Padrão Relacionante

Bruno Pera 3304821
Dairton L. Bassi Filho 3346871

Intenção

Prover relacionamentos entre dois ou mais objetos, mantendo os relacionamentos consistentes após a destruição dos mesmos.

Motivação

Em muitos sistemas os objetos participantes precisam se relacionar, porém não é possível saber de antemão quantos e quais serão os relacionamentos nem o grau e a duração dos mesmos.

Por exemplo, considere um sistema que gerencia uma biblioteca, é necessário modelar o relacionamento entre livros e usuários para indicar quais os livros estão emprestados para cada usuário. Uma possível solução para este problema seria colocar uma lista de livros na classe usuário, no entanto se o usuário precisar relacionar-se com objetos de outro tipo (que não o livro) podem ser necessárias modificações na classe usuário.

Uma solução melhor é desvincular qualquer tipo de relacionamento da classe usuário criando uma classe para lidar especificamente com a relação entre os objetos. Este é o padrão Relacionante.

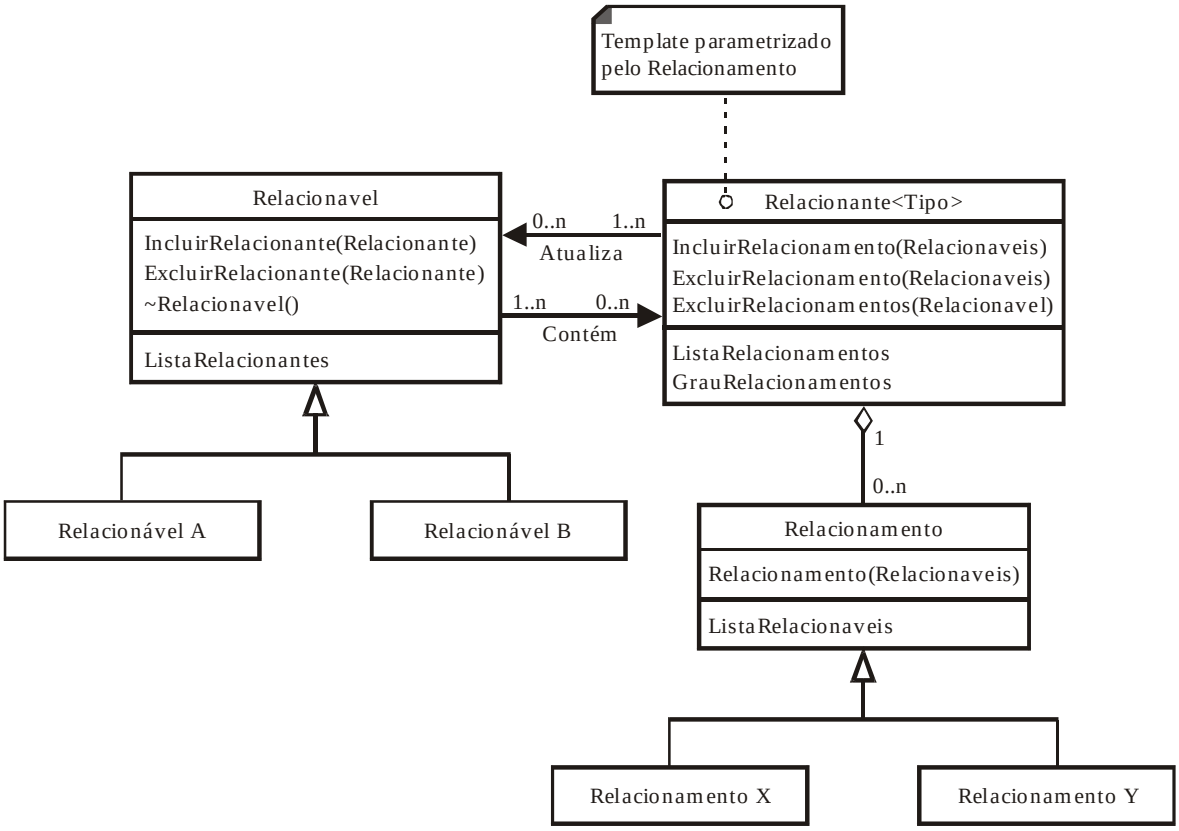
Usando o padrão Relacionante é possível criar os mesmos relacionamentos criados em banco de dados relacionais, mantendo a integridade de tais relacionamentos.

Aplicabilidade

Use o padrão Relacionante quando:

- for necessária a criação de relações entre dois ou mais objetos.
- Não for possível saber quais e quantas serão as relações entre os objetos.
- Quando as relações ou os objetos participantes são criados e destruídos constantemente em tempo de execução.

Estrutura



Participantes

- **Relacionável**
  - Objeto que participa de algum relacionamento.
  - Mantém referências para todos os Relacionantes que possuam Relacionamentos que o envolva.
  - Antes de ser destruído deve avisar todos os Relacionantes de sua lista, possibilitando manter a integridade dos Relacionamentos.
- **Relacionante**
  - Agrupa todos os Relacionamentos de um mesmo tipo.
  - Responsável por criar novos Relacionamentos, bem como destruir os já existentes.
  - Ao criar um novo Relacionamento ou destruir um já existente informa os Relacionáveis participantes para que esses possam atualizar suas listas de Relacionantes.
  - Impede a criação de dois Relacionamentos iguais e Relacionamentos com referências nulas.
- **Relacionamento**
  - Mantém uma lista de referências a objetos Relacionáveis que participam do Relacionamento em questão.
  - É responsável por assegurar que os Relacionáveis participantes sejam dos tipos corretos.

Colaborações

- Ao ser destruído um Relacionável solicita que todos os Relacionantes de sua lista destruam todos os Relacionamentos que contenham o Relacionável em questão.

- Ao destruir um Relacionamento, um Relacionante pode precisar solicitar que os Relacionáveis envolvidos atualizem suas listas de Relacionantes, isto é, caso o Relacionável não possua mais nenhum Relacionamento, este deve excluir o Relacionante de sua lista.

Conseqüências

- O uso do padrão Relacionante pode aumentar a complexidade de manter os relacionamentos, pois freqüentemente é necessário determinar se um dado relacionamento está ou não presente no Relacionante. Tudo depende da estrutura de dados usada para armazenar os Relacionamentos e do algoritmo usado para realizar buscas.

- Com este padrão os objetos relacionados (Relacionáveis) não precisam ter consciência dos relacionamentos de que participam, assim é possível que os objetos participem de diferentes relacionamentos durante a execução do programa, bem como relacionamentos que não podem ser previstos antes do tempo de execução.

- Diferentes instancias da classe Relacionante separam semanticamente conjuntos distintos de relacionamentos.

- A especialização da classe Relacionamento permite que sejam armazenadas informações adicionais de Relacionamentos específicos, por exemplo, o Relacionamento entre usuário e livro pode conter a data de devolução como um atributo específico.

- Ao especializar a classe Relacionamento pode ser feita a verificação de tipos dos objetos relacionados, tornando assim os Relacionamentos mais robustos e consistentes.

Implementação

A seguir apresentamos alguns tópicos que devem ser considerados durante a implementação:

- Um dos objetivos do padrão é fazer com que a classe Relacionante trate somente de relacionamentos do mesmo tipo, uma maneira relativamente simples de se atingir este objetivo é fazer uso do recurso de tipos parametrizados encontrado em algumas linguagens (chamado de templates em C++). Desse modo ao criarmos uma instância de Relacionante devemos passar a classe especifica de Relacionamento a ser tratada, a partir daí Relacionante só criará Relacionamentos do tipo

previamente especificado, impedindo o agrupamento de relacionamentos de natureza diferente.

- Um Relacionamento é criado por um Relacionante chamando o método IncluirRelacionamento(Relacionaveis), esse método deve registrar o Relacionante em questão na Lista de Relacionantes de todos os Relacionáveis participantes, para tanto basta chamar o método IncluirRelacionante da classe Relacionavel.

- Outro objetivo importante do padrão é manter a integridade dos Relacionamentos, isto é, não permitir a existência de Relacionamentos entre Relacionáveis que já foram destruídos. Para conseguirmos esta funcionalidade devemos incluir código no destrutor da classe Relacionavel para que destrua todos os Relacionamentos de que participa. Para tanto basta chamar ExcluirRelacionamentos(Relacionavel) de todos os Relacionantes presentes na Lista do Relacionável a ser destruído.

- Finalmente se desejarmos criar sub classes de Relacionamento que garantam integridade de tipos, isto é, relacionamentos somente entre tipos específicos de relacionáveis, devemos incluir código para isso (provavelmente usando RTTI) no construtor da classe Relacionável.

Exemplo de código

Para exemplificar vamos codificar em C++ as partes mais importantes das classes envolvidas no padrão. Usaremos vetores e busca seqüencial somente para ilustrar as idéias.

A seguir temos a definição da super classe Relacionavel:

```
class Relacionavel
{
public:
      void IncluirRelacionante(Relacionante* rel_ptr);
      void ExcluirRelacionante(Relacionante* rel_ptr);
      virtual ~Relacionavel();

private:
      std::vector<Relacionante*> ListaRelacionantes;
};
```

Essa classe será especializada para qualquer classe que precise de suporte para relacionamentos, portanto deve conter somente a interface mínima necessária para prover as funcionalidades já mencionadas.

Vejamos agora parte da implementação dessa classe:

```
void Relacionavel::IncluirRelacionante(Relacionante* rel_ptr)
{
      for (int i = 0; i < ListaRelacionantes.size(); i++)
            if (ListaRelacionantes[i] == rel_ptr) return;
```

```
      ListaRelacionantes.push_back(rel_ptr);
}

void Relacionavel::ExcluirRelacionante(Relacionante* rel_ptr)
{
      for (int i = 0; i < ListaRelacionantes.size(); i++)
            if (ListaRelacionates[i] == rel_ptr)
                  ListaRelacionantes.erase(rel_ptr);
}

Relacionavel::~Relacionavel()
{
      for (int i = 0; i < ListaRelacionantes.size(); i++)
            ListaRelacionantes[i].ExcluiRelacionamentos(*this);
}
```

É importante notarmos o destrutor da classe Relacionavel, pois este é responsável por excluir todos os relacionamentos de que o objeto participa, para isso chama o metodo ExcluiRelacionamentos de todos os Relacionantes em sua lista.
Vejamos agora a definição da classe Relacionamento.

```
class Relacionamento
{
public:
      Relacionamento(std::vector<Relacionavel*> Relacionaveis);
      std::vector<Relacionamento*> obterRelacionaveis(void) const;

private:
      std::vector<Relacionavel*> ListaRelacionaveis;
};
```

Basicamente esta classe contém uma lista de referências para Relacionáveis, bem como métodos para manipula-los. Podemos ainda herdar essa classe para criarmos relacionamentos específicos que incluam outros dados.
A seguir temos a implementação de tais métodos:

```
Relacionamento::Relacionamento(std::vector<Relacionavel*> Relacionaveis)
{
      for (int i = 0; i < Relacionaveis.size(); i++)
            ListaRelacionaveis.push_back(Relacionaveis[i]);
}

Relacionamento::std::vector<Relacionamento*> obterRelacionaveis(void)
const
{
      return ListaRelacionamentos;
}
```

Finalmente vejamos a definição da classe Relacionante:

```
template<class Relacionamento>
class Relacionante
{
public:
   Relacionante(int grauRelacionamentos);
   bool IncluirRelacionamento (std::vector<Relacionavel*> Relacionaveis);
   bool ExcluirRelacionamento (std::vector<Relacionavel*> Relacionaveis);
```

```
   void ExcluirRelacionamentos(Relacionavel* O_Relacionavel);
   std::vector<Relacionamento*> obterRelacionamentos(void) const;
   ~Relacionante();

private:
   int GrauRelacionamento;
   std::vector<Relacionamento*> ListaRelacionamentos;
};
```

Aqui notamos o uso do recurso de template da linguagem C++, como já dito anteriormente isso é feito para que possamos assegurar que um dado Relacionante possa criar instâncias de Relacionamentos específicos.

```
Relacionante(int grauRelacionamentos)
{
      GrauRelacionamentos = grauRelacionamentos;
}

bool IncluirRelacionamento (std::vector<Relacionavel*> Relacionaveis)
{
   for (int i = 0; i < ListaRelacionamentos.size(); i++)
      if (ListaRelacionamentos[i]->obterRelacionaveis == Relacionaveis)
         return (false);

   ListaRelacionaveis.push_back(new Relacionamento(Relacionaveis));
   for (int i = 0; i < Relacionaveis.size(); i++)
      Relacionaveis[i] -> IncluirRelacionante(*this);

   return (true);
}

bool ExcluirRelacionamento (std::vector<Relacionavel*> Relacionaveis)
{
   for (int i = 0; i < ListaRelacionamentos.size(); i++)
      if (ListaRelacionamentos[i]->obterRelacionaveis == Relacionaveis)
      {
         ListaRelacionamentos.erase(ListaRelacionamentos[i]);
         /* para cada um dos Relacionaveis r que nao
            pertençam a nenhum relacionamento faça:  */
         {
            r->Excluir(Relacionante(*this));
         }
         return (true);
      }
      return (false);
}

void ExcluirRelacionamentos(Relacionavel* O_Relacionavel)
{
   for (int i = 0; i < ListaRelacionamentos.size(); i++)
      if (O_Relacionavel in ListaRelacionamentos[i]->obterRelacionaveis)
      ExcluirRelacionamento(ListaRelacionamentos[i]->obterRelacionaveis);
}

std::vector<Relacionamento*> obterRelacionamentos(void) const
{
      return (ListaRelacionamentos);
}

~Relacionante()
{
```

```
   for (int i = 0; i < ListaRelacionamentos[i].size(); i++)
      ExcluirRelacionamento(ListaRelacionamento[i]->obterRelacionaveis);
}
```

No método IncluirRelacionamento devemos primeiramente verificar se tal relacionamento já esta presente no Relacionante, caso não esteja um novo relacionamento do tipo apropriado é criado e adicionado à lista.

No método ExcluirRelacionamento devemos, após excluir o relacionamento, atualizar a lista de Relacionantes de todos os Relacionáveis que participavam do Relacionamento excluído.

O método ExcluirRelacionamentos simplesmente aplica o método ExcluirRelacionamento a todos os relacionamentos que possuam o argumento como um dos Relacionáveis.

Finalmente, antes de ser destruído um objeto Relacionante deve excluir devidamente todos os seus Relacionamentos.

OBS: Note que usamos pseudo código (como o uso de um operador *in* para testar pertinência) em algumas partes para não nos complicarmos com detalhes de busca.

Usos Conhecidos

No semestre passado (01/2002) durante o curso de MAC420 – Introdução a Computação Gráfica foi desenvolvido em C++ um sistema de Ray Tracing, um programa para renderização de imagens tridimensionais a partir da especificação geométrica de superfícies e fontes de luz.

O Ray Tracer relaciona superfícies com fontes de luz indicando quais superfícies são iluminadas por quais fontes de luz, relaciona também superfícies com pigmentos (um conjunto de pigmentos determina a cor de uma superfície). Para modelar estes Relacionamentos foram criadas instancias de Relacionantes para os diferentes tipos de Relacionamento.

Outro exemplo de aplicação foi durante o curso de MAC441 – Programação Orientada a Objetos, no qual foi desenvolvido um sistema de controle de matrículas de alunos e carga didática de professores. Neste sistema ocorrem relacionamentos entre alunos e matriculas, matriculas e turmas, turmas e professores, professores e disciplinas, professores e departamentos e alunos e departamentos. Para todos esses tipos de relacionamentos foi usada uma variante do padrão Relacionante.

Padrões Relacionados

Strategy

Pode ser usado para implementar diferentes algoritmos de busca na classe Relacionante.

Iterator

Pode ser aplicado para percorrer a lista de Relacionamentos da classe Relacionante, independente de como ela foi implementada.

Observer

O sistema de comunicação entre Relacionável e Relacionante pode ser feito através deste padrão.