira opção

As duas opções anteriores **são válidas**.

Se são aplicáveis ou não, depende do problema.

Estudaremos uma terceira opção, onde o relacionamento ainda é bidirecional, mas as alterações só podem ser feitas a partir de um lado do relacionamento.

Classes e funções amigas

Uma função ou classe **amiga pode acessar todos os membros da classe.**

Membros privados, protegidos ou públicos.

Classes e funções amigas

A partir de um classe A, podemos declarar que:

Existe outra classe B que é amiga:

```
friend class B;
```

Nesse caso, todas as funções membro de B podem acessar todos os membros de A.

Classes e funções amigas

A partir de um classe A, podemos declarar que:

Existe uma função global que é amiga.

```
friend tipoRetorno nomeFuncao(parametros);
```

Essa função (que não pertence a nenhuma classe específica) pode acessar todos os membros de A.

Classes e funções amigas

A partir de um classe A, podemos declarar que:

Existe uma função membro de determinada classe que é amiga:

```
friend tipoRetorno
NomeClasseAmiga::nomeFuncaoClasse(parametros);
```

A função nomeFuncaoClasse da classe NomeClasseAmiga pode acessar todos os membros da classe A.

Boas Práticas

Os amigos da classe devem ser os primeiros itens declarados na classe, e sem modificadores de acesso.

Exemplo

Usar o conceito de amizade para tornar as classes `Disciplina` e `SalaAula` mais autossuficientes.

Remova as funções membro `adicionarDisciplina` e `removerDisciplina`.

Membros externos da classe só podem listar as disciplinas que são ministradas na Sala de Aula.

Exemplo

Definir que a função setSalaAula da classe Disciplina pode acessar os membros privados (e públicos e protegidos) da classe SalaAula.

SalaAula.hpp

```
class SalaAula{
   friend void Disciplina::setSalaAula(SalaAula* salaAula);

   public:
       SalaAula(std::string nome, unsigned int capacidade);

       std::string getNome();
       void setNome(std::string nome);

       unsigned int getCapacidade();
       void setCapcidade(unsigned int capacidade);

       std::list<Disciplina*>& getDisciplinas();
   private:
       std::string nome;
       unsigned int capacidade;
       std::list<Disciplina*> disciplinasMinistradas;
};
```

Exemplo

Disciplina.cpp

```
Disciplina::Disciplina(std::string nome)
   :nome{nome},sala{nullptr} {
}

Disciplina::Disciplina(std::string nome, SalaAula* sala)
       :Disciplina{nome} {
   this->setSalaAula(sala);
}

void Disciplina::setSalaAula(SalaAula* sala){
   if(this->sala != nullptr)//se já existia uma sala, remover a disciplina dessa sala
       this->sala->disciplinasMinistradas.remove(this);
   this->sala = sala;
   if(this->sala != nullptr)
       this->sala->disciplinasMinistradas.push_back(this);//adicionar a disciplina na nova sala
}

//...
```

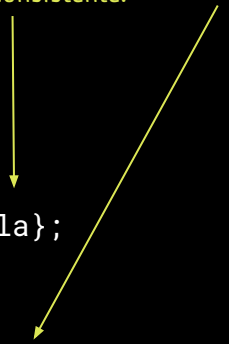
Exemplo

```cpp
#include <iostream>
#include <list>

#include "Pessoa.hpp"
#include "Disciplina.hpp"
#include "SalaAula.hpp"

#include<list>

int main(){
   SalaAula sala{"Lab Info 1", 20};

   Pessoa prof1{"Joao", 11111111111, 40};
   Disciplina dis1{"Orientacao a Objetos", &sala};
   dis1.setProfessor(&prof1);

   Pessoa prof2{"Maria", 22222222222, 30};
   Disciplina dis2{"Sistemas Operacionais", &sala};
   dis2.setProfessor(&prof2);

   std::cout << dis1.getSalaAula()->getNome() << '\n';
   std::list<Disciplina*> dis{sala.getDisciplinas()};
   std::list<Disciplina*>::iterator it{dis.begin()};
   for( ; it != dis.end(); it++)
       std::cout << (*it)->getNome() << '\n';

   return 0;
}
```

Exemplo

```
#include <iostream>
#include <list>

#include "Pessoa.hpp"
#include "Disciplina.hpp"
#include "SalaAula.hpp"

#include<list>

int main(){
   SalaAula sala{"Lab Info 1", 20};

   Pessoa prof1{"Joao", 11111111111, 40};
   Disciplina dis1{"Orientacao a Objetos", &sala};
   dis1.setProfessor(&prof1);

   Pessoa prof2{"Maria", 22222222222, 30};
   Disciplina dis2{"Sistemas Operacionais", &sala};
   dis2.setProfessor(&prof2);

   std::cout << dis1.getSalaAula()->getNome() << '\n';
   std::list<Disciplina*> dis{sala.getDisciplinas()};
   std::list<Disciplina*>::iterator it{dis.begin()};
   for( ; it != dis.end(); it++)
       std::cout << (*it)->getNome() << '\n';

   return 0;
}
```

A disciplina vai automaticamente atualizar a lista da sala de aula, se adicionando nessa lista de forma transparente. O programador não tem acesso direto a lista de disciplinas de sala de aula, evitando que ele possa deixar o sistema inconsistente.



Exemplo

Agora o programador não pode acessar as funções para adicionar ou remover disciplinas na sala de aula.

Mas ele ainda pode `setar` a sala de aula das disciplinas.

As disciplinas se encarregam de atualizar a sala de aula automaGicamente.

Tudo isso sem a utilização de lógicas sofisticadas.

Baixo overhead.

Algumas propriedades da amizade

Para que a classe (ou função) B seja amiga de A, **a classe A precisa explicitamente declarar que B é sua amiga.**

Algumas propriedades da amizade

Para que a classe (ou função) B seja amiga de A, **a classe A precisa explicitamente declarar que B é sua amiga.**

A amizade **não é simétrica.**

Se B é amiga de A, não significa que A é amiga de B.

Algumas propriedades da amizade

Para que a classe (ou função) B seja amiga de A, **a classe A precisa explicitamente declarar que B é sua amiga.**

A amizade **não é simétrica.**

Se B é amiga de A, não significa que A é amiga de B.

Amizade **não é transitiva**.

Se A é amiga de B, e B é amiga de C, não podemos inferir que A é amiga de C.

Algumas propriedades da amizade

Para que a classe (ou função) B seja amiga de A, **a classe A precisa explicitamente declarar que B é sua amiga.**

A amizade **não é simétrica.**

Se B é amiga de A, não significa que A é amiga de B.

Amizade **não é transitiva**.

Se A é amiga de B, e B é amiga de C, não podemos inferir que A é amiga de C.

Amizade **não é herdada.**

"Os filhos dos seus amigos não são seus amigos".

Veremos herança no futuro.

Ah, a amizade...

Se você acha que a amizade é uma coisa linda que deve ser cultivada, reveja seus conceitos.

Pelo menos na orientação a objetos.

Amizade é um conceito que deve ser usado com cautela.

Quando usar

Sempre pondere antes de usar o conceito de amizade.

Muitas vezes podemos resolver o problema sem usar classes amigas.

Quais problemas as classes amigas podem trazer?

Quando usar

Sempre pondere antes de usar o conceito de amizade.

- Muitas vezes podemos resolver o problema sem usar classes amigas.

Quais problemas as classes amigas podem trazer?

- Quebra de encapsulamento.
  - Para que definir dados e funções privadas, se populamos o programa com classes que ignoram as regras de acesso?
- Aumento de acoplamento.
- Pode tornar a manutenção mais difícil.

Quando usar

C++ assume que os Cientistas/Engenheiros são alfabetizados e bem alimentados!

São capazes de tomar decisões de projeto sobre quando e como usar determinados recursos.

Veja o comentário de Bjarne Stroustrup sobre a possível quebra de encapsulamento gerada pelas classes amigas:

www.stroustrup.com/bs_faq2.html#friend

Com grandes poderes vêm grandes responsabilidades!

Friendship em outras linguagens

A única linguagem O.O. que conheço que implementa o conceito de classes amigas é o C++.

Outras linguagens preferem não implementar, principalmente para evitar quebras de encapsulamento que podem ser perigosas.

Assumem que o programador não é alfabetizado e bem alimentado e vai fazer besteira.

Friendship em outras linguagens

A falta do conceito de amizade em muitas linguagens fazem com que elas precisem implementar outros mecanismos, já que em alguns (raros) momentos, precisamos dessa “quebra de encapsulamento”.

Exemplo: como funciona a visibilidade protected no Java?

Friendship em outras linguagens

Em Java, por exemplo, a visibilidade protected se estende para todas as classes do mesmo pacote, e não apenas para as suas classes filhas.

Gera alguns problemas:

- A quebra de encapsulamento se torna compulsória para todos os membros protegidos da classe.
- Muitos programadores não se dão conta que as classes do mesmo pacote podem acessar os itens protegidos da classe.

Templates e Sobrecarga de Operadores

O conceito de amizade pode se tornar ainda mais poderoso (e perigoso) quando utilizado em combinação com Templates e sobrecarga de operadores.

Veremos adiante na disciplina.

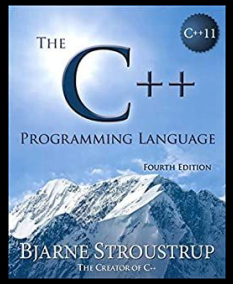
Exercícios

1. Modifique o exemplo da aula para que seja possível para o programador modificar a sala de aula de uma disciplina, e também seja possível adicionar uma disciplina em sala de aula. Suas classes devem se encarregar de manter tudo consistente. Para evitar lógicas complexas, utilize o conceito de classes amigas, onde a sala de aula é amiga de disciplina, e vice-versa (dica: faça isso para alterar diretamente os dados membro, sem precisar passar por validações nos sets). Exemplo:

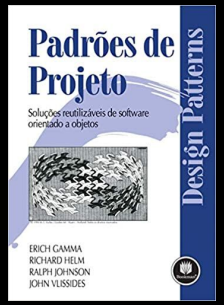
```
int main(){
      SalaAula sala{"Lab Info 1", 20};
      SalaAula sala2{"Lab Info 2", 40};

      Disciplina dis1{"Orientacao a Objetos", &sala};
      Disciplina dis2{"Sistemas Operacionais", &sala};

      dis1.setSala(&sala2); //isso deve ser possível
      sala2.adicionarDisciplina(&dis2); // isso também
      //...
}
```

Referências

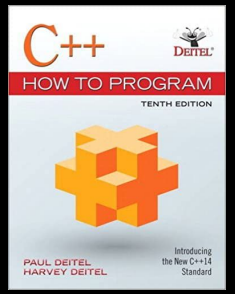
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



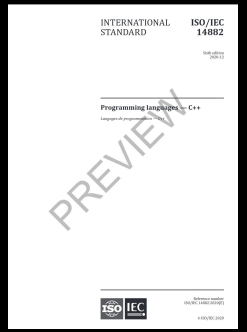
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



Deitel, H. M., Deitel, P. J. C++: como programar. 5a ed. Pearson Prentice Hall. 2006.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en



Licença