Identificando Objetos Através de Pronomes

Sylvia de Oliveira e Cruz

Carlos José Pereira de Lucena

José Lucas Mourão Rangel

e-mail: {lucena,rangel,sylvia}@inf.puc-rio.br

PUC-RioInf.MCC39/01 Agosto, 2001

Abstract: The need to know the object identification to request it's services is a restriction in the construction of object oriented system. To reduce this restriction we have proposed a form of generic reference to an object named pronoum. Pronouns allow the programmer to specify how to send a message without knowing the destination object by name. With pronouns, the visibility of an object increases without breaking the rule of encapsulation. With this feature decoupled server classes may be constructed and used in different contexts. Some pronouns are defined in this work. Their usefulness in the devolopment of object oriented systems is illustrated through examples. An implementation approach based on open languages is suggested

Keywords: Object Orientation, Reflection, Open Language, Metaobject Protocol.

Resumo: A necessidade de conhecer o identificador de um objeto para solicitar seus serviços é uma restrição à construção de sistemas orientados a objetos. Para minimizar esta restrição, propomos uma forma de se referenciar a um determinado objeto através de uma identificação genérica, chamada por nós de pronomes. Pronomes permitem que o programador especifique o envio de uma mensagem sem que o objeto destino seja nominalmente conhecido neste contexto. Com pronomes, a visibilidade de um objeto aumenta sem que regras de encapsulamento e visibilidade sejam quebradas. Com isso, classes servidoras completamente desacopladas de seus clientes podem ser construídas e utilizadas em diferentes contextos. Alguns pronomes são aqui definidos e suas utilidades demonstradas através de exemplos. Suas implementações são sugeridas utilizando-se linguagens abertas.

Palavras-chave: Orientação a objetos, Reflexão, Linguagem aberta, Protocolo de metaobjeto.

1. Introdução

A necessidade de conhecer o identificador de um objeto para solicitar seus serviços foi apontada por Shaw como uma séria restrição para a construção de sistemas orientados a objetos [SG96]. Apesar disso, poucas linguagens, padrões de projetos ou arquiteturas de software estão atacando este problema. Na realidade, soluções propostas por linguagens de programação envolvem relaxamento de conceitos de orientação a objetos como encapsulamento ou visibilidade. Padrões de projetos e arquiteturas de software, por outro lado, propõem soluções que exigem que programadores escrevam códigos bastante complexos para permitir referências genéricas (como recomendo nos padrões "Composite" e "Chain of Responsability" [G+95]).

Com pronomes, o programador pode especificar o envio de uma mensagem sem que o emissor conheça a identificação o receptor. A idéia é que o envio de uma mensagem poderia ser especificada para um pronome e, em tempo de execução, esta seria desviada para o objeto, ou grupo de objetos, associados a este pronome.

Considerada com a solução para problemas de reuso de software, a metodologia orientada a objetos começou a ser fortemente questionada quando começou a ser utilizada em sistemas grandes e complexos. Este tipo de sistema precisava de novas e mais complexas abstrações, levando a criação de relações mais frágeis que herança e agregação. A criação explícita destas relações transformou sistemas complexos em verdadeiras "teias" de associações, aumentando consideravelmente o acoplamento entre classes e dificultando o reuso. Por causa disto, embora orientação a objetos continue a ser extensamente utilizada, esta utilização é feita em um nível bem abaixo de todo seu potencial. Na prática, reuso de classe não é priorizado nos objetivos finais de um projeto, principalmente quando este é inter-sistemas (entre classes de sistemas diferentes). Encapsulamento, a regra básica de orientação a objetos para a criação de sistemas adaptáveis, é também constantemente violado. A redução da necessidade de associações, sem quebra de encapsulamento e sem que isso impossibilite a comunicação entre objetos, é o objetivo final deste trabalho. Acreditamos que pronomes, como descritos aqui, permitirá que reuso possa ser melhor explorado, tornando mais próximo o sonho de utilizar orientação a objetos em todo seu potencial.

Embora freqüentemente falemos de projeto de sistemas, associações e acoplamento, a ênfase deste trabalho é em linguagens de programação. Isso porque acreditamos ser a redução da distância entre projeto e implementação uma necessidade para a simplificação da implementação final de sistemas complexos. A crescente diversidade de requisitos não funcionais, como distribuição e concorrência, a que projetistas de software têm que conviver enfatiza a relevância desta redução. Apesar da crescente variedade de ferramentas CASE que suportam geração automática de código, a distância entre especificação de projeto e sua realização final na linguagem alvo não foi reduzida. Estas ferramentas normalmente diluem as especificações de projeto pelo código de tal forma que, muitas vezes, nem o próprio projetista é capaz de identificá-las.

Este trabalho é estruturado da seguinte forma: na seção 2, pronomes são descritos e sua utilidade explorada através de exemplos de seu uso em algumas arquiteturas de software e padrões de projetos; na seção 3, algumas reflexões sobre implemantação são feitas; na seção 4, este trabalho

é comparado com trabalhos anteriores com objetivos similares. Finalmente, na seção 5, descrevemos o estágio presente do trabalho e discutimos seu futuro.

2. Comunicação baseada em pronomes

Em ambientes orientados a objetos, estes são organizados em árvores de agregação. Nestas, um objeto conhece apenas seus filhos e, com isso, a troca de mensagens pode ser feita apenas em uma direção: de pai para filho. Para que mensagens possam ser trocadas em outras direções, referências cruzadas precisam ser criadas, o que aumenta sensivelmente o acoplamento do sistema.

Para reduzir a necessidade de referências cruzadas sem quebra de encapsulamento ou visibilidade, propomos a especificação genérica de pronomes para representar objetos específicos. Através de pronomes, mensagens podem ser despachadas para objetos sem que seus identificadores sejam expostos.

Muitas linguagens orientadas a objetos conhecidas já usam um pronome para se referenciar especificamente ao objeto corrente da execução. Este pronome é um nome (THIS, em Java ou C++; SELF, em Smalltalk) através do qual o programador se refere a um objeto específico (que pode mudar de uma ativação de um método para outra ativação do mesmo método, apesar do código executado ser o mesmo). Nossa proposta estende este conceito, permitindo pronomes para descrever outros objetos.

Existe um paralelo entre os pronomes que propomos e o uso de pronomes em linguagem natural. No entanto, não podemos permitir em linguagem de programação o tipo de ambigüidade normalmente encontrada em linguagens naturais, onde o contexto é usado para associar um pronome ao que ele realmente se refere. Em programação orientada a objetos existe apenas um contexto claramente definido por todo o tempo, que é o objeto corrente. Qualquer informação extra precisa ser explicitamente dada. Esta é a razão porque linguagens de programação normalmente oferecem um pronome, que descreve exatamente o objeto corrente.

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. Entretanto, pouca vantagem 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. Programadores já podem, na realidade, fazer isso sem a adição de nenhuma nova facilidade a linguagem, a menos da introdução de algumas construções como açúcar sintático que simplifiquem o uso de funções que determinem coleções de objetos.

Os pronome nos quais estamos interessados são aqueles que descrevem objetos ou conjunto de objetos que não podem ser determinados no contexto do objeto corrente. Suponhamos que nós queiramos, por alguma razão, 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, devemos de alguma forma salvar (uma referência para) o objeto criador. Toda vez que um objeto é criado, podemos passar uma referência para o criador (como um parâmetro do construtor) 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 for necessário.

Como outro exemplo, suponha que queiramos enviar uma mensagem a todos os objetos modelados por uma determinada classe C. Não há como fazer isso diretamente, exceto se adicionarmos um repositório de objetos a cada classe e se modificarmos todo construtor da classe C para adicionar (uma referência para) os objetos criados ao respectivo repositório. Se isso for feito, podemos então iteragir sobre os objetos deste repositório e enviar a mensagem a cada um destes. Se a linguagem, no entanto, inclui um pronome ALL, tudo que precisa ser feito é o envio da mensagem a ALL. A vantagem é que nenhum código extra é necessário e a desvantagem é que para toda classe, o sistema deve, em todos os casos, manter uma lista de objetos criados e não destruídos ainda, ou explicitamente ou através de algum mecanismo implícito de coletor de lixo.

Os exemplos anteriores mostram quais as vantagens e as desvantagens 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 descritos, 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.

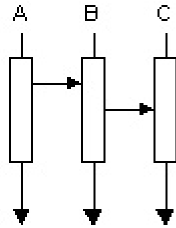
O uso de pronomes torna visível relações implícitas que existem entre objetos de um sistema, como a descrita entre um objeto e seu criador. É claro que não esperamos nomear através de pronomes todas as possíveis relações entre objetos. Primeiro porque delimitar este conjunto não é uma tarefa fácil e segundo porque, pela desvantagem descrita acima, qualquer sistema que tentasse fazê-lo tornar-se-ia extremamente pesado, pois precisaria manter um conjunto muito grande de relações implícitas. Ao invés disso, propomos alguns pronomes úteis para a implementação de arquiteturas de software e padrões de projetos a fim de validar o conceito de pronomes e mostramos sua implementação de forma flexível para que linguagens possam incorporar, de forma similar, outros pronomes.

2.1. O objeto principal

Programas orientados a objetos são essencialmente mediados [BMR+96] e a raiz da árvore principal de execução é o mediador principal do programa. Nestes sistemas, mensagens podem ser trocadas apenas de pai para filho. Desta forma, ninguém pode enviar mensagens para o objeto principal, o que reduz bastante suas funções mediadoras. Identificando-o pelo pronome MAIN, todos os objetos do sistema podem enviar-lhe mensagens, permitindo-lhe um controle maior do sistema.

O objeto principal é criado pelo ambiente de execução e não é destruído durante toda a execução, sendo, portanto, o lugar ideal para se definir, como objetos agregados, "singletons"[G+95] usados pela aplicação. A possibilidade de enviar mensagens diretamente para o objeto principal torna estes objetos facilmente acessíveis por todo o sistema, sem que este precisem ser globais. Um exemplo desta facilidade é a implementação de relógios lógicos usando o pronome MAIN. Relógios lógicos são usados para ordenar eventos em programação distribuída [Lam78]. Esta odenação é a base para muitos protocolos de sincronização descentralizados [And91].

Processos em um programa distribuído executam ações locais e ações de comunicação. Estas são envio e recepção de mensagens e afetam a execução de outros processos já que transferem informação e são a base para o mecanismo de sincronização.



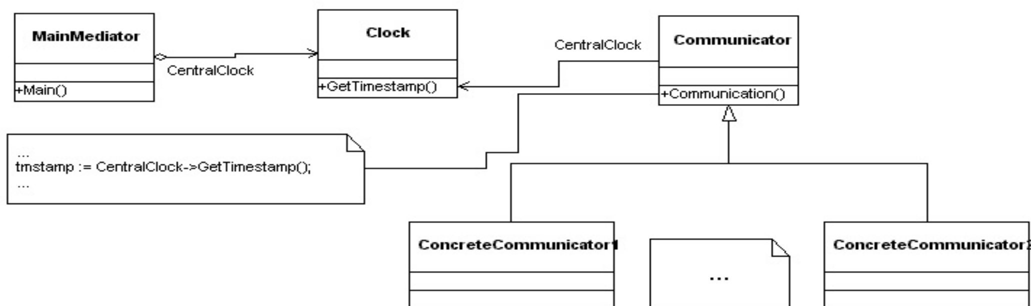
A figura mostra três processos em um programa distribuído executando ações locais e de comunicação. Se estes processos executassem apenas ações locais, não existiria forma de saber a ordem relativa em que as ações eram executadas. No entanto, se A envia uma mensagem para B e B subseqüentemente envia uma mensagem para C, como mostra a figura, existe uma ordenação total entre as ações de comunicação: O envio por A acontece antes do recebimento por B, que acontece antes do envio por B, que, por sua vez, acontece antes do recebimento por C.

Existe uma ordenação total entre eventos que afetam outros, como descrito acima. Existe, no entanto, apenas uma ordenação parcial em toda a coleção de eventos em um programa distribuído. Isto porque seqüências de eventos não relacionados – por exemplo, ações de comunicações entre conjuntos diferentes de processos – podem ocorrer antes, depois ou concorrentemente com cada outro.

Se existir um único relógio centralizado, ações de comunicação poderiam ser totalmente ordenadas atribuindo-se a cada evento um timestamp único. Em particular, quando um processo envia uma mensagem, ele poderia ler o relógio e anexar o valor lido a esta. Por outro lado, quando um processo recebe uma mensagem, ele poderia ler o relógio e registrar o tempo no qual o evento recebido ocorreu [And91].

Este relógio pode ser implementado tanto como um relógio físico quanto como um relógio lógico (simples contador que é incrementado quando um evento ocorre), não tendo esta escolha influência no ponto de interesse desta discussão.

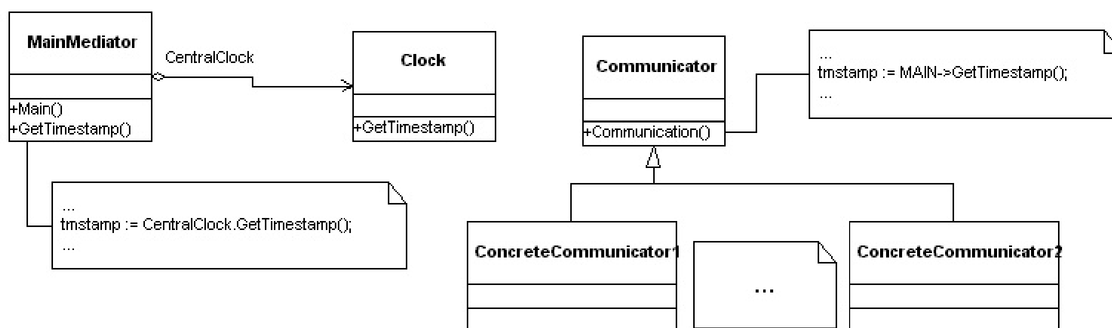
A implementação de um relógio central exige que todos os objetos que o utilizam (que executam alguma ação de comunicação) conheça-o. Tipicamente, ele seria implementado como um objeto global ou, em ambientes orientados a objetos mais puros, como atributo do objeto principal do sistema e tendo sua referência passada para todos os objetos que executem ações de comunicação. A estrutura da solução é mostrada a seguir:



- O objeto principal do sistema cria o relógio central em sua inicialização;
- A referência ao relógio central é disponibilizadas para todos os objetos comunicadores;
- Quando uma ação de comunicação é executada, o timestamp corrente é obtido por sua referência e anexada à mensagem.

A propagação da referência observada nesta solução aumenta bastante o acoplamento do sistema além de forçar a adoção de algum mecanismo de proteção, a fim de evitar alguma má utilização da referência.

O acesso direto e genérico ao objeto principal do sistema, através do pronome MAIN e a implementação deste como mediador [G+95] entre o relógio central e os objetos que executam ações de comunicação resolve estes problemas pois evita a propagação da referência. A estrutura da solução é mostrada a seguir:



- O objeto principal do sistema cria o relógio central em sua inicialização;
- Quando uma ação de comunicação é executada, o timestamp corrente é obtido pelo pronome MAIN.
- O objeto principal do sistema obtem o timestamp corrente por CentralClock.

Nesta solução, a referência ao relógio central não é diretamente disponibilizado a outros objetos do sistema, sendo qualquer acesso a ele feito através de um mediador (o objeto principal do sistema).

2.2. O conjunto de todos os objetos do sistema

Recentemente, tem sido considerável o interesse em uma técnica de integração referenciada como invocação 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, o sistema por ele mesmo invoca todos os tratadores que tenham sido registrados para o evento.

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, e facilita a evolução do sistema [SN92], porque um módulo pode ser trocado por outro sem afetar as interfaces dos módulos que implicitamente dependem dele.

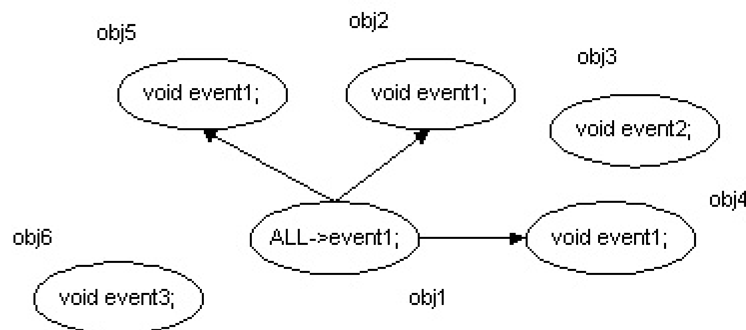
Visando estas vantagens, muitos sistemas usam invocação implícita como forma principal de composição. A forma de uso, no entanto, pode variar de sistema para sistema, sendo possível agrupá-las em três categorias.

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 através de recursos providos pelo sistema operacional. Poucas aplicações, no entanto, podem conviver com o overhead gerado por processos separados, limitando o uso desta técnica.

A segunda, frameworks de aplicação, incluem sistemas que usam o padrão observer [G+95] como registro-anúncio de eventos ou usam um mediador de eventos 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 invocados implicitamente quando um evento ocorre ou uma relação é alterada.

A solução usando o pronome ALL, para representar o conjunto de todos os objetos do sistema, pode ser incluído nesta última categoria. Nesta solução, um evento é anunciado através de uma mensagem para o pronome ALL e seu encaminhamento é feito automaticamente para cada objeto que o trate, isto é, para todo objeto que tenha registrado interesse neste.



Esta situação é ilustrada na figura acima. Nesta, um objeto (obj1) anuncia um evento (event1), enviando uma mensagem a todos os objetos que a tratam (obj2, obj4 e obj5). Obj3 e obj6 não são informados porque não manifestaram interesse no evento. Isto é, não o tratam.

A principal diferença entre a solução usando pronome e a solução usando triggers é onde a semântica de invocação é amarrada com o evento. Enquanto em triggers a semântica é amarrada nos servidores (nos tratadores dos eventos), com pronomes a semântica é amarrada nos clientes (no comando de envio da mensagem). A conseqüência imediata é que na solução com pronomes, um único tratador de eventos pode ser usado com diferentes semânticas de invocação.

Uma especialização possível é restringir o conjunto de todos os objetos ao conjunto de todos os objetos modelados por uma determinada classe. Nesta, a classe restritiva é especificada no comando de envio.

2.3. Um objeto qualquer

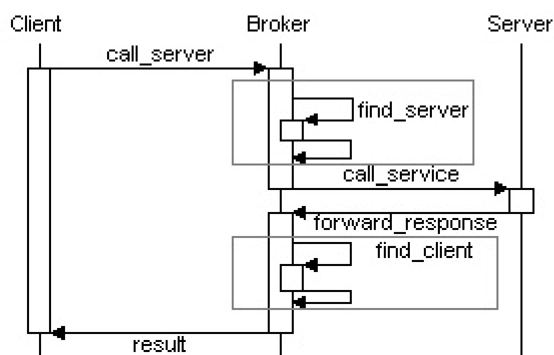
O pronome seguinte não representa um objeto, ou conjunto de objetos, específico, mas um objeto qualquer. Uma mensagem endereçada ao pronome ANY é endereçada ao primeiro objeto disponível para tratá-la. A mensagem é, na realidade, oferecida a todos os objetos e enviada ao primeiro objeto que atender ao oferecimento solicitando-a. Um especialização possível é restringir o conjunto de objetos destino para qualquer objeto de uma determinada classe.

Este tipo de envio deve ser usado quando o serviço puder ser executado por qualquer objeto (na maioria das vezes por qualquer objeto de uma determinada classe) com o objetivo de balanceamento de carga do sistema ou, simplesmente, para desacoplar componentes do sistema que puderem repassar seus serviços.

O padrão arquitetural Broker [BMR+96], usado para estruturar softwares distribuídos com desacoplamento de componentes que interagem por invocações remotas, propõe exatamente isto. Um componente mediador é introduzido para coordenar comunicações entre clientes e servidores, conseguindo desta maneira melhor desacoplamento destes. Servidores registram-se com o mediador, e disponibilizam seus serviços para clientes através de interfaces de métodos. Clientes acessam a funcionalidade de servidores enviando pedidos via mediador. Uma tarefa do mediador é localizar o servidor apropriado, repassando o pedido para este e transmitindo resultados e exceções de volta ao cliente.

O cenário simplificado acima mostra o comportamento do padrão quando um cliente envia uma solicitação de serviço a um servidor.

Apesar de bastante complexo, vamos nos concentrar em um aspecto deste padrão: a tarefa do mediador de localizar servidores e clientes. Este é exatamente o pedaço do cenário envolto em linhas cinzas na figura a seguir.



Nos passos para implementação do padrão é definido que: “O mediador deve manter um repositório para localizar outros mediadores ou gateways para os quais ele repassa 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 é feito na implementação CORBA da Iona [Iona95] através da seguinte seqüência de passos:

1. O componente mediador local é contactado para gerar uma lista de nomes de hosts. Nesta lista, os nomes são arranjados em uma ordem randômica.
2. O 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 randômica permite que um componente mediador não seja carregado por requisições, enquanto outros ficam inativos durante a maior parte do tempo.

Para implementar estes passos, o componente mediador precisa manter uma lista de hosts (com outros componentes mediadores) e gerar, randomicamente, uma ordem de acesso a ela.

Usando o pronome ANY, o mecanismo de localização de servidores pode ser implementado através de um simples envio de mensagem. A solicitação de serviço pode ser enviado diretamente a qualquer mediador, ou a qualquer servidor se a implementação CORBA realizar a variante "Direct Communication Broker System" [OMG92], que responde ao estímulo esperado. Não se torna, portanto, mais necessário manter uma lista de todos os mediadores nem gerar uma lista de acesso randômica a esta.

A mensagem é enviada ao primeiro mediador, ou servidor, em estado de espera. A variante do pronome ANY (qualquer objeto de uma determinada classe) garante o envio apenas a mediadores em uma implementação padrão de CORBA.

2.4. O objeto que declarou o objeto corrente da execução

Projetos orientados a objetos encorajam a distribuição de comportamento entre objetos. Esta distribuição pode levar a uma estrutura com muitas conexões entre objetos onde, no pior caso, cada objeto precise conhecer cada um dos outros.

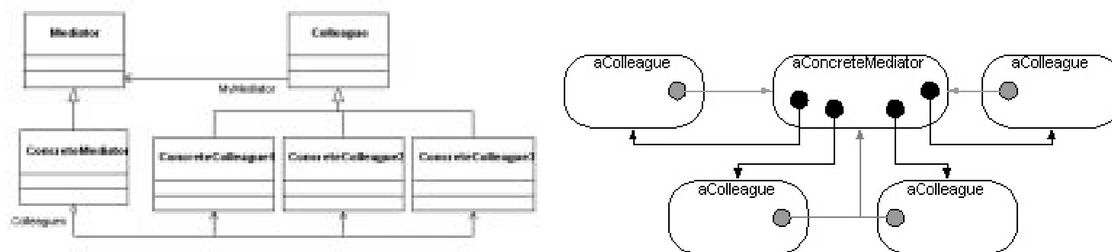
Para minimizar estes problemas algumas relações entre objetos, como a relação de pai e filho, podem ser expressas por pronomes. Definida como "o pai é um objeto que, em sua estrutura de declaração, contem a declaração de seus filhos", a relação é estabelecida quando um objeto é declarado.

O objeto pai, porque declarou o objeto filho, conhece-o nominalmente, podendo enviar-lhe mensagens diretamente. O objeto filho, por sua vez, não conhece o objeto pai sendo necessário o uso de referências cruzadas ou de códigos que permitam referências genéricas para que possa comunicar-se com ele. Para evitar estas soluções complicadas e, apesar disto, permitir a comunicação de um objeto filho para um objeto pai, o pronome PARENT pode ser usado para representar o objeto pai no contexto do objeto filho.

O padrão "Mediator" [G+95] tenta também 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 interação e os objetos conhecem apenas o mediador, reduzindo assim o número de conexões.

A estrutura do padrão e uma estrutura de objetos típica são mostrados a seguir:



Embora muito usado em softwares baseados em componentes, o padrão força o programador a manter referências cruzadas e amarra o componente a um tipo de mediador, ou a uma família de mediadores.

Esta estrutura pode ser simplificada com a existência e utilização do pronome PARENT. Mais especificamente, a referência cruzada dos objetos para o mediador pode ser eliminada, já que é mantida automaticamente pelo sistema e disponibilizada para referência através do pronome PARENT.

Sem a referência ao objeto mediador, este objeto pode ser modelado por qualquer classe (sem a restrição de estar na mesma árvore hierárquica), tornando as classes “Colleagues” (ou componentes) ainda mais reusáveis.

2.5. O criador do objeto corrente da execução

Outra boa candidata a ser expressa por um pronome é a relação de criador e criatura. Definida como “o criador é um objeto que, enquanto atendendo a uma solicitação de serviço, cria outro objeto, sua criatura”, a relação é estabelecida quando um objeto é criado.

Como o criador cria a criatura, conhece seu nome e pode lhe enviar mensagens diretamente. A criatura, por outro lado, não conhece seu criador e o envio de mensagens neste sentido exige o uso de referências cruzadas ou de complicados códigos que permitam referências genéricas. Para evitar estas soluções, o pronome CREATOR pode ser usado para representar o objeto criador no contexto da criatura e, com isto, permitir o envio de mensagens no sentido da criatura para seu criador.

A relação criador e criatura é usada para troca de mensagens quando objetos são criados dinamicamente, sem um controle central, para a execução de tarefas e valores precisam ser retornados para algum cálculo posterior. É este o caso, por exemplo, do algoritmo paralelo de divisão e conquista que será explicado para exemplificar o uso do pronome CREATOR.

A estratégia de divisão e conquista consiste de três passos principais. O primeiro é a divisão da entrada em partes de tamanhos iguais. O segundo é resolver recursivamente o subproblema definido por cada parte da entrada. O terceiro é combinar as soluções dos diferentes subproblemas em uma solução para todo o problema [JaJ92].

[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 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 trapezóide para aproximar a área de cada subintervalo.

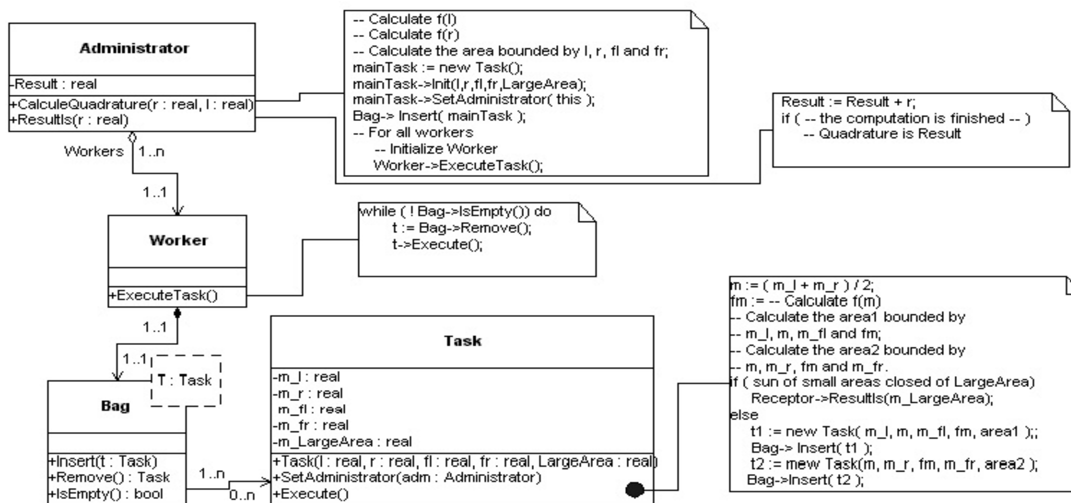
O problema pode ser resolvido estaticamente ou dinamicamente. A solução dinâmica, chamada quadratura adaptativa, começa com um intervalo de l até r e calcula o ponto médio m entre l e r. Calcula, então, as áreas de três trapezóides:

1. o maior delimitado por l, r, f(l) e f(r);
2. o primeiro menor delimitado por l, m, f(l) e f(m);
3. o segundo menor delimitado por m, r, f(m) e f(r).

Depois, a área do maior trapezóide é comparado com a soma das duas áreas menores. Se estas estão suficientemente próximas, a área do trapezóide maior é considerada uma boa aproximação da área sob f. Caso contrário, 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 que a solução de cada subproblema seja aceitável. As respostas dos subproblemas são, então, somados 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 introduzido para ser solucionado. Múltiplos processos trabalhadores pegam tarefas da bolsa e os processam, gerando eventualmente novas tarefas (correspondendo a subproblemas) que são inseridos na bolsa. O algoritmo termina quando todas as tarefas tiverem sido processadas.

A solução paralela. Se 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 e suas colaborações são mostradas a seguir:

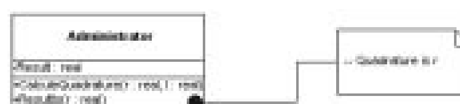


- O administrador gera o primeiro problema, insere-o na bolsa de tarefas, cria os trabalhadores e os coloca para trabalhar.
- Trabalhadores compartilham um canal simples, a bolsa, que contem os problemas a serem resolvidos. Enquanto estão executando tarefas (enquanto a bolsa não ficar vazia), os problemas resolvidos são removidos da bolsa e solucionados.
- Um problema é composto pelos valores extremos do trapezóide e pela área deste. A solução divide o trapezóide em dois e calcula suas áreas, verificando se a área do trapezóide original é uma boa aproximação da área real da função. Se sim, a área do trapezóide original é retornado ao objeto administrador. Se não, duas novas tarefas são criadas para os dois trapezóides menores e inseridas na bolsa.
- O método ResultIs do administrados recebe resultados, verificando se o processo terminou.

No último passo do algoritmo está sua principal dificuldade: a determinação da condição de 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 em estado de espera; esta situação, no entanto, é difícil de ser detectada. No código apresentado, por exemplo, seria necessário que os trabalhadores controlem e exportem seus estados para que o administrador possa detectar o fim do algoritmo.

A solução usando a relação criador e criatura. Com a relação criador e criatura, algumas facilidades são introduzidas e o problema da determinação do término é resolvido. Ao contrário da solução anterior, agora as tarefas não retornam o resultado para o objeto administrador mas sim ao objeto que o gerou. O retorno do resultado, no método Execute() da classe Task, deve ser substituído por CREATOR->ResultIs(m_LargeArea).

Então, o objeto administrador gera uma tarefa e espera o retorno de um resultado e o algoritmo termina quando este é retornado. O código do método ResultIs da classe Administrator é simplificado para:



O atributo Result, da classe Administrator na solução original, não é mais necessário já que resultados não são mais acumulados aqui.

Por sua vez, cada tarefa que gera outras duas tarefas espera receber dois resultados e precisa ter um método para recebê-los. Este acumula o resultado em um novo atributo (Result) e controla o número de chamadas em outro novo atributo (nResult). Dois resultados são acumulados e a soma destes retornada ao objeto que criou a tarefa. O estado do trabalhador também não é mais necessário.

Além da solução do problema de determinação de término, o objeto administrador não precisa ser publicado nem armazenado pelas tarefas, diminuindo o acoplamento do sistema e a necessidade de mecanismos de proteção para a referência.

2.6. O emissor da mensagem corrente

A terceira e última relação, exemplificada aqui, é a relação emissor-receptor. Definida como “o emissor é o objeto que enviou a mensagem que está sendo atualmente tratada pelo objeto corrente da execução, seu receptor”, a relação é estabelecida quando uma mensagem é enviada.

A visibilidade da relação é dada do emissor para o receptor. O pronome SENDER pode ser usado para representar o objeto emissor no contexto do receptor e, com isto, permitir que a relação seja percorrida também no sentido contrário.

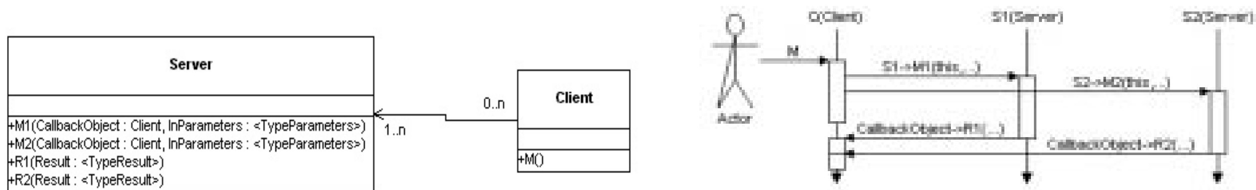
Mensagens para o emissor podem ser enviadas para retorno de resultados quando mensagens assíncronas (sem retornos ou parâmetros de saída) são necessárias. Esta situação é exemplificada com uma discussão sobre a implementação do padrão “Distributed Callback” [MM97].

“Distributed Callback” é um padrão comportamental bastante usado no contexto de arquiteturas de sistemas distribuídos que usam CORBA. Apesar de ser aplicável em outros contextos, é neste que o padrão é aqui discutido.

O problema focado pelo padrão consiste em clientes que necessitam de serviços de servidores, mas precisam continuar seus processamentos enquanto os servidores fazem suas partes. Alguma forma de mecanismo deve ser usado para que os objetos servidores possam comunicar seus resultados, que podem ser enviados de volta ao cliente em qualquer momento e em qualquer ordem.

Uma solução para esta situação 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. Os parâmetros de saída originais devem fazer parte de uma segunda

operação “oneway” que é chamada 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, ele usa o objeto de retorno como receptor da segunda operação “oneway”, enviando os parâmetros de saída de volta ao cliente. Esta solução é mostrada a seguir.

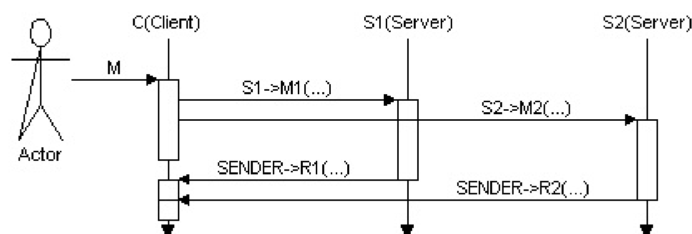


Alguns comentários sobre esta solução:

- A menos que algum mecanismo específico de proteção de parâmetro exista, e que este seja usado na relação com o objeto de retorno, sua referência pode ser propagada pela operação do servidor. Isto pode gerar resultados indesejáveis, como um particular objeto de retorno de um cliente ser usado 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 der 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 poderem simplificar a redefinição do método, uma grande dose de disciplina é necessária para evitar erros de programação, especialmente se o número de métodos que precisem ser rescritos for grande.
- Objetos de retorno amarram servidores a clientes específicos, reduzindo suas generalidades.

A solução usando pronomes mantém todos os benefícios apresentados no padrão original, e evita seus principais problemas. Em particular, também permite que servidores levantem exceções e garantem a generalidade do servidor.

Usando a relação emissor-receptor, no entanto, a comunicação do servidor para o cliente pode ser estabelecida facilmente. O objeto de retorno é substituído pelo pronome SENDER, deixando o servidor completamente independente do cliente. Além disso, não apresenta o perigo da propagação da referência. A nova visão dinâmica é mostrada a seguir:



Nesta solução, a ordem em que clientes recebem mensagens dos servidores deve ser irrelevante, ou alternativamente um algoritmo de escalonamento deve existir na classe que modela a fila de mensagem do cliente para garantir a necessária disciplina na recepção.

3. Reflexões sobre Implementação

Pronomes podem ser incorporados ao kernel de qualquer linguagem orientada a objetos. A determinação de quais pronome incorporar, no entanto, não é uma tarefa imediata.

Cada pronome, pela característica da relação que modela, facilita a implementação de um determinado tipo de aplicação. Por um lado, como uma linguagem é, na maioria das vezes, de propósito geral, quanto mais pronomes incorporar melhor pois um maior número de aplicações será beneficiado. Por outro lado, no entanto, 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 gerenciada. O ponto ótimo seria, então, um número mínimo de pronomes que facilitem um número máximo de aplicações. A determinação deste ponto ótimo não é, no entanto, tarefa trivial.

3.1 Linguagens abertas

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.

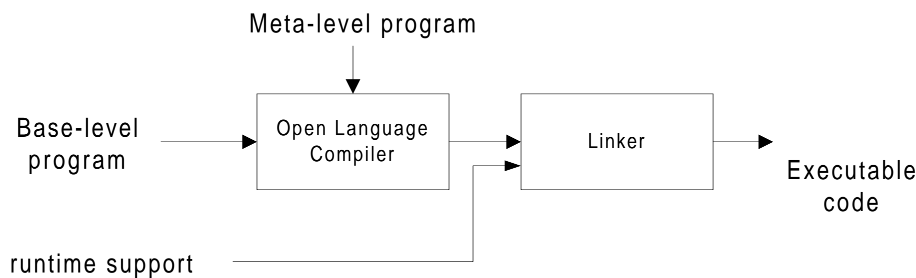
O programa de nível meta é escrito de acordo com uma interface chamada protocolo de metaobjeto, que expõe a estrutura interna do compilador com abstração de orientação a objetos. Este programa é compilado, pelo compilador da linguagem aberta e (dinâmica ou estaticamente) ligado, como um plug-in, ao compilador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

A compilação de uma linguagem aberta consiste normalmente de três passos:

1. Preprocessamento;
2. Transcrição da linguagem aberta para a linguagem comum; e
3. 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.



Linguagens abertas são, portanto, ideais para a incorporação de facilidades como padrões e pronomes em que a completude é, como comentado anteriormente, muito difícil de ser atingida.

3.2 Pronomes em OpenC++

A programação do MOP em OpenC++ [Chi95] é feita em três passos: “(1) decidir o que o programa de nível base deve ver, (2) definir o que deve ser transcrito e que código de suporte de tempo de execução é necessário, e (3) escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte de tempo de execução” [Chi96]. Implementamos os pronomes sugeridos anteriormente seguindo estes passos.

3.2.1 MAIN

Em C++, a função main() (onde a execução se inicia) é global, não sendo membro de nenhum objeto. Desta forma, não há objeto principal em C++, não fazendo sentido o pronome MAIN. Como C++ permite objetos globais, a função de mediador principal pode ser executada por um objeto deste tipo.

3.2.2 SENDER

(1) O que o programa de nível base deve ver ?

- Envio de mensagem ao pronome SENDER apenas.

(2) O que deve ser transcrito e que código de suporte a tempo de execução é necessário ?

A referência ao pronome SENDER deve ser transcrita para a referência ao respectivo objeto (emissor da mensagem correntemente tratada).

SENDER -> msg(); => SenderObj -> msg();

Para que esta transcrição possa ser feita, a referência ao objeto emissor da mensagem corrente (SenderObj) deve ser conhecida neste contexto. Chamadas de funções (e suas respectivas assinaturas) são, para isso, transformadas com a introdução de mais um parâmetro que seria o objeto corrente da execução no momento da chamada (objeto emissor no ambiente da função chamada).

Obj -> proc1(...); => Obj -> proc1(this, ...); e

void proc1(...) => void proc1(CObject* SenderObj, ...);

(3) Escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte a tempo de execução.

Para implementar pronomes, definimos uma nova metaclasse PronounClass, que é uma metaclasse para todas as classes. PronounClass redefine a função TranslateMemberCall(), para obter a transcrição das chamadas de função sobre objetos. Quando o objeto receptor é o pronome SENDER, este é substituído pelo parâmetro SenderObj e, como esta é também uma chamada de função, o objeto corrente da execução é inserido na lista de argumentos. Quando o objeto receptor não é o pronome SENDER, apenas o objeto corrente da execução é inserido na lista de argumentos.

A função TranslateFunctionCall() é redefinida, para que o objeto corrente da execução possa ser introduzido na lista de argumentos em chamadas de função sem o objeto receptor especificado.

A função TranslateMemberCall() é redefinida, para que o objeto corrente da execução possa ser introduzido na lista de argumentos em chamadas de função em que o objeto receptor especificado é o pronome this.

A função TranslateClass() é redefinida para obter a alteração nas assinaturas das funções. Em toda função membro da classe, é introduzido um parâmetro novo nomeado SenderObj.

3.2.3 CREATOR

(1) O que o programa de nível base deve ver ?

- Envio de mensagem ao pronome CREATOR apenas.

(2) O que deve ser transcrito e que código de suporte a tempo de execução é necessário ?

A referência ao pronome CREATOR deve ser transformada na referência ao respectivo objeto (objeto criador do objeto corrente da execução).

CREATOR -> msg(); => m_CreatorObj -> msg();

Para que esta transformação possa ser feita, a referência ao objeto criador do objeto corrente da execução (m_CreatorObj) deve ser conhecida em seu contexto. Um novo membro é introduzido e atualizado, por uma também nova função, toda vez que um novo objeto é criado.

A criação de um objeto depende da natureza de sua declaração e do contexto onde esta aparece. Objetos declarados como referências, são criados explicitamente através de um comando new. A transcrição deste comando é feita, então, sucedendo-o com a atualização do criador do objeto recém criado com o objeto corrente da execução (que executou o comando new).

Obj1 := new <class1>;   é sucedido pelo comando

Obj1.SetCreator(this);

Objetos não declarados como referência, têm sua criação dependente do contexto da declaração. Quando este é um objeto membro, sua criação se dá quando o objeto pai é criado e, portanto, seu criador deve ser o mesmo do objeto pai pois foi este que provocou sua criação. Esse efeito cascata é conseguido na implementação da função SetCreator(). Quando o objeto é um objeto local ou um parâmetro de método, sua criação se dá quando o método é invocado e seu criador

deve ser o objeto que chamou o método pois, mais uma vez, foi quem provocou sua criação. Esta associação é conseguida com a introdução de um prólogo em cada método[1].

(3) Escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte a tempo de execução.

A metaclasse PronounClass redefine a função TranslateClass(), para que o novo membro e a nova função sejam introduzidos na classe.

No código de SetCreator(), além de associar o objeto recebido ao novo membro (m_CreatorObj) este é propagado, recursivamente, a todo objeto membro. Nesta propagação, ao objeto criador de cada objeto membro é associado o mesmo objeto criador do objeto pai, utilizando-se as respectivas funções SetCreator().

A implementação de SetCreator() é introduzida redefinindo-se a função TranslateMemberFunction(). Nesta, é introduzido também o prólogo onde, ao criador de objetos locais e parâmetros, é atribuído o objeto chamador da respectiva função.

Para referências, criações explícitas são feitas. Da mesma forma, a chamada a SetCreator() deve ser, também, explícita. Isso é feito na redefinição da função TranslateAssign(), onde a chamada a SetCreator() é inserido após o comando, caso a expressão deste seja new.

A metaclass PronounClass redefine ainda a função TranslateMemberCall(), para obter a transcrição das chamadas de função sobre objetos. Quando o objeto receptor é o pronome CREATOR, este é substituído pelo atributo m_CreatorObj já preenchido.

3.2.4 PARENT

(1) O que o programa de nível base deve ver ?

- Envio de mensagem ao pronome PARENT apenas.

(2) O que deve ser transcrito e que código de suporte a tempo de execução é necessário ?

A referência ao pronome PARENT deve ser transcrito para a referência ao respectivo objeto (objeto que declarou o objeto corrente da execução).

PARENT -> msg(); => m_ParentObj -> msg();

Para que esta transcrição possa ser feita, a referência ao objeto que declarou o objeto corrente da execução (m_ParentObj) deve ser conhecida em seu contexto. Para tanto, um novo membro é introduzido e atualizado, por uma também nova função, toda vez que um novo objeto é criado. O processo de criação de objetos foi descrito anteriormente e a associação do objeto pai é bastante semelhante a do objeto criador. As diferenças são dadas pela função que associa o objeto pai ao novo membro (m_ParentObj), SetParent(), e pelo objeto passado a esta para ser associado. Comandos new são, da mesma forma, sucedidos por uma chamada a SetParent() e, como um

---

[1] O emissor da mensagem é disponibilizado no contexto do método tratador da mensagem através de parâmetro extra como descrito na transcrição do pronome SENDER.

objeto para criar outro explicitamente deve conhecê-lo nominalmente (ter declarado-o), o objeto corrente da execução é passado como objeto declarador do recém criado objeto.

Obj1 := new <class1>;   é sucedido pelo comando

Obj1.SetParent(this);

De forma similar ao SetCreator(), a função SetParent() associa a informação de declarador a todo objeto membro não declarado como referência. Diferentemente da função SetCreator(), que repassa aos objetos membros o seu próprio criador como criador destes também, a função SetParent() passa o objeto corrente aos objetos membros como declarador destes. Para objetos locais e parâmetros, também de forma similar, a associação é feita em um prólogo do método em questão.

(3) Escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte a tempo de execução.

Como pode ser visto no item (2), a implementação de PARENT é bem parecida com a já apresentada para o pronome CREATOR. Não apresentaremos aqui detalhadamente todos os passos como na descrição dada a implementação do pronome CREATOR, nos limitando a citar apenas as diferenças entre elas.

Na redefinição da função TranslateClass(), o novo membro m_ParentObj e a nova função SetParent() são introduzidos na classe. A implementação de SetParent() é obtida redefinindo-se a função TranslateMemberFunction(). Nesta, é introduzido também o prólogo onde ao parent de objetos locais e parâmetros é atribuído o objeto chamador da respectiva função.

Para referências, no entanto, criações explícitas devem ser feitas. Da mesma forma, a chamada a SetParent() deve ser, também, explícita. Isso é feito na redefinição da função TranslateAssign(), onde a chamada a SetParent() é inserido após o comando new.

A metaclass PronounClass redefine ainda a função TranslateMemberCall(), para obter a transcrição das chamadas de função sobre objetos. Quando o objeto receptor é o pronome PARENT, este é substituído pelo atributo m_ParentObj já preenchido.

3.2.5 ALL

(1) O que o programa de nível base deve ver ?

- Envio de mensagem ao pronome ALL apenas.

(2) O que deve ser transcrito e que código de suporte a tempo de execução é necessário ?

O envio de uma mensagem através do pronome ALL é transcrito para n envios, um para cada objeto do sistema que trate a mensagem em questão. O armazenamento dos objetos vivos é feito em cada classe do sistema em um membro estático inserido nesta.

Além do novo membro, funções estáticas são introduzidas também para prover a manutenção e a visualização desta lista. São elas :

- InsertModeledObject() : para a inserção de um objeto na lista de objetos modelados pela classe.

Deve ser chamado toda vez que a classe de um objeto é definida (em sua criação) ou alterada (em uma atribuição).

Em caso de criação explícita, o comando new deve ser sucedido por um comando que insira o recém criado objeto e todos os seus membros nas listas de suas respectivas classes. Esta inserção em cascata é executada por um novo método inserido na classe do objeto. Assim,

obj1 = new <class1>   é sucedido por

obj1-> InsertNewModeledObject();

Além de objetos membros, objetos declarados como parâmetros ou locais a métodos são criados também implicitamente quando o método é invocado. Para que estas criações sejam registradas, a função InsertNewModeledObject() é chamada para cada um destes objetos em um prólogo do método.

- RemoveModeledObject(): para a remoção de um objeto da lista de objetos modelados pela classe.

Deve ser chamado toda vez que um objeto é destruído ou sua classe é alterada (em uma atribuição).

No caso de destruição explícita, o comando delete deve ser precedido por um comando que retire o objeto e seus membros das listas de suas respectivas classes. Esta remoção em cascata é executada por um novo método inserido na classe do objeto. Assim,

obj1-> RemoveModeledObjectDestructed();   precede

delete obj1;

Além de objetos membros, objetos declarados como parâmetros ou locais a métodos são destruídos, também implicitamente, quando o método é finalizado. Para que estes e seus membros sejam antes retirados das listas de suas respectivas classes, a função RemoveModeledObjectDestructed() é chamada para cada objeto local em um epílogo do método.

Objetos podem mudar de classe quando uma atribuição de um objeto de outra classe (subclasse da classe original) lhe é atribuído. Neste momento, o objeto deve ser retirado da lista de objetos modelados pela classe original e ser inserido na lista da nova classe. Assim sendo, um comando de atribuição é sucedido pelas respectivas operações.

obj1 = obj2;   é sucedido por

obj1->RemoveModeledObject(obj1);

obj2->InsertModeledObject(obj1)

Como neste caso, a inserção e a remoção não possuem operações implícitas (já que objetos membros não mudaram de classe) as operações estáticas podem ser diretamente utilizadas.

- GetNthModeledObject : para a obtenção de um determinado objeto da lista de objetos modelados pela classe.

(3) Escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte a tempo de execução.

A metaclasse PronounClass redefine a função TranslateClass(), para que o novo membro e as novas funções sejam introduzidos na classe.

Além disso, as implementações de InsertModeledObject(), RemoveModeledObject(), GetNthModeledObject(), InsertNewModeledObject(), RemoveModeledObjectDestructed() são introduzidas redefinindo-se a função TranslateMemberFunction(). Nesta, são introduzidos também o prólogo e o epílogo de cada método onde locais e parâmetros são introduzidos e retirados das listas de suas respectivas classes.

Para referências, criações e destruições explícitas são feitas. Da mesma forma, as chamadas a InsertNewModeledObject() e RemoveModeledObjectDestructed() são, também, explícitas. Isso é feito na redefinição das funções TranslateAssign() e TranslateDelete().

A metaclass PronounClass redefine ainda a função TranslateMemberCall(), para obter a transcrição das chamadas de função sobre objetos. Quando o objeto receptor é o pronome ALL, este é substituído pelo envio da mensagem para todos os objetos modelados por todas as classes que a tratam.

3.2.6 ANY

(1) O que o programa de nível base deve ver ?

- Envio de mensagem ao pronome ANY apenas.

(2) O que deve ser transcrito e que código de suporte a tempo de execução é necessário ?

O envio de uma mensagem através do pronome ANY é transcrito para uma disponibilização de mensagem.

ANY -> msg(<args>); é transcrito para

g_BrokerAny->Send(this, <cod_msg>, 'msg');

g_BrokerAny é um objeto estático responsável por gerenciar disponibilizações e requisições de mensagens e é disponibilizado como suporte a tempo de execução. Seu método Send() registra que a mensagem <cod_msg> está disponível.

O controle passa então para os outros objetos que, quando terminarem um serviço qualquer e ficarem disponíveis, requisitam uma mensagem a g_BrokerAny.

g_BrokerAny->Receive(this);

Esta requisição é colocada como um epílogo em todos os métodos. Assim, todo objeto, quando termina um serviço, solicita uma mensagem disponibilizada. O método Receive, por sua vez, verifica se o objeto que está solicitando uma mensagem possui tratador para a mensagem disponível. Em caso positivo, passa o receptador ao objeto emissor para que este efetivamente a envie:

m_Sender->SendAny(Receiver, <cod_msg>);

Além disso, torna a mensagem indisponível para que nenhum outro objeto a receba.

Em toda classe que no corpo de seus métodos, enviar mensagem através o pronome ANY, deve ser criado o método SendAny(). Este verifica o código da mensagem e faz o respectivo envio.

Apesar desta implementação poder ser usada tanto em ambientes singletheads quanto multithreads, sua utilidade só pode ser verificada neste últimos, onde várias linhas de execução podem estar ativadas ao mesmo tempo e objetos concorrem efetivamente pela mensagem. Em ambientes singlethreads, a mensagem sempre será destinada ao próximo objeto que a trate acionado após o envio através do pronome ANY pois este será necessariamente o primeiro método a terminar após a disponibilização da mensagem.

(3) Escrever um programa de nível meta que faça a transcrição e escrever também o código de suporte a tempo de execução.

A metaclasse PronounClass redefine a função TranslateClass(), para que o novo método (SendAny) seja introduzido na classe. Sua implementação é introduzida redefinindo-se a função TranslateMemberFunction(). Nesta, são introduzidos também o epílogo de cada método onde a requisição de mensagem é feita.

PronounClass redefine também a função TranslateMemberCall(), para obter a transcrição das chamadas de função sobre objetos. Quando o objeto receptor é o pronome ANY, a chamada é substituída pela disponibilização da mensagem. Além disso, um novo caso é introduzido no corpo no método SendAny() para que o envio efetivo possa ser feito.

O objeto g_BrokerAny é provido como suporte a tempo de execução.

4. Trabalhos Relacionados

COOL, a linguagem proposta em [Coe92], usa pronomes como ALL, PARENT e ANY para comunicação entre objetos. COOL foi baseada no mesmo ambiente de desenvolvimento experimental em que foi baseado nosso trabalho. Neste, mensagens assíncronas enviadas para "owners" (nosso corrente CREATOR) foi fundamental para garantir independência de classe e reuso. Por causa disso, COOL pode ser considerado um estágio intermediário entre o corrente trabalho e a experiência original.

Para adaptar um objeto a diferentes necessidades de cliente de forma transparente, surgiu o padrão "role object" proposto em [BRS+97]. Associando papéis a objetos, um objeto pode ser usado em contextos diferentes, sendo conhecido, em cada um, por um diferente papel. Apesar do intuito de possibilitar a geração de códigos mais gerais ser o mesmo do nosso trabalho, a filosofia usada em [BRS+97] é oposta a usada por nós. Usando padrões para obter generalidade, a simplicidade e a concisão da linguagem é mantida mas a responsabilidade de manter a integridade das relações entre objetos (ou papéis) é transferido do sistema para o programador, gerando códigos difíceis de serem construídos e mantidos. Além disso, "padrões têm um identidade lógica e conceitual no nível de projeto mas esta é perdida quando passamos de projeto para implementação" [DR97].

Em [BLM98], Personalidades são apresentadas como artefatos lingüísticos a serem adicionados aos conceitos padrões de orientação a objetos para explicitamente encapsular papéis de decomposição funcional no nível da implementação. Um papel (personalidade) é associado a uma classe através de uma cláusula específica e, para ser uma personificação válida desta, deve obedecer regras bem definidas. O mesmo se dá com clientes desta personalidade. Similar ao

conceito de interfaces, como encontrado em Java, personalidades têm regras semânticas que são tão fortes quanto pronomes. Quando uma classe personifica uma dada personalidade, precisa declarar sua intenção e prover todos os métodos especificados em sua interface. De uma forma menos forte, pronomes não exigem que classes personifiquem coisa alguma, mas apenas tratem mensagens de seus interesses.

Manter automaticamente a consistência do grafo de dependência entre objetos foi também a motivação de Ducasse e Richner em [DR97] mas, diferentemente do nosso trabalho, a diretriz principal desta foi capturar relações de interações e não generalizar a comunicação entre servidores e clientes. Conectores são promovidos a objetos de primeira classe que representam relações de interação entre componentes, não apenas descrevendo, mas realmente controlando a comunicação entre estes componentes. Este controle é programado através de uma sintaxe especial e a consistência do grafo de dependência mantido automaticamente. Mensagens, no entanto, continuam a ser enviadas apenas nominalmente, sem redução de acoplamento.

O modelo de comunicação adotado em OASIS [SLR+99] é também similar a nossa linha de pesquisa. Neste, um sistema é representado por um conjunto de objetos autônomos interagindo em um modelo cliente-servidor. Tuplas no formato Cliente-Servidor-Serviço representam comunicações entre objetos. Clientes e servidores podem ser invocados por seus nomes ou por referências genéricas como SELF, SOMEONE e EVERYONE. Estes possuem semânticas bem parecidas com os pronomes ALL e ANY propostos aqui.

5. Conclusões e Futuros Trabalhos

O desenvolvimento de sistemas grandes é uma tarefa difícil que impõe um desgate grande aos projetistas. Por esta razão, na maior parte das vezes, generalidade não é uma funcionalidade priorizada. A primeira conseqüência disto é o pequeno reuso observado em desenvolvimento de software. Para diminuir a dificuldade na criação de códigos genéricos, pronomes são aqui propostos. Nesta proposta, nossa diretriz principal foi reduzir a distância entre o sistema projetado e seu código final, permitindo a eliminação de associações criadas apenas para servir de referências cruzadas.

No estágio atual do nosso trabalho, focalizamos nossos esforços em prover atalhos para a visibilidade obtida por uma identificação genérica de objetos. Temos uma experiência prática considerável com o uso desta característica. Em um estágio anterior de nosso trabalho[Car97], programamos uma ferramenta para construção visual de interfaces gráficas e uma biblioteca de classes modelando objetos visuais, para suporte a esta ferramenta. Em ambos os casos verificamos que mensagens assíncronas enviadas a “owners” (nosso CREATOR atual) foi fundamental para garantir independência de classe e, com isso, reuso.

Pronomes devem ser usados para a construção de classes servidoras puras que modelam objetos cuja ligação com o mundo exterior se faça apenas através do envio de mensagens. Isso reduzirá o acoplamento entre classes, evitando que parâmetros de saída sejam necessários para a transmissão de resultados para o ambiente chamador. Aumentará também a possibilidade de reuso pois estas classes podem assim ser inseridas em qualquer contexto.

O fluxo de mensagens é também reduzido com a introdução de representações genéricas para objetos importantes porque, com estas, qualquer objeto, em qualquer nível da árvore de agregação, pode enviar mensagens para eles diretamente, sem a necessidade que esta passe por vários objetos antes de chegar a seu destino.

A utilidade destas características é bastante sentida em sistemas mediados usando composição [Car98]. Este tipo de sistema é natural em ambientes orientados a objetos e, como resultado da composição adotada, relações de pai e filho são automaticamente estabelecidas.

Três outros pronomes estão sendo apreciados para aumentar o conjunto descrito anteriormente:

- CURRENT designa, em um loop, o último objeto retornado pelo iterador;
- CONTAINER designa a estrutura de dados (também objeto) que o objeto corrente é membro;
- SIBLINGS designa os filhos do mesmo pai que o objeto corrente.

Pronomes estão sendo também definidos para permitir a representação textual de padrões de projeto. Este é o caso de OBSERVERS, por exemplo. Junto com uma operação REGISTER predefinida, aplicável a sujeitos e invocadas por observadores passando-se como parâmetros, este pronome poderia ser usado por um sujeito para notificar seus observadores sobre mudanças em seus estados.

A definição de um pronome genérico cuja semântica poderia ser definida dinamicamente pelo programador chegou a ser pensada como extensão deste trabalho. No entanto, a facilidade encontrada na definição de pronomes em linguagens abertas fez com que esta idéia fosse abandonada. Hoje, trabalhamos na definição de um padrão para a implementação de pronomes nestas linguagens a fim de facilitar e automatizar o processo.

Em sistemas experimentais, foi observado que em um grande número de vezes um objeto trata uma mensagem enviada através de um pronome apenas para repassá-la a outro objeto (por ele conhecido). Uma forma de reduzir isto está sendo estudada através da introdução do conceito de redirecionamento. Com este, mensagens poderiam ser redirecionadas para atributos ou para outros pronomes de forma declarativa, sem que seja necessário escrever código para isso.

Referências

[And91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. ACM Computing Surveys, Vol. 23. No. 1. pp. 49-90. March 1991.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. A System of Patterns. Wiley, 1996.

[BLM98] Luis Blando, Karl Lieberherr, and Mira Mezini. Modeling Behavior with Personalities. Northeastern University Technical Report (NU-CCS-98-08). 10/98.

[BRS+97] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The Role Objet Pattern. In Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP '97). Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997. Paper 2.1, 10 pages.

[Car97] Sergio E. R. Carvalho. DDL: An Object-Oriented Design Description Language. Technical Repost PUC-Rio Inf. MCC29/97, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Department of Informatics, Rio de Janeiro, Brazil, 1997.

[Car98] Sergio E. R. Carvalho. Mediator Based Object-Oriented Frameworks, II Argentine Symposium on Object Orientation, Facultad de Ingenieria, University of Buenos Aires, August 31, September 1, 1998, pp. 11-22.
[CGL86] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. Thirteenth ACM Symposium on Principles of Programming Languages. ALM SIGPLAN and SIGACT pp. 236-242. Williamsburg, Virginia, Jan. 1986.
[Chi95] Shigeru Chiba. A Metaobject Prococol for C++, In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, page 285-299, October 1995.
[Chi96] Shigeru Chiba. OpenC++ Tutorial Programmer's Guide for Version 2," Xerox PARC Technical Report, SPL-96-024, 1996.
[Coe92] Otávio Pêcego Coelho. COOL – Uma Linguagem Sob Orientação a Objetos Comunicantes. Tese de Doutorado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brazil, Dezembro 1992.
[DR97] Stéphane Ducasse and Tamar Richner, Executable Connectors: Towards Reusable Design Elements, In Proceedings of ESEC/FSE'97 (European Software Engineering Conference), LNCS (Lectures Notes in Computer Science), N 1301, sep, Springer - Verlag, pp. 483-500, 1997.
[G+95] Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[Gen81] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. Software Practical Experience. 11, 435-466. 1981.
[Ger89] Colin Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25. Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.
[GI90] David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environiments. In proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments. SIGSOFT'90, Irvine, California, December 1990.
[GM85] D. H. Grit and J. R. McGraw. Programming divide and conquer on a MIMD machine. Software Practical Experience 15, 1, 41-53, Jan. 1985.
[HGN91] A. Nico Habermann, David Garlan, and David Notkin. Generation of integrated task-specific software environments. In Richard F. Rashid, ed. CMU Computer Science: A 25th Commemorative, Anthology Series, pp.69-98. ACM Press, 1991.
[Iona95] IONA Technologies Ltd: Orbix Programmer's Guide, compare also http://www.iona.ie/, Dublin, Ireland, 1995.
[JaJ92] Joseph JáJá. An Introduction to Parallel Algorithms. Addison-Wesley 1992.
[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object Oriented Programming, 1(3):26-49, August/September 1988.
[Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed systems. Commun. ACM 21, 7(Jul.), 558-565.
[MM97] T. Mowbray, R. Malveau, *CORBA Design Patterns*, Wiley, 1997.
[OMG92] Object Management Group: The Common Object Request Broker: Architecture and Specification, OMG Document Number 95-3-31, 1995.

[Rei90] Steve P. Reiss. Connecting tools using message passing in the Field Environment. IEEE Software, 7(4):57-66, July 1990.
[SG96] M. Shaw, D. Garlan, *Software Architecture*, Prentice-Hall, 1996.
[SHO90] S. Sutton, D. Heimbigner, and L. Osterweil. Language constructs for managing change in process-centered environments. In Proceedings of ACM SIGSOFT'90: Fourth Symposium on Software Development Environments, December 1990.
[SLR+99] P. Sanchez, P. Letelier, I. Ramos, O. Pastor, "OASIS 3.0: Un Enfoque formal para el Modelado Conceptual Orientado a Objeto", Memorias IDEAS 99, San José, Costa Rica, March 1999, pp.205-215.
[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. ACM Transaction on Software Engineering and Methodology, 1(3):229-268, July 1992.