2

Comunicação Baseada em Pronomes

Este capítulo tem por objetivo expor os princípios da comunicação baseada em pronomes. Na Seção 2.1, o conceito de pronomes é apresentado e as vantagens e desvantagens de introduzi-los em uma linguagem orientada a objetos discutidas. Nas Seções 2.2 e 2.3, são apresentados, respectivamente, as semânticas de envio e recebimento de mensagens através de pronomes. Na Seção 2.4, são exemplificadas aplicações que teriam suas implementações facilitadas com a utilização de pronomes. Na Seção 2.5, é feito um resumo teórico de linguagens abertas a fim de justificar a implementação de pronomes através destas. Finalmente, na Seção 2.6, este trabalho é comparado com outros de alguma forma semelhantes.

2.1

Apresentação

Para reduzir a necessidade de referências cruzadas sem quebra de encapsulamento ou regras de visibilidade, neste trabalho, estende-se o conceito de pronome existente em [Car93], propondo-se especificações genéricas para representar objetos específicos. Através destas, mensagens podem ser despachadas para objetos sem que seus identificadores sejam explicitados.

Existe uma correlação entre pronomes, como propostos aqui, e pronomes encontrados em linguagens naturais. Não se pode, no entanto, permitir em uma linguagem de programação o tipo de ambigüidade normalmente encontrada em linguagens naturais, onde o contexto é utilizado para associar um pronome com seu significado real (o objeto referenciado). Em programação orientada a objetos, existe apenas um contexto que é claramente definido por todo o tempo, que é o objeto corrente da execução. Qualquer informação extra precisa ser explicitamente provida. É por esta razão que linguagens orientadas a objetos normalmente oferecem um pronome, que descreve exatamente o objeto corrente

da execução. Este trabalho estende este conceito, propondo pronomes para descrever outros objetos.

A principal idéia é especificar um pronome para um conjunto de objetos ou, na maioria dos casos realmente úteis, por uma propriedade que descreve os objetos associados com o pronome, dado um ponto no programa e o tempo de execução. Pouca vantagem, no entanto, é conseguida com a introdução em uma linguagem de programação de uma facilidade para descrever um pronome como uma função (declarada pelo programador) para determinar uma coleção de objetos quando chamada.

Os pronomes de interesse deste trabalho são aqueles que descrevem objetos ou conjunto de objetos que não podem ser determinados no contexto do objeto corrente.

Poderia ser interessante, por exemplo, poder enviar uma mensagem para o objeto “criador”, isto é, o objeto responsável pela criação do objeto corrente. Não existe forma de um objeto determinar seu criador e, portanto, seria necessário de alguma forma salvar (uma referência para) este. Toda vez que um objeto é criado, pode-se passar uma referência para o criador (como um parâmetro do construtor por exemplo) e esta referência será armazenada no objeto criado como um atributo para ser usado quando necessário. A alternativa é usar um pronome CREATOR, que dará acesso imediato ao objeto criador toda vez que este acesso é necessário.

Através do exemplo anterior, pode-se perceber vantagens e desvantagens da disponibilização de pronomes: como eles dão acesso a objetos que não podem ser identificados em uma programação regular no contexto do objeto corrente, eles evitam o uso de truques como o descrito, que implicam em código extra sendo adicionado. O acesso seguro a objetos sem programação extra é a vantagem. A desvantagem é que o sistema deve ser capaz de manter as informações necessárias para ter o objeto ou conjunto de objetos identificados corretamente quando se fizer necessário.

O uso de pronomes torna visíveis relações implícitas que existem entre objetos de um sistema, como a descrita entre um objeto e seu criador. É claro que não se espera nomear através de pronomes todas as possíveis relações entre objetos. Primeiro, porque a determinação deste conjunto é muito difícil e segundo porque, pela desvantagem descrita acima, qualquer sistema que tentasse fazê-lo tornar-se-ia extremamente pesado, pois seria obrigado a manter um conjunto muito grande de relações implícitas. Ao invés disso, compõe-se, neste trabalho, um conjunto de pronomes úteis para a implementação de

algumas arquiteturas de software e padrões de projetos bem conhecidos a fim de validar o conceito de pronomes.

Neste conjunto, incluem-se pronomes para representar objetos específicos, conjunto de objetos e objetos relacionados de alguma forma com o objeto corrente da execução. No primeiro caso, o sistema precisa manter a referência para os objetos representados. No segundo caso, é necessário que o conjunto possua as propriedades que regem seus elementos precisamente definidas, para que o sistema possa mantê-lo atualizado e, no terceiro caso, as relações mapeadas precisam ser automaticamente criadas e mantidas.

Pronomes representando objetos não relacionados com o objeto corrente da execução representam sempre os mesmos objetos, não variando com o objeto que está enviando a mensagem. Nestes casos, o sistema pode manter globalmente referências para os objetos destinos. De forma contrária, pronomes representando objetos relacionados com o objeto corrente da execução variam com este, sendo necessário manter as referências associadas aos objetos relacionados.

A existência de pronomes torna o sistema mais exposto, uma vez que disponibiliza acesso a serviços que, sem estes só seriam acessados com a declaração explícita de objeto modelado pela classe que disponibiliza o serviço.

2.2

Semântica de distribuição de mensagens através de pronomes

A semântica de distribuição de mensagens varia com o que o pronome representa. Para pronomes que representam objetos únicos (específicos ou relacionados de alguma forma com o objeto corrente da execução), referências para objetos são mantidas pelo sistema e o envio de uma mensagem através destes pronomes é transformado no envio para a respectiva referência. Para pronomes representando um conjunto de objetos, conjuntos são mantidos atualizados e o envio de uma mensagem através destes pronomes é transformado em vários envios, um para cada objeto do conjunto.

Pronomes mapeiam relações onde, muitas vezes, o objeto emissor não possui controle sobre o ciclo de vida do objeto receptor. É o caso, por exemplo, do pronome CREATOR onde a criatura (objeto emissor) não controla o ciclo de vida do criador (objeto receptor). Nestes casos, o objeto destino da mensagem pode já ter sido destruído, provocando erros de execução quando mensagens

lhe são enviadas. A fim de evitar este problema, linguagens com coletor de lixo são mais apropriadas para a incorporação de pronomes.

2.3

Semântica de recebimento de mensagens através de pronomes

Como a identificação de um objeto representado por um pronome não é conhecida estaticamente, o compilador não pode checar se a classe que o modela possui um tratador adequado para a mensagem enviada. Para evitar erros de execução e uma vez que mensagens são enviadas através de pronomes o são, na grande maioria das vezes, para anúncio de eventos, a semântica adotada para recebimento destas é: tratá-la se o tratador adequado for identificado e ignorá-la caso contrário. A identificação do tratador adequado passa pelo especificador de acesso associado a este. Tratadores privados só são identificador se o objeto que está enviando a mensagem for da mesma classe do objeto representado pelo pronome. Tratadores protegidos só são identificados se o objeto que está enviando a mensagem for da mesma classe ou de alguma subclasse da classe do objeto representado pelo pronome. Já tratadores públicos são identificados qualquer que seja a classe do objeto emissor da mensagem.

Esta semântica garante ao envio de mensagens através de pronomes um comportamento semelhante ao da programação por eventos, onde estes são tratados apenas pelos objetos que se registram para tal. No caso de pronomes, este registro se dá pela disponibilização de um tratador para o evento.

Como conseqüência desta semântica, objetos emissores de mensagens através de pronomes não têm a garantia que suas mensagens serão tratadas e, por isso, não podem ficar bloqueados aguardando que isto ocorra, nem esperar resultados destes envios. Por este motivo, mensagens enviadas através de pronomes são assíncronas.

2.4

Aplicações

A seguir serão apresentados alguns exemplos de arquiteturas ou padrões de projetos cujas implementações seriam facilitadas com a utilização de algum pronome. Os critérios de escolha destas aplicações foram:

Utilização : foram escolhidas arquiteturas e padrões com referências de utilização prática;

Validade : as arquiteturas e padrões foram retirados da literatura a fim de garantir suas validades;

Diversidade : foram escolhidos padrões e arquiteturas de características diversas a fim de garantir a abrangência do conjunto.

2.4.1

Distributed Callback [MM97]

Este padrão comportamental é bastante utilizado no contexto de arquiteturas de sistemas distribuídos que utilizam CORBA [OMG92]. Apesar de ser aplicável também em outros contextos, é neste primeiro que o padrão é aqui discutido.

O problema focado pelo padrão consiste em clientes que necessitam de serviços de servidores, mas precisam continuar seus próprios processamentos enquanto os servidores atendem suas requisições. Algum mecanismo deve ser utilizado para que os objetos servidores possam retornar seus resultados quando estes estiverem disponíveis.

Uma solução para esta situação, conforme ilustrado na Figura 2.1, consiste em rescrever os métodos servidores como operações *oneway*, que são tratadores de mensagens assíncronas. Desta forma, estes tratadores podem receber apenas parâmetros de entrada e suas execuções não bloqueiam o cliente, que pode continuar sua execução. Parâmetros de saída devem fazer parte de uma segunda operação *oneway*, esta do cliente, que é disparada pelo servidor. Como servidores não conhecem a identificação do cliente, este deve definir um objeto de retorno (*callback object*) para cada servidor e passar a referência a este objeto como parâmetro de entrada do método original. Quando o servidor estiver pronto para prover resultados, usa o objeto de retorno como

receptor da segunda operação *oneway*, enviando os parâmetros de saída de volta ao cliente. Esta solução é esquematizada a seguir.

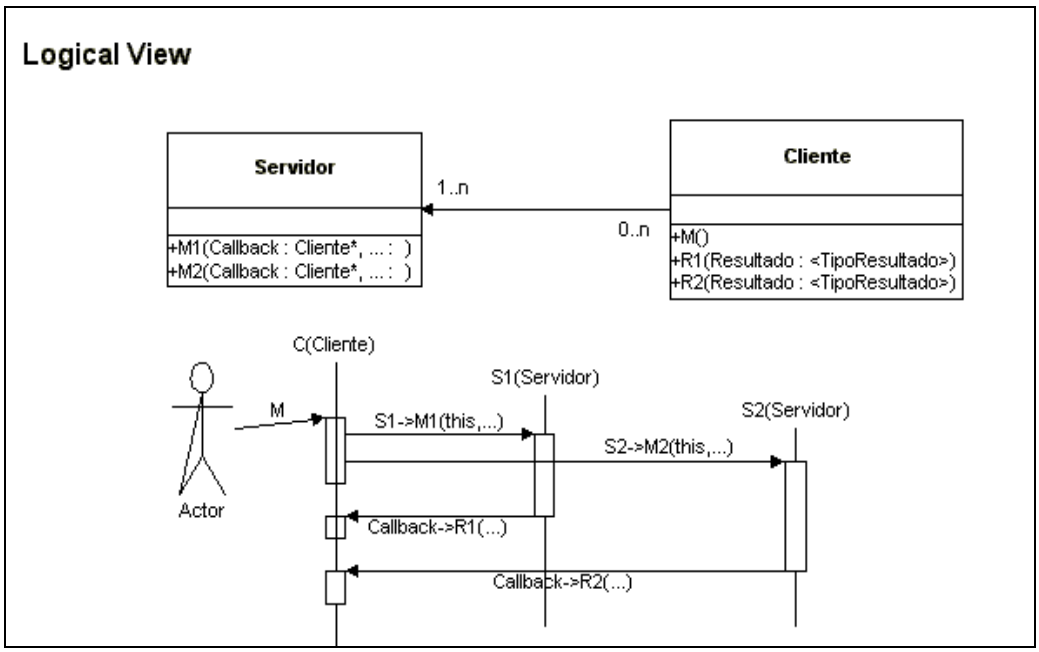

Figura 2.1 – Solução do Padrão *Distributed Callback*

Alguns comentários sobre esta solução:

- A menos que exista algum mecanismo específico de proteção de parâmetro, e que este seja usado em relação ao objeto de retorno, sua referência pode ser propagada pelo servidor. Isto pode gerar resultados indesejáveis, como um particular objeto de retorno de um cliente ser utilizado por um servidor errado.
- "Servidores não podem levantar exceções para seus clientes" [MM97]; qualquer erro no servidor deve ser parte da interface de retorno. Isto restringe a determinação e a recuperação de erros e pode ser inaceitável em certos domínios de aplicação, como sistemas de tempo real por exemplo.
- Apesar do uso de linguagens especiais para descrição de interfaces e de ferramentas que possam simplificar a redefinição de métodos, uma grande dose de disciplina é necessária para evitar erros de programação, especialmente se a quantidade de métodos a serem rescritos for grande.

- Objetos de retorno amarram servidores a clientes específicos, reduzindo suas generalidades.

Se o objeto servidor possuísse, no entanto, alguma forma de se referenciar ao objeto que lhe solicitou o serviço sem precisar conhecê-lo, o acoplamento entre estas duas classes poderia ser rompido.

O pronome SENDER, proposto neste trabalho, visa exatamente facilitar este tipo de comunicação entre objetos. Dispondo deste, o objeto servidor não precisa que lhe seja exposta uma referência do objeto cliente, o que o torna totalmente independente da classe cliente e, com isso, bem mais genérico. Além disso, o perigo da propagação da referência é eliminado.

2.4.2

Algoritmo Paralelo de Divisão e Conquista

A estratégia da divisão e conquista consiste de três passos principais [JaJ92]:

- divisão da entrada em partes de tamanhos iguais;
- resolução recursiva do subproblema definido por cada parte da entrada;
- combinação sas soluções dos diferentes subproblemas em uma solução para todo o problema.

[And91] ilustra, através de uma solução paralela para o problema da quadratura adaptativa para integração numérica [GM85], como paralelizar qualquer algoritmo de divisão e conquista, usando trabalhadores replicados e uma bolsa de tarefas. O único pré-requisito é que os subproblemas sejam independentes entre si.

O problema da quadratura. Dada uma função contínua f(x) e dois valores l e r, com l < r, o problema é calcular a área delimitada por f(x), o eixo x, e as linhas verticais passando por l e r. Então, o problema é aproximar a integral de f(x) de l até r. A forma mais conhecida é dividir o intervalo [l,r] em uma série de subintervalos e usar um trapézio para aproximar a área de cada subintervalo.

O problema pode ser resolvido estaticamente ou dinamicamente. A solução dinâmica, chamada quadratura adaptativa, começa com um intervalo de l até r e calcula o ponto médio, m, entre l e r. Calcula, então, as áreas de três trapézios:

- maior, delimitado por l, r, f(l) e f(r);

- primeiro menor, delimitado por l, m, f(l) e f(m);
- segundo menor delimitado por m, r, f(m) e f(r).

Depois, a área do maior trapézio é comparada com a soma das duas áreas menores. Se estas forem suficientemente próximas, a área do trapézio maior é considerada uma boa aproximação da área sobre f. Se não forem, o processo é repetido para resolver os dois subproblemas de calcular a área de l a m e de m a r. Este processo é repetido recursivamente até a solução de cada subproblema ser aceitável. As respostas dos subproblemas são, então, somadas para que o resultado final seja encontrado.

A solução usando bolsa de tarefas. A solução usa um canal compartilhado que contem uma bolsa de tarefas. Inicialmente, uma tarefa correspondendo a todo o problema é lá introduzida para ser solucionada. Múltiplos processos trabalhadores pegam tarefas da bolsa e os processam, eventualmente gerando novas tarefas (correspondendo a subproblemas) que são colocadas na bolsa. O algoritmo termina quando todas as tarefas tiverem sido processadas.

A solução paralela. Como cada subproblema é independente de outro, o algoritmo pode ser paralelizado. Uma forma é usar um processo administrador e alguns processos trabalhadores [CGL86][Gen81]. A estrutura da solução é ilustrada na Figura 2.2:

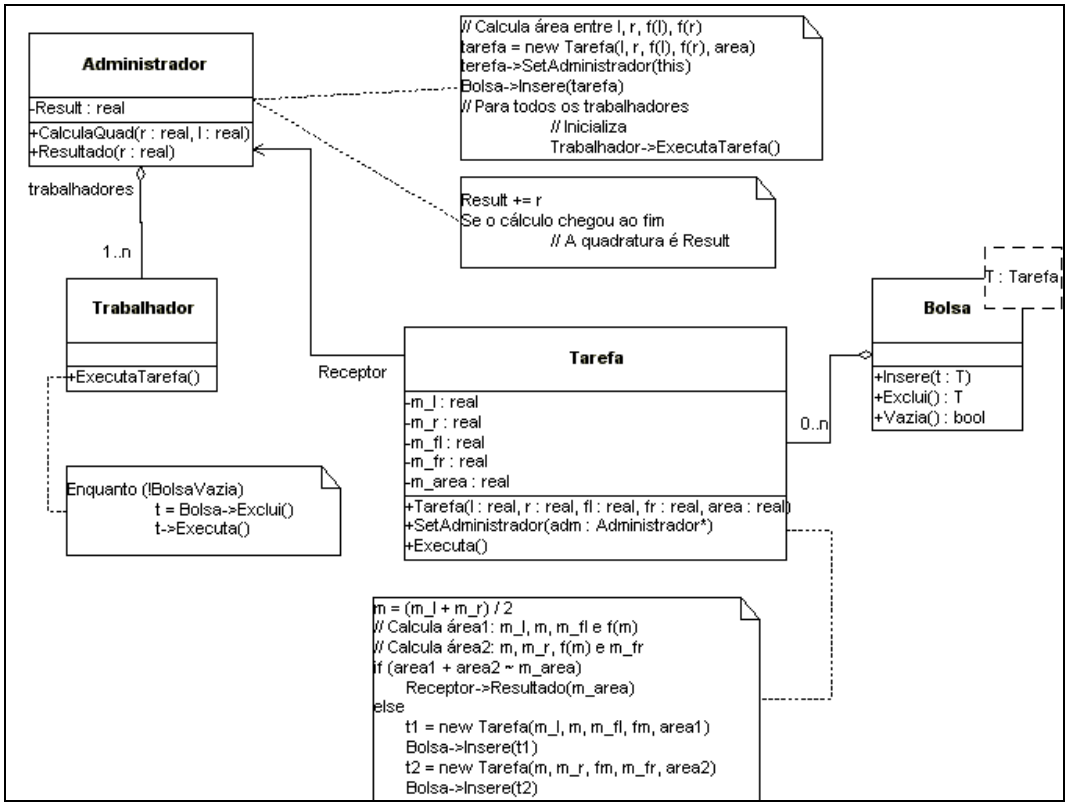

Figura 2.2 – Solução para o problema da quadratura utilizando bolsa de tarefas

O administrador gera o primeiro problema (em CalculaQuad), insere-o na bolsa de tarefas, cria os trabalhadores e os coloca para trabalhar.

Trabalhadores compartilham um canal simples (bolsa) que contem os problemas a serem resolvidos. Enquanto estão executando tarefas (enquanto a bolsa não fica vazia), os problemas resolvidos são removidos da bolsa e solucionados.

Um problema é composto pelos valores extremos do trapézio e pela área deste. A solução (implementada pelo método Executa) divide o trapézio em dois e calcula suas áreas, verificando se a área do trapézio original é uma boa aproximação da área real da função. Se for o caso, a área do trapézio original é retornada ao objeto administrador. Em caso contrário, duas novas tarefas são criadas para os dois trapézios menores e inseridas na bolsa.

O método Resultado, definido no Administrador, recebe resultados, verificando se o processo terminou.

No último passo está a principal dificuldade do algoritmo: a determinação do término. Como o processo Administrador sabe que o algoritmo terminou se subproblemas são gerados dinamicamente, sem sua interferência? Na realidade, o algoritmo termina quando a bolsa está vazia e todos os trabalhadores estão esperando; no entanto, esta situação é difícil de ser detectada. No código

apresentado, por exemplo, seria necessário que os trabalhadores controlassem e exportassem seus estados para que o administrador pudesse detectar o fim do algoritmo.

Este problema pode ser contornado se tanto a classe Administrador quanto a classe Tarefa disponibilizassem um método para recolhimento de resultados e exportassem uma referência de seus objetos para as tarefas eventualmente criadas. Através desta referência, a tarefa devolve o seu resultado e este é computado. Desta forma, cada tarefa ou administrador sabe quantos resultados esperar dado que receberá um resultado para cada tarefa criada. Uma vez recebidos todos os resultados esperados, a tarefa adiciona-os e repassa seu próprio resultado. Assim, o objeto administrador detecta o fim do algoritmo quando receber o resultado da tarefa por ele criada.

Nesta solução, o problema da detecção de fim é resolvido mas aparece o problema da propagação de referência, já abordado anteriormente. Além disso, aumenta bastante o acoplamento entre as classes Administrador e Tarefa pois, uma tarefa pode ser criada tanto pelo administrador quanto por outra tarefa e, para que o retorno possa ser feito para qualquer uma das duas, elas precisam ter uma superclasse comum.

Se um objeto puder se referenciar ao objeto que o criou, a solução para a determinação de término pode ser adotada sem tantos pontos contrários. Trabalhadores e tarefas não precisam conhecer o objeto administrador nem tarefas precisam conhecer a tarefa que as criou. Além disso, referências não precisam ser propagadas.

O pronome CREATOR visa facilitar a comunicação neste tipo de aplicação, onde há criação dinâmica de objetos.

2.4.3

Mediator [G+95]

O padrão *Mediator* [G+95] tenta evitar os muitos problemas de conexão existentes em sistemas orientados a objetos, encapsulando comportamento coletivo em um objeto mediador separado. O mediador serve como um intermediário que libera objetos de se referenciarem a outros diretamente. O mediador conhece todos os objetos que participam da iteração e os objetos conhecem apenas o mediador, reduzindo assim o número de conexões. A estrutura do padrão é apresentada na Figura 2.3.

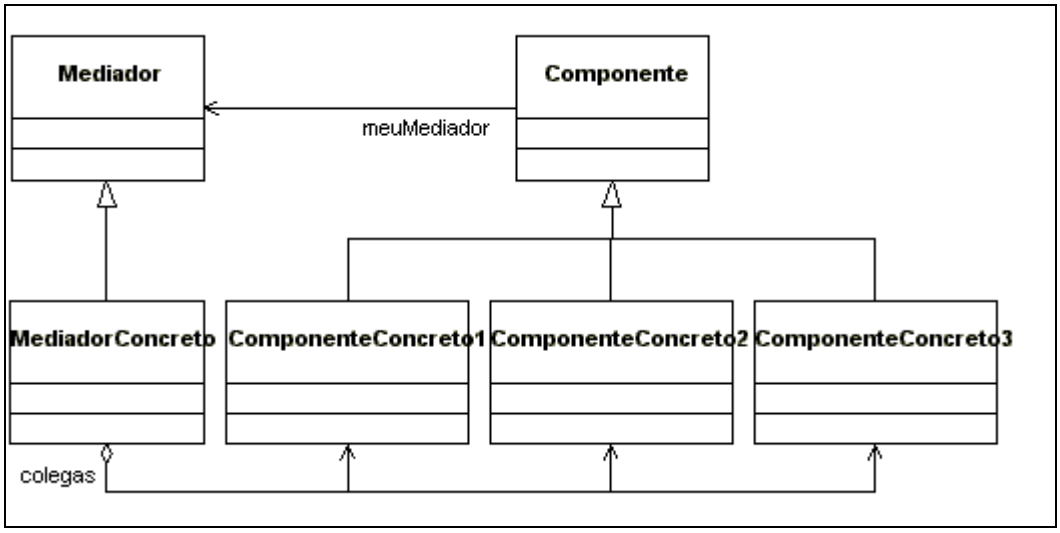

Figura 2.3 – Estrutura do padrão *Mediator*

Embora muito usado hoje em sistemas baseados em componentes para conectá-los, o padrão força o programador a manter referências cruzadas e amarrar o componente a um tipo de mediador, ou a uma família de mediadores.

Havendo uma forma de um objeto poder se comunicar com “seu pai”, ou melhor com o objeto que o declarou, não haveria necessidade dos componentes conhecerem o mediador. Com isso, este objeto pode ser modelado por qualquer classe (sem a restrição de estar na mesma árvore hierárquica), tornando as classes Componentes ainda mais reutilizáveis.

O pronome PARENT visa facilitar este tipo de aplicação, onde o envio de mensagens através da árvore de agregação no sentido filho-pai é necessário.

2.4.4

Invocação Implícita [SN92]

Recentemente, tem sido considerável o interesse em técnicas de integração referenciada como integração implícita, integração reativa, ou *broadcast* seletivo. Sua idéia básica é que, ao invés de enviar uma mensagem diretamente a um objeto, um objeto pode “anunciar” um ou mais eventos. Outros objetos no sistema podem registrar um interesse em um evento disponibilizando um tratador para ele. Quando o evento é anunciado, todos os tratadores que tenham sido registrados para o evento são invocados.

Invocação implícita provê um forte suporte para reuso, já que é possível integrar uma coleção de módulos simplesmente registrando seus interesses em eventos do sistema, além de facilitar a evolução do sistema, já que um módulo pode ser trocado por outro sem afetar as interfaces dos módulos que implicitamente dependem dele.

Por causa destas vantagens, muitos sistemas usam invocação implícita como forma principal de composição. A forma de uso, no entanto, pode variar de sistema para sistema, sendo possível agrupá-las em categorias.

A primeira, *frameworks* para integração de ferramentas, é usada em [Rei90, GI90, Ger89]. Nestes, sistemas são configurados como uma coleção de ferramentas executando como processos separados. Eventos de *broadcast* são tratados por um processo despachante separado que se comunica com as ferramentas do sistema através de recursos providos pelo sistema operacional. Poucas aplicações, no entanto, podem conviver com o alto custo de processos independentes, limitando o uso desta técnica.

A segunda, *frameworks* de aplicação, inclui sistemas que usam o padrão *observer* [G+95] como registro-anúncio de eventos ou usam um mediador de eventos como *singleton* [G+95] para captar eventos anunciados e despachá-los para os objetos interessados, como proposto em [SG96]. Estas soluções, apesar de poderem ser usadas em uma linguagem de programação de propósito geral, transferem para o programador a responsabilidade de criar e manter associações entre objetos e eventos, gerando códigos complicados.

A terceira disponibiliza invocação implícita através de notações especializadas e suporte de tempo de execução. Nesta categoria estão os métodos *when-updated* de algumas linguagens orientadas a objetos [SHO90,HGN91,KP88]. Em [SHO90], por exemplo, *triggers* (um tipo especial de tarefa Ada) são definidos pelo programador e implicitamente invocados pelo sistema quando um evento ocorre ou uma relação é alterada.

Mensagens de *broadcast* poderiam ser enviadas sem subterfúgios ou códigos muito complicados se houvesse uma forma de se referenciar, de forma direta, a todos os objetos do sistema. A todos estes a mensagem seria oferecida, sendo, no entanto, tratada apenas pelos objetos que tivessem interesse.

Podendo ser incluído na terceira categoria, o pronome ALL possibilita esta referência e visa facilitar a construção de aplicações que usam invocação implícita como forma de composição. A principal diferença entre esta solução e a solução usando *triggers* é quando a semântica de invocação é associada com o evento. Enquanto em *triggers* a semântica é associada nos servidores (nos

tratadores dos eventos), em pronomes a semântica é associada nos clientes (no comando de envio das mensagens). A conseqüência imediata é que com a solução utilizando pronomes, um mesmo tratador de eventos pode ser usado com diferentes semânticas de invocação.

2.4.5

Broker [BMR+96]

O padrão arquitetural *Broker*, usado para estruturar sistemas distribuídos com desacoplamento de componentes que interagem por invocações remotas, propõe que serviços possam ser executados por qualquer objeto, desde que este esteja livre e seja apto a executá-lo. Um componente mediador é introduzido para coordenar comunicações entre clientes e servidores, conseguindo desta maneira um melhor desacoplamento destes. Servidores registram-se com o mediador, e disponibilizam seus serviços para clientes através de interfaces de métodos. Uma tarefa do mediador é localizar o servidor apropriado, repassando o pedido para este e transmitindo resultados e exceções de volta ao cliente. A Figura 2.4 mostra, em um cenário simplificado, o comportamento do padrão quando um cliente envia uma solicitação de serviço a um servidor.

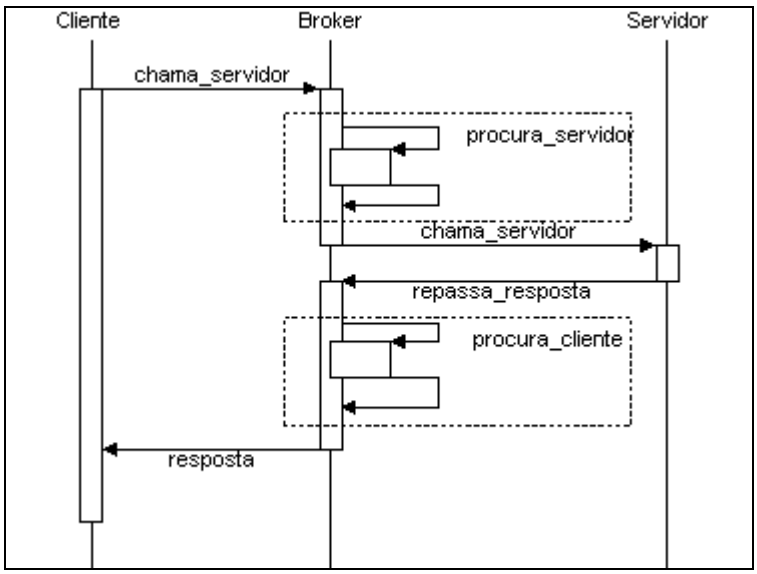

Figura 2.4 – Comportamento do padrão Broker atendendo a uma requisição de um cliente

Apesar de bastante complexo, será detalhado um aspecto deste padrão: a tarefa do mediador de localizar servidores e clientes. Esta funcionalidade está representada no cenário da Figura 2.4 envolvida por linhas tracejadas.

Nos passos para implementação do padrão [BMR+96], é definido que: “O mediador deve manter um repositório para localizar outros mediadores ou *gateways* para os quais ele possa repassar mensagens (...) a fim de localizar servidores ou clientes”.

Um exemplo famoso de uso do padrão Broker é a especificação do CORBA (Common Request Broker Arquitecture) definida pela OMG (Object Management Group). O mecanismo de localização de servidores na implementação CORBA comercializado pela Iona [Iona95] é feito, por exemplo, através da seguinte seqüência de passos:

1. componente mediador local é contactado para gerar uma lista de nomes de *hosts*. Nesta lista, os nomes são arranjados em uma ordem aleatória.
2. componente mediador iterage por esta lista, para verificar os servidores registrados em cada *host* até encontrar um onde o serviço desejado esteja registrado. A ordem aleatória permite que um componente mediador não seja carregado por requisições, enquanto outros ficam inativos.

Existindo uma forma de se referenciar genericamente a um objeto qualquer livre para atender uma requisição, o componente mediador não precisaria manter a lista de *hosts* (com outros componentes mediadores), nem gerar, aleatoriamente, uma ordem de acesso a ela.

O pronome ANY visa facilitar aplicações onde, com o objetivo de balanceamento de carga ou, simplesmente, para desacoplar componentes do sistema, não apenas um, mas um grupo de objetos é responsável por executar determinados serviços.

2.4.6

Singleton [G+95]

O padrão *Singleton* [G+95] visa garantir que determinadas classes sejam instanciadas apenas uma vez na aplicação e prover uma forma de acesso a este objeto único. A garantia de instanciação única é dada encapsulando-se na própria classe sua instanciação. Interceptando toda tentativa de criação de novos objetos, a classe garante que apenas a primeira seja efetivada. A todas as

outras é retornada a referência ao objeto já criado. A Figura 2.5, ilustra a estrutura proposta pelo padrão.

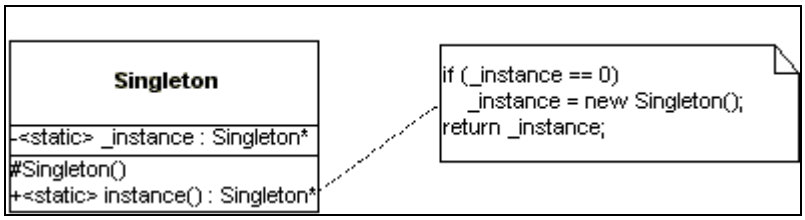

Figura 2.5 – Estrutura do padrão *Singleton*

Objetos *singletons* são muito utilizados para instanciação de controladores que precisam ser acessados por todo o sistema. Para tanto, uma forma de acesso fácil deve ser fornecida. A sugestão dada pelo padrão é que o acesso seja através de variáveis globais. No entanto, em algumas linguagens orientadas a objetos este artifício não pode ser usado, já que variáveis globais não são providas por todas as linguagens deste tipo. Nestas linguagens, estes objetos precisariam ser agregados a algum outro objeto (já que não existem globais) e as referências a estes precisariam ser exportadas a todos os objetos que necessitassem acessá-los.

É claro que o objeto agregador de *singletons* precisa ser também um *singleton* e isto poderia gerar um problema circular, não fosse a existência, em sistemas gerados por este tipo de linguagem, de um objeto *sigleton* por construção, que é o objeto principal do sistema. Este é instanciado no processo de inicialização e é através de um método seu que a execução do sistema propriamente dita se inicia. Graças a esta característica, outros objetos *singletons* podem ser agregados ao objeto principal garantindo assim suas unicidades.

As referências aos objetos *singletons* precisam ainda ser exportados a todos os objetos do sistema que necessitem acessá-los, a menos que algum acesso ao objeto principal seja disponibilizado. O pronome MAIN promove este acesso.

2.5

Linguagens Abertas

Pronomes podem ser incorporados ao núcleo de qualquer linguagem orientada a objetos desde que o compilador desta seja alterado para suportar o pronome incorporado. Dada a dificuldade em se alterar o compilador de uma linguagem, a escolha correta de quais pronomes incorporar torna-se de crucial importância.

Como cada pronome, pela característica da relação que modela, facilita a implementação de um determinado tipo de aplicação e por ser uma linguagem, na maioria das vezes, de propósito geral, quanto mais pronomes esta linguagem incorporar melhor pois um maior número de aplicações será beneficiado. Por outro lado, a incorporação de um pronome onera o suporte de tempo de execução da linguagem já que torna necessário o gerenciamento de alguma relação entre objetos antes não existente. O ponto ótimo seria, então, um número mínimo de pronomes que facilitem um número máximo de aplicações. Por ser este ponto ótimo difícil de ser determinado, optou-se por incorporar pronomes em linguagens reflexivas, conseguindo-se assim configurar o conjunto de pronomes por aplicação.

Arquiteturas abertas têm sido propostas [Kic96] para solucionar o problema de criar sistemas que sejam “adaptáveis”, ou seja, que possam ter seus requisitos funcionais alterados dinamicamente. Esta “adaptabilidade” pode ser conseguida com a introdução de um componente novo no sistema: o metaobjeto.

Normalmente atribuídas aos sistemas operacionais, tarefas como distribuição, segurança, persistência, concorrência, mobilidade e tolerância a falhas eram providas de forma transparente, ou quase transparente, para as aplicações. Estas facilidades foram aceitas e utilizadas da melhor forma possível, sem que nenhum questionamento, ou adaptação às reais necessidades da aplicação, fosse feito.

A adaptabilidade exigida pelos sistemas atuais, no entanto, tornou inviável manter a impossibilidade de alteração nas características destas tarefas. Sistemas operacionais mais flexíveis, como Delta [CPR+92] ou Grasshopper Kernel [Lin95] chegaram a ser desenvolvidos tentando suprir esta deficiência. Esta estratégia de flexibilizar os próprios sistemas operacionais, no entanto, se mostrou pouco eficiente por não ser suficientemente customizável nem extensível. Era necessário tratar estas tarefas de forma completamente diferente

da que estava sendo utilizada até então. Como primeiro passo, elas foram retiradas do sistema operacional e passaram a ser providas através de bibliotecas, o que as tornou mais flexíveis. Os sistemas que utilizavam estas funcionalidades, porém, precisavam conhecer as bibliotecas que as provinham, criando assim um acoplamento extra no sistema.

A descentralização destas tarefas, delegando-as a metaobjetos, por sua vez, torna-as flexíveis e extensíveis, além de ser uma solução transparente para os sistemas que as utilizam.

Uma arquitetura Meta se baseia em três conceitos básicos:

1. Reflectividade: característica que capacita um sistema computacional a "saber sobre si e decidir como agir com base neste conhecimento" [Mae87].
2. Metaprogramação: separação física entre código ligado diretamente ao domínio da aplicação, residente no nível base, e o código residente no nível meta que é responsável por supervisionar a execução do código de nível base.
3. Reificação: processo de representar, na forma de objetos, conceitos abstratos da linguagem de programação, como classes e métodos.

A reificação torna possível a supervisão do nível meta sobre o nível base, por promover a representação na forma de objetos do comportamento deste. Este comportamento, devido à sua complexidade, necessita de dois tipos de reificação [Fer89]:

- Reificação estrutural que representa como objetos aspectos estruturais do sistema como herança e tipos de dados.
- Reificação comportamental que representa como objetos a interação entre objetos.

A implementação de um ambiente de programação com uma arquitetura Meta traz a este as seguintes vantagens:

- Separação de conceitos: em ambientes de programação convencionais, o código funcional da aplicação é, muitas vezes, de difícil entendimento por possuir misturados em si códigos não-funcionais de controle e gerenciamento. Em ambientes de programação Meta, códigos não-funcionais podem ser separados fisicamente de códigos funcionais, não havendo a necessidade de referências entre um e outro;

- Aumento da produtividade: pela separação de conceitos, metaprogramação torna os programas mais estruturados e fáceis de serem implementados.
- Configurabilidade: arquiteturas Meta criam sistemas abertos [Kic96]. Novos controles podem ser implementados sem alteração no código da aplicação.
- Transparência: o acoplamento entre os níveis base e meta é transparente para o projetista do sistema.

Uma linguagem aberta é uma extensão de uma linguagem orientada a objetos comum. Ela permite ao programador escrever um programa de nível meta especificando como transcrever ou analisar um programa escrito na linguagem comum.

O programa de nível meta é escrito de acordo com uma interface chamada MOP (protocolo de metaobjeto), que expõe a estrutura interna do compilador com abstração de orientação a objetos. Depois é compilado, pelo compilador da linguagem aberta e (dinâmica ou estaticamente) ligado, como um *plug-in*, ao compilador propriamente dito da linguagem comum. O compilador resultante transcreve ou analisa o programa fonte (chamado de programa de nível base) como especificado pelo programa de nível Meta.

Um compilador de uma linguagem aberta consiste normalmente de três passos: pré-processamento, transformação da linguagem aberta para a linguagem comum, e a compilação da linguagem comum. O protocolo de metaobjetos é uma interface para controle do segundo passo e permite especificar como uma característica estendida pela linguagem aberta é transcrita para código da linguagem comum.

Características estendidas podem não consistir apenas de um programa nível meta mas também de um código de suporte a tempo de execução. Este deve prover classes e funções usadas pelo programa de nível base depois de transcrito para a linguagem base.

Na Figura 2.6, é esquematizado o processo de obtenção do código executável de um programa codificado em uma linguagem aberta.

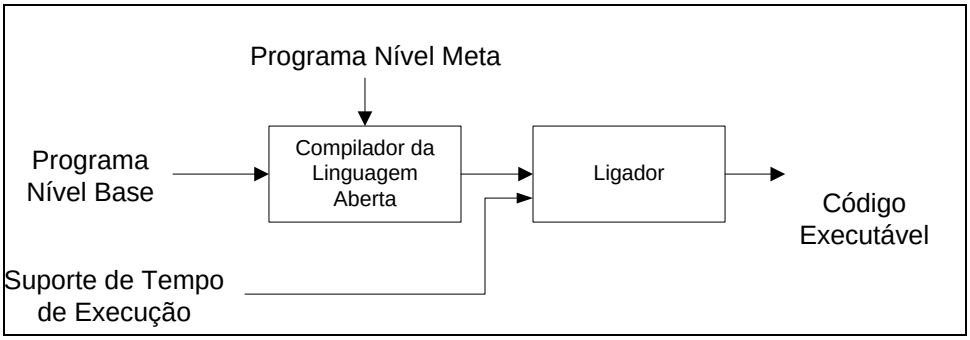

Figura 2.6 – Esquema de obtenção do código executável em linguagens abertas

Linguagens abertas são, portanto, ideais para a incorporação de facilidades como padrões e pronomes em que a completude do conjunto de funcionalidades é inatingível.

A linguagem aberta escolhida para a implementação dos pronomes explorados foi OpenC++ [Chip95] pela facilidade de se obter uma versão *open-source* bem documentada.

Os metaobjetos existentes em um MOP são, na sua maioria, para prover introspecção. Alguns, no entanto, representam o aspecto comportamental do programa e possibilitam a transformação do seu código.

O MOP de OpenC++ é formado pelos seguintes metaobjetos :

- metaobjetos *Ptree*: que representam a árvore sintática do programa;
- metaobjetos *Environment*: que representam ligações entre nomes e tipos;
- metaobjetos *TypeInfo:* que representam os tipos que aparecem no programa;
- metaobjetos *Class*: que representam definições de classes e controlam a transformação do código. Definindo-se subclasses desta, a transformação pode ser alterada; e
- metaobjetos *Member*: que representam membros da classe.

A programação do MOP em OpenC++ é feita em três passos: “(i) decidir o que o programa de nível base dever ver, (ii) definir o que deve ser transformado e que código de suporte de tempo de execução é necessário, e (iii) escrever um programa de nível meta que faça a transformação e escrever também o código de suporte de tempo de execução” [Chi96].

No programa de nível base de uma linguagem onde um pronome é incorporado, deve ser possível endereçar mensagens para o pronome

incorporado. Incorporando-se o pronome SENDER a C++, por exemplo, de ser possível escrever: SENDER->M(1);

O que deve ser transformado e o código de suporte de tempo de execução necessário variam de acordo com o pronome incorporado e serão detalhadamente expostos nos próximos capítulos. O comportamento do sistema após as transformações, no entanto, deverá ser:

- desviar a mensagem para o(s) objeto(s) destino(s) da relação mapeada pelo pronome; e
- garantir que, se este(s) não a tratar(em), a mensagem seja descartada sem que nenhum erro de execução seja gerado.

2.6

Trabalhos Relacionados

Apesar de pouco explorado, o problema de identificação de objetos possui algumas outras propostas de soluções.

O padrão *Role* proposto em [BRS+97], por exemplo, visa adaptar um objeto a diferentes clientes de forma transparente. Associando papeis a objetos, um objeto pode ser usado em diferentes contextos, sendo conhecido em cada um por um papel diferente. Embora a intenção de possibilitar a criação de códigos mais genéricos seja a mesma deste trabalho, o enfoque usado é oposto. Usando padrões de projetos para obter generalidade, a simplicidade e concisão da linguagem são mantidas, mas a responsabilidade de manter relações entre objetos (ou papéis) é transferida para o programador do sistema e o código torna-se difícil de se construir e manter. Além disso, “padrões possuem uma identidade lógica e conceitual no nível de projeto, mas esta identidade é perdida quando se passa do projeto para a implementação” [DR97].

Em [BLM98], “personalidades” são apresentadas como artefatos lingüísticos a serem adicionados a conceitos orientados a objetos habituais para explicitamente encapsular papéis da decomposição funcional no nível da implementação. Um papel (personalidade) é associado a uma classe através de uma cláusula específica e, a fim de que seja válida como uma personificação de uma personalidade, uma classe deve obedecer a regras bem definidas. O mesmo acontece para clientes de personalidades. Similares ao conceito de interfaces, encontradas em JAVA, personalidades possuem regras semânticas

mais fortes que pronomes. Quando uma classe personifica uma dada personalidade, precisa declarar suas intenções além de prover todos os métodos especificados na interface desta. De uma forma mais leve, pronomes permitem que classes que personificam uma dada personalidade só tratem mensagens de seu interesse.

Manter automaticamente a consistência do grafo de dependência entre objetos foi o enfoque de *Ducasse Richner* em [DR97] mas, diferentemente deste trabalho, sua principal preocupação foi capturar relações de interação e não generalizar a comunicação entre servidores e clientes. Conectores são promovidos a objetos de primeira ordem que representam a relação de interação entre componentes, não apenas descrevendo-a, mas realmente controlando a comunicação entre componentes. Este controle é programado através de uma sintaxe especial e a consistência do grafo de dependência automaticamente mantida. Mensagens, no entanto, continuam sendo enviadas apenas nominalmente, sem redução do acoplamento.

O modelo de comunicação adotado em OASIS [SLR+99] é também similar ao enfoque deste trabalho, sendo um sistema é representado por um conjunto de objetos autônomos interagindo em um modelo cliente-servidor. Tuplas cliente-servidor-serviço representam comunicações entre objetos. Clientes e servidores podem ser invocados por seus nomes ou por pronomes como SOMEONE e EVERYONE, que são comparáveis aos pronomes ALL e ANY propostos em [Coe92].

Programação orientada a assuntos foi proposta por Harrison e Ossher [SOP, HO93, OKH+96, OKK+96] como uma extensão do paradigma de orientação a objetos com o intuito de dar a estes perspectivas de assuntos. Com a introdução de perspectivas, conceitos podem ser claramente separados aproximando, como neste trabalho, o projeto de sua implementação.

Ao trabalho de Harrison e Ossher foi incorporado o conceito de filtros de composição, proposto por Aksit em [AT88, Ask89, Ber94, ATB96, CF], e de programação adaptativa, proposto em [Lie92, Lie96, Dem], para a obtenção do que hoje denomina-se programação orientada a aspectos [Cza98]. Tendo como principal objetivo a separação de conceitos e, com isso, a aproximação do projeto de sua implementação, apenas esta característica já seria suficiente para relacioná-la com este trabalho. No entanto, está na tecnologia utilizada para obtenção de abstrações para expressar aspectos o ponto de maior afinidade. Da mesma forma que pronomes, aspectos podem ser implementados:

- como uma biblioteca convencional;

- através de uma linguagem separada que pode ser implementada por um preprocessador, compilador ou por um interpretador;
- projetando-se uma extensão da linguagem para o aspecto.

As vantagens de se utilizar uma extensão especializada da linguagem original sobre uma linguagem separada ou uma biblioteca convencional são sumarizadas em [Cza98] e podem ser, também, aplicadas a pronomes. A extensão da linguagem original é obtida, da mesma forma que neste trabalho, através de transformações de código efetuadas por programação meta.