5

O Pronome PARENT

Neste capítulo, é explorado o pronome PARENT. Na Seção 5.1, a relação entre o objeto corrente e o objeto que o declarou é explicada. Na Seção 5.2, são mostradas as transformações de código necessárias para que o pronome PARENT seja corretamente executado. Na Seção 5.3, é apresentado o nível Meta necessário para a obtenção destas transformações. Finalmente, na Seção 5.4, a implementação do padrão *Mediator* [G+95], descrito na Subseção 2.4.3, utilizando o pronome PARENT é comparada com a implementação tradicional.

5.1

O Objeto que Declarou o Objeto Corrente da Execução

Do ponto de vista estático, um programa orientado a objetos é uma coleção de classes, possivelmente aninhadas, declaradas em um programa ou obtidas de uma biblioteca. Instâncias destas são agrupadas em objetos agregadores que orquestram seus comportamentos. Objetos agregadores, por sua vez, podem ser agrupados em objetos agregadores mais externos formando o que em orientação a objetos é denominado de árvore de agregação. A estrutura estática de um sistema orientado a objetos pode ser visto, portanto, como uma destas árvores, ou como uma coleção delas.

Nesta árvore, objetos pais conhecem seus filhos e, por isso, podem se comunicar com estes mandando-lhes mensagens. Já objetos filhos não conhecem seus pais e, portanto, não conseguem enviar mensagens a estes, a menos que uma referência cruzada para este seja mantida. O pronome PARENT pode ser usado para representar o objeto pai no contexto do objeto filho e, com isto, permitir que a relação seja percorrida também no sentido inverso.

Em linguagens onde o conceito de ponteiros é escondido do programador, os conceitos de PARENT e CREATOR se confundem. Em linguagens como C++, no entanto, onde objetos podem ser declarados como referência por um

dado objeto, estas referências passadas como parâmetro para algum método de outro objeto e, neste, os objetos serem criados, PARENT e CREATOR representam objetos distintos.

A Figura 5.1 mostra o cenário no nível base de três classes que utilizam o pronome PARENT. A classe *A*, em algum de seus métodos, envia a mensagem *m* através do pronome e possui um atributo regular modelado pela classe *B*. A classe *B*, por sua vez, possui um atributo dinâmico modelado pela classe *A* e envia a mensagem *n* a seu pai. A classe *C* não possui atributos, mas cria em seu método *z*, um objeto modelado pela classe *B*. As classes *A* e *B* declaram as mensagens enviadas via pronome através do especificador <<message>> introduzido na linguagem.

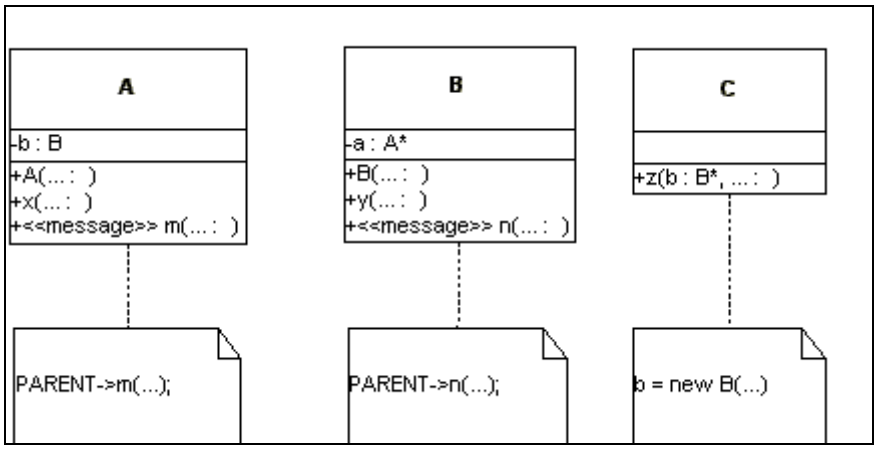

Figura 5.1 – Cenário de utilização do pronome PARENT em nível base

5.2

Transformações de Código

O cenário anterior deve ser transformado de modo que:

1. as classes *A*, *B* e *C* possuam uma superclasse que implemente de forma vazia um tratador para *m()* e para *n()*; e
2. o objeto pai do objeto corrente seja conhecido no momento em que uma mensagem a este seja enviada.

Para a primeira transformação, é adotada a mesma estratégia já descrita para os pronomes anteriores.

Para a segunda transformação, é necessário mapear quando a relação entre pai e filho é definida e sua duração. Ao contrário da relação entre criador e criatura onde a relação é definida exatamente quando o objeto é criado, a relação entre pai e filho é definida estaticamente quando um objeto é declarado. Quando o objeto é um objeto regular, sua criação coincide com a criação de seu pai podendo ser utilizada, portanto, a mesma estratégia da relação entre criador e criatura. Isto é, o objeto pai pode ser passado como parâmetro para o construtor da classe que modela o objeto filho. Quando o objeto declarado é dinâmico, no entanto, sua criação não coincide com a de seu pai, podendo se dar até fora de seu contexto. Assim, não é trivial a determinação do objeto pai de um objeto dinâmico no momento em que este é criado.

A solução adotada para este problema foi "guardar" a referência para o pai do objeto dinâmico, conhecida quando este é declarado, até o momento de sua criação. Para isso, como pode ser verificado na Figura 5.2, introduzimos, como suporte de tempo de execução, a classe genérica *Pont<T>* que funciona como uma "casca" para o objeto dinâmico. Todo objeto dinâmico tem seu tipo *t* substituído por *Pont<t>* e, com isso, o atributo, que era dinâmico passa a ser regular. Assim, passa a ser criado junto com seu pai e pode "guardar" a referência a este. No inicializadores dos construtores, portanto, o *parent* de todos os atributos é inicializado. Isso é feito através do método *SetParent()*, que é introduzido em toda classe raiz da árvore hierárquica e existente na classe genérica *Pont<T>*. Quando o atributo for dinâmico, é executado o método da classe *Pont<T>* e o objeto pai guardado. Quando o atributo for regular, o método da classe que o modela é executado e o objeto pai atribuído a um novo atributo também introduzido em toda raiz de árvore hierárquica.

Pont<T>, além do método *SetParent()*, disponibiliza também operadores para que o valor do objeto real possa ser alterado (operador de atribuição) e lido (operador ->). Assim o atributo dinâmico, que teve sua classe modificada pelas transformações, pode continuar sendo utilizado como se isso não tivesse ocorrido.

Quando um objeto é explicitamente criado, é introduzido um comando para que seja atualizada sua referência a seu pai que está devidamente guardada desde a criação deste.

A Figura 5.2 mostra o cenário transformado. Em todas as classes é introduzido o atributo *m_pParent* e o método *SetParent()*, já que estas são, originalmente, raízes de árvores hierárquicas. O atributo *a* da classe *B*, que é um ponteiro para um objeto da classe *A*, tem seu tipo alterado para *Pont<A>* e, no

construtor da classe *B*, o método *SetParent()* da classe *Pont<T>* é invocado para que a referência ao objeto modelado por *B*, que está sendo criado, seja "guardada". No construtor da classe *A*, o método *SetParent()* do atributo *b* é invocado para que a referência ao objeto que está sendo criado seja armazenado como pai deste. Na classe *C*, o parâmetro do método *z*, que é modelado como um ponteiro para um objeto da classe *B*, tem seu tipo alterado para *Pont<B>.* Assim, quando uma referência a um objeto não criado é passada, a referência ao objeto que o declarou é também recebida para, caso o objeto seja, neste contexto, criado, como acontece no exemplo, sua referência interna possa ser imediatamente atualizada.

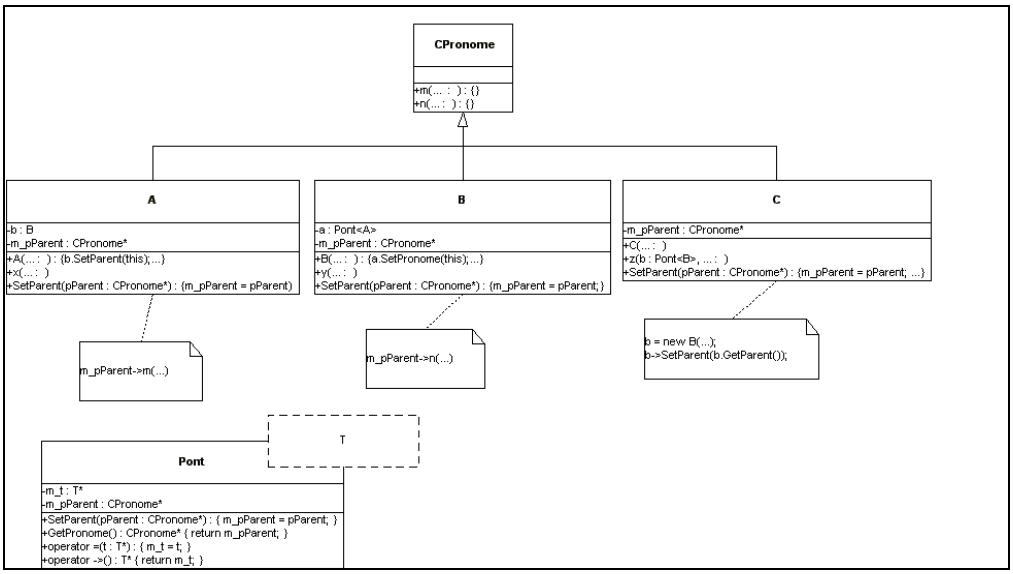

Figura 5.2 – Cenário de utilização do pronome PARENT transformado

5.3

O Nível Meta

Para obter estas transformações, a todas as classes do nível base é associada a metaclasse *MCParent*, a menos da classe *CPronome* que é associada a uma metaclasse especial denominada *MCRaizPronome*. O nível meta e sua associação com o nível base mostrado anteriormente são detalhados na Figura 5.3.

Os nomes dos métodos que aparecem na Figura 5.3 são os definidos em OpenC++, podendo no entanto ser mapeados para qualquer outra linguagem aberta desde que suas características de reflexão sejam preservadas. Os

métodos *Initialize(), RegisterNewAcessSpecifier(), RegisterNewClosureStatment(), TranslateClass(), BaseClasses(), ChangeBaseClasses(), AppendMember(), TranslateUserStatment()* e *TranslateNew()* da metaclass *Class*, os métodos *IsFunction(), IsConstructor(), Signature()* e *GetUserAccessSpecifier()* da metaclass *Member* e a função *IsPointerType()* da classe *TypeInfo* já foram descritos em capítulos anteriores. Alguns, no entanto, estão sendo apresentados nesta transformação:

- void Class::ChangeMember(Member& changed_member): altera o membro da classe especificado por changed_member;
- void Class::TranslateMemberFunction(Environment* env, Member& m): transforma a implementação da função membro especificada por m;

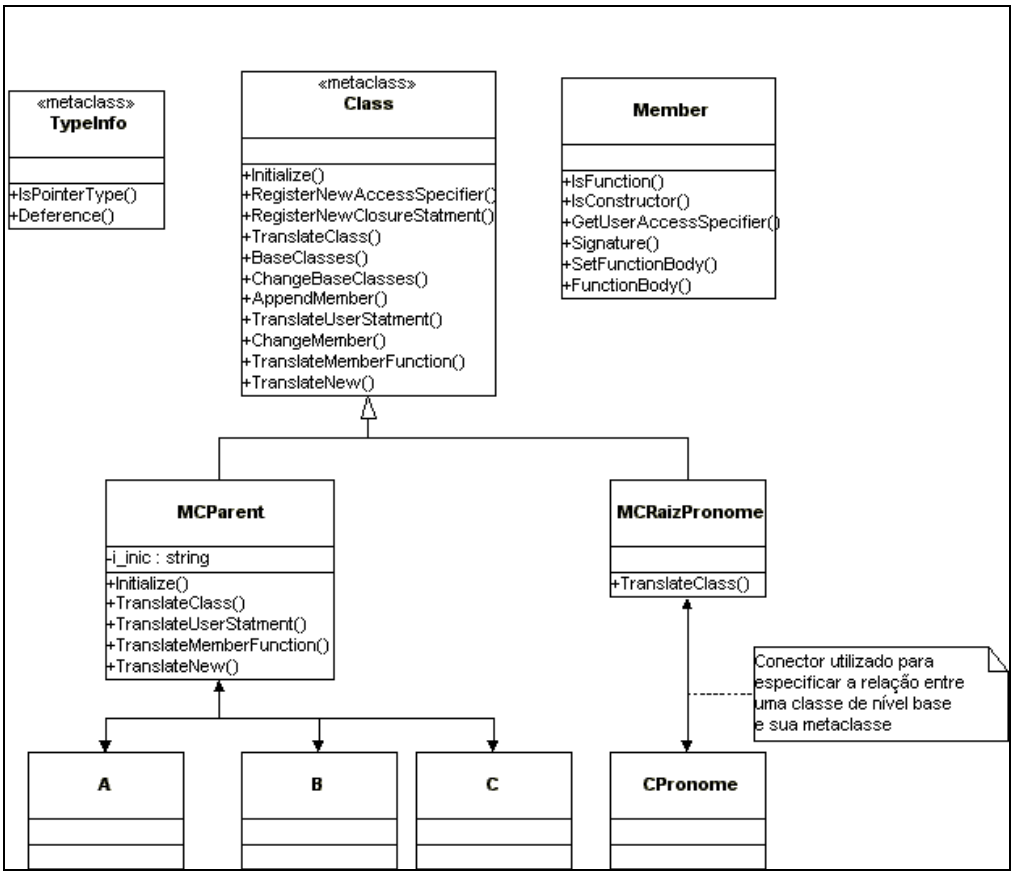

Figura 5.3 – Nível Meta para implementação do pronome PARENT

- *Ptree* Member::FunctionBody()*: retorna o corpo da função membro;
- *void Member::SetFunctionBody(Ptree* body)*: troca o corpo da função membro para *body*;

- *void TypeInfo::Deference(TypeInfo& t)*: retorna o tipo apontado por *t*;

Para que o especificador (message) e o novo comando (envio de mensagem para o PARENT) seja identificado, a metaclasse *MCParent* redefine o método *Initialize()* e faz os dois registros:

- RegisterNewAcessSpecifier(“message”)
- RegisterNewClosureStatment(“PARENT”)

As inserções da nova herança, do novo atributo e do novo método nas classes raízes das hierarquias de herança, além da transformação dos atributos dinâmicos e do armazenamento das mensagens enviadas através do pronome são feitos na redefinição do método *TranslateClass()*.

Para o armazenamento das mensagens enviadas através do pronome, foi utilizado o objeto *Singleton* [G+95] *RepositorioMensagens* que armazena as mensagens declaradas com o especificador <<message>> e as disponibiliza para ser introduzidas na classe *CPronome*.

As inserções, no corpo dos construtores, da inicialização do atributo extra e do pai dos atributos dinâmicos, além da transformação dos parâmetros dinâmicos de métodos, são feitas pelo método *TranslateMemberFunction()*.

A inserção do comando de inicialização da referência ao objeto pai quando um objeto é criado explicitamente é feita pelo método *TranslateNew()*.

Por fim, a transformação do envio efetivo da mensagem através do pronome PARENT é feita redefinindo-se o método *TranslateUserStatment()*. Este substitui no comando original a palavra-chave PARENT pelo atributo *m_pParent* que, pela transformação efetivada pelo método *TranslateClass()*, está disponível para toda a classe.

Ao término da transformação da última classe associada a *MCParent*, todas as mensagens enviadas através do pronome PARENT estão armazenadas.

A metaclasse *MCRaizPronome* redefine o método *TranslateClass()* e, neste, insere na classe *CPronome* uma função membro para cada mensagem armazenada no repositório, sempre com implementações vazias.

5.4

O Padrão *Mediator* Utilizando PARENT

O padrão *Mediator* [G+95], descrito na Subseção 2.4.3, pode ser implementado utilizando-se o pronome PARENT. Esta implementação é esquematizada na Figura 5.4.

Nesta solução, colegas não precisam manter uma referência para o mediador, o que torna esta classe mais genérica, por possibilitar que objetos possam ser mediados por qualquer tipo de mediador. Com isso, não existe mais a necessidade da existência da árvore hierárquica de mediadores.

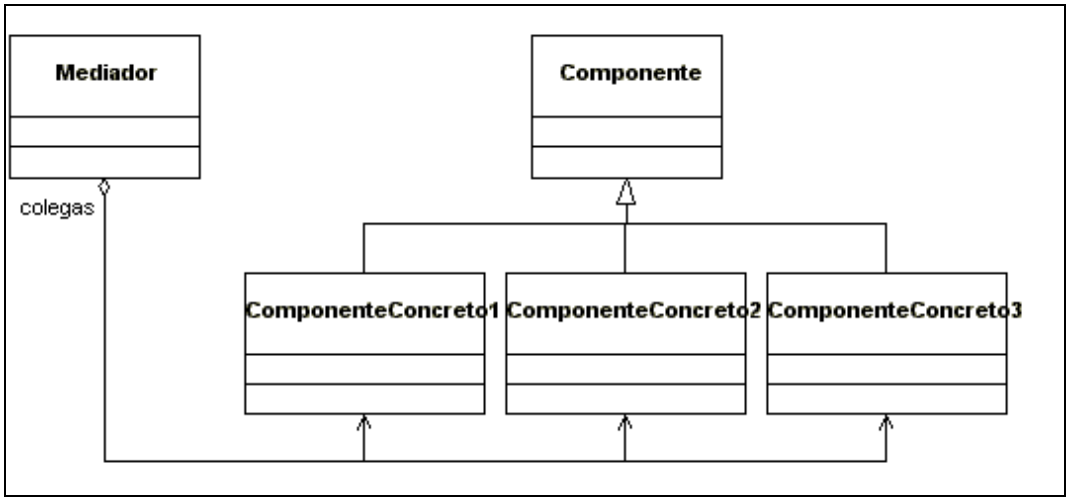

Figura 5.4 – Esquema da implementação do padrão *Mediator* utilizando PARENT

Os tempos de execução da implementação tradicional do padrão, proposto em [G+95], e da implementação utilizando o pronome PARENT podem ser comparados na Tabela 5.1 e no gráfico mostrado na Figura 5.5. Estes tempos foram conseguidos executando-se o algoritmo nas duas implementações e variando-se o número de atributos extras nas classes. Isto é, os atributos necessários para a implementação do padrão aparecem de forma constante em todas as implementações, sendo incluídos atributos extras a cada configuração. Imaginando-se a implementação do algoritmo como um módulo componente aplicação, a variação é feita incluindo-se novos atributos em outros módulos que não este.

A variação escolhida foi devido ao fato de ser exatamente neste tipo de construção que a implementação do pronome PARENT adiciona código a ser executado e, com isso, aumenta o tempo de execução do sistema.

Da mesma forma que nos pronomes descritos anteriormente, cada configuração do exemplo foi executada 1000 vezes, sendo os tempos mostrados na Tabela 5.1 a média destas execuções. Os desvios padrões das amostras consideradas são apresentados na Tabela 5.2. O ambiente de execução destes experimentos foi o mesmo do utilizado para o pronome SENDER. Isto é, um computador Pentium4, com 256 MB de RAM, utilizando-se Linux Mandrake 8.1.

Número de atributos	Implementação em [G+95] (μs)	Implementação com PARENT(μs)	Diferença (μs)	Aumento Percentual
0	2004	2096	92	4,5908
1	13637	14063	426	3,1238
10	22939	23786	847	3,6924
100	429862	442833	12971	3,0174
1000	4408247	4605459	197212	4,4737
10000	44082469	46054575	1972106	4,4736
100000	440824680	463959858	23135178	5,2481
1000000	4408246700	4673012356	264765656	6,0061
10000000	44082466000	46863532783	2781066783	6,3087
100000000	4,40825E+11	4,70769E+11	29944042095	6,7927
1000000000	4,40825E+12	4,71982E+12	3,11573E+11	7,0679
10000000000	4,40825E+13	4,77103E+13	3,62786E+12	8,2297

Tabela 5.1 – Tempos de execução do padrão *Mediator*

Número de atributos	Desvios Padrões Implementação Padrão	Desvios Padrões Implementação com PARENT
0	0,2829	0,2189
1	0,2934	0,2291
10	0,0274	0,3186
100	0,2389	0,2951
1000	0,2834	0,3132
10000	0,2194	0,2962
100000	0,2308	0,2165
1000000	0,3484	0,2190
10000000	0,3119	0,3983
100000000	0,2098	0,2930
1000000000	0,2631	0,4198
10000000000	0,2509	0,3194

Tabela 5.2 – Desvios Padrões das amostras de tempo utilizadas para análise da execução do padrão Mediator

Os dados apresentados na Tabela 5.1 são melhor visualizados no gráfico da Figura 5.5.

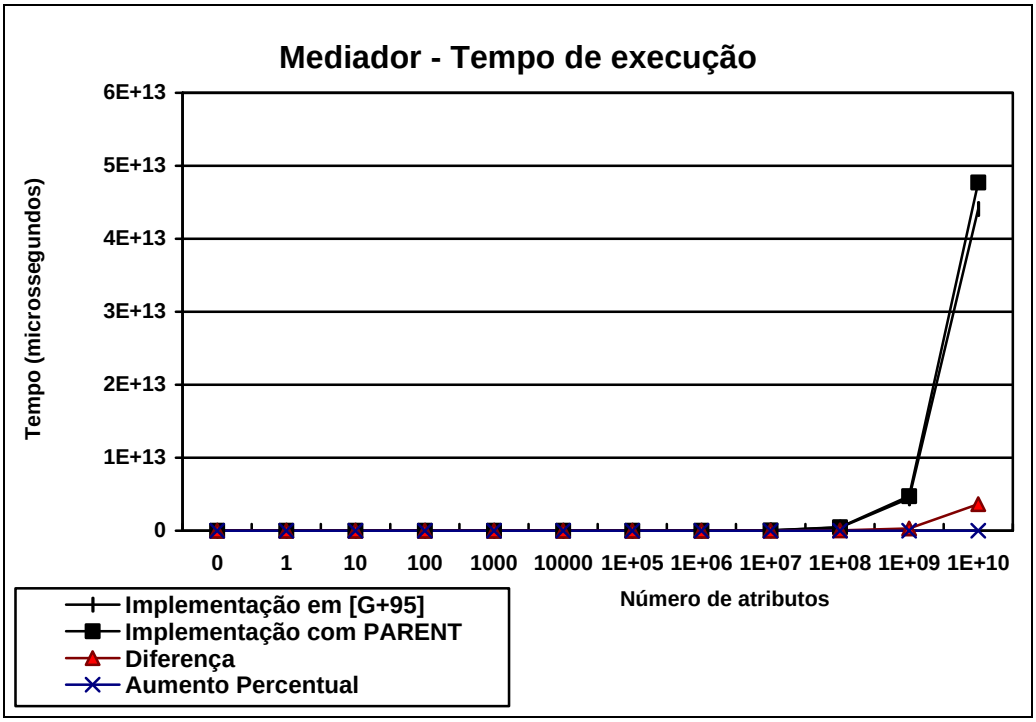

Figura 5.5 – Gráfico de comparação entre os tempos de execução do padrão *Mediator*

Na Tabela 5.1 e no gráfico mostrado na Figura 5.5 pode-se verificar que o tempo de execução na implementação do padrão utilizando-se o pronome PARENT é sempre um pouco maior, o que já era esperado. O aumento percentual máximo obtido, no entanto, foi de 8%, evidenciando a não degradação do sistema com o uso do pronome.