Flávia M. B. Greggio
Fábio Braga de Oliveira

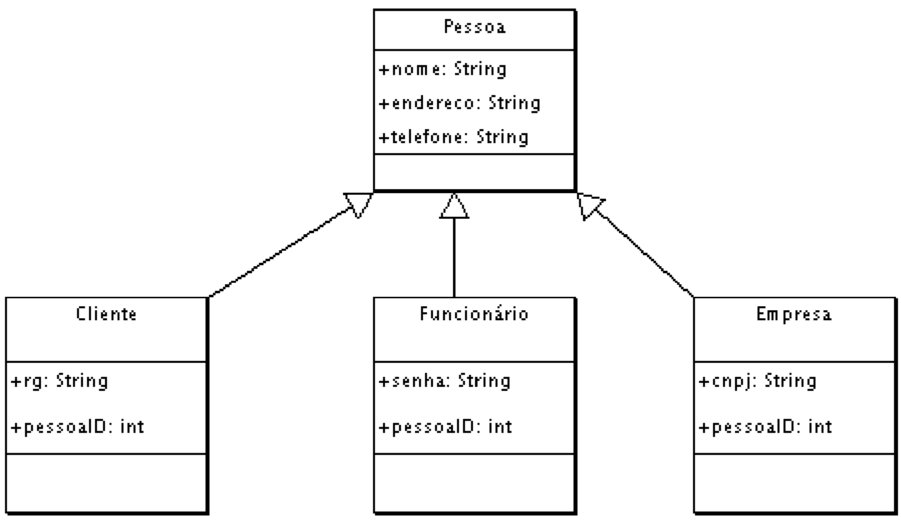
Herança Relacional

Intenção

Define uma forma de criar herança entre tabelas de um banco de dados relacional, de forma a não degradar o desempenho com a sua manipulação.

Motivação

Muitas vezes encontramos durante o projeto de um banco de dados situações similares às encontradas num projeto de software, onde entidades relacionadas parecem sugerir a existência de uma superclasse. De novo, e pelos mesmos motivos de sempre, a duplicação de esquemas de tabelas torna o banco de dados difícil de se realizar manutenção ou alterações, além de propício a informaçãoes inconsistentes. Usando referencias, não é difícil criar uma estrutura superclasse-subclasse. Cria-se uma nova tabela com os campos que aparecem repetidos das tabelas-filhas, e cada tabela-filha ganha uma referencia sua, como na figura:

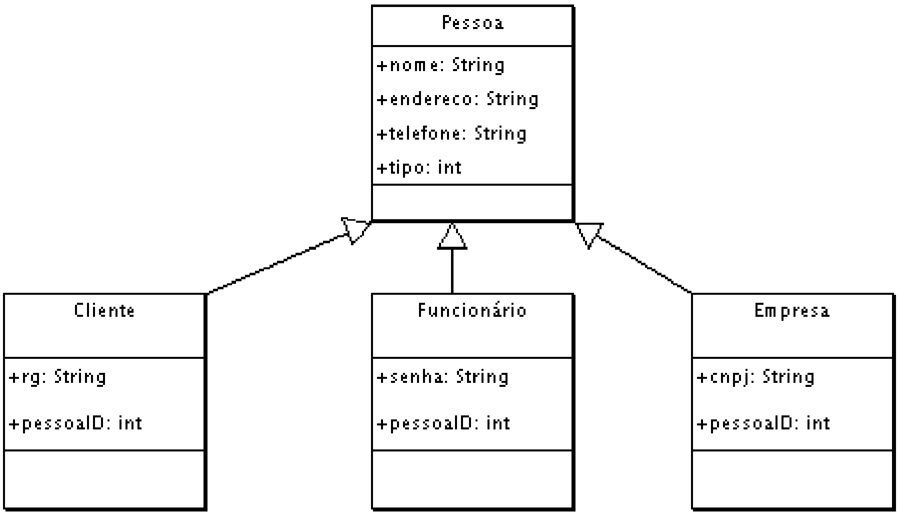


No entanto, esta conveniente estrutura nova contem um sutil problema de desempenho, que para operações críticas pode ser proibitivo. Uma operação de seleção de dados da tabela

Funcionário teria a seguinte forma:

```
SELECT *
FROM Pessoa, Funcionario
WHERE Pessoa.pessoaID = Funcionario.pessoaID
```

Como a tabela Pessoa contém os dados de todas as suas subclasses, o número de combinações necessárias para realizar esta seleção seria muito grande e desnecessário. Se uma empresa tivesse 500 funcionários, 15000 clientes e 500 fornecedores, seria algo em torno de (500 + 15000 + 500) X 500 = 8000000 de comparações! Mas este problema é facilmente contornável, adicionando um campo a tabela Pessoa, que discrimina o tipo de informação que contém em relação as subclasses, para realizar uma prévia filtragem dos dados.



Por exemplo, o campo Pessoa.tipo é um inteiro, que obedece a seguinte codificação: 1 = Cliente, 2 = Funcionário, 3 = Empresa. Agora, o comando de seleção fica:

```
SELECT *
FROM Pessoa, Funcionario
WHERE Pessoa.tipo = 2 AND Pessoa.pessoaID = Cliente.pessoaID
```

Agora, com a adição deste campo, o número de combinações para seleção cai drasticamente, para 500 x 500 = 250000.

Para inserir novos registros nestas tabelas, uma primeira tentativa inocente seria inserí-los começando da superclasse até as subclasses. Mas como a maioria dos bancos de dados dispoêm de linguagens de programação para funções armazenadas (Stored Procedures), a solução mais modular e reutilizável seria implementar funções armazenadas para as subclasses e para a superclasse de forma que a primeira execute a segunda com os devidos parâmetros. Se uma nova tabela fosse inserida na hierarquia, bastava criar seu procedimento armazendo e reaproveitar o código já escrito.

```
CREATE PROCEDURE InserirCliente(nome, endereco, telefone, rg) AS
DECLARE
        ID INTEGER;
BEGIN
        ID := GEN_ID(Cliente);
        INSERT INTO Cliente(pessoaID, rg)
        VALUES (ID, rg);
```

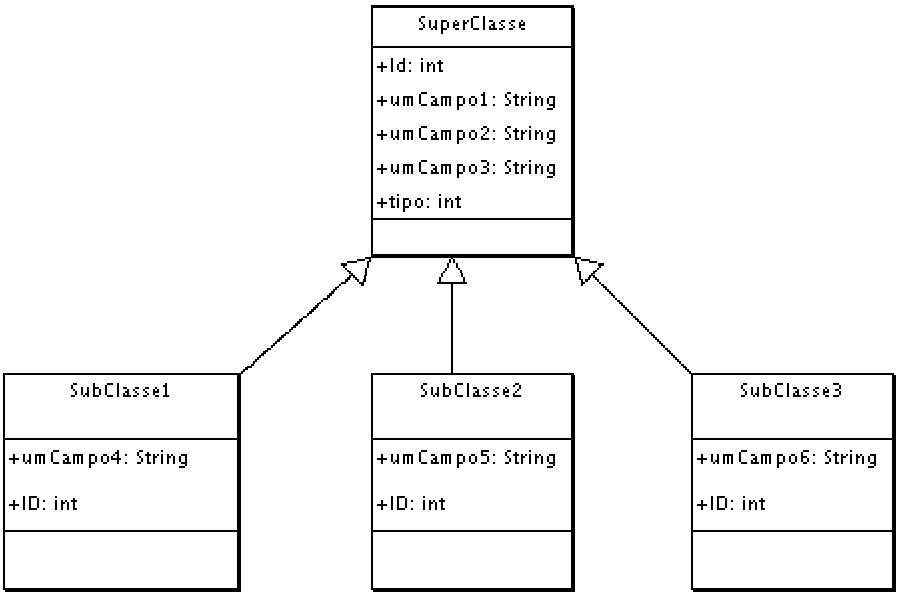
```
		EXEC(InserirPessoa(nome, endereco, telefone, ID));
	END;

	CREATE PROCEDURE InserirPessoa(nome, endereco, telefone, pessoaID) AS
	BEGIN
		INSERT INTO Pessoa(pessoaID, nome, endereco, telefone)
		VALUES (nome, endereco, telefone, pessoaID);
	END;
```

A alteração da tabela segue o mesmo princípio.

Aplicação

Quando se tem muitas entidades no banco de dados com informações comuns.
Quando se mapeia uma arquitetura orientada a objetos em um banco de dados relacional.

Estrutura



Participantes

- SuperClasse
  Tabela com estrutura igual a parte comum entre as tabelas-filhas.
  Um de seus campos é um tipo enumerado que distingue de qual tipo se refere seus dados.
- SubClasse
  Tabela com dados que diferem entre subtipos
  Um de seus campos contém a chave primária de seu registro na tabela ancestral.

Colaborações

O campo de tipo da tabela ancestral permite uma filtragem prévia por tipo, otimizando a busca por dados em tabelas filhas.

A chave primária da tabela ancestral e da tabela filha devem ser as mesmas.

Consequências

Existe uma desvantagem óbvia do uso desta estrutura. Se as tabelas não fossem convertidas, a seleção por um campo seria uma simples busca, e não um produto cartesiano entre os registro. Há uma perda em desempenho significativa com seu uso. Em compensação, sua estrutura é mais enxuta do que as tabelas originais, ganhando-se então espaço em disco. É a velha dualidade espaço X desempenho que vemos em computação.

Futuras alterações no banco de dados ficam simplificadas, reaproveitando a estrutura e código existentes.

Uma vantagem extra ganha seria o polimorfismo em banco de dados. Imagine uma tabela onde seriam registradas as vendas de uma empresa, e o restante do banco de dados igual a estrutura analisada anteriormente. Uma venda poderia ser feita tanto para uma empresa, quanto para um cliente pessoa física ou até mesmo um funcionário! Basta guardar na tabela de venda a chave primária pessoaId da tabela Pessoa, e tudo passa a funcionar como se estivessemos utilizando polimorfismo.